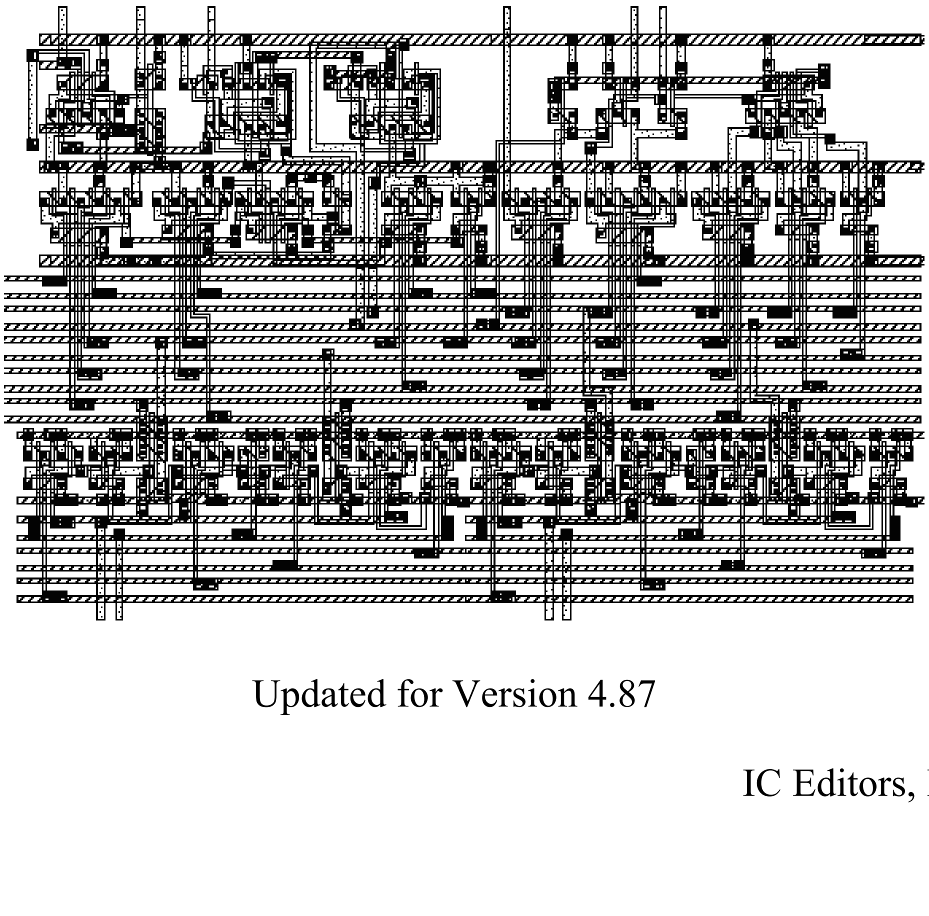
ICED™

Command File Programmer's

Reference Manual



Updated for Version 4.87

IC Editors, Inc.

Acknowledgments

The majority of this manual was written or revised by Ference Professional Services in Sonora, CA. We are also responsible for formatting the text and creating the screen captures that illustrate the examples. The original ICED-16 manual was written by Mark Stegall. Pieces of this manual can still be found in this ICED™ manual.

Michael Gentry of MGC, Inc. created the layout that is used on the cover and as a frontispiece. It is a section of a CMOS simulation of a 74181 4-bit ALU.

Table of Contents

Introduction

The tutorial on command files in the Classroom Tutorials Manual is a great way to become familiar with command file programming.

Command files are files of ICED™ commands that can be executed in the layout editor with a single command or keystroke. Their primary purpose is to make repetitive tasks easier, but once you are familiar with command files you can perform many complicated tasks that you cannot perform with editor commands alone.

This manual is intended to be a companion manual for the IC Layout Editor Reference Manual. This other manual covers all commands that can be used interactively in the editor (e.g. ADD and COPY.) Most commands and other subjects that are of interest only to writers of command files have been removed from the main reference manual and are covered here instead. However, many commands that may interest command file writers are covered only in the main reference manual. You really need to use both manuals to be an effective command file programmer.

One of the most useful features added to ICED™ for command files is macro manipulation. Macros implement string substitution allowing you to use stored strings as values in commands. Macros store strings that can be interpreted as text, numbers, or coordinates. Their function is similar to that of variables in other programming languages. An overview on macros begins on page 38. Details on macro definition are covered beginning on page 133. System macros are covered on page 249.

System macros are macros created and updated automatically by the layout editor.

Other features added to ICED™ primarily for use in command files include program loops (the WHILE command), conditional blocks of code (the IF command), mathematical functions, shells to the operating system, and many other features. Advanced tasks can be accomplished with data exported to an external program.

The creation and support of most of these features in ICED™ has been performed in response to the needs of users. The commands and other features added to support macros and command files will continue to develop over time, as users' needs change. You should be aware that the behavior of these features is not as fully tested as other features of the program and you should always test your command files carefully.

Table of Useful Examples in This Manual

We include the following table for two reasons. First, it demonstrates the types of tasks you can accomplish with command files. Second, if you are in a hurry and the task you need to accomplish is similar to one in the table, you may be able to just cut and paste from the manual to your command file for an instant solution.

Task category	**Example**	**Page**
Enhancing basic commands	Assign ADD TEXT command to function key	151
	Select exactly one component	96
	Delete all components on a specific scratch layer despite protection or blank status	84
	Prompt user for layer name and delete all components on the layer despite protection or blank status	98
	Unblank all cell layers then restore original status	274
	Blank some layers to make selecting a shape easier	275
	Prompt user for cell name and swap all components in that cell on one layer to new layer	108
	Prompt user for string using a default value	146
	Prompt user for layer using menu	147
	Prompt user for spacing distance using default	279
	Prompt for cell name and process cell if loaded	262
	Prompt user for integer and verify it is within range	305
	Change the width of a wire using ITEM command	180
	Select fill pattern from menu and replace with new pattern	277
	Test if vertex is within selection box	292
	Edit components by modifying their ADD commands	318
	Crete substring of existing string	223
	Build position list string for selected polygon	179

Creating components	Add an array of a component after digitizing a single copy	271
	Make multiple copies of an existing component	272
	Create simple spiral of wires	148
	Add label to wire automatically after digitizing it	118 272
	Add polygon in the shape of the number 0	317
	Add serial number polygons to array	312
	Create new cell using template cell	110
	Create a resistor after prompting the user for resistance and width	309
	Covert vertices if snap angle is non-zero	294
	Create displacement intervals on grid	236
	Transform a single wide wire into a routed bus	327
	Remove components added since benchmark	270
Command file control	Passing values into nested command file	97
	Loop prompting user for string and validating it	94
	Loop through all layers calling a command file to process each layer	99
	Loop processing all editable cells	264 297
	Loop processing all components in cell	269
	Loop processing all components in list	284
	Measure time to terminate infinite loop	298
	Post progress message periodically in long loop	162
	Copy a file using a DOS command	121
	Copy a file using a DOS command, display error message upon failure	122
	Search for file and open in text editor	265
	Create a directory if it does not yet exist	227
	Determine if a file has changed since last access	228
	Triggering an error handler block of code	166

Displaying or collecting information	Save and restore blank status of layers	258
	Select all contacts with a minimum area	185
	Report component count on a layer in a nested cell	102
	Report bounding box for specific cell name	221
	Report layers with CIF layer name defined	276
	Report layers with Stream layer name defined	280
	Get cell information for cell not yet loaded	222
	Report cell's library name	262
	Select cell and determine if it is in a protected library	263
	List all cell libraries	282 283
	List layers using a specific color	276
	Create sorted list of subcells	260
	List layers with default width less than minimum	281
	Change view window if necessary to display point	301
	Display SHOW information for a component in a nested cell	325

Figure 1: References to interesting examples of command files

Creating and Executing a Simple Command File

All ICED™ products are designed to be launched from a DOS console window. The best way to open a DOS window for our purpose is to use the ICED™ icon. This icon was created on your desktop during installation and displays the representation of a silicon wafer. Using the ICED™ icon sets the system's search path. This ensures that operating system looks in the correct directory for the program executable files. It also sets the current directory to the ICED™ directory, Q:\ICWIN[1]. Double click on the ICED icon now.

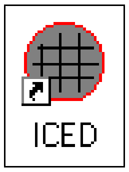


Figure 2: ICED desktop icon

We suggest that you use the TUTOR subdirectory to store the cell files and command files created while you explore this manual. You generally change to the directory where cell files are stored before launching the layout editor. To change the current directory, type the following command at the DOS prompt in the console window:

CD TUTOR <Enter>

ICED™ is usually launched by executing a project batch file that sets several environment variables and ICED™ command line options. The installation provides you with a sample batch file, Q:\ICWIN[1]\ICWIN.BAT. If you use this batch file, all you have to supply is the name of a cell to begin editing. Use this batch file to open the layout editor and create a new cell with the name "MYCELL". At the DOS prompt, type:

ICWIN MYCELL <Enter>

[1] Throughout this manual, Q: and \ICWIN are used to represent the drive and directory where you have installed the ICED™ software. If you have installed the software on your C drive in the directory \ICED, you should replace Q: with C: and \ICWIN with \ICED.

See page 15 to see why it is important to store command files used in real projects outside of cell libraries.

The simplest way to create a command file is to open the NOTEPAD text editor (supplied with all Windows installations) to create a new file in the current directory. You can do this directly from the ICED™ layout editor command line with the command shown below. (If you prefer, you can modify the command to open your favorite ASCII text editor.) Type the following command in the layout editor.

Example: **SPAWN –NOTEPAD MYCMD.CMD**

The SPAWN command executes an operating system command in a new window. See page 120.

If the file MYCMD.CMD contains text from some other use, delete it and save the file, (File → Save).

Type the following in the MYCMD.CMD text editor window (or cut and paste into the text editor window if you are reading this with Adobe Acrobat.)

Example: **ADD BOX AT (0 0), (10 10)**

Now save the file. (File → Save)

In the ICED™ window, type the following command to execute the MYCMD.CMD command file.

Example: **@MYCMD**

The command file is executed and the ADD BOX command in it adds the box component.

You will probably be executing the "SPAWN –NOTEPAD MYCMD.CMD" and "@MYCMD" commands often as you test examples in this manual. It will save time if you assign these commands to function keys. Then you can execute each one with a single keystroke. This will allow you to instantly edit the test command file from this cell and execute it at any time in the future. The process described on the next page will assign the commands above to function keys.

The ARROW command controls the behavior of the arrow keys.

Close the text editor window. (Use the 'X' button on the upper right corner.) Now use the <↑> key in the layout editor until the SPAWN command is displayed again on the command line. Edit the command line to look like the following (replace "NOTEPAD" with your favorite text editor if desired), then hit <Enter>.

Example: **KEY F11="SPAWN –NOTEPAD MYCMD.CMD"**

Now press the <F11> key to open the text editor to edit the test command file MYCMD.CMD.

To assign the @MYCMD command to the <F12> key, retrieve the command with the <↑> key and edit it to the following, then hit <Enter>.

Example: **KEY F12=@MYCMD**

Executing Command Files

Using the @file_name Command

The simplest way to execute a command file is to type the *@file_name* command. As mentioned in the introduction, to execute a command file with the name MYCMD.CMD in the layout editor, simply type at the command prompt:

Example: **@MYCMD**

The complete description of the *@file_name* command begins on page 159.

The file extension of .CMD will be added automatically when you do not supply a file extension in the *@file_name* command.

You can include other commands on the same line as the *@filename* command. These commands will be executed as though they are the first commands in the command file.

Example: **@MYCMD; LOG SCREEN OFF**

See page 130 to learn more about the implications of the LOG SCREEN OFF command.

When the MYCMD.CMD file is executed using this syntax, LOG SCREEN OFF is executed as if it was the first line of the command file. This LOG command will prevent commands in the command file from being echoed on the screen. This can make the command file run faster.

Adding commands on the same line as the *@file_name* command can be used to pass parameters into command files. See page 97 for examples.

Command File Search Path

Unless you specify the directory path to the command file in the *@file_name* command, a command file must be located in certain directories for the program to find it.

The order ICED™ will use when searching for a command file is:

- the current working directory
- directories in the ICED_CMD_PATH environment variable, and finally
- the AUXIL subdirectory(ies) of the ICED_HOME directory(ies)

ICED_CMD-_PATH and ICED_HOME are usually defined in the project batch file used to open the editor.

A command file in the working directory will hide any other command files with the same name in the other directories.

Where to Store Command Files

As you create and debug new command files, you may find it convenient to locate the command file in the same working directory as your test cell file. The editor will always search for a command file in this directory first. However, once the debug phase is complete, you should relocate the command file to a directory different from your cell files. This prevents problems when multiple copies of a command file exist, and you become unsure about which copy is really being executed. Having command files in your cell file directories can also cause problems when sharing files with other users.

The Q:\ICWIN[2]AUXIL directory contains many general-purpose command files created by the installation. If you create other command files that may be used in many different projects or technologies, they should be stored in this directory where they will be found automatically by the layout editor for all projects.

Other command files are technology or project dependent. The most important example is the startup command file that initializes a variety of technology-dependent parameters such as layer number-layer name correspondences, resolution grid definitions, etc. Some sample technology dependent command files are created in the Q:\ICWIN\TECH\SAMPLES directory by the ICED™ installation.

[2] Remember that Q:\ICWIN represents the drive letter and path where you have installed ICED™.

Any command file that uses technology dependent values such as layer numbers, contact size, wire spacing distances, or specific cell names should be located separately in a directory specific to that technology.

You should never keep technology dependent command files in the AUXIL subdirectory of the installation directory where they could be accidentally accessed in other projects that use other technologies. It would be too easy to execute such a command file in the wrong cell file, and you may not realize your mistake for a long time. If you realize this type of mistake after fabrication, you will be particularly displeased.

Once you create a technology directory, you can copy old technology dependent files to it and modify the important values accordingly. The technology directory (or directories if you prefer) can then be added to the command file search path of a particular project in the **ICED_CMD_PATH** environment variable definition in the project batch file. See the IC Layout Editor Reference Manual for more details.

Command File Names

There are few restrictions on the name of a command file.

- The file name can be from 1 to 32 characters long (not including the path or file extension.)
- Valid characters include all alphanumeric characters and ".", "-", "_", "#", and "$".
- Names are case-insensitive
- Blanks are valid only when you surround the file name with quotes.

We strongly discourage the use of blanks in command file names due to the syntax errors that will occur if you forget to surround the file name with quotes.

It is best to use a file extension of ".CMD" for all command files. This is not required, but only files with this extension will be automatically included in menu lists. This extension is added automatically to command file names any time you omit an extension. Using only one extension also makes it easier for you to search for command files using the operating system.

If you use a "_" prefix in the command file name (e.g. _GET_ANS.CMD), the command file will not be included in menu lists. These types of command files are meant to be called as helper files from other command files. They serve no purpose when executed directly from the command line or a menu.

Methods of Executing Command Files

There are several methods of executing command files other than typing the *@file_name* command at the command prompt.

From a Menu

The directories searched for command files are described on page 14.

The menu option 3:@%.cmd[3] allows you to select a command file from a menu list. The command files are listed by directory. To move on to the next directory, select the NextPATH option from the top of the menu list.

You can create custom menus that list your own command files in any order you like. See the MkMenu utility in the IC Layout Editor Reference Manual. You may want to use your custom menu as a shell. If so, refer to the SHELL command on page 207.

[3] In menu notation, "3:" indicates the third top-level menu, "@%.cmd" is the menu entry.

With a Keystroke

Key assignments are saved with a cell file.

If you have an especially useful command file that you execute often, you can assign the *@filename* command to a keystroke combination. Then you can execute the entire command file by pressing a single key or a combination of keys. (See page 151 for a more complete explanation).

Example: **KEY F12=@BUSROUTE.CMD**

Once the definition above is made, pressing the F12 key will execute the BUSROUTE.CMD command file. The .CMD extension could be omitted if desired. If it is omitted, it is added automatically by the program before searching for the file.

Automatic Execution when Editor Opens

The ICED.EXE command line is usually stored in a project batch file.

There are several options on the ICED.EXE command line to execute a command file as soon as the editor opens. The most common option is the STARTUP option that executes a command file when the indicated cell file does not yet exist. This command file is referred to a startup command file.

Example: **Q:\ICWIN\ICED … STARTUP=Q:\ICWIN[4]\TECH\SAMPLES\NEW …**

The startup command file is also executed when editing a freshly imported cell for the first time.

This is a fragment of the command line used to open the layout editor in the project batch file supplied with the installation (ICWIN.BAT.) The file name Q:\ICWIN\TECH\SAMPLES\NEW.CMD is stored as the startup command file name. It will be executed as soon as the editor opens if the cell name specified on the command line does not already exist. This sample startup command file contains layer definitions and other technology-specific commands used to initialize a new cell.

[4] Q:\ICWIN represents the drive letter and path of your first ICED™ home directory, usually C:\ICWIN.

STARTUP is one of five command line options for executing a command file. All of these options are described completely in the IC Layout Editor Reference Manual.

	ICED.EXE command line option	Use
Use only one of these	**EXIT**	Execute command file then close editor with an EXIT command. This will overwrite the cell file.
	LEAVE	Execute command file then close with a LEAVE command. This will overwrite the cell file only if changes have been made to the geometry of the cell.
	QUIT	Execute command file then close with a QUIT command. This will not overwrite the cell file.
	ALWAYS	Execute command file and leave editor open.
	STARTUP	Execute command file only for a new cell and leave editor open.

Figure 3: ICED.EXE command line options that automatically execute a command file

When the string "@DO-_NOTHING" is executed it has no effect.

You can combine the STARTUP option with one of the others. When STARTUP is used, the file name is saved in the START.CMD system macro. When one of the other options is used, the file name is stored in the ALWAYS.CMD system macro. Unless a corresponding option was defined, the value stored in both of these system macros is the string "DO_NOTHING".

Example:

```
Q:\ICWIN\ICED MYCELL EXIT=PROCESS_CELL.CMD …
        … STARTUP=Q:\ICWIN[5]\TECH\SAMPLES\NEW …
```

The fragment of the command line above will open the editor to edit the cell MYCELL.CEL. If this cell file does not already exist, the indicated startup command file will be executed. Then the PROCESS_CELL.CMD file will be executed and the editor will close automatically after saving the cell file.

[5] Q:\ICWIN represents the drive letter and path of your first ICED™ home directory, usually C:\ICWIN.

If an error is encountered in the command file specified with one of these options, the remainder of the command file is unprocessed, an error message is posted, and the editor is left open. If you prefer to have the editor close automatically with an error code, add the option BATCH=YES to the ICED.EXE command line. See an example of using the EXIT and BATCH options on page 25.

Automatic Execution when Editor Closes

These macros are not saved with the cell file. See page 116 to learn how to save and restore these macro values automatically.

There are several macros that have special significance when they are defined. You need to define these with macro definition statements before they can be used. For example, these definitions can be made in your startup command file. One of these macros is the EXIT.ROOT macro.

When you close the layout editor using an EXIT command (or a LEAVE command that results in saving the root cell), the editor first checks to see if the macro EXIT.ROOT exists. If it does, the editor will execute the command string stored in the macro before terminating the editor.

Example: **GLOBAL #EXIT.ROOT="@_EXIT_ROOT.CMD"**

See another example of EXIT.ROOT on page 116.

When this macro definition has been executed in the current edit session and the editor closes, before it saves the cell file(s) it will execute the command file _EXIT_ROOT.CMD.

IF you add the NOW keyword to the EXIT or LEAVE command that terminates the editor, EXIT.ROOT is ignored.

Automatic Execution when Subcell is Opened or Closed

Two other user-defined macros that can force command file execution when they are defined are:

ENTER.SUBCELL the command string stored in this macro is executed when a cell is opened with an edit command (EDIT, P_EDIT or T_EDIT.)

EXIT.SUBCELL the command string stored in this macro is executed when a cell opened with an edit command is closed with an EXIT or LEAVE command that indicates that the cell file will be saved when the editor closes.

For example, suppose the following macro definition has been executed in the current editor session. This definition might have been made in a startup or always command file.

Example: **GLOBAL #EXIT.SUBCELL="@_LOG_CELL.CMD"**

See an example on page 25 that appends a line to a file.

Once the above definition is made, whenever you exit a subcell using an EXIT command (or a LEAVE command that results in saving the subcell), the command file _LOG_CELL.CMD will be executed.

If the NOW parameter is added to the EDIT, P_EDIT or T_EDIT command, any string stored in ENTER.SUBCELL is ignored. Similarly, when NOW is added to an EXIT or LEAVE command, the EXIT.SUBCELL definition is ignored.

Automatic Execution when Error is Encountered

The last user macro that can trigger command file execution is the ERROR.CMD macro. When this macro is defined, the string stored in it is executed when a syntax error is encountered, or when the ERROR command is executed. See the ERROR command on page 165 for more details.

Example: **GLOBAL #ERROR.CMD="@_ERROR_HANDLER.CMD"**

Executing a Command File on Many Cells

See the next section on page 23 to learn how to execute a command file on every cell in a library.

You can execute a command file in many cells using a loop that opens each cell, executes the command file, and then exits the cell. There are two command files supplied with the installation for this purpose, or you can write your own example based on either of these files or the examples in this manual listed below.

These examples are all designed to be executed when you are editing the main cell of a design. You should exit from all nested edits and have only the main cell open when you execute any of these methods.

_LOOP.CMD[6] Executes a command string on all subcells of the current cell. (See page 297)

See page 106 to learn more about cell libraries and protection levels.

_LOOP2.CMD[6] Executes a command string on all subcells of the current cell that are stored in direct-edit libraries. Alternately, you can override the protection level and edit or browse cells at higher levels of protection (e.g. copy-edit or read-only.) This command file contains extra processing to modify cells at the lowest level of the hierarchy first. It continues up through the hierarchy so that no cell is modified before all of its subcells are modified. The command string is then executed on the main cell as well.

_LOOP.CMD and _LOOP2.CMD are explained more fully in the tutorial on Looping Through All Subcells in the Classroom Tutorials Manual.

Simple example of using MARK_SUBCELLS to edit every subcell in a loop. See page 109. MARK_SUBCELLS makes a loop that edits every subcell very efficient (especially when only you restrict the subcell marking to cells that contain certain layers.) The two command files listed above also use MARK_SUBCELLS.

Simple example of loop to process all loaded cells. This is the only example that will modify not only all editable subcells of the current cell, but all other editable cells that have been loaded in the current editor session. See page 264.

[6] These command files are stored in Q:\ICWIN\AUXIL which is always on the command file search path.

Example: **@_LOOP2.CMD; #LOOP.LEVEL = 1; #LOOP.OP = @MYCMD.CMD;**

Macros used as arguments for command files must be defined in the same statement. See page 97.

The example above is a single statement to execute _LOOP2.CMD with overrides for the LOOP.LEVEL and LOOP.OP macros. It will open every subcell of the current cell and execute MYCMD.CMD in each one. Since the LOOP.LEVEL is overridden, even read-only and copy-edit subcells will be opened. If MYCMD.CMD contains commands that modify components, then the command file would fail as it tries to exit a modified read-only or copy-edit cell.

When you use _LOOP2.CMD and do not override the LOOP.LEVEL macro, only cells stored in directly-editable libraries will be opened and have the command string in LOOP.OP executed in them.

When you do not override LOOP.OP, the default behavior is to open an interactive shell session for each subcell using the SHELL command. You must close each shell by typing an EXIT command to allow the command file to continue to the next subcell.

Batch Execution

You can create a batch file with multiple ICED.EXE command lines to open the editor and execute a command file in each of multiple cells. You use one of the command line options described on page 18 on each ICED.EXE command line to automatically execute a command file as the editor opens. If you use the appropriate option, the editor will close automatically after the command file is completed.

The AllCells utility is completely described in the IC Layout Editor Reference Manual.

We provide a utility (ALLCELLS.EXE) to create a batch file that includes an ICED.EXE command line for every cell in a given library. Even if the batch file created by the AllCells utility does not exactly meet your needs, it is an excellent example to modify for your own purposes.

Some Windows operating systems do not process batch files correctly. The processes started by a batch file can overlap and/or the error level is not reported back to the calling batch file correctly. For this reason, you must avoid batch file processing the Windows 95, 98, or Me operating systems. AllCells will recognize these operating systems and will refuse to create the batch file.

Executing the AllCells utility on one of the supported operating systems will create a batch file to execute a command file in each cell file in the current directory. You specify the name of the command file to execute and the exit mode to use during the execution of AllCells.

The choice of the exit mode determines which of the options described on page 18 will be used in the batch file.

If you specify that ICED™ should run minimized, then the RUNMIN utility will be used in the batch file to prevent the creation of a visible window for each editor session.

If the cells you want to open contain subcells that are located in different cell libraries, you **must** make sure that the ICED_PATH environment variable is defined to tell ICED™ where to look for subcell cell files. To define ICED_PATH in the batch file, execute your project batch file with no arguments before executing the AllCells utility. This will define ICED_PATH. Then respond with a <y> to the prompt about including environment variable definitions when you execute AllCells. This will include the ICED_PATH definition in the new batch file.

The utility creates a batch file with the name XCELLS.BAT. You execute this batch file to process the cells. See a sample XCELLS.BAT file in Figure 4.

Since the EXIT option is used to specify the command file in each of the ICED.EXE command lines in the example in Figure 4, the editor will close automatically after the X.CMD file is executed in each cell. Since the BATCH=YES option is used, the editor will close even if an error is encountered in the command file. In this case, a return code of 20 will be posted to the operating system, and the "IF ERRORLEVEL==1…" line will cause the cell name to be added to the XCELLS.ERR error log file.

The DOS ECHO command writes a line to the console. When you redirect it to a file, it appends the line to the file.

```
rem ***Sample XCELLS.BAT***
Q:
CD Q:\ICWIN\TUTOR
DEL *.JOU
DEL *.JO1
DEL XCELLS.ERR

SET ICED_DIST=Q:\ICWIN
SET ICED_HOME=Q:\ICWIN;
SET ICED_PATH=Q:\ICWIN\SAMPLES
SET ICED_USER=Q:\ICWIN

rem  NOTE: errorlevel==1 really means errorlevel is at least 1.

RUNMIN ICED CELLA EXIT="X.CMD" PAUSE=0 WINDOWS=500 BATCH=YES
IF ERRORLEVEL==1 ECHO CELLA >> XCELLS.ERR

RUNMIN ICED CELLB EXIT="X.CMD" PAUSE=0 WINDOWS=500 BATCH=YES
IF ERRORLEVEL==1 ECHO CELLB >> XCELLS.ERR

IF NOT EXIST XCELLS.ERR GOTO DONE
@ECHO OFF
ECHO ***************************************
ECHO ICED failed for at least one cell file.
ECHO A list of failed cells is in XCELLS.ERR
ECHO ***************************************
:DONE
```

Figure 4: Sample batch file created by AllCells utility

The BATCH Command Line Option

Note the BATCH=YES command line option used in the sample batch file above. This option has the following effects on the way the editor operates:

A batch file will probably not be able to test the error code when using the Windows 95, 98, or Me operating systems.

1) If an error is encountered as the command file is executed, the editor will terminate with an error code of 20 rather than being left open with the error message displayed below the command line.
2) If you need to pass arguments of the form *keyword*=*value* when executing the batch file, using BATCH=YES on the command line will allow this if you use *keyword*#*value* instead. This is best explained by example. See the example below.

If you pass arguments into a batch file, the '=' is considered whitespace by the DOS command interpreter and converted into a blank space. This prevents you from passing an argument into a batch file that contains an '='. For example, if you want to pass the string "PAUSE=10" as an argument into a batch file, it will be stored as "PAUSE 10". This can cause problems when the argument is used on the ICED.EXE command line. The '=' is required by many command line options including PAUSE. The BATCH option allows you to use '#' in place of '=' for succeeding command line options. The '#' is not stripped by the DOS command interpreter.

Example: **Q:\ICWIN\ICED MYCELL1 EXIT=MYCMD BATCH=YES %1 %2 %3**

Suppose the line above is stored in a batch file with the name MYBAT.BAT. Now suppose that you execute this batch file by typing the following at the console window prompt.

MYBAT PAUSE#10

As it executes the batch file, the command interpreter will replace the %1 with the first argument "PAUSE#10". The following line will then be executed.

Q:\ICWIN\ICED MYCELL1 EXIT=MYCMD BATCH=YES PAUSE#10

This will add a delay to the opening and closing of the editor allowing you to see any messages generated by the editor.

Even if you do not add an argument when executing MYBAT, the BATCH=YES option will prevent the editor from being left open in interactive mode if an error is encountered in MYCMD.CMD. The editor will close with an error code. If MYBAT.BAT contains the following lines after the ICED.EXE command line, then the batch file will recognize that an error has occurred and will take the appropriate action.

Example:

```
IF ERRORLEVEL==1 GOTO ERROR
GOTO DONE
:ERROR
@ECHO OFF
ECHO *****************
ECHO **Error encountered
ECHO *****************
:DONE
```

The test "errorlevel==1" is true anytime the error code is greater than 0.

Command File Syntax

Review Of ICED™ Command Syntax

ICED™'s command parser reads and interprets commands. The same routines are used to interpret commands typed at the command prompt in the layout editor, selected from the menus, or executed from command files. So, the syntax used in ICED™ command files is identical to that used in the editor. With few exceptions, all of the extra commands and other features described in this manual can be executed at the prompt line in the editor, and all of the commands described in the IC Layout Editor Reference Manual can be used in a command file.

Since most experienced layout designers normally enter commands with the keyboard, ICED™'s command syntax was optimized for keyboard input. To accomplish this, the parser minimizes the number of keystrokes required to enter a command and makes the syntax as forgiving as possible. The parser accepts keyword abbreviations, places minimal restrictions on keyword order, and, in most contexts, treats common delimiters as blanks.

In order to understand some of the syntax we will cover later, it is best to start with a review of the command syntax described in the layout editor reference. You may never have had occasion to use some of these syntax constructs, and it is best to have this information fresh in your memory when we go on to describe the additional syntax constructs used command files.

We will keep this review of basic ICED™ command syntax brief.

Commands, Continuation Lines, and Statements

When you enter a command from the keyboard, a command line is everything you type until you hit the <Enter> key.

When you type commands in a command file, a command line is an initial line plus 0 or more continuation lines. A command is continued on the next line if the last non-blank character in the line is an ampersand '&'.

Example:

```
ADD WIRE  LAYER=WELL TYPE=2  WIDTH=3.0  AT       &
      (-71.5, 38.0) (-14.5, 38.0) (-14.5,0.0) (48.0,0.0)        &
      (48.0, 43.0) (102.5, 43.0) (102.5,-1.0) (119.5,-1.0)
```

This example shows a command line that consists of an initial line plus two continuation lines. As a first step in reading commands, the command parser joins an initial line with all its continuation lines to form one command line. This command line can be up to 8000 characters long.

This manual often uses the term "statement" to refer to the entire command line once the continuation lines have been joined together.

One statement can contain several commands separated by semicolons (';'). The following example shows a statement with three commands.

Example:

```
UNSELECT ALL; SELECT CELL * ALL; UNGROUP
```

Comments

If the command parser encounters an exclamation point '!' outside of a quoted string or a Boolean expression (described later), the rest of the statement is treated as a comment (i.e. ignored).

Example:

```
ADD TEXT "!CLOCK" AT (0, 0)   ! This part is a comment.
```

The command interpreter joins continuation lines before it does anything else, so the example below is interpreted as a single comment line.

Example:

```
! ******Next line will be ignored*********; &
ADD BOX AT (0, 0) (10, 10)
```

The ADD BOX command in the example above will not be executed. Semicolons ';'s and other special characters are ignored in this type of comment.

ICED™ supports a second type of comment that uses a '$' as the first character in the statement.

Example: **$ This comment will echo in journal file**

$Comments that begin with "$$" are processed differently. See page 161 for more details on $comments.

$Comments and !comments differ in that $comments are displayed on the screen and entered in the journal file. The parser also does macro substitution and evaluates expressions in $comments. (These subjects will be described later.) !comments are used primarily to allow the writer of a command file to document features of a command file and to make it more understandable to the reader. $comments are used to display messages to the user of the command file and to enter comments in the journal file.

Line Labels

You can label statements with a label name string followed by a colon (':'). You can then refer to these labels in BACK_TO or SKIP_TO commands as you would use GOTO commands in other programming languages.

Example:

```
MYLABEL:
!missing statements
BACK_TO MYLABEL
```

You can include commands on the same line as a label.

Example:

```
MYLABEL:    UNSEL ALL
```

The statement label string cannot use macro references. The restrictions for labeled statements are shown below.

- The label must be a string from 1 to 32 characters long.
- Valid characters include letters, digits, and the special characters: '#', '_', and '$'. Blanks are not allowed in the label. The first character may not be a digit or one of the special characters listed above.
- Labels are case-independent.
- You must follow the label with a ':' when defining the labeled statement. Further references to the labeled statement omit the ':'.
- The label must be the first string on a line. However, it does not need to be in the first column.
- A command can follow the label on the same line, or the label can be on a line by itself.
- You should not label statements inside of a WHILE, IF, or ELSEIF block. This will not generate an error, but using SKIP_TO or BACK_TO to jump into a block can have unfortunate consequences.

Delimiters

The parser usually treats the comma ',', open and close parenthesis '(' and ')', and equals sign '=' as blanks. They can be added to improve the readability of a command, but with few exceptions they can be omitted to save typing.

The next three statements will all execute the same command.

Example:

```
ADD WIRE LAYER=POLY WIDTH=5 AT  (0, 0) (10, -10) (10, -20)

ADD WIRE LAYER POLY  WIDTH 5 AT 0 0  10 -10  10 -20

ADD WIRE=LAYER POLY WIDTH (5 AT  0) (0, 10) (-10, 10) -20
```

The use of '^' to delimit macro names is covered on page 44.

Tab characters are also converted to a single blank space. You can use tabs freely to make your command file easier to read.

Semicolons are optional at the end of commands. However, if you want to place several commands on a single statement, you must delimit the commands with semicolons.

Example: **UNSELECT ALL; SELECT CELL * ALL; UNGROUP**

Underscores

Underscores can be inserted inside of or at the end of keywords, layer names, and color names to improve readability. Thus, NOPEN, NO_PEN and NO_PEN_ all mean the same thing. However, _NOPEN is invalid.

Case Insensitivity

With few exceptions, ICED™ does not distinguish between upper and lower case letters. Thus, "Add Box", "ADD BOX", and "add box" all mean the same thing. You can type using whatever case you feel comfortable with.

Most of the examples in this manual are typed in upper case. This does not imply that you must (or even should) type your command files in upper case. The manual examples are in upper case simply to make them stand out from the other text.

Abbreviating Keywords, Layer Names, and Color Names

The parser allows you to abbreviate keywords, layer names, and color names, so long as you type enough characters to make their interpretation unambiguous.

For most keywords, two characters will do. The following three commands are equivalent:

Example: **LAYER MET1 GREEN**

LA MET1 GREEN

LA ME GRE

The last command would fail with an error message if more than one layer name begins with the letters "ME" since that would make the abbreviation ambiguous.

Quotes and Strings

The simple definition of a string is a series of characters. In ICED™, we usually restrict the term "string" to mean a series of characters to be stored or displayed as a unit. Strings can contain references to macros and the parser will replace the references as it parses the string. Examples of strings include comments, prompt messages, and labels to be added by the ADD TEXT command.

Strings are often enclosed in quotation marks. This can make their interpretation unambiguous, and also improves readability. Any string that contains characters that might be misinterpreted by the parser (e.g. command keywords, quotes, etc.) should be surrounded by quotes. However, quotes are not usually required around a string.

There are four legal quotation marks: ', `, ~, and ". The opening and closing quotation marks at either end of the string must be the same. Thus, "abc", 'abc', ~abc~, and `abc` are all the same valid quoted string. Similarly, "a'b", ~a'b~, and `a'b` are all equivalent, but 'a'b' is illegal.

% Prompts and Position Prompts

% prompts are a simple way to prompt the user for a value when a command is executed. ICED™ commands include many constructs of the form:

KEYWORD=*value*

For example, this is the syntax of the ADD BOX command:

ADD BOX LAYER=*layer_id* AT *pos1 pos*2

"LAYER=*layer_id*" is an example of a "KEYWORD=*value*" construct. If you use a '%' for the *layer_id*, ICED™ will prompt the user for the value.

Example: **ADD BOX LAYER=% AT 50,50 60,60**

When this command is executed, the program will prompt the user with the phrase "Enter layer name or number:" and then wait until something is typed. After <Enter> is pressed, the characters typed by the user will be substituted for the '%' in the command.

You can use more than one % prompt in a command.

Example: **LAYER 10 NAME=% COLOR=%**

When the statement above is executed, the user will first be prompted with:
"Enter layer name:"

After the user has typed the layer name and pressed <Enter>, the user will be prompted with:
"Enter color name or number:"

After the user has responded to this second prompt, the LAYER command will be executed using whatever values the user has typed in with the keyboard.

The '%' can be omitted when it is the last character in the command. Therefore, the example above could also be written as shown below and it will still prompt the user for both the layer and the color as shown above.

Example: **LAYER 10 NAME=% COLOR**

You cannot use a '%' prompt to prompt the user for position information normally supplied with the mouse.

Example: **ADD BOX LAYER 5 AT %** ! WRONG

The command above uses illegal syntax and will fail.

When no positions are included in a command that requires them, the user will automatically be prompted to supply them with the mouse.

Example: **ADD BOX LAYER=5 AT** ! OK

No '%' is required in this command to prompt the user to digitize the positions for the new box.

Introduction to Macros

The primary addition to ICED™ command syntax for command files is macro substitution. The following pages cover the basics you need to know to use macros. The full syntax required for macro definition statements is completely described later in the manual on page 133.

Many macros are pre-defined with information about the layout editor environment and design geometry. These system macros are described beginning on page 249.

In ICED™, a macro consists of a name and a value. The value of a macro is also called the "contents" of the macro and is said to be "stored" in the macro. When you use the correct syntax to refer to a macro in a command, the macro reference will be replaced with the contents of the macro.

String Substitution

In ICED™, macros store strings that can be interpreted later in many different ways. Referring to a macro name with a '%' in front of it results in string substitution. For example, the following lines define a macro with the name COORD, then use this macro in an ADD command.

Example:

```
LOCAL  #COORD = "(20.5, 65.0)"
ADD TEXT "BUS0" AT %COORD
```

The command parser will substitute the macro reference, "%COORD", with the contents of the COORD macro, the string "(20.5, 65.0)". Once this substitution is performed, the following command will be executed:

```
ADD TEXT "BUS0" AT (20.5, 65.0)
```

Use of the '#' and '%' Characters

The '#' in the macro definition on the previous page indicates to the program that a macro name follows. When you want to assign a value to a macro, this is how you designate which macro will receive the value.

The '%' in the second statement indicates a macro reference that should be replaced with the value of the macro.

Any time you type a macro name, you should indicate that it is a macro name and not just a string by prefixing it with either '#' or '%'. While the '#' is optional in some cases, the '%' is always required to force macro substitution.

Example: **ADD TEXT "BUS0" AT COORD** !Oops, forgot the %

If this statement was used in your command file, rather than the one on the previous page, the program would not realize that you were referring to a macro name. No string substitution would be performed. The program would halt the command file and display the message:

```
ADD TEXT "BUS0" AT <<>>COORD
error: Expected coordinate or end of command.
```

Since it is very easy to forget this aspect of using macros, we recommend that you avoid syntax mistakes by following the rule below.

Every time you type a macro name,
use either the '#' or '%' prefix.

Macro Values are Stored as Strings

When ICED™ assigns a value to a macro, it does not make assumptions about how the value will be used. It is simply a stored string. Since no restrictions are made on how the value can be used, there is no automatic syntax or range checking on the value as is common with variables in other programming languages.

If the COORD macro was defined with a value of "42", and the ADD TEXT command from the previous example was executed, the following command would be generated:

```
ADD TEXT "BUS0" AT 42
```

The command above would fail since 42 is not a valid coordinate pair. You could set COORD to "abc", "30 40 50 60", or "" and the error would only be found when the ADD TEXT command was executed.

Since macros result in simple string substitution, there are few limits to their use. You can place an entire command, or even several commands separated by semicolons, into a macro and then execute the statement later with a macro reference.

Example:

```
LOCAL   #CMD = ADD TEXT "BUS0" AT (20.5, 65.0)
.
.        !missing statements
.
%CMD
```

This reference to the CMD macro would execute the statement stored in the string. If the statement contained a syntax error, it would be found only when the statement was executed.

Macro Substitution in Strings

Macro substitution takes place even in a quoted string. The '%' used to indicate a macro reference is one of only two special characters that cause special interpretation in a quoted string. (This is why you must sometimes use "%-" when you need to have the percent sign in a quoted string. See below.)

Example:

```
LOCAL  #LABEL = BUS
ADD TEXT = "%LABEL 0" AT 50,50
```

The first statement above stores the string "BUS" in a macro with the name LABEL. The macro substitution indicated by the %LABEL macro reference will take place even though it occurs in a quoted string. The statement after macro substitution is:

```
ADD TEXT = "BUS 0" AT 50,50
```

The Percent Sign '%' In Strings

Since the syntax %*macro_name* has special meaning even in a quoted string, if you need to use the character '%' in a string where it does not indicate a macro reference, you may need to type "%-" instead of '%'. The dash is removed by the parser.

Example: **ADD TEXT "50%-CLOCK" AT (0, 0)**

This command actually adds the text "50%CLOCK".

The dash is required to prevent the parser from looking for a macro name only if the '%' is immediately followed by a character that may represent the beginning of a macro name (i.e. a letter, an underscore '_', a dollar sign '$', or a period '.'). If a number or a blank space follows the '%', the dash is permitted but not required.

Overview of Macro Definition and Assignment

A macro must be defined with a macro definition statement before it can be used in another statement.

Macro Scope

Most macro definitions (including the examples above) begin with the LOCAL keyword that sets the scope of the macro. With few exceptions, either the LOCAL keyword or the GLOBAL keyword must be used in each macro definition statement.

LOCAL macros:

- can only be used in the command file in which they are defined.
- hide the existence of global macros with the same names defined outside of the current command file.
- will be deleted automatically at the end of the command file in which they are defined.

You can test if a macro name has already been defined with the MACRO-_EXISTS function.

GLOBAL macros:

- can be used outside of the command file that defines them.
- persist until the end of the current layout editor session unless they are explicitly deleted with the REMOVE command.
- can have their values displayed with the SHOW command after the command file is completed.

For example, let us assume that your command file defines the following macros:

```
GLOBAL   #COORD1 = ""
GLOBAL   #COORD2 = ""
```

After the command file was finished, you could display the final values of the both macros on the screen with the command:

Example: **SHOW USER=COORD* FILE=***

Macro Names

You cannot use the name of system macro to define a macro of your own. See the list on page 255.

There are few restrictions on the names of macros. Since each macro name should be prefixed by a '#' or a '%', the parser will not confuse macro names with command keywords or function names.

Macro names can contain up to 32 characters. The characters in a macro name may include letters, digits, and the special characters: '.', '_', and '$'. You may not use a digit as the first character. No blanks are allowed. Macro names are case-independent.

Although the '#' is optional before the macro name in a macro definition, we recommend that you use it as a prefix for the macro name for readability and consistency.

Example: **LOCAL #COORD = "(20.5, 65.0)"**

This is the macro definition we used earlier. The name of the macro is COORD.

Macro Substitution in Macro Names.

You can build the name of a macro by referring to other macros. For example, there is a system macro with the name LAYER.NAME.*layer_spec*, where *layer_spec* represents some layer number. You could use this macro in the form "LAYER.NAME.6". However, it is more common to specify the layer number with another macro.

Example: **#LAYER=%LAYER.NAME.%LAYER_NUMBER**

There are several reserved macro names that have special meaning when they are defined. See the list on page 153.

First the parser will replace the rightmost macro reference, resulting in a statement similar to:

#LAYER=%LAYER.NAME.6

Then the next macro reference is processed, which transforms the statement into:

#LAYER=M1

Finally, this statement will be executed which will store the string "M1" in the macro LAYER.

Simple string substitution allows you to implement an array of coordinates by using macro names with subscript notation. You can create arrays with any number of dimensions, simply by using string substitution to build the macro names. See several examples beginning on page 148.

Delimiting a Macro Name

The ICED™ parser normally recognizes the end of a macro name when it comes to a blank or some other character that cannot be part of a macro name.

When you cannot use a blank after the macro name, the '^' special character is used to delimit (i.e. mark then end of) a macro name. This situation arises when you use macro substitution to build strings like subscripted array macro names, fully qualified file names, or prompt messages.

Example: **#NET_NAME = %BASE_NAME^1**

The example above will add the character "1" to the end of whatever string is stored in the BASE_NAME macro and store the result in the NET_NAME macro. No blank space will be added between the "1" and the rest of the string. The '^' is discarded by the parser. If BASE_NAME contains the string "BUS_", then the NET_NAME macro will be set to "BUS_1".

Example: **#FILE_NAME = %TMP^%CELL^.TXT**

See another example of building file names on page 256.

This example builds a file name by concatenating three strings. TMP is a system macro that contains the path to the directory used by ICED™ to store temporary files. CELL is a system macro used to store the name of the current cell. If TMP = "Q:\ICWIN\TMP\" and CELL = "MYCELL", then the macro FILE_NAME will be set to "Q:\ICWIN\TMP\MYCELL.TXT".

Assume that we have stored the string "LUCY" in a macro with the name NAME1 and stored the string "RICKY" in macro NAME2. The results of various concatenations are shown below.

Syntax used	**Result string**
%NAME1 %NAME2	"LUCY RICKY"
%NAME1%NAME2	Error: macro "%NAME1RICKY" is not declared
%NAME1^%NAME2	"LUCYRICKY"
%NAME1^.%DIM1^.%DIM2	"LUCY.1.2" (if DIM1 = 1 and DIM2 = 2)
FRED ETHEL LUCY^RICKY	"FRED ETHEL LUCY^RICKY"

Note that the last example indicates that the parser will treat a '^' character as an ordinary text character when it does not follow a macro reference.

Methods of Assigning Macro Values

Each macro must be defined with an initial value. If you do not provide an initial value, the user of the command file will be prompted to supply the value when the macro definition statement is executed. There are several ways to prompt the user for the value. Some methods require the user to supply the value by typing at the keyboard, others require the use of the mouse to digitize one or more positions with the cursor.

Example: **LOCAL #NUM = $PROMPT "Type in number of copies"**

The macro definition above will prompt the user with the message shown and wait until something is typed. The characters typed before <Enter> is pressed will be stored as the value of the macro.

Remember that no verification is performed on macro values. The user could type "ABC" in response to the prompt above and that would be stored as the value of the NUM macro. We discuss methods of verifying these type of user responses on page 87.

Example: **LOCAL #COORD $PROMPT "Digitize initial position" POS**

This macro definition will display the indicated prompt message and wait for the left mouse button to be clicked. The coordinate pair of the current cursor position is then stored as the value of the macro.

The entire list of prompt methods available in a macro definition is covered when we discuss the exact syntax of the macro definition statement on page 143.

You can change the value stored in a macro with an assignment statement after the macro is defined.

Example: **#NUM = {%NUM +1}**

This statement uses an expression with the addition operator to increment the value in the previously declared NUM macro. The curly braces "{}" are required. They prevent the program from storing the "+ 1" as ordinary string characters rather than forcing the evaluation of the mathematical expression and storing the result. We will discuss this in more detail later on.

You can also use the user prompt keywords of a macro definition statement in a macro assignment statement.

Example: **#COORD = $PROMPT "Digitize next position" POS**

This macro assignment will prompt the user with the message shown, wait for the user to digitize a position with the mouse, then replace the old value of the COORD macro with the new position digitized by the user.

Delayed Substitution

This is a rarely used feature of ICED™ command file syntax that you may want to skip over. It allows you to delay macro substitution until a string is actually executed. (This feature can be very handy when defining keyboard macros. A keyboard macro is executed when the user presses a certain key combination. We cover this subject in detail on page 151.)

Use a pair of percent characters, "%%" to force delayed evaluation of a macro reference. When you assign a value to a macroA that contains a reference to macroB in the form "%%macroB", the value of macroB will not be substituted for the "%%macroB" at that time. Instead, the string will be stored with the macro reference intact. However, when the value of macroA is evaluated in a command, the **current** value of macroB will be used, rather than the value of macroB when macroA was created.

Example: **GLOBAL #KEY.F5 = "ADD TEXT=% AT %%LAST.POS"**

The macro definition above will create a keyboard macro with the name KEY.F5. Once this macro is defined, pressing the <F5> key will execute the indicated ADD TEXT command. The user will be prompted to type in the text for the command. Since the system macro LAST.POS is referenced with a "%%", the value of this macro at the time the <F5> key is pressed will be used rather than the value at the time the keyboard macro was created.

Overview of System Macros

In addition to the macros you define yourself, there are a number of global macros called system macros that are automatically defined and updated by ICED™. You do not need macro definition statements to define system macros and you cannot change their values directly with macro assignment statements. However, most system macro values can be altered with appropriate editor commands.

These system macros contain useful information about editor settings (e.g. resolution step size, layer properties, etc.), the current cell (e.g. the cell name, the number of currently selected components, etc.), and recent user actions (e.g. the last position digitized). See the complete list on page 252.

Example: **#COORD = %LAST.POS**

The LAST.POS system macro (described on page 272) stores the last position digitized with the cursor. The statement above copies this coordinate pair into the macro COORD.

Expression Evaluation

In ICED™, an expression is a string that should be evaluated and the result of the expression substituted for the expression string in the statement.

Expressions can include:
mathematical expressions,
Boolean expressions,
and
function calls.

We will discuss each of these types of expressions after we discuss something common to all expressions in ICED™.

Expressions Should Be Surrounded By {}

You can use '{' or '}' in a quoted string without causing an attempt at evaluation.

When you need the parser to evaluate an expression and replace the expression with the result before the statement is executed, the expression must usually be surrounded by curly braces "{}". For example, if the curly braces are omitted in a macro assignment, the expression will simply be interpreted as a string of characters and stored without evaluation.

Example: **LOCAL #VAL = 50 + 1.6** !No evaluation will be done

Since no curly braces are included in this macro definition, the program will not attempt to evaluate this mathematical expression before storing the string. The value of the VAL macro will be "50 + 1.6".

Example: **LOCAL #VAL = {50 + 1.6 }** !Expression will be evaluated

This is the correct way to write the macro definition when you want to store the result of the expression, "51.6", in the macro VAL.

If an expression is used in a command, the parser may misinterpret the first part of the expression as a simple value, store that part of the expression as the value of a parameter, then assume that the rest of the expression is a syntax error.

Example: **ADD BOX AT 0,0 12 ∗ 1.6, 14 ∗ 1.8** !Syntax error

Function calls also need to be surrounded by curly braces to force evaluation.

This command will fail. The 12 will be parsed and then stored as the second x-coordinate. Succeeding characters cannot modify that value, and since the '∗' is not valid as the second y-coordinate, the program will respond with the following error message:

```
ADD BOX AT 0,0  12 <<∗>> 1.6, 14 ∗ 1.8
error: Unpaired coordinate
```

Example: **ADD BOX AT 0,0 {12 * 1.6}, {14 * 1.8}** !Correct syntax

Once the curly braces are added, the parser will evaluate each expression and replace the expressions with the results to execute the following statement:

```
ADD BOX AT  0,0  19.2, 25.2
```

There are some commands that expect an expression, such as the IF command. For these commands, the curly braces are not required around an expression. When in doubt, you can add curly braces around any expression. We will cover some exceptions to the requirement for curly braces that can save on typing, but you can always insure that an expression will be evaluated by surrounding it with curly braces.

Mathematical Expressions

Mathematical Operators

We discuss the precedence of operators on page 59.

Statements that require simple mathematical operations can use expressions with the operators shown in the table. When an expression using these operators is surrounded with curly braces, the parser will perform the operations and replace the expression with the result before the statement is executed.

Operator symbol	**Purpose**
+	Addition: single numbers or coordinate pairs
-	Subtraction: single numbers or coordinate pairs *or* reverse sign of single number or coordinate pair
*	Multiplication: single numbers or coordinate pair and number
/	Division: single numbers or coordinate pair and number

Figure 5: Mathematical Operators

(ICED™ also supports many functions to perform mathematical operations like trigonometric functions, minimums and maximums, and square roots. See the list of mathematical functions on page 220.)

Example: **MOVE SIDE X { 64 / 8 }**

The parser will evaluate the expression, perform the division operation, then execute the statement:

```
MOVE SIDE  X   8
```

Of course, mathematical operations are more useful in a command file when you use macros as the operands.

Example: **MOVE SIDE X { %TOT_WIDTH / %NUM }**

The parser will replace the macro references with the numbers stored in them, evaluate the expression, then execute the statement. However, if the NUM macro contains the number 0, the statement will fail with the message:

```
error: Cannot divide by 0
```

The '-' operator can be used to reverse the sign of a single number.

Example: **ADD BOX 1, {- -3} 5, 5**

When the '-' operator is used in this fashion, the curly braces are required since evaluation is required to reverse the sign of the number.

Example: **ADD BOX 1, –1 10, 10**

In this case, the '-' indicates that the sign of the first y-coordinate is negative. You do not need to add curly braces to enter negative coordinates where the '-' is followed by a single coordinate. Note that even though commas are really ignored by the parser, it will not misinterpret the string "1, -1" as "1 –1" and replace this string with "0". The requirement for curly braces to force evaluation makes this statement unambiguous.

Mathematical Operations on Coordinate Pairs

The POSN function returns a single coordinate pair from a position list. See page 234. The X and Y functions return a single coordinate from a pair. See page 246.

Mathematical calculations can be performed on single numbers or on coordinate pairs. When performing math on coordinate pairs, you must specify coordinates with the following syntax:

(*x-coord*, *y-coord*)

Where *x-coord* and *y-coord* are single real numbers. Blanks can be inserted for readability, but the **parentheses and the comma are required** for the parser to be able to perform mathematical operations on each coordinate of the pair.

Example: **ADD BOX AT (5,5) {(0,1) + (10,11)}**

This statement will result in addition of coordinate pairs. The x-coordinate "0" will be added to the x-coordinate "10", and the y-coordinate "'1" will be added to the y-coordinate "11". The result of the expression, "(10,12)" will replace the expression and the program will execute the statement:

```
ADD BOX AT (5,5)  (10,12)
```

When you need to perform calculations on the values stored in macros, the syntax is exactly the same. Be sure to surround the coordinate pair with parentheses and separate each coordinate in the pair with a comma.

Example:

LOCAL #COORD = (5,5)
LOCAL #X_DISP = 10
LOCAL #Y_DISP = 12
ADD BOX AT %COORD {%COORD + (%X_DISP,%Y_DISP)}

It is a good idea to define macros that represent coordinate pairs with the full (*x-coord,y-coord*) syntax as shown in this definition of the COORD macro.

You can use the '-' operator to subtract coordinate pairs, to reverse the sign of a single coordinate, or to reverse the signs of both coordinates of a pair. See the examples in the table. The examples using the macro COORD assume that COORD = "3".

Expression	**Result**
{(3,3) – (2,2)}	(1,1)
{-(3,3)}	(-3,-3)
{-(-3,3)}	(3,-3)
{(-%COORD, %COORD)}	(-3, 3)
{-(%COORD, %COORD)}	(-3, -3)

Figure 6: Using the '-' operator on coordinate pairs.

Multiplication or division can also be performed using a coordinate pair and a number.

Example:

LOCAL #MULT = 12
LOCAL #DISP = {%SNAP.STEP ∗ %MULT}

See page 293 for more information on SNAP.STEP.

This example performs multiplication on a coordinate pair. The system macro SNAP.STEP contains a pair of numbers that represent the minimum displacement in the x and y directions for the snap grid. Let us assume that SNAP.STEP is set to (0.5,0.5). After macro substitution the statement above would be:

The ROUND function resolves a coordinate to the resolution grid. See page 235.

```
LOCAL #DISP = {(0.5,0.5) ∗ 12)}
```

After performing the multiplication, the statement executed will be:

```
LOCAL #DISP = (6,6)
```

Boolean Expressions

Boolean Values

A Boolean expression is one that evaluates to either TRUE or FALSE. ICED™ implements TRUE and FALSE with simple numbers. When a Boolean expression is evaluated, TRUE and FALSE are defined as:

The ICED™ functions that return a Boolean value always return a 0 for FALSE and a 1 for TRUE.

FALSE = 0
TRUE = 1 or any non-zero number

Boolean expressions are used most often in condition expressions for WHILE (page 212), IF (page 168) and ELSEIF (page 171) commands. They are used to define a condition that determines whether or not a block of statements will be executed.

If the Boolean condition expression is TRUE, the WHILE, IF, or ELSEIF command will cause ICED™ to execute the statement or block of statements controlled by the command. If the expression is FALSE, the statement(s) will not be executed.

When a Boolean expression is contained in the condition expression between the "()" in an IF, ELSEIF, or WHILE statement, the curly braces "{}" are not required to force evaluation. The parser already expects an expression in this case. However all other uses of Boolean expressions should be contained in curly braces to force the parser to replace the expression with the result of the expression.

A Boolean condition expression can be as simple as a single number.

Example: **IF (0) $This statement will not be executed.**

Since the Boolean condition above is FALSE, the statement will **not** be executed.

Example: **IF (1) $This statement will be executed.**

Since the Boolean condition is TRUE, the statement will be executed.

Example: **IF (7) $This statement will be executed.**

This Boolean condition is also TRUE since it is non-zero. The statement will be executed.

Number Comparison Operators

See page 65 to learn how to compare strings.

There are six operators to compare numbers. The result of expressions that use these operators is either a '0' for FALSE or a '1' for TRUE.

	Purpose
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
>=	Greater than or equal to
>	Greater than

Figure 7: Number Comparison Operators

Example: **IF (6 <= 7) $This statement will be executed.**

The parser will evaluate the expression in the IF condition and determine that it is TRUE. The expression will be replaced with the value '1'. The statement that will be executed is:

IF (1) $This statement will be executed.

You must use a double equals operator "==" to compare two numbers. A single equals "=" in an expression will result in a syntax error.

Example: **IF (6 == 6) $This statement will be executed.**

The statement above is an example of the correct way to compare two numbers in a condition expression when you need to determine if the two numbers are equal. The parser will evaluate the expression in the IF condition and determine that it is TRUE. The expression "6==6" will be replaced with the value '1'. The statement controlled by the IF command will be executed.

The "not equals" operator ("!=") will result in an expression evaluating to FALSE when two numbers are equal.

Example: **IF (6 != 6) $This statement will not be executed.**

It is unlikely that you will be comparing two numbers explicitly like this. It is more common that one of the numbers is contained in a macro.

Example: **IF (%COUNTER < 10) $This statement may be executed.**

The parser will replace the macro reference with the value stored in the macro. Then the expression will be evaluated. For example, if the value of the COUNTER macro is 6, then the expression will evaluate to TRUE since 6 is less than 10. The expression will be replaced with the value '1'. The $*comment* statement will then be executed.

Boolean Operators

If you need to create a more complex Boolean expression, you can use the "&&" (Boolean AND) and "||" (Boolean OR) operators to combine the results of several Boolean expressions into a single Boolean value.

Purpose	
&&	Boolean AND
||	Boolean OR

Figure 8: Boolean operators

A Boolean AND operation will evaluate to a TRUE ('1' in this case) only when **both** operands are TRUE. Either operand is TRUE whenever it is a non-zero number. The result of the expression is always '1' or '0'.

op1	*op2*	*op1 && op2*
TRUE	TRUE	1
0	TRUE	0
TRUE	0	0
0	0	0

Figure 9: Boolean AND

Example: **IF (7 && 5) $This will be executed.**

This Boolean expression will evaluate to '1', and the statement controlled by the IF will be executed.

It is more common to have both operands be Boolean expressions.

Example: **IF (7 <=10 && 5 >= 3) $This statement will be executed.**

The Boolean expression on the left, "7 <=10" will evaluate to '1'. So will the Boolean expression on the right, "5 >=3". The statement will then be:

IF (1 && 1) $This statement will be executed.

Since both operands are TRUE, the expression "1 && 1" will evaluate to '1' and the statement controlled by the IF command will be executed.

However, when one of the operand expressions evaluates to FALSE ('0'), then the result of the AND operation will also be FALSE ('0').

Example: **IF (7 <=10 && 5<= 3) $This statement will not be executed.**

The Boolean expression on the right, "5<=3" will evaluate to '0'. The statement will then be:

IF (1 && 0) $This statement will not be executed.

Since one of the operands is FALSE, the expression "1 && 0" will evaluate to '0'.

A Boolean OR operation will evaluate to a TRUE ('1') whenever **either** operand is TRUE. As with an AND expression, the operands are TRUE whenever they are non-zero numbers, but the result of the expression will be '1' if it is TRUE.

op1	***op2***	***op1 || op2***
TRUE	TRUE	1
0	TRUE	1
TRUE	0	1
0	0	0

Figure 10: Boolean OR

Example: **IF (7 <=10 || 5<= 3) $This statement will be executed.**

The Boolean expression on the left, "7 <=10" will evaluate to '1'. The Boolean expression on the right, "5<=3" will evaluate to '0'. Since one of the operands is TRUE, the expression "1 || 0" will evaluate to '1'.

Compound Boolean Expressions

You can use more than one "||" or "&&" operator in a Boolean expression. You can combine as many operand expressions as needed. (You may sometimes need to surround some expressions with parentheses "()". We will discuss this subject beginning on page 61.)

Example:

```
IF (    %X_COORD < 0        ||        &
        %X_COORD >10000  ||        &
        %Y_COORD < 0        ||        &
        %Y_COORD >10000  )….
```

In the example above, if the value of X_COORD is outside of the range 0:10000, or if the value of Y_COORD is outside of the same range, then the compound Boolean expression will evaluate to TRUE and the statements controlled by the IF command will be executed. The '&' continuation characters make this statement much easier to read than if it had all been typed on one line.

Storing the Result of a Boolean Expression

You can assign the result of a Boolean expression to a macro just as you would any number.

Example:

```
#RESULT = {%DISTANCE < 10.5}              !correct syntax
```

This is the correct syntax to use if you use a Boolean expression outside of an IF, ELSEIF, or WHILE command. The macro substitution and evaluation will be performed and either TRUE ('1') or FALSE ('0') will be stored in the macro RESULT.

Example:

```
#RESULT = %DISTANCE < 10.5                !missing {}
```

When the curly braces are omitted, no evaluation of the Boolean expression is performed. If the value of the DISTANCE macro is 10, this statement will store the string "10 < 10.5" in the macro RESULT. This probably not what you intended.

Programmers often call a simple stored Boolean value a "flag". Your command file may want to perform some action if the flag is TRUE, and/or some other action when it is FALSE. A Boolean condition expression in an IF, ELSEIF, or WHILE command can be as simple as the value of a single flag macro, as seen in the next example.

Example:

```
LOCAL #SUCCESS=0
.
.       !Missing statements that set %SUCCESS to non-zero number if the
.       !command file performs the operation correctly
.
IF (%SUCCESS )      RETURN;$ The command file was successful
.
.       !Command file continues to solve problem
.
```

The flag macro in this case is named SUCCESS. If it is non-zero when the IF command is executed, then the command file will terminate at that point with the RETURN command. If the flag is FALSE, the command file will continue.

The NOT Boolean Operation

While the "!=" operator can be used to determine when two numbers are not equal, ICED™ does not currently support a true NOT function or operator. However, it is very easy to use this syntax instead:

***val* == 0**

When *val* is FALSE, then the result of this expression is TRUE. If *val* is TRUE, then the result of the expression is FALSE.

Let us say that you need to perform an action when a flag macro is FALSE. If the flag macro is the same SUCCESS macro we used in the previous example, we could write an IF statement as follows:

Example:

```
IF (%SUCCESS == 0 )    RETURN;$ The command file was not successful
```

In this case, the statement controlled by the IF command will be executed when the SUCCESS macro is FALSE.

Operator Precedence and Associativity (or, when are () required in an expression?)

When you have a complex mathematical or Boolean expression, the order in which the operations should be performed may often look ambiguous.

{2 + 3 * 4}

Does this simple expression really mean

{(2 + 3) * 4} → { 5 * 4} → 20

or

{2 + (3 * 4)} → { 2 + 12} → 14

The correct answer is 14.

Of course, if you have forgotten your grammar school algebra, and don't want to bother looking it up, you could always add the parentheses and write the expression as "{2 + (3 * 4)}". This is perfectly acceptable. In fact if you prefer to skip this entire explanation, and always add extra parentheses when an expression looks ambiguous, that is also perfectly acceptable. ICED™ will always evaluate portions of expressions surrounded by parentheses before evaluating the rest of the expression.

ICED™ uses the same precedence of operators and rules of associativity as the "C" programming language (except of course for the missing "C" operators that ICED™ does not support), so if you are an experienced programmer, you also get to skip this material.

Those of you in the above categories can skip ahead to page 63.

Definitions of Precedence and Associativity

Precedence is the order in which different operators will be evaluated in the absence of parentheses.

In the example above:

```
{2 + 3 ∗ 4}
```

the '∗' operator has higher precedence than the '+' operator. The expression "3∗4" will be evaluated first and replaced with "12" before the addition operation is evaluated.

If you want the addition to be performed first, you would have to force the program to do this with parentheses and write the expression as:

```
{(2 + 3) ∗ 4}
```

When you write an expression using multiple operators at the same level of precedence, the order in which the operations are evaluated is determined by their **associativity**.

In other words, the expression using the operator with the highest precedence is performed first. But if two operators have the same precedence, an operand is grouped with the operator on the right or left to form the first expression evaluated depending on whether the operator is right-associative or left-associative.

All ICED™ operators that support associativity are left associative.

Let us consider the division operator.

{12 / 2 / 3}

Which division operation is evaluated first? The '2' operand is the operand that could go either way, and since the division operator is left-associative, it will be associated with the '/' operator on the left and "12 / 2" will be evaluated first. The expression will then be:

```
{6 / 3}         which evaluates to '2'.
```

If you wanted the expression evaluated the other way, you would have to force this order with parentheses:

{12 / (2 / 3)} ! this evaluates to 18

Table of Precedence of ICED™ Operators

This table has the operators at the highest level of precedence at the top, and precedence decreases as you go down the column. Operators in the same row, have the same precedence. With one exception, the order of evaluation of expressions using operators in the same row is left-associative.

Operator	**Associativity**
- (when used to reverse sign)	left
* /	left
+ - (when used to subtract)	left
<= < == != > >=	non-associative
&&	left
||	left

Figure 11: Precedence Table for ICED™ operators.

The exception is the row of number comparison operators. These operators are non-associative. This means that you cannot use more than one of them in an expression unless you specify the order of evaluation yourself using parentheses.

{1 < 2 < 3} !Syntax error

The expression above is illegal because the '<' operator is non-associative. You must add parentheses to explicitly define the order of evaluation.

{1 < (2 < 3)} !Will evaluate to 0

Precedence in Compound Boolean Expressions

One common area where it is easy to make mistakes because of precedence rules is Boolean expressions using both the "&&" and "||" operators. The "&&" operator has higher precedence, so expressions using a Boolean AND will be processed first.

Example:

```
IF (    %X_COORD < 0        ||   &
        %X_COORD >10000     ||   &
        %Y_COORD < 0        ||   &
        %Y_COORD >10000     && %ERROR ==0)….  !Not written well
```

This example shows a compound expression that is written poorly. The intent is that the statements controlled by the IF command will not be executed whenever the ERROR flag has been set to a non-zero number. However that is not the result.

Let's take a closer look at this condition expression assuming that ERROR = 1, X_COORD = -5 and Y_COORD = 3. Then the expression looks like:

```
IF (    -5 < 0       ||     &
        -5 >10000    ||     &
        3 < 0        ||     &
        3 >10000     &&     1==0)….
```

The expressions with the higher precedence number comparison operators are evaluated first resulting in the following expression:

```
IF (    1            ||     &
        0            ||     &
        0            ||     &
        0            &&     0)….
```

Then the "&&" operator is processed.

```
IF (    1            ||     &
        0            ||     &
        0            ||     &
        0                   )….
```

The result is '1' since the "&&" operation in the last line is processed before the "||" operations. Only when the final "%Y_COORD >10000" is TRUE will the value of ERROR have any effect on the result of the expression.

If you want to prohibit the execution of the statements controlled by the IF command whenever the ERROR flag is TRUE, you must surround the "||" expressions with parentheses. This forces them to be evaluated as a group before combining the result with the result of the "%ERROR ==0" expression.

Example:

```
IF ( (  %X_COORD < 0      ||      &
        %X_COORD >10000   ||      &
        %Y_COORD < 0      ||      &
        %Y_COORD >10000  )   && %ERROR ==0)….
```

Now, if the expression is evaluated with the same values as above, it resolves to the following expression:

```
IF ( (  1             ||      &
        0             ||      &
        0             ||      &
        0             ) &&    0)….
```

The compound Boolean OR is then processed resulting in the expression:

```
IF ( 1 && 0)
```

The final result of the expression is now '0', or FALSE. The value of the ERROR macro now controls whether or not the statements controlled by the IF command are executed, regardless of the result of the other tests.

Functions

ICED™ supports many functions. Some perform mathematical operations, others manipulate strings or coordinate lists. New functions are often added with program updates. The complete list as of the writing of this manual is shown on page 219. Descriptions and examples of each function in alphabetical order follow this list.

Syntax for Function Calls

All functions are called using a similar syntax:

***Function_name*(*argument*)**

The parentheses are required, and the open parenthesis '(' must be typed immediately after the function name with no space in between. This forces the parser to recognize that the string represents a function name and argument.

Calling a Function in a Command

When a function call cannot be misinterpreted by the parser, you do not need to enclose the function call in curly braces. This includes uses of a function call in a command where the command expects a number. The parser will call the function and replace the function call string with the result.

Example: **ADD BOX AT 0,0 COS(45), SIN(45)**

When the parser interprets "COS(45)" it expects a number, but sees a function call instead. The parser is smart enough to realize that it can evaluate the entire string "COS(45)" to a number and will do so. The same is true for "SIN (45)". The statement executed will be:

```
ADD BOX AT 0,0  .707  .707
```

Expressions in the Argument of a Function Call

Expressions in the argument of a function call will be evaluated without the need for curly braces. The parser expects an expression in this case.

Example: **ADD BOX AT 0,0 COS(45 + 10), SIN(45 - 10)**

This statement will be evaluated to the following before it is executed.

```
ADD BOX AT 0,0 0.574, 0.423
```

Calling a Function in a Macro Assignment

When a function call is used to set the value of a macro, it must be enclosed in curly braces. In this case, the parser is not already expecting a number, so the parser will assume that function call is a simple string.

Example: **#MAC = COS(45 + 10)** !Function call will not be evaluated

In this case the value assigned to the macro MAC will be the string "COS(45 + 10".

However, when you surround the function call with curly braces, the parser will be forced to evaluate the string and store the result in the macro.

Example: **#MAC = {COS(45 + 10)}** !Function call will be evaluated

The parser will now consider the function call an expression that must be evaluated. The value stored in the MAC macro will be "0.574".

String Comparison Functions

Two functions that are used frequently by writers of command files are the string comparison functions.

Remember that the "==" operator is used only to compare numbers. To compare strings, you must use one of these functions.

	Purpose	**Page**
CMP	Compare two strings, ignore case	224
XCMP	Determine if two strings are e**X**actly the same, including case of each character	224

Figure 12: String Comparison Functions

Example: **IF (%RES.MODE == "HARD"){** !Incorrect syntax

RES.MODE is a system macro that stores the resolution mode for coordinates. See page 290.

The statement above fails with the error message:

```
IF (HARD <<>> == "HARD"){
error: Expected ( or { following function name
```

This is because it encounters the "==" operator, but no numeric operand was to the left of the operator. It then attempts to interpret the string contained in the system macro RES.MODE (in this case "HARD") as function call, but it cannot find the argument in parentheses that it expects.

Example: **IF (CMP(%RES.MODE, "HARD")==0) {** !Correct syntax

The CMP function will return 0 if the two strings match regardless of case. When the RES.MODE system macro contains the string "HARD", the statement above will evaluate to:

```
IF (0==0){
```

In this case since the two numbers are equal, the condition expression evaluates to TRUE and the statements in the IF block will be executed.

The CMP and XCMP functions do not return Boolean values. They follow the "C" programming language convention of returning a value of '0' when two strings match, a negative value when *string1* is less than *string2*, and a positive value when *string1* is greater than *string2*. While this makes the functions more useful, users that expect a Boolean return value often make mistakes.

Calculations in Layout Editor

You can type mathematical operations or function calls in the layout editor to perform the functions of a calculator. Simply type the mathematical expression on the command line surrounded by curly braces and prefixed by a '$' comment indicator.

Example: **${ 20.2 + ((56/2)*3) }**

Typing this command on the command line will report the result of the mathematical expression on the prompt line.

Another method for performing calculations in the layout editor is to execute the CALC.CMD command file supplied with the installation. Simply type on the command line:

Example: **@CALC**

You will be prompted to enter an expression. You can enter it without the curly braces, since they will be added by the command file. The result of the expression is reported on the prompt line.

Summary of Special Characters

Here is a review of all special syntax characters used in command files:

#macro_name Whenever you want to set the value of a macro, prefix the macro name on the left side of the '=' with a '#'. (Details on page 39.)

%macro_name When you want ICED™ to replace a macro reference with the value of the macro, prefix the macro name with a '%'. (Details on page 39.)

%macro^string The '^' character is used to delimit the end of a macro name without a blank space. (See details on page 44.)

%- When you need a real '%' character in a string where characters that could be misinterpreted as a macro name immediately follow the '%', you must type the combination "%-". The '-' is stripped from the string by the parser, but no attempt at macro substitution will be performed. (Details on page 41.)

%%macro_name Using a double percent instead of a single percent means that the value of the macro will not be substituted until it used in an executed command. This is referred to as delayed substitution. (Details on page 46.)

***keyword*=%** When you type a command in a command file, and you want the user to supply the value of a parameter when the statement is executed, you can specify the parameter with a '%'. (Details on page 36.)

**$** In a macro assignment statement, the '$' is used to indicate that the user will need to define the value of a macro at execution time. The method the user must use to define the value is determined by the following keyword. See page 143.

$*comment*	The '$' at the beginning of a statement indicates that the string that follows is a comment that will be echoed in the journal file. The comment will also be echoed on the prompt line of the layout editor window. When a $*comment* is the last line executed in a command file, the comment will still be displayed on the screen when control is returned to the layout editor.
$$*comment*	If a comment is prefixed with "$$", it will be processed even if the LOG mode prevents the logging or display of $comments. See page 162.
!*comment*	This type of comment is not echoed. Comments like these can be very useful in making your command file easy to understand, but they are ignored by the parser. Comments can be added at the end of any line or on lines of their own.
@file_name	This indicates that the statements in the specified command file should be executed. (Details on page 159.)
stmt*;*stmt	A ';' allows more than one command to be typed on one line. When the first statement is a RETURN command, the additional statements are executed before the command file is terminated. (See an example on page 58.) When the first statement is a *@file_name* command, macro definitions or other statements can be typed on the same line. The additional statements are executed as though they are statements in the command file. (See page 97 for examples.)
***stmt*;**	Semicolons at the end of a statement are ignored. You can type them if you are used to programming this way.
{*expression*}	Curly braces force an expression to be evaluated. Otherwise, the expression is often treated like an ordinary string of characters. (Curly braces in a quoted string do not force evaluation. Therefore, you can use either '{' or '}' in quoted string without causing the program to attempt an evaluation of an expression.)

"*string*"	
'*string*'	
~*string*~	
`` `*string*` ``	Any of these characters can be used as a pair of quotes. They are all treated in exactly the same manner. Having more than one pair of valid quotes allows you to include quote characters in a quoted string. (See an example on page 142.) No processing of special characters (except for '%' macro references and '&' line continuation characters) will take place in a quoted string.
&	When an '&' is added at the end of a line of characters, after a whitespace character, it suppresses the end-of-line indicator. At the end of a line of text in a command file, this indicates that the statement continues on the next line. This can be used to make long statements more readable. The '&' is processed first, before the statement is interpreted. (Details on page 30.) Another way the '&' can be used to suppress an end of line character is to use a '&' at the end of a keyboard macro string.
&&	Boolean AND operator. (Details on page 55.)
||	Boolean OR operator. (Details on page 56.)
<	Less than operator.
<=	Less than or equal to operator.
==	Equals operator. When you need to compare two numbers in an expression you must use the double equals, "==".
!=	Not equals operator.
>=	Greater than or equal to operator.

>	Greater than operator.
+	Addition operator.
-	Subtraction or sign reversal operator.
*	Multiplication operator.
/	Division operator.

Review of Statement Parsing

Let us review briefly what happens as a statement like the one below is parsed now that you can easily follow what happens when ICED™ parses a statement.

```
ADD CELL MYCELL AT                &
          {X(%MYITEM.POS.%N)},    &
          {Y(%MYITEM.POS.%M)}
```

Step 1) Command lines are concatenated when a line ends with a '&'.

```
ADD CELL MYCELL AT {X(%MYITEM.POS.%N)},{Y(%MYITEM.POS.%M)}
```

Step 2) Macro references with a '%" prefix are replaced with the contents of the macro. Macro references are replaced from right to left.

```
ADD CELL MYCELL AT {X(%MYITEM.POS.1)},   {Y(%MYITEM.POS.2)}
ADD CELL MYCELL AT {X((-14.0,42.5))},    {Y((-9.5,58.5))}
```

Step 3) Expressions are evaluated. (This includes function calls.)

```
ADD CELL MYCELL AT -14.0,                58.5
```

Step 4) The command is executed.

Overview of Programming Techniques

The exercises in the Classroom Tutorials Manual are another great way to learn command file programming techniques.

This section covers general information and techniques that are very useful when creating different types of command files. The organization is task-oriented. You can refer only to the subjects you need to complete a specific command file. However, reading all of the subjects will help prepare you to be a well-rounded command file programmer.

Selecting Components

Command files that operate on components must select those components before the operation, or make sure that they are already selected. The process of insuring that the correct number and right type of components is selected is very important. Many operations will cause an error and terminate the command file if the wrong number or types of components are selected. By default, most operations will pause waiting for input when no components are selected. Either situation can be very confusing to the user of your command file.

Selection Status at the Beginning of a Command File

When a command file begins, the selection status of components is not cleared. That is, if components are selected prior to the execution of the command file, they will be operated on by all relevant commands in the file.

For example, say part of your command file swaps components on layer OLD to layer NEW. The lines will be similar to those below:

Example:

```
SELECT LAYER OLD ALL
SWAP LAYER OLD AND NEW
```

Suppose that there are components on LAYER NEW already selected before the command file begins. When this is the case, the SWAP command will swap these components from layer NEW to layer OLD.

If you want to make sure that the only components selected are selected during your command file, you can add an UNSELECT ALL or UNSELECT PUSH command near the beginning of the file. The UNSELECT PUSH command has the advantage that you can add a SELECT POP command at the end of your command file to restore the selection status of components. (See page 111 for more details.)

If you want to check how many components are selected, use the N.SELECT system macro. See an example on page 96. We'll go into methods of verifying that the correct type of components are selected on page 176.

Embedded Selects and the XSELECT Mode

Holding down the <Shift> key during a embedded SELECT command allows the user to select multiple components.

When no components are selected prior to executing a component modification command, the command will issue an embedded SELECT command. This embedded SELECT command will pause and wait for the user to select a component. You may want to use this behavior in some circumstances. See page 85 for an example that uses this feature to allow the user to select a component for a copy operation.

However, in many cases these embedded SELECT commands (primarily intended for use outside of command files) cause problems in a command file. For example, let us consider the SWAP OLD to NEW example on the previous page assuming that no components are already selected. If no components exist on layer OLD, then the SEL LAYER OLD ALL command would select no components. This is not an error. Since no components are selected, the SWAP command would then issue an embedded SELECT command. This embedded SELECT command will wait for the user to select something. The only way to continue without selecting something is to cancel the command file.

See methods to cancel a command file on page 125.

In this case, it would be much better if the SWAP command did nothing and the command file continued without error when no components exist on layer OLD. This is accomplished by adding the XSELECT OFF command to the command file before the SWAP command. All embedded selection commands are disabled by the XSELECT OFF command. We'll see an example on page 84.

Whenever you write a command file that contains a command that operates on selected components, and you want that command to "do nothing" when no components are selected, be sure to add the XSELECT OFF command to the command file. See page 215 for more details.

The UNPROTECT and UNBLANK Commands

Refer to these commands in the IC Layout Editor Reference Manual for more details.

Components or entire layers can be made unselectable prior to execution of your command file through the use of BLANK and/or PROTECT commands. The blank and protection status of components and layers is preserved as you execute a command file. If you want to make sure that all components are selectable for modification, your command file should include the UNBLANK ALL and UNPROTECT ALL commands.

You can save and restore the blank and protection status of layers. See page 258.

If your command file should not modify any protected or blanked components, your command file should not include these commands. If you want to leave the decision to the user of the command file, you can ask the user if protected or blanked components should be included in the operation. (See page 274 for an example.)

Selection Criteria

There are several ways to select components by their properties. For example you can select components by layer, by unique id number, or by component type (e.g. BOX, WIRE, CELL, etc). Reread the SELECT command description in the IC Layout Editor Reference Manual for complete details and many examples. A few SELECT command examples are provided on the next page.

Example: **SELECT LAYER PDIF+NDIF ALL**

The example above uses the LAYER keyword and a simple layer list to select all unblanked, unprotected components on layers PDIF and NDIF.

Example: **SEL IDS AFTER 6; UNSEL IDS AFTER 10**

This example selects all components with a unique id between 7 and 10. Note that the SELECT and UNSELECT keywords are often abbreviated.

Example: **SEL CELL=NAND IN**

This example selects all NAND cells within a selection box digitized by the user when the command is executed.

Example: **SEL SIDE NEAR %LAST_POS**

See more on the LAST_POS system macro on page 272.

This example selects the sides of all wires and polygons within the near box. The center of the near box will be the last position digitized by the user. The relevant components are called "partially selected". Only the selected sides are modified by commands like MOVE.

Example: **SEL PARTS**

If components are partially selected, they will be fully selected by this command.

Example: **SEL NEW**

This command will select all of the components operated on by the previous command. For example, if your command file used an ADD command to add a component, if this SELECT NEW command is the next command executed, it will select the component just created.

Selecting Components from a List

You can place all selected components on a list and them select them one at a time from the list. See the LIST command on page 182 for more details.

The SELECT stack operations (like UNSELECT PUSH) can also be used to build a set of selected components. In this case, a single set of components on the stack can all be selected at once. See an example of this on page 185.

Selecting A Single Component

Use the *item*.TYPE macro created by the ITEM command to test what type of component is selected. See page 176.

Sometimes you need to insure that a single component is selected. For example, the ITEM command will fail unless exactly one component is selected.

This type of operation is so common that we show you how to create a reusable command file that you can call from your command file to insure that exactly one component is selected. This example is covered on page 96. If you prefer you can use the statements in this example directly in your command file to select a single component.

Allowing User to Use Multiple SELECT Commands

It can occasionally be difficult for the user to select the exact group of components necessary with a single SELECT command in your command file. A series of SELECT and UNSELECT commands, or the use of the 2:SHOW→@one menu option, can often be the easiest or only way for the user to select the required component(s).

There are two ways to allow this.

1) Create the command file so that it will work on components already selected. You should have some sort of IF or WHILE block with SELECT commands to handle the case where the correct components are not selected prior to the command file.
2) You can use a shell allowing the user to issue as many commands from the keyboard or regular menus as required to select the correct component(s). Refer to the SHELL command on page 207.

Adding Components

Command files often add components to the current cell with ADD commands. You may also occasionally find the COPY command useful for adding copies of new or existing components. (See an example on page 272.)

ADD Commands

The following features make ADD commands easier to write in command files:

***keyword*=% prompts** allow the user to type the missing parameters when the command file is executed without a lot of extra statements.

The OFFSET keyword allows you to define coordinates as offsets from a base set of positions without extra mathematical calculations.

The use of the '%' symbol to take the place of missing parameters is supported in other commands besides the ADD command, but it is most useful in the ADD TEXT command. This is because the user will usually be prompted for missing required parameters, but the text string must be included as the first parameter in any ADD TEXT command, and if additional parameters are required something must take the place of the missing text.

Example: **ADD TEXT=% AT %LAST.POS**

This ADD TEXT command will prompt the user with the prompt "Enter text" when it is executed. You do not need to define any macros or write any prompt statements yourself to have the command file prompt the user for the text string. The text will then be added on the current layer at the last position digitized by the user since the reference to the LAST.POS system macro is used to specify the location of the new text component.

See more on the LAST_POS system macro on page 272.

Example:

```
ADD TEXT=%  LAYER=%  AT %LAST.POS
```

The user can select a layer from a menu of valid choices. See page 147.

If you prefer to prompt the user for the layer you can write the command as shown above. If the user cancels the command by typing an <Esc> instead of supplying either parameter, the remainder of command file containing this command will not be executed.

The following example demonstrates how the OFFSET keyword makes it easier to write ADD commands that create components offset from a base position.

Example:

```
ADD TEXT="I=%I, J=%J"                                  &
              OFFSET={%I*%ROW_STEP, %J*%COL_STEP}      &
              AT=%BASE_COORD
```

Note that in the ADD command the OFFSET parameter must come before the AT parameter.

This statement is intended to be included in a WHILE loop that increments the I and J macros. Each text component is created at an offset from a base position. You could write the ADD command without the OFFSET, but that would require extra function calls to return the X and Y coordinates of the base position resulting in the less readable statement:

Example:

```
ADD TEXT="I=%I, J=%J"                                  &
        AT=  {X(%BASE_COORD) + (%I*%ROW_STEP)},        &
              {Y(%BASE_COORD) + (%J*%COL_STEP)}
```

The X and Y functions return a single coordinate from a coordinate pair. See page 246.

The OFFSET keyword is even more useful when you are adding components that have a long list of vertices (e.g. wires or polygons). Without the OFFSET keyword, you would have to add the offset to each vertex in the ADD command. See the SERIAL.CMD advanced example on page 313 to see how the OFFSET keyword can be used to make ADD commands for complex polygons easier to write.

Adding Components with ITEM Macros

One other feature useful when adding components is the ITEM command. (See page 173.) This command stores a variety of information about a single selected component in a series of macros. You can use these macros in one or more ADD commands to create similar components. Or you can modify the values in the item macros and add a new component using the modified values by executing the ADD command string stored in the ADD.*item_name* macro. (See an example on page 180.)

Snapping Coordinates to Resolution Grid

The POSN function returns a single coordinate pair from a coordinate list. See page 234.

Coordinates in an ADD command will NOT be forced to lie on the resolution or snap grids. When coordinates are digitized using the mouse, then you can be sure that the coordinates are on the snap and resolution grids. However, if your command file uses mathematical operations to calculate coordinates, they probably do not lie on grid. You will need to perform extra processing to snap the coordinates to grid before using them in an ADD command.

The SNAP.STEP system macro (see page 293) can be used in coordinate calculations when you want to keep coordinates on the snap grid.

If you calculate coordinates, you should use the functions below to snap the coordinates to grid. If you prefer to avoid rounding coordinates to grid, your command file should check the value of the system macro RES.MODE to determine if the user has set the resolution mode to "SOFT". (See an example on page 290.) If the resolution mode is "SOFT", then the decision is up to you. However, if the resolution mode is "HARD" then the user has indicated that all coordinates should be on grid.

The ROUND() function is usually used to round a coordinate pair to the nearest point on the resolution grid. The ROUND1() or ROUND2() functions may be preferable in some special circumstances. See page 235.

Methods that Enable Undoing ADD Commands

One drawback to command files that add components is that after the user executes the command file, if he doesn't like the results, the UNDO command will not reverse the effects of the command file. One method to allow the user to undo all of the ADD commands is to add the components on a scratch layer, then prompt the user to accept or reject the components. If the user accepts the components, swap the layer of the components to a real layer. If the user rejects them, delete the entire scratch layer.

Any time you add components on a scratch layer, use layer numbers in the range 250:255. These layers are reserved for scratch layers in command files. Using other layers for scratch work may corrupt the design of some future user (maybe you) if he happens to use that layer for real design work.

Another method that will allow you to remove all of the added components is to keep track of the id numbers. Each component has a unique id number. If you store the id numbers of the first and last components added, you can use the id related keywords of the SELECT command to select just these components then delete them all with one DELETE command. The ED.CMD and UNED.CMD command files on covered beginning on page 318 use this method. UNED.CMD can undo the results of ED.CMD. If you store the id numbers in a manner consistent with ED.CMD, you can use UNED.CMD as an UNDO command for your command files.

Adding Components using SHOW Command File

The ED.CMD command file uses an interesting method for adding components. The SHOW SELECT FILE=*file_name* command will export selected shapes to a command file. An ADD command will be generated in the file for each selected component. If you execute this command file, it will create identical components. You may even execute this file in a different cell, or allow the user to edit it before executing the file. See page 318 of this manual for an example. See the IC Layout Editor Reference Manual for complete details on the SHOW command.

Deleting Components

Using DELETE commands in a command file has some interesting ramifications. Most of these methods involve careful selection of components before issuing the DELETE command. If you have trouble selecting exactly the components you need to delete, see page 74 of this manual or carefully re-read the description of the SELECT command in the IC Layout Editor Reference Manual to see if there are features that will assist you.

There are three things you must be especially careful with when selecting components for a DELETE command:

> UNSELECT all components that may happen to be selected in the current session before the command file is executed. The best method for this is the UNSELECT PUSH command. It is a good programming practice to restore these selections at the end of your command file with an UNSELECT POP command. (See page 111 for more details.)
>
> Insure that the components are neither blanked nor protected. This will prevent them from being selected for the DELETE. (See the BLANK and PROTECT commands in the IC Layout Editor Reference Manual.)
>
> When a DELETE command is executed and no components are selected, the default behavior of the DELETE command is to issue an embedded SELECT command and wait for the user to select a component to delete. This can be a puzzling event for the user of your command file. Add an XSELECT OFF command before the DELETE command to force the DELETE command to do nothing and continue to the next command when no components are selected.

See more about embedded SELECT commands on page 75.

The following example is a command file to delete components. It will delete all components on layer SCRATCH in the current cell. This command file is far more powerful than a single DELETE command since extra commands in the file insure that even protected or blanked components will be removed.

Example:

```
! CLR_SCR.CMD
UNPROTECT LAYER SCRATCH
UNBLANK ROOT LAYER SCRATCH
UNSELECT PUSH
SELECT LAYER SCRATCH ALL
XSELECT OFF
DELETE
XSELECT ON
SELECT POP
```

If no components exist on layer SCRATCH, the command file will not pause waiting for user input; it will simply have no effect.

We will expand this example on page 98 to prompt the user to enter the layer name. This makes the command file more useful since it can then be used to delete all components on any desired layer.

See page 22 to learn about executing a command file in all subcells. Only components in the current cell will be deleted by this command file. **Components on layer SCRATCH nested in subcells will not be deleted**.

User Interaction

Some command files (like the CLR_SCR.CMD example above) operate without user input. However, your command files can be far more versatile and powerful when the user has some control over how the command file operates.

When you want to allow the user to perform a series of editor commands before your command file continues, you can use the SHELL command. See page 207.

One of the most common types of user interaction in command files is prompting the user to supply a parameter. Several methods of prompting the user to supply a parameter are supported.

1) Using "*keyword*=%" in an editor command will prompt the user for the value when the command is executed. (See an example on page 79.) The value may be a location digitized by the user. No macro definition or creation of a prompt message is required.

2) Commands that operate on selected components will allow the user to select the component(s) when the command is executed (unless the XSELECT mode is off; see page 75.)

3) The user can be prompted to supply the value of a macro when the command file executes. This selection can be from a menu of valid choices. See page 147.

When you want the user to select a component using method 2, you should display a prompt so that the user understands what they are supposed to do.

Example:

```
UNSELECT ALL
PROMPT "Select component to copy."
COPY BY 1,1
```

Holding down the <Shift> key during a embedded SELECT command allows the user to select multiple components.

When no components are selected, the default behavior of the COPY command is to issue an embedded SELECT NEAR command and wait for the user to select the component. By default, no prompt message is shown on the screen. However, when you add the PROMPT command directly before the COPY command, that message remains on the screen while the program waits for the user to select the component.

When you want to use method 3 above to prompt the user to enter the value of a macro, there are several keywords that make this possible. Any time a macro is defined without a value, the user is prompted to supply it with the default prompt message "Enter value for macro *macro_name*." You can write a more explanatory prompt message yourself by adding the $PROMPT keyword to the macro definition.

Example: **LOCAL #LAY_NAME $PROMPT="Enter layer to delete:"**

You can let a user select a layer name from a menu of choices. See an example on page 147.

The macro definition statement above will display the indicated prompt message on the screen and wait for the user to type something on the keyboard. If the user types <Esc> instead of typing a value, the entire command file is cancelled. This method does not insure that a valid layer has been entered. We will explore methods of verifying a user response in a moment.

There are several response type keywords that require the user to digitize macro values with the mouse.

Example: **LOCAL #CENTER $PROMPT="Digitize center point." POSITION**

When the POSITION keyword is used in a macro assignment, the program will display the prompt string and wait for the user to digitize a position with the cursor. This insures not only that a valid coordinate has been entered, but that the point is on the resolution and snap grids.

Other user response keywords require the user to digitize a set of positions or make a selection from a menu of choices. See page 143 for more details and examples.

Methods of displaying a message to the user are covered beginning on page 88.

Other methods of getting the user to supply a value are covered in the next subject. These include several command files supplied with the installation that request the user to supply a specific type of value and go on to verify that the value is valid.

Verifying User Input

When you request that the user enter a macro value, no validation of the user's response is performed automatically. For example, suppose the user mistyped a layer name ("1M" instead of "M1") in the LOCAL #LAY_NAME macro definition example above. The command file contained a command like the following, it would fail with the mysterious message indicated:

Example: **UNPROTECT LAYER %LAY_NAME**

```
UNPROTECT LAYER  1 <<M>>
Error: Expected end of command
```

Whenever the user types in a macro value, and you do not verify that an appropriate value has been entered, a later command may fail with an error message that only confuses the user.

When you need the user to supply a coordinate, the best method is to let them digitize it with the cursor. See an example on the previous page. When the cursor is used to digitize a position, this insures not only that a valid coordinate has been entered, but that the point is on the resolution and snap grids.

You can let a user select a layer name from a menu of choices. See an example on page 147.

However, you may need to verify that the user has entered a macro value that represents valid layer name, integer, or other type of value. Several functions can be used to verify user responses. All of the functions in the table on the following page can be used in the following way:

Example: **IF (VALID_INT("%NUM_COPIES")) {**

In addition, several command files are provided with the program that can be called by your command file to insure that the user types a valid response. These command files support default values and range checking. They do not terminate until a valid value is stored, or until the user cancels the command file. See the list of these command files on the following page.

See a complete list of functions on page 220.

The ROUND function will snap a coordinate pair to the resolution grid.

These command files can be found in the AUXIL subdirectory of the installation. This is on the default command file search path.

Function name	**Purpose**	**Pg**
DEVICE_EXISTS	Test if device (e.g. printer) exists	226
DIR_EXISTS	Test if file directory exists	226
FILE_EXISTS	Test if file exists	227
STD_COORD	Format coordinate string	240
VALID_INT	Test that string is a valid integer	242
VALID_REAL	Test that string is a valid real number	245
VALID_LAYER	Test that string is a valid layer name	243
VALID_CELL_NAME	Test that string is valid as a cell name	241
Command file name		
_GET_ANS.CMD	User must select a character from a list of choices (usually "Yn" for yes/no)	163
_GET_DEV.CMD	User must enter the name of a valid printing device	-
_GET_INT.CMD	Requests that user enter integer, performs value and range validation	305
_GET_REAL.CMD	Requests that user enter number, performs value and range validation	100
_GET_LAY.CMD	Requests that user enter a layer and verifies that the response is a valid layer name or number	279

Figure 13: Value Validation

Displaying Messages to the User

There are several commands used to display a message to the user. In all of these methods, macro references in the message string will be replaced with the contents of the macro(s).

Command name	**Purpose**	**Pg**
PROMPT	Replaces the prompt to the left of the '>' on the command line, command file continues without pause	201
PAUSE	Displays message and waits until user presses button or key to continue	199
$comments	Displays message on command line and in log file, command file continues without pause	161
$$comments	Displays message on command line and in log file even when log mode is off, command file continues without pause	162
RETURN	Terminates command file leaving the message visible below command line	205
ERROR	Terminates command file leaving the word "Error" in red and the message visible below command line	164

Figure 14: Message commands

Quotes are required around the message string only when it could be misinterpreted by the parser.

Whenever a user response is required, insure that meaningful prompt is displayed. Both the PROMPT and $*comment* commands are useful for showing a prompt message when you have a command in your command file that requires the user to digitize coordinates or select components. The string from the previous PROMPT command will be displayed on the prompt line during the next user interaction command. See an example on page 96. The previous $*comment* command is visible below the command line. See page 271.

$*comment* commands are very useful when debugging a command file. Since these comments are recorded in the journal file, you can record the values of all of your important macros at any given moment in the journal file.

The LOG mode determines whether or not commands are logged to the journal file and the screen.

$$comments are useful in displaying updates during very long command files. They operate in the same manner as $comments except that they are displayed and logged even when the LOG mode is off. If you have a huge command file, or a very long loop, a few $$comments will indicate the progress of the command file and prevent the impression that the command file has crashed the session. (See examples on pages 131 and 162.) Leaving the LOG mode on might also show progress, but only at the cost of slowing down the command file considerably.

During debugging, if you want to pause the command file while the user looks at the comment message, use a PAUSE command. This allows the user to cancel the entire command file by pressing both mouse buttons if the macro values are not what were expected.

Example: **PAUSE "Processing cell %CELL. <Enter> to continue."**

You can use the SHOW command after a command file is over to display the contents of GLOBAL macros.

It is a good programming practice to provide a quick summary of the success of the command file in the RETURN statement. Whenever a message is included in an ERROR or RETURN command, the message will remain on the screen after the command file is complete. This type of message that tells the user exactly what action was taken by the command file and gives the user important feedback. For example, this may inform the user that no components were deleted on a given layer.

Example: **RETURN SUCCESS: %CLR.NUM_DELETED Components deleted & from layer %CLR.LAYER.**

Conditional Execution

The LOG LEVEL = DEBUG command can add comments to the journal file to help you debug command files with these commands. See page 127.

ICED™ command files do not have to execute every statement once in top-down order. The IF and ELSEIF commands give you the option to execute a block of statements if a Boolean condition expression is TRUE or FALSE. The WHILE command loops through a block of statements until a certain condition is met.

Command	Page
IF	168
ELSEIF	171
WHILE	212

Figure 15: Conditional statement commands

We have already covered Boolean expressions on page 53. Briefly, a Boolean expression evaluates to FALSE (the number 0) or TRUE (any non-zero number). When the Boolean condition expression is TRUE, the block of statements controlled by an IF, ELSEIF, or WHILE command will be executed.

Boolean condition expressions in these commands do not require curly braces. However, the expression should be surrounded by the ordinary parentheses "()" required by the command.

Curly braces are used in this context to indicate the block of statements controlled by the command. The first curly brace must be at the end of the same line as the IF, ELSEIF, or WHILE statement. The closing curly brace must always be on a line by itself.

Example:

```
IF (%N.SELECT == 0){                                          !(a)
        PROMPT Select components for operation                    !(a1)
        SELECT IN                                                 !(a2)
        IF (%N.SELECT > 1)  #MULT_COMPONENTS = 1                  !(a3)
        ELSE                #MULT_COMPONENTS = 0                  !(a4)
}
ELSEIF (%N.SELECT > 1){                                       !(b)
                            #MULT_COMPONENTS = 1
}
ELSE {                                                        !(c)
                            #MULT_COMPONENTS = 0
}
IF (% MULT_COMPONENTS){…                                      !(d)
```

The N.SELECT system macro contains the number of currently selected components. See page 289.

If the Boolean condition in statement (a) "(%N.SELECT == 0) evaluates to TRUE (i.e. the value of the N.SELECT system macro is equal to 0), then the statements in between the curly braces "{}", statements (a1) through (a4), will be executed.

After the statements in the block are executed, control passes to the next statement beyond any ELSEIF or ELSE blocks, statement (d).

If the condition in statement (a) evaluates to FALSE, then control passes directly to statement (b). This new condition, "(%N.SELECT > 1)" is then evaluated. If it is TRUE, then the statements between that curly brace and the next closing curly brace are executed. Control then passes to the next statement beyond any following ELSEIF or ELSE blocks, statement (d).

Only when the conditions in statements (a) and (b) both evaluate to FALSE, will statements in the (c) ELSE block be executed.

Note that it is acceptable to nest IF constructs. Statements (a3) and (a4) form a second IF/ELSE construct. These statements use single statement syntax rather than curly braces.

When the block controlled by an IF, ELSE, ELSEIF, or WHILE command is a single statement, the curly braces are not required, but the statement must be on the same line as the IF, ELSEIF, ELSE, or WHILE command. (The continuation character '&' allows you to type the statement on the next line.)

Example:

```
IF (%N.SELECT > 1)            &
        #MULT_COMPONENTS = 1       !interpreted as being on a single line
```

You can use an IF command without any following ELSE or ELSEIF commands. When ELSE or ELSEIF commands are used, no other statements can come between the end of the IF block and the ELSE or ELSEIF.

You can use as many ELSEIF commands as required to test all possible conditions. This command allows you to build the equivalent of a case or select structure found in other programming languages. However the ELSEIF is more versatile than a case statement because there are no constraints on the condition expressions in the ELSEIF commands. There is also no error checking to verify that you have written a consistent set of conditions. We recommend that you always include a final ELSE command to catch any conditions you may not have thought of when writing your list of ELSEIF commands.

Example:

The LEN function returns the length of a string. See page 229.

The VALID_LAYER function is described on page 243.

```
LOCAL #MY_LAYER $PROMPT =                        &
                                "Enter layer number [<Enter> to abort]:"
IF (LEN("%MY_LAYER")==0) {
        RETURN Command file aborted
}
ELSEIF (VALID_LAYER("%MY_LAYER")==0) {
        RETURN Invalid layer "%MY_LAYER" entered
}
ELSEIF ((%MY_LAYER == 0) && (%LAYER_0_VALID==0)) {
        RETURN Layer 0 is not acceptable
}
ELSE   USE LAYER %MY_LAYER  !MY_LAYER is a valid layer
```

The command file fragment on the previous page uses RETURN commands to abort the entire command file when certain conditions are met.

Note the compound condition expression using the Boolean AND operator "&&" in the second ELSEIF command. Both of these conditions must evaluate to TRUE for the block controlled by the command to be executed. When you write a compound expression like this, surround each expression with parentheses, and then surround the entire compound expression with another set of parentheses.

When writing long compound Boolean expressions, it is useful to use the line continuation character, '&', to write the compound expression on several lines as seen in the next example. Be sure to surround each expression with parentheses.

Example:

```
IF (   (%ERROR_FLAG == 0)                &&        &
       (  (%X_COORD == X(%MY_POS) ) ||             &
          (%X_COORD > 1)            ||             &
          (%Y_COORD == Y(%MY_POS) ) ||             &
          (%Y_COORD > 0)            ) ) {
```

Refer to page 61 to learn about operator precedence in compound Boolean expressions.

The five lines above are all part of one long IF statement. It is much easier to read the statement when it is typed this way, rather than typed all on one line. The first Boolean expression, "(%ERROR_FLAG == 0)", must be TRUE for the entire Boolean expression to be TRUE since the AND operator "&&" is used to combine it with the following compound Boolean expression. Only one of the following Boolean expressions must be TRUE for the entire statement to be TRUE since they are combined with the Boolean OR operator "||".

See an example of the importance of insuring that all conditions in a compound Boolean expression can be evaluated on page 177.

In some programming languages, if the first expression in a compound Boolean AND expression is FALSE, then the remainder of the compound expression is not interpreted. However, this is not the case in ICED™. The entire statement is interpreted and macro substitution is performed before any part of it is executed. This means that the entire statement must be valid or the command file will fail. For example, even if the macro ERROR_FLAG is set to 1, if the MY_POS macro did not exist, the IF command would fail with an error message and the entire command file would be aborted.

The WHILE command should be used when you need to continue executing a block of statements as long as a condition is TRUE.

Example:

```
WHILE(%VALID==0){
        #MY_LAYER=$PROMPT =   &
                                "Enter layer number [<Enter> to abort]:"
        IF (LEN("%MY_LAYER")==0) {
                RETURN Command file aborted
        }
        #VALID = {VALID_LAYER("%MY_LAYER")}
        IF(%VALID){
                IF((%MY_LAYER == 0) && (%LAYER_0_VALID==0)){
                        #VALID=0
                        PAUSE " Layer 0 not acceptable "
                }
        }
        ELSE{
                PAUSE " Invalid layer '%MY_LAYER' entered "
        }
}
```

This simplified fragment of the _GET_LAY.CMD command file will loop until a valid layer name or number is entered. The PAUSE commands leave the messages displayed until the user presses a key to be reprompted with the message "Enter layer number [<Enter> to abort]:".

The WHILE loop will continue to execute the statements before the final '}' until the WHILE condition expression evaluates to FALSE. This means that the value of the VALID macro must be non-zero when the condition expression is evaluated. Note that the VALID macro may be briefly set to 1 in the loop, then reset back to 0 before the WHILE condition expression is re-evaluated. This will not terminate the loop.

WHILE loops are often used to execute a statement or block of statements a certain number of times. In this case a counter macro should be incremented.

Example:

```
LOCAL #LAY_NUM = 1
WHILE (%LAY_NUM <= 10){
        #LAY_NAMES = %LAY_NAMES %LAYER.NAME.%LAY_NUM
        #LAY_NUM = {%LAY_NUM +1}
}
```

The command file fragment above will execute the block of statements 10 times. The last statement in the block increments the counter macro. Since %LAY_NUM + 1 is a mathematical expression, it must be surrounded by curly braces to force evaluation. (See page 48 for an explanation of expressions.)

Nesting Command Files

Command files can call other command files. In fact, you can nest command files 16 levels deep.

One important thing to remember when using nested command files is that macros defined as local macros in a nested command file will not be available in the calling command file. In addition, macros local to the calling command file will not be available in the nested command file. However, macros defined with the GLOBAL keyword can be used in both command files, no matter where they were defined.

You can use nested command files in a manner similar to subroutines in other programming languages. This makes many complicated tasks easier to write, debug, and reuse. You can also reuse working command file source in more than one command file without a lot of cutting and pasting.

Let us say that you have a command file that requires the user to select a single component. You cannot just have a prompt and a single SELECT NEAR, since SELECT NEAR may select more than 1 component. Selecting more than 1 component would make your command file fail.

The example on the next page uses an existing selection if exactly one component is already selected. If not, the statements will loop until exactly one component is selected.

Example:

```
! _SEL_ONE.CMD
LOCAL #DONE =0
WHILE (%DONE == 0){
        IF      (%N.SELECT != 1){
                UNSELECT ALL
                PROMPT "Please select a single component"
                SELECT NEAR
        }
        ELSE    #DONE = 1
}
```

Now let us say that you need to select exactly one component in more than one place in your command file. You could repeat these lines again, but your command file will be much neater and easier to read if you place the lines above in a separate command file with the name SEL_ONE.CMD.

Now you can add the following single line wherever it is needed to insure that exactly one component is selected.

Example:

```
@_SEL_ONE
```

This will make your command file much easier to write and easier to read.

What happens if the user cancels the SELECT NEAR command in the SEL_ONE command file by pressing both buttons? The answer is that the SEL_ONE command file is also canceled, and so is the command file that called it. Whenever a command in a nested command file is canceled, all calling command files are also canceled and control is immediately returned to editor.

Your _SEL_ONE.CMD file would be even more useful in other command files if it was enhanced to insure that exactly *n* components were selected. If you want to make a more multi-purpose command file like this, you need to be able to pass parameters like *n* into the nested command file. We cover how to do this next.

Passing Arguments into Command Files

ICED™ does not support passing arguments into a command file in the traditional manner (i.e. similar to a function call). However, you can simulate this type of argument passing.

One method is to use global macros for all values that the nested command file may need to access. However, this makes the nested command files less independent from the command file that calls it. The nested command file will not be as easy to use in other command files.

While global macros can be used to pass information into a command file, it is best to keep the use of global macros to a minimum. There is a better way to pass information into a command file.

Whenever you add additional statements on the same line as a nested command file reference, the extra commands are executed before the command file begins.

Example:

```
@MY_FILE;                                  &
        LOCAL #GEORGE="GEORGE BUSH";       &
        LOCAL #BILL="BILL CLINTON"
```

When we use this method to execute the nested command file MY_FILE.CMD, the macros GEORGE and BILL are defined as though they were the first lines in the nested command file. The &'s in this example indicate that the statement continues on the next line. The semicolons are required to separate the commands.

It is acceptable to omit the LOCAL scope definition keyword in macro definitions on the same line as a *@file_name* command. This type of macro definition will default to local scope, so the LOCAL keyword is not necessary.

Example:

```
@MY_FILE;                            &
        #GEORGE="GEORGE BUSH";       &
        #BILL="BILL CLINTON"
```

This statement is shorthand for the example given above.

The macro definitions provided on the same line as the *@file_name* command override macro definitions provided in the nested command file. If the contents of MY_FILE.CMD begin with the lines:

Example:

```
DEFAULT LOCAL #GEORGE="GEORGE JESSEL"
DEFAULT LOCAL #BILL="BUFFALO BILL"
```

Then those defaults will be overridden by definitions supplied on @MY_FILE statements.

However, the nested command file will use the default values defined in the file when it is called without macro definitions as shown below.

Example:

```
@MY_FILE
```

Now let us cover a somewhat more realistic example. The following command file will delete all components on a given layer despite their blank or protection status.

Example:

```
!CLR_LAYR.CMD
!Delete all components from a given layer.
!Use the following syntax to avoid the prompt for the layer:
!           @CLR_LAYR;  #CLR.LAYER = layer_name

DEFAULT   LOCAL    #CLR.LAYER $PROMPT=  &
                   "Enter layer to delete from current cell."
          LOCAL    #CLR.NUM_DELETED = 0

UNPROTECT LAYER %CLR.LAYER
UNBLANK ROOT LAYER %CLR.LAYER
UNSELECT ALL
SELECT LAYER %CLR.LAYER ALL
#CLR.NUM_DELETED = %N.SELECT
XSELECT OFF
DELETE
XSELECT ON

$SUCCESS: %CLR.NUM_DELETED Components deleted.
```

This example is expanded from a simpler example on page 84. See that page for explanations of the statements that delete the components.

When the command file above is executed with the following syntax, the user will be prompted for the layer to delete.

Example:

```
@CLR_LAYR
```

Since the CLR.LAYER macro is defined with the DEFAULT keyword, you can avoid the prompt for the layer by defining the CLR.LAYER macro on the same line as the @CLR_LAYR command.

Example:

```
@CLR_LAYR;  #CLR.LAYER=M1
@CLR_LAYR;  #CLR.LAYER=M2
```

The lines above will execute the command file twice. The first will clear the M1 layer; the second will clear the M2 layer. The user will not be prompted.

You can even use macro substitution to supply the parameters. Even though statements on the same line as the *@file_name* statement are executed as though they are inside the nested command file, the macro substitution is performed first in a separate step, so that macros local to the calling command file can be properly substituted.

Example:

```
LOCAL #LAY_NUM = 250

WHILE (%LAY_NUM <= 255){
        @CLR_LAYR; #CLR.LAYER=%LAY_NUM
        #LAY_NUM = {%LAY_NUM +1}
}
```

Even though the LAY_NUM macro is local to the calling command file, the macro substitution is performed as you would expect and the CLR_LAYR command file will be executed 6 times on the layers 250:255. No prompts for the layer will be issued.

You can also use IF commands on the same line as a *@file_name* command to conditionally pass values into a command file.

Example:

```
LOCAL #DISP=-9
LOCAL #MY_MIN=0
LOCAL #MY_MAX=100
```

The _GET_REAL-.CMD command file can be found in the AUXIL subdirectory of the installation. This is on the default command file search path.

```
@_GET_REAL; #MIN = %MY_MIN;                                  &
            #MAX = %MY_MAX;                                  &
            #PROMPT =                                        &
              "displacement in range %MY_MIN:%MY_MAX ";&
            IF (MACRO_EXISTS(#PREV_DISP))                    &
               #DEFAULT =%PREV_DISP
#DISP = {ROUND(%RET.VALUE)}
GLOBAL #PREV_DISP = %DISP
```

This example uses the _GET_REAL.CMD command file supplied with the installation to have the user enter a displacement. The MIN, MAX, and PROMPT default macros in the _GET_REAL.CMD file are overridden using macros local to the calling command file.

The DEFAULT macro is defined only when the macro PREV_DISP exists. The MACRO_EXISTS function is used to test if the macro has already been defined by a previous execution of these lines. It will return FALSE when the PREV_DISP macro does not exist. In this case, the _GET_REAL.CMD will complete without a default return value.

The macro RET.VALUE is created by the _GET_REAL command file. It is rounded to the resolution grid by the ROUND function before being stored in the macro DISP. If the resolution step size is set to .5, and the user types 2.61, the DISP macro will be set to 2.5. This value is then saved in a global macro for the next time these lines are executed.

The MACRO_EXISTS function will also tell you if an existing macro is local or global. For example, if your nested command file can be called with or without a default value for a parameter, you could use statements like these in your nested command file GET_SIDES:

Example:

```
IF(MACRO_EXISTS(#DEFAULT)==2){
        #MSG="Enter number of sides [%DEFAULT]:"
}
ELSE   #MSG="Enter number of sides"
```

This command file fragment will use the value in the macro DEFAULT in the message if that macro has been defined on the same line as the @GET_SIDES command as in the statement:

Example:

```
@GET_SIDES; #DEFAULT = 6;
```

However, suppose a global macro with the name DEFAULT is left over from some other unrelated command file. In that case MACRO_EXISTS would return a 1. If no macro with the name DEFAULT is defined, then MACRO_EXISTS would return a 0. In either case, the message will be built without a default indicated between "[]".

The MACRO_EXISTS function is fully described with other examples on page 231.

The GET_INT.CMD command file is another excellent example of how to use MACRO_EXISTS as well as other value passing features. See page 305.

Opening Other Cells

The Edit Commands

You are not limited to modifying only the current cell in a command file. You can use any of the edit commands (EDIT, PEDIT, and TEDIT) to edit other cells. The rest of this discussion will refer to the use of any of these three commands with the generic term "edit command".

The EXIT command will cause the edited cell file to be overwritten, even if it is unmodified.

Be sure to pair each edit command with an exit command (i.e. EXIT, LEAVE or QUIT) that will return you to the parent cell. In a command file that modifies geometry, the LEAVE command is the best termination command to use since it will not cause ICED™ to overwrite the cell file of any cell that was not actually modified. The QUIT command always returns to the parent cell without saving the edited cell. EXIT will always mark the cell file for saving.

Example:

```
LOCAL #MY_CELL  $PROMPT "Type subcell name for swap:"
LOCAL #NUM_SWAPPED = 0

EDIT CELL %MY_CELL
        UNSEL ALL
        SEL LAY M1+M2 ALL
        XSELECT OFF
        SWAP LAY M1 AND M2
        #NUM_SWAPPED = %N.SELECT
LEAVE
RETURN "%NUM_SWAPPED Components swapped"
```

The XSELECT OFF statement is critical to this command file example. See page 75 for details.

The command file above will prompt the user for a cell name, then use the EDIT command to begin editing that cell. The commands between the EDIT command and the LEAVE command will be executed in the nested cell rather than in the parent cell.

Opening Cells in VIEW-ONLY Mode

Sometimes, you only need to open another cell to get information on the contents. In these cases, you can use the VIEW_ONLY=TRUE option in an edit command when you do not intend to save the modified cell. In this case, you do not need to be concerned if the cell you are about to open might be in a protected library.

Example:

```
VIEW ON
UNSELECT ALL
PROMPT "Select cell for M1 component count."
        !continued on next page
```

The P_EDIT command will leave the parent cell displayed but dimmed out.

```
SELECT CELL * NEAR
IF (%N.SELECT == 1){
    PEDIT SELECT VIEW_ONLY=TRUE
        UNSEL ALL; SEL LAYER M1 ALL
        PAUSE "Cell %CELL has %N.SELECT components on layer M1."
    QUIT
}
ELSE ERROR "More than 1 cell selected."
RETURN
```

See DEEPSHOW-.CMD on page 325 to see another example of the usefulness of the VIEW-ONLY mode.

The P_EDIT command in the example above uses the SELECT keyword to edit the selected cell. This keyword is valid for all three edit commands. It will open the currently selected cell if exactly one copy is selected. This is the reason the P_EDIT command is located inside of a block executed only when a single component is selected.

Errors and Interruptions during Nested Edits

Suppose the user presses both mouse buttons to cancel the PAUSE command while the M1 component count example above is being executed. In this case, the rest of the command file will not be executed. This will leave the LEAVE command unprocessed. The nested cell will remain the current cell rather than the cell the user was editing when he executed the command file. This is likely to confuse the user. The only way he can return to the parent cell is to execute an exit command explicitly. Some users may not realize what has happened and may terminate the edit session with a JOURNAL command to recover.

Any time a command file is cancelled or terminated with a failed command while the command file was editing a nested cell, the editor remains in that nested cell.

The same situation happens when there is a syntax error in one of the commands after an edit command is processed. This is one of the most perplexing things that can go wrong with command files. Be sure to debug command files using nested edit commands carefully.

The Cell Table and Open Cells

See page 22 for an overview of executing commands on many cells.

Before we can explore more advanced cell edit methods, we need to review the meaning of the cell table and the definition of an open cell.

The cell table is a list of all cells loaded in the current layout editor session. This list includes the root cell (i.e. the cell the editor was launched to edit) and all of its subcells. The list also includes all cells that have been explicitly opened by edit commands and the subcells of those cells. (If either the QUIT or LEAVE command was used to unload the edited cell without saving, it and its subcells are removed from the cell table.) New cells created with the GROUP command are also in the list. When you delete a subcell, it is not removed from the cell table.

The MAX.CELL system macro contains the last valid index into the cell table. See page 286.

The root cell (i.e. the cell the editor was launched to edit) is always the first entry in the cell table with an index of 1.

Many system macros use the index of a cell in this cell table to access information about the cell. For example, the methods of testing whether or not a cell is stored in a protected cell library use the cell index to specify the cell in question.

A cell in the cell table may be **open**. When you launch ICED™ to edit root cell CellA, then use an edit command to edit CellX, CellA is still open, even though it is not the current cell. If an edit command tries to edit a cell that is already open, the command will fail and terminate the command file.

Cells that contain an open cell are also open. Consider Figure 16. CellA contains a subcell, CellB. CellB contains a subcell, CellC. Assume that you launch the ICED™ session to edit CellA. Then you used an edit command to open CellC. All three cells are now open. CellB is also open since it contains an open cell.

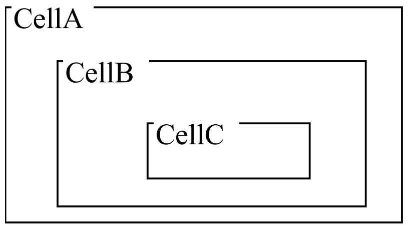


Figure 16: Example of nested cells.

Determining if a Cell Exists and is Loaded

If you do want to create a new cell, see page 110.

When you execute an edit command on a cell that does not exist in any of the known cell libraries, a new cell will be created in the current directory. Sometimes this is not the intended effect and can represent an error.

Consider the swap example on page 102. If the user mistypes the name of the cell, and enters a cell name that does not already exist in a cell library, a new cell will be created by the EDIT command.

See a complete description of the CELL function on page 221.

The CELL function will return value of –1 if a cell does not exist in any cell library. If a cell exists, but is not currently loaded, the CELL function will return 0. If the cell is already loaded, the positive, non-zero index of the cell in the cell table is returned.

Example:

```
#MY_CELL_INDEX = {CELL("%MY_CELL")}
IF (%MY_CELL_INDEX == -1) {…         ! cell doesn't exist and is not loaded
ELSEIF (%MY_CELL_INDEX == 0) {… ! cell exists and is not loaded
ELSE …                                ! cell exists and is in the cell table
```

The statement:

#MY_CELL_INDEX = {CELL("%MY_CELL")}

will store the cell index of the cell with the cell name typed by the user. Various blocks of commands not shown in the example will be executed depending on the value returned from the CELL function.

The "{}" around the CELL function call are required to force the parser to realize that the function call is an expression requiring evaluation. The quotes around the argument string are strongly recommended. If the CELL function is called with an argument of an unquoted string containing invalid characters, the parser may fail to interpret the function call correctly. In this case, ICED™ will post an error message and the remainder of the command file will remain unprocessed. However, when the string is quoted, even a null string ("") or a string containing invalid characters (e.g. ',' or '*') will not cause a syntax failure.

Determining if a Cell is Open or Protected

Consider again the swap example on page 102. If the cell is already open from another edit command, the EDIT command will fail. If the cell is stored in a protected library, the command file will pause when executing the EDIT command and display a warning prompt on the bottom of the screen.

For these reasons, when you use an edit command in a command file to modify and then save a cell you first need to verify the edit status of the cell. This status tells you if the cell is already open, and what type of protection is defined for the library where the cell file is stored.

Remember that there are three types protection for cell libraries:

Cell libraries are defined by the ICED_PATH environment variable.

	Direct-edit	Cells in direct-edit libraries can be modified and then saved in the same directory.
PROTECTED	**Copy-edit**	Modified cell files for cells in copy-edit libraries can only be saved to the current directory. The original cell files remain unchanged. This allows you to use your copy of the cell, while allowing other users to use the original version.
	Read-only	Cells in a read-only library cannot be modified and then saved. You can edit them only in "VIEW ONLY" mode.

Figure 17: Cell Library Types

In a command file, you normally want to edit only cells located in direct-edit libraries. These are cells you are allowed to modify without warning. For example, if you share a library of cells with other users, you usually don't want a command file to inadvertently modify some of these cells and make local copies in your current directory unless you are made aware that this has happened.

Unless you add special keywords to an edit command, when it tries to open a cell in a protected library, the program will prompt the user with a warning message and the user must reply to proceed. This is not a good command file practice.

There are three system macros that contain the edit status of a cell. All three of these system macros require the cell table index to specify the cell. If all you have is a cell name, you must obtain the index with the CELL function described on the previous page. If the cell is not loaded, you will need to load it first with an edit command.

System Macro Name	Use	Page
CELL.EDIT.*cell_index*	For any valid *cell_index*, this system macro will contain one of the values shown in the table in Figure 19.	261
SUBCELL.EDIT.*cell_index*	This system macro contains the same values as CELL.EDIT *cell_index* except that any cell that is not a subcell of the current cell will have a '0' stored for it. Also, using this system macro is more efficient when you will be editing many cells in a loop.	296
CELL.LIB.TYPE.*cell_index*	The only difference between this system macro and CELL.EDIT.*cell_index* is that open cells will not automatically have a 0 stored for them.	263

Figure 18: Edit Status System Macros

The MARK-_SUBCELLS command must be executed to store the values for the SUBCELL-.EDIT macros.

Value	Meaning
0	The cell cannot be edited since it is already open, contains an open cell, or the *cell_index* does not refer to a valid entry in the cell table.
1	The cell is in a view-only library that cannot be edited then saved.
2	The cell is in a copy-edit library so that if you edit and then save it, it will be saved to the current directory rather than its original library.
3	The cell is directly editable.

Figure 19: Possible values for CELL.EDIT.*cell_index* macro

We next expand the swap example to test the edit status of the cell name entered by the user. First, the cell index of the cell is determined with the CELL function. This insures that the cell name does not refer to a non-existent cell. Next, the CELL.EDIT.*cell_index* system macro is used to determine the edit status of the cell. Only when the edit status is equal to '3' will the command file edit the cell. This insures the cell is not in a protected library.

Example:

```
LOCAL #MY_CELL $PROMPT "Type subcell name for swap:"
LOCAL #MY_CELL_INDEX = ""
LOCAL #NUM_SWAPPED = 0

#MY_CELL_INDEX = {CELL("%MY_CELL")}
IF (%MY_CELL_INDEX > 0) {
    IF (%CELL.EDIT.%MY_CELL_INDEX==3){
        EDIT CELL %MY_CELL
                UNSEL ALL
                SEL LAY M1+M2 ALL
                XSELECT OFF
                SWAP LAY M1 AND M2
                #NUM_SWAPPED = %N.SELECT
        LEAVE
        RETURN %NUM_SWAPPED Components swapped
    }
    ELSE ERROR Cell %MY_CELL is not in a direct edit library, &
        or is already open.
}
ELSE ERROR Cell %MY_CELL is not loaded in this session
```

The two lines:

```
IF (%MY_CELL_INDEX > 0) {
     IF (%CELL.EDIT.%MY_CELL_INDEX==3){
```

must be in separate IF commands. If both conditions were included in a single IF statement as in the following statement:

```
IF ((%MY_CELL_INDEX > 0) && (%CELL.EDIT.%MY_CELL_INDEX==3))
```

and MY_CELL_INDEX was equal to –1, the second condition would be parsed even though the first condition is FALSE. The parser would try to evaluate:

```
(%CELL.EDIT.-1==3)
```

and the statement would fail with a syntax error.

When you execute a command file like the one above, you want to first exit from any nested edits you may have opened in the editor session. Open cells cannot be modified by this command file.

Looping Through All Subcells

The cells marked by MARK-_SUBCELLS can be restricted to those with components on certain layers.

It is more common to edit cells in a loop than to have the user specify a cell name explicitly. In this case, you can loop through every cell in the cell table.

If you want to restrict loop processing to subcells of the current cell, use the SUBCELL.EDIT.*cell_index* system macro to test the edit status. Only subcells of the current cell when MARK_SUBCELLS is executed will have non-zero values.

Example:

MAX.CELL is a system macro containing the last valid index into the cell table. See page 286.

See details on the CELL.NAME.*n* system macro on page 264.

```
LOCAL #N = 1;                           ! define counter macro

MARK_SUBCELLS                           ! initialize subcell.edit.n macros
WHILE(%N <= %MAX.CELL){                 ! loop through each cell
        IF(%SUBCELL.EDIT.%N==3){        ! get edit status
                EDIT CELL %CELL.NAME.%N;
                !cell processing commands go here
                LEAVE;
        }
        #N = {%N + 1};                  ! increment counter
}
```

See page 22 for other methods of executing commands in many cells.

Allowing Local Copies of Cells

You can use the LOCAL_COPY=TRUE option on an edit command. This will avoid a warning prompt when the indicated cell already exists in a copy-edit library. Instead, the modified cell will be saved in the current directory rather than overwriting the cell file in the protected library. Use this option only when you know the cell is located in a copy-edit library. If the cell is located in a direct-edit library or a read-only library, the edit command will fail.

Example:

```
ELSEIF (%SUBCELL.EDIT.%N==2){
        EDIT CELL %CELL.NAME.%N LOCAL_COPY=YES
        !cell processing commands go here
        LEAVE
}
```

If the lines above were added to the command file on the previous page just before the line that increments the counter, the command file would be able to perform the processing for cells located in copy-edit libraries as well. The modified cell files would be saved in the same directory as the root cell file.

Creating New Cells with an EDIT Command

If your command file allows the user to create a new cell by typing in the name of the cell, you can use the VALID_CELL_NAME function to test that the string is valid to name a cell. This will prevent an edit command from failing with a syntax error about the invalid name. It is also a good idea to use the CELL function described above to test that the cell does not already exist.

Example:

```
DEFAULT LOCAL #NEW_CELL $PROMPT "Type new cell name"

IF (CELL("%NEW_CELL") != -1){
         ERROR Cell %NEW_CELL already exists
}
IF (VALID_CELL_NAME("%NEW_CELL")) {
        EDIT CELL %NEW_CELL
                ADD CELL TEMPLATE AT 0,0
                SEL CELL TEMPLATE
                UNGROUP
        LEAVE
        ADD CELL %NEW_CELL
}
ELSE  ERROR "%NEW_CELL is not a valid cell name"
```

This command file fragment prompts the user for a new cell name. The user's response is tested to insure that a cell with that name is not already used in the design, and that it is a valid string to name a cell. The new cell has template shapes (perhaps standard cell bus wires) added to it. Then the cell is added to the current cell at coordinates digitized by the user when the ADD CELL %NEW_CELL command is executed.

The new cell file (as with any saved cell file) will include the cell environment properties (e.g. layer names) of the current cell.

Saving and Restoring Settings

You will often change the selection status of components or use commands to alter editor settings during your command files. It is a good practice to restore these properties to their original condition before the end of your command file.

Saving the Selection Status of Components

When the user has components selected, they are often selected for a reason. The selection status of components may be the result of a complicated series of SELECT and UNSELECT commands that the user does not want to repeat. If the user then executes your command file and it has unselected everything, he may be a little upset. It is a very good command file practice to save and restore the selection status of components in command files that need to modify these selections.

The SELECT command has a stack feature for saving and restoring the selection status of components. You can "push" the selection status of all components onto the stack, then "pop" it back off later. Only one set of selection information can be saved on this stack. Pushing another set of selections onto the stack will replace a set already there. Popping twice in a row will only reselect the single set of components saved on the stack.

SELECT PUSH	Push selection information onto stack.
UNSELECT PUSH	Push selection information onto stack, then unselect all components.
SELECT POP	Select components that were selected when push was performed. Other components that are already selected remain selected.
UNSELECT POP	Unselect all components, then select components that were selected when push was performed.
SELECT EXCHANGE	Push selection information onto stack; unselect all components; then select the components that were pushed in a previous command.
SELECT FAIL	Push selection information onto stack, then select only components that caused a previous command to fail.

Figure 20: Select stack commands

Example:

```
UNSELECT PUSH
SELECT LAYER SCRATCH ALL
XSELECT OFF
DELETE
XSELECT ON
SELECT POP
```

This fragment of the CLR_SCR.CMD command file (from page 84) demonstrates how to use the SELECT stack commands to save and retrieve the selection status of components. It is especially important to remember to unselect all components when you perform a DELETE in the command file. Otherwise, components that are selected before the command file begins will disappear as well.

When your command file may be used on large flat designs, and speed is an issue, the UNSELECT PUSH command can be relatively expensive in time. Do not add it if it is not required. UNSELECT ALL is much faster than UNSELECT PUSH. The best method you can use is to perform the push only if components are selected.

Popping information from the stack before it has been pushed will result in an unpredictable set of selections. You should be careful to avoid popping when a push may or may not have been performed.

Example:

```
IF(%N.SELECT!=0){
  UNSELECT PUSH;
}
.
.         ! Missing commands that select and manipulate components
.
UNSELECT POP        ! May have unpredictable results
```

This example always executes the UNSELECT POP, but conditionally executes the UNSELECT PUSH. If no components were selected when the command file is executed, the UNSELECT POP may restore selections from a push performed earlier, even in a previous edit session.

Example:

```
LOCAL #POP.FLAG=0

IF(%N.SELECT!=0){
  UNSELECT PUSH
  #POP.FLAG=1;
}
.
.         ! Missing commands that select and manipulate components
.
IF(%POP.FLAG!=0) UNSELECT POP
ELSE UNSELECT ALL
```

This command file fragment uses a much better way to save and restore selections if components are selected when the command file executes, while still avoiding the expensive UNSELECT PUSH command when it is not required. Since the flag macro is tested, the UNSELECT POP will be executed only when UNSELECT PUSH was performed.

Saving the Editor Settings

Other settings may need to be saved and restored by your command file. The view window, the resolution and snap settings, the default layer, and layer properties are all examples of settings that should have their initial values saved and restored if your command file needs to alter them temporarily.

There are two methods to save and restore editor settings:

- Save the value of a setting in a local macro using the appropriate system macro, then use the correct command to restore the value.
- Use the TEMPLATE command to save all system settings in a command file, then restore the settings by executing the command file.

Category	**Setting**	**System macro name(s)** ***.xxx* means several macros**	**Page**	**Command used to restore value**
Coordinate resolution	Resolution step size	RES.STEP	291	RESOLUTION
	Resolution mode	RES.MODE	290	RESOLUTION
	Snap grid settings	SNAP.*xxx*	293+	SNAP
View window settings	Current view window	VIEW.BOX	301	VIEW
	Center of current view window	VIEW.CENTER	302	VIEW
	Current menu file	MENU	287	MENU
Layer settings	Blank status of components in nested cells	LAYER.BLANKED.CELL-.*layer_spec* and BLANKED.CELL.LAYERS	274 257	[UN]BLANK
	Blank status of components in current cell	LAYER.BLANKED.ROOT-.*layer_spec* and BLANKED.ROOT.LAYERS	274 257	[UN]BLANK
	Protection status of layer	LAYER.PROTECTED-.*layer_spec*	279	[UN]PROTECT
	Various layer properties	LAYER.*xxx*	275+	LAYER
	Spacing cursor properties	SPACER.*xxx*	294+	SPACER
	Default layer	USE.LAYER	299	USE
	Other defaults used in ADD commands	USE.*xxx*	299+	USE

Figure 21: System macros to save editor settings.

Example:

```
LOCAL #ORIG_SNAP_ANGLE = %SNAP.ANGLE
SNAP ANGLE = 90
.
.        ! Missing commands that depend on 90º snap angle
.
SNAP ANGLE = %ORIG_SNAP_ANGLE
```

The system macro SNAP.ANGLE contains the value of the current snap angle. The command file fragment above saves this value in the local macro ORIG_SNAP_ANGLE. This macro is then used to restore the snap angle at the end of command file using the SNAP command.

You may want to restore the original view window if your command file modifies the view window explicitly or anytime the user is allowed to digitize coordinates or select components. The view window may be changed by the user during any user interaction command.

Example:

```
LOCAL #ORIG_VIEW = %VIEW.BOX

IF (%N.SELECT == 0){
        $Select component(s) for operation
        SELECT NEAR
}
! Missing statements that manipulate the selected component(s)
VIEW BOX = %ORIG_VIEW          !restore view setting
```

During the command file indicated above, the user may alter the view window with nested view commands (selected with the menu after pressing <Esc> during the SELECT NEAR command). The view window will not be restored automatically to its original state unless the VIEW BOX command is executed as shown above. Of course, you may prefer to leave the view window displaying the new widow chosen by the user. If this is the case, omit the VIEW BOX command.

As of this writing, some settings cannot be saved and restored with the macro method described above since no system macros save the settings' values. However, the TEMPLATE command can be used to save and restore just about all editor settings.

Example:

```
LOCAL #ORIG_TEMPLATE = %TMP^TMPLSAVE.CMD
TEMPLATE %ORIG_TEMPLATE
.         !Missing commands that alter editor settings
@%ORIG_TEMPLATE                      !restore editor settings
```

The command file fragments above demonstrate how to use the TEMPLATE command to save and restore editor settings. First a command file name is defined in the ORIG_TEMPLATE macro. The system macro TMP is used to define the directory path of the temporary file. Then the TEMPLATE command is used to create this file with commands that restore editor settings. The @%ORIG_TEMPLATE command at the end of the example executes the file created by the TEMPLATE command.

The settings saved by the TEMPLATE command include color definitions, grids, layer properties, keyboard macros and many other settings. You can execute the TEMPLATE * command to display the entire list on the screen.

The following settings are **not** saved by the TEMPLATE command:

view window,

selection status of components, and

default layer.

Saving Macros for Future Sessions

The only user-defined macros saved in a cell file are keyboard macros. Other macros are removed automatically at the end of an editor session. If you want to save other macros you must use a SHOW command to export the macros to a command file. You can then execute this command file in a future editor session to restore the macros. When you automate this process, you have the equivalent of your own system macros available to any edit session.

See more on EXIT.ROOT on page 20.

When you exit ICED™ using an EXIT command (or a LEAVE command that results in saving the root cell), the editor first checks to see if the macro EXIT.ROOT exists. If it does, ICED™ will execute the command string stored in the macro before terminating. If EXIT.ROOT contains a command to save macros in a file, then this file can be executed in a later session to restore the macros.

When using option 4b, the EXIT.ROOT macro does not need to be defined in the startup command file. See the example at the end of this section.

1) Create the global macro EXIT.ROOT as shown below in your startup command file. You can optionally create other global macros useful to your command files with default values.
2) Have your command files create, use, and update global macros as required.
3) When EXIT.ROOT is defined with a SHOW command (as shown below), it will automatically save a set of global macro definitions in a command file every time you exit the editor.

4a) Create an ALWAYS=*command_file* option on your layout editor command line to execute the command file created by step 3 every time you open the layout editor. (See example below.)

or

4b) Execute the command file created by step 3 in each of your command files when a necessary macro does not already exist.

The MACRO-_EXISTS function can be used to test if a macro has been defined.

Example:

```
GLOBAL #EXIT.ROOT="SHOW USER MY.*         &
FILE=%HOME.1^SETTINGS\MY.MAC";            &
SHOW USER EXIT.ROOT APPEND"
```

The system macro HOME.1 stores the name of base installation directory. The '^' delimits the macro reference without creating a space in the file name string.

When this macro definition is executed, it creates a macro EXIT.ROOT. A definition like this is usually found in a startup command file so that it is created automatically for all new cells. Once the definition is made, when the editor closes and saves the cell file, another file is saved automatically in the SETTINGS subdirectory with the name MY.MAC. This file contains macro definition statements for all global macros with names beginning with the string "MY.". In addition, the definition of the EXIT.ROOT macro is added to the end of the MY.MAC file by the second SHOW command.

Suppose that one or more of your command files needs to refer to and update a text label suffix. Your startup command file might contain the following macro definition in addition to the one above.

Example:

```
GLOBAL #MY.NET_SFX = "0"
```

Then you might have a command file to add and label wires that is similar to the following:

Example:

```
!ADDBUSWIRE.CMD
!This line allows the command file to work everywhere
DEFAULT GLOBAL #MY.NET.SFX = $PROMPT "Initial net suffix"
LOCAL #NETNAME = BUS_%MY.NET.SFX
ADD WIRE
ADD TEXT "%NETNAME" AT %LAST.POS
#MY.NET.SFX = {%MY.NET.SFX + 1}
```

The system macro LAST.POS stores the coordinate pair for the last point digitized with the mouse.

The macro MY.NET.SFX is defined with the DEFAULT keyword so that the command file will be successful even for users who do not have the macro pre-defined. Since you do have MY.NET.SFX pre-defined, the first time you execute the command file, the label "BUS_0" will be added. The second time will add the label "BUS_1". You can then exit the editor and the macro definition "GLOBAL MY.NET.SFX="2" will be stored in the file MY.MAC.

Now you change your project batch file to use the following option on the ICED.EXE command line:

Example: **ALWAYS=Q:\ICWIN[7]\SETTINGS\MY.MAC**

The next time you open the editor, the MY.MAC command file will be executed automatically, defining the MY.NET.SFX macro in the current editor session. When you execute the ADDBUSWIRE.CMD command file again, the label "BUS_2" will be added as though you were continuing the previous edit session.

Since EXIT.ROOT was also redefined by MY.MAC, when you exit this edit session, the file MY.MAC will be stored again with the current value of MY.NET.SFX.

Alternately, if you want to avoid using the ALWAYS option on the command line, you can have each of your command files execute MY.MAC when a required macro does not already exist. In this case, EXIT.ROOT is not defined in every session, only in sessions where a macro was used and perhaps updated.

[7] Q:\ICWIN represents the drive letter and path of your first ICED™ home directory, usually C:\ICWIN.

To alter this example for this method, replace the first statement in ADDBUSWIRE.CMD with the following:

Example:

```
IF (MACRO_EXISTS(#MY.NET.SFX) ==0){
        IF (FILE_EXISTS("%HOME.1^SETTINGS\MY.MAC")){
                @%HOME.1^SETTINGS\MY.MAC
        }
        ELSE GLOBAL #MY.NET.SFX = $PROMPT "Initial net suffix"
}
```

In this case, the EXIT.ROOT command does not really need to be defined in the startup command file. It can be defined in the original copy of MY.MAC.

See the _EXIT.ICED.CMD command file supplied with the installation for a more elaborate example of saving macros for future sessions. It is used in combination with the DRCNOW.CMD file.

Calling Other Programs

A command file can call another program by shelling out to DOS and executing any command string that is valid at the DOS prompt. This simple feature can result in some very powerful command files.

Use the DOS command to temporarily shell to a DOS console, perform an operation, and then return to the next command in the command file when the operation is complete. Use the SPAWN command instead to allow the new operation to continue while control immediately proceeds to the next command in the command file. This procedure can be completely transparent to the user of your command file, or they can interact with the spawned program if the program supports this (e.g. the text editor NOTEPAD.EXE, supplied with all Windows installations).

You can even import or export component data to or from another program. We describe how to use command files of ADD commands as an import/export file format later on page 122.

The ICED_HOME directories (e.g. Q:\ICWIN) are added to the end of the environment variable PATH for the duration of the DOS or SPAWN commands.

To execute a program (or a batch file) from a command file, the operating system needs to be able to find the executable file. You can explicitly specify the path to the program in the DOS or SPAWN command, or allow the operating system to search for the file using the directories stored in the PATH environment variable.

If you want to distribute your command file to other users, you must also distribute any executable programs it calls (unless they are commonly available system utilities or utilities supplied with the ICED™ installation.) The executable file must be located where the operating system can find it, or you must specify it's location in your command file when you call the program. This can make portability on various users' computers an issue.

The best method in this situation is to store the executable file in the same place as the command file. Then use the EXEC.DIR system macro (see page 267) to specify the location of the executable file. EXEC.DIR will always contain the directory path of the currently executing command file.

Shelling Out to a GUI Program

Refer to the DOS and SPAWN commands in the IC Layout Editor Reference Manual.

The following command in your command file will suspend the command file and the layout editor and let the user edit an ASCII file for as long as necessary using the NOTEPAD editor supplied with all Windows installations. When the user closes the NOTEPAD window, control returns to the command file.

Example: **DOS –NOTEPAD.EXE MYFILE.TXT**

To allow the NOTEPAD window to remain open while the command file continues, use the SPAWN command instead.

Example: **SPAWN –NOTEPAD.EXE MYFILE.TXT**

A GUI window will usually remain open until the user closes it.

When the '-' is used as a prefix in the executable command string, it prevents the creation of a temporary console window unnecessary to a GUI (Graphical User Interface) application.

Shelling Out to a DOS Command

You can perform DOS commands such as COPY (to copy files), MD (to create a directory), or CD (to change a directory) using the DOS command

Example:

DOS "^COPY %MY.FILE %NEW.FILE >NUL"

The quotes are recommended to prevent the parser from misinterpreting the command string, but they are rarely necessary.

The command above will execute the DOS COPY command in a temporary console window. The command executes so quickly that the console window will flicker on then off too quickly to see any information reported by the COPY command. So the '^' prefix is added to the front of the command string to prevent the display of the temporary window.

The '>' redirection character redirects the output of the command so it is not reported to the console window. Redirecting to the NUL device discards any output to the console window. This prevents unwanted behavior in some operating systems that can hang when console output is not redirected. Always redirect the output of DOS commands to prevent problems.

One limitation of this command is that any information reported by the COPY command will not be visible to the user. If the file name stored in MY.FILE did not exist, then the COPY command would do nothing, but the command file would continue anyway with no indication to the user.

You can capture the console output of a DOS command and redirect it to a file with the '>' redirection character.

Example:

DOS "^COPY %MY.FILE %NEW.FILE > %TMP^DOSLOG.TXT"

Learn more about the TMP system macro on page 299.

This command redirects the output of the command (usually the string "1 file(s) copied") to the file DOSLOG.TXT. This redirection allows you to view the results of the COPY command if your command file does not perform as expected. The DOSLOG.TXT file will be stored in the directory defined in the system macro TMP.

It is best to verify that the operation succeeded in your command file. If the operation failed, you can display the log file on the screen rather than allow the command file to continue. This is shown in the following example.

Example:
```
DOS "^COPY %MY.FILE %NEW.FILE > %TMP^DOSLOG.TXT "
IF (FILE_EXISTS(%NEW.FILE) == 0) {
        SPAWN -NOTEPAD.EXE %TMP^DOSLOG.TXT
        ERROR "See log file. File %MY.FILE not copied to %NEW.FILE"
}
```

Using Other Programs to Manipulate Component Data

When you want to import or export component data to/from another program in your command file, ICED™ supports a simple mechanism to do this. The SHOW command can be used to create a command file of ADD commands for all currently selected components. This command file can then be used to import the data into a program that expects this format. If the program creates a file of component data the form of ADD commands, this file can be executed to create the components in the layout editor.

The ED.CMD (page 312) and BUSROUTE.CMD (page 327) command files both use this method. Both command files use the following steps:

- The SHOW command builds a file of component information in the form of ADD commands.
- The name of this file is then passed as an argument to a program called with the DOS command.
- The program manipulates the data in this file and either modifies the original file or creates a new file filled with ADD commands.
- After the DOS program is complete, the new or modified file is then executed as a nested command file to add new components.

The ED.CMD example uses the NOTEPAD.EXE to allow the user to modify selected components by editing the ADD commands directly with the text editor. ED.CMD implements a very nice method to undo the results by saving the original command file as well as the modified version and storing the id numbers of the new components. The UNED.CMD file can then delete the new components with these id numbers and then re-create the old components with the original command file.

The BUSROUTE.CMD example is more complex. It routes a bus through a path digitized by the user. After prompting the user for several parameters, it prompts the user to digitize the path of the bus with a single ADD WIRE command. The coordinates of this wire are then exported to a file with the SHOW command. This file is passed as an argument, along with other parameters, to a C program written specifically for this application, BUSROUTE.EXE. This program calculates the coordinates of each wire in the bus and then exports these new wires as ADD commands in a new command file. BUSROUTE.CMD then adds these new wires by executing the new file.

The C code for BUSROUTE.EXE, including subroutines for parsing the output of the SHOW command, is provided in full in the description of this example. You may want to use these subroutines if you need to create a similar application.

NOTE: You must use caution whenever you use the SHOW command to export data for more than the current cell. Carefully read the important information about the NOLIBS keyword in the SHOW command description in the IC Layout Editor Reference Manual to avoid corrupting nested cells when exporting data from protected libraries.

Testing, Error Checking, and Recovery from Errors

You can see exactly what commands were executed by a command file by viewing the journal file with the 1:FILE→ edit.JOU menu option. See page 127.

Remember that ICED™ command files are interpreted, not compiled. Each statement is parsed for the first time just before it is executed. If you have written a statement incorrectly, the command may fail when it is executed, and control is returned to the editor with the remaining statements in the command file unprocessed. An error message will be displayed on the view window prompt line.

More puzzling errors occur when the incorrectly written statement does not result in an immediate error, but instead results in an inappropriate value being stored in a macro. The error will only cause a problem later when this value is used in a correctly written command later on in the command file, so it can be difficult to spot the problem.

One common source of this type of problem is when the user is prompted for a value. The best way to avoid confusing errors is to validate the value before it is used. See the table on page 88 for a list of methods to validate values.

When you are having trouble debugging a command file that uses local macros, you may want to change them to global macros. This will allow you to see their final values with the SHOW command after the command file is complete.

Example: **SHOW USER**

Typing this command at the command prompt will display all user-defined global macros.

Canceling a Command File

Either of these options will cancel the entire command file and immediately return control to the layout editor.

There are two methods for canceling a command file. Both of these methods can be used only when the command file is paused, waiting for input from the user. If you want to give yourself the option of canceling a command file while you are debugging it, the easiest method is to add a few PAUSE commands. You can then cancel the command file during any of these PAUSE commands.

To cancel the command file while it is waiting for the user to type something at the keyboard,

- press the <Esc> key, or
- press both the left and right mouse buttons.

When digitizing a position, pressing the <Esc> key will bring up the nested view menu.

To cancel the command file while it is waiting for the user to digitize a position,

- press both the left and right mouse buttons.

Error Handlers

Normally, when the command interpreter finds a syntax error in a command file, the error is reported to the screen, and the command file is immediately terminated. You can change this behavior by defining an ERROR.CMD macro.

When ERROR.CMD is defined, the string stored in it is executed when an error is encountered. The example on page 165 opens up a notepad window displaying the journal file when an error occurs. You may want to perform other actions when a command fails. You can even execute a command file when a syntax error occurs as shown in the next example.

Example: **LOCAL #ERROR.CMD=@ERR_HANDLER.CMD**

Mysterious Errors

Some command file errors can be difficult to find. Here are a few common mistakes that you should consider when searching for command file errors.

Carefully review each macro reference for the % prefix when you have a mysterious error.

When a macro reference is made without the % prefix, the name of the macro is used rather than the value. This can lead to many perplexing errors. Consider the following command file fragment.

Example:

```
LOCAL #MY_CELL  $PROMPT "Type subcell name:"
#MY_CELL_INDEX = {CELL("MY_CELL")}         !Oops, forgot the %
IF (%MY_CELL_INDEX != -1) {…
```

The second line is a function call that is supposed to use the cell name typed by the user to determine the cell table index of the cell with that name. However "MY_CELL" was used instead of the macro reference "%MY_CELL". This means that the CELL function will always return –1, and the statements controlled by the IF command will never be executed, unless you happen to have a cell with the name "MY_CELL".

Another common error is to forget the {} when you type an expression.

Example:

```
#COUNTER = %COUNTER + 1        !Incorrect syntax
```

If this statement is part of a loop that is executed repeatedly, and the initial value of the COUNTER macro is 0, the first time through the loop the value of the COUNTER macro would be:

"0 + 1"

The second time through the loop, the value will be the following string:

"0 + 1 + 1"

If you want the expression to be evaluated as a mathematical expression, you must write it as:

Example:

```
#COUNTER = {%COUNTER + 1}      !Correct syntax
```

When the statement on the previous page is executed, the mathematical expression will be evaluated, and the COUNTER macro will be set to "1" the first time through the loop. It will be set to "2" the second time through the loop, etc.

The best way to solve this type of mystery, if you are not yet familiar enough with command files to spot the missing {}, is to look in the journal file to see what values are being stored in the macros.

The Journal File

By default, most every editor command and macro assignment executed is logged to the journal file. Commands like IF or WHILE are not logged; only commands that add or modify components or editor settings. The intent is to be able to re-execute every command that altered the cell or the editor environment if the journal file is executed as a command file.

To learn more about using a journal file for cell file recovery, see page 129.

This enables the journal file to be used as a recovery mechanism if a command file has corrupted the cell. The file can also be browsed to determine exactly what happened when a command file did not do what you expected.

The LOG command is completely described in this manual on page 190.

If you prefer that commands in a command file are not logged to the journal file, add the LOG OFF command **as the first line of the command file**. This may result in a substantial speed improvement for large command files or command files that execute loops.

However, when a command file with user interactions (e.g. one that prompts the user to enter a value or select a component) is executed in the log off mode, the user's responses are not recorded in the journal file. Only the call to the command file is recorded and will be executed by the journal file. This means that work done cannot be recovered automatically by executing the journal file. Add the LOG OFF command to only those command files without user interaction when speed is a concern.

If your command file contains user interactions, we recommend that you use the LOG SCREEN OFF instead of LOG OFF. This prevents commands from being logged to the status line of the display, but does not affect the journal file.

During debugging, it can be helpful to add LOG LEVEL=DEBUG to the beginning of your command file. This will add extra information comments to the journal file. Commands like IF and WHILE will be logged with comments. This information will not aid in recovery, but can be helpful when using the journal file to determine why a command file did not do what you expected.

One problem with journaling during command files is that the journal file is not updated immediately after every command in a command file. ICED™ will buffer the information to be copied to the journal file. The results of a command file will be written to the journal file only after the buffer is full. (This is much faster than opening and closing the file for each command.) If you want to browse the journal file immediately after your command file is completed, and you do not see the results of your command file, execute a command in the editor to force the buffer to get dumped to the journal file, then browse it again.

UNDOing Command Files

See the IC Layout Editor Reference Manual to learn more about journal files and recovery.

When a command file is canceled, terminates with an error, or just completes its task incorrectly, **the UNDO command will not undo any commands in the command file**. The UNDO command is designed to undo only the last edit command. Since it cannot reverse the effects of all of the commands in a command file, the UNDO command is disabled at the end of a command file rather than allowing it to undo only the last edit command in the file.

Unless the command file has a special mechanism for undoing the results, the only way to undo the results of a command file is to use the JOURNAL command and edit the journal file before using it for recovery. This will restore the cell to what it was before the command file was executed.

Using the Journal File for Recovery

The procedure below can be used to restore a cell to what it was before a command file was executed.

1) Execute the JOURNAL command to terminate the editor and avoid saving any cell files.

2) Edit the journal file to remove all commands from the bottom of the file up to the line that called the command file that did the damage.

3) Finally, relaunch the editor to edit the same cell. The editor will automatically give you the option to execute the journal file and recover the work done by all of the commands remaining in the journal file.

Every time a command file is executed, a comment with the name of the command file is inserted in the journal file. This will allow you find the point in the journal file where the commands in the command file began.

Using UNED.CMD to Undo Command File

A much simpler way of "undoing" the results of a command file is to add a recovery mechanism to the file. One method is to keep track of the component id numbers assigned to components added by the command file. The new components can then be deleted by selecting them by these id numbers.

The ED.CMD and UNED.CMD command files use this method. Once you understand how these examples work, you can use the same method of keeping track of the id numbers in your command file. Then the user can undo the results of your command file by executing @UNED. See page 318 for descriptions of ED.CMD and UNED.CMD

Control File Efficiency

See page 189 for the efficiency of the LIST command when processing many components.

When your command file takes a long time to execute, most of the time ICED™ takes to execute the commands may be spent updating the display or logging commands. You can speed up your command file considerably by controlling how ICED™ performs these tasks.

Disable View Window Update

The VIEW command is completely described beginning on page 210.

The most speed improvement is achieved by preventing the view window from being updated after every command. For this reason, the default behavior during a command file is to avoid updating the view window, except during commands requiring user interaction. However, if your command file contains a VIEW ON command, this will slow down the execution considerably.

The usual reason for adding a VIEW ON command to your command file is to allow the user to see visible evidence that the command file is still working. However, after the command file is debugged, you should remove this command and allow the command file to execute more quickly. If you want the user to see progress updates, use $$*comment* commands. (See page 162.)

Disable Command Logging to Screen

The LOG command is completely described beginning on page 190.

By default, as commands are executed, ICED™ echoes each one on the status line of the screen. This is the default behavior because the user may be disconcerted if he sees no activity happening while a command file is executing. You can speed up a slow command file by preventing this logging of commands to the screen with a LOG SCREEN OFF command. However, unless you add some sort of progress indicator for the user, the session may appear "frozen".

$comments are described on page 161.

The LOG SCREEN OFF command makes an exception for the $*comment* command. These comments will result in messages visible to the user. If your command file takes a long time, add several of these $comments at key points in your command file so the user knows that the command file is progressing normally. If you want the comments to display even when the LOG OFF mode is in effect (see next section) use a "$$" before the comment.

Example:

```
LOG SCREEN OFF
$$ Initializing arrays.  Please wait.
..        !time consuming commands
$$ Computing offsets….
..        !time consuming commands
$$ Adding components….
..        !time consuming commands
$$ Command file completed successfully.
```

If your command file has a particularly lengthy loop, you may want to add $$comments to the loop so progress is easy to follow.

Disable Command Logging to Journal File

A LOG OFF command on the same line as the *@file_name* command is considered the first command in the file.

See more details on the journal file on page 128.

In addition to the echo of each command to the screen, the command is logged into the journal file. This command logging can take a significant amount of time in a long command file. You can turn off this logging with the LOG OFF command as the **first** command in the command file.

Since the journal file is the only mechanism to recover work done to a cell in the event of a system crash or power interruption, you must be careful about using this command in your command files. You can prevent the automatic recovery of data. To avoid this result, you should never use the LOG OFF command in the following types of command files. When you do, the values stored in macros have not been preserved in the journal file. Automatic recovery of work using the journal file will be prevented.

- Any command file that prompts the user for input
- A command file that is called by passing arguments on the same command line (See an example on page 97.)

We suggest that you restrict using LOG OFF to command files that modify no geometry or those that add many components with no user interaction (e.g. DRC results files.) In these cases, re-executing the entire command file during recovery will be enough to recover any work done by it.

Macro Definition

Overview

Refer to an overview of macros on page 38 for more information on how macro references are parsed by the interpreter.

Macros allow you to store and manipulate values. While macros are really implemented in ICED™ using string substitution, you can use them in a manner similar to variables in other programming languages.

Basically, a macro is an object that has a name and a value. The value is always a string, but these strings can represent text, numbers, coordinate lists, or any other element of an ICED™ command. When you assign a number or a coordinate pair to a macro, it is stored as a string. The value of the macro will be interpreted as a string unless it is later used in a mathematical expression surrounded by "{}" or in a function or a command that expects a number or a coordinate list.

Example:

```
LOCAL  #COORDS = (32.5, -156.3) (39.5, -151.3)
ADD BOX AT %COORDS
```

When the first statement is executed, a macro with the name COORDS is defined. The string "(32.5, -156.3) (39.5, -151.3)" is stored as its value. When the second statement is executed, ICED™ will first perform string substitution to create the command:

```
ADD BOX AT (32.5, -156.3) (39.5, -151.3)
```

Then the ADD command is executed creating the box at the indicated coordinates.

Special character	**Use**
#	Refer to name of macro
%	Substitute value of macro
^	Delimit macro reference without a space

There are several special characters that are used to control how statements using macros are interpreted. The special characters '#', and '%' are the most important. As seen in the examples above, the '#' should be used before a macro name to indicate that you are referring to the name of a macro. The '%' before a macro name means that ICED™ should replace the macro reference with the value of the macro.

It is a good habit to **always precede a macro name with either the '#' or the '%' character**. While the '#' is not required in a macro definition, for almost all other uses, one symbol or the other is required for the program to recognize that you are referring to a macro name rather than typing a string.

One of the most common errors in command files is to forget to use a prefix when typing a macro name, and this can lead to confusing errors.

Example:

```
LOCAL  #COORDS = (32.5, -156.3) (39.5, -151.3)
ADD BOX AT COORDS                         ! oops, forgot the %
```

When the program executes the second statement, the parser will not realize that you referring to a macro. No string substitution will be performed and the command will fail causing the entire command file to fail.

There are two kinds of macros: user-defined and system.

System macros cannot be changed by any user commands. Their values are set and updated by the system. Refer to the table all of the system macros on page 255.

If you do want to save user-defined macros for use in a later session, you can use the SHOW command to save them in a command file. See page 116.

User-defined macros allow you to store values temporarily. They are not stored in the cell file. The data stored in them is discarded at the end of the command file, or when you terminate the layout editor. (Keyboard macros, usually defined with the KEY command, are an exception. They are stored in the cell file. See page 148.)

Any value you need to store in a command file must be assigned to a macro. User-defined macros must be defined before they can be used. We explore many options for macro definition and assignment on the following pages.

You **must** assign an initial value when a macro is defined. If you do not assign a value in the definition statement, the user will be prompted for the value when the macro definition statement is executed.

ITEM Macros

One class of user macros is defined in a different manner than the methods we cover below. The ITEM command will create several user macros filled with information about a single component. See the table on page 175 for a complete list of the macros created for each type of component.

User Macro Definition Syntax

Square brackets "[ ]" are used to indicate optional parameters and keywords. Parentheses "( )" indicate that a choice of keywords is required.

You can use macros outside of command files. When typed in the editor, the shorthand syntax ***#macro_name*= *macro_value*** is enough to define and initialize a macro since the scope is automatically global. However, this syntax is not valid in a command file.

[DEFAULT] (GLOBAL | LOCAL) [#]***macro_name*** [=] [*macro_value*]

macro_value commonly takes one of the four following forms:

["]*string*["]

{*expression*}

$[PROMPT="*string*"] *response_type*

$MENU *menu_name*[:*submenu_name*]

POSITION
BOX
POLYGON
DISP
X_DISP
Y_DISP
NEAR

Figure 22: Valid $*response_type* keywords.

We will cover the use of each parameter or keyword quickly and then cover each in more detail with examples.

The optional **DEFAULT** keyword will cause the program to ignore the entire definition if a macro with the same name already exists.

The GLOBAL and LOCAL keywords specify the scope of the macro. **You must usually include exactly one of these keywords when defining a macro in a command file.** A **GLOBAL** macro can be used outside of the command file that defines it. A **LOCAL** macro can only be used in the command file in which it is defined. LOCAL macros are automatically deleted at the end of the command file in which they are defined.

When you define macros on the same line as a *@file_name* command, their scope is local by default.

The ***macro_name*** can be up to 32 characters long. We will cover all of the restrictions on macro names on page 139. It is a very good idea to include the optional '#' prefix before the macro name. We will cover the reasons for this later.

Now that we have covered the basics of macro definition, we will cover each of the options in more detail.

[DEFAULT]

Learn other ways to pass arguments into command files on page 97.

This optional keyword is used to provide a value for the macro only if it is not already defined. This can be useful when passing arguments into command files, or for making command files more versatile. The entire macro definition will be ignored if the macro already exists.

This next example uses one of the $*response_type* options to prompt the user for a string. We will cover these later, but in this example we use this option to demonstrate that the entire macro definition, including the prompt, is ignored when the DEFAULT keyword is used.

Assume that a macro with the name NET_BASE has already been defined in the current layout editor session with a statement similar to:

Example: **GLOBAL #NET_BASE = "BUS"**

Now in a separate command file, the following statement defines a macro with the same name:

Example: **DEFAULT LOCAL #NET_BASE = $PROMPT="Enter net base name:"**

Since the macro NET_BASE already exists, the user will not be prompted and the existing value for NET_BASE, the string "BUS", will remain the value of the macro.

However, if in a different ICED™ session the first statement is **not** executed, then when the second statement is executed, the user will be prompted to type in the value of the NET_BASE macro.

(GLOBAL | LOCAL)

The only macros that are saved with a cell file are keyboard macros (see the discussion on page 148). All other user-defined macros are deleted automatically at the end of each layout editor session.
See page 116 for a method of saving user-defined macros to be restored in a different session.

Exactly one of these two keywords must be used in each macro definition in a command file. They determine the scope of the macro.

LOCAL macros	will be deleted at the end of the command file in which they are declared
GLOBAL macros	will persist until the end of the edit session or until they are explicitly deleted with the REMOVE command .

Any macro defined with a command typed in the layout editor (i.e. outside of any command file) is global by default. A macro defined on the same line as a *@file_name* command is local by default. You can omit the scope keyword in either of these cases.

Local macros will be removed automatically at end of the command file in which they are declared. They are deleted even when the command file terminates prematurely with an error. You will not be able to see their final values when command file is finished.

You can report the final value of a local macro by ending the command file with a $*comment* or RETURN statement. See examples on pages 98 and 205.

Global macros can be useful if you want to use the values of macros in other command files. They are also useful while debugging a command file, so you can see their final values with the SHOW command. Once your command file is debugged, you can change them to local macros so they do not persist once the command file is completed.

Local macros hide the existence of global macros with the same name in a higher-level command file. If a command file defines a local macro (without the DEFAULT keyword) and a different macro already exists with the same name, the macros are kept separate and will not interfere with each other.

If the command file defines a global macro with the same name as an existing macro, this definition will override the previous definition. The global macro will retain the value assigned in the command file even after that command file is complete.

## [#]*macro_name*

Macros that use the name KEY.*key*, are treated as keyboard macros. See page 148 for more details.

The restrictions on the name of a macro are:

- The name must be a string from 1 to 32 characters long.
- Valid characters include letters, digits, and the special characters: '.', '_', and '$'. (The $ character should usually be avoided since it is used as a special character in certain cases.)
- The first character may not be a digit.
- Macro names are case-independent.

It is good practice to include the '#' prefix before the macro name in a macro definition. This character makes it obvious that you are referring to a macro name. If you neglect to add the '#', mistakes in typing the macro definition may cause the parser to interpret the command differently than you intended.

Only global macros can be displayed with the SHOW command after a command file is complete.

It is a good idea to name all user-defined macros with a prefix similar to the name of the command file. This makes it easy to remove all macros if necessary. It also makes the command file easier to read since system macro names will be obviously different. Also, if all user-defined macros begin with the same string, it is easy to use the SHOW command to display the values of all your macros when the command file does not perform as expected.

Example: **SHOW USER=MY_CMD***

If all of your macros have been defined with names that begin with the string "MY_CMD", the SHOW command above can be used after the command file or in an error handler to see what their final values were.

Example: **REMOVE MY_CMD***

This single command can be used to delete all of the macros defined in your command file if you have named all of those macros with "MY_CMD" as a prefix.

macro_value

All of these methods can also be used in any macro assignment statement after the macro is already defined.

The ***macro_value*** can be specified in any of the five following ways:

["]*string*["]	You can explicitly type the value of the macro as a number, coordinate pair or string. The quotes are usually not required. More on this below.
{*expression*}	A mathematical expression can be typed. You must surround the expression with curly braces {} if you want ICED™ to evaluate the expression and store the value, rather than storing the expression as a string. See page 143.
$*response_type*	Use one of the keywords in the table on page 143 to prompt the user to define the value with the cursor or keyboard.
$MENU	The user can select a choice from a menu using this option. See page 147.

The fifth method is to omit the *macro_value* entirely. If you do not use one of the other methods, the user will be prompted to supply the value with the following often-cryptic message:

"Enter value for *macro_name*"

The first two methods are just different ways of specifying a simple value. The last three methods require user input at the time the command file is executed.

The maximum length of the string stored in a macro is 7900 characters. This is slightly less than the maximum command line length of 8000 characters.

["]*string*["]

Use this syntax when assigning a simple value to your macro. The string can represent a word, phrase, number, or coordinate pair. In any case, the value is stored as a string. However, that string will be interpreted later as a number or coordinate pair by commands or functions that expect such values.

Example: **LOCAL #MY_MAC = 99**

The statement above defines a macro with the name MY_MAC. The string "99" will be stored as its initial value. This string can be used as a number in other statements. The following command is an example:

```
MOVE X %MY_MAC
```

After macro substitution, this statement will read:

```
MOVE X 99
```

Example: **LOCAL #MY_MAC = (-32.5, 67.3)**

The statement above assigns the string "(-32.5, 67.3)" to the macro MY_MAC. The macro can then be used in any statement that expects a coordinate pair. The following ADD command is a good example:

```
ADD TEXT="MY TEXT" AT %MY_MAC
```

After macro substitution is performed, the command above will be interpreted as:

```
ADD TEXT="MY TEXT" AT (-32.5, 67.3)
```

While many commands do not require the parentheses and comma when entering coordinates, it is a good idea to include them when assigning a coordinate pair to a macro. Some functions that operate on coordinates require the "(*x-coord,y-coord*)" syntax.

Example: **LOCAL #MY_MAC = Four score and seven years ago**

The statement above will assign the string "Four score and seven years ago" to the macro. Quotes are not required even though the string contains blanks.

Quotes are required around the string in two cases:

The string contains a ';', '!', or quote character.

The string ends with a &.

The second restriction is necessary because a '&' at the end of a line means that the statement is continued on the next line.

When quotes are necessary, ICED™ supports four different quote characters. This allows you to include quote characters in a string.

Valid quote characters
"
'
~
`

Example: **LOCAL #MY_MAC = 'ADD TEXT "MY TEXT" AT 0,0'**
%MY_MAC

The pair of lines above will result in the execution of the following command:

```
ADD TEXT "MY TEXT" AT 0,0
```

The quote character used in a quoted string cannot be included in the string. No string can contain all four valid quotes.

You can use the string stored in system macro or a previously defined user macro to assign a value to another macro.

Example: **LOCAL #MY_MAC = %LAST.POS**

See a list of system macros on page 252.

This statement uses the system macro LAST.POS to assign the coordinate string of the last point digitized with the cursor to the macro MY_MAC.

{*expression*}

You can assign to a macro the result of a mathematical expression or a function call by surrounding the expression with curly braces.

Example: **LOCAL #MY_MAC = {SIN(45)/2}** !Correct syntax

See a list of supported mathematical operators on page 50. Page 220 has a list of mathematical functions.

This statement will assign the sine of a 45° angle divided by 2 (0.3535533906) to the macro.

If you forget to surround the expression with curly braces, ICED™ will interpret the expression as a string instead.

Example: **LOCAL #MY_MAC = SIN(45)/2** !Incorrect syntax

This statement will assign the string "SIN(45)/2" to the macro.

The PROMPT command (page 201) is used to temporarily replace the prompt string on the left of the '>' on the command line.

$[PROMPT="*string*"] *response_type*

There are nine different options for prompting the user to supply the value of a macro when a macro definition in a command file is executed. The prompt string is displayed on the command line while the user defines the macro value by typing at the keyboard or selecting positions with the cursor.

Example: **LOCAL #MYMAC $PROMPT="Click with mouse." POSITION**

The example above displays the string "Click with mouse" below the command line and waits for the user to click the left mouse button. The location of the cursor is then stored as the value of the macro.

The first option in the table on the next page represents definitions where no *macro_value* or response type keyword is included. The other options use the keywords indicated to control how the user will define the macro value.

$response_type	**Default prompt**	**User must respond by:**
	Enter value for macro *macro_name*	Typing number, coordinate pair, or string
$PROMPT="*string*"	*string*	Typing number, coordinate pair, or string
$[PROMPT="*string*"]POSITION	Use mouse to enter POSITION.	Clicking mouse once to define a coordinate pair
$[PROMPT="*string*"]BOX	Use mouse to enter BOX.	Clicking mouse twice to define two coordinate pairs (The coordinates are reordered so lower left corner is the first coordinate pair.)
$[PROMPT="*string*"]POLYGON	Redigitize starting vertex to signal end of command.	Clicking mouse to define several coordinates, ending by redigitizing the first one
$[PROMPT="*string*"]DISP	Use mouse to enter DISPLACEMENT	Clicking mouse twice to define pair of numbers representing displacements in x and y directions
$[PROMPT="*string*"]X_DISP	Use mouse to enter X DISP	Clicking mouse twice to define number representing displacement in the x-direction
$[PROMPT="*string*"]Y_DISP	Use mouse to enter Y DISP	Clicking mouse twice to define number representing displacement in the y-direction
$[PROMPT="*string*"]NEAR	Use mouse to enter NEAR	Clicking mouse once to define coordinate pair representing the center of a near box

To allow the user to use the mouse to select from a menu of choices, see page 147.

Each of these options prints a prompt on the prompt line of the layout editor window and waits for the user to supply the information requested. If the user does not supply the information, the entire command file is aborted. (See page 125 to see methods of aborting command files when user input is expected.)

The first row in the table on the previous page is meant to describe a macro definition where no *macro_value* is supplied. In this case the prompt string the user sees is "Enter value for macro *macro_name*", where *macro_name* is the name of the macro.

Example: **LOCAL #BASE_LABEL** !No *macro_value* is defined

This macro definition provides no value or prompt option for the macro. When this statement is executed, the user will see the following prompt in the layout editor window:

Enter value for macro BASE_LABEL

The program will wait until the user hits the <Enter> key. Any text typed by the user before the <Enter> is typed will be stored as the value of the BASE_LABEL macro.

If the user types no text before pressing <Enter>, then the null string "" will be stored as the value of BASE_LABEL.

The $*response_type* methods all have an optional PROMPT="*string*" parameter that allows you to write the prompt string yourself. The prompt *string* **must be enclosed in quotes**. Any one of the four valid quote characters listed on page 142 can be used.

If you use $PROMPT="*string*" by itself, any type of value can be requested. The string typed by the user will be assigned as the value of the macro.

Example: **LOCAL #BASE_LABEL $PROMPT="Type base label:"**

When this macro definition statement is executed, the text "Type base label:" will appear on the prompt line in the layout editor window. The text typed by the user before <Enter> is typed will be stored in the BASE_LABEL macro.

If the user presses <Enter> without typing any text, the null string "" will be stored as the value of the macro. When the macro is used in a later command, this may cause a syntax error. For this reason, it is a very good idea to test the length of the return string in your command file before using the value of the macro.

Example:

```
LOCAL      #DEFAULT_LABEL = "BUS"
LOCAL      #BASE_LABEL                    &
           $PROMPT="Enter base label, [%DEFAULT_LABEL]:"
IF ( LEN("%BASE_LABEL") == 0) {
      #BASE_LABEL = %DEFAULT_LABEL
}
```

The LEN function returns the number of characters in a string. See page 229.

These extra lines will assign a default value to the BASE_LABEL macro if the user presses <Enter> without typing a string. The user is shown what the default value is surrounded by "[]". Remember that macro substitution takes place even in a quoted string. The value for the DEFAULT_LABEL macro will be inserted into the prompt string before it is printed on the screen.

You can request that the user type a number or coordinates instead of a string. However, ICED™ does no automatic verification of the contents of the typed string. The user may type inappropriate data. See page 88 to learn about methods of verifying the value typed by the user.

Displacements can also be defined with the RULER command. See an example on page 273.

Extra processing to insure that coordinates are entered correctly can be avoided if you use one of the following $*response_type* options to prompt the user for coordinate data: POSITION, BOX, POLYGON, DISP, X_DISP, Y_DISP, or NEAR. These options require that the user move the cursor to a location in the layout and then press the left mouse button to digitize the position(s). The digitized position(s) will then be stored as the value of the macro. You can be sure that the coordinates are on the snap grid and that they use the correct syntax. (In the case of the displacement options, the displacement between the positions is stored as the value of the macro.)

The user cannot type the data at the keyboard when these options are used for $*response_type*.

Example:

```
LOCAL      #CENTER = $PROMPT="Digitize center" POSITION
```

When this macro definition is executed, the user will be prompted with the phrase, "Digitize center", and then the program will wait until the left mouse button is pressed to digitize the current location of the cursor. The coordinate pair selected is then stored as the value of the macro.

When you do not specify PROMPT="*string*" for these options, the prompt shown in the table on page 144 will be used.

Example: **LOCAL #CENTER = $POSITION**

When you use this option to prompt the user for the value of the CENTER macro, the user will see the prompt "Use mouse to enter POSITION."

Example: **LOCAL #CENTER = $POS**

Note that like most ICED™ keywords, the POSITION keyword can be abbreviated to a few characters.

$[PROMPT="*string*"] MENU *menu_name*[:*submenu_name*]

See the CHANGES-LAY.TXT file for information on recompiling older menus to be used with this feature.

You can allow the user to select the value of a macro with the mouse from a menu of choices. (This really just a special case of the previous method with the MENU keyword and parameter in the place of the *response_type* keyword.) You can use any of the predefined submenus available in the default menu, M1.

Example: **LOCAL #MYLAYER = $MENU M1:LAYER_NAMES_1**

See an example of selecting a pattern from a menu on page 277.

This example displays the layer selection submenu defined in the M1 menu supplied with the installation. Only layers with names defined for them will be listed. Each layer in the list will be displayed with a rectangle drawn using the color and pattern defined for the layer.

You can browse the source for the M1 menu in the Q:\ICWIN-\SAMPLES-\M1.DAT and M1A.DAT files.

There are many submenus to choose from in the M1 menu. Some of these are filled in at the time of execution, such as the layer menu in the example above. All submenus include all of the relevant options available for a specific command.

Specify the menu name without the .MEN menu file extension. If you want a submenu displayed, add the :*submenu_name* after the menu name without a space. When you omit the submenu name, the first top-level menu is displayed.

If you want a prompt displayed in yellow below the command line to tell the user what they are supposed to do, add the PROMPT keyword and a quoted string to the command.

Example:

```
GLOBAL #MYLAYER= $ PROMPT= "Choose layer for processing." &
                                MENU M1:LAYER_NAMES_1
```

If you create your own menu, it must be defined in ICED™ syntax and compiled with MkMENU.EXE to create the compiled .MEN file from the .DAT source file. Move the compiled menu file to the Q:\ICWIN[8]\AUXIL directory.

Arrays

There is no special storage mechanism for arrays in ICED™. However you can easily create collections of macros that you can use as arrays. Since numbers are valid in macro names, and since you can use macro substitution in macro names, you can create macros that can be referred to in the same manner as arrays. Also, there is practically no limit in the number of dimensions of your arrays.

However, the special characters '[', ']', '(', and ')' are not valid in macro names. You can use the special characters '.' or '_' to separate the base of the array name from the numeric index if you desire.

Example:

```
! Add Spiral of wire components
LOCAL#BASE_COORD = "(0,0)"
LOCAL#NUM_POINTS = 10
LOCAL#COUNTER = 0
LOCAL#COORD_LIST = ""
LOCAL#QUADRANT = 1
LOCAL#XSIGN = ""
LOCAL#YSIGN = ""

        !continued on next page
```

[8] Remember that Q:\ICWIN represents the drive letter and path where you have installed ICED™.

```
WHILE (%COUNTER <= %NUM_POINTS){
        IF ((%QUADRANT == 1) || (%QUADRANT == 2))       #XSIGN = ""
        ELSE                                            #XSIGN = "-"
        IF ((%QUADRANT == 1) || (%QUADRANT == 4))       #YSIGN = ""
        ELSE                                            #YSIGN = "-"

        !Array element definition statement
        LOCAL #COORD%COUNTER =          &
                        "( %XSIGN %COUNTER , %YSIGN %COUNTER )"

        #QUADRANT = {%QUADRANT + 1}
        IF (%QUADRANT > 4) #QUADRANT = 1
        #COUNTER = {%COUNTER + 1}
}
#COUNTER = 0
WHILE (%COUNTER < %NUM_POINTS){

        #COORD_LIST = {%BASE_COORD + %COORD%COUNTER}
        #COUNTER = {%COUNTER +1}
        #COORD_LIST =           &
                %COORD_LIST {%BASE_COORD + %COORD%COUNTER}
        ADD WIRE TYPE=2 AT %COORD_LIST
}
```

When your positions involve more complex calculations, you should use the ROUND() function to snap each coordinate to grid. See page 235.

The example above will create a simple spiral construct using separate wire components for each leg of the spiral. It uses a single dimension array of macros to store the coordinates of the endpoints of each wire. The macros are defined in the first WHILE loop with the statement:

LOCAL #COORD%COUNTER =…

Each time through the loop, as the COUNTER macro is incremented, a new macro is created based on the base name of "COORD" with the value of COUNTER concatenated on the end of the macro name. The first time through the loop, the macro COORD0 is defined. The second time through the loop the macro COORD1 is defined, etc.

You can also easily create an array of two or more dimensions. Macro substitution can be used to add as many array subscripts as you need. There are only two tips to remember.

- When macro references are used to supply array subscripts, they must be delimited with '^'s to separate one macro reference from the next without a space.
- Some other delimiter (e.g. '.' or '_') must be used to separate one subscript from the next to avoid confusion between multi-digit subscripts and several single digit subscripts.

These tips are best explained by example. To define one of the macros in a multi-dimension array, use a macro definition similar to:

Example: **LOCAL #ARRAY.%I^.%J^.%K = …** !Correct syntax

This macro definition is typical of a multi-dimensional array defined in nested loops that increment the I, J, and K macros. The first few times through the loops macros similar to the following are created:

ARRAY.1.1.1
ARRAY.1.1.2
ARRAY.1.1.3

When the subscripts reach the multi-digit stage, there will be no confusion. The meaning of ARRAY.1.2.345 is clearly understood. If we removed the '.' delimiters from the macro definition, the macro created would be ARRAY12345, which is ambiguous. It could mean ARRAY.12.3.45, or ARRAY.1.234.5, etc.

If the '^' delimiters are omitted from the macro definition, the program will not be able to parse the statement properly to determine the macro references. If the macro definition was written as:

```
LOCAL #ARRAY.%I.%J.%K = …        !Incorrect syntax
```

The first macro substitution would be interpreted correctly resulting in a statement similar to:

```
LOCAL #ARRAY.%I.%J.1 = …
```

The program would then attempt the next macro substitution, however the next macro reference is "%J.1". Unless there is a macro with the name "J.1" the command file would fail at this point.

Keyboard Macros

This special class of macros is used to store a command string that can be executed by pressing a single key or a combination of keys at the beginning of a command line in the editor. Unlike other user-defined macros, these macros **are** saved in the cell file.

The KEY command can also be used to define keyboard macros.

Keyboard macros can be defined with the following syntax:

[GLOBAL] [#]**KEY.*key*** [=] [*command_string*]

The GLOBAL keyword is required if the definition is in a command file. It is the default if the definition is typed on the command line in the layout editor.

Delayed evaluation can be useful in keyboard macros. See an example on page 47.

The *key* portion of the name must be one or more characters. The following *key* strings have special significance:

F*n*	Function key <F*n*> (e.g. F1)
AF*n*	Key combination <Alt><F*n*>
CF*n*	Key combination <Ctrl><F*n*>
SF*n*	Key combination <Shift><F*n*>

See an example that assigns opening a text editor window to a key on page 10.

If *command_string* contains any quote characters, you must surround it with different quote characters. The four valid quote characters are ",',~, and `. If you omit *command_string,* the user will be prompted to enter the value.

Example: **GLOBAL #KEY.AF8 = ADD TEXT**

After this keyboard macro definition is executed, you will be able to execute an ADD TEXT command the layout editor by holding down the <Alt> key then pressing the <F8> key. The user will be prompted for the text string to add as a component.

When you use other strings for *key*, typing those keys as the first characters on the command line will execute the *command_string*.

Example: **GLOBAL #KEY.ZT = ADD TEXT=% LAYER=M1_TEXT SIZE=2**

Once this keyboard macro definition is executed, typing ZT on the command line will execute the indicated ADD TEXT command. Since the optional parameters LAYER and SIZE must come after the TEXT parameter, the special character '%' must used as the text string to allow the user to be prompted for the text string.

When a keyboard macro definition uses ordinary characters for *key* that would make it ambiguous whether the user meant to execute the keyboard macro or a command that begins with the same characters, ICED™ will wait until the user presses <Enter> before assuming that the keyboard macro is intended.

Example: **GLOBAL #KEY.TE = ADD TEXT**

See an example of executing an operating system command with a keystroke on page 12.

After this definition is executed, if "TE" is typed as the first characters on the command line, ICED™ will wait to see what is typed next before executing the "ADD TEXT" command. If <Enter> or a space is typed, then the keyboard macro will be triggered, executing the "ADD TEXT" command. If a <M> is typed instead, then ICED™ will assume that you are typing the TEMPLATE command and the keyboard macro will not be triggered.

The keyboard macro is never triggered when the *key* string is typed in the middle of a command. The *key* string must be the only characters typed at the command prompt before an <Enter> or space is typed.

Reserved Macros

You cannot use the name of a system macro as a user macro name. See the list on page 255

There are several user macro names that have special significance when they are defined. Do not use these names for your macros unless you want to use them for the intended purpose.

ERROR.CMD	Defines an error handler. When a syntax error occurs, or an ERROR command is executed, the string stored in this macro is executed. See page 165.
ENTER.SUBCELL	When an edit command is executed to open a child cell (any cell except the root cell), the string stored in this macro is executed. See page 21.
EXIT.SUBCELL	When an EXIT, QUIT, or LEAVE command is used to return from editing a child cell, the string stored in this macro is executed before the cell is closed. See page 21.
EXIT.ROOT	When an EXIT, QUIT, or LEAVE command is used to terminate the editor, the string stored in this macro is executed before the editor closes. See page 20.

Commands Used Primarily in Command Files

The following commands are described here, rather than in the IC Layout Editor Reference Manual, because they are rarely used outside of command files.

Category	Command	Use	Page
Condi-tional execution	IF, ELSE, and ELSEIF	Conditionally execute statements or blocks of statements.	168
	WHILE	Conditionally execute a statement or a block of statements more than once.	212
Goto type	*@file_name*	Suspend execution and execute a command file	159
	BACK_TO	Force interpreter to skip lines backward until a statement with a specific label is reached	163
	SKIP_TO	Force interpreter to skip lines until a statement with a specific label is reached	209
	RETURN	End command file immediately.	205
	ERROR	End command file immediately w/ error message	164
Informa-tion	$*comment*	Display comment on screen and in journal file	159
	ITEM	Store component information in macros	173
	LIST	Save named list of components	182
	REMOVE	Delete macros	203
	MARK_SUBCELLS	Initialize contents of SUBCELL.EDIT macro to return edibility of given cells.	197
Command file control	LOG	Enable or disable logging of commands in journal file	173
	VIEW ON | OFF	Enable or disable screen updates during command file.	209
	XSELECT	Disable embedded SELECT commands	215
User interaction	PAUSE	Pause command file with an optional message	199
	PROMPT	Display message to user on screen	201
	SHELL	Suspends command file to allow menu access	207

Figure 23: Commands used primarily in command files

Table of Commands Covered in Other Manual

Some of the commands covered only in the IC Layout Editor Reference Manual have features added primarily for use in command files. However, **only commands used exclusively in command files are covered in this manual.** You may often need to refer to the description of commands in the other manual when writing command files.

Always reread the command description carefully for commands you use in command files. For example, the GROUP command creates a new cell from currently selected components. If a cell with the name indicated in the command already exists, you must respond with a prompt to the warning message. These prompts can make command files awkward. The [YES|NO] parameters were added to the GROUP command to allow you to avoid the warning prompt in command files.

Command	**Feature useful in command file**	**Expl in this manual**
@file_name	Execute command file	97
ADD	OFFSET keyword allows you to type list of ADD commands easily	271
[UN]BLANK	Make some components or entire layers unselectable and invisible	275
BLINK	Used to highlight component(s) for user	-
DELETE	If XSELECT mode is OFF, DELETE command will not issue embedded SELECT NEAR command to select component for deletion if no components are selected	84
DOS	Allows DOS programs to be executed	312
EDIT	Execute commands in other cells; LOCAL_COPY and VIEW_ONLY keywords allow you to avoid warning prompts for protected libraries	325
GROUP	YES | NO keywords allow you to avoid warning prompts	183
JOURNAL	If command file goes wrong and leaves nested cells corrupted, JOURNAL quits editor without saving any cells	129

KEY	Shorthand for creating GLOBAL macros that assign functions to keys	47
LEAVE	Using this command after an edit command instead of EXIT will prevent an unaltered cell from having it's cell file overwritten	108
P_EDIT	LOCAL_COPY and VIEW_ONLY keywords avoid warning prompts	325
PROTECT	Prevent command file from altering certain layers or components	185
REDRAW	Redraw screen when command file may leave screen cluttered	-
SELECT	ID keyword allows you to select components added before or after a benchmark id was issued	312
	Stack keywords PUSH, POP, and EXCHANGE allow you to save and restore component selections at beginning and end of command file	112
	FAIL keyword allows you to select components that caused a command in command file to fail	-
	LIST keyword allows you to select components one by one from a list	284
	TAG keyword allows you to select DRC generated shapes by rule number	-
SHOW	USER_MACROS keyword allows you display current macro values, or save definitions to be restored in future sessions	174
	SYSTEM_MACROS keyword displays current system settings	250
	FILE keyword allows you to export component or macro definitions to a file to be manipulated outside of editor	327
SNAP	Temporarily change the grid used for digitizing positions	330
SPAWN	Launch another application without pausing editor	265
T_EDIT	LOCAL_COPY and VIEW_ONLY keywords avoid warning prompts	325
TEMPLATE	Allows you to save and restore editor settings	116
VIEW	Change view window to area user needs to see during command file	301

Figure 24: Some commands described in IC Layout Editor Reference Manual that have features useful in command files

@file_name

Execute a command file.

@[path\]file_name

See an overview of methods to execute command files on page 13.

Learn more about the command file search path on page on page 14.

This command causes ICED™ to execute the commands in file *file_name*.

If *file_name* does not include an extension, ICED™ will add an extension of .CMD to *file_name* before looking for the file. If you omit *path*, ICED™ looks for the file in the following directories in the order shown:

1) the working directory,
2) the directories specified by the ICED_CMD_PATH environment variable,
3) the AUXIL directory(ies).

Example: **@GEORGE**

ICED™ will execute the commands in the file GEORGE.CMD if this file can be found in the working directory or in one of the other directories listed above.

Example: **@D:\WORK\NAND.LOG**

ICED™ will execute the commands in the file D:\WORK\NAND.LOG.

The @%.cmd Menu Option

Command files with a "_" prefix are meant to be called from other files.

The ICED™ menu option '@%.cmd' allows you to select a command file to execute from lists of the command files on the command file search path. Only files with a .CMD extension and without a "_" prefix in the file name will be listed.

Nested Command Files

The @* command executes the commands in the startup and always command files. See the IC Layout Editor Reference Manual for details.)

The VIEW, LOG, and XSELECT modes selected by using those commands in a command file will be used in any *@file_name* commands nested within the file. The VIEW and XSELECT modes can be overridden with commands in the nested *@file_name* command files.

If a command file contains a LOG=OFF command, logging to the journal file remains off until the command file is completed. There is no way to override the LOG OFF mode.

Command files can be nested up to 16 deep.

Other Commands in the Same Statement

If the *@file_name* command is issued with other commands on the same command line using semicolons to delimit them, the other commands will execute before the first command in the command file.

Example:

```
@CMDFILE;VIEW OFF; XSELECT OFF
```

In this example, the command @CMDFILE is parsed first by the command interpreter. Then the VIEW OFF command is executed as though it is the first line of the command file. The XSELECT OFF command is executed next. Finally, the commands in the file CMDFILE.CMD are executed.

Example:

```
@_GET_INT; LOCAL #MIN=0; LOCAL #DEFAULT=1;   &
        LOCAL #PROMPT="initial serial number"
```

The example above will result in the macro definitions being executed before the commands in the _GET_INT.CMD command file are processed. This allows you to pass values into a command file. See more details on page 97.

Note that the '&' continuation character allows this statement to be typed on more than one line.

$comment — ***Add comment to journal file and screen.***

$*comment*

or

$$*comment*

$Comments are displayed regardless of the mode set with the VIEW ON/OFF command.

Whenever ICED™ processes a statement with a '$' as the first character, the remainder of the line is treated as a comment. The '$' and the remainder of the line will be logged into the journal file as a comment and the line will echo on the screen below the command line. Even if there is a valid command in the comment, it will not be executed.

Example: **$My command file completed successfully; VIEW IN 3**

The entire line above will be treated as a comment and be will copied into the journal file and displayed on the bottom of the editor window exactly as shown. The VIEW command will **not** be executed.

Although commands in $comments will not be executed, macro substitution and expression evaluation will be performed.

Example: **$ 5 + %MYMACRO={ 5 + %MYMACRO }**

Adding a **PAUSE** command after a $*comment* will leave the comment on the screen until a key or mouse button is pressed.

If the value of MYMACRO is "10", the comment will produce the following echo on the screen and in the journal file:

$ 5 + 10=15

$Comment commands can be used to mark locations in the journal file. Let us say that you are performing a series of complicated editing tasks. After each successful step, you can add a comment that the step is complete. If the last step you performed went badly, and you need to recover the state of the design several steps back to correct the problem, you can edit the journal file using the comments to locate the step where things went wrong. After the journal file is edited, you can use it to recover your work up to the step where problems began.

Differences Between $comments, $$comments and !comments

The LOG SRCEEN=OFF command will not prevent the display of $comments.

During long command files, you may want to turn the logging of commands to the command file off with the LOG OFF command. This will speed up a long command file considerably. This also turns off the display and logging of $comments.

$$comments will still be displayed and logged to the journal file even when the LOG OFF mode prevents the display and logging of other commands. This is the only difference between $comments and $$comments.

Example:

```
LOG OFF
LOCAL #I = 0
WHILE (%I <= 10000){
                !missing processing on components
        IF (INT(%I/100) == (%I/100)) &
                $$ %I components processed successfully
        #I = {%I + 1}
}
```

The example above will update the message on the screen as the loop progresses. The message is updated only every 100 components so that it is left on the screen for a brief moment instead of being a blur from being overwritten every time through the loop.

!comments can also be used to create comments in command files, but these comments will not be logged into the journal file or displayed on the screen. They are used only as documentation within the command file itself.

BACK_TO *Force interpreter to go back to a specific line.*

BACK_TO *label_name*[:]

Use the SKIP_TO command instead to search forward for a labeled statement.

The BACK_TO command causes the command interpreter to search upward in the current command file for a statement with label *label_name*. Execution will continue from that statement.

The simplest way to label a statement is to type a label string followed by a colon ":". More details on label syntax are shown on page 32.

Example:

```
BEGIN:
        ! missing code that performs some action
@_GET_ANS;    #prompt="Are you happy with these results?";       &
              #choices="yn";
IF (%ret.value != 1) BACK_TO BEGIN     !if <n> repeat from BEGIN label
```

The _GET_ANS-.CMD command file is supplied with the installation.

This example assigns the label "BEGIN" to a statement. Assume that the lines following this label gather some information from the user and then perform some action. Then the _GET_ANS.CMD command file asks the user for an answer to the prompt. In this case it expects the user to type a <y> or a <n>.

_GET_ANS.CMD will set the ret.value macro to a 1 if the user types <y>. In this case, the BACK_TO BEGIN command is not executed. Execution will continue with the line after the IF statement. However, it the user types a <n> response, ret.value will not be set to 1 and the BACK_TO BEGIN command is executed. In this case execution continues with the line after the BEGIN label.

The BACK_TO command cannot use a label that does not exist in the current command file. You should avoid using BACK_TO to jump to a statement inside of a WHILE, IF, or ELSEIF block.

ERROR

Display error message.

ERROR "*err_msg_string*"

or

ERROR

The quotes around *err_msg_string* are recommended to avoid parsing problems, but can be omitted in most cases.

Use this command to display an error message to the user when your command file fails.

The two forms of the ERROR command are used quite differently Examples of both uses follow this page.

ERROR ***err_msg_string*** Posts error message and terminates the current command file, leaving the word "Error" drawn in red and the *err_msg_string* visible in the history area at the bottom of the ICED™ window.

ERROR This form is useful only when the ERROR.CMD macro is defined. (We'll cover this macro in more detail on page 165.) In this case, an ERROR *err_msg_string* command will not display the message on the screen and terminate the command file immediately. Instead, the command processor executes the code indicated by the ERROR.CMD macro. This code should include an ERROR command (without an argument) to post the last error message saved by a previous ERROR *err_msg_string* command. The command file is not terminated immediately. This gives you the opportunity for more error handling processing after the error message is posted.

Example:

```
LOCAL #MYLAYER = $PROMPT="Enter layer for operation:"

IF (CMP("%MYLAYER", "")==0) {                !User just pressed Enter
        ERROR "No layer entered."
}
```

The CMP function compares two strings and returns a 0 when they are the same. See page 224.

When the user responds by pressing <Enter> to the prompt in the example above, the editor will post the message "Error: No layer entered." at the bottom of the ICED™ window. The word "Error" will be displayed in red text to highlight the problem for the user. The command file would terminate at this point and no other commands in the file would be processed.

Macro substitution can be used in the *err_msg_string*. The example above might continue with the following lines:

Example:

```
ELSE {                                       !User typed response
        IF (VALID_LAYER("%MYLAYER")){
                $layer valid, add processing code here
        }
        ELSE{
                ERROR "%MYLAYER is not a valid layer."
        }
}
RETURN "Processing successful on layer %MYLAYER"
```

The VALID_LAYER function returns a 0 if the argument is not an existing layer. See page 243.

The string the user typed replaces the macro reference %MYLAYER in the error message string before it is displayed in the history area.

The RETURN command in the example above will be processed only when the user types a valid layer in response to the prompt.

ERROR.CMD Macro Processing

When an ERROR *err_msg_string* command is executed, or a syntax error is encountered, the command interpreter checks for the existence of a macro with the name ERROR.CMD. If such a macro exists, the string stored in it is executed as a command string instead of displaying the error message and terminating the command file.

See a brief overview of error handlers on page 125.

ERROR.CMD can contain any valid command string including a *@file_name* command to call another command file to process the error. However, ERROR.CMD typically stores a SKIP_TO command that triggers an error handler part of your command file. This section of code should contain an ERROR command without an argument to display the error message stored by the previous error.

Example:

The SKIP_TO command causes the command interpreter to skip immediately to the labeled statement. See page 209.

Always include a RETURN command to terminate the command file normally before an error block.

The JOU system macro stores the name of the journal file

```
LOCAL #ERROR.CMD="SKIP_TO FAIL"
LOCAL #MYLAYER = $PROMPT="Enter layer for operation:"

IF (CMP("%MYLAYER", "")==0) {           !User just pressed Enter
        ERROR "No layer entered."
}
ELSE {                                  !User typed response
        IF (VALID_LAYER("%MYLAYER")){
                $layer valid, add process code here
        }
        ELSE{
                ERROR "%MYLAYER is not a valid layer."
        }
}
RETURN "Processing successful on layer %MYLAYER"

FAIL:
$Error handler triggered
ERROR
DOS -NOTEPAD "%JOU"            !User must close new window to continue
PAUSE
```

The example above is similar to the previous example, except that since ERROR.CMD is defined (see first line), the string in it will be executed when either ERROR *err_msg_string* command is executed. All of the commands in the block beginning with the label "FAIL" will be executed. This error handler opens a Notepad editor window containing the journal file for the current session. This allows the user to see what commands were executed by the command file before the error occurred.

A PAUSE command may be required to allow the user to see the message displayed by the form of the ERROR command without arguments. When this form is used, execution is not halted and succeeding commands may erase the error message. The PAUSE is required only if the following types of commands will be executed in the command file after the ERROR command.

- a $*comment* command,
- a RETURN command that defines a *msg_string*
- a PROMPT command,
- an interactive command that requires use of the status lines, or
- another error.

This type of error handler is particularly useful when debugging command files. Errors caused by syntax problems in the command file will trigger the error handler rather than immediately terminating the command file.

If the command string stored in ERROR.CMD is not a SKIP_TO or BACK_TO command, after the command string is executed, execution of the command file will resume at the statement after the command that failed.

Example: **LOCAL #ERROR.CMD="#FAILED=1"**

This type of ERROR.CMD definition will prevent the display of any error message when a command fails or an ERROR command is executed. The FAILED macro will be set to '1' and the command file will continue. This is the method used in the ED.CMD file. See page 318.

IF, ELSEIF, ELSE — *Conditionally execute statements*

IF **(** ***boolean_expression*** **)** ***single_statement***

```
IF      ( boolean_expression ) {
                !Block of statements
        }
```

ELSEIF **(** ***boolean_expression*** **)** ***single_statement***

```
ELSEIF  ( boolean_expression ) {
                !Block of statements
        }
```

ELSE ***single_statement***

```
ELSE    {
                !Block of statements
        }
```

See an overview of conditional statements beginning on page 90.

(Unlike most syntax descriptions, the parentheses used above in the syntax descriptions must be typed in the command.)

FALSE	**0**
TRUE	**1** (or any non-zero number)

Figure 25: Boolean values

The IF command is used to execute a statement or block of statements only when a Boolean expression evaluates to TRUE, i.e. any non-zero number.

The ELSEIF and ELSE commands are optional after an IF command. They are executed only when the Boolean expression in the preceding IF command is FALSE, i.e. 0.

The IF Command

A complete explanation of Boolean expressions is provided on page 53.

The IF command is used to execute one statement or a block of statements only if the *boolean_expression* evaluates to TRUE. Briefly, a Boolean expression is an expression that evaluates to a number. If the number is 0, the expression is FALSE. If the number is non-zero, then the expression is TRUE.

Example:

```
LOCAL #ASSERTION = $PROMPT="Please type a 1 or a 0:"
IF (%ASSERTION)    $The assertion is true
```

This simple example uses the single statement syntax of the IF command. If the value of the ASSERTION macro is non-zero, the $*comment* command will be executed. This command will print the comment on the prompt line of the ICED™ window. If ASSERTION does equal 0, then the $*comment* command will not be executed.

Example:

```
LOCAL  #NAME  $PROMPT="Enter name [FRED]:"
IF (LEN("%NAME") == 0) {
        #NAME = FRED
        $Default name used
}
```

The LEN() function returns the number of characters in a string. See page 229.

The first line above prompts the user to type a string to define the initial value of the NAME macro. If the user simply presses <Enter> in response the prompt, the value of the NAME macro will be "". When NAME is "", the LEN("%NAME") function call will return 0 and the IF condition will be TRUE.

Always type a blank before the condition expression in ( ).

This example demonstrates the block form of the IF command. The block begins with the open curly brace '{' at the end of the line containing the IF command. The block ends with a line that contains only the closing curly brace '}'. Both commands "#NAME = FRED" and "$Default name used" will be executed when the condition in the IF command is TRUE. Both commands are skipped if the condition is FALSE.

If the Boolean condition expression evaluates to FALSE, then control passes immediately to the statement following the end if the IF block, indicated by the

'}'. Statements in the IF block are not executed or even evaluated. Macro substitutions or definitions in the block will not be performed.

Ordinarily, any time a function call or operator is used in an expression, the expression must be surrounded by "{}" to force ICED™ to evaluate the expression rather than interpreting it as a string. Note that in the example above, no "{}" are used to surround the *boolean_expression* "LEN("%NAME") == 0". This is because ICED™ will always interpret the *boolean_expression* in an IF, ELSEIF, or WHILE command as an expression. The "{}" are not required.

Note the double equals "==" in the condition statement of the IF command. This is the proper syntax to use in a Boolean expression to test that two values are equal. It is a common mistake to forget that the "==" is required. You can review Boolean expression syntax beginning on page 53.

The ELSE Command

The ELSE command can be used only immediately after an IF command (or after the closing '}' of an IF block). It is used to execute a single statement, or a block of statements, if the Boolean expression in the preceding IF command is FALSE (i.e. equal to 0). No statements can come between the end the IF command (or block) and the ELSE command.

Example:

```
LOCAL #ASSERTION = $PROMPT="Please type a 1 or a 0:"
IF (%ASSERTION)   $The assertion is true
ELSE              $The assertion is false
```

Example:

```
LOCAL  #NAME  $PROMPT="Enter name:"
IF (LEN("%NAME") == 0) {
        #NAME = FRED
        $Default name used
}
ELSE $Hello there %NAME
```

The ELSE command can control an entire block of commands by surrounding them with curly braces in the same manner as an IF block. The open curly brace must be at the end of the same line as the ELSE command. The closing curly brace should be on a line by itself.

Example:

```
IF (%SNAP.ANGLE == 90) {
        #VERTEX1 = (1,1)
        #VERTEX2 = (1,2)
        #VERTEX3 = (2,2)
}
ELSE {
        #VERTEX1 = (1,1)
        #VERTEX2 = (2,2)
        #VERTEX3 = ""
}
```

The system macro SNAP.ANGLE is described on page 293.

The command file fragment above demonstrates how to use the block form of both the IF and ELSE commands. In this case, different vertices are stored in the macros based on the value of the system macro SNAP.ANGLE.

The ELSEIF Command

The ELSEIF command is used to implement a form of a case statement. That means you can write a series of statements, beginning with an IF statement and continuing with one or more ELSEIF statements, that execute different commands depending on the values of various condition expressions.

If the Boolean expression in the IF condition is TRUE (i.e. non-zero), the statements in the IF block will be executed and any following ELSEIF or ELSE statements and their corresponding blocks will be ignored. When the Boolean expression in the IF command is FALSE (i.e. equal to 0), the program will then execute the following ELSEIF command. If the Boolean expression in the ELSEIF is TRUE, then the single statement or block of statements controlled by the ELSEIF command is executed and control passes to the next statement after all other ELSEIF or ELSE blocks. If the Boolean expression in the ELSEIF command is FALSE, the statements in the block are skipped and control passes to the next command. The next command can be another ELSEIF command, an ELSE command, or any other command.

No statements can come between the end the IF command or block and the ELSEIF command.

Example:

```
LOCAL #ASSERTION = $PROMPT="Please type a 1 or a 0:"
IF          (%ASSERTION == 1)         $The assertion is true
ELSEIF  (%ASSERTION == 0)         $The assertion is false
ELSE                                          $I told you to type a 1 or a 0
```

Only one of the three $*comment* commands in the example above will be executed.

Example:

```
LOCAL  #NAME  $PROMPT="Enter name:"

IF          (CMP(%NAME, "LUCY")==0){
          $Hello Lucy
}
ELSEIF  (CMP(%NAME, "RICKY")==0){
          $Luuuuuuuuucy!
}
ELSEIF  ((CMP(%NAME, "FRED")==0) ||          &
            (CMP(%NAME, "ETHEL")==0)){
          $Hide the talent scout in the closet.
}
ELSE   ERROR "%NAME is at the door"
```

The ERROR command aborts the command file with the message, or triggers an error handler.

The previous example uses the string comparison function CMP to test the value of macro NAME. The CMP function returns 0 when two strings are equal.

The CMP function is described on page 224. See a list of other comparison operations on page 54.

Note that this example uses more than one ELSEIF statement. You can use as many as required. Note also the compound Boolean expression in the second ELSEIF statement. If the value of NAME is either FRED or ETHEL, the command "$Hide the talent scout in the closet" will be executed. The use of the '&' to allow a single command to span more than one line makes this example a little more readable.

See other examples of IF commands on pages 92, 225, and 301.

ITEM

Get information on single component

ITEM [DEFAULT] [GLOBAL | LOCAL] [#]***item_name*** [BOX=(<u>BOX</u> | POLYGON)]

Information on the ITEM command is also provided in the Q:\ICWIN-\DOC\LIST.TXT file.

This command is used to place information on a **single component** into a collection of macros. The number of macros created by this command and the information stored in them depends on what type of component is selected when the command is executed. The complete list is provided in the table on page 175.

Exactly one component must be selected or this command will fail.

The values stored in the ITEM macros can be modified with macro assignment statements. A new component based on these new values can be easily created. See the example on page 180.

The DEFAULT keyword is used to force the editor to ignore a redefinition of an existing set of item macros. The values of the existing item macros remain unchanged.

See page 41 for a definition of scope.

The GLOBAL or LOCAL choice of keywords defines the scope of the macros created. The first time an ITEM command is executed in a command file to define a specific *item_name*, one of the keywords **must** be used. Once the macros have been created by a previous use of the command, then repeated uses of ITEM commands with the same *item_name* can omit both keywords.

See this command used in the advanced example SERIAL.CMD on page 312.

Choose *item_name* carefully so that none of the created macro names will conflict with existing macros (including system macros). If the ITEM command is unable to create all macros, then the value of the *item_name*.TYPE macro will be "ERROR".

The BOX keyword controls how information on boxes is stored. See page 179.

The following restrictions are enforced for an *item_name*:

- Names can be from 1 to 16 characters long.
- Valid characters consist of all alphanumeric characters and '.', '_', and '$'.
- Do not use a number or '#' as the first character in an item name.

Example: **ITEM GLOBAL #MYITEM**
SHOW USER #MYITEM*

Use the REMOVE command. to delete item macros See page 203.

This example creates a series of global macros with names beginning with "MYITEM". For example one of the macros created will be MYITEM.TYPE. If a single box is selected when the command is executed, the value of MYITEM.TYPE will be "BOX". The SHOW command displays the newly created macros. Since the GLOBAL keyword is used, the macros will persist until you delete them with a REMOVE command or terminate the edit session.

```
GLOBAL MYITEM.AREA="500"
GLOBAL MYITEM.BB.X0="-28.5"
GLOBAL MYITEM.BB.X1="-3.5"
GLOBAL MYITEM.BB.Y0="0"
GLOBAL MYITEM.BB.Y1="20"
GLOBAL MYITEM.ID="1"
GLOBAL MYITEM.LAYER="1"
GLOBAL MYITEM.N.POINTS="2"
GLOBAL MYITEM.OFFSET="(0, 0)"
GLOBAL MYITEM.PERIM="90"
GLOBAL MYITEM.POS.1="(-28.5, 0.0)"
GLOBAL MYITEM.POS.2="(-3.5, 20.0)"
GLOBAL MYITEM.TAG="0"
GLOBAL MYITEM.TYPE="BOX"
```

Figure 26: ITEM macros for a box displayed by SHOW command

See the example on page 269 for using ITEM in a loop when you need to process many components.

When the XSELECT mode is "ON" (the default mode), and no components are selected when the ITEM command is executed, then the ITEM command will generate an embedded SELECT NEAR command to allow the user to select a single component. Add an XSELECT OFF command (see page 215) before the ITEM command if you prefer to have the ITEM command create only the single macro *item_name*.TYPE with a value of "ERROR" when no component is selected.

Example: **IF (%N.SELECT ==1) ITEM GLOBAL #MYITEM**

The example above tests that exactly one component is selected before creating item macros. Insuring that exactly one component is selected before executing the ITEM command is important since the command will fail (and abort the command file) if more then one component is selected when the command is executed. See more details on this subject on page 180.

Macro name	Purpose	Box	Polygon	Wire	Line	Text	Cell	Array
item_name.AREA*	Area	X	X	X	X	X	X	X
item_name.BB.X0*	Bounding box lower left corner x-coordinate	X	X	X	X	X	X	X
item_name.BB.X1*	Bounding box upper right corner x-coordinate	X	X	X	X	X	X	X
item_name.BB.Y0*	Bounding box lower left corner y-coordinate	X	X	X	X	X	X	X
item_name.BB.Y1*	Bounding box upper right corner y-coordinate	X	X	X	X	X	X	X
item_name.CELL.NAME	Cell name						X	X
item_name.CELL.NO*	Index into cell table						X	X
item_name.COL.STEP	Pitch of array columns							X
item_name.ID*	Component id number	X	X	X	X	X	X	X
item_name.JUST	Text justification (See page 300 for codes)					X		
item_name.LAYER	Layer number	X	X	X	X	X	X	X
item_name.LEN*	Length of component (including type 2 extension)			X	X			
item_name.LINE.*n*	Text strings (number of macros stored in *item*.N.LINES)					X		
item_name.MY*	Mirror code: 1 if mirrored in Y direction, else 0					X	X	X
item_name.N.COLS	Number of columns							X
item_name.N.LINES	Number of lines of text					X		
item_name.N.POINTS	Number of vertices	X	X	X	X	X	X	X
item_name.N.ROWS	Number of rows							X
item_name.OFFSET	Always created as "(0,0)"; can be changed to relocate component added with ADD.*item_name*	X	X	X	X	X	X	X
item_name.PERIM	Perimeter	X	X	X	X	X	X	X
item_name.POS.*n*	Vertices (number of macros provided in *item*.N.POINTS)	X	X	X	X	X	X	X
item_name.ROT*	Rotation code ("0", "1", "2", or "3")					X	X	X
item_name.ROW.STEP	Pitch of array rows							X
item_name.SIZE	Height of text					X		
item_name.TAG	DRC tag integer (See DRCSTEP.CMD)	X	X	X	X	X	X	X
item_name.TRANS	Transformation code: "R0", "R1", "R2", "R3", "MY", "R1 MY", "R2 MY", "R3 MY"					X	X	X
item_name.TYPE	Component type (Use upper case when changing.)	X	X	X	X	X	X	X
item_name.WIDTH	Width of wire			X	X			
item_name.WIRE.TYPE	Wire end code: "0" (flush)or "2" (extended)			X	X			
* Ignored by ADD.*item_name* macro when creating a new component.								

Figure 27: Macros created by ITEM command

Using the ITEM Macros

It is a good programming practice to test the value of *item_name*.TYPE before using the macros created by the ITEM command. This macro will contain the string "ERROR" if the ITEM command failed for any reason. When this is the case, the creation and value of the other *item_name* macros is undependable.

The ADD.*item_name* macro creates a new component based on the current values in the other *item_name* macros. See page 180.

The existence and relevance of other *item_name* macros depend on the value currently in *item_name*.TYPE. For example, consider what happens when the example below is executed in the following situations:

1) The user selects a wire.
2) The user selects more than 1 component.
3) The user selects nothing, and no MYWIRE item macros already exist.
4) The user selects a box, and no MYWIRE item macros already exist.
5) The user selects a box, and GLOBAL MYWIRE item macros do already exist for some other wire component.

Example:

```
XSELECT OFF
UNSEL ALL
$Select wire for processing
SELECT NEAR
ITEM LOCAL #MYWIRE
LOCAL #ORIG_WIDTH = %MYWIRE.WIDTH       !this stmt may fail
```

In case number 1 (user selects a wire), then MYWIRE.TYPE = "WIRE" and the ITEM command will create the MYWIRE.WIDTH macro using the width of the selected wire. The reference to %MYWIRE.WIDTH will successfully resolve to the width of the selected wire.

See page 180 for a method of making sure a single component is selected.

In cases 2, 3 and 4, the command file will fail. In case number 2 (user selects more than 1 component), the ITEM command will fail and the command file will be terminated. In case number 3 (user selects nothing), MYWIRE.TYPE = "ERROR". In case number 4 (user selects box), MYWIRE.TYPE = "BOX". In neither of these cases will a macro with the name MYWIRE.WIDTH be created. So either case will fail with an error message when the %MYWIRE.WIDTH reference is processed.

Case number 5 is the most dangerous. In this case the user selects the wrong type of component, so MYWIRE.TYPE = "BOX" and no MYWIRE.WIDTH macro is created. However, since a MYWIRE.WIDTH macro already exists from a previous execution of the ITEM command, the command file does not fail. The information in the MYWIRE.WIDTH macro is incorrect, but no error is generated.

The CMP function returns 0 when two strings match. See page 224.

These problems are easily avoided if you test the value in the *item_name*.TYPE macro every time you use the ITEM command. Use lines similar to those below.

Example:

```
IF (CMP(%MYWIRE.TYPE, "WIRE" )!= 0)           &
        ERROR "Wire not selected or error encountered."
                    or
IF (CMP(%MYWIRE.TYPE, "ERROR")==0) ERROR "Nothing selected."
IF (CMP(%MYWIRE.TYPE, "WIRE")!=0) ERROR "Must select wire."
```

The ERROR command posts a message and terminates the command file. See page 163.

Another good programming tip to avoid problems like case 5, is to use the REMOVE command (see page 203) to delete all obsolete item macros just before executing the ITEM command.

Example:

```
REMOVE #MYWIRE*
```

Referring to ITEM Macros in Conditional Statements

When you create a WHILE or IF command condition statement using item macros, be sure that all macro references that will be evaluated refer to macros that are guaranteed to exist. Remember that **all** macro references in a compound IF condition will be evaluated. However, macro references inside an IF block that is not executed will not be evaluated.

Example:

```
ITEM LOCAL #THIS
IF ( CMP(%THIS.TYPE, "ERROR" )!= 0) {
     IF ((CMP(%THIS.TYPE, "WIRE")==0) &&     &
        (%THIS.AREA <= %MIN_AREA)) {
            LOCAL #MYWIDTH = %THIS.WIDTH
            !other processing
     }
}
```

See that table on page 175 to see what item macros are created for different types of components.

The first IF condition refers only to the *item_name*.TYPE macro. Even if the ITEM command failed because no component was selected, the *item_name*.TYPE macro will exist. The inner IF condition can refer to the *item_name*.TYPE and *item_name*.AREA macros because these macros are created for all valid components. For example, even if a line component is selected, the ITEM command is guaranteed to create the *item_name*.AREA macro. This value is not applicable to a line component, but the creation of the macro insures that executing a macro reference to it will not cause the command file to fail. The *item_name*.WIDTH macro is created only for wires, but the reference to it is safely inside the IF block that insures that the ITEM command was executed while a wire was selected.

Using the item_name.LAYER *Macro*

The MACRO-_EXISTS() function can be used to test that a specific item macro exists. See page 231.

The ITEM command stores the layer **number** of the component in the *item_name*.LAYER macro. This macro is guaranteed to be created for all component types. (When the component is a cell or array, then *item_name*-.LAYER is created with a value of "0".

To test the value of the layer macro against a specific layer name, you can use the system macro LAYER.NUMBER.*name* to derive the layer number from the layer name. Use syntax similar to the following.

Example: **IF (%MYWIRE.LAYER==%LAYER.NUMBER.M1) …**

See page 277 for LAYER-.NUMBER-*.layer_spec*.

(When changing the layer stored in the *item_name*.LAYER macro, you can use either the layer number or the layer name string.)

Position Lists

The ITEM command always creates an *item_name*.N.POINTS macro for any valid component. This macro will contain the number of vertex points. Each vertex point *n* will have a *item_name*.POS.*n* macro that stores the coordinate pair of the vertex.

(For arrays, cells, and text components, *item_name*.N.POINTS always equals '1' and the origin of the component is provided in *item_name*.POS.1.)

Example:

```
ITEM LOCAL #MYCOMP
LOCAL #n = 1
LOCAL #POS_STRING = ""
WHILE (%n <= %MYCOMP.N.POINTS){     !loop concatenating vertex list
        #POS_STRING = %POS_STRING %MYCOMP.POS.%n
        #n = {%n +1}          !don't forget to always use {} around expressions
}
```

When the selected component is a polygon (*item_name*.TYPE="POLYGON"), the ITEM command creates two extra macros: *item_name*.POS.0 and *item_name*.POS.*m*, where *m* = *item_name*.N.POINTS +1. For example, if you are testing angles on a polygon with *n* vertices, as you test *item_name*.POS.1, you can refer to the previous position as *item_name*.POS.0 instead of *item_name*.POS.*n*. When testing *item_name*.POS.*n*, you can refer to the following position as *item_name*.POS.*n*+1 instead of *item_name*.POS.1.

Always test *item_name*.N.POINTS when using the position macros. Remember that obsolete position information from vertices with indices greater than the current *item_name*.N.POINTS may be left behind from an earlier ITEM command.

Using the **BOX** *Keyword*

For box components, you have two choices on how positions are stored in the position macros. The default behavior is to set *item_name*.N.POINTS to "2" and store opposite corners in *item_name*.POS.1 and *item_name*.POS.2.

However, if you prefer to use the same code for parsing the coordinate lists of boxes or polygons, you can add the BOX keyword to the end of the ITEM command. When BOX=POLYGON is included in the ITEM command and a box component is selected, *item_name*.TYPE will be set to "POLYGON" instead of "BOX", *item_name*.N.POINTS is set to 4, and the positions of all four vertices will be listed in position macros *item_name*.POS.1 through *item_name*.POS.4.

Selecting a Single Component

You can put statements like these in a command file for re-use. See SEL_ONE on page 96.

Exactly one component should be selected when the ITEM command is executed.

If more than one component is selected when the ITEM command is executed, the command will fail, and any remaining commands in the command file will not be executed.

So it is best to select the component with lines similar to the following to insure that exactly one component is selected.

Example:

```
WHILE (%N.SELECT !=1){
        UNSELECT ALL
        $ Select single component.  Press both mouse buttons to cancel.
        SELECT NEAR
}
```

The N.SELECT system macro contains the number of selected components.

When the WHILE statement is executed for the first time, if one component is already selected, the WHILE block is ignored. When this is not the case, all components are unselected and the SELECT NEAR command is executed while the $*comment* is displayed as a prompt. If the SELECT NEAR command selects more than one component, then the loop executes again. If the user presses both mouse buttons, the command file terminates.

Using the ADD.*item_name* Macro

The SHOW command will not list an ADD.*item_name* macro.

One of the macros created by the ITEM command is ADD.*item_name*. When this macro is executed as a command string, a new component based on the current values in the other *item_name* macros is created.

The following example uses the ITEM command to change the width of a wire.

Example:

```
@_GET_REAL.CMD;#default=5;#prompt="new width of wire";#min=.5
LOCAL #NEW_WIDTH = %RET.VALUE          !save number typed by user
```

Learn about _GET_REAL-.CMD on page 100.

```
WHILE (%N.SELECT !=1){
        UNSELECT ALL
        $ Select wire
        SELECT WIRE NEAR
}                                                                  !continued on next page
REMOVE #WIRE_ITEM*              !Delete obsolete macros
ITEM LOCAL #WIRE_ITEM           !Create new item macros
IF (CMP(%WIRE_ITEM.TYPE, "WIRE" )!= 0){
        ERROR Wire not selected or error encountered.
}
#WIRE_ITEM.WIDTH = %NEW_WIDTH

%ADD.WIRE_ITEM

DELETE                          !This deletes old wire.
```

The ERROR command posts an error message and terminates the command file. See page 163.

The command file above begins by using the _GET_REAL.CMD command file to obtain and verify the real number to be used as the new width of the wire. The WHILE loop verifies that exactly one component is selected.

All obsolete information in any pre-existing WIRE_ITEM macros is deleted with the REMOVE command. Next, the ITEM command is used to create macros that contain the component information for the selected component. One of these macros is the WIRE_ITEM.TYPE macro that is tested to insure that the selected component is indeed a wire. If the user had a single non-wire component selected before the command file was executed, this test avoids a mysterious failure when the WIRE_ITEM.WIDTH macro is used. That macro will not exist if the selected component is a box or polygon.

Next, the width in the appropriate item macro is replaced. Then the %ADD.WIRE_ITEM statement executes an ADD command to create a new wire identical to the selected wire, except for the new width.

Since the DELETE command to delete the old wire is the last statement, if any statement in the command file fails, the old wire will not be deleted.

LIST

Save a named list of components.

LIST [DEFAULT] [GLOBAL | LOCAL] [#]*list_name*

Information on the LIST command is also provided in the Q:\ICWIN-\DOC\LIST.TXT file.

This command is used to save a named set of components allowing you to unselect everything and then select one component at a time from the list with a SELECT LIST command.

Lists are valid only in the cell in which they were defined. (Therefore, you cannot use a list to copy a set of components to another cell.) You cannot have more than 6 lists for a given cell at any one time and a single list cannot contain more than 10,000,000 components.

The easiest way to copy a set of components into another cell is to **GROUP** the components into a cell, **ADD** the new cell to another cell, then **UNGROUP** the new cell.

The DEFAULT keyword is used to force the editor to ignore a redefinition of an existing list.

The GLOBAL or LOCAL choice of keywords defines the scope of the list created. LOCAL lists are valid only in the command file that defines them, while GLOBAL lists persist until you exit the current cell. (See When Lists Are Removed on page 188 for more details on how long lists persist.)

The SHOW1.CMD file supplied with the installation uses the LIST command.

The first time a LIST command is executed in a command file to define a specific *list_name*, one of the [GLOBAL | LOCAL] keywords **must** be used. Once the list has been created by a previous use of the command, then repeated LIST commands using the same *list_name* can omit both keywords. When you execute this command at the prompt (outside of a command file), you can omit both keywords and a global list is created by default.

Example: **LIST GLOBAL #MYLIST**

This example defines a list with the name MYLIST. All selected components will be added to the list. These components remain selected after the command.

Example:

```
UNSELECT ALL
ADD CELL NAND
SELECT NEW
UNGROUP
SELECT NEW
LIST GLOBAL #MYLIST
UNSELECT ALL; SELECT LIST #MYLIST NEXT
```

SELECT LIST commands can contain the keywords, PREVIOUS, FIRST, or LAST instead of NEXT.

The example above demonstrates the LIST and SELECT LIST commands. The cell NAND is added to the current cell, then ungrouped into its components. Only those components are selected and placed in the list MYLIST. Then all components are unselected, and the first component in the list is selected by the last command. You can repeat the last statement over and over to select each component imported from the NAND cell one at a time.

Selecting Components for a List

Any time you are selecting components, you can hold the <Shift> key to allow you to select as may components as required. Click in an empty area to finish selection

The LIST_EXISTS function tells you if a list has been created. See page 188.

See page 74 for an overview on selecting components.

When no components are selected when a LIST command is executed, an embedded SELECT NEAR command is executed to allow you to select components for the list. If the user holds the <Shift> key down during the embedded SELECT NEAR, a set of multiple components can be selected.

It is usually best to select the components for a list prior to executing the LIST command. Both fully and partially selected components will be put on the list. A component that was partially selected before being put on the list is fully selected by a SELECT LIST command.

When the XSELECT mode is "ON" (the default mode), and no components are selected when the LIST command is executed, then the LIST command will generate an embedded SELECT NEAR command to allow the user to select components. Add an XSELECT OFF command (see page 215) before the LIST command if you prefer to have the LIST command create nothing when no components are selected.

List Names

You can use the VALID_LIST_NAME function to verify that a string is valid as a list name. See page 188.

The following restrictions are enforced for a *list_name*:

- List names can be from 1 to 16 characters long.
- Valid characters consist of all alphanumeric characters and '.', '_', and '$'.
- Do not use a number as the first character in a list name.

Use the SHOW LIST=* command to list currently defined lists.

The '#' before the *list_name* is optional when defining a list for the first time. This character is often used to identify macro names in many other commands, and a list is simply a special case of a macro. However, since the parser expects a *list_name* after the required LOCAL or GLOBAL keyword, the '#' is not required.

However, if you are redefining the contents of an existing list, you can omit the LOCAL or GLOBAL keywords. In this case, the '#' is required to tell the parser that the following string represents a list name. It is a good programming practice to always precede a list name with a '#'.

Example:

```
SELECT ALL
LIST #MYLIST          !might cause syntax error
```

In a command file, the LIST command above would result in a syntax error unless the list MYLIST was previously defined with either the LOCAL or GLOBAL keywords. If this is the case, the previous contents of the list are replaced with the list of all components in the current cell.

Building a Set of Components in a Loop

The following example selects small boxes on a contact layer. It saves the set of small contacts on a second list for further processing.

There is no simple way to add components to an existing list. Since we want to build up a new set of small contacts as the loop is executed repeatedly, we use the select stack to store the small contacts until the loop is complete, then create the final list from the set of components on the stack.

Example:

```
LOG LEVEL=BRIEF SCREEN=OFF      ! Speeds up command file
DEFAULT LOCAL #MIN_AREA = .5
DEFAULT LOCAL #CONTLAYER = "CONT"
LOCAL #NOT_DONE = 1

UNBLANK ALL; UNPROTECT ALL;     !Make all components selectable
UNSELECT ALL
SELECT PUSH          !empty select stack
XSELECT OFF          !required in case no components are on CONTLAYER
SELECT LAYER=%CONTLAYER BOXES
LIST LOCAL #ALLCONTACTS
UNSELECT ALL

WHILE (%NOT_DONE){
     SELECT LIST #ALLCONTACTS NEXT

     IF (%N.SELECT == 0)    #NOT_DONE=0          ! Set to FALSE
     ELSE{
        ITEM LOCAL #THIS

        IF ((CMP(%THIS.TYPE, "BOX" )== 0) &&            &
            (%THIS.AREA <= %MIN_AREA)){

            SELECT POP              !leaves this component selected
            UNSELECT PUSH           !pushes selected components on stack
        }
     }
     UNSELECT ALL
}

SELECT POP
LIST GLOBAL #SMALLCONTACTS      !Persists after end of command file
```

Learn more about the select stack in the SELECT command description in the IC Layout Editor Reference Manual.

The ITEM command stores information on a single component in a set of macros. See page 173. Note that you can specify the LOCAL scope keyword on the ITEM command in the loop.

Macros Created by the LIST Command

These macros are deleted when the REMOVE LIST *list_name* command is executed. See page 188.

All of these macros can be referenced only after the corresponding LIST #*list_name* command is executed. You cannot set their values directly. These macros are also described in more detail in the section on system macros beginning on page 284.

LIST.EOL. list_name

See _DRCTAG1-.CMD for an example using LIST.EMPTY and LIST.EOL.

This macro is set initially to "-1" for a new list. Every time a SELECT LIST command for that list is executed, the macro is reset to one of the following values:

"0" when a component was successfully selected by the SELECT LIST command.

"1" when the end-of-list flag is set and no component was selected by the SELECT LIST command.

Example:

```
UNSELECT ALL; SELECT LIST #MYLIST FIRST
WHILE (%LIST.EOL.MYLIST == 0){
        !process component
        UNSELECT ALL; SELECT LIST #MYLIST NEXT
}
$out of loop, already passed end of list
SELECT LIST #MYLIST NEXT        !This stmt causes an error
```

To see another example of a SELECT LIST command, refer to page 284.

This loop will process every component on the list one at a time until the SELECT LIST command does not select a component because it has reached the end of the list. This will not cause an error condition or halt the command file. Instead the LIST.EOL.MYLIST macro is set to 1 and the command file continues. When the WHILE condition is tested, the loop ends and control passes to the $*comment* after the WHILE block. However, when the final SELECT LIST statement is executed, the end-of-list flag is already set when the command is executed, so an error is generated and the command file is halted.

LIST.EMPTY.list_name

See an example on page 230.

This macro will be equal to "0" when components are in list *list_name*. Otherwise it is set to"1". This may be due to the fact that the XSELECT mode was off and that no components were selected when the LIST command was executed. Or components that were on the list when it was created have been deleted since the LIST command was executed.

LIST.LEN. *list_name*

This macro will contain the number of items added to the list. If components are deleted after the list was created, the value of the macro does **not** change.

LIST.INDEX. *list_name*

When a list is created by the LIST command, each component is assigned an index. The list is sorted by ID number (i.e. the component having the lowest ID number will have a list index of 1, the second lowest ID number is assigned an index of 2 etc.) The last has an index of *n*, where *n* is the number of components on the list. Deleting a component on the list (or otherwise removing it with a MERGE or GROUP command) does not affect the list indices of any of the remaining components on the list.

The LIST.INDEX. *list_name* macro holds current list index. For a new list, the value of the macro is 0. SELECT LIST commands change the value of the macro in the following ways:

SELECT LIST *list_name* FIRST sets the index macro to 1.

SELECT LIST *list_name* NEXT adds 1 to the index. Then the command tests to see that the component with this index exists. If it does it is selected. If it does not, the SELECT command adds 1 to the index and tries again. Once the index is greater than *n*, the LIST.EOL.*list_name* macro is set to 1.

SELECT LIST *list_name* LAST sets the index macro to *n*.

SELECT LIST *list_name* PREV subtracts 1 from the index. Then the command tests to see that the component with this index exists. If it does it is selected. If it does not, the SELECT command subtracts 1 from the index and tries again. Once the index is set to 0, the LIST.EOL.*list_name* macro is set to 1.

Functions Related to Lists

These functions are also covered in the section on functions on pages 244 and 230.

VALID_LIST_NAME()

This function will return "1" if the string in parentheses can be used as the name of a list. Otherwise it will return "0".

Example: **#VALID={VALID_LIST_NAME("MORE_THAN_16_CHARACTERS")}**

When this statement is executed, the value of the VALID macro will be set to "0". (Remember that curly braces are required around a function call, unless the function is called in the condition statement of an IF or WHILE command.)

LIST_EXISTS()

This function will return a "1" of the string in parentheses is the name of a previously created list. Other wise it returns "0".

Example: **IF (LIST_EXISTS("MYLIST")) {** !process list

When Lists Are Removed

Local lists are removed automatically when the command file in which they are declared is complete.

Global lists are removed automatically as soon as you EXIT, QUIT, or LEAVE the current cell.

Refer to the REMOVE command on page 203.

Lists can be deleted at any time with a REMOVE LIST #*list_name* command.

When a list is removed by any method, all macros created by the LIST command are deleted (e.g. LIST.EOL.*list_name,* etc.)

Efficiency of Using a List When Looping Through Components

You can use one of the methods described beginning on page 22 to execute a command file in all cells of your design.

Suppose that you need to perform individual processing on many components of your cell. You need to create a while loop that selects each relevant component one at a time. Using a SELECT ID=*int* command in the loop to select each component is inefficient. The entire component database is interrogated each time this command is executed to find the component with the required id. It is far more efficient to select each component from a list. See an example on page 185.

LOG *Speed command files by controlling how commands are logged.*

LOG OFF
or
LOG [SCREEN=(ON | OFF)] [LEVEL=(BRIEF | NORMAL | DEBUG)]

See an overview on control file efficiency on page 130.

The LOG command is used to control how commands in command files are logged (i.e. copied) to the journal file and echoed on the status line of the layout editor window. The primary use of these commands is to speed command file execution time.

The journal file is not only a record of what was done during the edit session, but it can also be used to recover from errors in the command file. To learn more about journal files, see page 128 or the IC Layout Editor Reference Manual.

When no LOG command overrides the default behavior, each of the commands in a command file will be echoed on the screen and logged in the journal file. In many command files, most the execution time is spent performing these command logs.

If you use the LOG OFF command as the first command in a command file, only the *@file_name* command will be logged in the journal file, and no commands will be echoed on the screen. When the LOG OFF command is used, it **must** be the first command executed in the command file. It is an error to execute a LOG OFF command outside of a command file. See **Effects of the LOG OFF Command** below to better understand how this command affects your ability to recover from errors.

The SCREEN keyword is used to control whether or not commands in the command file will be echoed on the screen display. See page 193. The LEVEL keyword is used to control what kind of commands will be logged. We will cover the different settings on page 194.

LOG [SCREEN=(ON | OFF)] [LEVEL=(BRIEF | NORMAL | DEBUG)] commands can be executed anywhere in a command file, and you can change the logging mode more than once in a command file. When a command file is completed, the LOG mode always returns the value it had before the command file began.

These commands can also be executed outside of a command file which will change the defaults for any command files executed during the session. In a session where no LOG commands have yet been executed, the defaults are:

```
LOG  SCREEN = ON  LEVEL = NORMAL
```

A LOG command with no parameters reports the current settings on the status line of the screen and in the journal file.

Effects of the LOG OFF Command

When the LOG mode is not turned off, ICED™ will insert comments in the journal file to indicate that control passed to a command file. Then each command in the command file that modified geometry or system settings will be logged into the journal file as well. The journal file will look similar to:

```
! @Q:\ICWIN\AUXIL\TEST.CMD
XSELECT OFF
SEL LAYER 100 ALL
DELETE
LEAVE
! Exit file Q:\ICWIN\AUXIL\TEST.CMD
```

Unless speed is a major concern, it is strongly recommended to leave the LOG mode on. The journal file is not only a history of what actions were taken with your design, but also a recovery mechanism if your system crashes during your session, or if you make a major mistake. It is best to have all commands executed logged into the journal file.

The only good reason to use LOG OFF is for speed during huge command files. Logging to the journal file does increase run time, but for a typical command file, the time increase is negligible.

We recommend that you restrict use of the LOG OFF command to large command files that do not create geometry that you may need to recover. The command files generated by the DRC (Design Rule Checker, available from IC Editors, Inc.) and NLE (Net List Extractor, part of the LVS tool available from IC Editors, Inc.) are good examples of command files where LOG OFF is appropriate. Since these command files generate only temporary geometry to indicate errors, you will never need to recover from an error in the command file.

Actions taken by command files that contain user interactions (e.g. a prompt for the user to supply a macro value or select a component) will not be automatically re-created by executing the journal file when the log mode is off. This is because the results of the user interactions will not be recorded in the journal file. **Do not use the LOG OFF command in these types of command files or you will prohibit automatic recovery of lost work with the journal file.**

If a command file that uses LOG OFF as the first command calls a nested command file, there is no way for the nested command file to turn the LOG mode back on.

If the LOG mode is on in command file, and a nested command file uses a LOG OFF command as the first statement in the file, the LOG mode will remain off until the nested command file is complete. The LOG mode in use by the first command file will then be back in effect.

Even when no LOG OFF command is included in a command file, you can turn logging off on the same line as the *@file_name* command. The LOG OFF command will then be executed as though it was the first command in the file.

Example: **@DRCOUT.CMD; LOG OFF**

The commands in DRCOUT.CMD will not be logged in the journal file. Only the call to the command file is logged to the journal file. In this case, the journal file will look similar to:

```
! @Q:\ICWIN\DRC\DRCOUT.CMD
! LOG OFF
@Q:\ICWIN\DRC\DRCOUT.CMD LOG=OFF
! Exit file Q:\ICWIN\DRC\DRCOUT.CMD
```

If this journal file is executed in a new ICED™ session to recover work, the command file will be called and re-executed. However, recovery of work done by that command file is not guaranteed. If the command file has been changed, the new version is executed. Or if a command file contains a SELECT command that requests the user to select a component, and the user does not select the same component he did the first time, the effect of the command file will be different the second time it is executed.

Showing Progress Messages During LOG OFF

The message in a RETURN command will not be displayed when the log mode is OFF or SCREEN=OFF.

The LOG OFF command also prevents commands from being echoed on the status line of the display screen. In a long command file, the session will appear "frozen" since no visible evidence that a command file is executing will appear on the display. The user may be confused by this, and even fail to realize when the command file is completed.

Adding $*comment* commands to the command file cannot help this situation, since they will be prevented from being displayed on the screen. However, if you prefix the comment with double dollar signs ("$$") then the comment will show on the screen and in the log file even when the log mode is off. This is a good way to show progress indicators in your command file. See the $*comment* command for more details.

Effects of the LOG SCREEN =(ON |OFF) Command

A LOG SCREEN=OFF command prevents all commands except for $*comment* commands from being echoed on the status line of the display. This command does not affect the logging of commands to the journal file.

Turning off the logging of commands to the screen display with this command can speed up a command file considerably, and recovery of a crashed session is not compromised as it is in the case of the LOG OFF command.

In order to see the speed increase associated with a LOG SCREEN OFF command, the view mode should also be off. (I.E. No VIEW ON command should be in effect.) When the view mode is on, commands will still be echoed on the command line as they are executed, despite the execution of the LOG SCREEN OFF command.

If the view mode is off, and you have turned off the echo of commands to the status line with a LOG SCREEN OFF command, the only commands displayed on the screen are $*comments*. You may want to add some $*comment* or $$*comment* commands to display progress reports. Otherwise, the user will see no visible evidence that your command file is executing.

Effects of the LOG [LEVEL=(BRIEF | NORMAL | DEBUG)] Command

Several commands can optionally produce !comments in the journal file. (These comments are not shown on the screen.) These comments provide extra information that can be of interest when determining exactly what happened during a command, but they will not affect how the commands in the journal file are executed. Adding these comments to the journal file takes extra time during the execution of a command file.

Type of command	**Example**	**LOG LEVEL**		
		BRIEF	**NORMAL**	**DEBUG**
Commands that alter geometry or alter system settings	ADD, MOVE, GRID, LOG	Command is logged	Command is logged	Command is logged
Commands that alter view window	VIEW IN	Command is logged	Command is logged and !comment reports new window	Command is logged and !comment reports new window
Macro definition or assignment	#I ={%I + 1}		!comment reports assignment	!comment reports assignment
Conditional execution	IF, WHILE			!comment reports if test was passed

Figure 28: Effects of log level on journal file for different commands

You can speed up a long command file by adding a LOG LEVEL=BRIEF command to your command file. This will prevent the generation of most !comments. This can decrease the size of the journal file considerably without affecting its ability to recover your work.

If you are having trouble debugging a command file, you can execute a LOG LEVEL=DEBUG command. This will add comments that help you follow exactly how the conditional statements were executed. The execution of conditional statement commands (IF, ELSE, ELSEIF, and WHILE) are not reflected in the journal file unless the LOG level is DEBUG.

Example:

```
LOCAL #FIRST_BOX $PROMPT="DIGITIZE FIRST BOX" BOX
LOCAL #I = 1
WHILE (%I < 4) {
        ADD BOX OFFSET={%I * (0.5, 0.5)} AT %FIRST_BOX
        #I = {%I + 1}
}
```

The lines above will add only 3 boxes, none of which will be at the location digitized by the user. Let us assume that this is not what the writer of the command file expected. If the lines above are executed when the log level is NORMAL (the default), the journal file will look like the following:

```
! LOCAL FIRST_BOX="(65.0,-55.0) (70.0,-50.0)"
! LOCAL I="1"
ADD BOX  LAYER=WELL  ID=163  OFFSET=(0.5, 0.5) AT (65.0,-55.0) (70.0,-50.0)
! LOCAL I="2"
ADD BOX  LAYER=WELL  ID=164  OFFSET=(1.0, 1.0) AT (65.0,-55.0) (70.0,-50.0)
! LOCAL I="3"
ADD BOX  LAYER=WELL  ID=165  OFFSET=(1.5, 1.5) AT (65.0,-55.0) (70.0,-50.0)
! LOCAL I="4"
```

It is not immediately obvious why the loop executed only 3 times since no record of the execution of the WHILE command is reported in the log file.

If a LOG LEVEL=DEBUG command is added before the lines are executed, the log will add !comments each time the test in the WHILE command is evaluated. The journal file will then look like the following:

```
! LOCAL FIRST_BOX="(60.0,-45.0) (65.0,-40.0)"
! LOCAL I="1"
! WHILE(1){ -- Begin block -- Line 4
ADD BOX  LAYER=WELL  ID=160  OFFSET=(0.5, 0.5) AT (60.0,-45.0) (65.0,-40.0)
! LOCAL I="2"
! } -- End WHILE block -- Line 7
! WHILE(1){ -- Begin block -- Line 4
ADD BOX  LAYER=WELL  ID=161  OFFSET=(1.0, 1.0) AT (60.0,-45.0) (65.0,-40.0)
! LOCAL I="3"
! } -- End WHILE block -- Line 7
! WHILE(1){ -- Begin block -- Line 4
ADD BOX  LAYER=WELL  ID=162  OFFSET=(1.5, 1.5) AT (60.0,-45.0) (65.0,-40.0)
! LOCAL I="4"
! } -- End WHILE block -- Line 7
! WHILE(0){ -- Skip to end of block -- Line 4
! } -- End of block -- Line 7
```

Now the writer can see exactly what the value of the macro I was during each execution of the loop. The line numbers mentioned in the !comments about the WHILE block refer to the line number of the WHILE statement in the command file.

MARK_SUBCELLS *Initialize SUBCELL.EDIT system macro*

MARK_SUBCELLS [LAYERS=*layer_list*]

The cell table is the list of all cells currently open in the layout editor. The list includes all subcells of explicitly opened cells. See page 104.

This ICED™ command is used to initialize the data in the CELL.DEPTH.*cell_index* and SUBCELL.EDIT.*cell_index* system macros. (See pages 259 and 296.) It will traverse the cell table and determine the cell hierarchy tree for all cells. This information is used to determine the values of the indicated system macros.

For the CELL.DEPTH.*cell_index* macros, MARK_SUBCELLS will store for each cell the depth of cells nested inside of it. The *layer_list* has no effect on the values stored in these macros.

For SUBCELL.EDIT.*cell_index* system macros, MARK_SUBCELLS will store an edit code for each cell. Only subcells of the current cell will have a non-zero code. The edit code of a cell is determined by the type of cell library in which it is contained. The SUBCELL.EDIT.*cell_index* macro will return different values for cells that can be edited and saved vs. cells that are in read-only libraries.

See some examples of layer lists on page 257.

If you prefer to restrict the list of cells marked with non-zero SUBCELL.EDIT.*cell_index* system macros to those cells that contain shapes on specific layers, add the LAYERS keyword followed by the list of layers in layer list syntax. (See page the IC Layout Editor Reference Manual for complete layer list syntax and examples.)

Example:

See page 126 to lean about incrementing counter macros like N in this example.

```
MARK_SUBCELLS  LAYERS=METAL1+METAL2
WHILE (%N < %MAX.CELL){
        IF (%SUBCELL.EDIT.%N >0) {
                !process cell
```

The possible values of the SUBCELL.EDIT.*cell_index* system macros are listed on page 296.

This use of the MARK_SUBCELLS command will initialize the data in the SUBCELL.EDIT.*cell_index* system macros so that they will contain non-zero numbers only for cells that meet both of the following criteria:

- the indicated cell must be nested in the cell hierarchy of the current cell , and
- the cell must directly contain shapes on layer METAL1 or METAL2.

When the N macro in the example above contains the index of a cell that meets both of these criteria, then the code inside of the IF block would be executed.

Note that a subcell must contain shapes on the indicated layer directly. A cell can contain a marked subcell without being marked itself.

The data stored in the system macros by this command is **not updated automatically**. If you run the MARK_SUBCELLS command, then add or delete a subcell from the current cell, then get the edit status for that subcell from the SUBCELL.EDIT.*cell_index* macro, the value returned will not reflect that the status has changed since the MARK_SUBCELLS command was executed.

See another example of this command in the SUBCELL.EDIT description on page 296.

The information stored by MARK_SUBCELLS is discarded when you terminate the editor. It is not saved with the cell file.

PAUSE — *Create a pause in a command file.*

PAUSE [*seconds*] [" [+ | - | *] *msg_string*"]

The PROMPT command can be used to temporarily replace the message in the prompt area of the command line.

The PAUSE command is used to insert wait states in command files. It is usually used to give a command file user a chance to see a message. (The command will have no effect if typed in during an ICED™ session except that it will be logged into the journal file.}

Use quotes around the *msg_string* when it can be misinterpreted. For example, when the string begins with a number that can be interpreted as the *seconds* parameter.

If used, the *seconds* parameter determines how long ICED™ will be frozen. The *seconds* parameter must be an real number in the range 0:60. When you omit *seconds*, or specify '0', then the command file is paused until the user hits a key or mouse button. A prompt indicating this replaces the command prompt near the bottom of the screen.

Example:

```
$ This message is left on history line during pause
PAUSE
```

These lines will leave the $*comment* displayed on the history line until the user presses a key or mouse button. The menu is not displayed. Once the user has interrupted the pause, the execution of the command file continues.

When you supply a *msg_string*, it is displayed on the command line during the wait. You can add one of the prefixes in Figure 29 to select the color used to display the message.

Prefix character	*msg_string* color
+	green
-	red
* or none (default)	white

Figure 29: PAUSE message colors

Example: **PAUSE This message is drawn in white on the command line.**
PAUSE -This message is drawn in red on the command line.

If *seconds* is non-zero, the command file will be paused for that length of time before the command file automatically continues. The user can interrupt the wait by clicking the mouse or typing any key on the keyboard. However, no message informing the user of this is displayed unless you supply an appropriate *msg_string* in the command.

Example: **$ This message is left on history line during pause**
PAUSE 10 Press key or mouse button to continue immediately.
$ This message is displayed on history line after pause is done

These lines will display the first two messages for 10 seconds. A countdown in yellow replaces the command prompt. When the countdown gets to 0, the execution of the command file continues.

The command parser performs macro replacement in the *msg_string.* You should surround the *msg_string* with quotes if it uses syntax that may cause parsing conflicts.

Example: **PAUSE "%MYMAC errors found"**

Before this command is executed, the value of the MYMAC macro will replace the %MYMAC reference in the message string. Since the example above omits the *seconds* parameter, if MYMAC contained a number, and the quotes were not present, the parser would interpret the number as the *seconds* parameter of the PAUSE command.

Updating the Display Window

Older versions of ICED™ required that you change the VIEW mode before a PAUSE command to force the display to be updated. Versions newer than 4.86 no longer require this. The display is now automatically updated before the pause no matter what the VIEW mode is.

PROMPT — *Change prompt message on the command line.*

PROMPT ="*string*"

In older versions of ICED™, the VIEW ON command was required for the user to see the prompt. This is no longer the case.

This command is used to replace the usual prompt string to the left of the '>' on the command line in the layout editor window. The normal "*Layer_id*; Sel=*n*; *cell_name*" prompt will temporarily be replaced with the *string* specified in the PROMPT command until after the next command is completed. The prompt *string* will be displayed in an easy-to-notice yellow color.

Keep *string* as brief as possible. Depending on the size of the screen, a prompt of more than 35 characters may fill more than half of the command line. If *string* is longer than 50 characters, only the first 50 characters will be displayed.

See the $*comment* and PAUSE commands to display an ordinary message to the user.

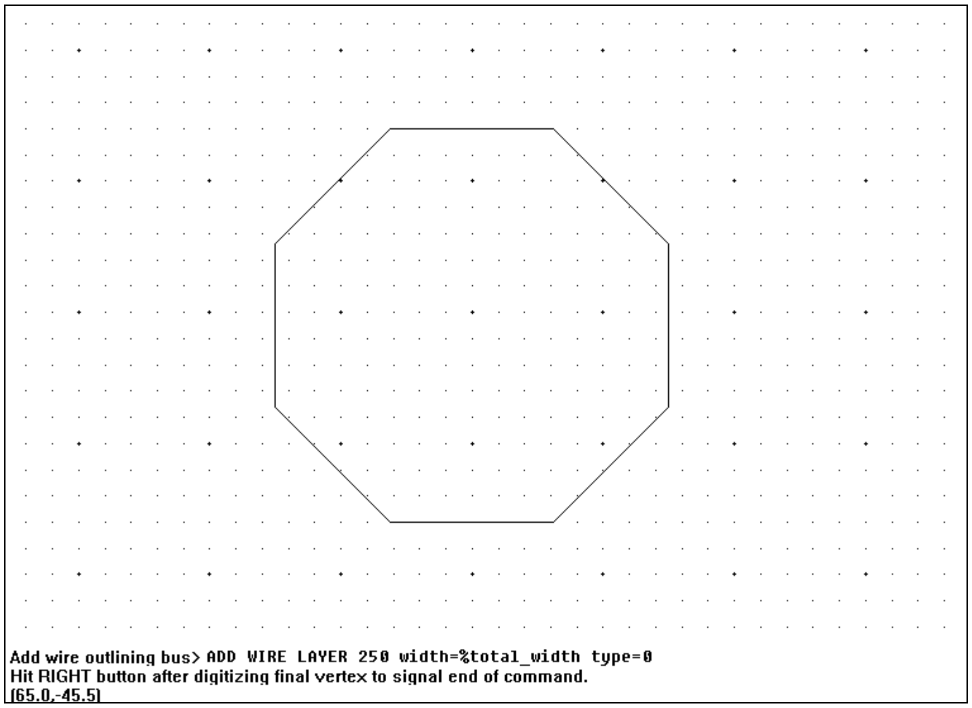
The PROMPT command is particularly useful prior to commands that have the user interact with the mouse. SELECT commands, commands that issue an embedded SELECT command (e.g. COPY), and ADD commands are all much more likely to have the desired result when you replace the prompt so that the user understands what to do.

The prompt returns to normal after one command is executed. Therefore, do not include any command between the PROMPT command and the user interaction command during which you want the prompt displayed.

Example:

```
PROMPT "Add wire outlining bus"
ADD  WIRE  LAYER 250  WIDTH=%TOTAL_WIDTH  TYPE=0
```

Let us assume that the command file fragment above comes after a local macro with the name TOTAL_WIDTH has been defined. The indicated prompt will be displayed on the screen while the ADD WIRE command is waiting for the user to digitize the wire. The screen will look similar to the image shown on the next page.



```
Add wire outlining bus> ADD WIRE LAYER 250 width=%total_width type=0
Hit RIGHT button after digitizing final vertex to signal end of command.
(65.0,-45.5)
```

Figure 30: ICED™ screen with command line prompt replaced.

REMOVE — *Delete macros*

REMOVE [#]*macro_name_string*
or
REMOVE LIST [#]*list_name_string*

Use this command to delete user-defined macros. Macros defined with the GLOBAL keyword will remain in storage until the end of your current session unless you explicitly delete them with this command. (Remember that macros defined with the LOCAL keyword are removed automatically at the end of the command file in which they are defined.)

Example:

```
GLOBAL #MY_MAC=0
GLOBAL #SUCCESS = 0
        !missing statements that process data and set SUCCESS to
        ! non-zero number if the command file is successful.
IF (%SUCCESS){
        REMOVE #MY_MAC
}
```

This command file defines a global macro named MY_MAC. Let us assume that the missing statements assign a value to MY_MAC that might help you figure out what went wrong if the command file is not successful. You want to be able to look at the final value of MY_MAC only if the command file is not successful. If the command file is successful, this macro will be deleted.

Example:

```
REMOVE LIST=MYLIST
```

The command above will delete a list with the name "MYLIST". Note that the '#' is not required before *name_string* in a REMOVE command. (We still recommend its use for consistency with other commands where it is required)

Wildcard characters can be included in the name string. This allows you to remove more than one macro or list with a single REMOVE command.

*	This wildcard character can be used only once in the string. All macros that have names matching the name string when zero or more characters replace the * will be deleted.
?	This wildcard character means that any single character can replace each '?' to match macro names. You can use multiple '?'s in one name string.

Example:

```
REMOVE #*MAC
REMOVE #MY_*
REMOVE #???MAC*
```

Any of these commands will remove a macro with the name MY_MAC.. Many other macro names may also match each of these *macro_name_string*s. The last command will also remove macros with the names "ABCMAC", "XXYMAC", and "XXYMAC1234". However, a macro with the name "ABMAC" would not be removed because it has only two characters before the "MAC".

Example:

```
REMOVE #RES*
```

The command above is used in the RES.CMD example on page 305 to delete all global macros created by the command file. It is a very good practice to name all of the macros in a command file with the same prefix string. One good reason is that you can delete them all with a single REMOVE command similar to the one above.

Example:

```
REMOVE *
REMOVE #*
```

Either command above will delete all user-defined macros. **This includes keyboard macros defined with the KEY command.** However, this command will **not** delete any lists. The LIST keyword must be included to delete a list.

It is not an error to delete a list or macro that does not exist. A command file will not be halted with an error message even when a REMOVE command has no effect. Removal of a system macro will have no effect.

RETURN — *Terminate command file or exit shell state*

RETURN ["*msg_string*"]

Recent versions of ICED™ use this command to return from a shell. See the next page.

This command immediately ends a command file. It is usually used in combination with an IF, ELSEIF, ELSE or WHILE command to end a command file before the end of the file if certain conditions are true. Since a command file will automatically return after the last statement in the file is executed, a RETURN command is not required at the end of a command file.

Example:

```
LOCAL #FAILURE = 0
.         !missing statements that process data and set FAILURE to
.         ! non-zero number if the commands are unable to perform their function.
IF (%FAILURE){
          RETURN "FAILURE: return code = %FAILURE"
}
```

The ERROR command can also be used to immediately end a command file with an error message.

Since the %FAILURE macro reference in the RETURN message is executed before the command file is terminated, the value of the local macro FAILURE is reported in the return message before it is deleted. This command will leave the indicated comment displayed near the bottom of the window below the prompt line when control is returned to the layout editor.

When you add an error handler block, precede it with a RETURN command so that it does not get executed when command file is successful. See page 165.

The return message is optional. The return message will not be visible when either a LOG OFF or LOG SCREEN OFF command is in effect. The return message will not be recorded in the journal file when the LOG OFF command is in effect.

A RETURN command in a nested command file will terminate only the command file in which it is located. Control will then pass to the statement after the *@file_name* command that called the nested command file.

The next example uses RETURN in a conditional statement to exit the command file from within an infinite loop. The command file terminates automatically after all components in a list have been processed.

Example:

See page 182 to learn about lists.

```
WHILE (1) {
  UNSEL ALL; SELECT LIST #tmp NEXT   ! Select next component on list
  IF(%N.SELECT==0) RETURN            ! Nothing left on list, we're done
        !missing processing statements that process selected component
}
! statements after WHILE block would never get executed
```

Returning from a Shell

The RETURN command is also used to return from a temporary command shell. You can type the RETURN command at the command prompt, or use it as a menu option in a special menu, to return control to the command file that created the shell. Refer to the SHELL command on page 207.

SHELL *Suspend command file and execute interactive commands.*

SHELL [MENU=(*menu_name* | *)[:*submenu_name*]] [NAME=*shell_name*]

See AUXUL\-_LOOP2.CMD for a sample command file that uses a shell.

This command is used to suspend the current command file and return control to the user interactively. The user can execute any commands with any of the standard methods (including executing other command files). When the user is finished with the shell, he should execute a RETURN command to return control to the command file at the statement after the SHELL command.

Example:

PAUSE "Press <Enter>, select the desired components, & then type 'RETURN' to continue."
SHELL

The DOS or SPAWN command opens a DOS console shell. See the IC Layout Editor Reference Manual.

When the commands above are executed, the command file is suspended and the user can interact in the editor as though no command file is currently executing. The only evidence that a shell has been entered is that the string "Shell=1" is added to the beginning of the command prompt. The user must type "RETURN" at the command prompt to close the shell and return to the command file.

If the optional MENU parameter is specified, the menu *menu_name*.MEN is loaded before control is given to the user. Do not specify the ".MEN" file extension in the *menu_name* parameter.

Example: **SHELL MENU=MYMENU**

To learn more about menus, see the MkMenu utility in the IC Layout Editor Reference Manual.

When the command above is executed, the effect is the same as the first example, except that the menu MYMENU.MEN is loaded before the command file is suspended. The original menu is restored when the user returns to the original command file.

The RETURN command can be assigned to a menu option in your menu file. In this case, the user can return to the command file by selecting the option from the menu.

Example: **SHELL MENU=MYMENU:MYSUB**

See DRCTAG.CMD and SHOW1.CMD in the AUXUL directory for sample SHELL commands that use menus and submenus.

If the file MYMENU.MEN defines a submenu "MYSUB" using lines similar to those below, that submenu is selected before the command file is suspended.

```
#unused mysub 2
* 0  1  "DELETE"  "delete" RET
-12  1  "RETURN"  "return" RET
#end
```

Example: **SHELL MENU=***

For an example of a custom menu source file, see SAMPLES\-DRC.DAT, the source file for AUXIL\-DRC.MEN.

When you have already loaded a menu and selected a submenu, this use of the SHELL command begins a shell with the default top-level menu selected.

When the optional NAME keyword is used and another shell with the same name is still active, ICED™ will report an error and halt the command file. Use this form of the SHELL command to prevent a user from accidentally launching a shell from within the same shell.

Example: **SHELL NAME=MYSHELL**

Suppose the SHELL command above is included in a command file X.cmd. If the user types @X, and opens the shell, then types @X at the prompt while the shell with the same name is already open, then an error is reported in red at the bottom of the window and rest of the second call to X.CMD is not processed. However the first call to X.CMD is not interrupted and the first shell remains open.

When the NAME keyword is not used, nothing prevents you from opening a succession of shells. The number of open shells is reported in the command prompt. You can have no more than 8 open shells.

SKIP_TO

Force interpreter to go directly to a specific line.

SKIP_TO *label_name*[:]

Use the BACK_TO command instead to search backward for a labeled statement.

The SKIP_TO command causes the command interpreter to keep skipping whole command lines in the current file until it reaches a statement with label *label_name*. Skipped commands will not be processed at all. Macro references will not be replaced. Even syntax mistakes in the skipped lines will not halt execution.

The simplest way to label a statement is to type a label string followed by colon ":". More details on label syntax are provided on page 32.

Example:

```
SKIPTO JAIL
$Passing GO, Collect $200       !this will not be executed
JAIL:
! lines following the label will be executed.
```

The SKIP_TO command is especially useful as an error handler in the ERROR.CMD macro. See page 165.

The SKIP_TO command above will cause the command interpreter to ignore all succeeding lines until it processes the statement labeled "JAIL". The $*comment* command will not be executed. Note that the '_' can be omitted from the SKIP_TO keyword.

Avoid labeling statements inside of WHILE, IF, or ELSEIF blocks. While it is not an error to label statements inside of a block, skipping into a block can have unforeseen consequences.

VIEW (ON | OFF) *Control display refresh during command files.*

VIEW (ON | <u>OFF</u>)

Outside of command files, ICED™ updates the display after each command it executes. The display update usually takes most of the time required to execute the command. The VIEW ON and VIEW OFF commands allow you to enable or inhibit the display update during the execution of command files. The VIEW OFF mode, which inhibits display update during command files, results in an order of magnitude decrease in execution time. This is the default for new cells.

See the LOG SCREEN commands to control the update of the status line during command files.

When the view mode is OFF, commands in a command file will not be displayed on the command line near the bottom of the window while they are being executed. If geometry is created or modified, this will not be reflected in the display until the command file is completed, or until a command is executed while the VIEW mode is set to ON. ($Comments are seen on the status line in either view mode.)

Commands in a command file that require the user to select or digitize components temporarily set the view mode set to ON. You no longer need to add VIEW ON commands to ensure that the view is updated correctly during these types of commands.

Example:

```
VIEW OFF
WHILE (%COUNT <=32000){
            !missing processing that creates components
        $ %COUNT components added.  Press both mouse buttons to abort.
        PAUSE
        #COUNTER = {%COUNTER + 1}
}
```

See page 162 to see an example of posting a message every 100th time through a loop.

The command file fragment above consists of a while loop that may execute 32000 times. The block of lines in the middle allows the user to see the progress of the command file and abort it if it is not performing as expected. Even when the VIEW mode is OFF, the display will be updated automatically to reflect the current design by the PAUSE command.

When ICED™ executes a VIEW ON or VIEW OFF command outside of a command file, the default view mode for all command files executed in the current session is modified. When one of these commands is executed in a command file, it affects the view mode only during the current command file. In either case, this command does **not** affect the automatic refresh after every command entered from the keyboard or menus.

When ICED™ executes a command file, it begins execution with the current value of the view mode. This value remains in effect until it is changed by another VIEW ON or VIEW OFF command. When ICED™ exits a command file, it resets the view mode to what it was before entering the command file. Thus, a VIEW ON or VIEW OFF command in a command file affects only the view mode for the current command file and any command files it calls.

There are two other commands that begin with the keyword VIEW. The VIEW command is used to alter the view window. The VIEW LIMIT command is used to speed screen refreshes by controlling which components are drawn. Learn more about these commands in the IC Layout Editor Reference Manual.

WHILE

Execute block of statements more than once

```
WHILE ( boolean_expression ) {
        !Block of statements
      }
```

A complete explanation of Boolean expressions is provided on page 53.

More examples of WHILE commands are provided in the overview of conditional statements beginning on page 90.

This command will continue executing a block of statements as long as the Boolean expression evaluates to TRUE (i.e. a non-zero number).

The Boolean expression is evaluated before the block of statements is executed for the first time. If the expression is FALSE (i.e. 0), control will pass to the next statement past the '}'. If the expression is TRUE, the statements in the block will be executed and then the Boolean expression will be evaluated again.

The most common use of a WHILE loop uses a counter to perform the loop a certain number of times. Create the Boolean expression to test that the counter is below or equal to a maximum number. You can include other conditions in the Boolean expression to terminate the loop early if necessary.

Example:

See a list of comparison operators on page 54.

The "&&" Boolean And operator is explained on page 55.

```
GLOBAL #COUNTER=0
LOCAL  #FAILURE = 0
WHILE ((%COUNTER < 10) && (%FAILURE == 0)){
    .
    .  !missing statements that set FAILURE to non-zero number
    .  !if the loop is unable to perform its function.
    .
    #COUNTER = {%COUNTER + 1} !Don't forget to increment the counter
}
```

Always type a blank before the condition expression in ( ).

This WHILE loop will execute the block of statements 10 times, unless the FAILURE macro gets set to a non-zero number. **The statement that increments the counter is critical. If you forget to include it in the loop block, the loop will continue indefinitely.** We refer to a loop that will never terminate as an infinite loop.

A common mistake with WHILE loops is to forget to add curly braces {} to the statement that increments the counter. In this case the program will perform string manipulation rather than evaluate a mathematical expression.

Example:

```
WHILE ((%COUNTER < 10) && (%FAILURE == 0)){
    .
    .
    .
    #COUNTER = %COUNTER + 1            !OOPS, forgot the {}
}
```

Many more examples of Boolean expressions are located in the IF command description beginning on page 168.

In this case, the first time through the loop the COUNTER macro will be set to:
"0 + 1".
The next time through the loop, it will be set to:
"0 + 1 + 1"

When the mathematical expression is surrounded by curly braces (as shown in the example on the previous page), the expression will be evaluated and COUNTER will be incremented as expected.

Note that in the example above, no "{}" are used to surround the *boolean_expression* "(%COUNTER < 10) && (%FAILURE == 0)". This is because ICED™ will always interpret the *boolean_expression* in an IF, ELSEIF, or WHILE command as an expression. The "{}" are not required.

The WHILE condition can be created so that the block of commands is executed once for each component or subcell. See examples on pages 269, 186 and 297.

Recovery using the journal file is explained in the IC Layout Editor Reference Manual.

If your WHILE loop does not require a counter, you should add some other mechanism to avoid an infinite loop. Infinite loops require that you terminate your edit session with a <Ctrl><Alt><Delete> combination (or a close window operation.) The cell files will not be saved. You must recover other work performed during the session with the journal file.

Example:

The TIMER system macro contains the time in seconds from the start of the current session. See page 298.

```
LOCAL #ERROR=0
LOCAL #COUNTER = 0
LOCAL #TIME = 0
LOCAL #TIME_START = %TIMER

WHILE((%TIME < 20) && (%ERROR==0) && (%COUNTER< 10)) {
        .
        .       !statements to perform a task that should complete in 20 seconds
        .
        #TIME = {%TIMER - %TIME_START}
}
```

The statements in the WHILE block will not be executed if any of the expressions separated by "&&" AND operators evaluate to FALSE. Even if you forget to increment the COUNTER macro, and never set ERROR to a non-zero number, this WHILE loop will always terminate after 20 seconds.

Another method to terminate an infinite loop is to add a RETURN statement in an IF block. This will terminate not only the loop, but the entire command file. See an example on page 206.

Other examples of WHILE loops appear on pages 94, 180, and 260.

XSELECT *Enable or disable embedded selects in command files.*

XSELECT (ON | OFF)

The XSELECT command can be used to enable or disable embedded select commands in command files. (An embedded select command is generated when you execute a command that usually operates on selected components, but no components are currently selected.) The XSELECT mode is on by default, this enables embedded select commands.

Consider the following command file fragment:

Example:

```
UNSELECT ALL
SELECT LAYER "JUNK" ALL
DELETE
```

If there are no components on layer JUNK, no components are selected when the DELETE command is executed. If the XSELECT mode is on, the DELETE command will execute an embedded SELECT NEAR command, bring up a near box, and wait for you to select something. The only way to bypass the SELECT NEAR command would be to cancel the command, and this would cause ICED™ to issue an error message and terminate the command file with any remaining commands unprocessed.

Example:

```
UNSELECT ALL
SELECT LAYER "JUNK" ALL
XSELECT OFF
DELETE
```

In this example, the embedded select command is disabled since the XSELECT OFF command is included. Thus the DELETE command will do nothing when no components are selected. No error is generated and the remainder of the command file is processed. The embedded select commands will remain disabled until you exit the command file or execute XSELECT ON.

Holding down the <Shift> key during a embedded SELECT command allows the user to select multiple components. Release <Shift> before selecting the last component, or click in empty area, to complete selection and return to the command file.

The XSELECT mode has no effect on ICED™'s behavior when executing commands from the keyboard or menus. However, if you execute a XSELECT OFF command outside of a command file, embedded select commands will then be disabled by default within all command files. This can be overridden within a specific command file with a XSELECT ON command.

When you come to the end of a command file, ICED™ forgets the current XSELECT mode and restores the mode in effect when you entered the command file.

This following paragraph is for your information only. You may have noticed XSELECT statements in journal files. This form of the XSELECT command is intended for use in these files only. Other uses are **not** supported!

When you execute a command that generates an embedded select command, ICED™ must record the select information in the journal file. The following journal entry represents the embedded select command associated with a MOVE SIDE command.

```
XSELECT  SIDE  IN (-41.0, 27.5) (-39.0, 29.5)
MOVE SIDE BY  (0.5, 6.0)
```

Functions

See an overview on the use of functions on page 63.

ICED™ supports many functions. When you call a function in a statement, ICED™ will replace the function call string with the value returned by the function. A function is called with the syntax:

> ***function_name*(*arg1*)**
> or
> ***function_name*(*arg1*, *arg2*)**

The function arguments, *arg1* and *arg2,* can be numbers, expressions, strings, or macro references. When macro references are used, the value of the macro is substituted before the function call is evaluated.

Example: **#VERTEX = {POS1(%VIEW.BOX)}**

This use of the POS1 function is a typical use of a function. The POS1 function returns the first coordinate pair of a list of coordinate pairs. VIEW.BOX is a system macro that contains 2 coordinate pairs that represent the corners of the current view window. This statement will result in the user macro VERTEX storing the coordinate pair of the lower left corner of the view window.

There are two important syntax restrictions demonstrated in the example above:

- **Do not insert a blank between a function name and the '(' character.**
- **Surround any function call in a macro assignment with curly braces, '{}'s.**

Do not omit '_'s when typing function names like DIR_EXISTS.

The following table lists every function currently supported by ICED™. They are grouped by purpose. Longer explanations and examples follow on the pages indicated.

Functions Sorted by Purpose

Category	Function name	Purpose	Pg.
Value validation	MACRO_EXISTS	Test if macro is defined	231
	DEVICE_EXISTS	Test if device (e.g. printer) exists	226
	LIST_EXISTS	Test if a specific list has been created	230
	DIR_EXISTS	Test if file directory exists	226
	FILE_EXISTS	Test if file exists	227
	FILE_TIME	Returns last modification time of file	228
	STD_COORD	Format coordinate string	240
	VALID_INT	Test that string is a valid integer	242
	VALID_REAL	Test that string is a valid real number	245
	VALID_LAYER	Test that string is a valid layer name	243
	VALID_ITEM_NAME	Test that string is a valid name for ITEM command	243
	VALID_LIST_NAME	Test that string is a valid name for LIST command	244
	VALID_CELL_NAME	Test that string is valid as a cell name	241
Coordinate manipula-tion	ROUND ROUND1 ROUND2	Round single coordinate or coordinate pair to resolution grid	235
	POS1	Returns first coordinate pair in list	233
	POS2	Returns second coordinate pair in list	234
	POSN	Returns *n*th coordinate pair in list	234
	X	X-coordinate of coordinate pair	246
	Y	Y-coordinate of coordinate pair	247
	X0	First x-coordinate of list of coordinate pairs	246
	Y0	First y-coordinate of list of coordinate pairs	247
	X1	Second x-coordinate of list of coordinate pairs	247
	Y1	Second y-coordinate of list of coordinate pairs	248

Continued on next page

String manipula-tion	CMP	Compare two strings, case independent	224
	XCMP	Compare two strings, case dependent	224
	CHAR	Returns *n*th character in string	223
	LEN	Find length of string	229
	CHAR_TO_N	Returns integer from ASCII character code	224
	N_TO_CHAR	Returns a single ASCII character from integer	233
Mathe-matical operations	SQRT	Square root of number (use positive numbers only)	239
	INT	Integer part of real number	229
	ABS	Absolute value of single number or vector length of coordinate pair	220
	SIN	SINE of angle in degrees	239
	COS	COSINE of angle in degrees	225
	TAN	TANGENT of angle in degrees	241
	ATAN	ARCTANGENT of single value or coordinate pair	221
	MIN	Minimum of two single numbers	232
	MAX	Maximum of two single numbers	232
Cell information	CELL	Provides cell table index given a cell name	221

Figure 31: ICED™ Functions

Functions Alphabetically

ABS(*val*)

or

ABS(*x_coord,y_coord*)

When used on a single number, this function will return the absolute value of the number. That means that if the number is negative, it is multiplied by –1.

When used on a coordinate pair in the form "(*x-coord,y-coord*)", this function will return the length of the vector from (0,0) to the point represented by the coordinate pair. The return value will be equal to:

$$\sqrt{x\text{-}coord^2 + y\text{-}coord^2}$$

Expression	Result
{ABS(3)}	3
{ABS(-3.2)}	3.2
{ABS(3,3)}	4.242640687
{ABS(-3,-3)}	4.242640687

ATAN(*val*)
or
ATAN(*x_coord,y_coord*)

When used on a single number, this function will return the arctangent of that number. When used on a coordinate pair, the arctangent of *y-coord* / *x-coord* will be returned. The result will be expressed in degrees.

Example:

```
#VAL = {ATAN(1)}
#VAL = {ATAN(4,4)}
```

When either of these statements are executed, the macro VAL will be assigned the number 45.

CELL("*cell_name*")

Refer to an overview of cell table indices on page 252

This function returns the cell table index of the cell with the name *cell_name*. This index can be useful when using cell related system macros. It will also tell you if a cell with that name is already defined in any cell library.

Example:

See another example of the CELL function on page 242.

```
LOCAL #CELL_NAME = $PROMPT "Enter the cell name:"
LOCAL #CELL_INDEX = {CELL('%CELL_NAME')}
IF (%CELL_INDEX > 0) {
        $The bounding box of cell %CELL_NAME is &
        %CELL.BOX.%CELL_INDEX
}
ELSE   $Cell %CELL_NAME is not loaded
```

The statements in the previous example use the CELL.BOX.*cell_index* system macro (see page 258) to report the bounding box of the cell with the name typed by the user. This system macro requires the cell index to refer to the cell.

Note the "{}" around the call to the CELL function. If the "{}" were not included, CELL_INDEX would be set to the string "CELL('MYCELL')" instead of the result of the function call.

To get the cell name from the cell table index, use the CELL.NAME-.*cell_index* system macro. See page 264.

If cell *cell_name* is in the cell table, (i.e. the cell has been loaded in the current editor session) the function will evaluate to a positive non-zero integer that represents it's position in the cell table. If the cell is not in the table, one of the following integers will be returned:

0	The cell is not currently loaded, but the cell file does exist on a directory in the cell file search path.
-1	The cell is not loaded, and no cell file with that name exists in any current cell library.

Any quote character may be used ",',~, or `.

The use of quotes around the *cell_name* parameter is strongly recommended. If quotes are not used, and *cell_name* is a macro reference that may contain the null string, or special characters, the parser may be unable to evaluate the function call properly and the command will fail. However, if quotes are used, the command will not fail even when *cell_name* evaluates to the null string. The return value of the function will just be –1.

Example:

The LOCAL_COPY = TRUE parameter allows you to edit cells from copy-edit libraries without a warning prompt.

```
LOCAL #CELL_NAME = $PROMPT "Enter the cell name:"
LOCAL #CELL_INDEX ={CELL('%CELL_NAME')}
IF (%CELL_INDEX == 0) {
        EDIT CELL %CELL_NAME LOCAL_COPY=TRUE    ! load cell
        EXIT
        #CELL_INDEX ={CELL('%CELL_NAME')}
}
IF (%CELL_INDEX <= 0) {
        PAUSE "Unable to load cell %CELL_NAME"
}
```

This example will try to load the cell if it is not already loaded. If the user simply hits return so that CELL_NAME = "", then the command file will not fail, it will simply execute the PAUSE command in the second IF block.

To test the protection level of a loaded cell, see the CELL.EDIT.-*cell_index* system macro on page 261.

If the user types an cell name that has been loaded, neither IF block will be executed. If the user types a cell that is not loaded, but exists in one of the cell libraries, it will attempt to load the cell. However, this example will not work if the cell is stored in a read-only cell library; the EDIT command will fail.

CHAR(*n*, "*string*")

This function returns a string consisting of the *n*th single character from the indicated string.

Example:

```
{CHAR(1,"fred")}
```

This expression will resolve to the string "f".

ICED™ does not currently have a function to return a substring, but this function can be used to accomplish the same thing.

Example:

```
LOCAL  #LONG_NAME = LONGNAME
LOCAL  #SHORT_NAME = ""
LOCAL  #N = 1

WHILE (%N <= 4) {
    #SHORT_NAME= %SHORT_NAME^{CHAR(%N,%LONG_NAME)}
    #N = {%N + 1}
}
```

The example above will place the first 4 characters of the string contained in the LONG_NAME macro in the SHORT_NAME macro.

CHAR_TO_N("*char*")

Do not omit the '_' when typing this function name.

Use this function to get the decimal integer ASCII character code for the single character *char*. Quotes are required around the character only when it is a special character that might confuse the parser.

Example: **#ASCII_VAL = {CHAR_TO_N("%")}**

The value of ASCII_VAL will be "37". Note the curly braces that force evaluation around the function call.

CMP(*string1*,*string2*) and XCMP(*string1*,*string2*)

Both of these functions return 0 if the two strings contain the same text. The CMP function translates both strings to upper case before they are compared. Therefore case is irrelevant for the CMP function. However, case is considered by the XCMP function that returns 0 only when the strings are e**X**actly the same.

Example:

{CMP("hello", "hello")}
{CMP("hello", "HELLO")}

Both expressions above will resolve to 0.

Example:

{XCMP("hello", "hello")}
{XCMP("hello", "HELLO")}

The first of these two expressions will resolve to 0. However, the second will resolve to a non-zero number since the case of the letters is different.

The strings are compared letter by letter. Leading or trailing blanks are not stripped before the comparison.

Example: **{XCMP("hello", " hello")}**

The expression above will resolve to a non-zero number.

See an overview of Boolean condition expressions on page 53.

These functions are usually used in Boolean condition expressions in WHILE, IF and ELSEIF statements. It is important to remember that this function returns 0 (or FALSE) when the strings are identical. (This conforms to the C programming convention for string comparison. A number less than 0 is returned when the first character of *string1* comes before the first character of *string2* in alphabetical order. A number greater than 0 is returned when the first character of *string1* comes after the first character of *string2*. 0 is returned to indicate that the strings are identical.) Always use this function in a condition expression in a manner similar to:

Example:

```
IF (CMP(%RES.MODE, "HARD")==0){    !TRUE if RES.MODE = HARD
        ⋮
}
```

This example will execute the statements in the IF block (the statements between the curly braces) when the system macro RES.MODE indicates that the resolution mode is HARD.

If you instead write the IF condition expression in the following manner:

Example:

```
IF (CMP(%RES.MODE, "HARD")){         !FALSE if RES.MODE = HARD
        ⋮
}
```

then the statements between the curly braces would be executed only when the RES.MODE macro is **not** equal to the string "HARD".

COS(*angle*)

This function will return the cosine of an angle expressed in degrees.

Example: **#VAL = {COS(30)}**

When this statement is executed, the user-defined macro VAL will be assigned the number 0.86602540378.

DEVICE_EXISTS("*dev_name*")

Do not omit the '_' when typing this function name.

This function is used to test if a device (e.g. a printer) is defined to the operating system. It will return TRUE ("1") if the device is found, and FALSE ("0") if the device cannot be found.

Example:

```
IF(DEVICE_EXISTS("%VALUE")==0){
        PAUSE "Invalid device <%VALUE> -- Device does not exist."
}
```

If the VALUE macro contains a valid device name (e.g. "LPT1"), then the IF block will not be executed.

The example above is based on the code in the command file _ GET_DEV.CMD. This command file (supplied with the installation) prompts the user for a device name and then validates the user response with the DEVICE_EXISTS function.

You should use quotes around the value or macro reference you are using as the function argument. Many device names contain blanks or slashes that may confuse the parser if quotes are not used.

New Windows operating systems change the operating system procedures for determining if a device name is valid. We suggest that you test this function on all relevant operating systems before using it for production purposes. Known problems exist in Windows 2000. On that operating system, this function will return TRUE even if the *dev_name* refers to a file rather than to a system device.

DIR_EXISTS("*path_name*")

See also the FILE_EXISTS function on page 227.

This function will return TRUE ("1") if the *path_name* refers to a valid directory path or file name. Otherwise, it will return FALSE ("0").

Example:

```
IF(DIR_EXISTS("%TMP^RESULTS") == 0){
    DOS "^MD %TMP^RESULTS>NUL "
}
```

The DOS command can be used to execute any valid system command. See the IC Layout Editor Reference Manual.

This example tests if a subdirectory named RESULTS exists in the directory used by ICED™ to store temporary files (usually "Q:\ICWIN\TMP".) The name of this temporary file directory is stored in the system macro TMP. (See page 299.) If the RESULTS subdirectory does not exist, it is created with the MD (Make Directory) DOS command.

The ">NUL" specification causes the output console messages to be discarded. This can prevent unwanted console effects from showing on the screen. See page 121.

The *path_name* can be an absolute or relative path. The reference above was an absolute reference; the path is defined using the drive letter and all directory names. If *path_name* begins without a drive letter or backslash ('\'), then it is assumed to be a relative path, a subdirectory of the current directory.

For example, assume that the current directory is "Q:\ICWIN\TUTOR". The following example would test if the Q:\ICWIN\TUTOR\MYTEST directory exists.

Example:

```
IF(DIR_EXISTS("MYTEST")){…
```

This function is tolerant of trailing backslashes ('\'). They are stripped before testing the path name.

You should use quotes around the value or macro reference you are using as the function argument. Many path names contain blanks or slashes that may confuse the parser if no quotes are used.

FILE_EXISTS("*path_name*")

See also the DIR_EXISTS function on page 226.

Use this function to determine a file exists. It will return TRUE ("1") if the *path_name* refers to an existing file name. Otherwise, it will return FALSE ("0").

The *path_name* argument can use an absolute path or a relative path. An example of a absolute path is the fully qualified file name "Q:\ICWIN-\TUTOR\TEMP.TXT". A relative path begins without a drive letter or backslash ('\'). The path of the current directory will be added as a prefix to a relative *path_name*. If "Q:\ICWIN\TUTOR" is the current directory, then the relative *path_name* of "TEMP.TXT" resolves to "Q:\ICWIN\TUTOR\TEMP.TXT".

Example:

```
IF (FILE_EXISTS("mycmd.cmd")) @mycmd.cmd
```

The example above uses a relative path to test if a command file exists before executing it. The command file MYCMD.CMD must exist in the current directory or it will not be executed.

You should use quotes around the value or macro reference you are using as the function argument. Many file names contain blanks or slashes that may confuse the parser.

FILE_TIME("*path_name*")

Use this function to determine the last modification time of the file with the name *path_name*. You can use either an absolute or relative path for the file name.

The return value is stored in "C" language time format, i.e. the number of seconds elapsed since midnight (00:00:00), January 1, 1970. While this is may be an awkward format for reporting, this format is ideal for comparing times.

Example:

```
GLOBAL #DRC.FILE.TIME = {FILE_TIME("%DRC.FILE")}
:
:
!from DRCRELOAD.CMD
IF(MACRO_EXISTS(#DRC.FILE.TIME)){
  IF(FILE_TIME("%DRC.FILE")!=%DRC.FILE.TIME){
    ERROR "Cannot reload -- %DRC.FILE has been modified"
  }
}
```

The MACRO-_EXISTS function returns TRUE only when the indicated macro exists. See page 231.

The example above comes from the DRCRELOAD.CMD command file supplied with the installation. If the user macro DRC.FILE.TIME was already created by some previous command file using the definition in the first line of the example, then it is compared to the modification time of the current version of file with the file name stored in the user macro DRC.FILE. If this file has changed since the DRC.FILE.TIME macro was assigned its value, the command file cannot continue.

Note that the function call in the DRC.FILE.TIME macro definition must be surrounded by {} to force evaluation rather than storing the function call as a string.

You should use quotes around the value or macro reference you are using as the function argument. Many file names contain blanks or slashes that may confuse the parser. The *path_name* can be a relative or absolute file name. See the FILE_EXISTS function on page 227 for more details on absolute vs. relative paths.

INT(*real_val*)

Expression	Result
{INT(3)}	3
{INT(3.2)}	3
{INT(3.999)}	3
{INT(-3.2)}	-3
{INT(-3.9)}	-3

This function is used to discard the part of a real number to the right of the decimal point. Numbers are not rounded to the nearest integer. Instead only the integer part of a real number is returned.

Example: **#Q = {INT(%REM / 10)}**

LEN("*string*")

Any quote character may be used ",',~, or `.

This function will return the number of characters in the string. You should surround the string with quotes if it contains characters likely to confuse the parser (e.g. blanks, commas or parentheses).

Example:

```
LOCAL  #NAME  $PROMPT="Enter name:"

IF (LEN("%NAME") == 0) {
        #NAME = FRED
}
```

This is a good example of how you can insure that a non-zero length string has been entered by the user of your command file. When this command file is executed, and the user just hits <Enter> at the "Enter name:" prompt, the statement in the IF block will store the default name of "FRED" in the NAME macro.

LIST_EXISTS("*list_name*")

See the LIST command on page 182 for more details on creating lists.

This function will return TRUE ("1") if a list with the name *list_name* has already been created. Otherwise, it will return FALSE ("0").

Example:

Do not omit the '_' when typing this function name.

```
IF (LIST_EXISTS(STEP.ERROR.LIST)) {
        IF (%LIST.EMPTY.STEP.ERROR.LIST==0){
                !list processing commands
        }
}
```

The LIST.EMPTY-.*list_name* system macro contains TRUE only when the list is empty. See page 284.

These lines from DRCSTEP.CMD (a command file supplied with the installation to step through DRC error shapes) test if a list exists and is not empty before processing the components in the list.

You should not combine the two IF commands in the example above into one command with a compound Boolean expression of the form below:

```
IF(LIST_EXISTS(STEP.ERROR.LIST) &&            &
         (%LIST.EMPTY.STEP.ERROR.LIST==0)){   !cmd file might fail
```

See more about Boolean expressions on page 53.

Even if the list STEP.ERROR.LIST does not exist, the macro reference in the second half of the compound expression above will still be evaluated. The LIST.EMPTY.STEP.ERROR.LIST macro will not exist and the command file will be immediately terminated with a syntax error.

MACRO_EXISTS("[#]*macro_name*")

See page 138 to learn more about macro scope.

This function tests that a macro has been defined. It also informs you of the scope of the macro (i.e. whether the macro is global or local). This function is used primarily in nested command files to test whether or not a specific local macro has been defined in the statement that called the command file. (See page 97 to see more examples of passing values into a command file.)

Do not omit the '_' when typing this function name. However, the square brackets are not typed. They indicate that the '#' is optional.

The return value of this function will be one of the following values:

0	*macro_name* does not exist or is only defined as a local macro in a different command file.
1	*macro_name* is defined as a global macro and no local macro with the same name is defined in the current command file.
2	*macro_name* is defined as a local macro in the current command file.

Example:

```
IF (MACRO_EXISTS(#DEFAULT)==2) #MSG="Enter value [%DEFAULT]:"
ELSE                           #MSG="Enter value:"
```

This function can be used to test the existence of macros created with the ITEM command.

This command file fragment builds different prompt messages depending on whether or not a local macro with the name DEFAULT exists. Since the IF command tests that the return value of the MACRO_EXISTS function must be equal to '2', the existence of a global macro with the name DEFAULT will be ignored. A global macro with that name that may have been created by some unrelated command file will not affect the prompt message in this command file.

Let us assume that the lines above are used in a command file with the name DETERMINE_VALUE. When this command file is called with the following syntax:

```
@DETERMINE_VALUE; LOCAL #DEFAULT = 6
```

Then the nested command file will build the message:

```
"Enter value [6]:"
```

However, when the command file is called without defining the DEFAULT local macro, then the message will be defined as:
"Enter value:"

Any quote character may be used ",',~, or `.

We recommend that you surround the *macro_name* string with quotes if you are using a macro reference for *macro_name*. Blanks or special characters can prevent the parser from interpreting the function call correctly unless the argument is surrounded by quotes.

Example:

```
LOCAL #TEST = $PROMPT ="Type macro name:"
IF (MACRO_EXISTS("%TEST"))    $ macro %TEST exists
ELSE                          $ macro %TEST does not exist
```

This command file fragment prompts the user to type the name of a macro, then tests if that macro exists. Since you are prompting the user to enter the macro name, anything might get stored in the TEST macro. For example, if the user types "(5,6)" instead of a macro name, and the quotes were not included in the function call (i.e. MACRO_EXISTS(%TEST)), then the function would fail with an error after macro substitution.

The _GET_INT.CMD command file is an excellent example of the usefulness of the MACRO_EXISTS function. See page 305.

MAX(*val1,val2*)

This function is used to determine the maximum of two real numbers.

Example:

```
#VAL = {MAX(%X_COORD1, MAX(%X_COORD2, %X_COORD3))}
```

This is the correct syntax to use if you need to find the maximum of a set of 3 numbers.

MIN(*val1,val2*)

This function is used to determine the minimum of two real numbers.

Example: **#VAL = {MIN(%X_COORD1, MIN(%X_COORD2, %X_COORD3))}**

This is the correct syntax to use if you need to find the minimum of a set of 3 numbers.

N_TO_CHAR(*int*)

Do not omit the '_' when typing this function name.

This function will return the single ASCII character represented by *int*. The value of *int* must be an integer in the range 32:128. Do not use quotes around *int*.

Example:
```
#STR = {N_TO_CHAR(%VAL)}{N_TO_CHAR(32)}&
       {N_TO_CHAR(%VAL)}
```

The ASCII character for the decimal integer 37 is the percent sign, '%'. The ASCII character for 32 is the blank space, ' '. If the value of macro VAL is "37", then the value of macro STR will be "% %". Note that each call to the function is surrounded by its own pair of curly braces to force evaluation.

POS1(*coordinate_pair_list*)

This function will return the first coordinate pair from a list. The list must contain **exactly** 2 coordinate pairs in the form:

(*x-coord1, y-coord1*),(*x-coord2, y-coord2*)

See this function used in another example on page 301.

The comma between the pairs is optional (any white space will do), but the parentheses and the comma in each pair are required.

Example:
```
#VERTEX_LIST = "(0,0) , (5,5)"
#VERTEX1 = {POS1(%VERTEX_LIST)}
```

The value of VERTEX1 after these statements are executed will be "(0,0)".

POS2(*coordinate_pair_list*)

This function will return the second coordinate pair from a list. The list must contain **exactly** 2 coordinate pairs in the form:

(*x-coord1, y-coord1*),(*x-coord2, y-coord2*)

The comma between the pairs is optional (any white space will do), but the parentheses and the comma in each pair are required.

Example:

#VERTEX_LIST = (0,0) , (5,5)
#VERTEX2 = {POS2(%VERTEX_LIST)}

The value of VERTEX2 after these statements are executed will be (5,5).

POSN(*n*, *coordinate_pair_list*)

You can use the ITEM command to get the position list and the number of vertices for a component. See page 173.

This function will return the *n*th coordinate pair from a list. The list must contain coordinate pairs in the form:

(*x-coord1, y-coord1*) (*x-coord2, y-coord2*)…(*x-coordi, y-coordi*)

The list can contain as many coordinate pairs as required. You can have commas separating one coordinate pair from the next. These commas will be ignored.

Use only **positive, non-zero** numbers for the *n* parameter.

Example:

#VERTEX_LIST = (0,0), (5,5), (5,7), (10,7), (10,3)
#VERTEXN = {POSN(%NUM,%VERTEX_LIST)}

If this example is executed when the user-defined macro NUM is set to 1, the value of VERTEXN will be set to "(0,0)". If NUM is set to "5", VERTEXN will be "(10,3)". If the statement is executed when NUM is set to "6", the command file will be terminated with the following error message:

```
Error: Too few coordinates – Cannot select position 6 from 5 coordinates
```

The STD_COORD function validates and formats a user typed coordinate pair. See page 240.

ROUND(*coord*)

ROUND1(*coord*)

ROUND2(*coord*)

or

ROUND(*x-coord, y-coord*)

ROUND1(*x-coord, y-coord*)

ROUND2(*x-coord, y-coord*)

The resolution grid consists of all coordinates that can be digitized with the mouse.

These functions are used to resolve a single coordinate or a coordinate pair to the nearest location on the resolution grid.

All three functions are identical except for the manner in which coordinates that are exactly halfway between points on the resolution grid are handled. Unless you need special control over how these coordinates are rounded, simply use the ROUND function in all situations. We will cover the use of ROUND1 and ROUND2 on the next page.

You can check the resolution mode with the RES.MODE system macro. See page 290.

When the coordinates of a shape are not located on the resolution grid determined by your technology, post-processing software that translates your design to machine instructions can distort the shape. These functions round calculated offsets or coordinates to the resolution grid. When you perform this rounding in the command file, you can inspect and possibly correct how your shapes are distorted by rounding .

Example: **#X_DISP = {ROUND(%WIDTH / %NUM_POINTS)}**

This example performs division on a single number stored in the user-defined macro WIDTH by the single number stored in the user-defined macro NUM_POINTS. The result of this division is then rounded to the nearest resolution grid displacement by the ROUND function.

Example: **#VERTEX1 = {ROUND(%VERTEX0 / 2)}**

The example above performs division on each coordinate of the coordinate pair stored in the user-defined macro VERTEX0, then rounds the result to the nearest point on the resolution grid.

Rounding coordinates can result in a shape shrinking or growing up to one resolution step.

Example:

```
#VERTEX0 = {ROUND(1.24, 1)}
#VERTEX1 = {ROUND(6.3, 3.1)}
ADD BOX %VERTEX0, %VERTEX1
```

You can test the resolution mode before approximating coordinates. See an example on page 290.

As we consider this example, let us assume that the resolution step size is 0.5. These lines will then result in the execution of the following statement:

```
ADD BOX (1,1), (6.5,3)
```

The coordinates for the box before rounding indicate that the width of the box should be 5.06 units. However, after rounding, the box is 5.5 units wide. This is because each x-coordinate is rounded towards the nearest grid point, not necessarily in the same direction.

Differences between the ROUND1 and ROUND2 Functions

See a brief overview on coordinate math on page 51.

The ROUND1 and ROUND2 functions are identical except in how they handle coordinates that are exactly halfway between points on the resolution grid.

ROUND1 is the traditional method of rounding numbers. Basically, numbers exactly halfway between multiples of the resolution step size are rounded away from the origin. This is usually the best method for rounding displacements (e.g. the distance between wires).

The ROUND2 function will instead always round coordinates exactly halfway between resolution points upwards or towards the right. This makes the function quadrant independent; i.e. coordinates are shifted in the same direction regardless of whether they are negative or positive. Performing the calculation in this manner avoids extra distortion when a shape contains coordinates on both sides of the origin. This means that ROUND2 is usually the best method for rounding coordinate pairs.

Example:

```
LOCAL #TOT_DISP $PROMPT "Digitize total displacement" X_DISP
LOCAL #SINGLE_DISP1 = {ROUND1(%TOT_DISP/6)}
LOCAL #SINGLE_DISP2 = {ROUND2(%TOT_DISP/6)}
```

The command fragment on the previous page prompts the user to digitize a total displacement in the x-direction then divides this number by 6, to obtain increments of this displacement on the resolution grid for further processing. We want the value to be independent of whether the user digitizes a positive displacement or a negative displacement. Should we use the SINGLE_DISP1 value calculated by the ROUND1 function, or the SINGLE_DISP2 value calculated by ROUND2?

Let us assume that the resolution step size is 0.5 and that the user digitizes a total displacement of 4.5 units. Then, each single displacement is 0.75 units.

If the user digitizes the displacement in the positive direction (i.e. from left to right), then both functions return the same number, '1'.

The ABS function returns the absolute value of a number. See page 220.

```
ROUND1(4.5/6)    →    ROUND1(.75)    →    1
ROUND2(4.5/6)    →    ROUND2(.75)    →    1
```

However, if the user digitizes the displacement from right to left, the total displacement is "-.75", and the results of the two functions are different.

```
ROUND1(-4.5/6)   →    ROUND1(-.75)   →    -1
ROUND2(-4.5/6)   →    ROUND2(-.75)   →    -0.5
```

So you can see that if you want negative and positive displacements handled in the same manner, you should use the ROUND1 function.

Example:

ADD BOX AT ROUND1(.25, .25) ROUND1(1.25, 1.25)
ADD BOX AT ROUND2(.25, .25) ROUND2(1.25, 1.25)

ADD BOX AT ROUND1(-.25, -.25) ROUND1(-1.25, -1.25)
ADD BOX AT ROUND2(-.25, -.25) ROUND2(-1.25, -1.25)

Figure 32 illustrates the effect of each of the round functions on two boxes whose corners are exactly halfway between points on the resolution grid (shown with the small crosses). The box in the positive quadrant is shifted to the same location by both functions. The box in the negative quadrant is shifted in the same direction by the ROUND2 function, leaving the boxes the same distance apart. However, the ROUND1 function shifts the box further away from the box in the positive quadrant.

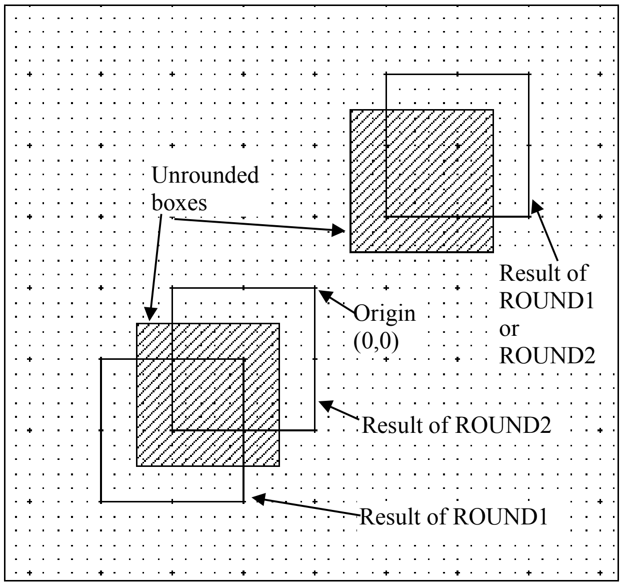


Figure 32: Result of ROUND1 and ROUND2 on two boxes

This demonstrates that ROUND2 is the better choice when rounding coordinate data.

ROUND Function Evaluates to ROUND1 or ROUND2

When you specify the ROUND function, the program will automatically choose between ROUND1 and ROUND2. If you call the ROUND function on a single number, the program will execute the ROUND1 function. When you call ROUND with a coordinate pair as the argument, then the program will use the ROUND2 function.

ROUND(x) $\Rightarrow$ **ROUND1(x)**
ROUND(x,y) $\Rightarrow$ **ROUND2(x,y)**

In most cases, this is the correct choice. You do not have to remember the details of the differences between ROUND1 and ROUND2. Simply specify the ROUND function and let the program resolve the function call to either ROUND1 or ROUND2.

However, in some cases you may want to specify either the ROUND1 or ROUND2 function instead of the multi-purpose ROUND function.

Example: **LOCAL #TOT_DISP $PROMPT "Digitize total displacement" DISP**
LOCAL #SINGLE_DISP = {ROUND(%TOT_DISP/6)}

This example is similar to the displacement example above, except that now the displacement is a coordinate pair with displacements in both the x and y directions since the DISP keyword is used rather than X_DISP. The multi-purpose ROUND function is called to resolve the result of the division to the resolution grid. Since the TOT_DISP macro is now a coordinate pair, the ROUND function will call ROUND2 rather than ROUND1. This is a poor choice for rounding displacements. In this case, it would be better to force the use of ROUND1 by explicitly calling it rather than the ROUND function.

SIN(*angle*)

This function will return the sine of an angle expressed in degrees.

Example: **#VAL = {SIN(30)}**

When this statement is executed, the user-defined macro VAL will be assigned the number 0.5.

SQRT(*pos_val*)

The SQRT function will return the square root of a positive number. Using a negative value for *pos_val* will result in an error message and the termination of a command file containing the expression.

Example: **${SQRT(49)}**

Typing this command on the command line will report the square root of 49 (i.e. 7) on the screen below the command line. Never type a blank between the function name and the '(' character. Always remember to surround a mathematical expression with curly braces.

STD_COORD("*string*")

When the POS keyword is used to prompt the user to digitize a position this function is not necessary. See page 143.

This function returns a coordinate pair in the standard "(*x*, *y*)" format from the pair of coordinates indicated in the argument *string*. If *string* does not contain a valid pair of real numbers, then the function will return the string "BAD". The primary purpose of this function is to verify user input in a command file where the user has been prompted to type in a coordinate pair.

Example:

The CMP function compares two strings and returns 0 when they are identical. See page 224.

```
LOCAL #COORD $PROMPT="Type new center coordinate pair"
LOCAL #VALID=0
WHILE(%VALID==0){
        #COORD = {STD_COORD("%COORD")}
        #VALID = {CMP("%COORD", "BAD")}
        IF(%VALID==0){
            $Invalid coordinate pair! <Esc> to quit or <Enter> to try again
            PAUSE 0
            #COORD $PROMPT="Type new center coordinate pair"
        }
}
```

Do not omit the '_' when typing this function name.

When the STD_COORD function is used in a WHILE loop like the one above, the command file will continue to prompt the user for a coordinate pair until the user has typed a valid pair of coordinates.

See a brief overview on coordinate math on page 51.

This function will ignore commas, open and close parentheses, and extra blanks in the argument string. If the user of the command file fragment above typed in any of the following strings, the STD_COORD function would return the string "(5, 6)":

```
"5 6"
"(5, 6)"
"5                6"
"5,6"
")5)6,"
```

The string returned by STD_COORD will be valid in any command, function, or expression that expects a coordinate pair. Some of the strings in the list above would cause a failure if you tried to use them in a mathematical expression or as the argument in a function like the X function (see page 246).

The use of quotes around the argument string when calling this function is almost always required. The string is likely to contain characters that have special meaning to the parser (i.e. ',', ')', and '(').

TAN(*angle*)

This function will return the tangent of an angle expressed in degrees.

Example: **#VAL = {TAN(30)}**

When this statement is executed, the macro VAL will be assigned the number 0.5773502692.

VALID_CELL_NAME("*string*")

Do not omit the '_' when typing this function name.

Returns TRUE ("1") if the string can be used as a valid cell name. The function will return FALSE ("0") if the string can not be used to name a cell.

The valid quote characters are ",',~, or `.

As with the other verification functions, the use of quotes around *string* is strongly recommended.

Example:

```
LOCAL #NEWCELL = $PROMPT "Enter the new cell name:"
LOCAL #INDEX = 0

#INDEX = {CELL("%NEWCELL")}

IF (%INDEX > -1) ERROR "Cell %NEWCELL already exists"
ELSEIF (VALID_CELL_NAME("%NEWCELL")){
        EDIT CELL %NEWCELL
        .
        .       !missing statements that create components in this new cell
        .
        EXIT
}
ELSE ERROR "%NEWCELL is not a valid cell name"
```

The CELL function returns the cell index of a cell already loaded, '0' if the cell is not loaded but does exist in a cell library, or '–1' if the cell does not exist See page 221.

This command file fragment prompts the user for a cell name and creates that cell as long as the cell does not already exist and the cell name is a valid string for naming a cell. If the call to VALID_CELL_NAME was not made, and the user typed an invalid string (e.g. "abc 123"), the EDIT command would fail and terminate the command file with an error.

The new cell will be saved in the current directory when the editor terminates.

VALID_INT("*string*")

This function insures that a string, (usually one stored in a macro) represents a valid integer. It returns TRUE (i.e. "1") if the *string* represents an integer. The function will return FALSE (i.e. "0") if the *string* does not represent a integer.

Example:

```
LOCAL #NUM_COPIES $PROMPT="Enter number of copies[1]:"
IF (VALID_INT("%NUM_COPIES")==0) #NUM_COPIES = 1
```

This example demonstrates a typical use of the VALID_INT function. If the user responds to the prompt with a valid integer, that will be used as the value of the NUM_COPIES macro. However, if the user simply hits <Enter> instead of typing a value, or if he enters a string that is not a valid integer (e.g. "twelve"), then the default value of "1" will be stored in NUM_COPIES.

See this function used in the sample command file for requesting the user to enter a valid integer, _GET_INT, on page 305.

The use of quotes around the string is strongly recommended, but not required. Blanks or special characters in the string may prevent the parser from interpreting the function call correctly unless the string is surrounded by quotes.

Any quote character may be used ",',~, or `.

In the example above, if the user pressed <Enter> without typing a value, then the function would be called with the following syntax after macro substitution:

```
VALID_INT("")
```

The statement above will be interpreted correctly and the return value will be 0. Without the quotes, the function call would fail and terminate the command file.

VALID_ITEM_NAME("*string*")

Use this function to verify that the string you want to use in the ITEM command can be used as a valid *item_name*.

Example:

```
LOCAL #NAME="8_ITEM"            !not a valid item name
IF (VALID_ITEM_NAME(%NAME)){
        ITEM LOCAL %NAME        !this stmt will not be executed
}
```

See page 173 for item name restrictions.

This example will not execute the ITEM command since the string stored in macro NAME does not contain a valid base name for item macros. The ITEM command would fail with a syntax error and halt the command file if it was executed with "8_NAME" as the *item_name*.

VALID_LAYER("*string*")

The user can select a layer from a menu of valid choices. See page 147.

This function is used to verify that a string represents a valid layer name or number. It returns TRUE ("1") if the *string* represents an existing layer name or valid layer number in the current cell. The function will return FALSE ("0") if the string does not represent a layer name or number.

Example:

Do not omit the '_' when typing this function name.

The _GET-_LAY.CMD file supplied with the installation can be used in place of this function. See page 279.

```
LOCAL #SCRATCH  = 254
LOCAL #TARGET  = $PROMPT="Enter target layer:"

XSEL OFF
UNSELECT ALL; SEL LAYER %SCRATCH ALL
IF(VALID_LAYER("%TARGET") && CMP("%TARGET", "0")!=0 ){
    SWAP LAY %SCRATCH AND %TARGET
}
ELSE  {
    ERROR "%TARGET not valid. Shapes are left on layer %SCRATCH."
        PAUSE 0
}
UNSELECT ALL
```

Layer 0 can be used to select subcells and arrays.

This command file fragment will swap shapes on a scratch layer with a layer to be entered by the user. If the user does not enter a valid layer name or number for the swap, no shapes are changed. Note that VALID_LAYER("0") will return TRUE. The extra test to determine if the target layer is equal to 0 is required if the layer will be used in a command that will fail if layer 0 is specified.

Any quote character may be used ",',~, or `.

As with the other verification functions, the use of quotes around *string* is strongly recommended. Unless *string* is surrounded by quotes, blanks or special characters in the string can make this function terminate with an error and the command file will fail with an error.

Determine if a list already exists with the LIST_EXISTS function. See page 230.

VALID_LIST_NAME("*string*")

Use this function to verify that the string you want to use in the LIST command can be used as a valid *list_name*.

Example:

The user can interrupt this type of loop and cancel the command file by pressing both mouse buttons.

```
LOCAL #NAME=$PROMPT="Enter name of list"
WHILE (VALID_LIST_NAME("%NAME")==0){
        PROMPT "Invalid list name <<%NAME>>"
        LOCAL #NAME=$PROMPT="Re-enter name of list"
}
LIST LOCAL %NAME
```

The example on the previous page uses a loop to insure that the list name is valid before executing the LIST command.

VALID_REAL("*string*")

This function returns TRUE ("1") if the *string* represents a real number or FALSE ("0") if the *string* does not.

Any quote character may be used ",',~, or `.

As with the previous function, the use of quotes around the string is strongly recommended, but not required. Unless the string is surrounded by quotes, blanks or special characters can make this function terminate with an error and the command file will fail with an error.

Example:

```
LOCAL #VALID = 0
WHILE (%VALID==0) {
        LOCAL #OFFSET = $PROMPT="Enter offset:"
        #VALID = {VALID_REAL("%OFFSET")}
}
```

See this function used in the sample command file for requesting the user to enter a valid real number, _GET_REAL-.CMD file supplied with the installation.

The command file fragment above will loop until the user has entered a valid real number. If the user presses <Esc> at the prompt instead, the entire command file will be canceled.

The use of curly braces {} around the expression with the call to the VALID_REAL function is critical. Any time a macro assignment contains a function call, the function call will not be performed unless the program is forced to evaluate the expression. When the curly braces surround the expression, it will be evaluated. When the curly braces are not present, the program interprets the expression as a string of characters with no special purpose.

X((*x-coord, y-coord*))

This function returns the x-coordinate from a single coordinate pair. In addition to the parentheses that indicate that this is a reference to a function, the coordinate pair must be surrounded by parentheses.

Example:

#VERTEX = (20.5,34.0)
#X_COORD = {X(%VERTEX)}

After macro substitution, the second statement above will read:

See this function used in another example on page 292.

#X_COORD = {X((20.5,34.0))}

This is the correct syntax. Once these statements are executed, the value of X_COORD will be 20.5.

X0(*coordinate_pair_list*)

This function will return the first x-coordinate from a list of coordinate pairs. The list must contain **exactly** 2 coordinate pairs in the form:

(*x-coord1, y-coord1*) (*x-coord2, y-coord2*)

Example:

#X_LEFT = {X0(%CELL.BOX.1)}
#X_RIGHT = {X1(%CELL.BOX.1)}

Refer to the CELL.BOX.*cell_index* system macro description on page 258.

The statement above will set the X_LEFT macro to the x-coordinate of the lower left corner of the bounding box of the root cell. The X_RIGHT macro will contain the x-coordinate of the upper right corner of the bounding box of the root cell.

X1(*coordinate_pair_list*)

This function will return the second x-coordinate from a list of coordinate pairs. The list must contain **exactly** 2 coordinate pairs in the form:

(*x-coord1, y-coord1*),(*x-coord2, y-coord2*)

The **XCMP** function is described on page 224.

Refer to the example on page 246.

Y((*x-coord, y-coord*))

This function returns the y-coordinate from a single coordinate pair.

Example:

```
#VERTEX = (20.5,34.0)
#Y_COORD = {Y(%VERTEX)}
```

After these statements are executed, the value of Y_COORD will be 34.

Y0(*coordinate_pair_list*)

This function will return the first y-coordinate from a list of coordinate pairs. The list must contain **exactly** 2 coordinate pairs in the form:

(*x-coord1, y-coord1*) (*x-coord2, y-coord2*)

Example:

```
#Y_BOT = {Y0(%LAST.BOX)}
#Y_TOP = {Y1(%LAST.BOX)}
```

Refer to the LAST.BOX system macro description on page 271.

This example will set the macro Y_BOT to the lower y-coordinate of the last box digitized with the cursor. Y_TOP will be set to the upper y-coordinate of the last box digitized with the cursor.

Y1(*coordinate_pair_list*)

This function will return the second y-coordinate from a list of coordinate pairs. The list must contain **exactly** 2 coordinate pairs in the form:

(*x-coord1, y-coord1*),(*x-coord2, y-coord2*)

Refer to the example on page 247.

ICED™ System Macros

Overview

There are also several user-defined macros that have special significance. See page 153 to learn about the EXIT.SUBCELL, EXIT.ROOT, ENTER-.SUBCELL, , and ERROR.CMD macros.

In addition to the user-defined macros covered back beginning on page 133 there are many ICED™ pre-defined system macros. Remember that macros are used for the same purposes as variables in other programming languages. System macros are like read-only variables. You cannot set their values directly.

System macros have their values set and updated by the ICED™ program. They contain useful information about the cells currently loaded, the editor settings, and recent user actions among other things.

New system macros are often added with program updates. Use the command below at the editor command line to see the current list.

Example: **SHOW SYSTEM_MACROS=***

The list as of the printing of this manual is shown in the table on page 252. The succeeding pages describe each system macro in alphabetical order with details and examples.

Refer to the $*comment* command on page 159.

You can find out the value of any of these system macros by typing a $*comment* command at the command prompt in the layout editor. When used this way, the comment is left as an echo on the bottom of the editor window. Macro substitution is performed as the comment is generated.

Example: **$ My startup command file is %START.CMD**

Typing this command at the command prompt will display the path and name of the startup command file. This information is stored in the START.CMD system macro.

Indexed System Macros

Many of the system macros are indexed. That is, some macros refer to values stored in an indexed table. These macros have names that use a prefix string followed by an index number or string that selects a unique macro from the table.

For example, the LIB.*lib_index* system macros store the names of all the current cell library directories. The first cell library name will be stored in LIB.1, the second (if it is defined) will be stored in LIB.2, etc.

Some system macros allow names to be used as the index for convenience. For example, all of the layer related system macros can be indexed by either the layer name or the layer number. One of these layer system macros is LAYER.COLOR.*layer_spec.* Suppose that layer 1 has the name "NWEL", in this case LAYER.COLOR.NWEL refers to the same system macro as LAYER.COLOR.1.

Using an index beyond the range valid for that system macro will result in an error and the immediate termination of the command file. Valid indices usually begin with 1 and end with a maximum index stored in a related system macro. You can use a reference to another macro as the index as shown below.

The SHOW command cannot be used to report the value of an indexed macro. If you want to report the value of an indexed macro, use a $*comment* command with the full system macro name.

Example: **$%CMD.DIR.%N.CMD.DIRS**

Typing the example above at the command prompt will report below the command line the full path of the last directory on the command file search path. The N.CMD.DIRS system macro stores the last valid index for the CMD.DIR.*dir_index* system macro. If N.CMD.DIRS contains the number 3, the entire macro reference resolves to %CMD.DIR.3. This macro will contain the path to a directory on the command file search path.

Cell Table Indices

See page 104for more information on the cell table.

Several of the cell-related system macros refer to cells by their index into the cell table. The cell table is a list of all cells loaded in the current layout editor session. This list includes the root cell (i.e. the cell the editor was launched to edit), all of its subcells, and all other cells that have been explicitly opened by the EDIT, PEDIT, or TEDIT commands (unless the QUIT command was used to unload the cell) as well as new cells created with the GROUP command.

The EXIT command closes a cell, but does not unload it from the cell table.

For example, the CELL.NAME.*cell_index* system macros store the cell names of all cells in the cell table.

Cell table indices always begin with 1. This is always the index of the root cell. The MAX.CELL system macro contains the current number of entries in the cell table; therefore it is also is the highest valid *cell_index*. See page 286 for the MAX.CELL system macro description.

You can obtain the cell table index from a cell name with the CELL() function. (See page an example on page 258.) You can obtain the cell table index for a selected cell with the ITEM command. (See an example on page 263.)

System Macros Sorted by Purpose

Category	**Macro name**	**Purpose**	**Page**
Coordinates	MAX_COORD	The maximum valid coordinate	286
	CELL.BOX.*cell_index*	Bounding box for specific cell	258
	SELECT.BOX	Bounding box of selected components	292
	LAST.POS	Last position digitized	272
	LAST.BOX	Last box digitized	271
	LAST.DISP	Last displacement digitized	271
	LAST.RULER.POS.0	First point defined in last RULER command	273
	LAST.RULER.POS.1	Final point defined in last RULER command	273

Continued on next page.

Cell macros	CELL	Name of cell currently being edited	258
	SHORT.CELL	First 8 characters of CELL macro	292
	CELL.ROOT	Name of cell ICED™ was launched to edit	264
	SHORT.ROOT	First 8 characters of CELL.ROOT	293
	CELL.NAME.*cell_index*	Name of cell for cell table index *cell_index*	264
	CELL.BOX.*cell_index*	Bounding box for specific cell	258
	CELL.DEPTH.*cell_index*	Nesting depth for specific cell	259
	CELL.LIB.NUMBER.*cell_index*	Cell library that contains specific cell	262
	CELL.LIB.TYPE.*cell_index*	Protection status of cell library that contains specific cell	263
	CELL.EDIT.*cell_index*	Indicates if cell can be edited	261
	SUBCELL.EDIT.*cell_index*	Indicates if cell is subcell that can be edited	296
	NEW.CELL	Test if cell was just created	289
	MAX.CELL	Last cell table index used	286
Directory and file macros	AUX	Name of AUXIL directory	256
	TMP	Name of TMP directory	299
	HOME.*dir_index*	Name of ICED home directory (ies)	268
	N.HOME	Number of ICED home directories	289
	CMD.DIR.*dir_index*	Name of command file directory (ies)	265
	N.CMD.DIRS	Number of command file directories	288
	DRC.DIR.*dir_index*	Name of DRC file directory(ies)	266
	N.DRC.DIRS	Number of DRC file directories	288
	LIB.*lib_index*	Cell library path name from index	282
	LIB.TYPE.*lib_index*	Protection status for cell library	283
	N.LIBS	Number of cell libraries	288
	EXEC.FILE	File name of current command file	268
	JOU	Journal file name	270
	START.CMD	Name of startup command file	295
	ALWAYS.CMD	Name of always command file	255
View window settings	VIEW.BOX	Current view window	301
	VIEW.CENTER	Center of current view window	302
	VIEW.SCALE	Scale of current view window	302

Continued on next page.

Editor settings and counters	N.SELECT	Number of currently selected components	289
	ID.MAX	Last component id used	269
	MENU	Name of menu file	287
	RES.STEP	Resolution step size	291
	RES.MODE	Resolution mode	290
	SNAP.ANGLE	Snap angle	293
	SNAP.OFFSET	Snap offset	293
	SNAP.STEP	Snap step size	293
	USE.ARC.TYPE	Default new arc wire end type	299
	USE.LAYER	Default layer	299
	USE.N.SIDES	Default number of sides for new circles	299
	USE.TEXT.JUST	Default text justification for new text	300
	USE.WIRE.TYPE	Default end type for new wires	300
	TIMER	Time (in seconds) from start of edit session	298
	SPACER.ON	Spacer cursor mode	294
	SPACER.SPACE	Spacer cursor spacing value	294
	SPACER.STYLE	Spacer cursor style number	294
	SPACER.TRACK.LAYERS	Auto spacer cursor distance mode set?	294
	LIST.EMPTY.*list_name*	Undeleted components remain on list?	284
	LIST.EOL.*list_name*	Has end of list been reached?	284
	LIST.INDEX.*list_name*	Current index of list	285
	LIST.LEN.*list_name*	Number of components on list	286
Layer settings	USE.LAYER	Default layer	299
	BLANKED.CELL.LAYERS	List of all layers blanked for nested cells.	257
	BLANKED.ROOT.LAYERS	List of all layers blanked in current cell.	257
	PROTECTED.LAYERS	List of all protected layers.	290
	LAYER.BLANKED.CELL-.*layer_spec*	Blank status of components on given layer in nested cells	274
	LAYER.BLANKED.ROOT-.*layer_spec*	Blank status of components on given layer in current cell	274
	LAYER.PROTECTED-.*layer_spec*	Protection status of components on given layer in current cell	279

Continued on next page.

Layer settings	LAYER.NAME.*layer_spec*	Layer name for layer number *layer_spec*	277
	LAYER.NUMBER.*layer_spec*	Layer number for layer name *layer_spec*	277
	LAYER.WIRE.WIDTH-.*layer_spec*	Default wire width for layer *layer_spec*	281
	LAYER.SPACE.*layer_spec*	Default spacing for spacer cursor	279
	LAYER.PAT.*layer_spec*	Pattern number used to display layer *layer_spec*	277
	LAYER.PEN.*layer_spec*	Pen number used to plot layer *layer_spec*	278
	LAYER.COLOR.*layer_spec*	Color number for layer *layer_spec*	276
	LAYER.STREAM.LAYER-.*layer_spec*	Stream layer number for layer *layer_spec*	280
	LAYER.STREAM.DATA-.TYPE.*layer_spec*	Stream data type number for layer *layer_spec*	280
	LAYER.STREAM.TEXT-.TYPE.*layer_spec*	Stream text type number for layer *layer_spec*	281
	LAYER.CIF.LAYER-.*layer_spec*	CIF layer name for layer *layer_spec*	275

Figure 33: ICED™ System macros

System Macros Alphabetically

Refer to the table on the previous pages to see ICED™ system macros sorted by purpose.

ALWAYS.CMD

See an overview on always command files on page 18.

This macro contains the path and filename of the always command file. This is the command file that is executed when one of the ALWAYS, LEAVE, EXIT, or QUIT command line options is used when the editor is launched.

Example: **@%ALWAYS.CMD**

Typing the command on the previous page at the command prompt will execute the current always command file. Another method to execute this command is to use the **menu** options 1:FILE→@ALWAYS or 3:@*. (The menu option 3:@* executes the startup command file and then the always command file.)

If no always command file is defined, then the value of this macro will be "DO_NOTHING". When the command above is executed with this value for ALWAYS.CMD, the command file is **not** halted with an error; the command simply has no effect.

AUX

See page 14 for more details on the command file search path.

The value of this macro is the name of the primary AUXIL directory. This is the directory where ICED™ looks for technology independent auxiliary files such as command files, menu files, plotter definition files, and fill pattern files. The value of this macro is the fully qualified name of the AUXIL subdirectory of the first home directory, usually "Q:\ICWIN[9]\AUXIL\".

Example: **DOS "^COPY %AUX^*.cmd>NUL"**

The DOS command executes the indicated command in a temporary DOS console window. See page 119.

Executing this command will cause the command files in the AUX directory to be copied to the current directory. Note that the macro substitution is performed despite the quotes around the command string. The value stored in the AUX macro ends with a final '\' character to facilitate using it to name files. The '^' character after the %AUX reference separates the name of the macro from the rest of the string without adding a space. After macro substitution, the command will be similar to:

```
DOS "^COPY Q:\ICWIN\AUXIL\*.cmd>NUL"
```

The ">NUL" discards the console messages generated by this command. It is strongly recommended when you use DOS console commands in a command file. See page 121 for details.

[9] Remember that Q:\ICWIN represents the drive letter and path where you have installed ICED™.

BLANKED.CELL.LAYERS

Set the value of this macro with the [UN]BLANK command. Refer to the IC Layout Editor Reference Manual.

The macro contains the list of layers blanked for nested cells. The list is stored in layer list format using the layer numbers.

If there are no layers blanked for nested cells, then value of the macro will be "(none)". You should check for this value before using the string in another command.

Example:

LOCAL #BLANKED_CELL_LAYLIST = %BLANKED.CELL.LAYERS

This example is expanded in the next macro description.

To find out about a specific layer, use the LAYER.BLAN-KED.CELL-.*layer_spec* system macro. See page 274.

If some layers are blanked for components in nested cells, then the string stored in this system macro will be the list of layer numbers delimited with '+'. This syntax is valid for the (UN)BLANK command. (See the IC Layout Editor Reference Manual for complete details on layer lists.) Some typical layer lists that might be stored in this system macro are shown below:

"1+3"	-Layers 1 and 3
"0:255"	-All layers
"0:5+7:255"	-All layers except for layer 6
"(none)"	-No cell layers are blanked

BLANKED.ROOT.LAYERS

Set the value of this macro with the [UN]BLANK command. Refer to the IC Layout Editor Reference Manual.

The macro contains the list of layers blanked for the current cell, the cell open for editing. The list is stored in layer list format using the layer numbers.

If there are no root layers blanked, then value of the macro will be "(none)". You should check for this value before using the string in another command.

Blanked components are not visible or selectable.

If some layers are blanked for components in the current cell, then the string stored in this system macro will be the list of layer numbers delimited with '+'. This syntax is valid for the (UN)BLANK command. Some typical layer lists that might be stored in this system macro are shown on in the previous macro description.

Example:

To find out about a specific layer, use the LAYER.BLANKED.ROOT-.*layer_spec* system macro. See page 274.

```
LOCAL  #BLANKED_CELL_LAYLIST = %BLANKED.CELL.LAYERS
LOCAL  #BLANKED_ROOT_LAYLIST = %BLANKED.ROOT.LAYERS
UNBLANK ALL
        !missing processing that requires all layers to be visible or selectable
!restore blank status of layers in nested cells
IF (CMP("%BLANKED_CELL_LAYLIST","(none)") != 0)          &
        BLANK CELL LAYERS=%BLANKED_CELL_LAYLIST
!restore blank status of layers in current cell
IF (CMP("%BLANKED_ROOT_LAYLIST","(none)") != 0)          &
        BLANK ROOT LAYERS=%BLANKED_ROOT_LAYLIST
```

CELL

The CELL.ROOT system macro contains the name of the root cell.

The value of this macro is the name of the cell currently being edited. This does not mean the name of the cell file, but merely the name of the cell itself.

Example: **$%NUM_DEL components deleted from cell %CELL**

Before this statement is executed, the %CELL reference will be replaced with the name of the cell being edited. The %NUM_DEL reference will also be replaced with the contents of that user-defined macro. The string will then be echoed in the journal file and on the screen. If this is the last command in the command file, the comment will be left on the screen when the command file is complete.

If you prefer to use the name of the cell shortened to 8 characters, use the SHORT.CELL system macro instead. See page 292.

CELL.BOX.*cell_index*

Refer to an overview of cell indices on page 252.

This macro contains the coordinates of the bounding box for the cell that has the indicated cell table index. A bounding box is the smallest rectangle square with the axes that encloses all of the components in the cell. The coordinates are in the coordinate system of the indicated cell. The lower left corner coordinate is provided first.

Example: **$ All root cell components are within the box: %CELL.BOX.1**

When this example is typed at the command prompt in the editor, the editor will report the coordinates of the bounding box of the root cell (i.e. the cell the editor was launched to edit) on the screen under the command prompt line. The comment created will look similar to:

```
$ All root cell components are within the box: (-117.0, 0.0) (8.0, 62.5)
```

To get the bounding box coordinates for a specific cell name, you can use the CELL function to get the cell table index. (See page 221.)

Example:

```
LOCAL       #CELL_NUM = {CELL("%CELL_NAME")}
LOCAL       #CELL_NAME = MYCELL
LOCAL       #CELL_BOX = ""

#CELL_BOX = %CELL.BOX.%CELL_NUM
```

See page 246 for an example that uses the X0 and X1 functions to parse the CELL.BOX.*cell_index* macro.

When you omit the *cell_index* entirely, it refers to the bounding box of the current cell.

Example:

```
LOCAL       #BOUNDS = %CELL.BOX
```

CELL.DEPTH.*cell_index*

For the purposes of this system macro, depth refers to how deeply cells are nested within the specified cell. A cell that contains no other cells has depth 0. A cell that contains a cell that contains no other cells has depth 1.

Refer to an overview of cell indices on page 252.

This system macro will contain the depth of the specified cell represented as an integer. The cell is specified by its index into the cell table. The depth information must have already been created by use of the MARK_SUBCELLS command.

Example:

```
MARK_SUBCELLS
LOCAL #CNAME = $PROMPT "Enter cell name."
$Cell Depth for cell %CNAME = %CELL.DEPTH.{CELL("%CNAME")}
```

The MARK_SUB-CELLS command fills a table with data used by this system macro as well as others. See page 197.

The example above uses the CELL function to get the cell table index for the cell name typed by the user. This function call is evaluated because it is contained in curly braces. The entire reference in curly braces is replaced by the integer cell index. The CELL.DEPTH.*cell_index* system macro reference can then be resolved. The number of levels of subcell nesting in the indicated cell will be reported. If the indicated cell contains no subcells, the reference will be "0".

When you omit the ".*cell_index*" suffix as shown below, the macro refers to the current cell, the cell you are editing.

Example:

```
LOCAL #MY_DEPTH = %CELL.DEPTH
```

The values stored in this system macro can be used for sorting cells. You can sort cells so that no cell in the list will contain a cell that is not earlier on the list. For example, a depth 3 cell cannot possibly contain a depth 4 cell.

Example:

This processing is very similar to that in the _LOOP2.CMD file supplied with the installation. This command file is designed to execute an operation in every subcell of the current cell.

```
LOCAL #I = 1; LOCAL #J = 1
LOCAL #MAX_DEPTH = 0
LOCAL #THIS_DEPTH = 0
!Determine maximum nesting depth
MARK_SUBCELLS
#MAX_DEPTH = %CELL.DEPTH
!Create list of macros with sorted cell indices
#THIS_DEPTH = 0
WHILE ( %THIS_DEPTH <=%MAX_DEPTH) {
        #I = 1
        WHILE  ( %I <= %MAX.CELL) {
                IF (%CELL.DEPTH.%I == %THIS_DEPTH){
                        GLOBAL #MYCELL.%J = %I
                        #J = {%J + 1}
                }
                #I = {%I + 1}
        }
        #THIS_DEPTH = {%THIS_DEPTH + 1}
}
```

The lines on the previous page will create an array of global user macros with the names MYCELL.1, MYCELL.2, etc. Each macro will contain a different cell table index. The macros will be created so that the cells with the lowest depth will be first in the array.

If you process these cells later in the order the macros were created, you can be sure that every subcell of the current cell has already been processed.

CELL.EDIT.*cell_index*

See page 252 for a definition of the cell table, cell libraries and cell indices.

This macro is used to determine the edit status of cells already loaded. The value of the macro informs you whether or not you can edit and then save the indicated cell. Specify the cell by setting *cell_index* to the positive integer that represents the index into the cell table.

See the related SUBCELL.EDIT macro for a more efficient method of looping through only certain subcells See page 296.

Value	Meaning
0	The cell cannot be edited since it is already open, contains an open cell, or the *cell_index* does not refer to a valid cell.
1	The cell is in a read-only library that cannot be edited then saved.
2	The cell is in a copy-edit library so that if you edit and then save it, it will be saved to the current directory rather than its original library.
3	The cell is directly editable.

Figure 34: Possible values for CELL.EDIT.*cell_index* macros

To determine the edit status for a specific cell name, you must use the CELL function first to obtain the cell table index. (See page 221.) If the cell it is not currently loaded, the CELL function also tells you whether or not the cell already exists in a cell library

Example:

```
LOCAL       #CELL_NUM = ""
LOCAL       #CELL_NAME = $PROMPT = "Enter cell name"

#CELL_NUM = {CELL("%CELL_NAME")}
IF (%CELL_NUM >0) {
        IF (%CELL.EDIT.%CELL_NUM == 3) {
                EDIT CELL %CELL_NAME
                        !Missing statements that modify cell
                LEAVE
        }
        ELSE  Return "Cell %CELL_NAME cannot be edited then saved."
}
ELSE  Return "Cell %CELL_NAME not loaded."}
```

The curly braces are required around the call to a function to force evaluation. See page 48.

The CELL.EDIT.*cell_index* macro is usually used in a loop with *cell_index* specified with a counter. See an example on page 264.

CELL.LIB.NUMBER.*cell_index*

Refer to an overview of cell indices on page 252.

This macro is used to determine the cell library that contains a specific cell. The cell is specified by its cell table index. The value stored in the macro is the cell library number where the cell file is saved. To get the cell library name from the number, use the system macro LIB.*lib_index*.

Example:

```
LOCAL #MYCELL_NAME = $PROMPT = "Enter cell name."

$Cell %MYCELL_NAME is stored in library &
%LIB.%CELL.LIB.NUMBER.{CELL("%MYCELL_NAME")}
PAUSE
```

The function CELL() returns the cell table index from the cell name. See page 221.

When you omit the *cell_index*, CELL.LIB.NUMBER refers to the library of the cell you are currently editing.

CELL.LIB.TYPE.*cell_index*

This macro is used to determine the protection status of the cell library that contains a specific cell. The protection status code determines whether or not the cell can be edited. The cell is specified by its cell table index.

Cell libraries and their protection status are defined with the ICED_PATH environment variable. See page 106.

This macro is very similar to the CELL.EDIT.*cell_index* system macro. For any *cell_index* except for those of open cells, the CELL.LIB.TYPE.*cell_index* system macro will have the same value as the CELL.EDIT.*cell_index* system macro. When the cell index is that of an open cell, this macro will store the cell library protection status code, while the CELL.EDIT.*cell_index* system macro will store 0, since open cells cannot be edited. See the CELL.EDIT.*cell_index* description on page 261 for the valid non-zero values stored by the CELL.LIB.TYPE *cell_index* system macro.

Example:

The N.SELECT system macro stores the number of selected components. The ITEM command requires that exactly one component is selected when the command is executed.

```
UNSEL ALL;SEL CELL=* NEAR
IF (%N.SELECT == 1) {
    ITEM LOCAL #MYCELL
    IF (CMP(%MYCELL.TYPE, "CELL") == 0){
       IF (%CELL.LIB.TYPE.%MYCELL.CELL.NO < 3){
           RETURN "Cell %MYCELL.CELL.NAME is not directly editable"
       }
    }
}
```

The example above uses the ITEM command to obtain the cell table index of a cell the user selects at the beginning of the command file. The cell table index will be stored in the %MYCELL.CELL.NO user macro by the ITEM command. This macro is created only when a cell is selected when the ITEM command is executed, so we test for this type of component before referring to the %MYCELL.CELL.NO user macro.

Say that the user selects a cell with a cell table index of 45. Then the %CELL.LIB.TYPE.%MYCELL.CELL.NO reference will resolve to %CELL.LIB.TYPE.45. This macro will contain the protection code for the cell library that contains that cell. If the value is less than 3, then the cell cannot be edited and saved back into its original cell library.

When you omit the *cell_index*, then it is interpreted as referring to the current cell, the one you are currently editing. This will provide the protection code of the cell library that stores the current cell.

Refer to an overview of cell indices on page 252.

CELL.NAME.*cell_index*

Refer to _LOOP.CMD for another way to execute a set of commands on every cell in the cell table.

This macro is used to determine the name of a cell from its index into the cell table.

The example below uses the CELL_NAME.*cell_index* macros to open by name each editable cell in the cell table.

Example:

MAX_CELL is described on page 286.

CELL_EDIT-.*cell_index* is described on page 261.

```
WHILE(%N <= %MAX.CELL){
     IF(%CELL.EDIT.%N==3){        !Edit cell only if it is directly editable
        EDIT CELL %CELL.NAME.%N;
        !process this cell
        LEAVE                     !saves the cell only if it was changed
     }
     #N = {%N +1}
}
```

When you omit the *.cell_index*, then it is interpreted as referring to the current cell, the one you are currently editing. The value stored in CELL.NAME is the same as that stored in the CELL system macro.

The function CELL() returns the cell table index from the cell name. See page 221.

CELL.ROOT

This macro contains the name of the cell ICED™ was launched to edit.

Example:

```
DOS "^COPY %CELL.ROOT^.JOU Q:\MYDIR >NUL"
```

See an overview of DOS commands on page 121.

The example on the previous page uses a DOS command to copy the current journal file to the Q:\MYDIR directory. The second '^' character delimits the name of the system macro from the rest of the string without a space. After substitution, the command executed by the DOS interpreter will be similar to:

```
COPY MYCELL.JOU Q:\MYDIR >NUL
```

Refer to an overview of indexed system macros on page 251

CMD.DIR.*dir_index*

These system macros contain the path(s) of the directory(ies) where ICED™ will search for command files.

To learn more about the command file search path, refer to page 14.

The editor will search for command files in a certain set of directories. The first directory searched is always the directory of the root cell. The editor will then search through all of the directories defined with the ICED_CMD_PATH environment variable. Finally the AUXIL subdirectory of each of the directories defined with ICED_HOME is searched.

The first valid value for *dir_index* is 1. The maximum *dir_index* is the number of directories in the command file search path. This number is stored in the N.CMD.DIRS system macro. (See page 288.)

Example:

See the WHILE command on page 212.

See the FILE_EXISTS function on page 227.

Refer to the IC Layout Editor Reference Manual for the SPAWN command.

```
! search for MYFILE on command file search path and open in text editor
GLOBAL #MYFILE = "BIPOL.TXT"
GLOBAL #N = 1

WHILE (%N <= %N.CMD.DIRS){
        $Look for %CMD.DIR.%N^%MYFILE
        IF (FILE_EXISTS("%CMD.DIR.%N^%MYFILE" )){
           SPAWN "-NOTEPAD.EXE %CMD.DIR.%N^%MYFILE"
           RETURN
        }
        #N = {%N + 1}
}
ERROR "File %MYFILE not found!"
```

When you search for files using commands similar to those on the previous page, rather than using fully qualified file names based on your installation, then the code will not need modification when you need to change the command file search path.

The CMD.DIR.*dir_index* macros will always end in '\' to make it easy to form file names. The "^" symbol in the example above is used to delimit the indexed macro name from the next macro reference without adding a space to the file name. (See page 44.)

When you omit the *dir_index* completely, it is the same as typing CMD.DIR.1. This is the directory of the root cell, the first directory searched for command files.

Refer to an overview of indexed system macros on page 251

DRC.DIR.*dir_index*

These system macros contain the path(s) of the directory(ies) where ICED™ will search for rules files used by the DRC (Design Rules Checker) program available separately from IC Editors, Inc. The editor will search for these files first in the directories defined on the DRC_PATH environment variable. If the required file is not found, the AUXIL subdirectory of each HOME directory tree is searched.

The first valid value for *dir_index* is 1. The maximum *dir_index* is the number of directories in the DRC rules file search path. This number is stored in the N.DRC.DIRS system macro. (See page 288.)

Example:

See the WHILE command on page 212.

See the FILE_EXISTS function on page 227.

```
! search for MYFILE rules file and copy it if not found
GLOBAL #MYDIR = "Q:\MYTECH\MASTER\"
GLOBAL #MYFILE = "BIPOL.RUL"
GLOBAL #N = 1
WHILE (%N <= %N.DRC.DIRS){
        $Look for %DRC.DIR.%N^%MYFILE
        IF (FILE_EXISTS("%DRC.DIR.%N^%MYFILE" )) RETURN
        #N = {%N + 1}
}
$File %MYFILE missing. Copying %MYDIR^%MYFILE to %DRC.DIR.1
DOS "^COPY %MYDIR^%MYFILE  %DRC.DIR.1^%MYFILE >NUL"
```

The DRC.DIR.*dir_index* macros will always end in '\' to make it easy to form file names.

You should use '^' symbols to delimit one macro reference from the next. This symbol is used to delimit a macro reference without adding a space to the string.

When you omit the *dir_index* completely, it is the same as typing DRC.DIR.1.

EXEC.DIR

The directory of the root cell is stored in the LIB system macro.

This system macro contains the path of the directory where the current command file is stored. This can be useful when other files required by your command file are stored in the same directory.

When no command file is being executed, the value of this system macro is "UNDEFINED".

Example: **DOS '^%EXEC.DIR^MYPROG.BAT >NUL'**

See another example of the use of this system macro in the BUSROUTE-.CMD example on page 327.

When this command is executed in a command file, the macro reference '%EXEC.DIR' will be replaced with the drive letter and directory path where the command file is stored. The '^' that follows this macro reference separates the macro reference from the name of the file without inserting a space.

After the macro references are resolved, if the current command file is stored in the directory "Q:\ICED\AUXIL" the command string executed will be:

```
DOS '^Q:\ICWIN\AUXIL\MYPROG.BAT >NUL'
```

See page 121 for details on executing DOS commands.

This macro allows your command file to be more portable. The command file can be relocated, or sections of code can be reused in command files stored in different locations, and statements like the one above can still be used without requiring edits to the command file to specify the directory. This method also does not require the user to modify the DOS PATH environment variable that stores the operating system's executable file search path.

EXEC.FILE

This system macro contains the fully qualified filename of the current command file. When no command file is being executed, the value of this system macro is "UNDEFINED".

Example: **RETURN "Success. Command file %EXEC.FILE completed successfully."**

The reference to EXEC.FILE is replaced in the string before the RETURN command is executed. The message is left on the screen after control is returned to the editor. You can add this command to the end of every one of your command files to provide feedback to the user that the command file has performed it's function and completed.

Refer to an overview of indexed system macros on page 251

HOME.*dir_index*

These system macros contain the path(s) of the directory(ies) where the ICED™ program files are installed.

To learn more about home directories, refer to the IC Layout Editor Reference Manual or see Q:\ICWIN\DOC\TREES.TXT.

Most installations have a single home directory, the Q:\ICWIN[10] directory. However, you can specify more than one directory in the ICED_HOME environment variable definition (usually stored in the project batch file.) In this case, each of these directories will have a HOME.*n* system macro, where *n* is the index of the directory in the ICED_HOME environment variable definition. The maximum *dir_index* is stored in the N.HOME system macro. (See page 289.)

Example: **DOS '^COPY X:\PROJX\TEST.CEL %HOME.%N.HOME^SAMPLES'**

See page 121 for details on executing DOS commands.

Let us assume that the command file fragment above is run by a user who has defined 3 home directories in the ICED_HOME definition in his project batch file. The last directory is intended to store his modified files, while the first two are located on networked drives. The number stored in this case in the N.HOME system macro is 3. The %HOME.3 macro contains the drive letter and path of the home directory where he stores his customized ICED™ files.

[10] Remember that Q:\ICWIN represents the drive letter and path where you have installed ICED™.

The use of the %HOME.%N.HOME^SAMPLES syntax to define the target path of the copied file in the command on the previous page will allow this command to work on any user's system regardless of the directory structure they have defined with ICED_HOME.

The HOME.*dir_index* macro(s) will always end in '\' to make it easy to form file names. The '^' symbol in the example above is used to delimit the macro name without adding a space to the file name.

When you omit the *dir_index* completely, it is the same as typing HOME.1.

ID.MAX

Each component added to a cell has unique identification number. The id number assigned to the first component in the cell is 1. The id number for a new component is always the previous maximum id number plus 1. Id numbers for deleted components are not reused. These id numbers can be used to select components.

The ID.MAX system macro contains of the id number assigned to the last component added to the current cell. This macro can be used as the maximum id number in a WHILE loop if you need to process each component.

Example:

```
LOCAL #I = 1
WHILE (%I <= %ID.MAX){
        UNSELECT ALL; SELECT ID=%I
        IF (%N.SELECT == 1) {
                ITEM LOCAL #THIS
                !process component
        }
        #I = {%I + 1}
}
```

The ITEM command creates a series of macros that contain information on a single component.

This macro can also be stored in a local macro to keep track of components added since a benchmark id was issued.

Example:

The CMP function compares two strings. See page 224.

The XSELECT mode determines how commands like DELETE behave when no components are selected. See page 215.

```
LOCAL #ID.START = %ID.MAX
LOCAL #RESULT = "SUCCESS"
  ! Assume that this command file is performing some processing
  ! that adds components, but that the results are not always
  ! successful.

!Undo results if the command file was not successful
IF (CMP(%RESULT,"FAILURE")==0) {
        UNSELECT ALL
        SELECT IDS AFTER %ID.START
        XSELECT OFF
        DELETE
}
```

See another example in the ED.CMD and UNED.CMD command files beginning on page 312.

JOU

See page 127 to learn more about the journal file.

This macro contains the fully qualified name of the journal file. This is the file where all commands executed in the current editor session are logged. It can be used to recover from crashes or from mistakes. It can also be used to browse the history of what happened to your layout during an edit session or a command file.

See page 125 to learn more about error blocks.

Let us say that you are developing a complicated command file. If a problem occurs, you may want to browse the journal file to track what happened as your command file executed. You can use the following error block to automatically open a window to browse the journal file and see the exact commands that were executed.

Example:

```
ERR_HANDLER:     DOS "-NOTEPAD.EXE %JOU"
```

The user would need to close the Notepad editor window to continue the command file.

LAST.BOX

This macro is used to record the coordinates of the last box digitized with the cursor.

Example:

```
$ Digitize corners for new BOX.
ADD BOX
ADD BOX OFFSET=(10,10) AT %LAST.BOX
ADD BOX OFFSET=(20,20) AT %LAST.BOX
```

You can use the X0, X1, Y0, and Y1 functions to parse this macro. See page 247.

This command file prompts the user to digitize the corners of a new box component, then adds two more components that are offset from the box digitized by the user. Note that the contents of the line with the $ prefix will be displayed on the bottom of the editor screen while the program waits for the user to digitize the positions for the first ADD BOX command.

The LAST.BOX macro stores the last pair of digitized coordinates from commands other than ADD BOX. The VIEW IN and SELECT IN commands will also set the value of the LAST.BOX macro. However, the ADD POLY command will not set the value of this system macro.

See the LAST.DISP, LAST.POS, and LAST.RULER.POS.*n* system macros for the values of other recently digitized coordinates.

LAST.DISP

The LAST.RULER. POS.*n* system macros can also be used to reuse a previously digitized displacement. See page 273.

This macro stores the last displacement digitized with the cursor. The COPY, MOVE, and VIEW MOVE commands all set the value of this macro. The RULER command does **not** alter the contents of the macro.

The next example makes multiple copies of a single component based on the first displacement digitized by the user.

Example:

```
LOCAL #NUM_COPIES=$PROMPT="How many copies?"
LOCAL #LOOP.N = 2

UNSEL ALL
$ Digitize displacement for multiple copy.
COPY
WHILE (%LOOP.N <= %NUM_COPIES){
        SEL NEW
        COPY BY %LAST.DISP
        UNSEL ALL
        #LOOP.N = {%LOOP.N +1}
}
$SUCCESS: {%LOOP.N – 1} copies made
```

The WHILE command allows a command file to loop. See page 212.

LAST.POS

This macro stores the coordinates of the last single point digitized with the cursor. All commands that allow the user to digitize positions with the cursor set the value of this macro.

Let us say that you are routing wires in your layout. You want a label on each wire. The following command file will automatically add a text component to each wire as you digitize it.

Example:

```
DEFAULT GLOBAL #WIRE_LABEL $PROMPT="Enter label"
$Digitize wire
ADD WIRE LAYER=M1
ADD TEXT "%WIRE_LABEL"  LAYER=M1_TEXT        &
        OFFSET = (0, -.5) AT %LAST.POS
```

A more realistic example using this system macro is shown on page 325.

The origin of the text component will be a slight offset from the last vertex of the wire digitized by the user. This simple example may add labels that are not well placed for naming nodes for the NLE or other programs. If the user always digitizes the wire ending with a horizontal segment digitized left to right, then the labels should be placed correctly. The origin of the text component should be covered by the wire to name the wire.

Repeated uses of the example on the previous page will not prompt the user for the label text. Since the WIRE_LABEL macro is defined with the DEFAULT GLOBAL keywords, the macro will persist until it is deleted with a REMOVE command. The user only has to enter the text for the first use of the command file, then repeated uses of the command file will add the same label.

LAST.RULER.POS.0 and **LAST.RULER.POS.1**

These macros store the coordinates of the two points digitized by the last successful RULER command. The points are not reordered, but are the exact points digitized by the user during the command.

Let us say that your command file needs to remember the coordinates of a bounding box defined by the user. You could create user macros with the $[PROMPT="*string*"]BOX syntax (see page 144), but simply using the RULER command and the LAST.RULER.POS.*n* macros is easier to code.

Example:

```
$Digitize the lower-left corner of the bounding box, then the upper-right corner
RULER
!missing processing that includes user digitizing a position MYPOS
IF (X(%MYPOS) < X(%LAST.RULER.POS.0)) {
        ERROR Position %MYPOS is out of range
}
```

The X() function returns the X coordinate from a coordinate pair. See page 246.

The $comment in the example above is left on the screen while the user digitizes points during the RULER command. The user may digitize many points with various commands before the values in these macros are used, but they will not change until a new RULER command is executed.

The values stored in these system macros will remain valid even if the previous RULER command was executed in a different command file, or outside of any command file.

LAYER.BLANKED.CELL.*layer_spec*

Set the value of this macro with the [UN]BLANK command.

This macro contains a 1 (i.e. TRUE) if layer *layer_spec* is blanked for components in nested cells. If the layer is not blanked for nested cells, the value of this macro is 0 (i.e. FALSE). Specify *layer_spec* using either a layer name or a layer number.

Example:

The _GET_ANS command file is supplied with the installation.

```
LOCAL #MODLAY = "M1"
IF (%LAYER.BLANKED.CELL.%MODLAY){
    @_GET_ANS;                                                    &
      #PROMPT="%MODLAY is blanked in nested cells.  Continue?";   &
      #CHOICES="yn"
    IF (%ret.value !=1) RETURN
    ELSE UNBLANK CELL LAY %MODLAY
}
```

If you need the entire list of layers, use the BLANKED.-CELL.LAYERS system macro. See page 257.

The example above tests to see if a layer is blanked for nested cells. The system macro LAYER.BLANKED.CELL.%MODLAY is used to test this. The macro reference %MODLAY is replaced with the string "M1" as the line is interpreted. So the entire macro reference resolves to "LAYER.BLANKED.CELL.M1". If this layer is blanked, then user is warned and if he/she does not respond with a <y> to the prompt, then the command file is terminated immediately with the RETURN command.

LAYER.BLANKED.ROOT.*layer_spec*

Set the value of this macro with the [UN]BLANK command.

The macro contains a 1 (i.e. TRUE) if layer *layer_spec* is blanked for components in the current cell. If the layer is not blanked, the value of this macro is 0 (i.e. FALSE). Specify *layer_spec* using either a layer name or a layer number.

Example:

```
LOCAL  #REBLANK_NWELL = 0
LOCAL  #UNBLANK_PWELL = 0

IF (%LAYER.BLANKED.ROOT.NWELL){           !NWELL blanked
        UNBLANK ROOT LAYER NWELL
        #REBLANK_NWELL = 1
}
IF (%LAYER.BLANKED.ROOT.PWELL==0){        !PWELL not blanked
        BLANK ROOT LAYER PWELL
        #UNBLANK_PWELL = 1
}
PROMPT="Select NWELL shape"
SELECT LAYER=NWELL NEAR

        !missing statements that manipulate NWELL shape

IF (%REBLANK_NWELL) BLANK ROOT LAYER NWELL
IF (%UNBLANK_PWELL) UNBLANK ROOT LAYER PWELL
```

If you need the entire list of layers, use the BLANKED.-ROOT.LAYERS system macro. See page 257.

The command file above turns off the display of the PWELL layer blanked so that the user can more easily select an NWELL shape to manipulate. The system macro LAYER.BLANKED.ROOT.*layer_spec* is used to test the initial blank status of both the PWELL and NWELL layers so that the original status can be restored at the end of the command file.

LAYER.CIF.LAYER.*layer_spec*

Set the value of this macro with the LAYER command.

This macro contains the CIF layer name for layer *layer_spec*. You can specify *layer_spec* as either a layer name or layer number.

If no CIF layer name is defined for the specified layer, the value of this macro is "NO_CIF".

Example:

```
LOCAL #LOOP.N = 1
LOCAL #CIF_COUNT = 0
$ CIF layer names currently in use
$ number   cif name
WHILE (%LOOP.N <=  255){
        IF ( CMP(%LAYER.CIF.LAYER.%LOOP.N, "NO_CIF")!=0)  {
                $ %LOOP.N   %LAYER.CIF.LAYER.%LOOP.N
                #CIF_COUNT = {%CIF_COUNT +1}
        }
        #LOOP.N = {%LOOP.N + 1}
}
$ %CIF_COUNT layers have CIF names assigned
```

The example command file above will record in the journal file the CIF layer names for all layers that have them defined. Only layers with CIF names assigned will be listed.

LAYER.COLOR.*layer_spec*

Set this value of this macro with the LAYER command.

This macro will contain the color number assigned to layer *layer_spec*. You can specify *layer_spec* as either a layer name or layer number.

Example:

A color can also be selected using a menu. See a related example on page 277.

```
LOCAL #LAYER_NAMES = ""
LOCAL #N= 1
@_GET_INT; #PROMPT="color number"; #MIN=1; #MAX=15
LOCAL #COLOR_NUM = %RET.VALUE
WHILE (%N<= 255){
    IF ( %LAYER.COLOR.%N== %COLOR_NUM)  {
        #LAYER_NAMES = %LAYER_NAMES %LAYER.NAME.%N
    }
    #N= {%N+ 1}
}
RETURN "Layers: %LAYER_NAMES use color %COLOR_NUM"
```

The command file above will loop through all 255 layers and display the names of those that are drawn with the selected color.

LAYER.NAME.*layer_spec*

Set this value of this macro with the LAYER command.

Use this macro to get the layer name from the layer number. The *layer_spec* can be either a layer name or number. While it may seem silly to use a layer name for *layer_spec*, when your command file prompts the user to supply a layer, they may type either a name or a number.

If the indicated layer has no name assigned to it, the layer number is returned.

The example on the previous page demonstrates the use of this macro.

LAYER.NUMBER.*layer_spec*

This macro contains the layer number for the indicated layer. As with the previous macro, *layer_spec* can be either a layer name or a layer number.

Example:

```
LOCAL  #LAYER $PROMPT="Enter layer"
LOCAL  #LAYER_NUM = %LAYER.NUMBER.%LAYER
```

The pair of macro definitions above insures that the layer number of the layer typed by the user is stored in the LAYER_NUM. This is true whether the user types a layer name or a layer number in response to the prompt.

LAYER.PAT.*layer_spec*

Set this value of this macro with the LAYER command.

This macro stores the stipple pattern number used to draw layer *layer_spec* in the screen display. You can specify *layer_spec* as either a layer name or layer number.

Example:

The CHAR function returns a single character from a string.

```
$Select pattern to replace
LOCAL #PAT_STR  $MENU=M1:PATTERN1A             !Display pattern list menu
IF ( LEN("%PAT_STR") == 7) { ! "  pat=n"
    LOCAL #PAT_NUM = {CHAR(7,"%PAT_STR")}
}
    !continued on next page
```

```
ELSEIF ( LEN("%PAT_STR") == 8) {! "  pat=nn"
     LOCAL #PAT_NUM = {CHAR(7,"%PAT_STR")}{CHAR(8,"%PAT_STR")}
}
ELSE ERROR "Invalid pattern returned from menu"

$Select new pattern
LOCAL #NEW_PAT_STR  $MENU=M1:PATTERN1A

LOCAL #LOOP.N = 1
WHILE (%LOOP.N <= 255){

     IF ( %LAYER.PAT.%LOOP.N==%PAT_NUM)  {
          LAYER %LOOP.N %NEW_PAT_STR
     }
     #LOOP.N = {%LOOP.N + 1}
}
```

The PATTERN1A submenu can be browsed in the file Q:\ICWIN[11]\-SAMPLES\-M1A.DAT. See page 147 for more details on using menus for macro definition.

This command file demonstrates how to allow the user to select a pattern from a menu. The user will see the $*comment* displayed below the command line and a sample of each pattern next to its number in the menu area. The pattern number must be parsed from the return string since the PATTERN1A submenu formats the return string as " pat=n" where n is the pattern number. This format is nice when building LAYER commands, but awkward when you need only the number.

Once the old and new patterns are selected, the WHILE loop tests the pattern number of each layer with the LAYER.PAT.*layer_spec* system macro and compares it to the pattern number in PAT_NUM. If the numbers match, the old pattern is replaced with the new one using the LAYER command.

LAYER.PEN.*layer_spec*

Set this value of this macro with the LAYER command.

Use this macro to determine the pen number used to plot layer *layer_spec.* You can specify *layer_spec* as either a layer name or layer number. The LAYER.PEN.*layer_spec* macro will have a value of "-1" for layers that have no pen number assigned to them.

[11] Remember that Q:\ICWIN represents the drive letter and path where you have installed ICED™.

LAYER.PROTECTED.*layer_spec*

Set the value of this macro with the [UN]PROTECT command.

This macro contains a 1 (i.e. TRUE) if the entire layer *layer_spec* is protected. If the layer is not protected, the value of this macro is 0 (i.e. FALSE). (If a layer is protected, it is still visible, but components on that layer cannot be selected.) Specify *layer_spec* using either a layer name or a layer number.

Example:

```
@_GET_LAY
LOCAL  #LAYER = %RET.VALUE

IF (%LAYER.PROTECTED.%LAYER)       &
        RETURN "Layer %LAYER is protected.  Cannot continue."
```

See the PROTECTED_ LAYERS system macro to get a list of all protected layers. See page 290.

The command file fragment above uses the _GET_LAY.CMD command file to prompt the user to enter a layer name or number. After verifying that a valid layer has been entered, this command file stores the layer number in the global macro RET.VALUE. The example then tests that this layer is not protected. If it is protected, this command file fragment prints a warning message on the screen and terminates the command file.

LAYER.SPACE.*layer_spec*

Set this value of this macro with the LAYER command.

Use this macro to determine the spacing distance assigned to the layer specified by *layer_spec*. You can specify *layer_spec* as either a layer name or layer number.

The spacing cursor is enabled by the SPACER command.

The spacing distance is used primarily by ADD commands when the spacing cursor is enabled. However, you can use this spacing value in your command files if desired.

Example:

```
@_GET_REAL;                              &
    #PROMPT="spacing distance";          &
    #DEFAULT=%LAYER.SPACE.%USE.LAYER
#MYSPACE = %RET.VALUE
```

The _GET-_REAL.CMD command file is supplied with the installation.

The example on the previous page uses the system macro USE.LAYER (see page 299) to get the layer number of the current layer. This is then used as the *layer_spec* in the LAYER_SPACE.*layer_spec* system macro reference. The spacing distance of the current layer then becomes the default value used by the _GET_REAL.CMD command file which prompts the user for a real number and verifies the response. If the user does not override the value when the command file is executed, this value is stored as the value of the MYSPACE user macro.

LAYER.STREAM.DATA.TYPE.*layer_spec*

Set this value of this macro with the LAYER command.

Use this macro to determine the stream data type number for layer *layer_spec*. You can specify *layer_spec* as either a layer name or layer number.

Example:

```
$%LAYER.STREAM.DATA.TYPE.M1
```

Typing the command above at the prompt in the editor will display the stream data type for layer name M1. (Note that the command "LAYER M1" will also report this information along with all other layer parameters.)

The LAYER.STREAM.DATA.TYPE.*layer_spec* macro will have a value of "-1" for layers that have no stream data type assigned to them. See the next example for a brief review of how these stream assignments are made.

LAYER.STREAM.LAYER.*layer_spec*

Set this value of this macro with the LAYER command.

This macro stores the stream layer number for layer *layer_spec*. You can specify *layer_spec* as either a layer name or layer number. This macro will have a value of "-1" for layers that have no stream layer number assigned to them.

Example:

```
@%START.CMD
LOCAL #NAMES = ""
LOCAL #N= 1
        !continued on the next page
```

```
WHILE (%N<= 255){
        IF ( %LAYER.STREAM.LAYER.%N> -1)  {
                #NAMES = %NAMES %LAYER.NAME.%N
        }
        #N= {%N+ 1}
}
RETURN "Layers: %NAMES will be exported by STREAM command"
```

This example begins with the execution of the startup command file. (See the START.CMD system macro on page 295.) This is done because no stream layer assignments are saved in a cell file. These definitions should be made in the startup command file, but they are discarded on purpose at the end of an editor session to prevent obsolete layer correspondences. The execution of the current startup command file redefines the stream numbers.

The loop adds the name of each layer with a valid stream layer assignment to the NAMES macro.

LAYER.STREAM.TEXT.TYPE.*layer_spec*

Set this value of this macro with the LAYER command.

This macro stores the text type number for layer *layer_spec*. You can specify *layer_spec* as either a layer name or layer number. Use this macro in the same manner as the LAYER.STREAM.DATA.TYPE.*layer_spec* LAYER.STREAM-.LAYER.*layer_spec* macros.

LAYER.WIRE.WIDTH.*layer_spec*

When you do not specify the wire width in an ADD WIRE command, the wire will be created with the default width assigned to the layer. The default wire width for layer *layer_spec* is stored in this macro. You can specify *layer_spec* as either a layer name or layer number.

Example:

```
LOCAL #LAYER_NUMBERS = ""
LOCAL #LOOP.N = 1
LOCAL #MIN_WIDTH = 2
WHILE (%LOOP.N <= 255){
        IF ( %LAYER.WIRE.WIDTH.%LOOP.N < %MIN_WIDTH)  {
                #LAYER_NUMBERS = %LAYER_NUMBERS %LOOP.N
        }
        #LOOP.N = {%LOOP.N + 1}
}
$ These layer(s) have default widths < %MIN_WIDTH: %LAYER_NUMBERS
```

The command file above will report the numbers of layers that have a default width less than a certain minimum.

LIB.*lib_index*

Indexed system macros are explained more on page 251.

This system macro contains the path of a directory on the cell library search path. You specify which cell library by an index into the cell library table. Indices are integers beginning with 1. The maximum index is stored in the N.LIBS system macro. (See page 288.)

Example:

A '^' is used to delimit macro names without a space. See page 44.

```
WHILE (%N <= %N.LIBS) {
        #PATHS = "%PATHS^%LIB.%N;"        !quotes are important here
        #N = {%N + 1}
}
```

The cell library paths are defined with the ICED_PATH environment variable. See page 106.

The example above will create a string in macro PATHS with all fully qualified cell library path names delimited with ';'. When complete, the value in PATHS will look similar to:

```
"Q:\ICWIN\TUTOR\;Q:\ICWIN\SAMPLES\;"
```

Note that the path name in each LIB.*lib_index* macro ends with a trailing '\'. This makes it easier to build file names.

(Note that in the example above the quotes are required around the #PATHS macro assignment statement. If they were not used, the trailing ';' would have been interpreted as a command delimiter and stripped from the string before execution.)

LIB.TYPE.*lib_index*

Indexed system macros are explained more on page 251.

This system macro contains a status code indicating the type of protection defined for a specific cell library. Valid values for this system macro are shown in Figure 35.

The cell library is specified by its index into the cell library table. Valid indices are integers beginning with 1. The maximum index is stored in the N.LIBS system macro. (See page 288.)

Value	Meaning
1	The library is read-only. Cells cannot be edited then saved.
2	The library is copy-edit. If a cell is edited, it will be saved to the current directory rather than its original library.
3	The library is direct-edit. Cells can be edited and saved in the original library.

Figure 35: Possible values for the CELL.EDIT.*cell_index* macro

This next example expands the example in the LIB.*lib_index* system macro description. It builds a delimited list of all cell libraries with the appropriate protection character suffixes suitable for use as an ICED_PATH environment variable definition.

Example:

The N.LIBS system macro contains the number of cell libraries. See page 288.

A '^' is used to delimit a macro name without a space. See page 44.

```
LOCAL #N=1
GLOBAL #PATHS = ""
WHILE (%N <= %N.LIBS) {
        #PATHS = "%PATHS^%LIB.%N"
        IF (%LIB.TYPE.%N == 2){              !copy-edit library
                #PATHS = "%PATHS^/C;"
        }
        ELSEIF (%LIB.TYPE.%N == 3){          !direct-edit library
                #PATHS = "%PATHS^/D;"
        }
        #N = {%N + 1}
}
```

LIST.EMPTY.*list_name*

See the LIST command on page 182 to learn more about lists.

Use this system macro to determine if a list is empty. Specify the list by its name.

Example:

```
LOCAL #LNAME = MYLIST
LIST GLOBAL %LNAME          !puts selected components in list MYLIST
IF(%LIST.EMPTY.%LNAME)  RETURN "No items in list."
```

The example above creates a list with the name MYLIST. The IF command tests to see if this list is empty. If the list is empty, the command file is terminated by the RETURN command.

A list can be empty even when the LIST.LEN.*list_name* macro (see 286) is non-zero. The value stored in LIST.LEN.*list_name* does not change when items are deleted. However, the LIST.EMPTY.*list_name* macro will become true when the last component on the list is deleted.

LIST.EOL.*list_name*

This system macro stores a flag for the indicated list.

- It is set initially to "-1" for a new list.
- It is set to FALSE (the number 0) every time a SELECT LIST command successfully selects a component.
- It is set to TRUE (the number 1) when a SELECT LIST command fails to select a component from the list. This is the case when you have passed the **E**nd **O**f the **L**ist.

Example:

Remember that "==" is required to test if two values are equal.

```
LIST GLOBAL %LNAME
UNSEL ALL; SEL LIST %LNAME FIRST
WHILE(%LIST.EOL.%LNAME==0) {
        !process selected component
        UNSEL ALL; SEL LIST %LNAME NEXT
}
```

The example on the previous page is a typical loop used when you need to process all components in a list one at a time. The WHILE condition of '%LIST.EOL.%LNAME==0' prevents the commands in the WHILE block from being executed unless a component was successfully selected from the list.

The first time a SELECT command passes the end of the list, this system macro flag is set to '1'. If another similar SELECT command is executed after the flag is set, an error occurs and the command file will be terminated.

If you reverse direction with the next SELECT LIST command (e.g. SEL LIST %LNAME PREV) and a component is successfully selected, then the flag stored in this macro is reset back to FALSE, '0'.

LIST.INDEX.*list_name*

The LIST.INDEX.*list_name* macro stores the current list index for the indicated list. Use the SELECT LIST (NEXT|PREV|FIRST|LAST) commands to change the current list index.

Example: **IF (%LIST.INDEX.MYLIST <= (%LIST.LEN.MYLIST / 2)) {…**

This example will test if the current list index is in the first half of the list, or the second half.

When a list is created by the LIST command, each component is assigned an index. The list is sorted by ID number (i.e. the component having the lowest ID number will have a list index of 1, the second lowest ID number is assigned an index of 2 etc.) The last has an index of *n*, where *n* is the number of components on the list. (See the LIST.LEN.*list_name* macro.) Deleting a component on the list (or otherwise removing it with a MERGE or GROUP command) does not affect the list indices of any of the remaining components on the list.

For a new list, the value of the macro is 0. SELECT LIST commands change the value of the macro in the following ways:

SELECT LIST *list_name* FIRST sets the index macro to 1.

SELECT LIST *list_name* NEXT adds 1 to the index. Then the command tests to see that the component with this index exists.

If it does it is selected. If it does not, the SELECT command adds 1 to the index and tries again. Once the index is greater than *n*, the LIST.EOL.*list_name* macro is set to 1.

SELECT LIST *list_name* LAST sets the index macro to *n*.

SELECT LIST *list_name* PREV subtracts 1 from the index. Then the command tests to see that the component with this index exists. If it does it is selected. If it does not, the SELECT command subtracts 1 from the index and tries again. Once the index is set to 0, the LIST.EOL.*list_name* macro is set to 1.

LIST.LEN.*list_name*

This macro will contain the number of components in the indicated list. If components are deleted after the list was created, the value of the macro does **not** change.

See the previous page for an example of this macro.

MAX.CELL

See page 252 for an overview of the cell table and cell table indices.

This macro stores the last cell table index used. You can use this macro in a WHILE loop to test that every cell currently loaded has been processed. See an example of this on page 109.

MAX.COORD

This macro stores the maximum valid coordinate in user units. If you use a value outside of the range +%MAX.COORD : -%MAX.COORD in any command that requires coordinates, an error is generated.

Example:

```
LOCAL #A =249996
IF (%A + 5 > %MAX.COORD) &
        PAUSE "End of drawing boundary passed!"
ELSE  &
        ADD BOX (%A, %A) ({%A + 5},{%A + 5})
```

The example above demonstrates how MAX.COORD might be used. In a typical design, MAX.COORD is set to 250,000. (We'll go into the specifics in just a moment.) So %A + 5 will be greater than MAX.COORD. The command file will execute the PAUSE command alerting the user to this situation rather than the ADD BOX command that would generate an error and terminate the command file.

The actual value of this macro depends on the version of ICED™ and the value of the NDIV command line parameter. The current version of the layout editor stores coordinate values as 32-bit integers. A 32-bit integer can represent numbers in the range from -2,147,483,647 to +2,147,483,647. (Future versions will support 64-bit coordinates.)

See the NDIV command line option in the IC Layout Editor Reference Manual for more details on user units and database units.

A coordinate is stored as an integer number of database units, where the number of database units in one user unit is given by the NDIV command line parameter (1000 by default.)

The current version of ICED™ only allows coordinates in the range of -250,000,000 to +250,000,000 database units. So a typical design can store coordinates in the range -250,000 to +250,000 user units, and MAX.COORD will be set to 250,000.

MENU

This macro contains the name of the currently loaded menu. The name is not fully qualified with directory or file extension. If a submenu is currently loaded, then the value of this macro will be of the form *menu_name:submenu_name*.

When a new menu is loaded with the MENU or SHELL MENU commands, this changes the value of the MENU system macro.

N.CMD.DIRS

This system macro contains the number of command file directories. These directories are searched automatically for command files by the program.

Example:

```
WHILE (%N <= %N.CMD.DIRS) {
        IF (FILE_EXISTS( %CMD.DIR.%N^%MYFILE)) {
```

See the WHILE command on page 212.

See the FILE_EXISTS function on page 227.

This example is expanded in the CMD.DIR.*dir_index* system macro description on page 265. See that description for more details on how command file directories are defined and used.

N.DRC.DIRS

This system macro contains the number of directories searched for DRC (Design Rule Checker) files.

Example:

```
SPAWN "-NOTEPAD.EXE %DRC.DIR.%N.DRC.DIRS^TEMP.RUL"
```

This example opens a temporary rules file in the last directory on the DRC search path with the NOTEPAD.EXE text editor.

See another example of this system macro in the DRC.DIR.*dir_index* description on page 266. Refer to that description for more details on the DRC file search path.

N.LIBS

This system macro stores the number of ICED™ cell libraries. These cell library directories are automatically searched for cell files.

Example:

```
IF ( %COUNTER > %N.LIBS) RETURN "All cell libraries processed"
```

See another example of this system macro in the LIB.*lib_index* description on page 282. Refer to that description for more details on cell libraries.

N.HOME

To learn more about home directories, refer to Defining Environment Variables in the IC Layout Editor Reference Manual or see Q:\ICWIN\DOC \TREES.TXT.

This system macro contains the number of ICED™ home directories. These directories are defined with the ICED_HOME environment variable. These directories are searched for a variety of program support files.

Most installations have a single home directory, the Q:\ICWIN[12] directory. However, you can specify more than one directory in the ICED_HOME environment variable definition. In this case, each of these directories will have a HOME.*n* system macro, where *n* is the index of the directory in the ICED_HOME environment variable definition.

Example: **DOS '^COPY X:\PROJX\TEST.CEL %HOME.%N.HOME^SAMPLES'**

This example is used in the HOME.*dir_index* description on page 268. Refer to that description for an explanation if you desire.

N.SELECT

This macro contains the number of currently selected components. See examples using this macro on pages 91 and 96.

NEW.CELL

This macro will contain TRUE ("1") if the cell you are currently editing was just created. It will contain FALSE ("0") if the cell was created in an earlier ICED™ session.

[12] Remember that Q:\ICWIN represents the drive letter and path where you have installed ICED™.

Example:

```
EDIT CELL %MYCELL
IF (%NEW.CELL == 0){
        PAUSE "Cell %MYCELL already exists!"
        QUIT
        RETURN
}
```

A cell is "new" if it was created by using a new cell name on the command line when the editor was launched, or when the cell was just created with the EDIT command.

PROTECTED.LAYERS

Set the value of this macro with the [UN]PROTECT command. Refer to the IC Layout Editor Reference Manual.

The macro contains the list of protected layers. Components on protected layers cannot be selected or modified, but they remain visible.

The list is stored in layer list format using the layer numbers. (See some examples of layer lists on page 257.) If there are no protected layers, then value of the macro will be "(none)". You should check for this value before using the string in another command.

Example:

To find out about a specific layer, use the LAYER.PRO-TECTED-.*layer_spec* system macro. See page 279.

```
LOCAL #PROTECTED_LAYLIST = %PROTECTED.LAYERS
UNBLANK ALL; UNPROTECT ALL
        !missing processing that requires all layers to be selectable
!restore protected status of layers
IF (CMP("%PROTECTED_LAYLIST","(none)") != 0)      &
  PROTECT LAYERS=%PROTECTED_LAYLIST
```

RES.MODE

Set the value of this macro with the RESOLUTION command.

This macro stores the resolution mode in effect. The value will be one of the strings "SOFT" or "HARD". When the resolution mode is soft, vertices calculated by commands like ADD or CUT will not be forced to lie on the resolution grid. When the mode is HARD, vertices will be rounded to lie exactly on the resolution grid. In either mode, ICED™ does not round vertices to lie on the resolution grid when they are explicitly defined in a command. You may want to use this macro to test whether or not you should round vertices calculated by your command file to lie on the resolution grid.

Example:

See a simple example that builds a spiral on page 148.

ROUND is a function to resolve a vertex to the resolution grid. See page 235.

```
! SPIRAL.CMD  Create spiral shape
LOCAL #SP.LAYER $PROMPT="Enter layer."
LOCAL #SP.RADIUS $PROMPT = "Enter radius."
LOCAL #SP.POS_LIST = ""
LOCAL #SP.CENTER = %LAST.POS
LOCAL #SP.VERTEX = ""

#SP.VERTEX = %SP.CENTER
WHILE ….
.
. !Sorry this example does not include real vertex calculation code
.
        IF (CMP(%RES.MODE, "HARD")==0){  !True if mode is HARD
                #SP.VERTEX = {ROUND(%SP.VERTEX)}
        }
        #SP.POS_LIST = %SP.POS_LIST %SP.VERTEX
}
ADD POLY LAYER=%SP.LAYER AT %SP.POS_LIST
```

The fragments of the command file shown above are part of the process to create a spiral polygon shape. The actual steps to calculate the vertex coordinates are missing. However, enough of the process is shown to demonstrate how you would use the RES.MODE macro and the ROUND function to resolve each calculated vertex to the resolution grid if the resolution mode is HARD.

RES.STEP

Set this value of this macro with the RESOLUTION command.

This macro stores the step size of the resolution grid. The resolution grid consists of all coordinates that can be digitized with the mouse. This grid is usually set by the startup command file according to technology requirements for the resolution of vertex data. ICED™ does not force vertices to lie on this grid when you explicitly define coordinate data in a command. You may want to use this value in command files that create or modify components with calculated vertex data so that the vertices will not be altered by post-processing software.

Example:

The ABS function returns the absolute value of a number. See page 220.

The X and Y functions return the X or Y coordinate of a coordinate pair. See page 246.

```
#X_DISP = { ABS(X(%VERTEX1) - X(%VERTEX2)) }
#Y_DISP = { ABS(Y(%VERTEX1) - Y(%VERTEX2)) }
IF ( (%X_DISP < %RES.STEP) || (%Y_DISP < %RES.STEP) ){     !Boolean OR
            .
            .  !Processing to remove or recalculate the shape
            .
}
```

This command file fragment shows how the RES.STEP macro might be used to determine if a shape is less than the minimum resolution in either direction.

Also see the related RES.MODE and SNAP.STEP system macros.

SELECT.BOX

This macro contains the bounding box of all of the currently selected components. A bounding box is the smallest rectangle square with the axes that surrounds all of the items.

Example:

```
IF (   (%X_COORD < X(%SELECT.BOX))    ||     &
       (%X_COORD > X(%SELECT.BOX))    ||     &
       (%Y_COORD < Y(%SELECT.BOX))    ||     &
       (%Y_COORD > Y(%SELECT.BOX))    ){
```

The example above tests if the vertex (%X_COORD, %Y_COORD) is outside of the box that just contains all of the currently selected components.

When no components are selected, the value of this system macro is "(0,0) (0,0)".

SHORT.CELL

In older versions of ICED™, this system macro contained the name of the current cell shortened to 8 characters to facilitate naming files in the DOS 8.3 format.

Now that this is no longer required in newer operating systems, this macro now contains the same string as the CELL system macro. It is retained to support older command files.

SHORT.ROOT

The root cell is the cell ICED™ was launched to edit.

In older versions of ICED™, this system macro contained the name of the root cell shortened to 8 characters to facilitate naming files in the DOS 8.3 format. Now that this is no longer required in newer operating systems, this macro now contains the same string as the CELL.ROOT system macro. It is retained to support older command files.

SNAP.ANGLE

SNAP.OFFSET

SNAP.STEP

Set the value of these macros with the SNAP command.

See the RES.STEP system macro for the finest grid for coordinate data.

These macros contain the snap grid settings in effect. When the cursor is used to define positions, these settings control what positions can be digitized.

ICED™ does not force vertices to lie on this grid when you explicitly define coordinate data in a command. You may want to use these macros in command files that calculate vertex positions if you want all of your vertices to lie on the snap grid.

When you need to temporarily reset the snap grid settings, you can copy one of these macros to a local macro to store the initial value of a setting. Restore the value with the SNAP command and the value stored in the local macro at the end of the command file.

SNAP.ANGLE controls the allowable angles between successive vertices in a wire or polygon. The possible values of this macro are 90, 45, and 0. When the snap angle is 0, any angle is allowed.

SNAP.OFFSET defines the origin of the snap grid. The value in this macro will always be a coordinate pair.

SNAP.STEP stores the minimum distance between points on the snap grid. Unlike the RES.STEP system macro, the snap step may be different in the X and Y directions. This macro is a coordinate pair containing the minimum distance between digitized points in the X and Y directions.

Example:

```
LOCAL #MULT = 10

#VERTEX1 = %SNAP.OFFSET

IF (%SNAP.ANGLE == 0) {
        #VERTEX2 = {%VERTEX1 + (%SNAP.STEP * %MULT)}
        #VERTEX3 = ""
}
ELSE {          ! (%SNAP.ANGLE equals 90 or 45)
        #VERTEX2 = {%VERTEX1 + ( 0, Y(%SNAP.STEP) * %MULT )}
        #VERTEX3 = {%VERTEX2 + (X(%SNAP.STEP) * %MULT , 0 )}
}
```

The X and Y functions return a single coordinate from a pair. See page 246.

The command file fragment above demonstrates how you might create vertices differently depending on the value of the snap angle.

SPACER.ON
SPACER.SPACE
SPACER.STYLE
SPACER.TRACK.LAYERS

The LAYER.SPACE *.layer_spec* system macro contains the spacer distance setting for a specific layer. See page 279.

These macros store the settings in effect for the SPACER command. (Refer to the IC Layout Editor Reference Manual for a complete description of this command.) The macros control how the spacer cursor is drawn while adding wires, boxes, or polygons.

SPACER.ON	"ON", or "OFF", "ON" only when the spacer cursor is enabled
SPACER.SPACE	The current spacing distance in user units
SPACER.STYLE	The integer that specifies the spacing cursor style
SPACER.TRACK.LAYERS	"ON", or "OFF", "ON" only when the spacer cursor distance is automatically changed when the default layer is changed.

The CMP function compares two strings. See page 224.

You cannot set these values directly. Use the SPACER command to set them.

Example:

```
IF (CMP(%SPACER.TRACK.LAYERS, "OFF") == 0) &
        SPACER SPACE = %LAYER.SPACE.%USE.LAYER
```

The USE.LAYER system macro contains the current layer number.

The example above changes the spacer guide to the minimum distance assigned to the current layer. This is done only if automatic spacer tracking is set to "OFF". Otherwise, ICED™ would have done this automatically when the current layer was changed.

START.CMD

Set the value of this macro with the STARTUP command line option when the editor is launched.

This macro contains the path and filename of the startup command file. This is the command file that is executed automatically in every newly created cell. It often defines layer and grid properties as well as key shortcut definitions and many other ICED™ settings.

Example:

@%START.CMD

This command will execute the current startup command file. Another method to execute this command is to use the **menu option** @START.

See the ALWAYS.CMD system macro for the value of the always command file.

If no startup command file is defined, then this system macro will contain the string "DO_NOTHING". It is not a syntax error to execute the command @DO_NOTHING, but the command will have no effect.

SUBCELL.EDIT.*cell_index*

See page 104 for a definition of the cell table.

This macro is used to determine the edit status of a subcell. The value of each macro informs you whether or not you can edit the specified cell and then save it. This macro differs from the CELL.EDIT macro primarily in that non-zero values are stored only for subcells of the current cell.

Refer to an overview of cell indices on page 252.

Specify the subcell by setting *cell_index* to the index into the cell table. *cell_index* must be greater than or equal to 1.

If you want to use this system macro to get the edit status of a cell by using the cell name, you must first use the CELL function to obtain the cell index. See an example on page 262.

MARK_SUB-CELLS can restrict the cells marked to those containing shapes on certain layers.

The data for this macro must be initialized by execution of the MARK_SUBCELLS command in the current layout editor session. (See page 197.) When your command file uses a loop to determine the edit status of each loaded cell, the SUBCELL.EDIT macro is more efficient than the CELL.EDIT macro. This is because the MARK_SUBCELLS command creates a table of the edit status for each loaded cell with one trip through the cell database. The CELL.EDIT macro interrogates the entire cell database each time it is used to get current information about the indicated cell. However, the SUBCELL.EDIT macro uses the information stored by the MARK_SUBCELLS command, even if that data is no longer current.

Refer to page 106 for definitions of view-only, copy-edit and direct-edit libraries.

Value	Meaning	
0	The cell is not a marked subcell of the current cell, the cell is already open, or the *cell_index* does not refer to a valid cell.	
1	The cell is a subcell of the current cell	but it is in a read-only library which cannot be edited.
2		and it is in a copy-edit library so the modified cell file will be saved to the current directory rather than its original library if you edit then save it.
3		and it is directly editable.

Figure 36: Posible values for the SUBCELL.EDIT.*cell_index* macro

Example:

```
! _LOOP.CMD (Simplified version)
! Execute command string in all subcells of current cell

XSELECT OFF
LOCAL #loop.n = 1;                          ! define counter macro
                                            ! define command string
DEFAULT GLOBAL #loop.op = "VIEW ALL"

MARK_SUBCELLS                               ! initialize subcell.edit macros
WHILE(%loop.n <= %max.cell){                ! loop through each cell
      $ subcell.edit.%loop.n=%subcell.edit.%loop.n    ! adds comment to log
      IF(%subcell.edit.%loop.n==3){         ! get edit status
           EDIT CELL %cell.name.%loop.n;    ! edit this cell if status = 3
           %loop.op;                        ! execute command string
           LEAVE;                           ! save cell if data has changed
      }    ! end of if block
      #loop.n = {%loop.n + 1};              ! increment counter
}                                           ! end of loop
```

This command file is supplied with the installation.

MAX.CELL is a system macro containing the last valid index into the cell table. See page 286.

The LEAVE command will save the cell file only if the cell has been modified.

This command file will execute the commands in the string stored in the user macro loop.op in every subcell of the current cell that has its cell file stored in a direct edit library.

The commands in loop.op will be executed only in cells that have an edit status of 3 stored for them by the MARK_SUBCELLS command. No open cells will be modified by this command file since the value of SUBCELL.EDIT.*cell_index* macro will always be 0 for open cells. If you execute this command file while editing a nested cell, all cells at higher levels of cell hierarchy, including the root cell, will not be modified.

If you want to execute loop.op in all editable subcells of the root cell, be sure to close all open cells, and execute loop.op on the root cell for this command file to have the desired effect.

Since loop.op is defined with the DEFAULT keyword, if a different loop.op is already defined, that command string will be executed instead.

Example:

```
@_LOOP;GLOBAL #loop.op =                                  &
            "UNSEL PUSH; SEL LAYER M1 ALL;                &
            SWAP LAYER M1 AND M2; UNSEL POP;";
LOOP.OP     !executes the command string in the current cell too
```

If the _LOOP.CMD command file is executed with the statement above, this definition of the loop.op macro will be executed in each subcell. The commands in loop.op in this case are surrounded by a pair of SELECT commands that save the original selection status of components in the cell and then restore that original selection status. The "SEL LAYER M1" and "SWAP LAYER M1 AND M2" commands will result in components on the M1 layer being moved to the M2 layer.

TIMER

This macro contains the number of seconds since the start of the edit session.

Example:

```
LOCAL #START_TIME = %TIMER

WHILE (1){
        IF ( (%TIMER - %START_TIME) >= 30) {
                RETURN;$ TIMED OUT
        }
        !missing lengthy processing
}
$SUCCESS
```

This command file shows an infinite loop that will time out after 30 seconds. In a real command file, you would have some more realistic condition in the WHILE statement, and some processing in the loop.

TMP

Set the value of this macro with the TMP command line option.

This macro contains the path to the directory used by ICED™ to store temporary files. If your command file needs to create temporary files, it would be a good practice to store them in this directory.

See the ED.CMD and UNED.CMD examples on page 312 for an example using this system macro.

USE.ARC.TYPE

The USE command sets the value of this macro.

This system macro contains the default wire type for arc components. ADD ARC commands will use this wire type by default. See the ADD WIRE command in the IC Layout Editor Reference Manual for more information on wire types. See the table on page 300 for possible values of this system macro.

USE.LAYER

The USE command or 1:USELAY menu option sets the value of this macro.

This system macro stores the layer number of the current layer. This is the layer used by default in any ADD command that does not specify the layer for a new component.

Example:

```
$ The default layer number is %USE.LAYER, &
        the layer name is %LAYER.NAME.%USE.LAYER
```

See the LAYER-.NAME.*n* system macro on page 277.

USE.N.SIDES

This system macro contains the N.SIDES parameter set by the USE command. This value is used by default when an ADD command is used to create a new CIRCLE, RING, ARC, or SECTOR component. This value determines the number of sides used to approximate a full circle.

Example:

```
IF (%USE.N.SIDES < 8) USE NSIDES=8
```

USE.TEXT.JUST

This system macro contains the text justification parameter set by the USE command. This value is used as a default when the ADD command is used to create a new text component. The two-letter code determines the location on the text component bounding box used as the origin of the component.

Example:

```
LOCAL #MYCMD =                      &
        'ADD TEXT "MY TEXT"    &
        JUST=%USE.TEXT.JUST'
```

If the current value stored in USE.TEXT.JUST is "CC", then the user macro definition above will result in the string 'ADD TEXT "MY TEXT" JUST = CC'.

just_code	**Default text justification**
LB	Bottom Left
LC	Center Left
LT	Top Left
CB	Bottom Center
CC	Center
CT	Top Center
RB	Bottom Right
RC	Center Right
RT	Top Right

Figure 37: Text justification codes

USE.WIRE.TYPE

This system macro contains the default wire type set by the USE command. ADD WIRE commands will use this wire type by default. See the ADD WIRE command in the IC Layout Editor Reference Manual for more information on wire types.

Value	Wire type
0	Flush ends
2	Extended ends

Figure 38: Valid values of USE.WIRE.TYPE

Example:

```
LOCAL #ORIG_WIRE_TYPE = %USE.WIRE.TYPE

USE WIRETYPE 2
.
.       ! missing commands that add and manipulate wire components
.
USE WIRETYPE %ORIG_WIRE_TYPE
```

The command file fragment above will save the default wire type before resetting it with the USE command. The original value is restored at the end of the command file.

VIEW.BOX

See page 115 for an example that uses this macro to save and restore the view window.

This macro contains the coordinates of the corners of the current view window. The first coordinate pair will be the lower left corner of the view window; the second is the upper right corner.

Any VIEW command, or panning of the screen display, will change the value of this macro. The VIEW BOX command explicitly sets the value using a pair of coordinates.

Example:

The POS1 and POS2 functions are described on page 233. The X and Y functions are described on page 246.

```
#LL_VIEW_CORNER =  {POS1(%VIEW.BOX)}          !lower left corner
#UR_VIEW_CORNER =  {POS2(%VIEW.BOX)}          !upper right corner

#X_COORD = {X(%VERTEX)}
#Y_COORD = {Y(%VERTEX)}

IF (  (%X_COORD < X(%LL_VIEW_CORNER))     ||      &
      (%X_COORD > X(%UR_VIEW_CORNER))     ||      &
      (%Y_COORD < Y(%LL_VIEW_CORNER))     ||      &
      (%Y_COORD > Y(%UR_VIEW_CORNER))     ){

         VIEW CENTER %VERTEX  !Set center of new view window
}
```

The command file fragment above tests if the coordinates stored in the local macro VERTEX are inside the current view window. The Boolean OR operator ("||", see page 56) is used to test that each coordinate is within the view window. If the vertex is not inside the current view window, the VIEW CENTER command is used to change the view window so the vertex is at the center of the view window.

VIEW.CENTER

Any VIEW command, or panning of the screen display, will change the value of this macro.

This system macro stores the coordinates of the center point of the current view window.

Typing the command below at the command prompt will report the coordinates of the center of the view window.

Example:

```
$%VIEW.CENTER
```

VIEW.SCALE

This system macro stores the scale of the current view window scale in pixels per user unit. Use the VIEW command to set this value.

Example:

```
VIEW ON
IF (%VIEW.SCALE < 7.5) VIEW SCALE =7.5
```

The GRID command controls the settings of the display grid.

The VIEW ON command is used in this example since most changes made to the view window in a command file are not reflected on the screen unless the VIEW mode is on. The view scale is changed only when the view window is zoomed out too far to digitize coordinates accurately. A view scale of 7.5 is approximately the minimum scale that allows a 1 user unit display grid visible.

Advanced Examples

The command files described below are supplied with your ICED™ installation. There are many others in the following locations that you may find useful.

Some of the command files are **technology independent**. Look for these files in the **Q:\ICWIN**[13]**\AUXIL** directory.

Other command files are **technology dependent**. These files are stored in **Q:\ICWIN\TECH\SAMPLES**. If you want to use one of these in a project using a specific technology, follow the following procedure:

1) Create a new subdirectory of the Q:\ICWIN\TECH directory. E.G. "Q:\ICWIN\TECH\BIPOL5".
2) Copy the desired file from Q:\ICWIN\TECH\SAMPLES to this new directory.
3) Edit the file as needed to customize it for your technology. The kind of values that may need editing include layer names, minimum sizes, or wiring pitches.
4) Add this new directory to the command file search path. See page 14 for details.

See the table on page 9 for a list of examples in this manual that you may find useful your own command files.

The first example has a command file name that begins with a '_' prefix. Command files use this prefix to keep them out of the lists of command files that appear in the 3:@%.cmd menu option. They are intend to be called from other command files and are not useful when called directly from the layout editor. Use them as required in your own command files.

The other examples can be called directly from the editor command line, or selected using the 3:@%.cmd menu option. Some of them are so useful, you may use them more often than many editor commands. DEEPSHOW.CMD is right on the menu as the 2:(SHOW)@deep menu option. You may also find referring to them very useful when writing your own command files.

While these examples have been tested in a variety of situations, we cannot guarantee that they will work in all situations. If you use these command files, be sure to verify your results. Please let us know if you find any bugs so that we can improve the examples in the next version of this manual.

[13] Remember that Q:\ICWIN represents the drive letter and path where you have installed ICED™.

_GET_INT.CMD Prompt user for integer and verify value

See this command file used in the advanced example SERIAL.CMD on page 312.

This command file prompts the user for an integer and loops until it verifies that a valid integer has been entered.

This command file demonstrates how to make a command file more useful in different situations by defining several macros with default values that can be overridden on the command line that calls the command file. (An example of how to override these defaults follows the command file.) When these macros are not overridden in the @_GET_INT statement, defaults are used.

Local macro to override	Default value
MIN	-2000000000
MAX	20000000000
PROMPT	"integer"
DEFAULT	No default

Figure 39: Defaults for _GET_INT.CMD

See a table of other similar command files on page 88.

```
REMOVE RET.VALUE
LOCAL #VALID=0
LOCAL #VALUE=""
DEFAULT LOCAL #PROMPT="integer"

! Add default value to prompt

IF(MACRO_EXISTS(#DEFAULT)==2)    &
           #PROMPT="Enter %PROMPT [%DEFAULT]:"
ELSE       #PROMPT="Enter %PROMPT:"

! If min or max were not passed as arguments,
! use "infinity"

IF(MACRO_EXISTS(#MIN) < 2) LOCAL #MIN=-2000000000
IF(MACRO_EXISTS(#MAX) < 2) LOCAL #MAX=2000000000

! Prompt user for integer

WHILE(%VALID==0){
   #VALUE=$PROMPT="%PROMPT"
```

```
    IF(  (CMP("%VALUE", "")==0) &&                &
         (MACRO_EXISTS(#DEFAULT)==2)){
      #VALUE = %DEFAULT;
    }
    #VALID = {VALID_INT("%VALUE")}
    IF(%VALID){
       IF((%MIN > %VALUE) || (%MAX < %VALUE)){
          #VALID=0
          $ Value (%VALUE) out of range [%MIN:%MAX]
          PAUSE 0
       }
    }
    ELSE{
       $Invalid input (%VALUE)
       PAUSE 0;
    }
}
GLOBAL #RET.VALUE=%VALUE
```

The command file begins by deleting the global macro used to return the integer at the end of the command file. This macro is removed since the macro RET.VALUE may still exist from a previous use of the command file. If the command file fails, or is cancelled by the user, the macro will not exist. You can test for this by using the SHOW command after control returns to the editor.

The local macros are defined next. VALID is a macro that contains a flag used to determine whether or not a valid integer has been entered. This macro will be used to control the WHILE loop.

The PROMPT local macro contains the default description for the value requested. Since this macro is defined with the DEFAULT keyword, you can override this default by calling the _GET_INT command file with the following syntax:

@_GET_INT; #PROMPT = "number of sides"

When you call the command file this way, the user will be prompted with a message similar to "Enter number of sides:" rather than "Enter integer:".

The next block of lines builds the entire prompt message that will be used. How this message is built depends on whether or not you have defined a local macro with the name DEFAULT. The MACRO_EXISTS() function returns 2 if the named macro exists and is defined as a local macro. If some global macro (or a local macro in a different command file) exists with the same name, that value will **not** be used. Only when the macro is local to this command file will the value stored in DEFAULT be used to build the prompt.

When you call _GET_INT with the following command:

```
@_GET_INT; #DEFAULT = 12
```

the prompt will be "Enter integer [12]:". If you do not define DEFAULT on the same line that calls the command file, then the prompt will be "Enter integer:"

The next block of lines uses the MACRO_EXISTS function to default the MIN and MAX macros to the equivalent of –∞ and +∞. However, if local macros with those names do exist, the values in those local macros will be used. You would define MIN and MAX on the _GET_INT command line in the same manner as the PROMPT and DEFAULT macros.

The WHILE loop comes next. This loop will continue to prompt the user for the integer using the prompt line built above until a valid integer is stored in the macro VALUE.

Note that the $PROMPT keyword allows you to prompt the user to define the value of a macro with the keyboard in a macro assignment statement.

The next IF block assigns the value stored in the macro DEFAULT to the macro VALUE if the user presses <Enter> without typing a value **and** the macro DEFAULT exists.

The next line verifies that the string stored in the VALUE macro represents a valid integer. The function VALID_INT will set the macro VALID to 1 if the macro VALUE contains a single integer. Otherwise VALID_INT sets VALID to 0.

The next block of lines tests that the value is within the range defined by the MIN and MAX macros. If value is outside of this range, VALID is reset to 0. The PAUSE command allows the comment "Value (%VALUE) out of range [%MIN:%MAX]" to remain on the screen until the user presses <Enter>.

If VALID is still equal to 0, the WHILE loop will execute again. Otherwise, the loop is terminated and the string stored in VALUE is used to define the global macro RET.VALUE. This is the macro you would use to refer to result of the command file.

Example:

```
@_GET_INT;   #MIN = 1;                                   &
             #MAX = 255;                                 &
             #PROMPT = "positive integer less than 255"; &
             #DEFAULT =6
#INT1 = %RET.VALUE
```

When this example is used to call the _GET_INT.CMD command file, the user will be prompted with the line "Enter positive integer less than 255 [6]:". The command file will loop until the user enters an integer in the specified range or presses <Enter> without entering any value. Since the command line defines the default value as '6', that value will be stored in INT1 when no value is entered by the user.

RES.CMD Create resistor from resistance and width

RES.CMD will add several shapes representing a polysilicon resistor with contacts to the current cell. The resistor is resized to have a resistance equal to the value entered by the user of the command file at execution time.

This command file requires the cell RES.CEL distributed with ICED™.

```
!---------------------------------------------------
! Set technology dependent parameters
!---------------------------------------------------
!
LOCAL OHMS_PER_SQUARE   = 40;
LOCAL UNSTRETCHED_LEN   = 6;
LOCAL UNSTRETCHED_WIDTH = 4;
LOCAL RES$CELL    = RES;
LOCAL SCRATCH = RES$TMP$;
!
!----------------------------------------------------
! Prompt for parameters for this resistor
!----------------------------------------------------
!
LOCAL OHMS  $PROMPT="Enter target resistance:"
LOCAL WIDTH $PROMPT="Enter channel width:"
LOCAL ROT_CODE -1;
WHILE(CMP(%ROT_CODE, 0)!=0 && CMP(%ROT_CODE, 1)!=0){
   #ROT_CODE  $PROMPT="Enter rotation code
(0=>Horizontal 1=>Vertical):"
}
!
!---------------------------------------------------
! Compute values
!---------------------------------------------------
!
LOCAL STRETCH.Y = {ROUND(%WIDTH - %UNSTRETCHED_WIDTH)}
LOCAL STRETCH.X =                       &
```

```
  {ROUND(%OHMS * %WIDTH /                  &
         %OHMS_PER_SQUARE - %UNSTRETCHED_LEN)}
!
!--------------------------------------------------
! Build resistor cell
!--------------------------------------------------
!
EDIT CELL %SCRATCH
XSELECT OFF;
SELECT ALL; DELETE;
ADD CELL %RES$CELL AT (0, 0)
SELECT CELL * ALL; UNGROUP;
SELECT SIDE IN (0, -10000) (10000, 10000)
MOVE SIDE X %STRETCH.X
UNSELECT ALL
SELECT SIDE IN (-10000, 0) (10000, 10000)
MOVE SIDE Y %STRETCH.Y
UNSELECT ALL
EXIT
!
!--------------------------------------------------
! Add and ungroup resistor cell
!--------------------------------------------------
!
UNSELECT PUSH
ADD CELL %SCRATCH R%ROT_CODE
SELECT NEW
UNGROUP
UNSELECT POP
```

This command file allows the user to type in a resistance, a width and a rotation code to modify and then add the simple resistor configuration shown on the right.

Note the "{}" around the computations of STRETCH.Y and STRETCH.X. If these were not included, the expression string would be saved in the macros rather than the numerical result of the mathematical expression.

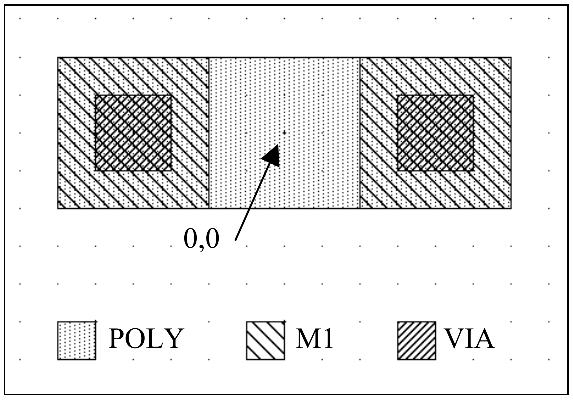


Figure 40:RES.CEL

This example could be improved by testing for a very short length since the resulting resistor may result in a short between the contacts.

SERIAL.CMD and _CHAR*n*.CMD — Add serial numbers to an array

This command file is used to add serial numbers to automatically label the cells in an array. These serial numbers are created as polygon components in the current cell.

As provided, this command file can be used to serialize only an array of cells with the cell name STEST.CEL. This sample cell file is supplied with the command file. It contains only a single box 7 user units high and 20 user units wide.

Before you can use this command file to serialize an array built from one of your own cells, you must edit the file to customize it with a few details about how you want the text created. See the table for the macro definitions you must add to the command file for a new *cell_name*.

Macro name	**Purpose**
MAG.*cell_name*	Font magnification, i.e. text height in user units
OFFSET.*cell_name*	Offset of lower left corner of text for cell relative to cell origin
LAYER.*cell_name*	Layer for serial number polygons
N.DIGITS.*cell_name*	Number of digits in serial number

Figure 41: Required information for additional cell

This command file will not work for rotated or mirrored arrays. The command file can recognize when the array is rotated or mirrored, and it will terminate with an error message. Arrays of cells with names longer than 24 characters are also not supported.

SERIAL.CMD

```
!
! The font used to label the array elements was taken
!  from Steve Stern's PGTEXT routines.
!
!********* Parameters for cell STEST **************
!
LOCAL #MAG.STEST =5.0       !Font magnification =
                            ! text height in microns
LOCAL #OFFSET.STEST =(1, 1) !Offset of lower left
                            ! corner of text in cell
                            ! relative to cell origin
LOCAL #LAYER.STEST = 1      !Text layer
LOCAL #N.DIGITS.STEST = 3   !Number of digits in serial
                            ! number
!
!* Use text editor to add parameters for other cells *!
!
!******** Select array to be serialized ************
!
LOCAL #N.SELECT.POP = %N.SELECT
IF(%N.SELECT!=0) UNSELECT PUSH
VIEW ON
WHILE(%N.SELECT != 1){
   UNSELECT ALL
   PROMPT = "Select array to be serialized"
   SELECT ARRAY * AT
}
ITEM LOCAL #SER
UNSEL ALL
VIEW OFF

IF(CMP("%SER.TRANS", "R0")!=0){
   RETURN; $SERIAL.CMD cannot handle rotated or &
           mirrored arrays
}
LOCAL #NAME=%SER.CELL.NAME
!
```

```
!******** Make sure parameters are defined *********
!
IF(MACRO_EXISTS(#MAG.%NAME)==0){
   RETURN; $ MAG.%name undefined. Edit SERIAL.CMD
}
IF(MACRO_EXISTS(#OFFSET.%NAME)==0){
   RETURN; $ OFFSET.%name undefined.  Edit SERIAL.CMD
}
IF(MACRO_EXISTS(#LAYER.%NAME)==0){
   RETURN; $ LAYER.%name undefined.  Edit SERIAL.CMD
}
IF(MACRO_EXISTS(#N.DIGITS.%NAME)==0){
   RETURN; $ N.DIGITS.%name undefined. Edit SERIAL.CMD
}
@_GET_INT; LOCAL #MIN=0; LOCAL #DEFAULT=1; &
          LOCAL #PROMPT="initial serial number"
LOCAL #N.SERIAL = %RET.VALUE
!
! Setup for UNED.CMD -- Part I
!
GLOBAL #ED.ID0 = %ID.MAX
GLOBAL #ED.FILE.NAME = %TMP^SERIAL.OUT
GLOBAL #UNED.FILE.NAME = "DO NOTHING"
!
!!******* Serialize cells in selected array *********
!
LOCAL #OFF0 = {%SER.POS.1 + %OFFSET.%NAME}
LOCAL #ROW = {%SER.N.ROWS - 1}
LOCAL #COL = 0
LOCAL #ROW.OFF = "UNDEFINED"
LOCAL #OFF = "UNDEFINED"
LOCAL #X.OFF = "UNDEFINED"
LOCAL #DIGIT = "UNDEFINED"
LOCAL #REM = "UNDEFINED"
LOCAL #Q = "UNDEFINED"
LOCAL #CHAR = "UNDEFINED"
LOCAL #MAX.SERIAL = &
   {%N.SERIAL + (%SER.N.COLS * %SER.N.ROWS) - 1}

LOG SCREEN=OFF LEVEL=NORMAL
```

```
WHILE(%ROW >= 0){
   #ROW.OFF = {%OFF0 + (0, %SER.ROW.STEP * %ROW)}
   #COL = 0
   WHILE(%COL < %SER.N.COLS){
      #OFF = {%ROW.OFF + (%SER.COL.STEP * %COL, 0)}
      #COL = {%COL + 1}
      #DIGIT = 1
      #REM = %N.SERIAL
$ N.SERIAL = %N.SERIAL / %MAX.SERIAL
      WHILE(%DIGIT <= %N.DIGITS.%NAME){
         #Q = {INT(%REM / 10)}
         #CHAR = {%REM - 10 * %Q}
         #REM = %Q
         ! FOR THIS FONT CHARACTER WIDTH=.9 * HEIGHT
         #X.OFF = {%MAG.%NAME * .9 *              &
                    (%N.DIGITS.%NAME - %DIGIT)}
         @_char%char; &
             LOCAL #LAYER=%LAYER.%NAME;           &
             LOCAL #FMAG=%MAG.%NAME;              &
             LOCAL #OFF={%OFF + (%X.OFF, 0)}
         #DIGIT = {%DIGIT + 1}
      }
      #N.SERIAL = {%N.SERIAL + 1}
   }
   #ROW = {%ROW - 1}
}
LOG SCREEN=ON LEVEL=NORMAL;
!
! SETUP FOR UNED.CMD -- PART II
!
GLOBAL #ED.ID1 = %ID.MAX
SELECT IDS AFTER %ED.ID0
UNSELECT IDS AFTER %ED.ID1
SHOW FILE=%ED.FILE.NAME
!
! Restore initial select state
!
IF(%N.SELECT.POP!=0) UNSELECT POP
ELSE UNSELECT ALL
```

The command file begins by defining the text specifications for arrays of cell STEST. Add your own cell text specifications below the "`*Add parameters for additional cells here*`" comment.

The next block of lines allows the user to select the array to be serialized. The VIEW mode is temporarily set to ON to insure that the geometry in the view window is current. The WHILE loop insures that a single array is selected.

The ITEM command is then used to create a variety of macros with information about the selected array. These macros will all have names that begin with the string "SER". One of these macros, SER.TRANS, is used to test that the array is not rotated or mirrored. The name of the cell stored in SER.CELL.NAME is copied to the new macro NAME.

The block of lines under the heading "`* Make sure parameters are defined *`" tests to make sure that the required text specification macros exist. Remember that these should be defined for additional cell names near the top of the command file. If any of these macros does not exist, the command file is terminated with an error message.

The "` Setup for UNED.CMD -- Part I`" section stores some of the information required to allow the results of this command file to be reversed with the UNED.CMD file (see page 318.) The rest of the UNED.CMD information is stored at the end of the command file.

The statements under the "`* Serialize cells in selected array *`" heading define the macros used to calculate the values needed to add the serial number components. The "`WHILE(%ROW >= 0)`" loop increments through each row of the array. The "`WHILE(%COL < %SER.N.COLS)`" loop performs the calculations for each cell in the row. The "`WHILE(%DIGIT <= %N.DIGITS.%NAME)`" loop makes one call to the CHAR*n*.CMD command file for each digit in the serial number for a particular cell.

@_CHAR%CHAR is the call to the _CHAR*n*.CMD command file that actually adds the geometry. The macro reference %CHAR determines which of the _CHAR*n*.CMD files is executed. If the macro CHAR is set to '0', then after macro substitution the reference will resolve to:

```
@_CHAR0
```

@_CHAR0 will execute the commands in _CHAR0.CMD.

CHAR0.CMD

This command file adds a polygon in the shape of the character '0' to the current cell. You must define three macros to specify the size, layer, and location for the polygon on the same line as the call to the command file. See an example in the SERIAL.CMD file description above.

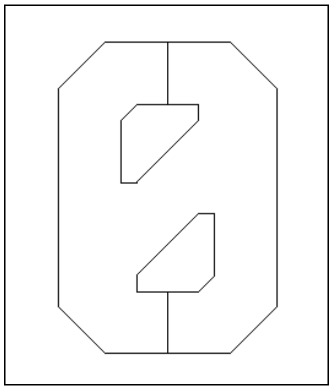


Figure 42: Polygon added by CHAR0.CMD

Macro	Purpose
LAYER	Layer
FMAG	Height
OFF	Location

Figure 43: Required macros for CHAR0

There are nine other command files (e.g. CHAR1.CMD, CHAR-2.CMD, etc.) that add characters for other digits. They are located in the same directory as the CHAR0.CMD file.

```
!
! This character outline was copied from Steve Stern's
!  PGTEXT routines
!
ADD  POLYGON  LAYER=%LAYER OFFSET=%OFF AT         &
     {%FMAG*(0.35, 0.2)}   {%FMAG*(0.25, 0.2)}   &
     {%FMAG*(0.25, 0.25)}  {%FMAG*(0.45, 0.45)}  &
     {%FMAG*(0.5, 0.45)}   {%FMAG*(0.5, 0.25)}   &
     {%FMAG*(0.45, 0.2)}   {%FMAG*(0.35, 0.2)}   &
     {%FMAG*(0.35, 0.0)}   {%FMAG*(0.55, 0.0)}   &
     {%FMAG*(0.7, 0.15)}   {%FMAG*(0.7, 0.85)}   &
     {%FMAG*(0.55, 1.0)}   {%FMAG*(0.35, 1.0)}   &
     {%FMAG*(0.35, 0.8)}   {%FMAG*(0.45, 0.8)}   &
     {%FMAG*(0.45, 0.75)}  {%FMAG*(0.25, 0.55)}  &
     {%FMAG*(0.2, 0.55)}   {%FMAG*(0.2, 0.75)}   &
     {%FMAG*(0.25, 0.8)}   {%FMAG*(0.35, 0.8)}   &
     {%FMAG*(0.35, 1.0)}   {%FMAG*(0.15, 1.0)}   &
     {%FMAG*(0.0, 0.85)}   {%FMAG*(0.0, 0.15)}   &
     {%FMAG*(0.15, 0.0)}   {%FMAG*(0.35, 0.0)}   &
     {%FMAG*(0.35, 0.2)}
```

ED.CMD and UNED.CMD Edit component properties (with undo capability)

The new ITEM command allows you to add a modified selected component without user interaction. See page 173.

These two command files allow the user to modify components by editing their definitions in ADD command format.

ED.CMD uses the SHOW SELECT command to build a file of ADD commands for the selected component(s), then the user edits this file with his or her favorite editor (NOTEPAD.EXE by default.) The original components are deleted, and the modified components are added to the cell.

UNED.CMD uses the global macros and files created by ED.CMD to reverse the entire process. The component id numbers stored by ED.CMD are used to delete the modified components and a backup file created by the SHOW command in ED.CMD is used to restore the original components.

Repeated calls to UNED.CMD toggle back and forth between the original components and the modified components.

```
!*****************

!      ED.CMD

!*****************

REMOVE #ED.CMD     ! PATCH FOR OBSOLETE .CMD FILES
REMOVE #UNED.CMD   ! PATCH FOR OBSOLETE .CMD FILES

GLOBAL #ED.FILE.NAME = %TMP^WORK.CMD
GLOBAL #UNED.FILE.NAME = %TMP^UNWORK.CMD
GLOBAL #ED.ID0=%ID.MAX
GLOBAL #ED.ID1=-1
GLOBAL #UNED.ENABLED=0
DEFAULT LOCAL #EDITOR="-NOTEPAD"
LOCAL  #FAILED=0;
```

```
SELECT PARTS ALL
SHOW FILE=%UNED.FILE.NAME
SELECT NEW
SELECT PARTS ALL
SHOW FILE=%ED.FILE.NAME
DOS %EDITOR %ED.FILE.NAME

LOCAL #ERROR.CMD="#FAILED=1"
@%ED.FILE.NAME
#ED.ID1=%ID.MAX
IF(%FAILED==0){
   #UNED.ENABLED=1
   #ED.ID1=%ID.MAX
   DELETE
   RETURN
}

WHILE(1){
   !
   ! UNDO BOTCHED JOB
   !
   UNSELECT PUSH
   SELECT IDS AFTER %ED.ID0 ALL
   XSELECT OFF
   DELETE
   SELECT POP

   ERROR  ! POST ERROR
   PAUSE
   !
   ! ASK IF USER WANTS TO CORRECT ERROR
   !
   @_GET_ANS;         &
      #CHOICES="YN"; &
      #PROMPT="DO YOU WANT TO RE-EDIT FILE [YN]?";
   IF(%RET.VALUE!=1) RETURN;
   !
   ! YES, HE WANTS TO
```

```
    ! REPOST ERROR SO IT WILL BE VISIBLE DURING EDIT
    !
    ERROR
    DOS "%EDITOR %ED.FILE.NAME"

    #FAILED=0
    LOCAL #ERROR.CMD="#FAILED=1"
    #ED.ID0=%ID.MAX

    @%ED.FILE.NAME
    IF(%FAILED==0){
       #UNED.ENABLED=1
       DELETE
       #ED.ID1=%ID.MAX
       RETURN;
    }
}
```

It is always a good idea to store temporary files in the directory stored in the system macro TMP so that they do not clutter up, or corrupt, files in your working directory.

The command file begins with a patch to prevent problems when a user has previously used an older version of these command files.

Next are several macro definitions. Most of these are global macros so that the related command file, UNED.CMD, can refer to them. The first two macro definitions use the system macro TMP to create temporary files in the TMP directory. The ED.ID0 macro uses the system macro ID.MAX to store the last component id number.

The ED.ID0 and ED.ID1 macros store component id numbers so that the related UNED.CMD command file can select only the components added by this command file. UNED.CMD deletes these components and adds copies of the original components.

The text editor name is stored in the EDITOR macro so that the user can override it. If the user already has an EDITOR macro defined with the name of their favorite editor, that editor will be used later on in the command file.

The "SELECT PARTS ALL" insures that any partially selected components will be fully selected. The "SHOW FILE=%UNED.FILE.NAME" command builds a file of ADD commands for the unmodified components. (This file can be used by UNED.CMD, to undo the results of this command file.)

Holding down the <Shift> key during a embedded SELECT command allows the user to select multiple components. Release <Shift> before selecting the last component, or click in empty area, to complete selection and return to the command file.

If no components are selected when the command file is executed, the SHOW command will generate an embedded SELECT NEAR command to select the desired component(s). If an embedded SELECT NEAR command was generated, the SHOW command automatically unselects the selected component(s) at the end of the command. The "SELECT NEW" after the SHOW command is added to reselect the component(s) in this case.

The "SHOW FILE=%ED.FILE.NAME" command builds the file of add commands that will be modified by this command file. The "DOS %EDITOR %ED.FILE.NAME" command launches the text editor to edit the file just created. This allows the user to modify the selected components by directly editing their ADD commands. Once the file is modified by the user, they must exit the editor to complete the command file.

The definition of the ERROR.CMD macro creates an error handler. From this point on in the command file, if a command fails (e.g. the user mistyped something in the file), then the command stored in the macro will be executed and the command file keeps going with the next command after the failed command. In this case an error flag in the FAILED macro is set.

The @%ED.FILE.NAME command execute the modified command file of ADD commands, thus adding the modified components. The id number of the last component added is then stored in the ED.ID1 macro.

If there were no errors in the modified command file, the next block of lines indicates that UNED.CMD can now be used successfully, deletes the original components (which are still selected), and terminates the command file.

If things went wrong, the command file begins a loop to try and fix the problem. First, the original components are pushed onto the select stack and unselected. Then any components added by the bad command file are selected by id numbers and deleted. The original components are then reselected.

Now the error message from the bad command in the modified command file is posted to the screen using the ERROR command with no arguments. The user is given the choice to edit the file and try again using the _GET_ANS.CMD command file.

```
!*****************
!     uned.cmd
!*****************
!
! The following two block-if's are a patch for
! obsolete .CMD files. Do not remove!
!
IF(MACRO_EXISTS(ED.CMD)){
   GLOBAL #ED.FILE.NAME=%ED.CMD
   REMOVE #ED.CMD
}
IF(MACRO_EXISTS(#UNED.CMD)){
   GLOBAL #UNED.FILE.NAME=%UNED.CMD
   REMOVE #UNED.CMD
}
!
! Check uned.enable flag
!
DEFAULT GLOBAL #UNED.ENABLED=0;
IF(%UNED.ENABLED==0) ERROR UNED uninitialized or disabled
!
! Save select state of any components currently selected
!
LOCAL #N.SELECT.POP = %N.SELECT
IF(%N.SELECT.POP!=0) UNSELECT PUSH
!
! Undoing ed.cmd involves the following steps:
!
! 1: Delete components added by %ed.file.name.
! 2: Add the components deleted by ed.cmd by executing
!   the commands in %uned.file.name. Update ed.id0 and
!   ed.id1 to record the range of id_numbers for the
!   added components.
! 3: Update the macros ed.file.name, and uned.file.name,
!   i.e. swap the roles of the files named by these macros.
!
IF(%ED.ID1==-1){ ! Implies ed.cmd crashed before
   ! updating ed.id1.
   ! Normally, components with id_numbers in the range:
   !
   !               %ED.ID0 < ID_NUMBER <= %ED.ID1
   !
   ! were added during the execution of ed.cmd.
   ! But we got here because ed.cmd crashed before
   ! updating ed.id1.  We must patch things up as best
   ! we can.  We will set ed.id1 equal to the current
   ! value of id.max.  This means we will delete all
   ! components added after the start of ed.cmd
   ! instead of just deleting components added during
   ! the execution of ed.cmd.
   !
   #ED.ID1=%ID.MAX
   !
   IF(%ED.ID1>%ED.ID0){
      SELECT IDS AFTER %ED.ID0
      !
      ! To prepare for future calls to uned.cmd, we
      ! must create a .CMD file that can replace the
      ! components we are about to delete. The string
      ! "do_nothing" is a flag for the next execution
      ! of uned.cmd.
      !
      IF(%N.SELECT>0){
         SHOW FILE="%ED.FILE.NAME"
         UNSELECT ALL
      }
      ELSE #ED.FILE.NAME=DO_NOTHING
   }
   ELSE #ED.FILE.NAME=DO_NOTHING
}
!
! Step 1: Delete components added by %ed.file.name.
!
SELECT IDS AFTER %ED.ID0
UNSELECT IDS AFTER %ED.ID1
XSELECT OFF
IF(CMP("%ED.FILE.NAME", "UNINITIALIZED")==0){
   #ED.FILE.NAME = %TMP^ED.FILE
   SHOW FILE="%ED.FILE.NAME"
}
DELETE
!
! Step 2: Replace components deleted by ed.cmd.
!   Update macros ed.id0 and ed.id1 for use in
!   future calls to uned.cmd.
! The macro uned.file.name normally holds a file name.
! It may also contain the words "do_nothing". Executing
! the command "@do_nothing" does nothing.
```

```
!
#ED.ID0=%ID.MAX
@"%UNED.FILE.NAME"; LOG SCREEN=OFF
#ED.ID1=%ID.MAX
!
! Step 3:  Prepare for future calls to uned.cmd by
!  swapping the roles of the files named in
!  ed.file.name and uned.file.name.
!
LOCAL #JUNK=%ED.FILE.NAME
#ED.FILE.NAME="%UNED.FILE.NAME"
#UNED.FILE.NAME=%JUNK
!
! Restore state of components selected at beginning
!
IF(%N.SELECT.POP!=0) UNSELECT POP
```

DEEPSHOW.CMD Enhance SHOW command for nested components

This command file uses a few extra commands that enhance the SHOW command to allow it to display component information even for deeply nested components.

```
LOCAL POS = NULL;
UNSELECT PUSH;

SELECT LAYERS 1:* NEAR
IF(%N.SELECT==0){
   #POS = %LAST.POS;
   PEDIT VIEW_ONLY=TRUE NEAR %POS;
   #POS = %LAST.POS;
   UNSELECT PUSH;
   SELECT NEAR %POS;
   SHOW FILE=*;
   SELECT POP;
   QUIT;
}
ELSE SHOW FILE=*;
UNSELECT ALL
SELECT POP
```

The N.SELECT system macro stores the number of currently selected components.

The command file begins with the definition of the POS macro that will store a position.

If components are already selected, the "UNSELECT PUSH" command will unselect them for later retrieval by the "SELECT POP" command at the end of the file.

"SELECT LAYERS 1:* NEAR" will wait for the user to digitize a position in the window. The digitized position defines the center of a near box to select all components on layers 1 through 255 that are within the near box. Since layer 0 is not in the layer list, the SELECT command is prevented from selecting nested cells.

If components are selected by the "SELECT LAYERS 1:* NEAR" command, the commands in the IF block will not be executed. The SHOW FILE=* command near the end of the command file is executed to display those components. In this case, the entire command has roughly the same result as a simple SHOW FILE=* command.

However, if no components in the current cell are within the near box, the statements in the IF block will be executed.

The system macro LAST.POS is used to retrieve the position just digitized by the user. The "PEDIT VIEW_ONLY=TRUE NEAR %POS" command will open the nested cell that directly contains the component near the position. If no cell is selected by the previous position, PEDIT will automatically issue a SELECT CELL NEAR command and the user will be given the chance to reposition the cursor until a cell is selected. The cell is opened and the new position is stored in the POS macro.

At this point the command file is editing a nested cell in the view-only mode. This means that even a cell in a protected library can be edited. The cell cannot be saved.

The current selection status of components in this new cell is stored by the "UNSELECT PUSH" command. This command also unselects all components. The "SELECT NEAR %POS" will select components within the near box defined by the stored position.

The remaining commands in the IF block SHOW the selected components and then quit the nested cell.

The final commands restore the original selection status of components.

BUSROUTE.CMD Replace single wide wire with routed bus of wires

The source files (written in 'C') for the executable program BUSROUTE-.EXE can be found in the Q:\ICWIN-\SAMPLES directory.

This command file uses the SHOW command to create a file with the ADD command for a single, wide wire that the user is prompted to create. A compiled program written in the C programming language then manipulates this file to transform the single wire into a bus of wires that follow the same shape. The command file then deletes the original wide wire and uses the file created by the program to add the new bus wires.

You will need to copy BUSROUTE.EXE from the Q:\ICWIN\SAMPLES directory to the Q:\ICWIN\TECH\SAMPLES directory to make it work.

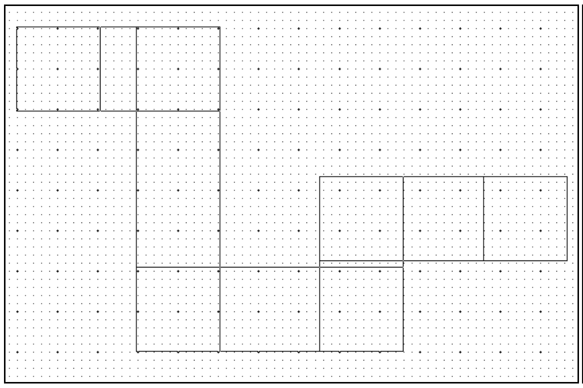


Figure 44: Digitizing wide wire for bus

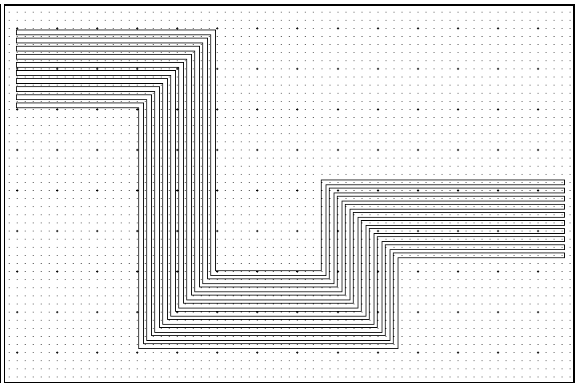


Figure 45: Routed bus

When you need to perform automatic component manipulation, an alternative method is the ITEM command (see page 173) or the DRC utility (available separately from IC Editors, Inc.). The DRC has many advanced features for automatically manipulating components.

```
!************BUSROUTE.CMD ***************************
!
DEFAULT GLOBAL #BUS.ADD.MESSAGE.ON = 1;
DEFAULT LOCAL #SCRATCH.LAYER = 251;

!****************************************************
!
! Get bus parameters - use old parameters as defaults
!
!****************************************************

! ************** Get bus layer: **************
```

The user can select a layer from a menu of valid choices. See page 147.

```
@_GET_LAY; &
   LOCAL #PROMPT="bus layer";              &
   IF(MACRO_EXISTS(#BUS.LAST.LAYER))       &
      LOCAL #DEFAULT=%BUS.LAST.LAYER;
LOCAL #LAYER = %RET.VALUE
GLOBAL #BUS.LAST.LAYER = %RET.VALUE

! *********** Get number of wires ************

@_GET_INT; &
   LOCAL #PROMPT="number of wires";             &
   LOCAL #MIN=1;                                &
   IF(MACRO_EXISTS(#BUS.LAST.N.WIRES))          &
      LOCAL #DEFAULT=%BUS.LAST.N.WIRES;
LOCAL #N.WIRES = %RET.VALUE
GLOBAL #BUS.LAST.N.WIRES = %RET.VALUE

! ************** Get wire width **************

@_GET_REAL;                                     &
   LOCAL #PROMPT="wire width";                  &
   IF(MACRO_EXISTS(#BUS.WIDTH.%LAYER))          &
      LOCAL #DEFAULT=%BUS.WIDTH.%LAYER;    &
   LOCAL #MIN=%RES.STEP;
LOCAL #WIDTH = %RET.VALUE
GLOBAL #BUS.WIDTH.%LAYER = %RET.VALUE
```

```
! ************* Get wire spacing *************

@_GET_REAL;                                         &
   LOCAL #PROMPT="wire spacing";                    &
   IF(MACRO_EXISTS(#BUS.SPACE.%LAYER))              &
     LOCAL #DEFAULT=%BUS.SPACE.%LAYER;
   LOCAL #MIN=%RES.STEP;
LOCAL #SPACE = %RET.VALUE
GLOBAL #BUS.SPACE.%LAYER = %RET.VALUE

!*****************************************************
!
!        Remove old BUSROUTE.OUT
!
!*****************************************************

! If an old BUSROUTE.OUT exists and BUSROUT.EXE
! crashes then BUSROUTE.CMD will mistake the existing
! file for new output generated by the upcoming call
! to BUSROUT.EXE.  We will try to avoid that
! possibility by deleting any existing BUSROUTE.OUT.
! This is not fool proof, since the system  will deny
! a request to delete a "protected" file.

DOS "^DEL %TMP^BUSROUTE.OUT > NUL"

!*****************************************************
!
!                  Do it
!
!*****************************************************

LOCAL #PITCH = {%SPACE + %WIDTH}
LOCAL #TOTAL.WIDTH = {%N.WIRES * %PITCH + %SPACE}
LOCAL #N.SELECT.PUSH = %N.SELECT
IF(%N.SELECT.PUSH) UNSELECT PUSH
IF(%BUS.ADD.MESSAGE.ON){
   #BUS.ADD.MESSAGE.ON = 0;
   $Press <Enter> to route bus outline (This message &
           will not appear next time)
   PAUSE 0
}
LOCAL #SAVE.SNAP.ANGLE = %SNAP.ANGLE
SNAP ANGLE = 90
VIEW ON; ! Update screen if view changes while adding wire
ADD WIRE LAYER=%SCRATCH.LAYER         &
         WIDTH=%TOTAL.WIDTH TYPE=0
VIEW OFF;
SNAP ANGLE = %SAVE.SNAP.ANGLE

!**Setup for UNED.CMD once BUSROUTE.CMD
!  cannot be cancelled

GLOBAL #UNED.FILE.NAME = %TMP^BUSROUTE.SHO
GLOBAL #ED.FILE.NAME = %TMP^BUSROUTE.OUT
GLOBAL #ED.ID1 = -1   ! flag indicating nothing done

SELECT NEW
SHOW PROG=1 FILE=%UNED.FILE.NAME
DOS "^%EXEC.DIR^BUSROUTE.EXE %TMP &
      %SCRATCH.LAYER %LAYER
      %USE.WIRE.TYPE %WIDTH %SPACE %N.WIRES"

GLOBAL #ED.ID0 = %ID.MAX
@%TMP^BUSROUTE.OUT; VIEW OFF
#ED.ID1 = %ID.MAX;

DELETE
IF(%N.SELECT.PUSH) SELECT POP;
```

The macro definition for BUS.ADD.MESSAGE.ON determines later whether this is the first time the command file is executed. It will determine whether or not a prompt message is printed. The SCRATCH.LAYER macro defines the layer used for scratch work. You will need to change this layer if you use layer 251 for real work.

The first several blocks of code prompt the user for the layer of the bus, the number of wires in the bus, the width of each wire, and the spacing between the wires. Each block works in the same manner. The appropriate user prompt command file _GET_LAY.CMD, _GET_INT.CMD, or _GET_REAL.CMD is used to prompt the user for the value. If a default has been saved from a previous execution of BUSROUTE.CMD, then that default is passed into the nested command file. The value entered by the user is stored in both a local macro and a global macro that stores the default for the next execution.

Next a DOS command is used to delete an old BUSROUTE.OUT file if one exists from a previous execution of BUSROUTE.CMD. This is the file created by the BUSROUTE.EXE program called below.

The next block of lines, below the "Do it" comment, calculate the total width of the bus. The currently selected components are pushed for later retrieval.

If BUSROUTE.CMD has been executed before, the commands in the next IF block will not be executed. Otherwise, a prompt message is displayed on the screen to explain to the user that he must route the bus outline.

The current snap angle is stored in the SAVE.SNAP.ANGLE local macro. Then the snap angle temporarily set to 90° while the user routes the bus outline on the layer stored in the SCRATCH.LAYER macro. The original snap angle is then restored.

Next, the command file sets up some macros that will enable the UNED.CMD command file to undo the results of this command file. (See page312.)

The routed bus outline wire is selected by the SELECT NEW command. Then the description of this component is stored in the file name stored in the UNED.FILE.NAME macro by the SHOW command.

The BUSROUTE.EXE program should be stored in the same directory as the command file so the system macro %EXEC.DIR correctly indicates the path to the program file.

This file is passed as an argument to the DOS program BUSROUTE.EXE. Other parameters including the wire width, spacing, etc. are also passed to this program. The program should execute so quickly that the user does not even realize that control has passed from the editor to a DOS procedure. The program will place ADD commands for each bus wire in the file BUSROUTE.OUT in the directory stored in the TMP system macro.

Before the ADD commands in the BUSROUTE.OUT file are executed, the last component id used is stored in the ED.ID0 macro. This is to enable the UNED.CMD command file to undo the results of the BUSROUTE.OUT command file. After BUSROUTE.OUT is executed, the last id used is stored in ED.ID1 for UNED.CMD.

The final two lines delete the routed outline wire on the scratch layer and restore the selection status of components before the command file began.

BUSROUTE.C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include "showlib.h"

extern int errno;

FILE *fp_sho, *fp_out;
char *sho_file_name, *output_file_name;

typedef struct{ double x, y; } FPOS;
typedef struct{ int dx, dy;} SHIFT_DIR;
SHIFT_DIR shift_table[4][4] =
     {{{ 0, 0}, {-1, 1}, { 0, 0}, { 1, 1}},
      {{-1, 1}, { 0, 0}, {-1,-1}, { 0, 0}},
      {{ 0, 0}, {-1,-1}, { 0, 0}, { 1,-1}},
      {{ 1, 1}, { 0, 0}, { 1,-1}, { 0, 0}}};
```

```
SHIFT_DIR end_shift_table[4] =
      {{ 0, 1}, {-1, 0}, { 0,-1}, { 1, 0}};

/******** Command line arguments ************

File names:

   argv[1]  string  tmp directory name (ends in \)

   Example: argv[1]=c:\iced\tmp\ =>
            .SHO file is      c:\iced\tmp\busroute.sho
            Output file is    c:\iced\tmp\busroute.out

Outline wire parameters:

   argv[2]  int  layer NUMBER for "outline" wire

Bus wire parameters:

   argv[3]  int     layer number
   argv[4]  int     wire type
   argv[5]  double  wire width
   argv[6]  double  wire spacing
   argv[7]  int     number of wires in bus

********* End Command line arguments *******/

/*#***************************************/
/*                                       */
/*void main(int argc, char *argv[])      */
/*                                       */
/*******-*******-************************/

void main(int argc, char *argv[])
{
   char *ptr0, *ptr;
   int len, wire_layer, wire_type, n_wires;
   int spine_layer, i, j, dir_code;
   double wire_pitch, wire_width, adjust;
   double wire_spacing, bus_width;
   double offset, x, y, dx, dy;

   /* The following arrays are static to avoid stack
    *overflows when using a 16-bit compiler */

   static FPOS pos[200]; /*Max 200 points in wire  */
   static SHIFT_DIR shift_dir[200];
   static char dir[199]; /*Max 199 segments in wire*/

   int n_pos = 0;

   /* Check argument count */

   if(argc != 8){
      crash("Command line error -- Bad arg count\n"
            "Expected 7 args -- found %d", argc-1);
      }
   len = strlen(argv[1]);

   /* Form file names from argv[1] */

   sho_file_name = malloc(len + 20);
   output_file_name = malloc(len + 20);
   if(sho_file_name==NULL || output_file_name==NULL){
      crash("Insufficient memory");
      }
   strcpy(sho_file_name, argv[1]);
   strcpy(sho_file_name+len, "BUSROUTE.SHO");

   strcpy(output_file_name, argv[1]);
   strcpy(output_file_name+len, "BUSROUTE.OUT");

   /* Process argv[]'s 2 thru 7 */

   if(sscanf(argv[2], "%d", &spine_layer)==0){
      crash("Arg 2 error-- Outline layer number\n"
            "Expected integer:found \"%s\"", argv[2]);
      }
```

```
if(spine_layer < 1 || spine_layer > 255){
   crash("Arg 2 error -- Outline layer number\n"
     "Value \"%s\" out of range [1:255]", argv[2]);
   }

if(sscanf(argv[3], "%d", &wire_layer)==0){
   crash("Argument 3 error -- Bus layer number\n"
      "Expected integer -- found \"%s\"", argv[3]);
   }
if(wire_layer < 1 || wire_layer > 255){
   crash("Arg 3 error-- Bus layer number\n"
     "Value \"%s\" out of range [1:255]", argv[3]);
   }

if(sscanf(argv[4], "%d", &wire_type)==0){
   crash("Arg 4 error -- Bus wire type\n"
      "Expected integer -- found \"%s\"", argv[4]);
   }
if(wire_type!=0 && wire_type!=2){
   crash("Arg 4 error -- Bus wire type\n"
 "Invalid value \"%s\": expected 0 or 2", argv[4]);
   }

if(sscanf(argv[5], "%le", &wire_width)==0){
   crash("Arg 5 error -- Bus wire width\n"
    "Expected real number:found \"%s\"", argv[5]);
   }
if(wire_width <= 0){
   crash("Arg 5 error-- Bus wire width\n"
     "Invalid value \"%s\":must be > 0", argv[5]);
   }

if(sscanf(argv[6], "%le", &wire_spacing)==0){
   crash("Arg 6 error -- Bus wire spacing\n"
    "Expected real number:found \"%s\"", argv[6]);
   }
if(wire_spacing<=0){
   crash("Arg 6 error -- Bus wire spacing\n"
    "Invalid value \"%s\": must be > 0", argv[6]);
   }
```

```
if(sscanf(argv[7], "%d", &n_wires)==0){
   crash("Arg 7 error -- Number of wires in bus\n"
       "Expected integer:found \"%s\"", argv[7]);
   }
if(n_wires < 1){
   crash("Arg 7 error-- Number of wires in bus\n"
    "Invalid value \"%s\": must be > 0", argv[7]);
   }

/* Open files */

/* errno is set (and used) by the system.
 * Sometimes a non-zero value is left over
 * from an internal system call */
errno = 0;
fp_sho = fopen(sho_file_name, "r");
if(fp_sho==NULL){
   sys_crash("Could not open input file \"%s\"",
         sho_file_name);
   }

fp_out = fopen(output_file_name, "w+t");
if(fp_out==NULL){
   sys_crash("Could not open output file \"%s\"",
         output_file_name);
   }

/* Read outline wire spine coordinates */

/* We will ONLY try to check for obvious errors
 * that could be caused by using the wrong SHOW
 * command in BUSROUTE.CMD.  We will NOT try
 * to check everything. */

ptr0 = find_line("ADD WIRE");
if(ptr0==NULL){
   crash("No ADD WIRE cmd in input file \"%s\"",
        sho_file_name);
   }
   ptr = skip_passed(ptr0, "ADD WIRE");
   ptr = skip_passed(ptr, "WIDTH=");
   bus_width = atof(ptr);
   ptr = skip_passed(ptr, "AT ");

   while((ptr = skip_passed(ptr, "("))!=NULL){
      if(sscanf(ptr, " %lg, %lg)", &x, &y)!=2){
         crash("Error reading ADD WIRE command\n"
               "Expected (x, y)... -- found \"%s\"",
                ptr-1);
         }
      pos[n_pos].x = x;
      pos[n_pos].y = y;
      n_pos++;
      }

/************ ALGORITHM *******************

Definitions:
   The SPINE wire is the wire used to outline the bus.
   A BUS wire is one of the wires in the bus.

We now have the coordinates of the spine the wire and
we want to use them to compute the coordinates of the
bus wires.  The coordinates of a bus wire are computed
by adding a shift_vector to the coordinates of the
spine wire.  The shift_vector for the i-th vertex of
the j-th wire is:

   shift_vector.x =
     (j - (n_wires-1)/2.) * wire_pitch * shift_dir.dx
   shift_vector.y =
     (j - (n_wires-1)/2.) * wire_pitch * shift_dir.dy

The direction of the shift vector for a given vertex
depends on the directions of the spine wire segments
that meet at the vertex.
```

```
The algorithm has the following steps:

1) Compute direction codes (dir[]) for each segment of
the spine wire.
2) Adjust spine wire endpoints to allow for extended
ends of bus and spine wires.
3) Use these codes and the preprepared look-up tables
to compute shift_dir[]for each spine wire vertex.
4) Compute (and output) the vertexes for the bus
wires.

1) Compute dir[i] for each segment:

   dir[i] = is based on the direction of a vector from
pos[i] to pos[i+1].
   dir[i]=0 => vector points in the + x direction
   dir[i]=1 => vector points in the + y direction
   dir[i]=2 => vector points in the - x direction
   dir[i]=3 => vector points in the - y direction

   The n_pos vertexes are numbered from 0 to n_pos-1.
   There are only n_pos-1 wire segments
     => n_pos-1 dir[]'s.
   We will number them from 0 to n_pos-2.
***************************************************/

   for(i=0; i<=n_pos-2; i++){
      dx = pos[i+1].x - pos[i].x;
      dy = pos[i+1].y - pos[i].y;
      dir_code = -1;
      if(dy==0){
         if(dx > 0) dir_code = 0;
         else if(dx < 0) dir_code = 2;
         }
      else if(dx==0){
         if(dy > 0) dir_code = 1;
         else if(dy < 0) dir_code = 3;
         }
```

```
        if(dir_code==-1){
            crash("Bad wire segment connecting points
               (%g,%g) and (%g,%g).\n"
                   "Points cannot be connected by a
                    horizontal or vertical line.",
            pos[i].x, pos[i].y, pos[i+1].x, pos[i+1].y);
            }
        dir[i] = dir_code;
        }

/* 2) Adjust spine wire endpoints to allow for
        extended ends of bus and spine wires: */

    adjust = 0.5 * (wire_type==2
              ? (bus_width - wire_width) : bus_width);

    dir_code = dir[0];
    if(dir_code==0)      pos[0].x -= adjust;
    else if(dir_code==1) pos[0].y -= adjust;
    else if(dir_code==2) pos[0].x += adjust;
    else if(dir_code==3) pos[0].y += adjust;

    dir_code = dir[n_pos-2];
    if(dir_code==0)      pos[n_pos-1].x += adjust;
    else if(dir_code==1) pos[n_pos-1].y += adjust;
    else if(dir_code==2) pos[n_pos-1].x -= adjust;
    else if(dir_code==3) pos[n_pos-1].y -= adjust;

/*3)Compute shift_dir[] for each spine wire vertex:*/

    shift_dir[0] = end_shift_table[dir[0]];
    shift_dir[n_pos-1] = end_shift_table[dir[n_pos-2]];
    for(i=1; i<=n_pos-2; i++){
        shift_dir[i] = shift_table[dir[i-1]][dir[i]];
        }
```

```
/*4) Compute and output vertexes for the bus wires:*/

   wire_pitch = wire_spacing + wire_width;
   offset = 0.5 * (n_wires-1) * wire_pitch;
   for(j=0; j<n_wires; j++){
      fprintf(fp_out,
        "ADD WIRE LAYER=%d WIDTH=%g TYPE=%d AT",
         wire_layer, wire_width, wire_type);
      for(i=0; i<n_pos; i++){
         x = pos[i].x;
         if(shift_dir[i].dx==1) x += offset;
         else if(shift_dir[i].dx==-1) x -= offset;
         y = pos[i].y;
         if(shift_dir[i].dy==1) y += offset;
         else if(shift_dir[i].dy==-1) y -= offset;
         fprintf(fp_out, " (%g, %g)", x, y);
         }
      offset -= wire_pitch;
      fprintf(fp_out, "\n", x, y);
      }
   exit(0);
   }
```

SHOWLIB.C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "showlib.h"

/* The following routines form the first draft of a
 simple library that is useful in processing .SHO
 files.  Notice that they make use of the following
 external variables: */

extern FILE *fp_sho, *fp_out;
extern char *sho_file_name, *output_file_name;

/* which must be declared and initialized in your
```

```
  program.  The external variable: */

extern int errno;

/* is declared and set by the operating system.

Required PROTOTYPES are declared in sholib.h */

/*#*********************************************/
/*                                             */
/*char *skip_passed(char *ptr0, char *string)  */
/*                                             */
/*****-*****************-***********************/

/* Purpose:

   Skip_passed(ptr0, string) searches a show line
starting at ptr0 until it finds a match for string.
If a match is found it returns a pointer to the
character following the matching section of the show
line.   If no match is found it returns NULL.
Skip_passed() uses a case dependent compare.  A blank
in string matches 1 or more blanks in the show line.
Thus, if string ends in a blank, the returned pointer
will not point to a blank character. */

char *skip_passed(char *ptr0, char *string)
{
   char *ptr = ptr0;
   char *str = string;

   while(*ptr0!='\0'){
      ptr = ptr0;  str = string;
      while(*str==*ptr){
         if(*str==' '){
            while(*ptr==' ') ptr++;
            while(*str==' ') str++;
            }
         else if(*str=='\0') return ptr; /* MATCH */
         else{
            ptr++;
            str++;
            }
         }
      if(*str=='\0') return ptr; /* MATCH */
      ptr0++;
      }
   return NULL;               /* NO MATCH */
   }
/*#*************************************************/
/*                                               */
/* char *find_line(char *string)                 */
/*                                               */
/******-****************-**********************/

/* Purpose:

   Reads stream *fp_sho until it finds a line that
starts with "string".  It returns a pointer to to the
first non-blank character of the line if a match is
found.  It returns NULL for no match.  This module
uses a case dependent compare.  A blank in string
matches 1 or more blanks in the show file line.

   Side effects:

   Find_line() uses read_show_line() to read *fp_sho.
Read_show_line()reads data into a static buffer.  The
pointer it returns points to data in this buffer.  Any
call to read_show_line() overwrites the data that was
in the buffer as a result of previous calls.

   The file position for *fp_sho is advanced to the
character following the end of the matching line.  If
no match is found the file pos will point to the end
of file. */
```

```
char *find_line(char *string)
{
   char *ptr, *ptr0, *str;

   while(1){
      ptr0 = read_show_line();
      if(ptr0==NULL)
           return(NULL); /* RETURN END OF FILE FLAG */
      str = string;
      ptr = ptr0;
      while(*str==*ptr){
         if(*str==' '){
            while(*ptr==' ') ptr++;
            while(*str==' ') str++;
            }
         else if(*str=='\0'){
            return ptr0; /* RET PTR TO MATCHING LINE*/
            }
         else{
            ptr++;
            str++;
            }
         }
         if(*str=='\0')
           return ptr0; /* RET PTR TO MATCHING LINE*/
      }
   }
/*#**********************************************/
/*                                              */
/* char *read_show_line(void)                   */
/*                                              */
/**********-****************-*******************/

/* Purpose:

   Read_show_line() reads the next "complete" line
from stream *fp_sho.   This routine reads whole lines,
i.e. if a line ends in an '&' (continuation mark) the
'&' is erased and the next line is read and
appended to it.  (There are likely to be five or more
blanks separating     the two parts   of the merged
line.)  Any '\t', '\n', or '\r' characters that appear
in the line are replaced by blanks.  Trailing blanks
are removed from the end of the line.

   Read_show_line() returns a pointer to to the first
non-blank character of the line.  It returns NULL if
it reads the end of file before reading a line.

   Side effects:

   Read_show_line() reads data into a static buffer.
The pointer it returns points to data in this buffer.
Any call to read_show_line() overwrites the data that
was in the buffer as a result of previous calls.

   The file position for *fp_sho is advanced to the
character following the end of the line.  If an end of
file (eof) is read, it repositions the file pointer to
the eof so that the next file read will also return
an eof.

   */
char *read_show_line(void)
{
   static char buffer[MAX_SHOW_LINE_LEN+1];
   int len = 0;
   int chr = 0;
   char *ptr;
   int n;

   while(1){
      n = read_show_line0(buffer+len,
            MAX_SHOW_LINE_LEN-len+1);
      if(n==-1){              /* End of file flag */
         if(chr=='&'){
            crash("Unexpected end of file  \"%s\"\n"
               "The last line ends with an '&' ",
               sho_file_name);
            }
         fseek(fp_sho, 0, SEEK_END);
         return(NULL); /* RETURN END OF FILE */
         }
      len += n;

      /* Check final non-blank character for '&'  */

      if(len==0) continue; /* skip empty line */
      ptr = buffer + len - 1;
      chr = *ptr;

      if(chr!='&'){ /* Skip leading white space */
         ptr = buffer;
         while(*ptr==' ') ptr++;
         if(*ptr=='\0') continue; /*skip empty line */
         return(ptr); /* RETURN COMPLETE LINE **/
         }
      /* chr=='&' */

      len--;
      };
   }
/*#*******************************************/
/*                                           */
/*read_show_line0(char *buffer, int buffer_len) */
/*                                           */
/*************-***************-************/

/* Purpose: Read_show_line0() is used by
read_show_line0() */

int read_show_line0(char *buffer, int buffer_len)
{
   char *ptr;

   /* Read line */

   errno = 0;
   ptr = fgets(buffer, buffer_len, fp_sho);
   if(ptr==NULL){
      if(feof(fp_sho)){
         return(-1);
         }
      sys_crash("Error reading \"%s\"",
           sho_file_name);
      }

  /*Replace tab,return,and newline chars with blanks*/

   for(ptr=buffer; *ptr!='\0'; ptr++){
      if(*ptr=='\t' || *ptr=='\n' || *ptr=='\r')
            *ptr = ' ';
      }

   /* Remove trailing blanks */

   while(ptr>buffer && *(ptr-1)==' ') ptr--;
   *ptr = '\0';
   return(ptr-buffer);
   }
/*#**************************************************/
/*                                                  */
/*   crash(char *fmt, ...)                          */
/*                                                  */
/*********-****************-*************************/

/* Purpose: Print message, close and remove output
file, exit */

void crash(char *fmt, ...)
{
   va_list ap;

   va_start(ap, fmt);
   vprintf(fmt, ap);
   va_end(ap);
   printf("\n");

   if(fp_out!=NULL){
      fclose(fp_out);
      remove(output_file_name);
      }
   exit(10);
   }
/*#*************************************************/
/*                                                 */
/* sys_crash(char *fmt, ...)                       */
/*                                                 */
/**************-****************-**************/

/* Purpose: Print message, print system error message,
close and remove output file, exit */

void sys_crash(char *fmt, ...)
{
   va_list ap;

   va_start(ap, fmt);
   vprintf(fmt, ap);
   va_end(ap);
   printf("\n");

   if(errno!=0) printf("System reports: %s",
           strerror(errno));

   if(fp_out!=NULL){
      fclose(fp_out);
      remove(output_file_name);
      }
   exit(10);
   }
```