Contents

Introduction ......................................................... 1
  How This Guide Has Changed .................................. 2
  Using AutoCAD Documentation .................................. 2
  Customizing AutoCAD ............................................. 3
  Typographical Conventions ....................................... 5
  AutoCAD 2000 Changes and Enhancements ....................... 6

Chapter 1 Customization Basics ................................ 9
  Basic Requirements .............................................. 10
    AutoCAD Environment ......................................... 10
    Library Search Path ......................................... 10
    Sample Directory Structure .................................. 11
    Multiple Configurations ..................................... 11
    Multiple Drawing Directories ................................. 12
    Command Search Procedure .................................. 13
    Customizable Support Files .................................. 14
  acad.pgp—Program Parameters File ............................. 16
  Defining External Commands ................................... 16
    Using Windows System Commands ............................ 18
    Using Custom-Defined Commands ............................. 19
    Command Aliases .............................................. 19
Custom Online Documentation ........................................... 20
  Windows Help Files ............................................. 21
  Windows Help System Components ......................... 21
  Using Windows Help Files with AutoCAD ............... 23
  HTML Files ....................................................... 25
  Using HTML Files with AutoCAD ......................... 25
  AutoCAD Help Files .......................................... 27

Chapter 2  Linetypes and Hatch Patterns ......................... 29
  Linetype Definition Files .................................... 30
    Creating and Modifying Linetype Definitions .......... 30
    Defining Linetypes ........................................ 30
    Simple Linetypes .......................................... 31
    Complex Linetypes ........................................ 32
      Shapes in Complex Linetypes ............................... 33
      Text in Complex Linetypes ................................ 34
  Creating Hatch Patterns .................................... 36
    Using the AutoCAD Pattern File—acad.pat .............. 36
    Constructing Hatch Patterns ............................... 37
    Patterns with Dashed Lines ................................ 37

Chapter 3  Shapes, Fonts, and PostScript Support ................. 41
  Using Shape Files ............................................. 42
  Compiling Shape/Font Files ................................. 42
  Compiling PostScript Fonts .................................. 43
  Creating Shape Definition Files ......................... 43
    Shape Descriptions .......................................... 44
      Vector Length and Direction Code ....................... 45
      Special Codes ............................................. 46
    Text Font Descriptions .................................... 52
      Big Font Descriptions ..................................... 54
        Defining a Big Font ....................................... 54
        Defining an Extended Big Font File ................... 54
        Using a Big Font ......................................... 59
        Alternate Uses for a Big Font ......................... 60
    Unicode Font Descriptions ................................ 61
      Format for Unicode Shape Definition Files ............ 62
    Advanced Shape Definition Technique .................. 63
Pull-Down and Shortcut Menus .......................... 96
  Creating Pull-Down Menus .......................... 97
  Creating Shortcut Menus .......................... 98
  Summary of Pull-Down and Shortcut Menu Label Syntax ........ 99
  Pull-Down Menu Bar Titles ......................... 100
  Cascading Submenus ................................ 101
  Separation of Menu Item Labels ................... 101
  Controlling the Display of Menu Item Labels ........ 102
    Disabling Labels .................................. 102
    Marking Labels .................................. 102
    Simultaneous Disabling and Marking .............. 103
    Referencing Pull-Down and Shortcut Menus ...... 103
    AutoLISP Access to Label Status ................ 105
  Pull-Down Menu Swapping and Inserting ............ 106
    Swapping Pull-Down Menus ......................... 106
    Inserting and Removing Pull-Down Menus ....... 106

Toolbars .............................................. 107
  Creating Toolbars .................................. 107
    General Toolbar Definition ........................ 108
    Toolbar Button Definition ........................ 108
    Toolbar Flyout Definition ......................... 109
    Toolbar Control Definition ....................... 110
    User-Defined Bitmaps ................................ 111
  Referencing Toolbars ................................ 111

Image Tile Menus ...................................... 111
  Image Tile Menu Items .............................. 112
    Image Tile Item Labels ............................ 112
    Image Tile Menu Macros ........................... 113
  Displaying Image Tile Menus ....................... 113
  Sample Image Tile Menus ........................... 114
  Preparation of Slides for Image Tile Menus ........ 115

Screen Menus .......................................... 116
  Creating Screen Menus ................................ 116
  Screen Submenus .................................... 117
  Item Labels ........................................ 120
  The MNUCTL System Variable ....................... 121

Tablet Menus .......................................... 121
  Creating Tablet Menus ................................ 121

Menu-Specific Help .................................... 122
  Accelerator Keys .................................... 123
## Chapter 5  Using DIESEL—String Expression Language          127

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Line Configuration—MODEMACRO</td>
<td>128</td>
</tr>
<tr>
<td>Understanding the MODEMACRO Variable</td>
<td>128</td>
</tr>
<tr>
<td>MODEMACRO Definitions</td>
<td>129</td>
</tr>
<tr>
<td>MODEMACRO Definitions with AutoLISP</td>
<td>130</td>
</tr>
<tr>
<td>DIESEL Expressions in Menus</td>
<td>132</td>
</tr>
<tr>
<td>DIESEL Expressions in AutoLISP</td>
<td>135</td>
</tr>
<tr>
<td>Catalog of DIESEL String Functions</td>
<td>135</td>
</tr>
<tr>
<td>+ (addition)</td>
<td>135</td>
</tr>
<tr>
<td>– (subtraction)</td>
<td>136</td>
</tr>
<tr>
<td>* (multiplication)</td>
<td>136</td>
</tr>
<tr>
<td>/ (division)</td>
<td>136</td>
</tr>
<tr>
<td>= (equal to)</td>
<td>136</td>
</tr>
<tr>
<td>&lt; (less than)</td>
<td>136</td>
</tr>
<tr>
<td>&gt; (greater than)</td>
<td>136</td>
</tr>
<tr>
<td>!= (not equal to)</td>
<td>137</td>
</tr>
<tr>
<td>&lt;= (less than or equal to)</td>
<td>137</td>
</tr>
<tr>
<td>&gt;= (greater than or equal to)</td>
<td>137</td>
</tr>
<tr>
<td>and</td>
<td>137</td>
</tr>
<tr>
<td>angtos</td>
<td>137</td>
</tr>
<tr>
<td>edtime</td>
<td>138</td>
</tr>
<tr>
<td>eq</td>
<td>139</td>
</tr>
<tr>
<td>eval</td>
<td>139</td>
</tr>
<tr>
<td>fix</td>
<td>139</td>
</tr>
<tr>
<td>getenv</td>
<td>140</td>
</tr>
<tr>
<td>getvar</td>
<td>140</td>
</tr>
<tr>
<td>if</td>
<td>140</td>
</tr>
<tr>
<td>index</td>
<td>140</td>
</tr>
<tr>
<td>linelen</td>
<td>140</td>
</tr>
<tr>
<td>nth</td>
<td>141</td>
</tr>
<tr>
<td>or</td>
<td>141</td>
</tr>
<tr>
<td>rtos</td>
<td>141</td>
</tr>
<tr>
<td>strlen</td>
<td>141</td>
</tr>
<tr>
<td>substr</td>
<td>141</td>
</tr>
<tr>
<td>upper</td>
<td>142</td>
</tr>
<tr>
<td>xor</td>
<td>142</td>
</tr>
<tr>
<td>Error Messages</td>
<td>142</td>
</tr>
</tbody>
</table>
Chapter 6  Programming Interfaces ........................................... 143
Command Scripts ...................................................................... 144
   Invoking a Script When Loading AutoCAD ....................... 144
   Creating Slide Shows .......................................................... 146
ActiveX Automation ................................................................. 146
   Using Automation Applications .......................................... 147
      Starting Your Application from the Command Line .......... 147
      Starting Your Application from a Menu ...................... 148
AutoCAD VBA ..................................................................... 149
   Developing with AutoCAD VBA ......................................... 150
   Using AutoCAD VBA Applications ..................................... 152
      Using the Command Line to Run a VBA Macro ............. 152
   Automatic Loading and Execution of VBA Projects .......... 152
      acadvba.arx—Automatic Loading of VBA ..................... 153
      acad.dvb—Automatic Loading of a VBA Project .............. 153
AutoLISP and Visual LISP ....................................................... 153
   Using AutoLISP Applications .......................................... 155
   Automatic Loading and Execution of AutoLISP Routines .... 156
      acad.lsp—Automatic Loading of AutoLISP .................... 156
      acaddr.doc.lsp—Automatic Loading of AutoLISP .......... 157
      acad.mnl—Automatic Loading of AutoLISP ................. 157
         Menu Functions ......................................................... 158
   Coding Tips for AutoLISP Startup Files ......................... 159
   Command Autoloader ......................................................... 159
      S::STARTUP Function—Post-Initialization Execution .. 160
ObjectARX ....................................................................... 161
   Using ObjectARX Applications ....................................... 161
   Automatic Loading of ObjectARX Applications .............. 162

Appendix A  ASCII Codes ......................................................... 165

Index .................................................................................... 169
Introduction

AutoCAD® is a general-purpose drafting system designed with an open architecture so you can customize and extend its many features. As a result, you can expand and shape AutoCAD according to your needs. The Customization Guide provides the details and procedures to implement customization features by modifying or creating ASCII text files. In addition, this guide introduces the application programming interfaces (APIs) to AutoCAD and includes pointers to the appropriate developer guide for each interface.

Your dealer can offer you independently developed applications and extensions that can further tailor AutoCAD to your needs.
How This Guide Has Changed

This release of AutoCAD opens the way for developers to access its open architecture with even more powerful and integrated programming interfaces. As a result, some information that was previously included in this guide is now presented separately.

AutoLISP® has been enhanced with a new development environment called Visual LISP™, and three documents are provided to assist you in developing AutoLISP applications. The Visual LISP Developer’s Guide explains how to use the Visual LISP integrated development environment (IDE), and how to build and run AutoLISP applications. The AutoLISP Reference defines, describes, and provides examples for every AutoLISP function. The AutoLISP Reference shows the syntax of a function and what a function returns. The Visual LISP Tutorial contains step-by-step instructions guiding you toward building a working Visual LISP application. These documents are available online.

The programmable dialog box reference that was formerly part of this guide has been incorporated into the Visual LISP Developer’s Guide.

DXF™ group codes define all of the objects in an AutoCAD drawing. These codes are used programmatically by AutoLISP and ObjectARX™ applications to manipulate drawing objects, and by developers and users to import and export drawing data. The DXF Reference, previously part of this guide, is provided as a Help file, in HTML format, and as a PDF file for printing. These files are available from the AutoCAD Technical Publications Web site at http://www.autodesk.com/techpubs/autocad/.

This revision of the Customization Guide focuses on the basic customization techniques available in AutoCAD to users with little or no programming experience. These modifications are accomplished by modifying or creating ASCII-based text files.

Using AutoCAD Documentation

In addition to this Customization Guide, several references help you learn and use AutoCAD, both online and in print. If you are new to AutoCAD, you can review AutoCAD concepts and AutoCAD features in the online AutoCAD 2000 Learning Assistance™ and try some of its exercises. If you’re upgrading from a release prior to AutoCAD 2000, read What’s New in AutoCAD 2000 (accessed on the AutoCAD Help menu) and refer to the
Customizing AutoCAD

The basic customization features for AutoCAD 2000 include the following:

Running external programs and utilities from within AutoCAD. You can, for example, copy a disk or delete a file from within AutoCAD by adding the appropriate external command to the program parameters file, acad.pgp. You can also create simple abbreviations, or aliases, for frequently used AutoCAD commands. For example, you might want to invoke the BHATCH command by entering b. You define a command alias in the program parameters file, acad.pgp.

Making a Help system for applications and in-house procedures. Help is accessible from menus and toolbars, and with AutoLISP, VBA, and ObjectARX programs. By creating a Help file, you can customize Help for AutoCAD commands, as well as for your own applications.

Creating custom dash and dot linetypes, hatch patterns, shapes, and text fonts. Create your own linetypes, hatch patterns, shapes, and text fonts that conform to your company standards and working methods.

Creating custom menus and toolbars to modify the AutoCAD user interface. Menus control many aspects of the AutoCAD interface. If you put the commands that you use most frequently in the most accessible locations on a menu and combine groups of commands into a single menu item, you can improve productivity. With menu customization, you can tailor the AutoCAD interface to meet your specific needs. In the menu file, you can define the behavior of your pointing device button and set the functionality and appearance of pull-down, screen, tablet, and image tile menus. Toolbars and accelerator keys are also defined by the menu file.

Customizing the status line. Use the DIESEL string expression language and the MODEMACRO system variable to provide additional information at the status line such as the date and time, system variable settings, or retrievable information using AutoLISP.

Automating repetitive tasks by writing scripts. A script is an ASCII text file containing AutoCAD commands that are processed like a batch file when you run


NOTE For the latest changes to AutoCAD information and procedures, refer to the readme.hlp file.
the script. For example, if a set of drawings needs to be plotted a certain way, you can write a script that opens each drawing, hides and displays various layers, and issues PLOT commands. You can then run the script at your convenience. Use scripts with slides to create automated or continuous presentations like those used at trade shows. A slide is a “snapshot” of the AutoCAD drawing area that cannot be edited. Use slides to include graphic information in image tile menus and dialog boxes.

Redefining or disabling selected AutoCAD commands, either at the Command prompt or as part of an AutoLISP or ObjectARX program. Redefine certain AutoCAD commands to issue supplementary messages and instructions, or create a drawing management system in which the QUIT command is redefined to write billing information to a log file before ending the editing session.

This guide introduces the application programming interfaces (APIs) to AutoCAD. AutoLISP, a specialized implementation of the LISP programming language, is an integral part of AutoCAD. You can adapt AutoCAD to your work needs by using AutoLISP to automate repetitive tasks and create new AutoCAD commands. You can write your own AutoLISP programs or use third-party and shareware programs. ActiveX™ Automation (previously known as OLE Automation) provides a modern alternative to AutoLISP and DCL (Dialog Control Language). You can access and control AutoCAD objects from any application that serves as an Automation controller (such as Visual Basic or any of the Microsoft® Office 97 applications). With the ObjectARX® programming interface, you can use the Microsoft C++® programming language to customize AutoCAD. You can run third-party ObjectARX application programs or write your own.

Appendix A, ASCII Codes. Shows the standard ASCII codes, which are useful in AutoLISP applications.
Typographical Conventions

To orient you to AutoCAD features as they appear on the screen, specific terms are set in typefaces that distinguish them from the body text. Throughout AutoCAD documentation, the following conventions are used.

<table>
<thead>
<tr>
<th>Typographical conventions</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text element</td>
<td></td>
</tr>
<tr>
<td>AutoCAD commands</td>
<td>ADCENTER, DBCONNECT, SAVE</td>
</tr>
<tr>
<td>AutoCAD system variables</td>
<td>DIMBLK, DWGNAME, LTSCALE</td>
</tr>
<tr>
<td>AutoCAD named-objects, such as linetypes and styles</td>
<td>DASHDOT, STANDARD</td>
</tr>
<tr>
<td>Prompts</td>
<td>Select object to trim or [Project/Edge/Undo]:</td>
</tr>
<tr>
<td>Instructions after prompt sequences</td>
<td>Select objects: Use an object selection method</td>
</tr>
<tr>
<td>File names and file name extensions</td>
<td>acad.exe, Readme file, .dwg file extension</td>
</tr>
<tr>
<td>Folder or directory names</td>
<td>Sample folder, c:\ACAD2000\support</td>
</tr>
<tr>
<td>Text you enter</td>
<td>At the Command prompt, enter shape.</td>
</tr>
<tr>
<td>Keys you press on the keyboard</td>
<td>CTRL, F10, ESC, ENTER</td>
</tr>
<tr>
<td>Keys you press simultaneously on the keyboard</td>
<td>CTRL+C</td>
</tr>
<tr>
<td>AutoLISP variable names, sample code, and text in ASCII files</td>
<td>The variable pi is preset to a value of pi</td>
</tr>
<tr>
<td>AutoLISP and DIESEL function names</td>
<td>command ads_command( )</td>
</tr>
<tr>
<td>Formal arguments specified in function definitions</td>
<td>The string and mode arguments</td>
</tr>
</tbody>
</table>
AutoCAD 2000 Changes and Enhancements

In AutoCAD 2000, many of the customization features of AutoCAD have been enhanced or changed. The general changes that affect customization are as follows:

- The *Customization Guide* has undergone significant changes since Release 14. These changes are described in “How This Guide Has Changed” on page 2.
- Multiple drawings can be opened within the same session of AutoCAD. To maintain script compatibility with older scripts, you may need to set the new SDI (single document interface) system variable to 0.
- The format of the external commands section of the `acad.pgp` file has changed. The obsolete `memory` field is now called `flags`, and is used to control aspects of the external application.
- Dictionary objects and symbol table objects, including layers, linetypes, and text styles, can have names of up to 255 characters in length, including extended characters. The `EXTNAMES` system variable still turns off extended symbol names for backward compatibility, but will be removed in a future version of AutoCAD.
- Layers have a setting to control whether they are plotted.
- Properties of single and multiple selection sets can be viewed and modified in the Properties window. This window replaces the `DDMODIFY` and `DDCHPROP` dialog boxes with a simpler interface. For command line modifications to selection sets, you can use the new `PROPERTIES` command.
- Visual LISP, an AutoLISP enhancement, offers an integrated development environment. Because Visual LISP has additional functionality, scripts may not behave in the same manner as AutoLISP.
- `DXFOUT` and `DXFIN` commands are now obsolete, but they are still accessible using scripts, AutoLISP, ActiveX Automation, and ObjectARX interfaces. DXF files are now written using the `SAVE`, `SAVEAS`, and `WBLOCK` commands, and can be opened and inserted into drawing files using the `OPEN` and `INSERT` commands. Support for the `DXFOUT` and `DXFIN` commands may not be provided in future releases.
- DXF files now have preview images associated with them. Thumbnail images are optionally created with DXF files using the `SAVE`, `SAVEAS`, and `WBLOCK` commands and the images can be previewed using `OPEN` and `INSERT`. 
Menu files have been enhanced to provide command-specific and object-specific shortcut menus.

- New control elements are available in toolbars.
- With the introduction of Multiple Design Environment (MDE) to AutoCAD, a new startup AutoLISP file has been introduced, acaddoc.lsp. The acad.lsp file is now intended to be used for application startup functions, and the acaddoc.lsp file is associated with document (drawing) initialization.

- Relative (also called global) menu swapping is no longer available. For example, the following syntax is no longer valid:
  
  ```lisp
  (menucmd "Gmenugroup1.menuname1=menugroup2.menuname2")
  ```

  However, other types of relative menu referencing are still valid.
Customization Basics

This chapter covers the basic AutoCAD® customization features. If you are new to customization, you should read this chapter first. It explains how to enter operating system commands from within AutoCAD and how to create abbreviations for frequently used commands. This chapter also describes how to develop a Help system for your own applications and procedures.
Basic Requirements

Most of the customization features described in this guide require that you modify or create ASCII text files. Therefore, you should be familiar with a text editor (such as Microsoft® Windows® Notepad) or a word processor (such as Microsoft Word) that can save files in ASCII format. You should also make backups of AutoCAD support files, such as menu files, before modifying them.

AutoCAD Environment

The default directory structure for the AutoCAD program and support files is designed to efficiently organize those files into logical groups. If this organization does not suit your needs, you can change it. However, some applications look for certain files in specific locations, and you should verify that your modifications do not conflict with the requirements of those applications. Without specifying the drive and directory prefix, AutoCAD can locate only files that are found in the library search path.

Library Search Path

AutoCAD searches for support files in the order specified by the library path, as follows:

- Current directory. (This is typically determined by the “Start In” setting in your shortcut icon.)
- Directory that contains the current drawing file.
- Directories listed in the search path specified in OPTIONS. For more information about the Support path, see “Specifying Search Paths, File Names, and File Locations” in chapter 3 of the User’s Guide.
- Directory that contains the AutoCAD program files.

Depending on the current environment, two or more directories may be the same.

If a file is not in this search path, you must specify both its path name and file name before AutoCAD can find it. For example, if you want to insert the part5.dwg drawing into your current drawing and it is not in the library search path, you must specify its full path name, as shown here:

Command: insert
Enter block name or [?] : /files2/olddwgs/part5

If the drawing exists at that location, AutoCAD prompts you to finish the INSERT command in the usual manner.
Sample Directory Structure

This section presents options to enhance your operating system directory structure and to make it easier to manipulate files. AutoCAD uses tree-structured directories and subdirectories.

It is recommended that you keep supplemental files (such as AutoLISP® applications and menu files) separate from the AutoCAD program and support files. This makes it easier to track possible conflicts and you can upgrade each application without affecting the others.

The sections that follow refer to the sample tree structure shown in the following figure as a method of organizing directories and files.

Sample AutoCAD tree structure

You can create a directory for your custom AutoLISP and Visual Basic applications, menu files, and other third-party applications (shown in the example as /AcadApps). If you want to maintain multiple drawing directories (for separate job files), you can create a directory, such as /AcadJobs, with subdirectories for each job.

Multiple Configurations

When you configure AutoCAD for a pointing device and plotter drivers, the information you supply is recorded in a configuration file. By default, the acad2000.cfg configuration file is placed in the directory that contains the AutoCAD program files, but you can specify an alternate path, file name, or both.
Typically, only a single configuration is necessary, but sometimes you may need multiple configurations. For example, if you use a mouse for most of your work but occasionally require a large digitizing tablet, you can set up your system to handle multiple configurations rather than reconfiguring each time you change devices.

The values of numerous AutoCAD system variables and the configuration options in the Options dialog box are stored in the configuration file. If you want different settings for these variables and operating parameters, you can save their respective values to different configuration files. For a list of the system variables and where they are stored, see appendix B, “System Variables,” in the Command Reference.

To take advantage of multiple configurations, you must set up AutoCAD to use different configuration files. Use the /c switch to specify alternate configuration files. See “Using Command Line Switches” in chapter 4, “Configuring and Optimizing AutoCAD,” in the Installation Guide.

Multiple Drawing Directories

Maintaining multiple drawing directories is not only convenient, but also often necessary. Keeping your drawing and other associated files in separate directories makes it easier to perform basic file maintenance. The scenario described in this section is based on the sample directory structure described earlier, but you can expand or alter it to meet your needs.

You can set up the /AcadJobs directory to contain your drawing subdirectories. The drawing subdirectories can contain other subdirectories that hold related support files for that particular drawing type or job. The /AcadJobs/Job1/Support directory can contain blocks and AutoLISP files specific to the drawing files in /AcadJobs/Job1. Specifying Support (with no path prefix) in the Support path, adds the Support directory within the current directory to the Support path. Notice that if you use the Options dialog box to specify a directory, AutoCAD creates a hard-coded path to that directory. To use the relative naming convention previously described, you must specify the Support path with the /s switch on the command line. See “Using Command Line Switches” in chapter 4, “Configuring and Optimizing AutoCAD,” in the Installation Guide.

If you make sure that the required drawing directory is the current directory when you start AutoCAD, all files and subdirectories in that directory are easily accessible. You can create a program icon or Start menu item that specifies the correct working directory for each job.

You can use a batch program as an alternative to using icons or menus. With batch programs you can create new job directories automatically. The following batch program verifies that a specified directory exists, sets that directory to be current, and then runs AutoCAD.
@echo off
C:
if exist \AcadJobs\Jobs\%1 goto RUNACAD
echo.
echo *** Creating \AcadJobs\Jobs\%1
echo *** Press Ctrl+C to cancel.
echo.
pause
mkdir \AcadJobs\Jobs\%1
:RUNACAD
cd \AcadJobs\Jobs\%1
start C:\AutoCAD\acad.exe

Using Notepad, or any ASCII text editor, save this batch program to a file named acad.bat. Be sure to change the drive and directory names to match those on your system. Place this file in a directory that is on your system search path (for example, C:\winnt). You can run this batch program using the Run command on the Start menu or by double-clicking the file in Explorer. If you saved the file as acad.bat, use the following syntax:

acad  jobname

where jobname is the name of the job directory to make current.

**Command Search Procedure**

When you enter a command, AutoCAD goes through a series of steps to evaluate the validity of the command name. A command can be a built-in command or system variable, an external command or alias defined in the acad.pgp file (described in the following section), or a user-defined AutoLISP command. Commands can also be defined by ObjectARX applications or a device driver command. You can enter a command on the command line or choose a command from the appropriate menu. Commands can be entered also from a script file or by an AutoLISP or ObjectARX application.

The following list describes the search order AutoCAD uses to validate a command name.

1. If the input is a null response (SPACEBAR or ENTER), AutoCAD uses the name of the last command issued. HELP is the default.
2. AutoCAD checks the command name against the list of built-in commands. If the command is in the list and is not preceded by a period (.), AutoCAD then checks the command against a list of undefined commands. If the command is undefined, the search continues. Otherwise, the command is run, unless another reason prevents it from doing so. Running it transparently or in Perspective mode might be impossible.
3. AutoCAD checks the command name against the names of commands defined by a device driver, and then by those defined by the display driver.
4. AutoCAD checks the command name against the external commands defined in the program parameters file (*acad.pgp). If the command name corresponds to a defined external command, that command runs, and the search is complete.

5. AutoCAD checks the command name against the list of commands defined by AutoLISP or ObjectARX applications. At this point, an autoloaded command is loaded. (For information about autoloading commands, see “Command Autoloader” on page 159.)

6. AutoCAD checks the command name against the list of system variables. If the command name is in the list, AutoCAD executes the SETVAR command, using the input as the variable name.

7. If the command name corresponds to a command alias defined in the program parameters file, AutoCAD uses the expanded command name and continues the search, starting a new search against the list of built-in commands.

8. If all the preceding steps fail, the search terminates with a warning message about illegal command names.

**Customizable Support Files**

AutoCAD uses support files for purposes such as storing menu definitions, loading AutoLISP and ObjectARX applications, and describing text fonts. Many support files are text files that you can modify with a text editor.

The following is a list of AutoCAD support files that can be edited (most are discussed in this guide). The support files are listed in alphabetical order by file extension. Please make backup copies of these files before modifying them.

<table>
<thead>
<tr>
<th>Customizable support files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*ahp</td>
<td>AutoCAD Help files. Associated help index files have the extension *.hdx.</td>
</tr>
<tr>
<td>asi.ini</td>
<td>Database connectivity link conversion mapping file.</td>
</tr>
<tr>
<td>*.dcl</td>
<td>AutoCAD Dialog Control Language (DCL) descriptions of dialog boxes.</td>
</tr>
<tr>
<td>*.lin</td>
<td>AutoCAD linetype definition files.</td>
</tr>
<tr>
<td>acad.lin</td>
<td>The standard AutoCAD linetype library file.</td>
</tr>
<tr>
<td>acadiso.lin</td>
<td>The standard AutoCAD ISO linetype library file.</td>
</tr>
<tr>
<td>*.lsp</td>
<td>AutoLISP program files.</td>
</tr>
</tbody>
</table>
### Customizable support files (continued)

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>acad.lsp</code></td>
<td>A user-defined AutoLISP routine that loads each time you start AutoCAD.</td>
</tr>
<tr>
<td><code>acaddoc.lsp</code></td>
<td>A user-defined AutoLISP routine that loads each time you start a drawing.</td>
</tr>
<tr>
<td><code>*mln</code></td>
<td>A multiline library file.</td>
</tr>
<tr>
<td><code>*mnl</code></td>
<td>AutoLISP routines used by AutoCAD menus. A MNL file must have the same file name as the MNU file it supports.</td>
</tr>
<tr>
<td><code>acad.mnl</code></td>
<td>AutoLISP routines used by the standard AutoCAD menu.</td>
</tr>
<tr>
<td><code>*mns</code></td>
<td>AutoCAD generated menu source files. Contains the command strings and macro syntax that define AutoCAD menus.</td>
</tr>
<tr>
<td><code>acad.mns</code></td>
<td>Source file for the standard AutoCAD menu.</td>
</tr>
<tr>
<td><code>*mnu</code></td>
<td>AutoCAD menu source files. Contain the command strings and macro syntax that define AutoCAD menus.</td>
</tr>
<tr>
<td><code>acad.mnu</code></td>
<td>Source file for the standard AutoCAD menu.</td>
</tr>
<tr>
<td><code>*pat</code></td>
<td>AutoCAD hatch pattern definition files.</td>
</tr>
<tr>
<td><code>acad.pat</code></td>
<td>The standard AutoCAD hatch pattern library file.</td>
</tr>
<tr>
<td><code>acadiso.pat</code></td>
<td>The standard AutoCAD ISO hatch pattern library file.</td>
</tr>
<tr>
<td><code>acad.pgp</code></td>
<td>The AutoCAD program parameters file. Contains definitions for external commands and command aliases.</td>
</tr>
<tr>
<td><code>fontmap.ps</code></td>
<td>The AutoCAD Font Map file. Used by PSIN; the catalog (or font map) of all fonts known to the AutoCAD PostScript® interpreter.</td>
</tr>
<tr>
<td><code>acad.psf</code></td>
<td>AutoCAD PostScript Support file; the master support file for the PSOUT and PSFILL commands.</td>
</tr>
<tr>
<td><code>acad.rx</code></td>
<td>Lists ObjectARX applications that load when you start AutoCAD.</td>
</tr>
<tr>
<td><code>*scr</code></td>
<td>AutoCAD script files. A script file contains a set of AutoCAD commands processed as a batch.</td>
</tr>
<tr>
<td><code>*shp</code></td>
<td>AutoCAD shape/font definition files. Compiled shape/font files have the extension .shx.</td>
</tr>
<tr>
<td><code>acad.unt</code></td>
<td>AutoCAD unit definition file. Contains data that lets you convert from one set of units to another.</td>
</tr>
</tbody>
</table>
**acad.pgp—Program Parameters File**

The AutoCAD program parameters file, *acad.pgp*, is an ASCII text file that stores command definitions. You can consider it a list of custom AutoCAD commands. When you enter a command that isn’t a normal command, AutoCAD looks for the command in *acad.pgp*. This file is divided into two sections. The first section defines external commands, and the second section defines command aliases. The file can contain comment lines preceded by a semicolon (;).

### Defining External Commands

While AutoCAD is running, you can invoke other programs or utilities, such as the following:

- Windows system commands and utilities, such as `start`, `type`, `dir`, or `copy`
- Applications such as text editors or word processors
- Database managers, spreadsheets, and communications programs
- User-supplied programs, such as batch files or Visual Basic applications

When you define an external command, you specify a command name to be used at the Command prompt and an executable command string that is passed to the operating system. Each line in the external commands section has five comma-delimited fields as follows:

```
command,[executable],[flags],[*,prompt],return_code]
```

- **command**: The command to enter at the Command prompt. If this name is an internal AutoCAD command name, it is ignored. The name is not case-sensitive.

- **executable**: This constant string is sent to the operating system when you enter the command name. It can be any command that you can execute at the operating-system prompt. The string can include switches or parameters. The case sensitivity of this string depends on the application you are running.
flags

This is a required bitcoded parameter. Add these integer values together in any combination to achieve the result you want.

0  Start the application and wait for it to finish.
1  Don't wait for the application to finish.
2  Run the application minimized.
4  Run the application “hidden.”
8  Put the argument string in quotes.

Bit values 2 and 4 are mutually exclusive; if both are specified only the 2 bit is used. Using values 2 or 4 without value 1 should be avoided, because AutoCAD becomes unavailable until the application has completed.

Bit value 8 allows commands like `del` to work properly with file names that have embedded spaces. Note that this eliminates the possibility of passing a space-delimited list of file names to these commands. If you prefer multiple file support, do not use the bit value 8.

prompt

This field is optional. It specifies the prompt to display on the AutoCAD command line. The response to this prompt is appended to the string supplied in the executable field. If the first character of the prompt field is an asterisk (*), the response can contain spaces and the user must press ENTER to terminate it. Otherwise, the response is terminated by either SPACEBAR or ENTER. If no prompt is specified, no input is requested; however, you must add a comma if a return code is to be supplied or if you want the prompt to have a trailing space.

return_code

This is an optional bitcoded parameter. You can add these integer values together in any combination to achieve the result you want. For example, if values 1 and 2 are required, you use 3 as the return code. The values are defined as follows (codes 0 and 4 are meaningless in a windowed environment and are therefore not included):
Chapter 1  Customization Basics

1  Loads a DXB file. AutoCAD loads the DXB file named $cmd.dxb into the drawing after the command is terminated. After the DXB file is loaded, the $cmd.dxb file is deleted. This action produces the same result as the DXBIN command.

2  Constructs a block definition from a DXB file. AutoCAD creates a block definition from the DXB file named $cmd.dxb. The response to the prompt field is used as the block name. This name must be a valid block name that does not currently exist in the drawing; therefore, this mode cannot redefine a previously defined block. After AutoCAD loads the DXB file, the $cmd.dxb file is deleted. The default name for the INSERT command is set to the newly defined block.

Using Windows System Commands

The start and cmd Windows system commands are very useful when defining external commands. If you specify an executable string that does not use the start or cmd commands, AutoCAD is unavailable until that window is closed.

The start command starts a separate window and runs a specified program or command. If start is used without any parameters, it opens a new command prompt window. The start command has many command line switches that affect the display of the new window. To launch a Windows application, use start without any switches. The start command is also very useful for starting a document that is associated with an application. For example, you can use start to directly open a document created with a word processor or an HTML file.

The cmd command opens a Command prompt window that acts as a shell of AutoCAD. This window must be closed before control returns to the AutoCAD Command prompt. Two command line switches, /c and /k, are useful for external commands. The /c switch carries out the specified command and then stops (the window closes). The /k switch carries out the specified command and then continues (the window remains open). When using the /k switch, you must close the command window (with the exit command).

In general, use start to start a new window or application that is to be a separate process from AutoCAD. Use cmd to run a batch file or command script that does not create a separate window, or to create a window that must be closed before control is passed back to AutoCAD. For more information about these commands and switches, see your Windows system command documentation.
Using Custom-Defined Commands

The following example defines three new commands: RUN, LISTSET and DXB2BLK.

RUN, cmd /c,0,*Batch file to run: ,
LISTSET,cmd /k SET,0
DXB2BLK,cmd /c DXBCOPY,0,DXB file: ,2

The RUN command runs a batch file or command script. The cmd command followed by the /c switch opens a command window, runs the batch file, and then closes.

The LISTSET command displays the current DOS environment variable settings. Because this example uses cmd /k rather than start, the command window must be closed before returning to AutoCAD. If you want this window to remain active, use start /realtime. For more information about these commands and switches, see your Windows system command documentation.

The DXB2BLK command creates a block definition from the specified DXB file. The DXB file converts all objects into lines. One beneficial by-product of this procedure is that it provides a simple method for exploding text objects into lines.

DXB2BLK passes the specified DXB file name to the dxbcopy batch file, which copies that file name to the file name $cmd.dxb. AutoCAD then creates a block from the specified DXB file. The name provided to the DXB file prompt is used as the new block name. To create the dxbcopy.cmd file, enter the following at a DOS prompt:

echo copy %1.dxb $cmd.dxb > dxbcopy.cmd

This creates the dxbcopy.cmd file in the current directory. Move this file to a directory that is in your DOS path, or explicitly specify the file's location in the acad.pgp file. For example, if the dxbcopy.cmd file is in D:\cad, enter the following in the external commands section of your acad.pgp file.

DXB2BLK, cmd /c D:\CAD\DXBCOPY,0,DXB file: ,2

To create a DXB file, choose AutoCAD DXB File Format as the current printer, and then plot to a file. For more information about configuring printers, see chapter 6, “Configuring Plotters and Printers” in the Installation Guide.

Command Aliases

You can abbreviate frequently used AutoCAD commands by defining aliases for them in the command alias section of acad.pgp. You can make any AutoCAD command, device driver command, or external command into an alias.
These two comma-delimited fields define a command alias in the `acad.pgp` file:

```plaintext
abbreviation,*command
```

- **abbreviation** This is the abbreviation of the command that you enter at the Command prompt.
- **command** This is the AutoCAD command being abbreviated. You must enter an asterisk (*) before the command name to identify the line as a command alias.

If you can enter a command transparently, you can also enter its alias transparently. When you enter the command alias, AutoCAD displays the full command name at the Command prompt and executes the command.

The following portion of a command alias section contains items similar to those in the standard `acad.pgp` file.

```plaintext
A,*ARC
C,*CIRCLE
CP,*COPY
```

The command alias section can include commands with the special minus symbol (–) prefix, such as those listed here. This lets you create aliases that access the command line version of certain commands.

```plaintext
BH,*-BHATCH
BD,*-BOUNDARY
```

**NOTE** You cannot use command aliases in command scripts. Using command aliases in menu files is not recommended.

---

**Custom Online Documentation**

It is a good practice to document all modifications or enhancements you make to AutoCAD. Complete and accurate documentation helps to ensure that anyone using your applications or menus can find the appropriate information.

Through AutoCAD you can easily access two types of user-definable documentation: Windows Help files and HTML files. Windows Help files provide highly customizable, context-sensitive online documentation that is included in a single file. Windows Help files are well-suited for applications that require dialog box or interface context sensitivity or documentation that does not need to be updated frequently. Many programs are available...
that simplify the creation and maintenance of HTML files. HTML files are very easy to update and can be indispensable for companies that need to provide documentation to remote sites.

The AutoCAD Help file format (AHP) is still accepted by AutoCAD but may not be available in a future release. You should convert any AHP files still in use, into WinHelp or HTML files. For instructions about how to convert AHP files to other formats see “AutoCAD Help Files” on page 27.

AutoCAD provides additional help features through the menu file (see “Menu-Specific Help” on page 122).

**Windows Help Files**

Windows Help (WinHelp) files provide an online documentation system for Windows. These files can be used as stand-alone documentation as well as from within AutoCAD and other Windows-based applications. WinHelp files can contain textual information in addition to graphics and multimedia. You can augment the functionality of WinHelp files through WinHelp macros and API functions, making them a very powerful documentation tool.

The next section provides an overview of the process and components involved in building a WinHelp file, and explains how these files are used with AutoCAD. For details you can refer to books and applications that are available. At a minimum, you’ll need the Microsoft Help Workshop. This application contains the Help compiler and a WinHelp file, which provides all of the essential information you need to build your own help files.

**Windows Help System Components**

The WinHelp compiler generates help files from specially coded RTF (rich text format) files. Because RTF is a standard output format from Microsoft Word, Word is the common authoring environment for WinHelp files. However, it is possible to create help files from hand-coded RTF files or from the output of other applications.

A WinHelp file is compiled from one or more RTF files. The Help Project File (.hpj) lists all of the RTF files to be included, in addition to other parameters, such as the location of graphic files and macro definitions and the Help window size, location, and colors. The Help compiler reads the HPJ file and builds the final Help file (.hlp) as specified. To view the Help file generated by the Help compiler you run it from File Manager or Explorer.

Windows Help files use the Help Topics dialog box to display the Contents, Index, and Find tabs. The Contents tab provides a graphical interface to the topics in one or more Help files. The entries and layout of the Contents tab are defined by a Contents file (*.cnt). To enable you and third-party developers to
augment the standard AutoCAD Help system, a method is provided for you to append additional topics to the Contents tab.

The `include.cnt` file, saved in the `help` directory, references all Contents files to append topics to the AutoCAD Contents tab. The `include.cnt` file should contain only `:Include` statements, such as the following:

```
:Include standards.cnt
:Include myhelp.cnt
:Include webhelp.cnt
```

The previous example adds the topics listed in the `standards.cnt`, `myhelp.cnt`, and `webhelp.cnt` files to the AutoCAD Contents tab. For information about the contents of CNT files, see “Adding Windows Help Files to the Contents Tab” on page 23 and “Adding HTML Files to the Contents Tab” on page 25.

---

**NOTE** After modifying the `include.cnt` file or any referenced CNT files, you must delete the `acad.gid` file before your changes will take effect.

The following list summarizes the files that are used or created by the WinHelp system.

### Files used by Windows Help

<table>
<thead>
<tr>
<th>File type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTF</td>
<td>WinHelp source file</td>
</tr>
<tr>
<td>HPJ</td>
<td>WinHelp project file</td>
</tr>
<tr>
<td>HLP</td>
<td>Compiled WinHelp file</td>
</tr>
<tr>
<td>CNT</td>
<td>Contents file</td>
</tr>
<tr>
<td>GID</td>
<td>WinHelp configuration file. Binary file created by the <code>winhelp32</code> executable. Contains information about the help file, including contents file links, file names of all included help files, keywords, and location of files.</td>
</tr>
<tr>
<td>FTS</td>
<td>Full-text search file. Binary file created by the <code>winhelp32</code> executable the first time you do a full-text search.</td>
</tr>
<tr>
<td>FTG</td>
<td>Full-text search groups file. Binary file created by the <code>winhelp32</code> executable.</td>
</tr>
</tbody>
</table>
Using Windows Help Files with AutoCAD

If you use WinHelp files as documentation for custom applications or company standards, you want the information to be easily available from within AutoCAD. The following sections present methods that AutoCAD provides for accessing WinHelp files.

Adding Windows Help Files to the Contents Tab

You can add your own WinHelp files (and links to other applications) to the Contents tab of the AutoCAD Help system. The `include.cnt` file (described in the section “Windows Help System Components” on page 21) defines additional entries that are displayed on the Contents tab. If you want the additional files to be included in keyword and full-text searches, use the `:Index` statement as shown in the following sample.

One possible addition to the AutoCAD Help system might be your company standards. The following example provides access to topics within the `standards.hlp` file and uses the `:Index` statement to include the `standards.hlp` file in any keyword or full-text searches.

```
:Index Standards=standards.hlp
1 Office Standards
   2 Architectural
      3 Overview=arch_overview@standards.hlp
      3 Design=arch_design@standards.hlp
      3 Drafting=arch_drafting@standards.hlp
      3 Reports=arch_reports@standards.hlp
   2 Engineering
      3 Overview=eng_overview@standards.hlp
      3 Design=eng_design@standards.hlp
      3 Drafting=eng_drafting@standards.hlp
      3 Reports=eng_reports@standards.hlp
   2 Support Staff
      3 Overview=supp_overview@standards.hlp

Remember to add the following line to the `help/include.cnt` file:

```
:Include standards.cnt
```

Now, delete the `acad.gid` file (if this file is not deleted, your additions to the Contents tab may not be displayed). The next time you run the `acad.hlp` file (from File Manager or Explorer) or choose AutoCAD Help Topics from the
AutoCAD Help menu, your new entries will be displayed after the existing entries.

Running Windows Help Files from the Command Line
A quick and easy way to provide access to your custom help file from the command line is to add the following line to the external commands section of the \acad.pgp file:

```
MYHELP, start c:\acad2000\help\myhelp.hlp, 0
```

This external command defines the new AutoCAD command MYHELP. Remember to use the REINIT command to reload the \acad.pgp file.

Running Windows Help Files from AutoLISP
The following AutoLISP code calls the myhelp.hlp WinHelp file:

```
(help "myhelp.hlp")
```

The `help` function accepts additional arguments. For more information, see “help” in the AutoLISP Reference.

The `setfunhelp` function associates context-sensitive help (when the user presses F1) with a user-defined command. For more information, see “setfunhelp” in the AutoLISP Reference.

You can also use the `startapp` function to run a WinHelp file. To display a WinHelp file with `startapp`, you must pass the application name (winhlp32 for WinHelp 4.0 files and winhelp for older WinHelp files) followed by the Help file name. For example, the following code uses `startapp` to display the bighelp.hlp file:

```
(startapp "winhlp32" (findfile "help/bighelp.hlp"))
```
For more information, see “startapp” in the AutoLISP Reference.

NOTE Application path or file names with embedded spaces must be explicitly quoted. Because the file name argument must be explicitly defined, use the findfile function to provide the full path and file name.

**HTML Files**

As more information is made available on the Internet, HTML files (Web pages) may become the documentation medium of choice. One of the key advantages of HTML documentation is that it can be updated without requiring users to reinstall or modify anything on their local systems. Linking to remote sites is most efficient if you have a direct Internet connection; it can be frustrating and counter productive if you do not. However, even without accessing the Internet, you can make efficient use of HTML files on your local system.

**Using HTML Files with AutoCAD**

If you want easy access to remote Web sites or if you want to use HTML as documentation for custom applications or company standards, you can set up AutoCAD to have easy access to this information. The following sections present methods that AutoCAD provides for accessing these files.

**Adding HTML Files to the Contents Tab**

You can add your own HTML files, or links to Web pages on the Internet to the Contents tab of the AutoCAD Help system. The include.cnt file defines additional entries that display on the Contents tab. For more information about CNT files, see “Windows Help System Components” on page 21.

For example, if you want to be able to easily access your HTML documentation, stuff1.htm, stuff2.htm, and stuff3.htm, in addition to a few important Web sites, create the help/morehelp.cnt file with the following contents:

```
1 Yet More Help
2 Our Info
3 Stuff 1=IEF("stuff1.htm","",4)
3 Stuff 2=IEF("stuff2.htm","",4)
3 Stuff 3=IEF("stuff3.htm","",4)
2 Related Web Sites
3 BeamsRUs=IEF("http://www.BeamsRUs.com","",4)
3 TheOtherGuys=IEF("http://www.TheOtherGuys.com","",4)
```

Then add the following line to the help/include.cnt file:

```
:Include morehelp.cnt
```
Now, delete the `acad.gid` file. If this file is not deleted, your additions to the Contents tab may not be displayed. The next time you run the `acad.hlp` file (from File Manager or Explorer) or choose AutoCAD Help Topics from the AutoCAD Help menu, your new entries are displayed after the existing entries.

**Launching HTML Files from the Command Line**

A simple way to start your browser with a specified HTML file on the command line is to add the following line to the external commands section of the `acad.pgp` file:

```
MYHTML, start c:\acad2000\help\myhtml.html,0
```

This external command defines the new AutoCAD command `MYHTML`. Remember to use the `REINIT` command to reload the `acad.pgp` file.

You can also use AutoLISP to define your own commands that call HTML files.

**Launching HTML Files from AutoLISP**

To display an HTML file with the `startapp` function, you must pass the application name, followed by the HTML file name. For example, the following code uses `startapp` to display the `myweb.html` file:

```
(setq app "\"C:\Netscape\Program\Netscape.exe\"")
(startapp app (findfile "help/myweb.html"))
```

For descriptions of the `startapp` function, see the Visual LISP Developer’s Guide.

**NOTE** Application path or file names with embedded spaces must be explicitly quoted. Because the file name argument must be explicitly defined, use the `findfile` function to provide the full path and file name.
AutoCAD Help Files

The AutoCAD Help file format (AHP) is still accepted by AutoCAD but may not be available in a future release. You should convert any AHP files still in use, into WinHelp or HTML files. The AHP file format is no longer described in this guide.
Linetypes and Hatch Patterns

AutoCAD® includes libraries of standard linetypes and hatch patterns. You can use the linetypes and hatch patterns as they are, modify them, or create your own.

If you use PostScript® fills you can define your own patterns. See “*fill—Defining Your Own PostScript Fill Patterns” on page 67.
Linetype Definition Files

AutoCAD linetypes are defined by linetype definition files. These files have the extension .lin. AutoCAD linetypes are composed of a series of dots and dashes separated by spaces, and can contain embedded shape and text objects. The default linetype file is acad.lin. You can print this file to better understand how to construct linetypes.

Linetypes that are composed of only dots, dashes, and spaces are considered simple linetypes. Linetypes that contain embedded shape and text objects along with dots, dashes, and spaces are complex linetypes. Although these two types of linetypes are handled similarly by AutoCAD, their definitions are considerably different.

Creating and Modifying Linetype Definitions

Two methods are available for creating and modifying linetype definitions. You can edit the LIN file using a text editor or word processor, or you can use the LINETYPE command’s Create option. You cannot create or modify complex linetypes at the command line.

A linetype file can contain many linetype definitions. You can add your custom linetypes to the acad.lin file or start building your own linetype library file. Comments can be included in a LIN file. Any text on a line that begins with a semicolon is ignored.

Defining Linetypes

Each linetype is defined on two lines in a LIN file. The first line defines the linetype name and provides for an optional description.

*linetype-name [, description]

This line must begin with an asterisk and must be immediately followed by the linetype name. If the description is provided, it must be separated from the name by a comma and no longer than 47 characters. The description is not used by AutoCAD; it is intended to help you visualize the appearance of the linetype.

The second line is the code that describes the actual pattern.

alignment, patdesc-1, patdesc-2, ...

This line begins with the alignment code (currently only A is allowed), followed by a comma-delimited list of pattern descriptors (no spaces are allowed).
When you create a linetype, it is not loaded into your drawing automatically. Use the Load option of the LINETYPE command.

**Simple Linetypes**

The following simple linetype, DD1, is an example of a repeating pattern:

- Dash, 0.5 drawing units long
- Space, 0.25 drawing units long
- Dot
- Space, 0.25 drawing units long

The linetype would be defined as

```
*DD1,____ . ____ . ____ . ____ .
A,.5,–.25,0,–.25
```

DD1 is the linetype name, and the *description* field is the description of the linetype displayed by the LAYER Ltype ? command sequence. In this case, the description is a simple representation of the dash-dot pattern.

The description is optional and can be a sequence of dots, spaces, and dashes, or a comment such as “Use this linetype for hidden lines.” If you omit the description, do not put a comma after the linetype name. If you include a description, it should be no longer than 47 characters long.

The *alignment* field specifies the action for pattern alignment at the ends of individual lines, circles, and arcs. Currently, AutoCAD supports only one alignment action. You specify it by entering A in the field. The alignment action is entered into the definition automatically when you use the LINETYPE Create option. However, if you create the linetype definition using a text editor, you need to include the A yourself. AutoCAD rejects any other character in the alignment field.

Each *pat-n* field specifies the length of a segment making up the linetype. If the length is positive, a pen-down segment will be drawn. A negative length denotes a pen-up (blank) segment. A dash length of 0 draws a dot. You can enter up to 12 dash-length specifications per linetype, provided they fit on one 80-character line.

With A-type alignment, AutoCAD guarantees that the endpoints of lines and arcs start and end with a dash. For example, suppose you create a linetype called CENTRAL that displays the repeating dash-dot sequence commonly used as a centerline. AutoCAD adjusts the dash-dot sequence on an individual line basis so that dashes and line endpoints coincide. The pattern fits the line so that at least half of the first dash specification begins and ends the line. If necessary, the first and last dashes are lengthened. If a line is too short
to hold even one dash-dot sequence, AutoCAD draws a continuous line between the endpoints. For arcs, the pattern is also adjusted so that dashes are drawn at the endpoints. Circles do not have endpoints, but AutoCAD adjusts the dash-dot sequence to provide reasonable displays.

The A-type alignment requires that the first dash length be 0 or greater (a dot or pen-down segment). The second dash length should be less than 0 (a pen-up segment). You must have at least two dash specifications for this alignment. Between the start and end dashes, the pattern dash specifications are drawn sequentially, beginning with the second dash specification and restarting the pattern with the first dash specification when required.

**Complex Linetypes**

Complex linetype descriptions are placed in LIN files along with simple linetypes. A complex linetype specifies a single, possibly broken, line interspersed with symbols. This linetype can denote utilities, boundaries, contours, and so on. As with simple linetypes, complex lines are dynamically drawn as the user specifies vertices. Shapes and text objects embedded in lines are always displayed completely; they are never trimmed.

The syntax for complex linetypes is similar to that of simple linetypes in that it is a comma-delimited list of pattern descriptors. Complex linetypes can include shape and text objects as pattern descriptors, as well as the dash-dot descriptors of simple linetypes.

The syntax for both shape and text object descriptors in a linetype description is as follows:

```
shape           [shapename,shxfilename] OR
                [shapename,shxfilename,transform]

text            ["string",stylename] OR
                ["string",stylename,transform]
```

where transform is optional and can be any series of the following (each preceded by a comma):

- **R=#=#** Relative rotation
- **A=#=#** Absolute rotation
- **S=#=#** Scale
- **X=#=#** X offset
- **Y=#=#** Y offset

In this syntax, #=# is a signed decimal number (1, –17, 0.01, and so on), the rotation is in degrees, and the remaining options are in linetype-scaled drawing units. The preceding transform letters, if they are used, must be followed by an equal sign and a number.
Shapes in Complex Linetypes

The following linetype definition defines a linetype named CON1LINE that is composed of a repeating pattern of a line segment, a space, and the embedded shape CON1 from the es.shx file. (Note that the es.shx file must be in the support path for the following example to work properly.)

*CON1LINE, --- [CON1] --- [CON1] --- [CON1]
A,1.0,-0.25,[CON1,es.shx],-1.0

Except for the code enclosed in square brackets, everything is consistent with the definition of a simple linetype. This example shows the minimal linetype definition that embeds a shape in a linetype.

As previously described, a total of six fields can be used to define a shape as part of a linetype. The first two are mandatory and position-dependent; the next four are optional and can be ordered arbitrarily. The following two examples demonstrate various shape definition field entries.

[CAP,es.shx,S=2,R=10,X=0.5]
This code draws the CAP shape defined in the es.shx shape file with a scale of two times the unit scale of the linetype, a tangential rotation of 10 degrees in a counterclockwise direction, and an X offset of 0.5 drawing units before shape elaboration takes place.

[DIP8,pc.shx,X=0.5,Y=1,R=0,S=1]
This code draws the DIP8 shape defined in the pc.shx shape file with an X offset of 0.5 drawing units before shape drawing takes place, and a Y offset of one drawing unit above the linetype with a scale equal to the unit scale of the linetype and 0 rotation.

The following syntax defines a shape as part of a complex linetype.

[shapename,shapefilename,scale,rotate,xoffset,yoffset]
The definitions of the fields in the syntax follow.

shapename The name of the shape to be drawn. This field must be included. If it is omitted, linetype definition fails. If shapename does not exist in the specified shape file, continue with linetype drawing without the embedded shape.

shapefilename The name of a compiled shape definition file (SHX). If it is omitted, linetype definition fails. If shapefilename is unqualified (that is, no path is specified), search the library path for the file. If shapefilename is fully qualified and not found at that location, remove the prefix and search the library path for the file. If it is not found, continue with linetype elaboration without the embedded shape.
**scale**  
$s=value$. The scale of the shape is used as a scale factor by which the shape’s internally defined scale is multiplied. If the shape’s internally defined scale is 0, the $s=value$ alone is used as the scale.

**rotate**  
$r=value$ or $a=value$. $r=value$ signifies relative or tangential rotation with respect to the lines elaboration. $a=value$ signifies absolute rotation of the shape with respect to the origin; all shapes have the same rotation regardless of their relative position to the line. The value can be appended with $d$ for degrees (if omitted, degree is the default), $r$ for radians, or $g$ for grads. If rotation is omitted, 0 relative rotation is used.

**xoffset**  
$x=value$. This field specifies the shift of the shape in the $X$ axis of the linetype computed from the end of the linetype definition vertex. If $xoffset$ is omitted or is 0, the shape is elaborated with no offset. Include this field if you want a continuous line with shapes. This value is not scaled by the scale factor defined by $s=value$.

**yoffset**  
$y=value$. This field specifies the shift of the shape in the $Y$ axis of the linetype computed from the end of the linetype definition vertex. If $yoffset$ is omitted or 0, the shape is elaborated with no offset. This value is not scaled by the scale factor defined by $s=value$.

**Text in Complex Linetypes**

Complex linetypes with text are used primarily for incorporating text as the shape to be elaborated. The primary difference between the use of shapes and the use of text is that text is associated with a text style in the drawing while shapes are associated directly with a shape file. The style associated with the linetype must be loaded into the drawing prior to loading the linetype.

The following is an example of a complex linetype definition that includes a text style.

```
*MCline, --- MC --- MC --- MC
A,1.0,-0.25,["MC",mystyle,S=1,R=0,X=0,Y=-0.25],-1.25
```

*MCline* is the name of the linetype, and "--- MC --- MC --- MC" is the ASCII description. The syntax of the second line of the linetype definition is as follows:

```
["string",style,S=scale,R=rotate,X=xoffset,Y=yoffset]
```
The definitions of the fields in the syntax follow.

`string`  
The text to be used in the complex linetype.

`style`  
The name of the text style to be elaborated. The specified text style must be included. If it is omitted, use the currently defined style.

`scale`  
`s=value`. The scale of the style is used as a scale factor by which the style's height is multiplied. If the style's height is 0, the `s=value` alone is used as the scale.

Because the final height of the text is defined by both the `s=value` and the height assigned to the text style, you will achieve more predictable results by setting the text style height to 0. Additionally, it is recommended that you create separate text styles for text in complex linetypes to avoid conflicts with other text in your drawing.

`rotate`  
`r=value or a=value`. `r=` signifies relative or tangential rotation with respect to the lines elaboration. `a=` signifies absolute rotation of the text with respect to the origin; all text has the same rotation regardless of its relative position to the line. The value can be appended with `d` for degrees (if omitted, degree is the default), `r` for radians, or `g` for grads. If rotation is omitted, 0 relative rotation is used.

Rotation is centered between the baseline and the nominal cap heights box.

`xoffset`  
`x=value`. This field specifies the shift of the text in the X axis of the linetype computed from the end of the linetype definition vertex. If `xoffset` is omitted or is 0, the text is elaborated by using the lower-left corner of the text as the offset. Include this field if you want a continuous line with text. This value is not scaled by the scale factor that is defined by `s=`.

`yoffset`  
`y=value`. This field specifies the shift of the text in the Y axis of the linetype computed from the end of the linetype definition vertex. If `yoffset` is omitted or is 0, the text is elaborated by using the lower-left corner of the text as the offset. This value is not scaled by the scale factor that is defined by `s=`.
Creating Hatch Patterns

Defining a hatch pattern requires knowledge, practice, patience, and a text editor. You can add a pattern to the `acad.pat` library file or store it in a file by itself, in which case the file name must be the same as the pattern name. For instance, a pattern named `PIT` would reside in the `pit.pat` file.

Regardless of where the definition is stored, the pattern has the same format. It has a header line that looks like this:

```
*patter-name [, description]
```

It also has one or more line descriptors of the following form:

```
angle, x-origin, y-origin, delta-x, delta-y [, dash-1, dash-2, ...]
```

For example, a pattern called `L45` that hatches with 45-degree lines separated by a spacing of 0.5 is defined as follows:

```
*L45, 45 degree lines
45, 0,0, 0,0.5
```

This simple pattern specifies that a line is to be drawn at an angle of 45 degrees, that the first line of the family of hatch lines is to pass through the (0,0) drawing origin, and that the spacing between hatch lines of the family is to be 0.5 drawing units. `L45` is the name of the pattern, and the description field is the optional description of the pattern displayed by the ? option of the `HATCH` command. (If you omit the description, do not include a comma after the pattern name.)

Each line in a pattern definition file can contain up to 80 characters. AutoCAD ignores blank lines and text to the right of a semicolon.

Using the AutoCAD Pattern File—`acad.pat`

The default hatch patterns used by AutoCAD are stored in the `acad.pat` file. You can add pattern definitions to this file or create your own files, as previously described.

The Boundary Hatch and Hatch Pattern Palette dialog boxes display the names of all the hatch patterns defined in the `acad.pat` file. You can add new hatch patterns to the dialog boxes by adding the hatch definitions to the `acad.pat` file. When a slide name in `acad.slb` matches a pattern name in `acad.pat`, the slide is also displayed by the dialog boxes. See BHATCH in the Command Reference.
Constructing Hatch Patterns

A hatch pattern is made up of one or more pattern lines. AutoCAD imposes no limit on the number of lines. Each pattern line is considered the first member of a line family, which is created by applying the delta offsets in both directions to generate an infinite family of parallel lines. The delta-y value sets the spacing between members of the family (that is, the value is measured perpendicular to the lines). The delta-x value sets the displacement between members of the family in the direction of the line; it is meaningful only for dashed lines. (A line is considered to be of infinite length; a dash pattern is superimposed on the line.)

The process of hatching consists of expanding each line in the pattern definition to its infinite family of parallel lines. Selected objects are checked for intersections with any of these lines; any intersections cause the hatch lines to be turned on and off according to the hatching style. If the hatch line is dashed, it is drawn with the dash pattern in those areas.

Because each family of hatch lines is generated by parallel transport from an initial line with an absolute origin, the hatching of adjacent areas always aligns correctly.

Patterns with Dashed Lines

To define dashed line patterns, you append dash-length items to the end of the line definition item. Each dash-length item specifies the length of a segment making up the line. If the length is positive, a pen-down segment will be drawn. If the length is negative, the segment is pen-up, and it won’t be drawn. The pattern starts at the origin point with the first segment, and cycles through the segments in circular fashion. A dash length of 0 draws a dot. You can specify up to six dash lengths per pattern line.

For example, modify a pattern for 45-degree lines to draw dashed lines with a dash length of 0.5 units and a spacing between dashes of 0.5 units. Such a pattern would be defined as follows:

*DASH45,Dashed lines at 45 degrees
45, 0,0, 0,.5, .5,-.5

This is the same as the original 45-degree pattern, but you have added a dash specification to the end. The pen-down length is 0.5 units, and the pen-up length is 0.5, meeting the stated objectives. If you wanted to draw a 0.5-unit dash, a 0.25-unit space, a dot, and a 0.25-unit space before the next dash, the definition would be as follows:

*DDOT45,Dash dot dash pattern: 45 degrees
45, 0,0, 0,.5, .5,-.25,0,.25
Now consider the effect of delta-x specifications on dashed-line families. First, consider the following definition:

*GOSTAK
0, 0, 0, .5, .5, -.5

This draws a family of lines separated by 0.5, with each line broken equally into dashes and spaces. Since delta-x is 0, the dashes in each family member align. An area hatched with this pattern would look like this:

- - - - - - - - -
- - - - - - - - -
- - - - - - - - -

Now change the pattern as follows:

*SKEWED
0, 0, .5, .5, .5, -.5

It is the same, except that you have set delta-x to 0.5. This setting offsets each successive family member by 0.5 in the direction of the line (in this case, parallel to the X axis). Because the lines are infinite, the dash pattern slides down the specified amount. The hatched area would look like this:

- - - - - - - - -
- - - - - - - - -
- - - - - - - - -

So far, the patterns described here use origin points of (0,0). One member of the line family passes through the origin, with its dash pattern starting at that point. In composing more complex patterns, you need to carefully specify the starting point, offsets, and dash pattern of each line family to form the hatch pattern correctly. In the following example, you want to draw a squared-off, inverted U pattern (one line up, one over, and one down). The pattern repeats in every unit, and each unit is 0.5 high and wide.

![Pattern Example]

The pattern is defined as follows:

*IUS, Inverted U's
90, 0, 0, 0, 1, .5, -.5
0, 0, .5, 0, 1, .5, -.5
270, .5, .5, 0, 1, -.5, -5

The first line (the up bar) is a simple dashed line with (0,0) origin. The second line (the top bar) should begin at the end of the up bar, so its origin is (0,.5).
The third line (the down bar) must start at the end of the top bar—at (.5,.5) for the first instance of the pattern—so its origin is at this point. The third line of the pattern could be the following:

90, .5,0, 0,1, .5,-.5

or

270, .5,1, 0,1, -.5,.5

The dashed pattern starts at the origin points and continues in the vector direction given by the angle specification. Therefore, two dashed-line families that are opposed 180 degrees are not alike, but two solid-line families are alike.

The following example is a more complex pattern of six-pointed stars.

Use this example to refine your skills at pattern definition. (Hint: 0.866 is the sine of 60 degrees.)

*STARS,Star of David
0, 0,0, 0,.866, .5,-.5
60, 0,0, 0,.866, .5,-.5
120, .25,.433, 0,.866, .5,-.5
Shapes, Fonts, and PostScript Support

With AutoCAD®, you can define shapes to use as drawing symbols and text fonts. This chapter explains how to create and compile your own shape and font files. It also introduces the advanced PostScript features of AutoCAD.
Using Shape Files

*Shapes* are objects that you use like blocks. First you use the LOAD command to load the compiled shape file containing the shape definition. Then you use the SHAPE command to insert shapes from the file into your drawing. You can specify the scale and rotation to use for each shape as you add it. AutoCAD SHP fonts are a special type of shape file, and are defined in the same way as shape files.

Blocks are more versatile and easier to use and apply than shapes. However, shapes are more efficient for AutoCAD to store and draw. User-defined shapes are helpful when you must insert a simple part many times and when speed is important.

Compiling Shape/Font Files

You enter the description of shapes in a specially formatted text file with a file extension of `.shp`. To create the file, use a text editor or word processor that enables you to save in ASCII format, and then compile the ASCII file. Compiling a shape definition file (SHP) generates a compiled shape file (SHX).

**To compile a shape or font file**

- On the Command line, enter `compile`.

In the Select Shape File dialog box, you can select a shape definition file (SHP) or PostScript font file (PFB). After you select the file name, compiling begins. If AutoCAD finds an error in the shape descriptions, a message is displayed telling you the type of error and the line number. When compiling is complete, the following messages are displayed:

Compilation successful.
Output file `name.shx` contains `nnn` bytes.

The compiled file has the same name as the shape definition file but with a file type of SHX. If the shape definition file defines a font, you use the `STYLE` command to define a text style. Then, you use one of the text placement commands (TEXT, DTEXT, or MTEXT) to place the characters in the drawing. If the shape definition file defines shapes, you use the LOAD command to load the shape file into the drawing. Then, you use the SHAPE command to place the individual shapes in the drawing (similar in concept to the INSERT command).
Compiling PostScript Fonts

To use a Type 1 PostScript® font in AutoCAD you must first compile it into an AutoCAD shape file. The COMPILE command accepts both SHP and PFB files as input and generates an SHX file. Compiled versions of PostScript fonts can take a lot of disk space, so compile only those fonts you use frequently.

AutoCAD cannot compile and load every Type 1 font. The PostScript font facilities in AutoCAD are intended to process a subset of Adobe fonts. If you receive an error while compiling a PostScript font, the resulting SHX file (if one is generated) may not load into AutoCAD.

For more information on the Adobe® Type 1 font format, refer to Adobe Type 1 Font Format Version 1.1. When you’ve purchased and installed these fonts, you can begin using them with AutoCAD.

NOTE Make sure you understand any copyright that accompanies the PostScript fonts you use. The same copyright restrictions generally apply to the SHX form of fonts you’ve compiled.

Creating Shape Definition Files

AutoCAD font and shape files (SHX) are compiled from shape definition files (SHP). You can create or modify shape definition files with a text editor or word processor that saves files in ASCII format.

The syntax of the shape description for each shape or character is the same regardless of the final use (shape or font) for that shape description. If a shape definition file is to be used as a font file, the first entry in the file describes the font itself rather than a shape within the file. If this initial entry describes a shape, the file is used as a shape file.

AutoCAD comes with two sample shape files: pc.shx and es.shx. One is for printed circuit layout and the other is for electronic schematics. Examining the contents of the associated shape definition files (SHP) and modifying their shape descriptions can help you master the definition of AutoCAD shapes.

Being able to create your own shape definitions is a valuable skill. Keep in mind, however, that this is a very complex subject to learn and requires patience.
Shape Descriptions

Each line in a shape definition file can contain up to 128 characters. Longer lines cannot be compiled. Because AutoCAD ignores blank lines and text to the right of a semicolon, you can embed comments in shape definition files.

Each shape description has a header line of the following form and is followed by one or more lines containing specification bytes, separated by commas and terminated by a 0.

```
*shapenumber,defbytes,shapename
specbyte1,specbyte2,specbyte3,...,0
```

The following describes the fields of a shape description:

- **shapenumber**: A number, unique to the file, between 1 and 258 (and up to 65535 for Unicode fonts), and preceded by an asterisk (*). Non-Unicode font files use the shape numbers 256, 257, and 258 for the symbolic identifiers Degree_Sign, Plus_Or_Minus_Sign, and Diameter_Symbol. For Unicode fonts these glyphs appear at the U+00B0, U+00B1 and U+2205 shape numbers and are part of the “Latin Extended-A” subset.

- **defbytes**: The number of data bytes (specbytes) required to describe the shape, including the terminating 0. The limit is 2,000 bytes per shape.

- **shapename**: The shape name. Shape names must be uppercase to be recognized. Names with lowercase characters are ignored and are usually used to label font shape definitions.

- **specbyte**: A shape specification byte. Each specification byte is a code that defines either a vector length and direction or one of a number of special codes. A specification byte can be expressed in the shape definition file as either a decimal or hexadecimal value. This section uses both decimal and hexadecimal specification byte values for its examples (as do many of the shape definition files). If the first character of a specification byte is a 0 (zero), the two characters that follow are interpreted as hexadecimal values.
Vector Length and Direction Code

A simple shape specification byte contains vector length and direction encoded into one specification byte (one *specbyte* field). Each vector length and direction code is a three-character string. The first character must be a 0, which indicates to AutoCAD that the next two characters are interpreted as hexadecimal values. The second character specifies the length of the vector in units. Valid hexadecimal values range from 1 (one unit long) through F (15 units long). The third character specifies the direction of the vector. The following figure illustrates the direction codes.

![Vector direction codes](image)

**Vector direction codes**

All the vectors in the preceding figure were drawn with the same length specification. Diagonal vectors stretch to match the X or Y displacement of the closest orthogonal vector. This is similar to the action of the snap grid in AutoCAD.

The following example constructs a shape named DBOX with an arbitrarily assigned shape number of 230.

```
*230,6,DBOX
014,010,01C,018,012,0
```

The preceding sequence of specification bytes defines a box one unit high by one unit wide, with a diagonal line running from the lower-left corner to the upper-right corner. After saving the file as *dbox.shp*, use the COMPILE command to generate the *dbox.shx* file. Use the LOAD command to load the
shape file containing this definition, and then use the SHAPE command as follows:

Command:  `shape`
Enter shape name or [?]:  `dbox`
Specify insertion point:  `1,1`
Specify height `<current>`:  `2`
Specify rotation angle `<current>`:  `0`

The resulting shape is shown in the following illustration.

![Diagram](image)

**Special Codes**

In addition to defining vectors, a specification byte can use the following special codes to create additional forms and specify certain actions. To use a special code, the second character of the three-character string (the vector length specification) must be 0, or you can specify only the code number. For example, 008 and 8 are both valid specifications.

<table>
<thead>
<tr>
<th>Specification byte codes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Code</strong></td>
</tr>
<tr>
<td>000</td>
</tr>
<tr>
<td>001</td>
</tr>
<tr>
<td>002</td>
</tr>
<tr>
<td>003</td>
</tr>
<tr>
<td>004</td>
</tr>
<tr>
<td>005</td>
</tr>
<tr>
<td>006</td>
</tr>
<tr>
<td>007</td>
</tr>
</tbody>
</table>
**Specification byte codes (continued)**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>008</td>
<td>$X$-$Y$ displacement given by next two bytes</td>
</tr>
<tr>
<td>009</td>
<td>Multiple $X$-$Y$ displacements, terminated (0,0)</td>
</tr>
<tr>
<td>00A</td>
<td>Octant arc defined by next two bytes</td>
</tr>
<tr>
<td>00B</td>
<td>Fractional arc defined by next five bytes</td>
</tr>
<tr>
<td>00C</td>
<td>Arc defined by $X$-$Y$ displacement and bulge</td>
</tr>
<tr>
<td>00D</td>
<td>Multiple bulge-specified arcs</td>
</tr>
<tr>
<td>00E</td>
<td>Process next command only if vertical text</td>
</tr>
</tbody>
</table>

**Code 0: End of Shape**

Code 0 marks the end of the shape definition.

**Codes 1 and 2: Draw Mode Control**

Codes 1 and 2 control Draw mode. Draw is activated at the start of each shape. When Draw mode is turned on, the vectors cause lines to be drawn. When Draw mode is turned off, the vectors move to a new location without drawing.

**Codes 3 and 4: Size Control**

Codes 3 and 4 control the relative size of each vector. The height specified with the SHAPE command is initially considered the length of a single orthogonal vector (directions 0, 4, 8, or C). Codes 3 and 4 are followed by a specification byte containing an integer scale factor (1 through 255). If you want the shape height to specify the size of the entire shape, and you use 10 vector lengths to draw it, you can use 3,10 to scale the height specification. The scale factor is cumulative within a shape; that is, multiplying by 2 and again by 6 results in a scale factor of 12. Usually, you should reverse the effect of your scale factors at the end of the shape, especially for subshapes and text font shapes. AutoCAD does not reset the scale factor for you.

**Codes 5 and 6: Location Save/Restore**

Codes 5 and 6 push (save) and pop (restore) the current coordinate position while drawing a shape so that you can return to it from a later point in the shape. You must pop everything you push. The position stack is only four locations deep. If the stack overflows because of too many pushes or too
many missing pops, the following message is displayed when the shape is
drawn.

Position stack overflow in shape nnn

Similarly, if you try to pop more locations than have been pushed onto the
stack, the following message is displayed when the shape is drawn.

Position stack underflow in shape nnn

Code 7: Subshape
Code 7 is a subshape reference. For a non-Unicode font the specification byte
following code 7 is a shape number from 1 to 255. For a Unicode font, code
7 is followed by a Unicode shape number from 1 to 65535. Unicode shape
numbers should be counted as two bytes (for specific information on the dif-
fferences between Unicode and non-Unicode fonts, see “Format for Unicode
Shape Definition Files” on page 62). The shape with that number (in the
same shape file) is drawn at this time. Draw mode is not reset for the new
shape. When the subshape is complete, drawing the current shape resumes.

Codes 8 and 9: X-Y Displacements
Normal vector specification bytes draw only in the 16 predefined directions,
and the longest length is 15. These restrictions help make shape definitions
efficient but are sometimes limiting. With codes 8 and 9 you can draw non-
standard vectors using X-Y displacements. Code 8 must be followed by two
specification bytes in the format:

\[ 8, \text{X-displacement}, \text{Y-displacement} \]

The X-Y displacements can range from \(-128\) to \(+127\). A leading + is optional,
and you can use parentheses to improve readability. The following example
results in a vector that draws (or moves) 10 units to the left and three units up.

\[ 8,(-10,3) \]

Following the two displacement specification bytes, the shape returns to
Normal Vector mode.

You can use code 9 to draw a sequence of nonstandard vectors. Follow this
by any number of X-Y displacement pairs. The code sequence is terminated
by a (0,0) pair. The following example draws three nonstandard vectors and
returns to Normal Vector mode.

\[ 9,(3,1),(3,2),(2,-3),(0,0) \]

You must terminate the sequence of X-Y displacement pairs with a (0,0) pair
in order for AutoCAD to recognize any Normal Vectors or special codes that
follow.
Code 00A: Octant Arc

Special code 00A (or 10) uses the next two specification bytes to define an arc. This is called an octant arc because it spans one or more 45-degree octants, starting and ending on an octant boundary. Octants are numbered counterclockwise from the 3 o’clock position, as shown in the following illustration.

![Octant Arc Diagram](image)

The arc specification is

\[10, \text{radius}, (–)0SC\]

The radius can be any value from 1 through 255. The second specification byte indicates the direction of the arc (counterclockwise if positive, and clockwise if negative), its starting octant (\(s\), a value from 0 through 7), and the number of octants it spans (\(c\), a value from 0 through 7, in which 0 equals eight octants, or a full circle). You can use parentheses to improve readability. For example, consider the following fragment of a shape definition:

\[...012,10,(1,–032),01E,...\]

This code draws a one-unit vector up and to the right, a clockwise arc from octant 3 (with a radius of one unit for two octants), and then a one-unit vector down and to the right, as shown in the following illustration.

![Octant Arc Example](image)

Code 00B: Fractional Arc

Special code 00B (11) draws an arc that doesn’t necessarily start and end on an octant boundary. The definition uses five specification bytes.

\[11, \text{start_offset}, \text{end_offset}, \text{high_radius}, \text{radius}, (–)0SC\]
The `start_offset` and `end_offset` represent how far from an octant boundary the arc begins or ends. The `high_radius` represents the most significant eight bits of the radius; the high radius will be 0 unless the `radius` is greater than 255 units. Multiply the `high_radius` value by 256 and add that value to the `radius` value to generate an arc radius greater than 255. The `radius` and ending specification byte are the same as for the octant arc specification (code 00A, described previously).

You determine the `start_offset` by calculating the difference in degrees between the starting octant’s boundary (a multiple of 45 degrees) and the start of the arc. Then, you multiply this difference by 256 and divide by 45. If the arc starts on an octant boundary, its `start_offset` is 0.

The `end_offset` is calculated in a similar fashion, but you use the number of degrees from the last octant boundary crossed to the end of the arc. If the arc ends on an octant boundary, its `end_offset` is 0.

For example, a fractional arc from 55 degrees to 95 degrees with a 3 unit radius would be coded as follows:

```
11,(56,28,0,3,012)
```

Here is the explanation:

- `start_offset` = 56 because \(((55 - 45) \times 256) \div 45) = 56\)
- `end_offset` = 28 because \(((95 - 90) \times 256) \div 45) = 28\)
- `high_radius` = 0 because \((radius < 255)\)
- `radius` = 3
- `starting octant` = 1 because arc starts in the 45 degree octant
- `ending octant` = 2 because arc ends in the 90 degree octant

**Codes 00C and 00D: Bulge-Specified Arcs**

Special codes 00C and 00D (12 and 13) provide another mechanism for including arc segments in shape descriptions. They are similar to codes 8 and 9 in that you can use them to specify X-Y displacements. However, codes 00C and 00D let you draw arcs by applying a bulge factor to the displacement vector. Code 00C draws one arc segment, while code 00D draws multiple arc segments (polyarcs) until it is terminated by a (0,0) displacement.

Code 00C must be followed by three bytes describing the arc:

```
0C,X-displacement,Y-displacement,Bulge
```

Both the X and Y displacement and the bulge, which specifies the curvature of the arc, can range from –127 to +127. If the line segment specified by the displacement has length \(D\), and the perpendicular distance from the midpoint of that segment has height \(H\), the magnitude of the bulge is \((2 \times H \div D) \times 127\). The sign is negative if the arc from the current location to the new location is clockwise.
A semicircle has bulge 127 (or −127) and is the greatest arc that can be represented as a single-arc segment using these codes (use two consecutive arc segments for larger arcs). A bulge specification of 0 is valid and represents a straight-line segment. Note, however, that using code 8 for a straight-line segment saves a byte in the shape description.

The polyarc code (00D, or 13) is followed by 0 or by more arc segment triples, and is terminated by a (0,0) displacement. Note that no bulge is specified after the final displacement. For example, the letter S might be defined by the following sequence:

13, (0,5,127), (0,5,-127), (0,0)

Zero bulge segments are useful within polyarcs to represent straight segments; they are more efficient than terminating the polyarc, inserting one straight segment, and then starting another polyarc.

The number −128 cannot be used in arc segment and polyarc definitions.

**Code 00E: Flag Vertical Text Command**

Special code 00E (14) is used only in dual-orientation text font descriptions, where the font is used in both horizontal and vertical orientations. When this special code is encountered in a character definition, the next code is either processed or skipped, depending on orientation. If the orientation is vertical, the next code is processed; if it is horizontal, the next code is skipped.

In horizontal text, the start point for each character is the left end of the baseline. In vertical text, the start point is assumed to be the top center of the character. At the end of each character, a pen-up segment is normally drawn to position to the next character’s start point. For horizontal text, it is to the right; for vertical text, it is downward. The special 00E (14) code is used primarily to adjust for differences in start points and endpoints, permitting the same character shape definition to be used both horizontally and vertically.
For instance, the following definition of an uppercase D could be used in either horizontal or vertical text.

```
*68,22,ucd
2,14,8,(-2, 6),1,030,012,044,016,038,2,010,1,06C,2,050,
14,8,(-4, -3),0
```

![Horizontal and vertical D shapes]

**Text Font Descriptions**

AutoCAD is packaged with numerous text fonts. You can use the `STYLE` command to apply expansion, compression, or obliquing to any of these fonts, thereby tailoring the characters to your needs. You can draw text of any height, at any baseline angle, and with either horizontal or vertical orientation using these fonts.

If you intend to create your own text fonts, study the `txt.shp` file and the other fonts that are packaged with AutoCAD. They provide working examples of the topics discussed here. The `txt.shp` file is available on the AutoCAD CD. It is also available at “Sample Files” in the online Customization Guide.

AutoCAD text fonts are files of shape definitions with shape numbers corresponding to the ASCII code for each character. For a list of the ASCII codes, see appendix A, “ASCII Codes.”
Codes 1 through 31 are for control characters, only one of which is used in AutoCAD text fonts:

10 (LF) The line feed (LF) must drop down one line without drawing. This is used for repeated TEXT commands, to place succeeding lines below the first one.

*10,5,lf
2,8,(0,-10),0

You can modify the spacing of lines by adjusting the downward movement specified by the LF shape definition.

Text fonts must include a special shape number 0 that conveys information about the font itself. The format has the following syntax:

*0,4,font-name
above,below,modes,0

The above value specifies the number of vector lengths above the baseline that the uppercase letters extend, and below indicates how far the lowercase letters descend below the baseline. The baseline is similar in concept to the lines on writing paper. These values define the basic character size and are used as scale factors for the height specified in the TEXT command.

The modes byte should be 0 for a horizontally oriented font and 2 for a dual-orientation (horizontal or vertical) font. The special 00E (14) command code is honored only when modes is set to 2.

The standard fonts supplied with AutoCAD include a few additional characters required for the AutoCAD dimensioning feature.

%%d - Degree symbol (°)
%%p - Plus/minus tolerance symbol (±)
%%c - Circle diameter dimensioning symbol (⌀)

You can use these and the %%%nnn control sequences, as described under TEXT in the Command Reference.

NOTE AutoCAD draws text characters by their ASCII codes (shape numbers) and not by name. To save memory, specify the shape name portion of each text shape definition in lowercase as shown in the following example. (Lowercase names are not saved in memory.)

*65,11,uca
024,043,04d,02c,2,047,1,040,2,02e,0
Because the shape name \texttt{uca} contains lowercase letters, AutoCAD doesn’t save the name in memory. However, you can use the name for reference when editing the font definition file. In this example, \texttt{uca} stands for uppercase A.

**Big Font Descriptions**

Some languages, such as Japanese, use text fonts with thousands of non-ASCII characters. In order for drawings to contain such text, AutoCAD supports a special form of shape definition file called a *Big Font* file.

**Defining a Big Font**

A font with hundreds or thousands of characters must be handled differently from a font containing the ASCII set of up to 256 characters. In addition to using more complicated techniques for searching the file, AutoCAD needs a way to represent characters with two-byte codes as well as one-byte codes. Both situations are addressed by the use of special codes at the beginning of a Big Font file.

The first line of a Big Font shape definition file must be as follows:

\*BIGFONT \texttt{nchars,nranges,b1,e1,b2,e2,…}

where \texttt{nchars} is the approximate number of character definitions in this set; if it is off by more than about 10 percent, either speed or file size suffers. You can use the rest of the line to name special character codes (escape codes) that signify the start of a two-byte code. For example, on Japanese computers, Kanji characters start with hexadecimal codes in the range 90-AF or E0-FF. When the operating system sees one of these codes, it reads the next byte and combines the two bytes into a code for one Kanji character. In the \*BIGFONT line, \texttt{nranges} tells how many contiguous ranges of numbers are used as escape codes; \texttt{b1, e1, b2, e2}, and so on, define the beginning and ending codes in each range. Therefore, the header for a Japanese Big Font file might look like this:

\*BIGFONT \texttt{4000,2,090,0AF,0E0,0FF}

After the \*BIGFONT line, the font definition is just like a regular AutoCAD text font, except that character codes (shape numbers) can have values up to 65535.

**Defining an Extended Big Font File**

To reduce the size of composite Kanji characters, you can define an extended Big Font file. Extended big fonts use the subshape code, followed immediately by a 0.
The first line of an extended Big Font file is the same as the regular Big Font file. This is the format for the remaining lines of the file:

*0,5,font-name
character-height, 0, modes, character-width, 0
...
*shape-number, defbytes, shape-name
code, 0, primitive#, basepoint-x, basepoint-y, width, height,
...
code, 0, primitive#, basepoint-x, basepoint-y, width, height,
terminator

The following list describes the fields of a Big Font definition file:

- **character height**: Used along with character width to indicate the number of units that define the font characters.
- **character width**: Used along with character height to indicate the number of units that define the font characters. The character-height and character-width values are used to scale the primitives of the font. In this context, primitives are the points, lines, polygons, or character strings of the font geometrically oriented in two-dimensional space. A Kanji character consists of several primitives used repeatedly in different scales and combinations.
- **modes**: The modes byte should be 0 for a horizontally oriented font and 2 for a dual-orientation (horizontal or vertical) font. The special 00E (14) command code is honored only when modes is set to 2.
- **shape-number**: Character code.
- **defbytes**: Byte size. It is always 2 bytes, consisting of a hexadecimal or a combination of decimal and hexadecimal codes.
- **shape-name**: Character name.
- **code**: Shape description special code. It is always 7 so that it can use the subshape feature.
- **primitive#**: Reference to the subshape number. It is always 2.
- **basepoint-x**: X origin of the primitive.
- **basepoint-y**: Y origin of the primitive.
- **width**: Scale of the width of the primitive.
height Scale of the height of the primitive.

terminator End-of-file indicator for the shape definition. It is always 0.

To arrive at the scale factor, AutoCAD scales down the primitive to a square unit and then multiplies it by the height and width to get the shape of the character. Character codes (shape numbers) in the Big Font shape definition file can have values up to 65535. The following table describes the fields of the extended Big Font file.

### Fields of the extended Big Font file

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Byte size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shape-number</td>
<td>xxxx</td>
<td>2 bytes</td>
<td>Character code</td>
</tr>
<tr>
<td>code</td>
<td>7,0</td>
<td>2 bytes</td>
<td>Extended font definition</td>
</tr>
<tr>
<td>primitive‡</td>
<td>xxxx</td>
<td>2 bytes</td>
<td>Refer to subshape number</td>
</tr>
<tr>
<td>basepoint-x</td>
<td>1 byte</td>
<td></td>
<td>Primitive X origin</td>
</tr>
<tr>
<td>basepoint-y</td>
<td>1 byte</td>
<td></td>
<td>Primitive Y origin</td>
</tr>
<tr>
<td>width</td>
<td>1 byte</td>
<td></td>
<td>Scale of primitive width</td>
</tr>
<tr>
<td>height</td>
<td>1 byte</td>
<td></td>
<td>Scale of primitive height</td>
</tr>
<tr>
<td>terminator</td>
<td>0</td>
<td>1 byte</td>
<td>End of shape definition</td>
</tr>
</tbody>
</table>

The following figure is an example of a $16 \times 16$ dot matrix that you could use to design an extended Big Font, such as a Kanji character. In the example, the distance between each dot is one unit. The callout points to a square unit.
The following figure shows examples of Kanji characters. Each character occupies an $M \times N$ matrix, (matrices don’t have to be square) similar to the one shown in the previous figure. The numbers above each figure are the associated shape numbers.

Examples of Kanji characters
The following figure shows Kanji primitives.

89A4  8BCA  8BE0  8C8E
王 玉 金 月

8CB3  8CC3  8CFB
元 古 □

Examples of Kanji primitives

**NOTE** Not all fonts are defined in a square matrix; some are defined in rectangular matrices.

The following is an example of a shape definition file for an extended Big Font.

*BIGFONT 50,1,080,09e
*0,5,Extended Font
15,0,2,15,0
*08D91,31,unspecified
2,0e,8,-7,-15,
7,0,08cfb,0,0,16,16,7,0,08bca,2,3,12,9,
2,8,18,0,2,0e,8,-11,-3,0
*08CD8,31,unspecified
2,0e,8,-7,-15,
7,0,08be0,0,0,8,16,7,0,08cc3,8,0,8,16,
2,8,18,0,2,0e,8,-11,-3,0
*08ADF,31,unspecified
2,0e,8,-7,-15,
7,0,089a4,0,0,8,16,7,0,08cb3,8,0,8,16,
2,8,18,0,2,0e,8,-11,-3,0
*08CE8,39,unspecified
2,0e,8,-7,-15,
7,0,089a4,0,1,5,14,7,0,08cc3,5,2,5,14,7,0,08c8e,9,0,7,
16,2,8,18,0,2,0e,8,-11,-3,0
*089A4,39,primitive
2,0e,8,-7,-15,2,8,1,14,1,0c0,
2,8,-11,-6,1,0a0,2,8,-12,-7,1,
0e0,2,8,-7,13,1,0dc,2,8,11,-1,
2,0e,8,-11,-3,0
Using a Big Font

To use a Big Font for drawing text, you must set up a text style by using the STYLE command and then specify the name of the Big Font file. The same text style can use a normal ASCII font as well; enter only the two file names, separated by a comma. The following example uses the command line version of the STYLE command. To enable Big Fonts from the Text Style dialog box, choose the Use Big Font option.

Command: –style
Enter name of text style or [?] <current>: style_name
Specify full font name or font file name (TTF or SHX): txt,greek

AutoCAD assumes that the first name is the normal font and that the second is the big font.

If you enter only one name, AutoCAD assumes it is the normal font and removes any associated Big Font.
By using leading or trailing commas when specifying the font file names, you can change one font without affecting the other, as shown in the following table.

### Input for changing fonts

<table>
<thead>
<tr>
<th>Input</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal, big</td>
<td>Both normal and Big Font specified</td>
</tr>
<tr>
<td>normal,</td>
<td>Normal font only (Big Font unchanged)</td>
</tr>
<tr>
<td>,big</td>
<td>Big Font only (normal font unchanged)</td>
</tr>
<tr>
<td>normal</td>
<td>Normal font only (if necessary, Big Font removed)</td>
</tr>
<tr>
<td>ENTER (null response)</td>
<td>No change</td>
</tr>
</tbody>
</table>

When you use the STYLE command to list styles or to revise an existing style, AutoCAD displays the normal font file, a comma, and the Big Font file. If the style has only a Big Font file, it is displayed with a leading comma, as in ,greek.

For each character in a text string, AutoCAD searches the Big Font file first. If the character is not found there, the normal font file is searched.

**Alternate Uses for a Big Font**

In some drafting disciplines, many special symbols can appear in text strings. The AutoCAD standard text fonts can be extended to include special symbols. However, extending standard text fonts has several limitations:

- The number of shapes is 255 per font file.
- Standard character set uses almost half the available shape numbers. Only codes 1 through 9, 11 through 31, and 130 through 255 are available.
- Multiple text fonts require duplication of the symbol definitions in each font.
- Special symbols require that you enter `%%nnn`, where `nnn` is the symbol’s shape number.

The Big Font mechanism avoids these problems. You can select one or more seldom-used characters, such as the tilde (˜) or the vertical bar(│), as escape codes, and use the next character to select the appropriate special symbol. For instance, you can use the following Big Font file to draw Greek letters by entering a vertical bar (│, ASCII code 124) followed by the equivalent Roman
letter. Because the first byte of each character is 124, the character codes are biased by $124 \times 256$, or 31744.

\*BIGFONT 60,1,124,124
\*0,4,Greek
above, below, modes, 0
\*31809,n,uca
  . . . uppercase Alpha definition, invoked by "|A"
\*31810,n,ueb
  . . . uppercase Beta definition, invoked by "|B"
\*31841,n,lca
  . . . lowercase Alpha definition, invoked by "|a"
\*31842,n,lcb
  . . . lowercase Beta definition, invoked by "|b"
\*31868,n,vbar
  . . . vertical bar definition, invoked by "||"

**Unicode Font Descriptions**

The standard AutoCAD fonts correspond to the character mapping used by the host operating system. This is because characters are stored directly in the database in the format in which they are obtained from the keyboard. The same character codes are used to generate fonts. This becomes a problem when using accented (8-bit) characters for which many character encoding standards exist.

Due to character mapping limitations, AutoCAD provides a set of Unicode fonts for the various code pages that AutoCAD uses. These fonts, while essentially the same, have some characters located in different places, depending on the code page they are defined for. If the font encoding used does not match that of the text in the drawing, the wrong characters may be drawn.

With Unicode fonts, text strings are converted to Unicode before being drawn, so it is no longer necessary to provide additional fonts for other languages or platforms. A single Unicode font, due to its large character set, is capable of supporting all languages and platforms. This feature is transparent to the user because the drawings are, if needed (due to differing code pages), converted to the AutoCAD system code page at load time. The drawings are always saved in the AutoCAD system code page.

**NOTE** Unicode does not provide adequate support for all Asian languages, so Big Fonts are still used by some or all Asian versions.
Format for Unicode Shape Definition Files

Unicode shape definition files are virtually identical in format and syntax to regular AutoCAD shape definition files. The main difference is in the syntax of the font header as shown in the following code:

*UNIFONT,6,font-name
above,below,modes,encoding,type,0

The font-name, above, below, and modes parameters are the same as in regular fonts. The remaining two parameters are defined as follows:

coding Font encoding. Uses one of the following integer values.
0 Unicode
1 Packed multibyte 1
2 Shape file
type Font embedding information. Specifies whether the font is licensed. Licensed fonts must not be modified or exchanged. Bitcoded values can be added.
0 Font can be embedded
1 Font cannot be embedded
2 Embedding is read-only

Another important, and sometimes confusing, difference is the handling of the code 7 subshape reference. If a shape description includes a code 7 subshape reference, the data following the code 7 is interpreted as a 2-byte value. This affects the total number of data bytes (defbytes) in the shape description header. For example, the following shape description is found in the romans.shp file:

*00080,4,keuroRef
7,020AC,0

The second field in the header represents the total number of bytes in the shape description. If you are not used to working with Unicode font descriptions, you may be inclined to use 3 bytes rather than 4, but this would cause an error during the compiling of the SHP file. This is true even if the shape number you are referencing is not in the two-byte range (below 255); the compiler always uses two bytes for this value, so you must account for that in the header.

The only other difference between Unifont shape definitions and regular shape definitions is the shape numbers. The Unifont shape definitions that AutoCAD provides use hexadecimal shape numbers as opposed to decimal values. Although hexadecimal numbers are not required their use makes it easier to cross-reference the shape numbers with the \U+ control character values.
Advanced Shape Definition Technique

The AutoCAD SHX fonts have limited superscript and subscript capabilities. However, it is relatively easy to modify shape definition files to improve superscript and subscript capability.

Creating superscripts and subscripts requires two steps. First, the “imaginary pen” that is creating the text, vector by vector, on your screen needs to be shifted up or down. Then, the font “scale” needs to be reduced. In addition, the reverse process has to take place to return to the normal font. The font needs to recognize four new keys: two for superscripts and two for subscripts. To avoid altering the existing font definitions, you can access these with the numeric keypad on your keyboard.

The following example procedure is based on the AutoCAD Romans font file, although a similar method applies to any AutoCAD font. This procedure adds four new shape definitions to a font: super_on, super_off, sub_on, and sub_off, which control the position and size of the characters that follow. For simplicity, this example replaces the left- and right-bracket characters ([ and ]) and the left and right curly brace characters ({ and }) with the new characters. You may choose to replace other characters or use a shape number in the extended range (ASCII codes 128 through 256). If you use an extended shape number, you need to use the \%%nnn method (where nnn is the ASCII value of the character) for placing the new characters.

To add superscript and subscript definitions to a font

1 Edit the SHP (in this case, romans.shp) file with an ASCII text editor. You can create a new file called romanss.shp rather than modify the original file. The romans.shp file is available on the AutoCAD 2000 CD. It is also available at “Sample Files” in the online Customization Guide.

2 Search for the shape definitions of the characters you are replacing. To comment out those definitions so the new definitions can take their place, insert a semicolon in front of each line of the shape definition. The shape definition may continue for a number of lines.

The left- and right-bracket characters have ASCII values of 91 and 93 (05B and 05D hex values, if the font is Unicode). The left and right curly brace characters have ASCII values of 123 and 125 (07B and 07D hex).

3 Add the first and second values on the second line of the definition, and divide the total by 2 as shown in the following example:

*UNIFONT,6,Extended Simplex Roman for UNICODE
21,7,2,0 21 + 7 = 28, then 28 / 2 = 14. This number is used later.
4 Add the following lines to the end of the SHP file:

```plaintext
*91,8,super_on
2,8,(0,14),003,2,1,0
*93,8,super_off
2,004,2,8,(0,-14),1,0
*123,8,sub_on
2,8,(0,-14),003,2,1,0
*125,8,sub_off
2,004,2,8,(0,14),1,0
```

Notice the 14 and –14 values in the preceding lines. They are Y axis offsets for the imaginary pen. The value 14 is half the maximum height of a character in this font, which is the correct approximation for superscripts and subscripts. This value needs to be calculated for each font file, but you can modify it any way you want.

5 Save the file.

6 Use the COMPILE command to compile the SHP file.

Once the shape is compiled and an appropriate style is defined, you can access the new pen-up and pen-down commands by entering the [, ], {, and } characters. The [ character initiates superscript and the ] character returns from superscript to normal. The { character initiates subscript and the } character returns from subscript to normal.

**PostScript Support**

With the PostScript support features of AutoCAD, you can use PostScript fonts with AutoCAD drawings. You can also export AutoCAD drawings as Encapsulated PostScript (EPS) files, import EPS files into AutoCAD drawings, and apply PostScript fills to 2D polylines.

**Exporting PostScript Images**

You use the PSOUT command to export the current view of a drawing as an EPS file. You can then insert the file into a desktop publishing, illustration, or presentation program.

The following sections describe the structure and purpose of the PSOUT master support file, `acad.psf`. These sections also provide information on advanced PostScript output processing.

**`acad.psf`—The AutoCAD PostScript Support File**

The AutoCAD PostScript Support file (`acad.psf`) is the master support file for AutoCAD PSOUT and PSFILL commands. This ASCII file can be edited and is
divided into sections that control font substitution and define PostScript fill patterns. The file also includes prolog sections that consist of PostScript-coded procedure definitions and constants. The sections of acad.psf can appear in any order, but for reasons of efficiency the *fonts section should appear first in the file.

Each section of acad.psf is introduced by a statement with an asterisk in column 1 followed by the section name. Lines beginning with a semicolon are treated as comments in all sections of this file.

The standard acad.psf file defines the Type 1 fonts formerly supplied with AutoCAD, as well as other common PostScript fonts. A font downloader is not supplied with AutoCAD, so you might want to modify acad.psf to reference fonts that are resident in your printer. The following sections describe the sections of acad.psf.

*fonts—Font Substitution Map

Each line in the Fonts section lists an AutoCAD font name (with either an .shx or .pfb file name extension), and the name of a PostScript font to substitute for text objects that use that font when you use the PSOUT command. PostScript font names defined by Adobe are case-sensitive; the AutoCAD font names are not. Verification is not performed on the PostScript font names. If a required font isn’t available when you print, most PostScript implementations issue a warning message and substitute a default font, such as Courier.

The following sample from acad.psf shows AutoCAD shape/font files mapped to a PostScript font. For example, agd.shx maps to the PostScript font, AvantGarde-Demi.

*fonts

; agd       AvantGarde-Demi
agdo      AvantGarde-DemiOblique
agw       AvantGarde-Book
agwo      AvantGarde-BookOblique
bdps      Bodoni-Poster
bkd       Bookman-Demi
bkdi      Bookman-DemiItalic
bkl       Bookman-Light
bklili     Bookman-LightItalic
c         Cottonwood
cibt      City-Blueprint
cob       Courier-Bold
cobo      Courier-BoldOblique
com       Courier
.
.
The first column gives the name by which AutoCAD knows a font; the second column gives the official PostScript name for the font. Trailing
underscores should be ignored. For example, if the file containing the font AvantGarde-Demi is called \textit{agd\_\_\_\_\_\_\_\_\_.pfb}, the entry in the first column is \textit{agd}.

\textbf{*figureprologue—PostScript Figure Prolog}

The \textit{*figureprologue} section defines the procedures to embed PostScript figures included with PSIN in Encapsulated PostScript files (EPS) created by \texttt{PSOUT}. By keeping this code in \texttt{acad.psf}, you can modify it as required to accommodate unusual PostScript files. This code is written at the point the first PSIN image is encountered.

\textbf{*isofontprologue—ISO 8859 Latin/1 Font Re-encoding}

The \textit{*isofontprologue} section defines the procedures to re-encode a standard PostScript font to be compatible with the ISO 8859 Latin/1 character set. This code is written when an ISO character is first encountered in a text object.

\textbf{*fillprologue—PostScript Polyline Fill Prolog}

The \textit{*fillprologue} section defines the code transcribed to the output when \texttt{PSOUT} encounters the first \texttt{PSFILL} request. It is used to fill polylines, and it defines the following fill procedures:

\begin{description}
  \item[Rangefilter] Called to range-check predefined and user-defined fill patterns.

  Any fill pattern invoked with parameters can use \texttt{Range\texttt{filter}} to check that the parameters are within a valid range for the pattern. The syntax is

  \begin{verbatim}
  value min max Rangefilter result
  \end{verbatim}

  If \texttt{value} is within the inclusive range from the minimum limit (\texttt{min}) to the maximum limit (\texttt{max}) then \texttt{value} becomes the result. Otherwise, the result is the limit closest to \texttt{value}.

  For example, consider the following fill pattern for \texttt{Grayscale}:

  \begin{verbatim}
  @@Fill
  /Grayscale %Grayscale,1, Grayscale=50
  { 0 100 Rangefilter 100 div 1 exch sub
      setgray fill
  } bind def
  \end{verbatim}

  Here, \texttt{Rangefilter} is invoked immediately when the stack argument is passed to the \texttt{Grayscale} function. If that
value is out of range, Rangefilter sets it to the closest limit, either min (0) or max (100) in the example.

You can also use Rangefilter within the fill pattern to calculate the bounds dynamically for the fill pattern.

PreFill Called after establishing the transformation used to render the polyline. It saves the transformation in a variable called CMatrix.

DoFill Called immediately after the path to fill has been completed; thus the polyline’s path, rendered into PostScript, is the current path. It saves the bounding box in the standard BbXxyy variables, and then establishes the standard transformation used by the fill procedures.

EndFill Called after invoking the doFill procedure. It can do any cleanup required, but at the moment does nothing.

DoOutline Called immediately after EndFill. It generates the outline of the polyline, whose path becomes the current path. The PostScript line width is set to the width of the Polyline and scaled to PostScript imaging space.

To omit the polyline outline, precede the name of the fill pattern with an asterisk (*).

*Fill—Defining Your Own PostScript Fill Patterns
The *fill section defines the PostScript fill patterns. All fill patterns are described between the *fill statement and the next line with an asterisk in column 1. To add a new PostScript fill pattern, edit this section of acad.psf.

Each fill pattern is introduced by this line starting in column 1:

```plaintext
@Fill
```

Immediately following this line, in a precisely defined format, is the first line of PostScript procedure that defines the pattern. The following example defines the RGBcolor pattern:

```plaintext
@Fill
/RGBcolor %RGBcolor,3, Red=50, Green=50, Blue=50,
{
 /Blue exch 0 100 Rangefilter def
 /Green exch 0 100 Rangefilter def
 /Red exch 0 100 Rangefilter def

  Red 100 div 1 exch sub
  Green 100 div 1 exch sub
  Blue 100 div 1 exch sub
  setrgbcolor fill
} bind def
```
This is how the first line of the procedure is written:

```
/proc_name %pattern_name,nargs, [argn=defaultn]
```

The variables are defined as follows:

- **proc_name**: Internal name of this PostScript fill procedure. The name should be unique; it can’t duplicate any other fill procedure in the file or any PostScript name in general.
- **pattern_name**: Name of the pattern you enter for PSFILL.
- **nargs**: Integer giving the number of arguments to the pattern, from 0 to 25.
- **argn**: Name of the \( n \)th argument, which appears at the prompt when the pattern is applied with PSFILL.
- **defaultn**: Default value for the \( n \)th argument when the pattern is used initially. Arguments have integer or real values, and you must specify a default.

When AutoCAD encounters a PostScript fill pattern in a drawing exported with PSOUT, its definition is located in `acad.psf`, and then copied into the output Encapsulated PostScript file (EPS). Every line starting with a forward slash (`/`) (like `/RGBcolor` in the previous example) up to the next line starting with `"%@"` (or end of section or file) is transcribed literally into the output PostScript file.

You call the fill pattern function with the outline of the polyline as the current PostScript path. The PostScript transformation is configured in the following reliable state:

- Rendering transformation is set so that the default units used in the fill function are thousandths of an inch.
- Variables `Bbllx`, `Bblly`, `Bburx`, and `Bbury` give the extents of the bounding box of the path in units of 1/1000 inch.
- Commands to save and restore (`gsave` and `grestore`) the fill procedure guarantee that its changes to the graphics state do not affect subsequent output of objects.
- Arguments to the fill procedure appear on the operand stack in the order they were declared in `acad.psf`. The last argument is on the top of the stack, the second to last argument is second to last on the stack, and so on. The fill procedure removes all the arguments from the stack.

With the first PostScript fill procedure, the contents of the `*fillprologue` section of `acad.psf` are transcribed to the output EPS file. This section defines procedures that can be used by any fill procedure.
PSPROLOG—Creating a Custom Prolog Section

The PSPROLOG system variable instructs PSOUT to reference an additional prolog section that you create in the acad.psf file. By creating a prolog section and placing PostScript code in it, you can customize the appearance of PostScript output in a number of ways. This is useful if you want to perform output functions such as assigning different line widths for different colors or creating special linetypes with the PostScript setdash function. A single prolog section can contain many PostScript functions.

To create a custom prolog section, add a section header to acad.psf with a text editor or word processor. A section header is a line of text preceded by an asterisk.

*lineweight

In this example, all lines without comments between *lineweight and the next acad.psf section are written to the PostScript file.

You can add as many prolog sections as you wish to acad.psf (one for each of your printing devices, for instance); however, only one is referenced during PSOUT. Use the PSPROLOG system variable to determine which one to reference.

PSPROLOG accepts a string as a value. That string is the name of the prolog section to be referenced. The following example references the *lineweight prolog created earlier.

Command: psprolog
Enter new value for PSPROLOG, or . for none <default>: lineweight

A commented example prolog, called *sampleprolog, is included in acad.psf.

Advanced PostScript Processing

The PSOUT command encodes structure from the AutoCAD drawing into the PostScript files it generates. Sophisticated PostScript-based applications can use this information to perform special processing based on properties of the original AutoCAD objects.

Layer Changes
Every layer change is flagged by a call to

(layer_name) ACADLayer

You can redefine this to take any action you want based on giving the layer name as its argument.

Color Changes
Color changes to display AutoCAD objects are signaled by a call to

color_no red green blue ACADColor
The variables are defined as follows:

- **color_no**: AutoCAD color number
- **red**, **green**, **blue**: Primary color components (from 0 through 1) of that color in the standard AutoCAD 256-color palette

By default, this call sets the PostScript output color with the `setrgbcolor` operator, but you can change it.

**Linetypes**
Changes in the effective linetype are signaled by a call to

```
(ltype_name) [dot/dash_array] (alignment) ACADLtype
```

The variables are defined as follows:

- **ltype_name**: Name of the linetype (CONTINUOUS, DASHED, and so on)
- **dot/dash_array**: Dash lengths of the pen-down and pen-up segments as specified in the AutoCAD line font definition
- **alignment**: Letter that gives the mode of alignment of the pattern to objects

You can use this call to emulate any PostScript line font; it’s triggered by the AutoCAD linetype properties passed as arguments. See the `*sampleprolog` section of *acad.psf* for sample routines that utilize the `ACADLayer`, `ACADColor`, and `ACADLtype` functions.

**Importing PostScript Images**

The following sections describe the PostScript features that enable AutoCAD to import PostScript images through the PSIN command.

**acadps.arx—The AutoCAD PostScript Interpreter**

The AutoCAD PostScript interpreter (`acadps.arx`) is an ObjectARX application that renders PostScript images into AutoCAD objects. When you insert PostScript graphics with the PSIN command, `acadps` is invoked to render the graphics according to the quality index you’ve set. You set the quality index with the PSQUALITY system variable, as described in the table of system variables in appendix B, “System Variables,” in the Command Reference. If the quality index is zero, `acadps` is not invoked, and PostScript objects appear only as their bounding rectangle and file name.

When the quality index is nonzero, PSIN loads `acadps` automatically. The `acadps` application does not remain in AutoCAD; to save memory, `acadps` reloads each time you invoke PSIN. Although this process is quite fast, you
can bypass it by loading *acadps* with *arxload*, adding *acadps* to *acad.rx*, or modifying *acad.lsp* to *arxload acadps*.

To work properly, *acadps* must be able to find its interpreter initialization files (*acadpsd.ps*, *acadpsf.ps*, *acadpsi.ps*, and *acadpss.ps*), font map file (*fontmap.ps*), font definitions, and fonts on the ACAD search path.

The *acadps* application is an implementation of the Ghostscript Interpreter from Aladdin Enterprises, distributed under the terms of the Ghostscript General Public License. A copy of this license agreement is available in the *copying.gs* file, in the AutoCAD *fonts* directory. Under these terms, you’re free to distribute copies of *acadps*. The source code for *acadps* is available directly from Autodesk. This application is distributed to you free of charge. Ghostscript is not a certified PostScript interpreter and is distributed by Autodesk with no warranty.

PostScript images and fills are stored in drawing files as extended data and are identified by the *AUTOCAD_POSTSCRIPT_FIGURE* application name.

**fontmap.ps—The AutoCAD Font Map File**

The AutoCAD Font Map file (*fontmap.ps*) is a catalog, or font map, of all fonts known to the AutoCAD PostScript Interpreter (*acadps*). During PSIN, this ASCII file maps PostScript language font names to the names of their respective font definition (PFB) files. Any font to be loaded automatically when named should be declared in *fontmap.ps*. You can place the fonts in any directory on the ACAD search path.

When you import an EPS file with PSIN, *acadps* uses *fontmap.ps* to locate PFB files corresponding to fonts named in the EPS file. If *acadps* finds a PFB file in *fontmap.ps* but not in your system, it displays a warning and substitutes a default font.

As shipped, *fontmap.ps* references a set of Type 1 fonts that were supplied with AutoCAD in Releases 12 and 13, as well as the fonts distributed with the Adobe Type Manager for Windows, Adobe Plus Pack, and Adobe Font Pack 1. If you’ve purchased additional fonts, you can edit the font map file and declare them the same way the existing fonts are declared.

Refer to the contents of *fontmap.ps* for additional information, including comments describing font substitution.

**Errors in Imported PostScript Images**

Errors in PostScript images imported by *PSIN* are handled as defined in the *PostScript Language Reference Manual*, also known as the “Red Book.” Fatal error messages result in abandoning the PSIN operation. Less serious error messages, such as references to unavailable fonts, result in font substitution.
If AutoCAD can’t render a valid PostScript image, and you still want to include it in your drawing, set the PostScript quality index to 0 and use PSIN to import the image. AutoCAD won’t render the image or perform any error checking. However, if the PostScript image does contain an error, any EPS file generated by PSOUT results in a file that generates an error when it’s finally printed.

You also have no guarantee that all PostScript files that work with an Adobe implementation of PostScript can be rendered correctly by the AutoCAD PostScript interpreter. The interpreter is compatible with Adobe PostScript, but subtle differences exist. If AutoCAD can’t load a valid PostScript illustration, you can set PSQUALITY to 0 and load the image anyway. It is represented in AutoCAD by its bounding box and file name.

**PostScript References**


*Ghostscript General Public License*, Aladdin Enterprises, 1989. A copy of this license agreement is available in the `copying.gs` file. This file is located in the AutoCAD fonts directory.


Custom Menus

If the standard menus in AutoCAD® do not contain the commands you use most often, you can customize them or add menus with the commands and macros you need. For example, if you frequently have several drawings open simultaneously, you can add the Previous Window command that returns you to the last drawing. You can also control the commands available on shortcut menus.
Using Menu Files

Menus are defined by menu files. You can modify menu files or create your own. By editing the text in a menu file, you can define the appearance and location of menu items. You can then assign menu macros that execute specific actions when a menu item is selected.

Menu macros can be simple recordings of keystrokes that accomplish a task, or they can be a complex combination of commands and AutoLISP® or DIESEL programming code. More complex menu macros have some decision-making ability. A menu macro is similar to a script in that it issues a series of commands. However, scripts have no decision-making ability and cannot pause for interaction to create custom menus to complement those supplied with AutoCAD.

Menu files define the functionality and appearance of menu areas, different parts of the AutoCAD interface that provide access to and feedback about the program’s functionality. The menu items in each menu area contain AutoCAD command strings and macro syntax that define the resulting action when the menu item is selected. The following areas are defined by menu files:

- Pointing-device button menus
- Pull-down and shortcut menus
- Toolbars
- Image tile menus
- Screen menus
- Digitizing-tablet menus
- Help strings and tooltips
- Keyboard accelerators

These menu areas are described later in this chapter.

Menu File Types

The term menu file actually refers to the group of files that work together to define and control the appearance and functionality of the menu areas. The following table describes the AutoCAD menu file types.
Bitmap Resource DLLs

Bitmap resource DLLs store the bitmaps used for toolbars, menus, and shortcut menus. For AutoCAD to access the bitmap resource, the following requirements must be met:

- The DLL file name is the same as the menu file name it's associated with.
- The resources are named rather than index numbered.
- The DLL file is located in the same directory as the menu file that uses it.

To use these resources in the menu, use the appropriate resource names in the `id_small` and `id_big` parameters for toolbar buttons. See “Toolbars” on page 107.

Loading Menu Files

Use the `MENU` command to load a new menu. Use the `MENULOAD` and `MENUUNLOAD` commands to load and unload additional menus (called partial menus) and to add or remove individual menus from the menu bar.

AutoCAD stores the name of the last loaded menu in the system registry. This name is also saved with the drawing, but it is used only for backward compatibility. When you start AutoCAD, the last menu used is loaded. As of Release 14, AutoCAD no longer reloads the menu between drawings.

### AutoCAD menu files

<table>
<thead>
<tr>
<th>File type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNU</td>
<td>Template menu file.</td>
</tr>
<tr>
<td>MNC</td>
<td>Compiled menu file. This binary file contains the command strings and menu syntax that defines the functionality and appearance of the menu.</td>
</tr>
<tr>
<td>MNR</td>
<td>Menu resource file. This binary file contains the bitmaps used by the menu.</td>
</tr>
<tr>
<td>MNS</td>
<td>Source menu file (generated by AutoCAD).</td>
</tr>
<tr>
<td>MNT</td>
<td>Menu resource file. This file is generated when the MNR file is unavailable, for example, read-only.</td>
</tr>
<tr>
<td>MNL</td>
<td>Menu LISP file. This file contains AutoLISP expressions that are used by the menu file and are loaded into memory when a menu file with the same file name is loaded.</td>
</tr>
</tbody>
</table>
AutoCAD finds and loads the specified file according to the following sequence. (This sequence is also used when AutoCAD loads a new menu with the MENU command.)

1 AutoCAD looks for a menu source file (MNS) of the given name, following the library search procedure.
   - If an MNS file is found, AutoCAD looks for a compiled menu file (.mnc) of the same name in the same directory. If AutoCAD finds a matching MNC file with the same or later date and time as the MNS file, it loads the MNC file. Otherwise, AutoCAD compiles the MNS file, generating a new MNC file in the same directory, and loads that file.
   - If an MNS file is not found, AutoCAD looks for a compiled menu file (.mnc) of the given name, following the library search procedure. If AutoCAD finds the MNC file, it loads that file.
   - If AutoCAD doesn’t find either a MNS or a MNC file, it searches the library path for a menu template file (.mnc) of the given name. If this file is found, it compiles an MNC and MNS file, then loads the MNC file.
   - If AutoCAD doesn’t find any menu files of the given names, an error message is displayed and you are prompted for another menu file name.

2 After finding, compiling, and loading the MNC file, AutoCAD looks for a menu LISP file (.mnl), using the library search procedure. If AutoCAD finds this file, it evaluates the AutoLISP expressions within that file.

The acad.mnl file contains AutoLISP code used by the standard menu file, acad.mnu. The acad.mnl file is loaded each time the acad.mnu file is loaded.

Each time AutoCAD compiles an MNC file it generates a menu resource file (MNR) which contains the bitmaps used by the menu. The MNS file is an ASCII file that is initially the same as the MNU file (without comments or special formatting). The MNS file is modified by AutoCAD each time you make changes to the contents of the menu file through the interface (such as modifying the contents of a toolbar).

Although the initial positioning of the toolbars is defined in the MNU or MNS file, changes to the show/hide and docked/floating status or changes to the toolbar positions are recorded in the system registry. After an MNS file has been created, it is used as the source for generating future MNC, and MNR files. If you modify the MNU file after an MNS file has been generated, you must use the MENU command to explicitly load the MNU file so that AutoCAD will generate new MNS and MNC files and your changes will be recognized.

**NOTE** If you use the interface to modify the toolbars, you should cut and paste the modified portions of the MNS file to the MNU file before deleting the MNS file.
The MENU command initially requests the MNS or MNC file. To reload a modified MNU file, choose the Menu Template item from the file type list, and then choose the MNU file from the list of files. Doing so protects the MNS file from accidentally being rebuilt, thus losing any toolbar or partial menu modifications done through the interface. While building and testing a menu file, you may find this procedure awkward. The following AutoLISP routine defines a new command, MNU, which reloads the current MNU file without going through all the prompts.

```lisp
(defun C:MNU ()
  (command "_menu" (strcat (getvar "menuname") ".mnu"))
  (princ)
)
```

If you add this code to your acad.lsp file, the MNU command is automatically defined when you restart AutoCAD.

**Base and Partial Menus**

AutoCAD uses the concepts of base and partial menus. The *base menu* is the last menu loaded with the MENU command. A *partial menu* is any menu that is loaded with the MENULOAD command. The MENULOAD command loads a partial menu and lets you add and rearrange the menus on the menu bar. When a partial menu is loaded, all menu areas defined by that menu are loaded into AutoCAD and are available for use. The pull-down menus are not displayed until they are explicitly inserted into the menu bar using the Menu Bar tab of the Menu Customization dialog box or AutoLISP. When you load a partial menu, AutoCAD uses the same procedure described previously to generate MNC, MNR, and MNS files. AutoCAD also loads the associated MNL file and DLL (resource file), if one exists.

When you exit AutoCAD, it records the name of the base menu, the names of up to 24 partial menus (defined by their menu group), and up to 24 menus that are currently present in the menu bar. This ensures that the AutoCAD interface remains the same from session to session.

Using partial menus, users and developers can make effective use of multiple menus. AutoCAD provides a convenient method for referencing specific menu items. Each menu item is assigned a name tag, and each menu file is assigned a menu group name. By specifying a menu group name and name tag, you can enable, disable, and check the status of any menu item.
Controlling Partial Menus with AutoLISP

You can enhance your AutoLISP applications by taking advantage of the partial menu feature. To determine if a particular menu file is currently loaded, use the following syntax with any Pop menu item:

```
(menucmd "Gmenugroup.name_tag=?")
```

If the menu file is currently loaded, it returns a non-nil value; otherwise it returns nil. This syntax is discussed in greater detail in “Referencing Pull-Down and Shortcut Menus” on page 103.

Once your application has determined whether the related menu is loaded, it can take the appropriate action.

Loading Partial Menus

Users load partial menus and modify the menu bar with the MENULOAD Menu Customization dialog box. For an application to load a partial menu, it must use the command line version of the MENULOAD command. The following example uses the AutoLISP `command` function to issue the MENULOAD command and specify the menu file `testmenu` as the partial menu to load.

```
(command "menuload" "testmenu")
```

After the menu is loaded, the application must explicitly place any newly defined menus by using the AutoLISP `menucmd` function as shown in the following examples.

```
(menucmd "P6=+sample.pop1")
(menucmd "Gsample.pop1=+sample.pop2")
```

The first line inserts the `POP1` menu of the `SAMPLE` menugroup immediately before the pull-down menu in the sixth position (from the left). The second line uses global menu referencing and inserts the `POP2` menu of the `SAMPLE` menugroup immediately before the `POP1` menu of the `SAMPLE` menugroup.

This syntax is described in “Referencing Pull-Down and Shortcut Menus” on page 103.

Unloading Partial Menus

The MENUUNLOAD command unloads the partial menu from the base menu, which removes the menus from the menu bar. You can also remove individual menus without unloading the entire menu group. The syntax is similar to that used for displaying pull-down menus:

```
(menucmd "Gsample.pop1=-")
(menucmd "Gsample.pop2=-")
```

The code removes the pull-down menus from view in the current menu; however, the menu group is still loaded.
Frequent changes to the contents of a menu bar result in a poor user interface. It is not recommended that you change the state of the menu bar visually unless requested by the user. For example, if you want to unload an application, menus referenced specifically by the application might be removed as well.

The menu might also be completely reinitialized, removing all loaded partial menus, by executing the MENU command and loading a new base menu file. The procedure also removes associated tag definitions.

**Menu File Structure**

Menu files are divided into sections that relate to specific areas of the AutoCAD interface. Depending on its functionality, each menu area can be defined by one or more sections. Each section contains menu items that provide instructions for the appearance and action related to the menu selection. Menu items consist of the following elements: a name tag, a label, and a menu macro. Although menu items are similar in structure and functionality, each section uses a special syntax for its menu item labels.

A menu file does not need to contain entries for every menu section. You need only those that pertain to your application. It is recommended that you divide your menus into several smaller menu files that can be loaded and unloaded on demand (with the MENULOAD command). Not only does this give you better control of your system resources, but it also makes development and maintenance easier because you can work with smaller pieces.

Menu file sections are identified by section labels that use the format ***section_name. Section labels and their associated menu areas are listed in the following table.

<table>
<thead>
<tr>
<th>Section label</th>
<th>Menu area</th>
</tr>
</thead>
<tbody>
<tr>
<td>***MENUGROUP</td>
<td>Menu file group name</td>
</tr>
<tr>
<td>***BUTTONSN</td>
<td>Pointing-device button menu</td>
</tr>
<tr>
<td>***AUXn</td>
<td>System pointing device menu</td>
</tr>
<tr>
<td>***POPN</td>
<td>Pull-down/shortcut menu areas</td>
</tr>
<tr>
<td>***TOOLBARS</td>
<td>Toolbar definitions</td>
</tr>
</tbody>
</table>

Menu File Structure | 79
You can include comments in menu files for use as copyright notices, documentation, or notes. Any line that begins with two slashes (//) is ignored by the menu compiler, as shown in this example:

//This line is a comment

**Menu Item Syntax**

The general syntax of menu items is consistent across all menu sections that support menu items. Each menu item can consist of a name tag, a label, and a menu macro (except the Image section, which does not permit name tags). A menu item normally resides on one line of the file and uses the following format:

```
name_tag label menu_macro
```

The following sample menu code from a Pop menu illustrates a simple menu item:

```
ID_Quit [Exit]^C^C_quit
```

The first item `ID_Quit` is the name tag and the label `[Exit]` displays Exit in the menu. When this menu item is selected by a user, the menu macro `^C^C_quit` is issued.

**Name Tags**

A name tag is a string consisting of alphanumeric and underscore (_) characters that directly precedes the menu item label. This string uniquely identifies an item within a menu file. Name tags provide the following functionality:
- Link Pop menu items and toolbar buttons to their associated status line help (see “Menu-Specific Help” on page 122)
- Link keyboard key sequences to a corresponding Pop menu macro (see “Accelerator Keys” on page 123)
- Enable menu items to switch from one state to another (such as enabled to disabled and marked to unmarked) from menu macros or AutoLISP (see “Controlling the Display of Menu Item Labels” on page 102)

Name tags are not available in Buttons, Aux, and Image sections. They are available in Screen and Tablet sections but serve no purpose.

Labels

The format and use of menu item labels differ for each menu section. The label is contained within square brackets ([ ]) and defines what is displayed or presented to the user. Menu sections that have no interface for displaying information do not require labels (such as the Buttons, Aux, and Tablet sections); however, they can be used for internal notes. The following table shows how each menu section uses labels.

<table>
<thead>
<tr>
<th>Menu section</th>
<th>Use of label</th>
</tr>
</thead>
<tbody>
<tr>
<td>POPn</td>
<td>Defines the content and formatting of pull-down and shortcut menu selections.</td>
</tr>
<tr>
<td>TOOLBARS</td>
<td>Defines the toolbar name, status (floating or docked, and hidden or visible), and position. Also, it defines each button and its properties.</td>
</tr>
<tr>
<td>IMAGE</td>
<td>Defines the text and image displayed in the Image tile menus.</td>
</tr>
<tr>
<td>SCREEN</td>
<td>Defines the text displayed in the screen menus.</td>
</tr>
<tr>
<td>HELPSTRINGS</td>
<td>Defines the status line help related to menu items in the Pop and Toolbars sections.</td>
</tr>
<tr>
<td>ACCELERATORS</td>
<td>Associates keyboard action with menu macros.</td>
</tr>
</tbody>
</table>

The specific syntax and formatting for menu labels are described in the following sections.
Menu Macros

If you intend to include command parameters in a menu item, you must know the sequence in which that command expects its parameters. Every character in a menu macro is significant, even the blank spaces. As AutoCAD is revised and enhanced, the sequence of prompts for various commands (and sometimes even the command names) might change. Therefore, your custom menus might require minor changes when you upgrade to a new release of AutoCAD.

Most of the examples in this section apply specifically to Pop and Screen menu sections; however, with little or no revisions, most menu macros work equally well in all sections.

When command input comes from a menu item, the settings of the PICKADD and PICKAUTO system variables are assumed to be 1 and 0, respectively. This preserves compatibility with previous releases of AutoCAD and makes customization easier because you are not required to check the settings of these variables.

Menu Macro Syntax

The following table provides a synopsis of special characters used in menu macros. The use of these characters is described in detail in later sections of this chapter.

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>Issues ENTER</td>
</tr>
<tr>
<td>^M</td>
<td>Issues ENTER</td>
</tr>
<tr>
<td>^I</td>
<td>Issues TAB</td>
</tr>
<tr>
<td>SPACEBAR</td>
<td>Enters a space; blank space between command sequences in a menu item is equivalent to pressing the SPACEBAR</td>
</tr>
<tr>
<td>\</td>
<td>Pauses for user input (cannot be used in the Accelerators section)</td>
</tr>
<tr>
<td>_</td>
<td>Translates AutoCAD commands and keywords that follow</td>
</tr>
<tr>
<td>+</td>
<td>Continues menu macro to the next line (if last character)</td>
</tr>
<tr>
<td>=*</td>
<td>Displays the current top level image, pull-down, or shortcut menu</td>
</tr>
<tr>
<td>*^Cc</td>
<td>Prefix for a repeating item</td>
</tr>
</tbody>
</table>
Macro Termination

When a menu item is selected, AutoCAD places a space at the end of the macro before processing the command sequence. AutoCAD processes the following menu macro as though you had entered `line + SPACEBAR`.

```
[Line]line
```

However, some commands, such as TEXT, must be terminated by ENTER, not by a space. Also, it sometimes takes more than one space (or ENTER) to complete a command, but some text editors don’t let you create a line with trailing blanks. Two special conventions prevent these problems:

- If a semicolon (`;`) appears in a menu macro, AutoCAD substitutes an ENTER.
- If a line ends with a control character, a backslash (`\`), a plus sign (`+`), or a semicolon (`;`), AutoCAD does not add a blank after it. (For an example of how to use a control character as a menu item terminator, see “Conditional Macro Expressions” on page 89.)
Look at the `Erase 1` menu item in the following example:

```plaintext
[Erase 1] erase \;
```

If this item simply ended with the backslash (which indicates user input), it would fail to complete the ERASE operation because AutoCAD doesn’t add a blank after the backslash. Therefore, the menu macro uses a semicolon (`;`) to force an ENTER after the user input. Here are more examples:

```plaintext
[UCS] ucs
[UCS W] ucs;
[Address] text \4 0 DRAFT Inc;;Main St;;City, State;;
```

Selecting the first item enters `ucs` and SPACEBAR on the command line, and the following prompt appears:

```
Enter an option
[New/Move/orthoGraphic/Prev/Restore/Save/Del/Apply/?/World] <World>:
```

Selecting the second item enters `ucs`, SPACEBAR, and `;` (interpreted as ENTER) at the command line, which accepts the default value, World. No difference between the first and second item would be evident on the screen; naturally, you wouldn’t put both on the same menu.

Selecting the third item displays a prompt for a start point and then displays the address on three lines. The first of the three semicolons (`;;`) ends the text string; the second causes repetition of the TEXT command; and the third calls for the default placement below the previous line.

**Pausing for User Input**

Sometimes it is useful to accept input from the keyboard or the pointing device in the midst of a menu macro by placing a backslash (`\`) at the point where you want input.

```plaintext
[Circle-1] circle \1
[Layoff] layer off \;
```

`Circle-1` pauses to ask the user for the center point and then reads a radius of 1 from the menu. Note that there is no space after the `\`. `Layoff` pauses to ask the user to enter one layer name, and then turns that layer off and exits the LAYER command. The LAYER command normally prompts for another operation and exits only if you press SPACEBAR (blank) or ENTER (`;`).

Normally, the menu macro resumes after one item is entered. Therefore, it isn’t possible to construct a menu macro that accepts a variable number of inputs (as in object selection) and then continues. However, an exception is
made for the SELECT command; a backslash suspends the menu item until object selection has completed. Consider the following menu item:

```
[Make Red] select \ change previous ;properties color red ;
```

This item uses the SELECT command to create a selection set of one or more objects. It then issues the CHANGE command, references this selection set using the Previous option, and changes the color of all selected objects to red.

**NOTE** Because the backslash character (\) causes a menu macro to pause for user input, you cannot use a backslash for any other purpose in a menu macro. When specifying file directory paths, use a forward slash (/) as the path delimiter: for example, /direct/file.

The following circumstances delay resumption of a menu macro:

- If input of a point is expected, Object Snap modes may precede entry of the actual point.
- If X/Y/Z point filters are used, the menu item remains suspended until the entire point has been accumulated.
- If the SELECT command is used, the menu item doesn’t resume until object selection has been completed.
- If the user responds with a transparent command, the suspended menu macro remains suspended until the transparent command is completed and the originally requested input is received.
- If the user responds by choosing another menu item (to supply options or to execute a transparent command), the original macro is suspended, and the newly selected item is processed to completion before the suspended macro is resumed.

**Foreign-Language Support in Menu Macros**

You can develop menus that can be used with a foreign-language version of AutoCAD. The standard AutoCAD commands and keywords are translated automatically if you precede each command or keyword with the underscore character (_). The acad.mnu file uses this feature extensively.

The following example shows a portion of a Pop menu.

```
[->Arc]
[3-point]C°C_arc
[Start, Cen, End]C°C_arc;_c
[Start, Cen, Angle]C°C_arc;
[Start, Cen, Length]C°C_arc;
[Start, End, Angle]C°C_arc;
[Start, End, Radius]C°C_arc;
```

Menu Item Syntax | 85
Also, the AutoLISP `getcname` function can be used to retrieve the English or localized name of an AutoCAD command (see "getcname" in the AutoLISP Reference).

**Canceling a Command**

To make sure that you have no previous incomplete commands, use the string `\c\c` in a menu item. The `\c\c` string is the same as pressing ESC twice from the keyboard. Although a single `\c` cancels most commands, `\c\c` is required to return to the Command prompt from a DIM command. Therefore, `\c\c` ensures that AutoCAD returns to the Command prompt in most cases.

**Echoes and Prompts**

 Normally, characters read from a menu item appear in the command area of the screen just like keyboard input, and prompts are displayed even if a menu item provides the responses. You can suppress these displays by means of the MENUECHO system variable. If menu input echoing is disabled, a `\p` in the menu item turns echoing on.

**Control Characters in Menu Items**

You can place ASCII control characters in the command string portion of a menu item by entering a caret (`^`) followed by another character. For example, `\c` converts to a single character, CTRL + C (which, when called from a menu macro, issues Cancel, not Copyclip). The nonalphabetic control characters are as follows:

- `^@` (ASCII code 0)
- `^{` (ASCII code 27)
- `^\` (ASCII code 28)
- `^}` (ASCII code 29)
- `^^` (ASCII code 30)
- `^_` (ASCII code 31)

When used in a menu macro, the caret character (`^`) maps to the CTRL key on the keyboard. You can combine the caret with another character to construct menu macros that do such things as toggle the grid on and off (`^G`) or cancel a command (`^C`).

- `[GridFlip]^G`
- `[^Cancel+]^C`

You may want a menu macro to enter one or more characters but not submit them as final input. For example, you can create a series of menu items to act as a numeric keypad.

- `[1]1x^H`
- `[2]2x^H`
- `[3]3x^H`
When you choose one of these items, the appropriate digit is entered. Another character follows (the letter x in this case), and that character is removed by \^H. (CTRL + H is the ASCII code for a backspace.) Each of these menu items ends with a control character, and AutoCAD doesn’t add a space or ENTER to such items. Thus, you can choose \{2\}, \{2\}, \{3\}, \{1\}, constructing the input 2231. Press ENTER to enter the completed number.

The following menu items perform the function just described—this time using the backslash.

\{1\}1\ 
\{2\}2\ 
\{3\}3\ 

The first of these two methods is recommended. Although the second method is easier to implement and in most cases produces the same result, a command issued while a menu pause is active might not function as expected.

To assign menu macros to keys and key sequences initiated from the keyboard, see “Accelerator Keys” on page 123.

**Long Menu Macros**

If an item in the menu file doesn’t fit on one line, you can continue it on the next line. To do this, provide a plus sign (+) as the last character of the line to be continued.

\{Setup \}layer set ground-floor;;grid on;fill off;+ 
limits 0,0 12,9;status

This item, which you might use to set initial conditions for a new drawing, continues onto a second line. Menu items can continue on as many lines as necessary.

**NOTE** Line continuations are not preserved when AutoCAD creates the MNS file.

Use AutoLISP-defined commands or functions that are defined in an MNL file to issue very long menu macros. This keeps your menu easier to read and your code more modular.

**Menu Macro Repetition**

Once you have selected a command, you are likely to use it several times before moving on to another command. That is how you use tools; you pick up a tool, do several things with it, pick up another tool, and so on. To avoid picking up the tool before each use, you can use an AutoCAD command repetition capability, triggered by a null response. Do not use this feature to specify command options.
Using menu macro repetition, you can repeat frequently used commands until you choose another command. If a menu macro begins with `**c`c immediately following the item label, the macro is saved in memory. Subsequent Command prompts are answered by that macro until it is terminated by ESC or by the selection of another menu item.

**NOTE** Do not use `^C` (Cancel) within a menu macro that begins with the string `**c`c; this cancels the menu macro repetition.

The following is an example of the repetitive, or modal, approach to command handling.

```plaintext
[Move  ]*`c`cMOVE Single
[Copy  ]*`c`cCOPY Single
[Erase ]*`c`cERASE Single
[Stretch]*`c`cSTRETCH Single Crossing
[Rotate ]*`c`cROTATE Single
[Scale ]*`c`cSCALE Single
```

Menu macro repetition does not work for items in Image tile menus.

**Use of Single Object Selection Mode**

Single object selection puts object selection in Single selection mode, disables the normal dialog conducted by object selection, and causes the selection to return the first objects selected by a subsequent option. This can be quite handy in a menu as shown in the following menu item:

```
[Erase]*`c`cERASE single
```

This item terminates the current command and activates the ERASE command with the Single selection option. After you select this item, you either point to the single object to be erased, or point to a blank area and specify a window. An object selected in this way is erased, and the menu item is repeated (due to the leading asterisk) so that you can erase something else. Single selection mode leads to more dynamic interaction with AutoCAD.

**Menu Swapping**

AutoCAD provides a method for replacing the contents of an active Buttons, Aux, Pop, Screen, or Tablet menu section. The new menu content can be from another section or submenu in the base menu, or it can come from a partial menu. Menu-swapping information that is related to each menu section is included with the description of that menu section.

The following syntax enables menu swapping from a menu macro:

```plaintext
$section=[menugroup.]menuname
```
These are the descriptions:

\[
\begin{align*}
\$ & \quad \text{Instructs AutoCAD to load a menu section.} \\
section & \quad \text{Specifies the menu section. Valid names are} \\
A1–A4 & \quad \text{for AUX menus 1 through 4} \\
B1–B4 & \quad \text{for BUTTONS menus 1 through 4} \\
P0–P16 & \quad \text{for POP menus 0 through 16} \\
I & \quad \text{for the IMAGE menu} \\
S & \quad \text{for the SCREEN menu} \\
T1–T4 & \quad \text{for TABLET menus 1 through 4} \\
\text{menugroup} & \quad \text{Specifies the Menugroup that } \textit{menuname} \text{ is a member of (not necessary if } \textit{menuname} \text{ is in the base menu).} \\
\text{menuname} & \quad \text{Specifies which section or submenu to insert. It is the main label or alias for the section to load.}
\end{align*}
\]

The following menu items illustrate submenu referencing:

\[
\begin{align*}
\$S=\text{PARTS} \\
\$T1=\text{EDITCMDS} \\
\$T2=\text{SCREEN}
\end{align*}
\]

You can activate the submenu mechanism in the middle of a command without interrupting it. For example, the following command strings are equivalent:

\[
\begin{align*}
\$S=\text{ARCSTUFF} \text{ ARC} \\
\text{ARC } \$S=\text{ARCSTUFF}
\end{align*}
\]

Each menu item starts the ARC command, switches to the ARCSTUFF screen submenu, and awaits the entry of arc parameters. A space must follow the submenu reference to separate it from subsequent commands in the menu item.

You can swap menus only of the same type—one Aux for another, one Pop for another, and so on. Trying to swap between types may result in unpredictable and undesired behavior. However, within a given type, you can swap any menu for any other menu. Swapping can lead to some strange behavior for Tablet menus, because they, typically, do not all have the same number of macros.

Only one occurrence of each Pop menu can be present in either the menu bar or on the active shortcut menu.

**Conditional Macro Expressions**

You can use the \$M= command within a menu macro to introduce macro expressions written in DIESEL (see chapter 5, “Using DIESEL—String Expression Language”). AutoCAD evaluates the portion of the item following \$M=, and the result is reevaluated as the menu macro. The format is:

\[
\text{\$M=expression}
\]
Introducing the macro with $M=\text{tells AutoCAD to evaluate the following string as a DIESEL expression, and that }expression\text{ is the DIESEL expression.}$

\[\text{[Fillfilp]}\text{FILLMODE }$M=$(-,1,$(\text{getvar,fillmode}))\]

Fillfilp switches FILLMODE on and off by subtracting the current value of FILLMODE from 1 and returning the resulting value to the FILLMODE system variable. You can use this method to toggle between the states of system variables whose valid values are 1 or 0.

If you use the DIESEL string language to perform “if-then” tests, conditions might exist where the normal terminating space or semicolon (resulting in ENTER) is undesirable. If you follow the menu macro with `z, AutoCAD does not add ENTER to the end of the macro expression.

As with other control characters in menu items, the `z used here is a string composed of ` (a caret) and z, not CTRL+z.

In the following examples, `z is used as a menu macro terminator.

\[\text{[Model]}\text{}`C$M=$(\text{if,$(=,$(\text{getvar,tilemode}),0),$S=mview _mspace })`Z\]

\[\text{[Paper]}\text{}`C$M=$(\text{if,$(=,$(\text{getvar,tilemode}),0),$S=mview _pspace })`Z\]

If these menu macros do not end with `z, AutoCAD appends ENTER unnecessarily, reissuing the last command entered. For more information and examples, see chapter 5, “Using DIESEL—String Expression Language.”

**Use of AutoLISP in Menu Macros**

You can use AutoLISP variables and expressions, as described in the *Visual LISP Developer's Guide*, to create menu macros that perform complex tasks. This section provide some useful examples.

AutoCAD accepts up to 255 characters of AutoLISP code in menu macros. To use more characters, break up the code into separate modules separated by semicolons (;) so that AutoCAD can read and execute the code in blocks.

For information about using AutoLISP to invoke a dialog box from a menu file, see the *Visual LISP Developer's Guide*.

To use AutoLISP conveniently and efficiently in menu macros, you can place AutoLISP code in a separate MNL file. The menu file and the AutoLISP code are then easier to write and maintain. AutoCAD loads the MNL file when it loads a menu file with the same name.

**Calling a Menu Macro**

To programmatically execute a Pop menu macro as though the user chose it, you can use the following syntax:

\{(menucmd “Gmenugroup.name_tag=”\})
This works only if the Pop menu macro is part of a Pop menu that is on the AutoCAD menu bar and available for use. For more information about this syntax, see the AutoLISP Reference.

**Preset Values**
An application that uses block insertion presets could provide menu items like these:

\[
\text{[Set WINWID]}\text{C}^\text{P}(\text{setq WWID (getreal"Enter window width: "))}\text{C}^\text{P} \\
\text{[Set WALLTHK]}\text{C}^\text{P}(\text{setq WTHK (getreal"Enter wall thickness: "))}\text{C}^\text{P} \\
\text{[Insert Window]}\text{C}^\text{P}(\text{insert window XScale !WWID YScale !WTHK} \\
\text{This code inserts the block "window," scaling its X axis to the current window width and its Y axis to the current wall thickness. In this example, the actual values come from the user-defined AutoLISP symbols \textit{WINWID} and \textit{WALLTHK}. The rotation is up to the user to decide so that the window can be rotated in the wall.}

**Resizing Grips**
With the following menu items grip size adjustment can be done on the fly:

\[
\text{[GRIP-up]}\text{C}^\text{P}(\text{setq "gripsize"(1+(getvar"gripsize"))) (redraw) (princ)} \\
\text{[GRIP-dn]}\text{C}^\text{P}(\text{setq "gripsize"(1-(getvar"gripsize"))) (redraw) (princ)} \\
\text{To add validity checking to these menu items, values less than 0 and greater than 255 cannot be used for the \textit{GRIPSIZE} system variable.}

**Prompting for User Input**
The following menu item prompts the user for two points and draws a rectangular polyline with the specified points as its corners.

\[
\text{[BOX]}\text{C}^\text{P}(\text{setq a (getpoint "Enter first corner: ") (cadr b) (setq b (getpoint "Enter opposite corner: ") (cadr (list (car b) (cadr a)) (redraw) (princ)} \\
\text{Creating these menu items is a more advanced way to use the AutoCAD custom menu feature. Carefully study the preceding example and the information in the AutoLISP Reference. Experimentation and practice will help you use this feature effectively.}

**Menu Groups**
The ***MENUGROUP= label defines the contents of a menu file as members of a particular group. The purpose of the group name is to distinguish its items from other items when using partial menus. This label must precede all menu sections that use the name tag mechanism. The label defines the menu group name for these sections. A Menugroup definition is a string of up to 32 alphanumeric characters (spaces and punctuation marks cannot be used).
Each menu has its own menu group label. However, multiple partial menus can use the same name tag. A name tag, therefore, must be unique only within the menu file in which it was defined. For this reason, applications that load partial menu files can access each other’s items.

Menu group names combined with Pop menu names or aliases provide the following functionality:

- Interactive loading and unloading of partial menus (see `MENULOAD` in the Command Reference).
- Control of menu display and layout from menu macros or AutoLISP (see the Visual LISP Developer’s Guide).

The Menugroup section contains no menu items.

## Buttons and Auxiliary Menus

The Buttons (***BUTTONS*) and Aux (**AUX*) menus are identical in format. Their use depends on the type of pointing devices you are using. The system mouse uses the auxiliary menus, and any other pointing device (digitizer puck or other input device) uses the button menus. All references to auxiliary menus apply equally to button menus. The `BUTTONS1` menu functions identically to the `AUX1` menu, and so on.

Buttons and Aux menu sections are only valid if the menu is used as a primary menu; Buttons and Aux sections are ignored in partial menus.

### Creating Buttons and Auxiliary Menus

The `AUXn` sections of the menu file define the menu macros associated with the buttons on your mouse. Each line in this section represents a mouse button. You can access each button menu with the key/button sequences shown in the following table.

<table>
<thead>
<tr>
<th>Key/button sequence</th>
<th>Menu sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple button press</td>
<td>AUX1 and BUTTONS1</td>
</tr>
<tr>
<td>SHIFT + button press</td>
<td>AUX2 and BUTTONS2</td>
</tr>
</tbody>
</table>
Although the areas 1 through 4 are the only active sections, you can define additional sections and swap them into the active sections. See “Button and Auxiliary Menu Swapping” on page 94.

Creating or customizing button menus can make using the pointing device more efficient and dynamic. By selecting the menu items and adding them to the button menu, you can personalize your pointing device to fit your needs.

Examine the following AUX1 section example (similar to the standard acad.mnu file):

```plaintext
***AUX1
; ^c^c
^b
^o
^g
^d
^e
^t
```

The first line after the menu section label, ***AUX1, represents the next button after the pick button on your pointing device. If the pick button is button number 1, the semicolon (;) assigns ENTER to button number 2 on your pointing device. The second line after the menu section label represents the third button.

**NOTE** The first line after the menu section label ***AUX1 or ***BUTTONS1 is used only when the SHORTCUTMENU system variable is set to 0. If SHORTCUTMENU is set to a value other than 0, the built-in menu is used. Similarly, the second line after the ***AUX1 or ***BUTTONS1 label is used only when the MBUTTONPAN system variable is set to 0.

You cannot reassign the pick button in the menu file. The pick button assignment is controlled by the operating system, or a device specific configuration. The default pick button can be different on each pointing device, depending on the manufacturer.

Because labels in button menus are not displayed, you can use the labels as comments. The following example uses the label area to note the button number.

<table>
<thead>
<tr>
<th>Key/button sequence</th>
<th>Menu sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL + button press</td>
<td>AUX3 and BUTTONS3</td>
</tr>
<tr>
<td>CTRL + SHIFT + button press</td>
<td>AUX4 and BUTTONS4</td>
</tr>
</tbody>
</table>

---

Buttons and Auxiliary Menus | 93
The macro assigned to button number 3 in the example causes another menu to be displayed. It has the following format:

$Pn=*

The $ is the special character code for loading a menu area; $Pn specifies the Pop menu section; and $* displays what is currently loaded to the specified menu area.

Therefore, in the example from acad.mnu, choosing button number 3 displays the menu assigned to the P0 menu area (the P0 menu area is the shortcut menu typically called from the Buttons or Aux menu). Typically, the * *PO P0 section of the menu file is assigned to the P0 menu area.

Each remaining line in that section assigns a command sequence to each subsequent button on the pointing device. For example, ‘‘C’C (ESC twice) is assigned to button 4, and ‘‘B (Snap mode toggle) is assigned to button 5. Your pointing device can recognize as many lines as it has assignable buttons.

**Button and Auxiliary Menu Swapping**

If necessary, you can swap the contents of the active Buttons and Aux menus with that of another menu section of the same type. In addition to the standard AUX1 through AUX4 (and BUTTONS1 through BUTTONS4) sections, you can define additional Aux and Buttons sections for specific purposes. These section names must be an unbroken string (no spaces) with the appropriate prefix, AUX or BUTTONS, followed by any additional text you wish.

**NOTE** The use of text characters following the keyword is not guaranteed to work in future releases of AutoCAD. To ensure compatibility, you should use only index numbers after the keyword. So, even though labels such as * *AUXTEST and ***BUTTONS1-2 are currently valid, labels such as ***AUX10 or ***BUTTONS15 are preferred for their long-term compatibility.

These additional sections can be followed by * *label aliases. The labels must come between the * *section line and the first menu item line for that section. The alias label string can be any string. It does not need to contain any keyword. You can have as many aliases as you like for each section. The alias labels, as well as the section labels, can be used to identify the menu for swapping purposes.
For example, if you want to change the standard right-click action so that AutoCAD displays a different shortcut menu, you can use the following menu syntax.

```plaintext
### AUX2
// Shift + button
$P0=SNAP $p0=* 

### AUX4
// Control + Shift + button - Toggles to custom A2
$A2=CUSTOM_A2 $A4=CUSTOM_A4 "P(princ ">> Custom A2 <<")(princ) "P

### AUX12
**CUSTOM_A2
// Shift + button - Displays the MYPOP menu
$P0=MYPOP $p0=* 

### AUX14
**CUSTOM_A4
// Control + Shift + button - Toggles back to default A2
$A2=AUX2 $A4=AUX4 "P(princ ">> Default A2 <<")(princ) "P
```

Replacing the standard AUX4 definition with the one shown in the example allows you to swap the content of the A2 and A4 menus. After reloading the MNU file, issuing a CTRL+SHIFT+button click loads the contents of the menus defined by the alias CUSTOM_A2 into the A2 menu position and CUSTOM_A4 into the A4 position. Then, using AutoLISP, AutoCAD displays a message on the command line. Now when you issue a SHIFT+button click, the MYPOP menu is loaded into the P0 menu position and is used as the shortcut menu. To return to the default P0 menu, another CTRL+SHIFT+button click (which now calls the CUSTOM_A4 menu) loads the default AUX2 menu back into the A2 menu position and the AUX4 menu back into the A4 position.

You can also use the AutoLISP `menucmd` function to swap-in Aux and Buttons menus. Assuming that the previous examples are in a MNU file with a menu-group of MYGROUP, the following function call loads the CUSTOM_A2 menu into the A2 menu position.

```
(menucmd "A2=mygroup.custom_a2")
```

**NOTE** Previous AutoCAD releases allowed the **label syntax after menu items within a menu section; these were known as submenus. The submenu syntax is still accepted by AutoCAD, but is converted into a ***section label in the MNS file. This syntax is not guaranteed to be valid in future releases. It is recommended that you change all **label submenus into ***section labels. The **alias label is valid, provided that it occurs after a section label and before any menu items.
Special Use of the Backslash

When you select a menu item with one of the menu buttons on a multi-button pointing device, AutoCAD receives not only the button number but also the coordinates of the screen crosshairs at the time you press the button. By carefully constructing the macros in the Buttons and Aux sections of the menu file, you can ignore these coordinates or use them with the command activated by the button.

As described earlier, you can include a backslash (\) in a menu item to pause for user input. For the buttons menus, the coordinates of the screen crosshair are supplied as user input when the button is pressed. This occurs only for the first backslash in the menu item; if the item contains no backslashes, the pointer coordinates are not used. Consider the following menu items:

```plaintext
***AUX2
line
line \n
The first menu button issues an ordinary LINE command and solicits the Specify first point prompt in the normal fashion. The second menu button also issues a LINE command, but AutoCAD reads the current pointer location and uses it as the First point.
```

Pull-Down and Shortcut Menus

The pull-down and shortcut menus are displayed as cascading menus (also known as walking or hierarchical menus). Thus, you can create a logical layout of menus without swapping menu areas. The shortcut menu can provide quick access to often-used menu items such as Object Snap modes. Pull-down and shortcut menu items are similar to items in other menu sections, and you define menu macros similar to the way you define standard screen or tablet menus.

Pull-down menus are defined in the ***POP1 through ***POP499 menu sections, and shortcut menus are defined in the ***POP0 and ***POP500 through ***POP999 sections. The shortcut menus in the upper range are also referred to as context menus. A pull-down menu can contain up to 999 menu items. A shortcut menu can contain up to 499 menu items. Both limits include all menus in the hierarchy. If menu items in the menu file exceed these limits, AutoCAD ignores the extra items. If a pull-down or shortcut menu is longer than the available space on the graphics screen, it is truncated to fit on the screen.

Pull-down menus are always pulled down from the menu bar, but the shortcut menu is always displayed at or near the crosshairs or cursor on the graphics screen, text window or command line, or toolbar areas. The syntax for both
of these POPn menu sections is the same except that the shortcut menu title isn’t included in the menu bar. The shortcut menu title is not displayed at all (but you must still enter a dummy title). Access to the shortcut menu is through the $P0=* menu command, which can be issued by another menu item (such as a BUTTONSN menu item) or by an AutoLISP or ObjectARX program. While the shortcut menu is active, the menu bar is not available.

Creating Pull-Down Menus

AutoCAD scans for POPn menu sections while loading each menu file. For the POP1 through POP16 menu sections, it constructs a menu bar containing the titles of those sections. If no POP1 through POP16 section is defined, AutoCAD inserts default File and Edit menus. Any menu sections greater than POP16 and less than POP500 are available to be inserted into the menu bar with the MENULOAD command or through the menu swapping process (see “Pull-Down Menu Swapping and Inserting” on page 106).

The following example illustrates the syntax that is used to create a pull-down menu.

```plaintext
***POP13
**MYTOOLS
M_Tools [MyTools]
M_Save [Save] <Ctrl+S> "C\_gsave
[--]
M_ShwTB [Show MyToolbar] "C\_toolbar mytools s
M_HidTB [Hide MyToolbar] "C\_toolbar mytools h
[--]
M_EMenu [Edit MyMenu] "C\P(command="notepad"(findfile"my.mnu")) "P
M_LMenu [Reload MyMenu] "C\P(command="\_menu""my.mnu") "P
[--]
M_EPgp [Edit PGP] "C\P(command="notepad"(findfile"acad.pgp")) "P
M_LPgp [Reload PGP] "C\_re-init 16
```
Each menu section can have one or more aliases that are defined by **alias labels following the **popn menu section label. In the previous example, **mytools is an alias for the pop13 menu. For additional information about menu aliases, see “Button and Auxiliary Menu Swapping” on page 94.

**NOTE** The popn menu sections no longer support the **submenu syntax used in previous releases. The **alias syntax is valid, provided that it occurs after a section label and before any menu items.

### Creating Shortcut Menus

Shortcut menus are defined using the same syntax as pull-down menus. The POP0 menu section defines the default Object Snap shortcut menu, and the menu sections from POP500 through POP999 are used for Context shortcut menus.

AutoCAD references the Context shortcut menus by their alias (as in **grips) and uses them in specific situations. The actual popn number is not important, but the alias names must follow the proper naming conventions in order to be used. The following aliases are reserved for use by AutoCAD:

- **GRIPS**
  The content of this menu defines the Hot Grip shortcut menu (right-click in the drawing area while an object’s grip is hot [selected]).

- **CMDEFAULT**
  The content of this menu defines the Default mode shortcut menu (right-click in the drawing area while no command is active and no objects are selected).

- **CMEDIT**
  The content of this menu defines the Edit mode shortcut menu (right-click in the drawing area while one or more objects are selected [but no grips are hot], and no command is active).

In addition to the content of the **CMEDIT menu, the appropriate object menu (if it exists) is inserted into this menu when one or more of a specific object type are selected. Object menus use the following naming convention:

- **OBJECT(S)_objectname**
  If a single object is selected, the **OBJECT_objectname menu is used, and if more than one of the same object is selected, the **OBJECTS_objectname is used. If no **OBJECT_objectname


is available, AutoCAD uses the `OBJECTS_objectname` menu (if it exists).

The object name is the DXF name of the object in all cases except the insert object. To differentiate between a block insertion and an xref, use the names `BLOCKREF` and `XREF`.

The following AutoLISP code defines the command `OTYPE`, which reports the selected object’s DXF name.

```lisp
(defun C:OTYPE()
  (cdr (assoc 0 (entget (car (entsel))))))
```

The content of this menu defines the Command mode menu (right-click in the drawing area while a command is active). In addition to the content of the `CMCOMMAND` menu, the command line options (keywords within the square brackets) are inserted into this menu.

Like the `CMEDIT` menu, the `CMCOMMAND` menu can have context-sensitive information added to it. Any menu named `COMMAND_commandname` is appended to the `CMCOMMAND` menu. The text of `commandname` can be any valid AutoCAD command, including any custom-defined or third-party commands.

To make this work with a hyphen-prefixed command (such as `–INSERT`), you need to name the menu `COMMAND_-INSERT`.

**Summary of Pull-Down and Shortcut Menu Label Syntax**

The following table describes the characters that have a special function when enclosed in a pull-down or shortcut menu label. Each of these characters is described in the sections that follow.

<table>
<thead>
<tr>
<th>Special label characters</th>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>–—</td>
<td>–</td>
<td>Item label that expands to become a separator line in the pull-down and shortcut menus (when used with no other characters).</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>Continues macro to the next line (if last character).</td>
</tr>
</tbody>
</table>
## Chapter 4 Custom Menus

The only nonalphanumeric characters that can be used as the first character in a menu label are those listed above. Nonalphanumeric characters not listed in the previous table are reserved for future use as special menu characters.

### Pull-Down Menu Bar Titles

For pull-down menus, the first label defines the menu bar title; succeeding labels define menu and submenu items. The following example is the top portion of the POP2 pull-down menu section.

```plaintext
***POP2
ID_MnEdit  [&Edit]
ID_U        [&Undo	Ctrl+Z]_u
ID_Redo     [&Redo	Ctrl+Y]^C^C_redo
```

<table>
<thead>
<tr>
<th>Special label characters (continued)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>–&gt;</td>
<td>Label prefix that indicates that the pull-down or shortcut menu item has a submenu.</td>
</tr>
<tr>
<td>&lt;–</td>
<td>Label prefix that indicates that the pull-down or shortcut menu item is the last item in the submenu.</td>
</tr>
<tr>
<td>&lt;-&lt;-. . .</td>
<td>Label prefix that indicates that the pull-down or shortcut menu item is the last item in the submenu, and terminates the parent menu. (One &lt;– is required to terminate each parent menu.)</td>
</tr>
<tr>
<td>$!</td>
<td>Enables the pull-down or shortcut menu item label to evaluate a DIESEL string macro if $( are the first characters.</td>
</tr>
<tr>
<td>~</td>
<td>Label prefix that makes a menu item unavailable.</td>
</tr>
<tr>
<td>!.</td>
<td>Label prefix that marks a menu item with a check mark.</td>
</tr>
<tr>
<td>&amp;</td>
<td>An ampersand placed directly before a character specifies that character as the menu accelerator key in a pull-down or shortcut menu label. For example, S&amp;ample displays as Sample.</td>
</tr>
<tr>
<td>/c</td>
<td>Specifies the menu accelerator key in a pull-down or shortcut menu label. For example, /aSample displays as aSample.</td>
</tr>
<tr>
<td>\t</td>
<td>Specifies that all label text to the right of these characters is pushed to the right side of the menu.</td>
</tr>
</tbody>
</table>
On the first line after the ***POP2 section label, the label [Edit] causes Edit to be displayed as a menu bar title and the letter E is underscored to indicate that it is the menu accelerator key. The name tag associated with the menu title (ID_MnEdit) can be used to enable and disable this entire menu. Pull-down menu titles cannot have an associated menu macro.

Although shortcut menus must define a title, this title is not displayed.

**Cascading Submenus**

Pull-down and shortcut menu labels use special characters (such as –>, <-, and <=<...) to control the hierarchy of cascading menus. These special characters indicate submenus and last items in submenus, and can also terminate all parent menus. Each special character string must be the first characters in an item label.

The special character –> indicates that this item has a submenu, as in the following example:

[–>Point Filters]

If you pull down the Assist menu and choose the Point Filters item or move the cursor to the right end of the item, the Point Filters submenu is displayed.

The special character <- indicates this item is the last item in a submenu, as in the following example:

[<–.YZ].YZ

Special characters <=<... indicate this item is the last item of a submenu and also of its parent menu, as in the following example:

[–>Text]

[–>Attributes]

[<=<–Extract...]^C^Cddatext

**Separation of Menu Item Labels**

To create separator lines, use a label of two hyphens.

[---]

Because the width of each pull-down and shortcut menu is determined by its widest label, the preceding example expands to become a separator line filling the entire width of the menu. You cannot choose separator lines from the menu, and any menu macro assigned to them is ignored.
Controlling the Display of Menu Item Labels

Another customization feature of pull-down and shortcut menus is the ability to control the way that item labels are displayed. You can gray out the labels, making them unavailable to the user or mark them with a check mark.

Labels can also contain DIESEL string expressions to modify the contents of the label. This can disable, mark, or interactively change the text of the displayed label. See “DIESEL Expressions in Menus” on page 132.

When disabling and marking menu item labels, be sure to use an appropriate technique that keeps track of changes that affect the state of the label.

Disabling Labels

A menu item label that begins with a tilde (~~) will be disabled (grayed out). By convention, this indicates that the item is not available. Any commands associated with the item are not issued, and any submenus are inaccessible.

For example, the following menu labels are disabled.

```
[~Line]
[~->Pline]
```

The child menu of a disabled label is inaccessible.

Using DIESEL to Disable Labels

Menu item labels can contain DIESEL string expressions that conditionally disable or enable labels each time they are displayed. For example, the DIESEL string expression within the following menu item label disables the label while a command is active.

```
[$(if,$(getvar,cmdactive),~)MOVE]\^C^C_move
```

Using AutoLISP to Disable Labels

The AutoLISP menucmd function can be used to disable and enable labels from a menu macro or application. For examples, see “Referencing Pull-Down and Shortcut Menus” on page 103.

Marking Labels

You can mark a menu item label with a leading check mark by including an exclamation point and a period (!.). Marking a menu item does not restrict your ability to choose the item, although a marked item can be disabled.

In the following example, the Line menu item is marked with a check.

```
[!.Line]
```

Using DIESEL to Mark Labels

Menu item labels can contain DIESEL string expressions that conditionally mark labels each time they are displayed. The following example places a
check mark to the left of menu labels whose related system variables are currently enabled.

[$(if,$(getvar,orthomode),!.)Ortho]^O
[$(if,$(getvar,snapmode),!.)Snap]^B
[$(if,$(getvar,gridmode),!.)Grid]^G

Using AutoLISP to Mark Labels
The AutoLISP \texttt{menucmd} function can be used to mark labels from a menu macro or application. For examples, see “Referencing Pull-Down and Shortcut Menus” on page 103.

Simultaneous Disabling and Marking
You can mark and disable menu items at the same time. This is the format:

[-!\textunderscore labeltext]

or

[!\textunderscore labeltext]
The \textunderscore is the special character code to disable a menu item and !. is the special character code to mark a menu item.

In the example that follows, the Line menu item is disabled and marked with a check mark. As with the previous examples, a DIESEL expression can be used to simultaneously disable and mark a menu item label.

[-!Line]

Referencing Pull-Down and Shortcut Menus
The two methods for referencing a pull-down or shortcut menu are relative and absolute. \textit{Relative referencing} uses the menu group and name tag; \textit{absolute referencing} uses the absolute position of the menu item in the menu hierarchy. The former method is recommended because of its dynamic nature, which allows it to function properly, regardless of the current state of the menu.

Relative Referencing of Pull-Down and Shortcut Menu Items
To reference a pull-down or shortcut menu item based on its menu group and name tag, use the AutoLISP \texttt{menucmd} function. The following syntax references a menu item based on its name tag.

(menucmd "Gmenugroup.name_tag=value")

The following example disables the menu item \texttt{ID\_Line} in the sample menu group. It works regardless of the menu item’s location in the menu.

[Disable Line](menucmd "Gsample.ID\_Line=-")
If the author of a partial menu is aware of the contents of the base menu, the syntax of a menu item can reference a tag from the base file. An excerpt from the base file `acad.mnu` might look something like the following:

```
***MENUGROUP=ACAD
***POP0
   (and so forth...)
....
***POP6
   ID_MnHelp   [Help]
   ID_Contents [Contents]^C^C_HELP
   ID_About    [About]^C^C_ABOUT
```

A menu item in a partial menu can be modified to have an additional menu item that references the tag in the base menu.

```
***POP2
[Title2]
[Disable Help Contents]^P(menucmd "Gacad.ID_Contents=~") ^P
```

In this manner, multiple partial menu files and specific base files can work together. AutoCAD enforces strict menu group definition so that no two menus can define the same menu group. Attempts to load a menu with a conflicting menu group results in cancellation of the MENULOAD request.

**Absolute Referencing of Pull-Down and Shortcut Menu Items**

In addition to referencing a menu item, you can activate or deactivate a menu item with the `$Pn=xxx` syntax. This is the format:

```
$Pn.i=xxx
```

The `$Pn` specifies the active `POPn` menu section (0 through 16 are valid values); `i` specifies the menu item number; and `xxx` (if present), specifies a string defining the action.

The following example uses the AutoLISP `menucmd` function to reference a pull-down or shortcut menu item. Because AutoCAD menu files are dynamic (through the loading of partial menus), the following syntax won’t work in all cases.

```
[Disable Line Old Way](menucmd "$P1.2=")
```

This syntax relies on the location of the menu item and does not work if a new item was inserted into the `POP1` section by the menu author or if a new pull-down menu is inserted before `POP1` by the MENULOAD command.

You can use the `$Pn=xxx` syntax from a menu macro if it follows the `$` command. The following example disables item 4 in the `POP3` section.

```
$P3.4=--
```

The following example adds a check mark to item 1 in the `POP7` section.

```
$P7.1=!.
```
The following example removes any disabling or mark character from item 1 in the `POP7` section.

\$P7.1=

Menu item numbering is consecutive without regard to the hierarchy of the menu file; item 1 is the first item following the title.

```plaintext
***POP5
[Assist ] Title
[Help! ]'?' Item 1
[Cancel ]"C"C"C Item 2
[-- ] Item 3
[Undo ]"C"C_U Item 4
[Redo ]"C"C_redo Item 5
[-- ] Item 6
[->Osnap ] Item 7
[Center ]center Item 8
```

To make it easy for an item to address itself without regard to location in the menu hierarchy, use these forms:

- \$P@.@=xxx References the current or most recently chosen menu item
- \$P@.n=xxx References item \(n\) in the current or most recently chosen menu

### AutoLISP Access to Label Status

The AutoLISP `menucmd` function accepts \$P\(n\)=xxx command strings but without the leading $. For these functions, the xxx portion of the command string can have special values.

- \(Pn.i=?\) Returns the current disabled and marked status for the specified item as a string (for example, ~ for a disabled item, . for an item with a check mark, and ** for an item that is neither grayed out nor marked).
- \(Pn.i=#?\) Returns the same type of string as described for \(Pn.i=?\), but with the \(Pn.i=\) prefix. This is useful in conjunction with the @ forms, because the actual menu and item number are returned.

For example, if the fifth item in the `POP6` section is disabled, the following `menucmd` code returns the following string values.

```auto
(menucmd "P6.5=?") returns "~"
(menucmd "P6.5=#?") returns "P6.5=~"
```

See "Use of AutoLISP in Menu Macros" in the *Visual LISP Developer's Guide*. 
Pull-Down Menu Swapping and Inserting

Because AutoCAD pull-down menus are the cascading type, you usually don’t need to swap menus. Also, swapping menus can detract from the consistency of the user interface. However, inserting and removing a pull-down menu can be appropriate when the user specifically loads or unloads an application that requires an additional menu.

Swapping Pull-Down Menus

Using $ commands in menu macros, you can swap pull-down menus in specific POPn locations. This method, however, is not recommended unless you can verify that the menu you are replacing is really the one you think it is. Because of the dynamic nature of AutoCAD menus, a menu you inserted at position P6 might not actually be at that location. If you try to swap this menu for another, you might be removing an unintended menu. An alternate method for menu swapping involves relative (or global) referencing (see “Inserting and Removing Pull-Down Menus” on page 106). Using this method, you can insert the new menu in front of a known menu, and then remove the known menu.

For menu-swapping purposes, the active pull-down menu areas are named P1 through P16. The following menu macro replaces the menu at position P3 with the menu named JoesMenu in the menu group MYMENU.

$P3=MyMenu.JoesMenu

The same thing can be done with the menucmd function as follows:

(menucmd "Gmenugroup1.menuname1=+menugroup2.menuname2")

You can use the $Pn=* special command from within any menu macro to force the menu currently assigned to area POPn to be displayed.

NOTE The swapping of pull-down menus does not conform to the Microsoft user interface guidelines and is not guaranteed to be available in future releases of AutoCAD.

Inserting and Removing Pull-Down Menus

You can use the AutoLISP menucmd function to insert or remove a pull-down menu. The syntax is similar to that used to swap pull-down menus except that the left side of the assignment is the pull-down menu before which you want the new menu to be inserted. The right side of the assignment is a plus sign (+), followed by the name of the menu group, followed by a period and the menu’s alias, as shown in the following syntax:

(menucmd "Gmenugroup1.menuname1=+menugroup2.menuname2")
You can also insert a menu with the $Pn= syntax. The following menu macro inserts a menu after the $P5 menu. (You can also use the menucmd function with this format.)

$P5=mymenu.new3

If you use this method for inserting a menu, remember that you cannot rely on it being at the $P6 menu location as you might expect. There are two reasons that this may not be the case:

- If the current menu bar has only three menus, inserting a menu after menu $P5 results in the new menu's location being $P4.
- If the user inserts or removes a menu with the MENULOAD command or when another application inserts or removes menus, menu numbering can get out of sync.

This is the syntax for removing a menu:

(menucmd "Gmenugroup.menuname=-")

The following example removes the menu NEW3 that is a member of the MyMenu group.

(menucmd "Gmymenu.new3=-")

As you might expect, the preceding format is preferable to the $Pn= format because it removes only the specified menu. The following example removes the menu at the $P4 location (whatever it is).

$P4=-

**Toolbars**

The ***TOOLBARS*** section specifies the default layout and contents of the toolbars. It contains a submenu for each toolbar defined by the menu.

**Creating Toolbars**

Five distinct types of items can be specified for toolbars. The syntax of each kind is provided in the following example. All lines other than the separator begin with a standard name tag, which is used to associate help information with the item. In the example, the ***TOOLS1*** declaration is a submenu that uses the alias TOOLS1 as a label to reference the subsequent toolbar definition.
***TOOLBARS
**TOOLS1
TAG1 [Toolbar ("tbarname", orient, visible, xval, yval, rows)]
TAG2 [Button ("btnname", id_small, id_large)]macro
TAG3 [Flyout ("flyname", id_small, id_large, icon, alias)]macro
TAG4 [Control (element)]
[--]

The first line (TAG1) of a toolbar submenu is the toolbar definition, which defines the characteristics of the toolbar. The remaining lines in the submenu can be a mix of the remaining toolbar items. The second line in the example above (TAG2) defines a button. The third line (TAG3) defines a flyout control, and the fourth line (TAG4) defines a special control element. The fifth line defines a separator (--).

**General Toolbar Definition**

The toolbar definition uses the keyword Toolbar followed by a series of options contained in parentheses.

TAG1 [Toolbar ("tbarname", orient, visible, xval, yval, rows)]

The options are as follows:

- **tbarname**
  The string that names the toolbar. The string must include alphanumeric characters with no punctuation other than a dash (-) or an underscore (_). This name along with the alias enables the toolbar to be referenced programmatically.

- **orient**
  The orientation keyword. The acceptable values are Floating, Top, Bottom, Left, and Right and are not case-sensitive.

- **visible**
  The visibility keyword. The acceptable values, Show and Hide, are not case-sensitive.

- **xval**
  A numeric value specifying the X coordinate in pixels, measured from the left edge of the screen to the left side of the toolbar.

- **yval**
  A numeric value specifying the Y coordinate in pixels, measured from the top edge of the screen to the top of the toolbar.

- **rows**
  A numeric value specifying the number of rows.

**Toolbar Button Definition**

Toolbar button definitions use the keyword Button followed by a series of options contained in parentheses.

TAG2 [Button ("btnname", id_small, id_large)]macro
The options are as follows:

`btnname` The string that names the button. The string must include alphanumeric characters with no punctuation other than a dash (–) or an underscore (_). This string is displayed as a tooltip when the cursor is placed over the button.

`id_small` The string that names the ID string of the small-image resource (16 × 15 bitmap). The string must include alphanumeric characters with no punctuation other than a dash (–) or an underscore (_). This can also specify a user-defined bitmap (see “User-Defined Bitmaps” on page 111).

`id_big` The string that names the ID string of the large-image resource (24 × 22 bitmap). If the specified bitmap is not 24 × 22, AutoCAD scales it to that size. The string must include alphanumeric characters with no punctuation other than a dash (–) or an underscore (_). This can also specify a user-defined bitmap (see “User-Defined Bitmaps” on page 111).

`macro` The definition is followed by a command string that follows the standard menu item syntax for command strings.

**Toolbar Flyout Definition**

Toolbar flyout definitions use the keyword `Flyout` followed by a series of options contained in parentheses.

TAG3 [Flyout ("flyname", id_small, id_large, icon, alias)]macro

The options are as follows:

`flyname` The string that names the flyout. The string must include alphanumeric characters with no punctuation other than a dash (–) or an underscore (_). This string is displayed as a tooltip when the cursor is placed over the flyout.

`id_small` The string that names the ID string of the small-image resource (16 × 15 bitmap). The string must include alphanumeric characters with no punctuation other than a dash (–) or an underscore (_). This can also specify a user-defined bitmap (see “User-Defined Bitmaps” on page 111).

`id_big` The string that names the ID string of the large-image resource (24 × 22 bitmap). If the specified bitmap is not 24 × 22, AutoCAD scales it to that size. The string must include alphanumeric characters with no punctuation other than a dash (–) or an underscore (_). It can also specify a user-defined bitmap (see “User-Defined Bitmaps” on page 111).
icon

The Boolean keyword that controls whether to display either its own icon or the last icon selected (other). The acceptable values, OwnIcon and OtherIcon, are not case-sensitive.

alias

The reference to the toolbar to display as the flyout. The alias refers to a toolbar submenu defined with the standard **aliasname syntax.

macro

The definition is followed by a command string that follows the standard menu item syntax for command strings.

**Toolbar Control Definition**

Toolbar control definitions use the keyword Control followed by a name specifying the type of control element requested contained in parentheses.

TAG4 [Control (element)]

The values for the element parameter are as follows (they are not case-sensitive):

- **_Color**
  Specifies the color control element. This element is a drop-down list that provides specification of the current color.

- **_Dimstyle**
  Specifies the dimension style control element. This element is a drop-down list that provides specification of the current dimension style.

- **_Layer**
  Specifies the layer control element. This element is a drop-down list that provides control of the current layers in the drawing.

- **_Linetype**
  Specifies the linetype control element. This element is a drop-down list that provides specification of the current linetype.

- **_Lineweight**
  Specifies the lineweight control element. This element is a drop-down list that provides specification of the current lineweight.

- **_PlotStyle**
  Specifies the plot style control element. This element is a drop-down list that provides specification of the current plot style.

- **_Refblkname**
  Specifies the xref name control element. This element displays the current xref name in edit mode.

- **_UCSManager**
  Specifies the UCS control element. This element is a drop-down list that provides specification of the current UCS.
**View**
Specifies the view control element. This element is a drop-down list that provides specification of the current standard 3D views.

**ViewportScale**
Specifies the viewport scale control element. This element is a drop-down list that provides specification of viewport scaling in Layout mode.

**User-Defined Bitmaps**
User-defined bitmaps can be used in place of the `id_small` and `id_big` image resource names. A user-defined bitmap must be of the proper size for the `id_small` parameter (16 pixels wide by 15 pixels high) and must reside in the library search path. For the `id_big` parameter, if the specified bitmap is not $24 \times 22$, AutoCAD scales it to that size. Specify a user-defined bitmap with the file name and `.bmp` extension as shown in the following example:

```
TAG34 [Button ("My Command", mycmd16.bmp, mycmd24.bmp)]^C^CMYCMD
```

**Referencing Toolbars**
To control toolbars across partial menus, use the following syntax at the Toolbar Name prompt of the TOOLBAR command:

```
menugroup.toolbar-name
```

The following AutoLISP code displays the toolbar `MYBAR` in the menu group `MYGROUP`. (This code assumes that the `MYGROUP` menu is already loaded.)

```
(command "toolbar" "mygroup.mybar" "show")
```

If `menugroup` is not included, then AutoCAD defaults to the base menu.

**Image Tile Menus**

The main purpose of an Image tile menu is to provide an image when the user must select a graphical symbol. You define an Image tile menu by providing an ***IMAGE section in the menu file.

AutoCAD displays images in groups of 20, along with a scrolling list box containing the associated slide names or related text. Image tile submenus are unlimited in length. If an Image tile submenu contains more than 20 slides, AutoCAD provides Next and Previous buttons that the user can press to leaf through pages of images.
Image Tile Menu Items

The Image section uses submenus similar to the Toolbars and Screen sections. As with Pop menu sections, the first line of the submenu is its title. The title is displayed as the label of the dialog box that contains the images. Each submenu should be separated by at least one blank line to clear out items from a previous submenu.

Image tile menu items use item labels to define the text of the scrolling list and the image itself. The label is followed by an associated menu macro. Image tile menus cannot contain name tags.

Image Tile Item Labels

Labels in an Image tile menu generally refer to slide file names instead of text labels that are displayed on the screen. The slide file name, which can be a single slide or part of a library, should appear exactly as you would enter it at the VSLIDE command. The slide file contains the image to show for that selection. The SLIDELIB utility can be used to combine multiple slide files into a slide library.

Image tile menu labels are displayed in a scrolling list box that can accommodate up to 19 characters per label. The slide file name is typically displayed; however, the following icon menu-labeling options are also available.

\[\text{sldbname}\]

The slide name \text{sldbname} is displayed in the list box, and the slide \text{sldbname} is displayed as an image.

\[\text{sldbname}, \text{labeltext}\]

The text \text{labeltext} is displayed in the list box, and the slide \text{sldbname} is displayed as an image.

\[\text{slidlib}(\text{sldbname})\]

The slide name \text{sldbname} is displayed in the list box, and the slide \text{sldbname} in the slide library \text{slidlib} is displayed as an image.

\[\text{slidlib}(\text{sldbname}, \text{labeltext})\]

The text \text{labeltext} is displayed in the list box, and the slide \text{sldbname} in the slide library \text{slidlib} is displayed as an image.
When you supply the text `blank` as an icon label, a separator line is displayed in the list box and a blank image is displayed.

When the first character of an item label is a space, the text supplied as `labeltext` is displayed in the list box and no image is displayed. In this case you can include related commands and simple items such as Exit without needing to make slides that contain those words.

**Image Tile Menu Macros**

Image tile menu macros can perform the same function as other menu macros; however, you cannot use the menu macro repetition feature. These menu macros can contain menu commands, including `$I=` commands. It is possible, therefore, to construct hierarchical Image tile menus in which a selection displays another Image tile menu, and so on. Because the activation of these menus is sequential rather than nested, the complexity of the structures you can create has no limits.

**Displaying Image Tile Menus**

The `$I=` macro command calls the Image tile menu. Before you can display an Image tile menu you must load it. The following syntax loads an Image tile menu.

```
$I=[menugroup.]menuname
```

The `$I=` macro command displays the currently loaded Image tile menu. For example, the following macros load and display the `IMAGE_POLY` image menu in the base menu.

```
$I=image_poly $I=* 
```

The following example loads and displays the `MYBLOCKS` Image menu from a partially loaded menu group `MYGROUP`.

```
$I=mygroup.myblocks $I=* 
```

You can also use the AutoLISP `menucmd` function to load and display Image tile menus. The following code produces the same result as the previous example.

```
(menucmd "I=mygroup.myblocks")
(menucmd "I=")
```
**Sample Image Tile Menus**

The following example shows an Image tile submenu called **3DOBJECTS**.

```
**3DOBJECTS
[3D Objects]
[acad(box3d)]"cai_box
[acad(Pyramid)]"cai_pyramid
[acad(Wedge)]"cai_wedge
[acad(Dome)]"cai_dome
[acad(Sphere)]"cai_sphere
[acad(Cone)]"cai_cone
[acad(Torus)]"cai_torus
[acad(Dish)]"cai_dish
[acad(Mesh)]"cai_mesh
```

The resulting figure is a portion of the Image tile menu.

**3DOBJECTS Image tile menu sample**

In the following example, an Image tile menu is used to insert various electronic parts. The text label is an item that swaps to another Image tile submenu that contains various fasteners.

```
***IMAGE
***IPARTS
[Electronic Parts]
[cap]"insert cap
[res]"insert res
[neon]"insert neon
[triode]"insert triode
[tetrode]"insert tetrode
[Fasteners]$I=ifast $I=* 
```

**IFAST
[Fasteners]
[nut632]...

To activate this Image tile menu, you could choose a menu item such as the following from any menu.

```
[Electronic parts]$I=iparts $I=* 
```
In the following variation, the images are retrieved from a slide library named elib; only the slide name appears in the list box.

```plaintext
***IMAGE
**IPARTS
[Electronic Parts]
[elib(cap)] "Cinsert cap
[elib(res)] "Cinsert res
[elib(neon)] "Cinsert neon
[elib(triode)] "Cinsert triode
[elib(tetrode)] "Cinsert tetrode
```

When successive icon slides from the same library are displayed, the library file remains open. Therefore, the time required to display an Icon menu is significantly reduced.

**Preparation of Slides for Image Tile Menus**

You can use any slide generated by AutoCAD as an image. However, the optimal use of Image tile menus requires that you take care in preparing slides that will serve as images. Follow these guidelines:

- **Keep it simple.** When an Image tile menu is displayed, the user must wait for all images to be drawn before making a selection. If you are using the Image tile menu to show the user numerous complex symbols, make the images simple versions of the symbols rather than full renditions. An image should be as simple as possible and yet immediately recognizable.

- **Fill the box.** Screen space is limited, and images appear in small portions of the full screen. When making a slide for an image, be sure to fill the screen with the image before entering the MSLIDE command. If the image is very wide and short, or long and thin, the Image tile menu will look best if you center the image on the screen with the PAN command before making the slide.

Images are displayed with an aspect ratio of 3:2 (3 units wide by 2 units high). If your drawing area has a different aspect ratio, it can be difficult to produce image slides that are centered in the Image tile menu. If you work within a floating viewport that has an aspect ratio of 3:2, you can position the image and be assured that it will look the same when it is displayed in the Image tile menu.

- **Solid-filled areas** such as wide polylines, traces, and filled solids are visible in Image tiles only if the slide was generated after issuing the –SHADE command. Otherwise, they display as outlines.

- **Do not overuse images** as you encode abstract concepts into symbols.
Screen Menus

The screen menu section controls the screen menu area. By default, the screen menu is disabled. You can enable the screen menu by choosing Display Screen Menu on the Display tab of the Options dialog box.

Creating Screen Menus

The ***SCREEN section label represents the beginning of the AutoCAD screen menus. The submenu section label shown here is identified by the string **S. A simple, concise name, such as this, is convenient when many separate items reference this submenu, as shown in the following example:

<table>
<thead>
<tr>
<th>Screen menu file section</th>
</tr>
</thead>
<tbody>
<tr>
<td>***SCREEN</td>
</tr>
<tr>
<td>**S</td>
</tr>
<tr>
<td>[AutoCAD ]^C^C^P(ai_rootmenus) ^P</td>
</tr>
<tr>
<td>[* * * ]$S=ACAD.OSNAP</td>
</tr>
<tr>
<td>[FILE ]$S=ACAD.01_FILE</td>
</tr>
<tr>
<td>[EDIT ]$S=ACAD.02_EDIT</td>
</tr>
<tr>
<td>[VIEW 1 ]$S=ACAD.03_VIEW1</td>
</tr>
<tr>
<td>[VIEW 2 ]$S=ACAD.04_VIEW2</td>
</tr>
<tr>
<td>[INSERT ]$S=ACAD.05_INSERT</td>
</tr>
<tr>
<td>[FORMAT ]$S=ACAD.06_FORMAT</td>
</tr>
<tr>
<td>[TOOLS 1 ]$S=ACAD.07_TOOLS1</td>
</tr>
<tr>
<td>[TOOLS 2 ]$S=ACAD.08_TOOLS2</td>
</tr>
<tr>
<td>[DRAW 1 ]$S=ACAD.09_DRAW1</td>
</tr>
<tr>
<td>[DRAW 2 ]$S=ACAD.10_DRAW2</td>
</tr>
<tr>
<td>[DIMENSION]$S=ACAD.11_DIMENSION</td>
</tr>
<tr>
<td>[MODIFY1 ]$S=ACAD.12_MODIFY1</td>
</tr>
<tr>
<td>[MODIFY2 ]$S=ACAD.13_MODIFY2</td>
</tr>
<tr>
<td>[HELP ]$S=ACAD.14_HELP</td>
</tr>
<tr>
<td>[ASSIST ]$S=ACAD.ASSIST</td>
</tr>
<tr>
<td>[LAST   ]$S=ACAD.</td>
</tr>
</tbody>
</table>
Screen Submenus

Screen menu submenu labels have the following format:

**menuname [startnum]

The menuname is a string of up to 33 characters containing letters, digits, and the dollar ($), hyphen (–), and underscore (_) characters. The submenu label must reside on a menu file line by itself and must not contain embedded blanks. An optional integer startnum, which specifies the start line of the submenu, can follow menuname.

A submenu can contain any number of items, but the total size of the screen menu is limited by the setting of the SCREENBOXES system variable (typically set to 28). For instance, if a screen menu submenu has 21 items, but the screen can display only 20 items at a time, the last item in the submenu is inaccessible.

When a submenu is activated, its items normally replace those of the previous menu starting at the beginning (menu box 1) and continuing through all items of the submenu. Thus, a submenu can replace only a portion of the previous menu. You can add an item number after the section or submenu label to specify a replacement starting with a menu item other than 1, as shown in the following example:

**SAMPLE 3

When the SAMPLE submenu is activated, the first two menu boxes are unchanged and submenu replacement begins with menu box 3.

To restore the previous screen items, a menu item must issue the following code without a submenu label.

%S=

AutoCAD keeps track of the last eight submenus. If you exceed eight, the first menus are discarded.
The following sample screen menu section demonstrates the use of submenus.

```
***SCREEN
[EASYmenu]

[DRAW... ]$S=Draw_Root
[EDIT... ]$S=Edit_Root
[Bye     ]end

[ -MAIN- ]$S=SCREEN

**Draw_Root 2
[Line    ]line
[Circle  ]circle
[Arc     ]arc

**Edit_Root 2
[Erase   ]$S=Sel_obj erase
[Copy    ]$S=Sel_obj copy
[Move    ]$S=Obj_sel move

**Obj_sel 2
**Sel_obj 2
[Last    ]last
[Previous]previous
[Window  ]window
[Crossing]crossing

[ -PREV- ]$S=

***BUTTONS1
;
redraw
```

Blank line

Three blank lines fill out this page of the menu to 10 lines and blank out items displayed by the submenus. Because no submenu extends below this line, it is displayed in all menus. It recalls the main menu.

The 2 after the submenu name starts this menu on the line after [EASYmenu].

At least one blank line.

Notice the use of a menu alias. At least two blank lines cover up the Sel_obj menu items.

You can use both Obj_sel and Sel_obj to call this menu.

The $S= calls the previous menu.

Pointing-device button menu. Assigns ENTER to button 2. Assigns REDRAW command to button 3.
The previous example contains three submenus: Draw_Root, Edit_Root, and Sel_obj.

Draw_Root and Edit_Root are called from the main screen menu when you select the Draw or Edit menu items. The Draw_Root submenu provides three selection items that correspond to AutoCAD commands. The Edit_Root submenu also contains three selection items, each of which calls the submenu Sel_obj before executing the appropriate command.

In all cases, a –MAIN– selection item recalls the main screen menu. A screen menu writes over (erases) only as many lines of the previous screen menu as it contains. If a screen menu contains more items than boxes on the screen, or if a buttons menu contains more items than buttons available, the excess items are ignored.

You can use blank lines in menu files to lengthen submenus so that they cover up previous menus. You can also include blank lines to improve the readability of the file.

Selecting a menu item called Zoom from the main screen menu can activate a submenu containing the options for the ZOOM command. For an alternative method of calling a command submenu, see “The MENUCTL System Variable” on page 121.

The following example references the submenu **01_FILE in the ACAD menugroup.

```
[FILE ]$S=ACAD.01_FILE
```

Most screen menus in acad.mnu are loaded at menu box 3, enabling the menu labels [AutoCAD] and [* * *] to remain on the screen.
The following example shows how the **01_FILE submenu is displayed on the screen. Notice that the first line (for New) is displayed at menu box 3.

<table>
<thead>
<tr>
<th>Screen menu file section</th>
</tr>
</thead>
<tbody>
<tr>
<td>**01_FILE 3</td>
</tr>
<tr>
<td>[New] ^C^C_new</td>
</tr>
<tr>
<td>[Open] ^C^C_open</td>
</tr>
<tr>
<td>[Qsave] ^C^C_qsave</td>
</tr>
<tr>
<td>[Saveas] ^C^C_saveas</td>
</tr>
<tr>
<td>[Export] ^C^C_export</td>
</tr>
<tr>
<td>[Config] ^C^C_config</td>
</tr>
<tr>
<td>[Plot] ^C^C_plot</td>
</tr>
<tr>
<td>[Audit] ^C^C_audit</td>
</tr>
<tr>
<td>[Recover] ^C^C_recover</td>
</tr>
<tr>
<td>[Purge] ^C^C_purge</td>
</tr>
<tr>
<td>[Quit] ^C^C_quit</td>
</tr>
</tbody>
</table>

The menu items Assist, and Last are displayed at the bottom of the screen menu area because they are part of the **S submenu that is not overwritten by the **01_FILE submenu.

**Item Labels**

If a screen menu item does not contain an item label, the first eight characters of a menu macro appear on the screen menu. The command in the following example would be displayed as SNAP 0.0.

SNAP 0.001
If a label is provided, the first eight characters of the label are displayed in the appropriate screen menu box. Any additional characters can serve as comments.

**NOTE** The maximum number of menu items depends on your system. You can retrieve the number of screen menu boxes with the SCREENBOXES system variable.

### The MNUCTL System Variable

The MNUCTL system variable controls the automatic swapping of screen submenus when a corresponding command is issued. When MNUCTL is set to 1 (on) and an AutoCAD command is called from a menu item, AutoCAD issues a $S=cmdname (where cmdname is the name of the command), which calls a screen submenu of the same name as the command. The Standard menu acad.mnu takes advantage of this feature by setting MNUCTL to 1 from the acad.mnl file. Setting MNUCTL to 0 (off) affects the operation of the Standard menu but may be preferable for older custom menus.

### Tablet Menus

With AutoCAD you can configure up to four areas of your digitizing tablet as menu areas for command input. The sections of the menu file labeled TABLET1 through TABLET4 define the menu macros associated with tablet selections in these areas.

### Creating Tablet Menus

The menu items in TABLETn sections use the same syntax as those in the other sections. Item labels are treated like those in the BUTTONSn sections. Labels can be used as comments, and are not displayed.

The tablet menu areas that you define with the Cfg option of the TABLET command are divided into equal-sized menu selection boxes, which are determined by the number of columns and rows you specify in each area. These tablet menu selection boxes correspond directly to the lines that follow the TABLETn section labels in a left-to-right, top-to-bottom order (whether or not they contain text).

For example, if you configure a menu area for five columns and four rows, the menu item on the line immediately following the section label corresponds to the leftmost selection box in the top row. Similarly, the menu item on the
eighth line following the section label corresponds to the third box from the left in the second row. AutoCAD can recognize up to 32,766 menu items in each tablet section, which should be more than enough for any tablet menu. You can add your own menu macros to the ***TABLET1 section of acad.mnu. The menu items in this area correspond to the 225 boxes at the top of your tablet template (rows A through I and columns 1 through 25). Customize the lines containing menu labels only.

Locate the line in your menu file that contains ***TABLET1 and note that the next 225 lines contain menu labels.

***TABLET1
[A-1]
[A-2]
[A-3]
...

[I-25]

These labels correspond to the template's grid system. You can add your menu macro after the corresponding row-column menu label, using the format described in “Menu Item Syntax” on page 80. Modifying any lines following box [I-25] is not recommended.

## Menu-Specific Help

Status line help messages are an important aspect of a Help system. These are the simple, descriptive messages that appear in the status line when a menu item is chosen. The Helpstrings menu section defines this form of help.

The following example shows a simple menu file that makes use of the Helpstrings menu section.

***MENUGROUP=sample
***POP1
ID_Title  [/TTitle]
ID_Cancel [Cancel Command]`C`C
ID_Line   `[/LLine]`C`C_line
            [Disable Line](menucmd "Gsample.ID_Line=-")
            [Check Line](menucmd "Gsample.ID_Line=!.")

***POP2
[/2Title2]
[Another Pull Down](menucmd "Gsample.ID_Line=-")

***HELPSTRINGS
ID_Title  [This is the Title menu]
ID_Cancel [This item cancels the previous command]
ID_Line   [This draws a simple line]
The syntax for the Helpstrings section is a name tag followed by a label. When a menu item is highlighted, the name tag for that item is queried for a corresponding entry in the ***HELPSTRINGS section. If a match occurs, the string contained within the label is displayed in the status line.

**Accelerator Keys**

AutoCAD supports user-defined accelerator keys. The following is a short example of an Accelerators section.

```plaintext
***ACCELERATORS
ID_Line    [SHIFT+CONTROL+"L"]
[CONTROL+"Q"]^C^C_quit
[CONTROL+SHIFT+"Z"]^C^Czoom extents
```

The Accelerators section contains items in one of two formats. The first is a name tag (such as ID_Line), followed by a label containing modifiers. The modifiers are followed by either a single-character or a special virtual key string (such as "F12") enclosed in quotation marks. This type of item maps a key sequence to a menu item. You can concatenate more than one modifier with another by using the plus symbol (+), as in the first example. When a special key sequence is recognized, the menu item associated with the name tag is executed as if the user had chosen the menu item.

The second method of defining an accelerator uses a label containing a modifier and key string, followed by a command sequence (menu macro). This method maps a key sequence to a command string and does not have a corresponding menu item. The formatting and special characters used in the menu macro are the same as those used in other menu items, except that the backslash character (\) cannot be used as the PAUSE command. If you want to make use of the backslash character, you can use the previous method for defining an accelerator to map a modifier and key string to a menu item that performs the command sequence that includes the desired pause.

The following table lists the valid modifiers.

<table>
<thead>
<tr>
<th>String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTROL</td>
<td>CTRL key</td>
</tr>
<tr>
<td>SHIFT</td>
<td>SHIFT key, either right or left</td>
</tr>
</tbody>
</table>
The following table lists the special virtual keys. (These keys must be enclosed in quotation marks.)

<table>
<thead>
<tr>
<th>String</th>
<th>Description</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>F1 key</td>
<td>Although the F1 key can be assigned a menu macro, this is discouraged because this key is generally associated with Help. Using a modifier with this key is acceptable.</td>
</tr>
<tr>
<td>F2</td>
<td>F2 key</td>
<td>Unmodified, this toggle key switches the state of the text window.</td>
</tr>
<tr>
<td>F3</td>
<td>F3 key</td>
<td>Unmodified, this key runs OSNAP.</td>
</tr>
<tr>
<td>F4</td>
<td>F4 key</td>
<td>Unmodified, this toggle key switches TABMODE on or off.</td>
</tr>
<tr>
<td>F5</td>
<td>F5 key</td>
<td>Unmodified, this toggle key switches ISOPLANE on or off.</td>
</tr>
<tr>
<td>F6</td>
<td>F6 key</td>
<td>Unmodified, this toggle key switches COORDS on or off.</td>
</tr>
<tr>
<td>F7</td>
<td>F7 key</td>
<td>Unmodified, this toggle key switches GRIDMODE on or off.</td>
</tr>
<tr>
<td>F8</td>
<td>F8 key</td>
<td>Unmodified, this toggle key switches ORTHOMODE on or off.</td>
</tr>
<tr>
<td>F9</td>
<td>F9 key</td>
<td>Unmodified, this toggle key switches SNAPMODE on or off.</td>
</tr>
<tr>
<td>F10</td>
<td>F10 key</td>
<td>Unmodified, this toggle key switches Polar Tracking on or off.</td>
</tr>
<tr>
<td>F11</td>
<td>F11 key</td>
<td>Unmodified, this toggle key switches Object Snap Tracking on or off.</td>
</tr>
<tr>
<td>F12</td>
<td>F12 key</td>
<td>None</td>
</tr>
<tr>
<td>INSERT</td>
<td>INS key</td>
<td>None</td>
</tr>
<tr>
<td>DELETE</td>
<td>DEL key</td>
<td>None</td>
</tr>
</tbody>
</table>
**Special virtual keys (continued)**

<table>
<thead>
<tr>
<th>String</th>
<th>Description</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESCAPE</td>
<td>ESC key</td>
<td>Although the ESC key can be assigned a menu</td>
</tr>
<tr>
<td></td>
<td></td>
<td>macro, it is discouraged because this key is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>generally associated with Cancel. CONTROL+</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ESCAPE and CONTROL+SHIFT+ESCAPE cannot be</td>
</tr>
<tr>
<td></td>
<td></td>
<td>assigned a menu macro; these sequences are</td>
</tr>
<tr>
<td></td>
<td></td>
<td>controlled by Windows. Using the SHIFT modifier</td>
</tr>
<tr>
<td></td>
<td></td>
<td>with this key is acceptable.</td>
</tr>
<tr>
<td>UP</td>
<td>↑ key</td>
<td>Must be used with the CONTROL modifier.</td>
</tr>
<tr>
<td>DOWN</td>
<td>↓ key</td>
<td>Must be used with the CONTROL modifier.</td>
</tr>
<tr>
<td>LEFT</td>
<td>← key</td>
<td>Must be used with the CONTROL modifier.</td>
</tr>
<tr>
<td>RIGHT</td>
<td>→ key</td>
<td>Must be used with the CONTROL modifier.</td>
</tr>
<tr>
<td>NUMPAD0</td>
<td>0 key</td>
<td>None</td>
</tr>
<tr>
<td>NUMPAD1</td>
<td>1 key</td>
<td>None</td>
</tr>
<tr>
<td>NUMPAD2</td>
<td>2 key</td>
<td>None</td>
</tr>
<tr>
<td>NUMPAD3</td>
<td>3 key</td>
<td>None</td>
</tr>
<tr>
<td>NUMPAD4</td>
<td>4 key</td>
<td>None</td>
</tr>
<tr>
<td>NUMPAD5</td>
<td>5 key</td>
<td>None</td>
</tr>
<tr>
<td>NUMPAD6</td>
<td>6 key</td>
<td>None</td>
</tr>
<tr>
<td>NUMPAD7</td>
<td>7 key</td>
<td>None</td>
</tr>
<tr>
<td>NUMPAD8</td>
<td>8 key</td>
<td>None</td>
</tr>
<tr>
<td>NUMPAD9</td>
<td>9 key</td>
<td>None</td>
</tr>
</tbody>
</table>
Users who make extensive use of coordinate entry, might find the following menu enhancement very useful.

```
[NUMPAD5] @x^h
[NUMPAD6] <0
[NUMPAD9] <45
[NUMPAD8] <90
[NUMPAD7] <135
[NUMPAD4] <180
[NUMPAD1] <-135
[NUMPAD2] <-90
[NUMPAD3] <-45
```

If this code is added to the Accelerators section, your numeric keypad is modified as follows: the ENTER key enters the @ symbol and the other number keys enter the less-than symbol (<), followed by the angular value represented by its location in the keypad. For example, if you wanted to draw a square that was 3 units on each side, you would enter the following:

Command: **line**
From point: *(specify start point)*
To point: *(press number 5) 3 (press number 6)*
To point: *(press number 5) 3 (press number 2)*
To point: *(press number 5) 3 (press number 4)*
To point: c
Using DIESSEL—String Expression Language

This chapter shows how you can use DIESSEL (Direct Interprettively Evaluated String Expression Language) to alter the AutoCAD® status line through the MODMACRO system variable. You can also use DIESSEL in menu items as a macro language instead of AutoLISP®. DIESSEL expressions accept strings and generate string results.

Because DIESSEL expressions handle strings exclusively, the USERS1-5 system variables are useful for passing information from an AutoLISP routine to a DIESSEL expression. DIESSEL expressions are evaluated by AutoLISP routines through the use of the AutoLISP menucmd function.
Status Line Configuration—MODEMACRO

The status line is a valuable resource. It provides you with important information without interrupting the work flow. The calculated value of the MODEMACRO system variable is displayed in a left-aligned panel in the status bar at the bottom of the AutoCAD window. The number of characters that can be displayed in the status line is limited only by the size of the AutoCAD window (and your monitor). The default panels move to the right as the content of the MODEMACRO panel grows. It is possible to push the default panels completely off the screen (if you so desire).

You can use the MODEMACRO system variable to display in the status line most data known to AutoCAD. The calculation, decision, and editing facilities of MODEMACRO can be valuable when you want to define the status line to your precise specifications.

Understanding the MODEMACRO Variable

MODEMACRO controls the user-defined status line. This variable is set to the null string when you start AutoCAD. It is not saved in the drawing, the configuration file, or anywhere else. If you want to set MODEMACRO to a specific value each time you open a drawing, you can load a MODEMACRO definition with the \texttt{S::STARTUP} function defined by an \texttt{acad.lsp} file. For information on the \texttt{S::STARTUP} function, see “S::STARTUP Function—Post-Initialization Execution” on page 160.

MODEMACRO is a user-string variable. It can be set to any string value, and is limited in length by restrictions that AutoLISP imposes on string variables and by the size of the AutoLISP to AutoCAD communications buffer. A maximum string length of 460 characters works on all systems. To set MODEMACRO, you enter \texttt{modemacro} at the Command prompt or use the \texttt{SETVAR} command with an AutoLISP routine. If you modify the MODEMACRO setting, you can experiment with various status line formats; however, the maximum number of characters you can enter in this manner is 255.

If MODEMACRO is the null string—which you set by entering a period (.) or by passing it an empty string (“”) with the AutoLISP \texttt{setvar} function—AutoCAD displays the standard status line.
MODEMACRO Definitions

The value of MODEMACRO determines what is displayed in the mode status line. The simplest (and least useful) MODEMACRO consists of constant text. For example, to display a company name in the status line, you enter the following:

Command:  \texttt{modemacro}
New value for MODEMACRO, or . for none <"">:  \texttt{Joe's Bank and Grill}

This MODEMACRO always displays the same text; the status line does not reflect changes to the AutoCAD internal state. It doesn't change until you change MODEMACRO.

To make the status line reflect the AutoCAD current state, include “macro expressions” (using the DIESEL language) in the status line. These macro expressions are written in the following form:

\$(\texttt{somefun}, \texttt{arg1}, \texttt{arg2}, \ldots)

In the macro expression, \texttt{somefun} is the name of the DIESEL function (similar to an AutoLISP function name) and \texttt{arg1}, \texttt{arg2}, and so on, are arguments to the function, interpreted according to the function's definition. Unlike AutoLISP, DIESEL macro expressions have only one data type: strings. Macros that operate on numbers express the numbers as strings and convert back and forth as required.

The DIESEL \$(\texttt{getvar}) function retrieves the value of any system variable, providing useful feedback on the status line. Using this function, we can display the current text style name as defined by TEXTSTYLE, updating the style name each time it changes. The MODEMACRO expression is written as follows:

Command:  \texttt{modemacro}
New value for MODEMACRO, or . for none <"">:  \texttt{Style: $(getvar, textstyle)}

\textbf{NOTE}  The following examples show the MODEMACRO string continued onto additional lines. You enter it as one long string in the prompt line area.

Expressions can be nested, and they can be as complex as you want. Consider the following. You want to display the current snap value and angle (in degrees) in the status line. The example that follows uses nested expressions to convert the snap angle from radians to degrees, while truncating the value to an integer.

Command:  \texttt{modemacro}
New value for MODEMACRO, or . for none <"">:  \texttt{Snap: $(getvar, snapunit)$\left(\text{fix,$(\text{\texttt{\$$(getvar, snapang)$$, $(/,$(180, 3.14159))}}$)$}\right)}
You can also display the values in the current linear and angular UNITS modes.

Command: `modemacro`
New value for MODEMACRO, or . for none <"">:
Snap: $(rtos,$(index,0,$(getvar,snapunit))),
$(rtos,$(index,1,$(getvar,snapunit)))$(angtos,$(getvar,snapang))

DIESEL copies its input directly to the output until it comes to the dollar sign character ($) or a quoted string. You can use quoted strings to suppress evaluation of character sequences that would otherwise be interpreted as DIESEL functions. You can include quotation marks in quoted strings by using two adjacent quotation marks. In the following example, the current layer is set to LAYOUT, and MODEMACRO is set to the string.

Command: `modemacro`
New value for MODEMACRO, or . for none <"">:
`"$(getvar,clayer)= "$(getvar,clayer)""`

The status line displays the following:
$(getvar,clayer)="LAYOUT"

**MODEMACRO Definitions with AutoLISP**

Another way to set MODEMACRO definitions is to use AutoLISP. You can save the code samples shown here as ASCII format text files and load them with the AutoLISP `load` function.

The following AutoLISP command defines a MODEMACRO string that provides similar information to that in the built-in status line. Because AutoLISP cannot continue strings from line to line, you use the AutoLISP `strcat` function to assemble the complete MODEMACRO string from shorter component strings.

```lisp
(defun C:ACADMODE ( )
  (setvar "modemacro" (strcat
"Layer $(substr,$(getvar,clayer),1,8)"
"$(if,$(getvar,orthomode), Ortho)"
"$(if,$(getvar,snapmode), Snap)"
"$(if,$(=,$(getvar,tilemode),0),"
"$(if,$(=,$(getvar,cvport),1), P)"
  )")
)
)
```
Save this AutoLISP routine in a file called acadmode.lsp. When you load the routine and execute it, it displays information in the status line. This is not the most useful application of this feature; it is provided only as an example.

The following sample acad.lsp file uses the S::STARTUP function to set the MODEMACRO variable to a string defined by the AutoLISP file mode1.lsp.

```lisp
;;; Sample ACAD.LSP file that uses S::STARTUP to load the
;;; file MODE1.LSP which defines a MODEMACRO string
(defun S::STARTUP ()
  (load "mode1")
  (princ)
)
;;; Additional AutoLISP files can also be defined or
;;; loaded here
```

When the AutoLISP file (mode1.lsp) is loaded, it uses the MODEMACRO system variable to define a status line that displays L: followed by the first eight characters of the layer name, the drawing name and a portion of the path, and ends with the first letter of each name of the currently active modes. The position of the drawing name remains constant, regardless of the length of the layer name.

```lisp
;;; MODE1.LSP
;;; (setvar "modemacro"
; strcat
; "L:${substr,${getvar,clayer},1,30}"
; "${substr,","1,${(-,30,${(strlen,${getvar,clayer})})}}" ;;
; "<.." Note the 8 spaces here
; "${if,${eq,${getvar,dwgname},UNNAMED},UNNAMED,"
; "${substr,${getvar,dwgname}},"
; "${if,${>,${strlen,${getvar,dwgprefix}),29}},{""
; "${(-,${strlen,${getvar,dwgprefix}),29},1" "}",""
; "${strlen,${getvar,dwgname}}"
; "")"
; ")" ")="" ")="" ")="" ")="" ")="" ")=""
; "${if,${getvar,orthomode}, O, )"
; "${if,${getvar,snapmode}, S, )"
; "${if,${getvar,tabmode}, T, )"
; "${if,${and,"
; "${=,${getvar,tilemode},0},${=,${getvar,cvport),1)),P)"
}
```

Indenting code improves the readability of AutoLISP files and DIESEL strings.
DIESEL Expressions in Menus

An additional way to create macros is to implement DIESEL string expressions in menu files. These expressions can return string values in response to standard AutoCAD commands, AutoLISP and ObjectARX™ routines, and other menu macros. They can also return string values to the menu itself, thereby altering the appearance or content of a menu label.

If you are not familiar with menu customization, see chapter 4, “Custom Menus” before continuing with this section.

A DIESEL expression that you use in a menu item must follow the $section=submenu format where the section name is $ and the submenu is the DIESEL expression you want. Frequently, you can implement a menu macro more easily with AutoLISP.

The following examples show two menu items that produce the same result; one uses DIESEL, and the other uses AutoLISP.

This menu item uses the DIESEL expression:

```
[Ps/Ms]^C^P$M=$(if,$(=,$(getvar,cvport),1),mspace,pspace)
```

This menu item uses the AutoLISP expression:

```
[-Ps/Ms-]^C^P(if (= (getvar "cvport") 1)(command "mspace")+(command "pspace"))(princ) ^P
```

Both menu items provide a way to switch between paper space and model space (if TILEMODE is set to 0), but the DIESEL expression is shorter and is evaluated transparently, not requiring the call to the AutoLISP `princ` function. If the special character `^P` (which switches MENUECHO on and off) is omitted in both cases, the DIESEL expression displays only the issued command, whereas the AutoLISP expression displays the entire line of code.

Because the value returned by a DIESEL expression is a text string, it can be used in response to an AutoLISP `getxxx` function call. This functionality enables menu items to evaluate current drawing conditions and to return a value to an AutoLISP routine.

The next example is based on four assumptions:

- The AutoLISP routine is loaded into memory.
- The menu excerpt is included in the current menu.
- The symbols to insert are one unit high by one unit wide.
- The DIMSCALE variable is set to the drawing’s scale factor (that is, a drawing to be plotted at a scale of 1" = 10' would have a scale factor of 120, or a 1/4" = 1' scale drawing would have a scale factor of 48).
If you load and execute the sample AutoLISP routine, AutoCAD inserts the symbol at the size and location you have specified. When plotted, the symbols are the specified size (if the drawing is plotted at the same scale as that specified by DIMSCALE).

The following is a sample AutoLISP routine.

```lisp
(defun C:SYMIN ()
  (setq sym
    (getstring
      "Enter symbol name: ") ; Prompts for a symbol name
  )
  (menucmd "s=symsize") ; Switches the screen menu
    ; to the symsize submenu
  (setq
    siz (getreal
      "Select symbol size: ") ; Prompts for a symbol size
    pl (getpoint
      "Insertion point: ") ; Prompts for insertion point
  )
  (command "insert" ; Issues the INSERT command
    sym ; using the desired symbol
    pl siz siz 0) ; Insertion point, and size
  (menucmd "s=") ; Switches to the previous
    ; screen menu
  (princ) ; Exits quietly
)
```

**NOTE** An AutoLISP routine that you use regularly should include error checking to verify the validity of user input.

The DIESEL expressions in the following menu file multiply the current value of DIMSCALE by the specified value, and return an appropriate scale factor. This cannot be done with similar AutoLISP code; a value returned by an AutoLISP expression cannot typically be used as a response to a `getxxx` function call (such as, the `getreal` function in the preceding sample).

The following is a menu file excerpt.

```plaintext
**symsize 3
[SIZES]

[3/8"] $M=$(*,$(getvar,dimscale),0.375)
[1/2"] $M=$(*,$(getvar,dimscale),0.5)
[5/8"] $M=$(*,$(getvar,dimscale),0.625)
```

DIESEL expressions can also return string values to pull-down menu item labels, so that you can make menus unavailable or otherwise alter the way they are displayed. To use a DIESEL expression in a pull-down menu label, make sure that the first character is the $ character.
In the next example, the current layer is set to `BASE` and the following DIESEL expression is used as the label portion of a `***POPn` section in a menu file.

```
[$(eval,"Current layer: " $(getvar,clayer))]
```

The result is that the appropriate pull-down menu is displayed and updated whenever the current layer changes.

**Current Layer:** BASE

You can also use this method to interactively change the text displayed in a pull-down menu. You use an AutoLISP routine that sets the `USERS1-5` system variables to the selected text, which can be retrieved by a DIESEL macro in a menu label.

**NOTE** The width of pull-down and shortcut menus is determined when the menu file is being loaded. Menu labels generated or changed by DIESEL expressions after a menu is loaded are truncated to fit within the existing menu width.

If you anticipate that a DIESEL-generated menu label will be too wide, you can use the following example to ensure that the menu width will accommodate your labels. This example displays the first 10 characters of the current value of the `USERS3` system variable.

```
[$(eval,"Current value: " $(getvar,users3))+(if, $(eq,$(getvar,users3),""), 10 spaces )]"C"users3
```

You cannot use trailing spaces in a menu label to increase the menu width, because trailing spaces are ignored while the menu is being loaded. Any spaces you use to increase the width of a menu label must be within a DIESEL expression.

The next example uses the same DIESEL expression as the label and a portion of the menu item. It provides a practical way to enter the current day and date into a drawing.

```
[$(edtime,$(getvar,date),DDD", "D MON YYYY)"C"text + \\
\\$M=$(edtime,$(getvar,date),DDD", "D MON YYYY);
```

Also, you can use a DIESEL macro to mark pull-down menu labels or make them unavailable. The following pull-down menu label displays an unavailable ERASE while a command is active. The text is displayed normally when a command is not active.

```
[ $(if,$(getvar,cmdactive),~)ERASE]erase
```

You can use a similar approach to place a mark beside a pull-down menu item or to interactively change the character used for the mark.
DIESEL Expressions in AutoLISP

To use DIESEL expressions in AutoLISP routines you call the AutoLISP `menucmd` function. The format is similar to that used for DIESEL expressions in menu items.

The following code fragment sets variable `c_time` equal to the current time.

```lisp
(setq c_time (menucmd "M=$(edtime,${getvar,date},HH:MM a/p})"))
```

You can use AutoLISP to experiment with DIESEL. The following sample routine defines a new command that you can use to enter DIESEL expressions at the command line.

```lisp
;;; DIESEL.LSP
;;; Lets you enter DIESEL expressions at the command line
(defun C:DIESEL ( / dsl )
  (while (/= dsl "M=")
    (setq dsl (strcat "M=" (getstring T "DIESEL: "))
    (princ (menucmd dsl)))
  (princ)
)
```

Once this routine is defined, entering `diesel` on the command line displays a DIESEL prompt. You can enter any DIESEL expression. If it is valid, it returns the result; if it is invalid, it returns an appropriate DIESEL error message. This routine continues to prompt with DIESEL until you press ENTER to give a null response.

Catalog of DIESEL String Functions

Status retrieval, computation, and display are performed by DIESEL functions. The available functions are described in this section.

All functions have a limit of 10 parameters, including the function name itself. If this limit is exceeded, you get a DIESEL error message.

### + (addition)

Returns the sum of the numbers `val1`, `val2`, … `val9`.

```lisp
$(+, val1 [, val2, …, val9])
```

If the current thickness is set to 5, the following DIESEL string returns 15.

```lisp
$(+, $(getvar,thickness),10)
```
– (subtraction)

Returns the result of subtracting the numbers \(val2\) through \(val9\) from \(val1\).

\(\text{\$\{\, val1 \, \{, \, val2 \, , \ldots \, , \, val9\}\}}\)

* (multiplication)

Returns the result of multiplying the numbers \(val1\), \(val2\), \ldots, \(val9\).

\(\text{\$\{\, val1 \, \{, \, val2 \, , \ldots \, , \, val9\}\}}\)

/ (division)

Returns the result of dividing the number \(val1\) by \(val2\), \ldots, \(val9\).

\(\text{\$\{\, val1 \, \{, \, val2 \, , \ldots \, , \, val9\}\}}\)

= (equal to)

If the numbers \(val1\) and \(val2\) are equal, the string returns 1; otherwise, it returns 0.

\(\text{\$\{\, val1 \, \, val2\}}\)

< (less than)

If the number \(val1\) is less than \(val2\), the string returns 1; otherwise, it returns 0.

\(\text{\$\{\, val1 \, \, val2\}}\)

The following expression gets the current value of HPANG; if the value is less than the value stored in the system variable USERR1, it returns 1. If the value 10.0 is stored in USERR1 and the current setting of HPANG is 15.5, the following string returns 0. (For information about the USERR1 system variable, see USERR1-5 in the Command Reference.)

\(\text{\$\{\, \{\text{\$(getvar, hpang)\}} \, \{\text{\$(getvar, userr1)\}}\}}\)

> (greater than)

If the number \(val1\) is greater than \(val2\), the string returns 1; otherwise, it returns 0.

\(\text{\$\{\, val1 \, \, val2\}}\)
!= (not equal to)
If the numbers $val1$ and $val2$ are not equal, the string returns 1; otherwise, it returns 0.

$(!=, val1, val2)$

<= (less than or equal to)
If the number $val1$ is less than or equal to $val2$, the string returns 1; otherwise, it returns 0.

$(\leq, val1, val2)$

>= (greater than or equal to)
If the number $val1$ is greater than or equal to $val2$, the string returns 1; otherwise, it returns 0.

$(\geq, val1, val2)$

and
Returns the bitwise logical AND of the integers $val1$ through $val9$.

$(\text{and}, val1 [, val2, \ldots, val9])$

angtos
Returns the angular value in the format and precision specified.

$(\text{angtos}, value [, mode, precision])$
Edits the given value as an angle in the format specified by the mode and precision as defined for the analogous AutoLISP function. (The values for mode are shown in the following table.) If mode and precision are omitted, it uses the current values chosen by the UNITS command.

<table>
<thead>
<tr>
<th>Angular units values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode value</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
Edits the AutoCAD Julian date given by `time` (obtained, for example, from `$getvar,date`) according to the given `picture`. The `picture` consists of format phrases replaced by specific representations of the date and time. Characters not interpretable as format phrases are copied literally into the result of `$edtime()`. Format phrases are defined as shown in the following table. Assume that the date and time are Saturday, 5 September 1998 4:53:17.506.

<table>
<thead>
<tr>
<th>edtime format phrases</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Format</strong></td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>DD</td>
</tr>
<tr>
<td>DDD</td>
</tr>
<tr>
<td>DDDD</td>
</tr>
<tr>
<td>M</td>
</tr>
<tr>
<td>MO</td>
</tr>
<tr>
<td>MON</td>
</tr>
<tr>
<td>MONTH</td>
</tr>
<tr>
<td>YY</td>
</tr>
<tr>
<td>YYYY</td>
</tr>
</tbody>
</table>
Enter the entire AM/PM phrase as shown in the preceding table; if AM is used alone, the A will be read literally and the M will return the current month.

If any AM/PM phrases appear in the picture, the H and HH phrases edit the time according to the 12-hour civil clock (12:00–12:59 1:00–11:59) instead of the 24-hour clock (00:00–23:59).

The following example uses the date and time from the preceding table. Notice that the comma must be enclosed in quotation marks because it is read as an argument separator.

\[
$(edtime, $(getvar, date), DDD"," DD MON YYYY - H:MM am/pm)\]

It returns the following:

Sat, 5 Sep 1998 – 4:53am

If \textit{time} is 0, the time and date at the moment that the outermost macro was executed is used. This avoids lengthy and time-consuming multiple calls on $\$(getvar, date)$ and guarantees that strings composed with multiple $\$(edtime)$ macros all use the same time.

\textbf{eq}

If the strings \textit{val1} and \textit{val2} are identical, the string returns 1; otherwise, it returns 0.

\[
$(eq, val1, val2)\]

The following expression gets the name of the current layer; if the name matches the string value stored in the USERS1 system variable, it returns 1. Assume the string “PART12” is stored in USERS1 and the current layer is the same. (For information on the USERS1 system variable, see USERS1-5 in the Command Reference.)

\[
$(eq, $(getvar, users1), $(getvar, clayer)) \quad Returns 1
\]

\textbf{eval}

Passes the string \textit{str} to the DIESEL evaluator and returns the result of evaluating it.

\[
$(eval, str)\]

\textbf{fix}

Truncates the real number \textit{value} to an integer by discarding any fractional part.

\[
$(fix, value)\]
getenv

Returns the value of the environment variable varname.

\$(getenv, \text{varname})

If no variable with that name is defined, it returns the null string.

getvar

Returns the value of the system variable with the given varname.

\$(getvar, \text{varname})

if

Conditionally evaluates expressions.

\$(if, \text{expr}, \text{dotrue}, \text{dofalse})

If expr is nonzero, it evaluates and returns dotrue. Otherwise, it evaluates and returns dofalse. Note that the branch not chosen by expr is not evaluated.

index

Returns the specified member of a comma-delimited string.

\$(index, \text{which}, \text{string})

Assumes that the string argument contains one or more values delimited by the macro argument separator character, the comma. The which argument selects one of these values to be extracted, with the first item numbered 0. This function is most frequently used to extract X, Y, or Z coordinates from point coordinates returned by \$(getvar).

Applications can use this function to retrieve values stored as comma-delimited strings from the USERS1-5 system variables.

linelen

Returns the length, in characters, of the longest status line that the display can present to the user.

\$(linelen)

You can use this to vary the format of the status line, depending on the display capacity. This function is useful for MODEMACRO status-line configuration only.
The space available to MODEMACRO on the status line is currently fixed at 240 characters. The $(linelen)$ function will always return 240. This function may not be available in future releases of AutoCAD.

**nth**

Evaluates and returns the argument selected by `which`.

\[ $(nth, which, arg0 [, arg1, ..., arg7]) \]

If `which` is 0, `nth` returns `arg0`, and so on. Note the difference between $(nth)$ and $(index)$; $(nth)$ returns one of a series of arguments to the function, while $(index)$ extracts a value from a comma-delimited string passed as a single argument. Arguments not selected by `which` are not evaluated.

**or**

Returns the bitwise logical OR of the integers `val1` through `val9`.

\[ $(or, val1 [, val2, ..., val9]) \]

**rtos**

Returns the real value in the format and precision specified.

\[ $(rtos, value [, mode, precision]) \]

Edits the given `value` as a real number in the format specified by the `mode` and `precision` as defined by the analogous AutoLISP function. If `mode` and `precision` are omitted, it uses the current values selected with the UNITS command.

**strlen**

Returns the length of `string` in characters.

\[ $(strlen, string) \]

**substr**

Returns the substring of `string`, starting at character `start` and extending for `length` characters.

\[ $(substr, string, start [, length]) \]

Characters in the string are numbered from 1. If `length` is omitted, it returns the entire remaining length of the string.
**upper**

Returns the *string* converted to uppercase according to the rules of the current locale.

\[ $(\text{upper, string}) \]

**xor**

Returns the bitwise logical XOR of the integers \( val1 \) through \( val9 \).

\[ $(\text{xor, } val1 [, \text{ val2},..., \text{ val9}]) \]

### Error Messages

Generally, understanding the mistakes you made in a DIESEL expression will be easy to see. Depending on the nature of the error, DIESEL embeds an error indication in the output stream.

<table>
<thead>
<tr>
<th><strong>DIESEL error messages</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Error message</strong></td>
</tr>
<tr>
<td>$?$</td>
</tr>
<tr>
<td>$$(\text{func,??})$$</td>
</tr>
<tr>
<td>$$(\text{func})??$$</td>
</tr>
<tr>
<td>$$(++)$$</td>
</tr>
</tbody>
</table>
In addition to its command line and menu interfaces, AutoCAD® provides command-scripting capabilities and application programming interfaces (APIs) that you can use to control drawings and databases. This chapter provides an overview of the scripting and API features in AutoCAD.

The programming interfaces introduced here are ActiveX® Automation, VBA (Visual Basic® for Applications), AutoLISP®, Visual LISP™, and ObjectARX™. The type of interface you use depends on your application needs and programming experience.
Command Scripts

AutoCAD provides a *script* feature that reads commands from a text file. This feature is useful for executing a sequence of commands. You can invoke a script when you start AutoCAD, or you can run a script from within AutoCAD using the SCRIPT command. A script provides an easy way to create continuously running displays for product demonstrations and trade shows.

You create script files outside AutoCAD, using a text editor (such as Microsoft® Windows® Notepad) or a word processor (such as Microsoft Word) that saves the file in ASCII format. The file extension must be `.scr`.

Script files can contain comments. Any line that begins with a semicolon (`;`) is considered a comment, and AutoCAD ignores the line while processing the script file.

When command input comes from a script, the settings of the PICKADD and PICKAUTO system variables are assumed to be 1 and 0, respectively. This preserves compatibility with previous releases of AutoCAD and makes customization easier (because you don’t have to check the settings of these variables).

The AutoCAD Undo feature considers a script to be a *group*, reversible by a single UNDO command. However, each command in the script causes an entry in the undo log, which can slow script processing. If you like, you can use UNDO Control None to turn the Undo feature off before running the script (or at the beginning of the script itself). Be sure to turn it back on (UNDO All) when the script is finished.

All references to long file names that contain embedded spaces must be enclosed in double quotes. For example, to open the drawing `my house.dwg` from a script, you must use the following syntax:

```plaintext
don "my house"
```

The current script terminates when another SCRIPT command is invoked.

Invoking a Script When Loading AutoCAD

To run a script when you start AutoCAD, use the following command form:

```
acad [existing_drawing] [/t template] [/v view] /b script-file
```

The script file must be the last parameter listed in the `acad` program call line; a file extension of `.scr` is assumed. If AutoCAD can’t find the script file, AutoCAD reports that it can’t open the file.
Using a script when starting AutoCAD to set up your drawing environment is a good example of how a series of commands can be run automatically. For instance, every time you begin a new drawing, you turn the grid on, set the global linetype scale to 3.0, and set layer 0 as your current layer, with red as the color. You can do this using a prototype drawing, but for the purposes of this section, do it instead with the following script, stored in a file called setup.scr.

```plaintext
; Turn grid on
grid on
; Set scale for linetypes
ltscale 3.0
; Select current layer and its color
layer set 0 color red 0

; Blank line (above) to end LAYER command
```

To create a drawing using the file MyTemplate.dwt as the template, you start AutoCAD as follows:

```plaintext
acad /t MyTemplate /b setup
```

This command creates a new drawing and issues a sequence of setup commands from the setup.scr file. When the script has finished running, the Command prompt is displayed.

You can no longer start a new drawing with a given name. AutoCAD applies the file name to the drawing when it is saved.

You must be very familiar with the sequence of AutoCAD prompts to provide an appropriate sequence of responses in the script file. Keep in mind that AutoCAD prompts and command names may change in future releases, so you may need to revise your scripts when you upgrade to a later version of AutoCAD. Similarly, avoid the use of abbreviations; future command additions might create ambiguities. Also, note that each blank space in a script file is significant; AutoCAD accepts either a space or an ENTER as a command or data field terminator.

The following system prompt syntax specifies a script for reconfiguration:

```plaintext
acad -r drawing-name script-file
```

To run a script for a drawing that uses the default prototype (a new unnamed drawing) use the following syntax:

```plaintext
acad /b make_dwg
```
Creating Slide Shows

Scripts are useful for creating slide shows. Ordinarily, the speed with which you can display slides is limited by the number of disk accesses necessary to read the slide file. You can, however, preload the next slide from disk into memory while the audience is assimilating the current slide, and then quickly display the new slide from memory.

To preload a slide, place an asterisk before the file name in the VSLIDE command. The next VSLIDE command senses that a slide has been preloaded and displays it without asking for a file name. The following example script illustrates the preload feature:

; Begin slide show, load SLIDE1
VSLIDE SLIDE1
; Preload SLIDE2
VSLIDE *SLIDE2
; Let audience view SLIDE1
DELAY 2000
; Display SLIDE2
VSLIDE
; Preload SLIDE3
VSLIDE *SLIDE3
; Let audience view SLIDE2
DELAY 2000
; Display SLIDE3
VSLIDE
; Let audience view SLIDE3
DELAY 3000
; Cycle
RSCRIPT

The disk-access time to load the next slide overlaps with the viewing time for the current slide. Additional delays are used as well.

ActiveX Automation

ActiveX Automation, a technology developed by Microsoft and based on the COM (component object model) architecture, is a new programming interface for AutoCAD. You can use it to customize AutoCAD, share your drawing data with other applications, and automate tasks. Through Automation, AutoCAD exposes programmable objects, described by the AutoCAD Object Model, that can be created, edited, and manipulated by other applications. Any application that can access the AutoCAD Object Model is an Automation controller, and the most common tool used for manipulating another application using Automation is Visual Basic for Applications (VBA). This form of Visual Basic is
found as a component in many Microsoft Office applications. You can use these applications, or other Automation controllers, such as Visual Basic and Delphi, to drive AutoCAD.

For example, you might want to prompt for input, set preferences, make a selection set, or retrieve drawing data. You can decide on the controller to use, depending on the type of manipulation.

Using Automation, you can create and manipulate AutoCAD objects from any application that serves as an Automation controller. Thus, Automation enables macro programming across applications, a capability that does not exist in AutoLISP. With Automation you can combine the features of many applications into a single application.

The displayed objects are called Automation objects. Automation objects make methods and properties available. Methods are functions that perform an action on an object. Properties are functions that set or return information about the state of an object.

For detailed information about using VBA to control AutoCAD ActiveX Automation, see the ActiveX and VBA Developer’s Guide and ActiveX and VBA Reference.

**Using Automation Applications**

Virtually any type of application can access the displayed Automation objects within AutoCAD. These applications can be stand-alone executables, dynamic linked library (DLL) files, and macros within applications such as Microsoft Word or Microsoft Excel. The most common of these is most likely the stand-alone executable. If you are using applications from application developers, follow their instructions for installation and use of their product. This section describes a few ways to run a stand-alone executable from within AutoCAD.

**Starting Your Application from the Command Line**

You can use the acad.pgp file to define a new AutoCAD command that runs an external command to start your application. The following example defines the RUNAPP1 command, which runs the application app1.exe in the C:\vbapps\ directory. (Add this code to the external commands section of your acad.pgp file.)

```
RUNAPP1, start c:\vbapps\app1, 0
```
If your application requires command line parameters, you can use the following code:

```
RUNAPP2, start c:\vbapps\app2, 0, *Parameters: ,
```

The previous example defines the RUNAPP2 command, which prompts you for parameters and then passes them to your application. For more information on the `acad.pgp` file, see “acad.pgp—Program Parameters File” on page 16.

You can also use the AutoLISP `startapp` function to start an application that makes use of Automation. Once AutoLISP starts the external application, it has no control over its actions. You can, however, use AutoLISP to locate and run different applications based on certain parameters.

**Starting Your Application from a Menu**

After defining a new command to start your application (as described in “Starting Your Application from the Command Line”), you can use that command in a menu macro. The menu macro can be called from a menu item in any menu area. If you use only one or two applications, you can add them to one of the standard pull-down menus. If you have a group of applications, you can add your own pull-down menu or toolbar that is specifically dedicated to those applications. For more information on the available options for menu customization, see chapter 4, “Custom Menus.”

The following example is a complete menu file that defines a new pull-down menu called MyApps. The menu file defines the `MYAPPS` menu group and two groups of menu items. The first three menu items use the AutoLISP `startapp` function to start the associated application, and the last two items assume that the commands APP4 and APP5 have been defined (in the `acad.pgp` file). This file also includes a Helpstrings section, which provides status line Help when the menu item is selected. The Accelerators section defines accelerator keys that run each of the menu macros. These menu sections use name tags (such as `ID_App1`) to link their associated actions.
After saving this code to a file named `myapps.mnu`, use the `MENULOAD` command to add the menu to the menu bar. When loading the file the first time, you need to specify the MNU file type (.mnu).

**AutoCAD VBA**

Microsoft® Visual Basic for Applications (VBA) is an object-based programming environment designed to provide rich development capabilities. Implemented in AutoCAD, these capabilities shorten the development times of custom business solutions. VBA first appeared in Microsoft Excel and Microsoft Project in 1994. Aside from being a new addition to AutoCAD, VBA 5.0 is now a core component of Microsoft Office 97 and is integrated into Microsoft Word and Microsoft PowerPoint®.
VBA has full Visual Basic language syntax, the new forms package, and support for ActiveX Controls. AutoCAD VBA is an in-process controller, providing better performance in AutoCAD. It also allows integration with other applications that use VBA. AutoCAD can control Automation in other applications by using their object libraries.

The stand-alone development editions of Visual Basic, which must be purchased separately, complement AutoCAD VBA with additional components such as an external database engine and report-writing capabilities.

The advantage of implementing VBA for AutoCAD is fourfold:

- The Visual Basic programming environment is easy to learn and use.
- VBA runs with AutoCAD to improve performance.
- Dialog construction is quick and effective. Developers can prototype applications and receive feedback on designs quickly.
- Projects can be stand-alone or embedded in drawings, providing developers great flexibility in the distribution of their applications.

VBA sends messages to AutoCAD by the AutoCAD ActiveX Automation Interface. AutoCAD VBA permits the Visual Basic environment to run simultaneously with AutoCAD and provides programmatic control of AutoCAD through the ActiveX Automation Interface. This linking of AutoCAD, ActiveX Automation, and VBA provides an extremely powerful interface. It not only controls AutoCAD objects, but it also sends data to or retrieves data from other applications.

**Developing with AutoCAD VBA**

The integration of VBA into AutoCAD 2000 provides an easy-to-use visual tool for customizing AutoCAD. The Visual Basic development environment runs simultaneously with AutoCAD providing programmatic control of AutoCAD through the ActiveX Automation feature. Using the AutoCAD ActiveX Automation and VBA together provides an extremely powerful method for manipulating AutoCAD objects and exchanging data with other applications. For example, you can create an application that extracts attribute information automatically, inserts the results directly into an Excel spreadsheet, and performs any data transformations you need.

Three fundamental elements define VBA programming in AutoCAD. The first is AutoCAD itself, which has a rich set of objects that include AutoCAD entities, data, and commands. AutoCAD is an open-architecture application with multiple levels of interface. To use VBA effectively, familiarity with
AutoCAD programmability is highly desirable. However, you will find that the VBA object-based approach is quite different from that of AutoLISP.

The second element is the AutoCAD ActiveX Automation Interface, which establishes messages (communication) with AutoCAD objects. Programming in VBA requires a fundamental understanding of ActiveX Automation. A description of the AutoCAD ActiveX Automation Interface can be found in the *ActiveX and VBA Developer's Guide*.

The third element that defines VBA programming is VBA itself. It has its own set of objects, keywords, constants, and so forth, that provide program flow, control, debugging, and execution. The Microsoft extensive Help system for VBA is included with AutoCAD VBA.

The AutoCAD ActiveX/VBA interface provides several advantages over other AutoCAD API environments:

- **Speed**: Running in-process with VBA, ActiveX applications are faster than AutoLISP applications.
- **Ease of Use**: The programming language and development environment are easy-to-use and come installed with AutoCAD.
- **Windows Interoperability**: ActiveX/VBA are designed to be used with other Windows applications and provide an excellent path for communication of information across applications.
- **Rapid Prototyping**: The rapid interface development of VBA provides the perfect environment for prototyping applications, even if those applications will be developed eventually in another language.
- **Programmer Base**: Millions of programmers around the world already use Visual Basic. AutoCAD ActiveX/VBA opens up AutoCAD customization and application development to these programmers as well as those who will learn Visual Basic in the future.

AutoCAD ActiveX provides a mechanism to control AutoCAD programmatically from within or outside of AutoCAD. It does this by displaying AutoCAD objects to the “outside world.” Once these objects are displayed, they can be accessed by many different types of programming languages and environments, and by other applications such as Microsoft Word VBA or Microsoft Excel VBA.

The advantage of implementing an ActiveX interface for AutoCAD is two-fold:

- Programmatic access to AutoCAD drawings is opened up to many more programming environments. Before ActiveX Automation, developers were limited to an AutoLISP or C++ interface.
- Sharing data with other Windows applications, such as Microsoft Excel and Microsoft Word, is made dramatically easier.
Using AutoCAD VBA Applications

Although Microsoft Office 97 applications store VBA projects, macros, and programs inside a specific document, AutoCAD uses a separate file with the .dvb extension. In this way, VBA interfaces with AutoCAD in much the same way that AutoLISP and ObjectARX do. Because VBA projects are stored in a separate file, a VBA project can open and close different AutoCAD drawings during an AutoCAD session.

NOTE AutoCAD VBA projects are not binary compatible with stand-alone Visual Basic projects (VBP files). However, forms, modules, and classes can be exchanged between dissimilar projects using the IMPORT and EXPORT VBA commands in the VBA integrated development environment (IDE).

You load a VBA project with the VBALOAD command. Once loaded, its modules and macros are available in the Macros dialog box. To run the VBA module you use the VBARUN command. If no VBA project is loaded, the options are unavailable. Procedures listed in the Macro Name box use the following syntax:

```
module.macro
```

In the Macros dialog box you choose the Macro Scope and select from the listed modules.

Using the Command Line to Run a VBA Macro

AutoCAD command line prompt equivalents are available using the –VBARUN command (signified by a minus sign in front of the VBARUN command). You can run VBA macros from the command line, scripts, and other AutoCAD programming environments. The only argument for the command is the Visual Basic module name using the `module.macro` syntax. The syntax looks like this:

```
-vbarun <module.macro>
```

Because macros with the same name can be duplicated in modules, the `module.macro` syntax differentiates the macro and allows for unique selection.

Automatic Loading and Execution of VBA Projects

As you build up a number of VBA projects, you can load them automatically each time you run AutoCAD. The macros they contain are immediately available. Additionally, the APPLOAD command provides a Startup Suite option that automatically loads the specified applications.
**acadvba.arx—Automatic Loading of VBA**

You cannot load VBA until an AutoCAD VBA command is issued. If you want to load VBA automatically every time you start AutoCAD include the following line in the *acad.rx* file:

```
acadvba.arx
```

You can automatically run a macro in the *acad.dvb* file by naming the macro AcadStartup. Any macro in your *acad.dvb* file called AcadStartup automatically executes when VBA loads.

**acad.dvb—Automatic Loading of a VBA Project**

The *acad.dvb* file is useful if you want to load a specific VBA project that contains macros you want each time you start AutoCAD. Each time you start a new AutoCAD drawing session, AutoCAD searches for the *acad.dvb* file and loads it.

If you want a macro in your *acad.dvb* file to run each time you start a new drawing or open an existing one, add the following code to your *acaddoc.lsp* file:

```
(defun S::STARTUP()
  (command "-_vbarun" "updatetitleblock")
)
```

---

**AutoLISP and Visual LISP**

AutoLISP is based on the LISP programming language, which is simple to learn and very powerful. AutoCAD has a built-in LISP interpreter that lets you enter AutoLISP code on the command line or to load AutoLISP code from external files.

AutoLISP has been enhanced with Visual LISP (VLISP), which offers an integrated development environment (IDE) that includes a compiler, debugger, and other development tools to increase productivity. VLISP represents the next generation of LISP for AutoCAD. It adds more capabilities and extends the language to interact with objects using ActiveX. VLISP also enables AutoLISP to respond to events through object reactors.

Unlike ObjectARX, or VBA, each document opened in the Multiple Design Environment (MDE) has its own Visual LISP namespace and environment. A namespace is an insulated environment keeping AutoLISP routines that are specific to one document from having symbol or variable name and value conflicts with those in another document. For example, the following line of code sets a different value to the symbol a for different documents.

```
(setq a (getvar "DWGNAME"))
```
Visual LISP provides mechanisms for loading symbols and variables from one namespace to another. More information about namespaces can be found in the *Visual LISP Developer’s Guide*.

AutoLISP applications or routines can interact with AutoCAD in many ways. These routines can prompt the user for input, access built-in AutoCAD commands directly, and modify or create objects in the drawing database. By creating AutoLISP routines you can add discipline-specific commands to AutoCAD. Some of the standard AutoCAD commands are actually AutoLISP applications.

Visual LISP provides three file format options for AutoLISP applications including the following:

- Reading ASCII AutoLISP text from an LSP file (.lsp)
- Reading an FAS file (.fas)
- Reading a VLX file (.vlx)

An LSP file is an ASCII text file that contains AutoLISP program code. An FAS file is a binary, compiled version of a single LSP program file. A VLX file is a compiled set of one or more LSP and/or dialog control language (DCL) files.

**NOTE**  Like-named AutoLISP application files load based on their *Modified* time stamp; the LSP, FAS, or VLX file with the most recent time stamp is loaded unless you specify the full file name (including the file name extension). This also applies to the special startup files described in “Automatic Loading and Execution of AutoLISP Routines.”

Because AutoCAD can read AutoLISP code directly, no compiling is required. While Visual LISP provides an IDE, you may choose to experiment by entering code at the Command prompt, which allows you to see the results immediately. This makes AutoLISP an easy language to experiment with, regardless of your programming experience. Once you become proficient with AutoLISP, you will find that you use it as an extension to the basic AutoCAD commands.

Even if you are not interested in writing AutoLISP applications, your AutoCAD package includes many useful routines. Routines are also available as shareware through third-party developers. Knowing how to load and use these routines can enhance your productivity.

**NOTE**  When command input comes from the AutoLISP *command* function, the settings of the PICKADD and PICKAUTO system variables are assumed to be 1 and 0, respectively. This preserves compatibility with previous releases of AutoCAD and makes customization easier (because you don’t have to check the settings of these variables).
For information about AutoLISP programming, see the *Visual LISP Developer's Guide*, and for information about AutoLISP and Visual LISP functions, see the *AutoLISP Reference*. AutoLISP programs can use dialog boxes with their applications. Programmable dialog boxes are described online only in the *Visual LISP Developer's Guide*.

**Using AutoLISP Applications**

AutoLISP applications are stored in ASCII text files with the .lsp extension. These files generally have a header portion that describes a routine, its use, and any specific instructions. This header might also include comments that document the author and the legal information regarding the use of the routine. Comments are preceded by a semicolon (;). You can view and edit these files with a text editor or word processor that can produce an ASCII text file.

Before you can use an AutoLISP application, it must first be loaded. You can use the APPLOAD command or the AutoLISP `load` function to load an application (see “APPLOAD” in the *Command Reference*). Loading an AutoLISP application loads the AutoLISP code from the LSP file into your system’s memory.

Loading an application with the `load` function involves entering AutoLISP code at the Command prompt. If the `load` function is successful, it displays the value of the last expression in the file on the command line. This is usually the name of the last function defined in the file or instructions on using the newly loaded function. If `load` fails, it returns an AutoLISP error message. A `load` failure can be caused by incorrect coding in the file or by entering the wrong file name on the command line. The syntax for the `load` function is

```
(load filename [onfailure])
```

This syntax shows that the `load` function has two arguments: `filename`, which is required, and `onfailure`, which is optional. When loading an AutoLISP file on the command line, you typically supply only the `filename` argument. The following example loads the AutoLISP file `myfile.lsp`.

Command:  
```
(load "myfile")
```

The .lsp extension is not required. This format works for any LSP file in the current library path (see “Library Search Path” on page 10).

To load an AutoLISP file that is not in the library path, you must provide the full path and file name as the `filename` argument. When specifying a directory path, you must use a slash (/) or two backslashes (\\) as the separator, because a single backslash has a special meaning in AutoLISP.

Command:  
```
(load "d:/files/morelisp/newfile")
```
Automatic Loading and Execution of AutoLISP Routines

As you build a library of useful AutoLISP routines, you may want to load them each time you run AutoCAD. You may also want to execute certain commands or functions at specific times during a drawing session.

AutoCAD loads the contents of three user-definable files automatically: acad.lsp, acaddoc.lsp, and the .mnl file that accompanies your current menu. If one of these files defines a function of the special type S::STARTUP, this routine runs immediately after the drawing is fully initialized. The S::STARTUP function is described in “S::STARTUP Function—Post-Initialization Execution” on page 160. As an alternative, the APPLOAD command provides a Startup Suite option that loads the specified applications without the need to edit any files.

The acad.lsp and acaddoc.lsp startup files are not provided with AutoCAD. It is up to the user to create and maintain these files.

acad.lsp—Automatic Loading of AutoLISP

The acad.lsp file is useful if you want to load specific AutoLISP routines every time you start AutoCAD. When you start AutoCAD, the program searches the library path for an acad.lsp file. If it finds one, it loads the file into memory.

The acad.lsp file is loaded at each drawing session startup when AutoCAD is launched from the Windows desktop. Because the acad.lsp file is intended to be used for application-specific startup routines, all functions and variables defined in an acad.lsp file are only available in the first drawing. You will probably want to move routines that should be available in all documents from your acad.lsp file into the new acaddoc.lsp file.

The recommended functionality of acad.lsp and acaddoc.lsp can be overridden with the ACADLSPASDOC system variable. If the ACADLSPASDOC system variable is set to 0 (the default setting), the acad.lsp file is loaded just once; upon application startup. If ACADLSPASDOC is set to 1, the acad.lsp file is reloaded with each new drawing.

The ACADLSPASDOC system variable is ignored in SDI (single document interface) mode. When the SDI system variable is set to 1, the LISPINIT system variable controls reinitialization of AutoLISP between drawings. When LISPINIT is set to 1, AutoLISP functions and variables are valid in the current drawing only; each time you start a new drawing or open an existing one, all functions and variables are cleared from memory and the acad.lsp file is reloaded. Changing the value of LISPINIT when the SDI system variable is set to 0 has no effect.
The *acad.lsp* file can contain AutoLISP code for one or more routines, or just a series of *load* function calls. The latter method is preferable, because modification is easier. If you save the following code as an *acad.lsp* file, the files *mysessionapp1.lsp, databasesynch.lsp*, and *drawingmanager.lsp* are loaded every time you start AutoCAD.

```auto-lisp
(load "mysessionapp1")
(load "databasesynch")
(load "drawingmanager")
```

For additional information about coding in startup files, see “Coding Tips for AutoLISP Startup Files” on page 159.

**NOTE** Do not modify the reserved *acad2000.lsp* file. Autodesk provides the *acad2000.lsp* file, which contains AutoLISP defined functions that are required by AutoCAD. This file is loaded into memory immediately before the *acad.lsp* file is loaded.

---

**acaddoc.lsp—Automatic Loading of AutoLISP**

The *acaddoc.lsp* file is intended to be associated with each document (or drawing) initialization. This file is useful if you want to load a library of AutoLISP routines to be available every time you start a new drawing (or open an existing drawing). Each time a drawing opens, AutoCAD searches the library path for an *acaddoc.lsp* file. If it finds one, it loads the file into memory. The *acaddoc.lsp* file is always loaded with each drawing regardless of the settings of ACADLSPASDOC and LISPINIT.

Most users will have a single *acaddoc.lsp* file for all document-based AutoLISP routines. AutoCAD searches for an *acaddoc.lsp* file in the order defined by the library path; therefore, with this feature, you can have a different *acaddoc.lsp* file in each drawing directory, which would load specific AutoLISP routines for certain types of drawings or jobs.

The *acaddoc.lsp* file can contain AutoLISP code for one or more routines, or just a series of *load* function calls. The latter method is preferable, because modification is easier. If you save the following code as an *acaddoc.lsp* file, the files *mydocumentapp1.lsp, build.lsp*, and *counter.lsp* are loaded every time a new document is opened.

```auto-lisp
(load "mydocumentapp1")
(load "build")
(load "counter")
```

AutoCAD searches for an *acaddoc.lsp* file in the order defined by the library path; therefore, you can have a different *acaddoc.lsp* file in each drawing.
directory. You can then load specific AutoLISP routines for certain types of drawings or jobs.

For additional information about coding in startup files, see “Coding Tips for AutoLISP Startup Files” on page 159.

**NOTE** Do not modify the reserved acad2000doc.lsp file. Autodesk provides the acad2000doc.lsp file, which contains AutoLISP-defined functions that are required by AutoCAD. This file is loaded into memory immediately before the acaddoc.lsp file is loaded.

**acad.mnl**—Automatic Loading of AutoLISP Menu Functions

The other type of file that AutoCAD loads automatically accompanies your current menu file and has the extension .mnl. When AutoCAD loads a menu file, it searches for an MNL file with a matching file name. If it finds the file, it loads the file into memory.

This function ensures that AutoCAD loads the AutoLISP functions that are needed for proper operation of a menu. As an example, the standard AutoCAD menu, acad.mnu, relies on the file acad.mnl being loaded properly. This file defines numerous AutoLISP functions used by the menu. The MNL file is loaded after the acaddoc.lsp file.

**NOTE** If a menu file is loaded with the AutoLISP command function (with syntax similar to (command "menu" "newmenu")), the associated MNL file is not loaded until the entire AutoLISP routine has run.

For example, if you create a custom menu called newmenu.mnu and you need to load three AutoLISP files (new1.lsp, new2.lsp, and new3.lsp) for the menu to work properly, you should create an ASCII text file named newmenu.mnl as follows:

```lisp
(load "new1")
(load "new2")
(load "new3")
(princ "$Newmenu utilities... Loaded.")
(princ)
```

In this example, calls to the prin function can be used to display status messages. The first use of prin displays the following on the command line:

Newmenu utilities... Loaded.
The second call to `princ` exits the AutoLISP function. Without this second call to `princ`, the message would be displayed twice. As mentioned previously, you can include the `onfailure` argument with calls to the `load` function as an extra precaution.

**Coding Tips for AutoLISP Startup Files**

If an AutoLISP error occurs while you are loading a startup file, the remainder of the file is ignored and is not loaded. Files specified in a startup file that do not exist or that are not in the AutoCAD library path generally cause errors. Therefore, you may want to use the `onfailure` argument with the `load` function. The following example uses the `onfailure` argument:

```
(princ (load "mydocapp1" "\nMYDOCAPP1.LSP file not loaded."))
(princ (load "build" "\nBUILD.LSP file not loaded."))
(princ (load "counter" "\nCOUNTER.LSP file not loaded."))
(princ)
```

If a call to the `load` function is successful, it returns the value of the last expression in the file (usually the name of the last defined function or a message regarding the use of the function). If the call fails, it returns the value of the `onfailure` argument. In the preceding example, the value returned by the `load` function is passed to the `princ` function, causing that value to be displayed on the command line. For example, if an error occurs while AutoCAD loads the `mydocapp1.lsp` file, the `princ` function displays the following message and AutoCAD continues to load the two remaining files:

```
MYDOCAPP1.LSP file not loaded.
```

If you use the `command` function in an `acad.lsp`, `acaddoc.lsp` or MNL file, it should be called only from within a `defun` statement. Use the `S::STARTUP` function to define commands that need to be issued immediately when you begin a drawing session. The `S::STARTUP` function is described in “S::STARTUP Function—Post-Initialization Execution” on page 160.

**Command Autoloader**

When you automatically load a command using the `load` or `command` functions, the command’s definition takes up memory whether or not you actually use the command. The AutoLISP autoloader function makes a command available without loading the entire routine into memory. Adding the following code to your `acaddoc.lsp` file automatically loads the commands CMD1, CMD2, and CMD3 from the `cmds.lsp` file and the NEWCMD command from the `newcmd.lsp` file.

```
(autoload "CMDS" '("CMD1" "CMD2" "CMD3"))
(autoload "NEWCMD" '("NEWCMD"))
```
The first time you enter an automatically loaded command at the Command prompt, AutoLISP loads the entire command definition from the associated file. AutoLISP also provides the `autoarxload` function for ObjectARX applications. See “autoload” and “autoarxload” in the AutoLISP Reference.

**S::STARTUP Function—Post-Initialization Execution**

The startup LISP files (acad.lsp, acaddoc.lsp, and .mnl) all load into memory before the drawing is completely initialized. Typically, this does not pose a problem, unless you want to use the `command` function, which is not guaranteed to work until after a drawing is initialized.

If the user-defined function `S::STARTUP` is included in an acad.lsp, acaddoc.lsp or a .mnl file, it is called when you enter a new drawing or open an existing drawing. Thus, you can include a definition of `S::STARTUP` in the LISP startup file to perform any setup operations.

For example, if you want to override the standard HATCH command by adding a message and then switching to the BHATCH command, use an acaddoc.lsp file that contains the following:

```lisp
(defun C:HATCH ()
  (alert "Using the BHATCH command!"
  (princ "Enter OLDHATCH to get to real HATCH command.\n"
  (command "BHATCH"
  (princ
)
(defun C:OLDHATCH ()
  (command ".HATCH"
  (princ
)
(defun-q S::STARTUP ()
  (command "undefine" "hatch"
  (princ "\nRedefined HATCH to BHATCH!\n"
  )

Before the drawing is initialized, new definitions for HATCH and OLDHATCH are defined with the `defun` function. After the drawing is initialized, the `S::STARTUP` function is called and the standard definition of HATCH is undefined.

**NOTE** To be appended, the `S::STARTUP` function must have been defined with the `defun-q` function rather than `defun`.

Because an `S::STARTUP` function can be defined in many places (an acad.lsp, acaddoc.lsp, .mnl file, or any other AutoLISP file loaded from any of these), it’s possible to overwrite a previously defined `S::STARTUP` function. The following example shows one method of ensuring that your start-up function works with other functions.
(defun-q MYSTARTUP ()
  ... your start-up function ...
)
(setq S::STARTUP (append S::STARTUP MYSTARTUP))

The previous code appends your start-up function to that of an existing
S::STARTUP function, and then redefines the S::STARTUP function to include
your start-up code. This works properly regardless of the prior existence of an
S::STARTUP function.

ObjectARX

ObjectARX (AutoCAD Runtime Extension) is a compiled-language pro-
gramming environment for developing AutoCAD applications. The
ObjectARX programming environment includes a number of dynamic link
libraries (DLLs) that run in the same address space as AutoCAD and operate
directly with core AutoCAD data structures and code. These libraries take
advantage of the AutoCAD open architecture, providing direct access to the
AutoCAD database structures, graphics system, and AutoCAD geometry
engine to extend AutoCAD classes and capabilities at runtime. Addition-
ally, you can use DLLs to create new commands that operate exactly the
same way as native AutoCAD commands.

You can use ObjectARX libraries in conjunction with other AutoCAD pro-
gramming interfaces, such as AutoLISP or VBA, enabling cross-API
integration.

The ObjectARX programming language is described in the ObjectARX™
Developer's Guide. The documentation is part of the ObjectARX SDK, which is
available from the Autodesk Web site.

Using ObjectARX Applications

To load an ObjectARX application, you use the Load option of the ARX com-
mand. After loading, all commands defined by this application are available
at the Command prompt.

Some ObjectARX applications use large amounts of system memory. If you
are finished using an application and want to remove it from memory, use
the ARX command's Unload option.

You can also load an ObjectARX application with the arxload AutoLISP func-
tion. The syntax for the arxload function is almost identical to that of the
load function used with AutoLISP files. If the arxload function loads the
ObjectARX program successfully, it returns the program name. The syntax for the \texttt{arxload} function is as follows:

\begin{verbatim}
(arxload filename [onfailure])
\end{verbatim}

The two arguments for the \texttt{arxload} function are \texttt{filename} and \texttt{onfailure}. As with the \texttt{load} function, the \texttt{filename} argument is required and must be the complete path name description of the ObjectARX program file to load. The \texttt{onfailure} argument is optional and typically not used when you load ObjectARX programs from the command line. The following example loads the ObjectARX application \texttt{myapp.arx}.

\begin{verbatim}
(arxload "myapp")
\end{verbatim}

As with AutoLISP files, AutoCAD searches the library path for the specified file. If you need to load a file that is not in the library path, you must provide the full path name description of the file. Be sure to use a single forward slash (/) or two backslashes (\) as the directory separator when specifying a full path name.

Attempting to load an application that has previously been loaded results in an error. Before using \texttt{arxload} you should use the \texttt{arx} function to check the currently loaded applications.

To unload an application with AutoLISP, use the \texttt{arxunload} function. The following example unloads the \texttt{myapp} application.

\begin{verbatim}
(arxunload "myapp")
\end{verbatim}

Using the \texttt{arxunload} function not only removes the application from memory but also removes the command definitions associated with that application.

**Automatic Loading of ObjectARX Applications**

The \texttt{acad.rx} file contains a list of the ObjectARX program files that are loaded automatically when you start AutoCAD. You can edit this file with a text editor or word processor that produces files in ASCII text format. You can customize this file as you want, adding to or deleting from its contents and making the appropriate ObjectARX programs available for use. As an alternative, the APPLOAD command provides a Startup Suite option that loads the specified applications without the need to edit any files.
Because AutoCAD searches for the acad.rx file in the order specified by the library path, you can have a different acad.rx file in each drawing directory. This makes specific ObjectARX programs available for certain types of drawings. For example, you might keep 3D drawings in a directory called AcadJobs/3d_dwgs. If that directory is set up as the current directory, you could copy the acad.rx file into that directory and modify it in the following manner:

myapp1
otherapp

If you place this new acad.rx file in the AcadJobs/3d_dwgs directory and you start AutoCAD with that as the current directory, these new ObjectARX programs are then loaded and are available from the AutoCAD prompt line. Because the original acad.rx file is still in the directory with the AutoCAD program files, the default acad.rx file will be loaded if you start AutoCAD from another directory that does not contain an acad.rx file.

You can load ObjectARX programs from an .mnl file using the arxload function. This ensures that an ObjectARX program, required for proper operation of a menu, will be loaded when the menu file is loaded.

You can also autoload many ObjectARX-defined AutoCAD commands (see “Command Autoloader” on page 159 and “autoarxload” in the AutoLISP Reference).
Index

A
acad.bat (batch program) file, 13
acad.bhp (WinHelp) file, 22
acad.pat (default pattern) file, 36
acad.pgp (program parameters) file, 16
changes to, 6
command alias section, 19
acad.psf (AutoCAD PostScript support) file, 64
acad15.cfg (default configuration) file, 11
acadps.arx (AutoCAD PostScript interpreter) file, 161
accelerator keys definitions, 123–126
ActiveX Automation
controllers, 146–147
invoking a command script, 144–145
programming interface, 146–147
running stand-alone executables with, 147
starting from a menu, 148–149
starting from the command line, 147–148
using with VBA, 150–151
Adobe fonts, 65
AHP. See AutoCAD Help files
aliases, for commands, 19
and DIESEL string function, 137
anngos DIESEL string function, 137
APPLOAD command, 155
ARX command, Load option, 161
arxload AutoLISP function, 161
arxunload AutoLISP function, 162
ASCII
code conversion, 166
text files, 10
autoarxload AutoLISP function, 160
AutoCAD
changes and enhancements, 6
customization features, 3
doing and invoking a script, 144
pattern files, 36
PostScript support file, 64–69
shape font files mapped to PostScript, 65
version compatibility, 144
AutoCAD 2000 Learning Assistance, 2
AutoCAD environment
command search procedure, 13
customizable support files, 14
library search path, 10
multiple configurations, 11
multiple drawing directories, 12
sample directory structure, 11
AutoCAD Help files
AHP format for converting files, 21
augmenting system, 21
converting obsolete, 27
AutoCAD VBA
automatic loading and executing of VBA projects, 152–153
developing with, 150–151
interface with AutoCAD, 150–151
programming environment, 149–150
using VBA applications, 152
AutoLISP, 6, 150–161
about applications, 155
text files, 10
AutoLISP functions
arxload, 161–162
arxunload, 162
autoarxload, 160
autoarxload, 159
AutoLISP functions (continued)
command, 159
defun, 160
getxxx, 132
help, 24
load, 130, 155
menucmd, 78, 103
prin, 159
S::STARTUP, 160
setfunhelp, 24
startapp, 24, 26, 148
strcat, 130
See also AutoLISP Reference
AutoLISP Reference, 2
autoload AutoLISP function, 159
Automation
See ActiveX Automation
Automation. See ActiveX Automation auxiliary menus. See menus, Buttons and Aux
B
backing up files, 10
base menus, 77
batch files
running, 19
Big Fonts
defining, 54–58
using, 59–61
bitmap resource DLLs, 75
block definition, 19
Boundary Hatch dialog box, 36
bulge-specified arcs, 50–51
button menus. See menus, Buttons and Aux
C
canceling commands, 86
caret (^), mapping to CTRL key, 86
CHANGE command, in menu macros, 85
special characters
in hatch patterns, 36
in menu labels, 99–100
cmd (Windows system command), 18
.cnt file name extension, 21
CNT files, 21
command
aliases, 19–20
autoloader in AutoLISP, 159
input areas, 121–122
scripts, 144
search procedure, 13–14
command AutoLISP function, 159
command line
running a VBA macro, 152
running WinHelp, 24
command line (continued)
switches
/c, 18
/k, 18
commands
$M=, 89
APLOAD, 155
ARX, 161
BHATCH, 160
canceling, 86
CHANGE, 85
COMPILE, 42, 45
defining aliases for, 19–20
defining external, 16–19
DIM, 83
DTEXT, 42
DXFIN, 6
dxfout, 6
ERASE, 88
executing with scripts, 144
foreign-language support, 85
HATCH, 160
INSERT, 42
LAYER, 84
MENU, 75, 77
MENULOAD, 77
MENUUNLOAD, 75
MTEXT, 42
PAN, 115
PSOUT, 64
repeating in menu macros, 87
SCRIPT, 144
search procedure, 13
SELECT, 85
SHAPE, 42, 45
STYLE, 42, 52, 59
TEXT, 42, 83
TOOLBAR, 111
UNDO, in scripts, 144
validating names, 21
VSLIDE, 112–113
Windows system, 18
comments
in menu files, 80
in script files, 144
compiled
menu file, 75
WinHelp file (.hlp), 22
compiling
PostScript fonts, 43
shape/font files, 42
WinHelp files, 21
complex linetypes
shapes in, 33
syntax for, 32
text in, 34–35
configurations, multiple, 11
control characters, menu items, 86
custom menus
  base and partial menus, 77
  controlling with AutoLISP, 78–79
bitmap resource DLLs, 75
loading menu files, 75–77
menu file types, 74
using menu files, 74
customizable support files, 14
Customization Guide, changes to, 2

D

dashed-line hatch patterns. See hatch patterns
date information
  editing, 138–139
  entering, 134
default configuration file, 11
defun AutoLISP function, 160
dictionary objects, changes to, 6
DIESEL string expression language
  in AutoLISP routines, 135
  in conditional expressions, 89
  disabling labels, 102
  error messages, 142
  marking labels, 102
  in menu files, 132–134
  modifying the status line, 127
DIESEL string functions, 136
  - (subtraction), 135
  != (not equal to), 137
  * (multiplication), 136
  + (addition), 135
  / (division), 136
  < (less than), 136
  <= (less than or equal to), 137
  = (equal to), 136
  > (greater than), 136
  >= (greater than or equal to), 137
  and, 137
  angtos, 137
  changes to, 6
  edtime, 138
  eq, 139
  eval, 139
  fix, 139
  getenv, 140
  getvar, 140
  if, 140
  index, 140
  linlen, 140
  nth, 141
  or, 141
  rtostr, 141
  strlen, 141
  substr, 141
  upper, 142
  xor, 142

DIM command, 83
directories
  creating for custom applications, 11
  sample structure, 11
DLL bitmap resource, 75
DOS environment settings, 19
DXF files, changes to, 6
DXF group codes. See DXF Reference, 2
DXF Reference, 2
DXF, documentation, 2
DXFIN command, 6
DXFOUT command, 6

E

erasing in AutoLISP, 133
error checking in AutoLISP, 133
error messages
  DIESEL string expression language, 142
  PostScript, 71–72
eval DIESEL string function, 139
external commands, 19
EXTNAMES system variable, 6

F

file name extensions
  .cnt, 21
  .hpj, 21
  .lin, 30
  .lsp, 155
  .pat, 36
  .pfb, 65
  .scr, 144
  .shp, 42
  .shx, 33, 65
  for customizable support files, 14
  in support files, 14
files
  $cmd.dxb file, 18, 19
  acad.hlp file, 22, 23
  acad.lin file, 30
  acad.lsp file, 156–159
  acad.mnl file, 158–159
  acad.pat file, 36
  acad.pgp file, 19
  acad.pdf file, 64–69
  acad.rx file, 162
  acad.doc.lsp file, 157
  acadpsl.arx file, 70
  acadpsl.ps file, 71
  acadlpsl.ps file, 71
  acadlpsl.ps file, 71

Index | 171
fill patterns, defining in PostScript, 64
Find tab, in WinHelp, 21
fix DIESEL string function, 139
font files
  applying expansion, compression
  obliquing, 52–53
  compiling, 42–43
  creating, 52
fonts, 52
  Big Font descriptions
    alternate uses, 60
    defining extended, 54–58
    using, 59–60
  compiling, 42–43
  creating, 52
  mapping to PostScript, 65
  sample files, 43
  subscript and superscript, 63–64
  substitution in PostScript, 64, 65
  text font descriptions, 52–53
foreign-language menus, 85
FTG files, 22
FTS files, 22
G
getenv DIESEL string function, 140
getvar DIESEL string function, 140
getxxx AutoLISP functions, 132
GRIPSIZE system variable, 91
H
hatch patterns
  AutoCAD pattern file, 36
  character limits, 36
  constructing, 37
  dashed lines, 37–39
  default, 36
  defining, 36
  header line, 36
Help
  status line, 122–123
  system components, 21–22
  See also WinHelp files
help AutoLISP function, 24
Help Contents tab
  adding HTML files, 25
  adding topics to, 21
  WinHelp files, 21
Help Topics dialog box, 21
Helpstrings section syntax, 122–123
.hpj file name extension, 21
HPI files, 21
HTML files
  adding to Contents tab, 25
  documentation in, 25
  starting browser from, 26
  starting from AutoLISP, 26
  starting from command line, 26
  using with AutoCAD, 25
I
if DIESEL string function, 140
Image tile menus. See menus, Image tile
importing PostScript images
  acadps, 70
  errors, 71
  fontmap.ps, 71
  PSIN command, 70
index DIESEL string function, 140
Internet
  documentation, 25
  interpreter initialization files, 71
  ISO 8859 Latin/1 character set, 66
J
Japanese characters. See Kanji characters
K
Kanji characters, 56–58
keywords, foreign-language support, 85
L
LAYER command, 84
layers, new settings, 6
library search path, support files, 10
.lin file name extension, 30
LIN files, 30
line family, in hatch patterns, 37
line feed character, 53
linetype definition files
complex, 32–35
defining shapes, 33–34
defining text, 34–35
creating and modifying, 30
default, 30
file name extension, 30, 32
simple, 31–32
linetypes
complex, 33–35
creating and modifying, 30
defining, 30–31
simple, 31–32
linen
DIESEL string function, 140
load AutoLISP function, 130
loading
AutoCAD and invoking a script, 144–145
AutoLISP applications, 155
menu files, 75–77
ObjectARX applications, 161–162, 162–163
partial menus, 78
VBA projects, 152–153
lsp file name extension, 155
LSP files, 155

M
macros
creating with DIESEL, 132–134
VBA, 152
See also menu macros
mapping key sequences
to command strings, 123
virtual keys, 124
menu files
base and partial, 77–79
bitmap resource DLLs, 75
comments in, 80
definition, 75
DIESEL string in, 132–134
file types, 74
loading, 75–77
menu item syntax, 80
menu macros, 82–91
reloading, 77
section labels, 79
structure, 79–80
types, 74–75
menu groups, 91–92
menu item syntax, 80
labels, 81
menu macros, 82–91
canceling a command, 86
conditional expressions, 89, 89–90
control characters, 86–87
menu item syntax (continued)
echoes and prompts, 86
foreign-language support, 85
long, 87
menu swapping, 88, 88–89
pausing for user input, 84–85
repetition, 87
sequence of, 82
single object selection mode, 88
syntax for, 82
termination, 83–84
using AutoLISP, 90–91
name tags, 80
menu items, 80
activating and deactivating, 104–105
DIESEL string in, 132–134
disabling labels, 102
image section, 80
labels, 81, 120–121
marking, 102
menu macros, 82
name tags, 80–81
separating labels, 101
See also menu item syntax
menu labels
altering display, 133
changing width in DIESEL, 134
functionality in menu items, 81
for Image tile items, 112–113
marking, 102–103
marking using DIESEL, 134
pull-down and shortcut menus, 99–100
separating in menu items, 101
menu LISP file, 75
menu macros, 82
canceling a command, 85
conditional macro expressions, 89–90
control characters, 86
echoes and prompts, 86
foreign language support, 85
loading Image tiles, 113
long, 87
pausing for user input, 84–85
referencing submenus, 89
repetition, 87
single option selection mode, 88
special characters, 82–83
swapping menus, 88–89
syntax, 82–83
terminating, 83–84
terminators, 90
using AutoLISP, 90–91
menucmd AutoLISP function, 78, 103
MENUCTL system variable, 121
menus
areas, 74
auxiliary, 92–96
base and partial, 77
button, 92–96
menus (continued)

Buttons and Aux
 creating, 92–94
 replacing AUX menu section, 88–89
 replacing, 88–89
 swaping, 94–95
 changing display in DIESEL, 133
 changing text in DIESEL, 134
 controlling display and layout, 91–92
 customizing, 74
 file structure, 79–80
 foreign-language, 85
 group name functionality, 91–92
 Image tile
 activating, 114
 displaying, 113
 labels, 112–113
 menu macros, 113
 preparing slides for, 115
 purpose, 111
 sample submenu, 114–115
 menu item syntax, 80
 pull-down and shortcut, 96–107
 rearranging, 77
 removing, 78–79
 screen, 116–122
 status line help messages, 122–123
 tablet, 121–122
 areas for command input, 121–122
 creating, 121–122
 replacing section, 88–89

menus, pull-down
 cassing submenus, 101
 creating, 97–98
 inserting and removing, 106–107
 label status, 105
 limits, 96
 marking labels with DIESEL, 134
 menu bar titles, 100–101
 menu label syntax, 99–100
 separating menu item labels, 101
 special label characters, 99–100
 swapping, 106
 syntax for, 97–98

menus, screen
 controlling swapping, 121
 creating, 116
 item labels, 120–121
 recalling main, 119
 replacing section, 88–89
 sample, 118–119
 section label, 116
 submenus
 format, 117–120

menus, shortcut
 access to, 96
 bitmap resource DLLs, 75
 cascading submenus, 101
 changes to, 7
 creating, 97–98
 label status, 105
 limits, 96
 menu bar titles, 100–101
 menu label syntax, 99–100
 referencing menu items, 103–105
 separating menu item labels, 101
 special label characters, 99–100
 syntax for, 97–98

MNC files, 75
 MNL files, 75
 in menu macros, 90
 MNR files, 75
 MNS files, 75
 MNU files, 75
 mode status line. See status line
 MODEMACRO system variable
 defining status line, 129–131
 using with AutoLISP, 130–131
 mouse button menus, 92–94
 multiple configuration files, 111
 multiple drawings
 changes and enhancements, 6
 creating directories, 12
 opening, 6

N
 name tags, in menu items, 80–81
 nth DIESEL string function, 141

O
 ObjectARX
 applications, 161–162
 loading automatically, 162–163
 programming environment, 161
 objects, single selection mode, 88
 octant arc, in shape definition files, 49
 online documentation, 2
 customizing, 20–21
 HTML files, 25–27
 WinHelp files, 21–25
 or DIESEL string function, 141

P
 PAN command, for centering an image, 115
 partial menus, 77
 loading, 78
 syntax for, 78
 unloading, 78–79
Index

. pat file name extension, 36
patterns. See hatch patterns
pausing for user input in menu macros, 84–85
.pb file name extension, 65
PFB files, 65
pick button, 93
PICKADD system variable, 82
PICKAUTO system variable, 82
pointing devices, 92–94
polyline fill, in PostScript, 66–67
Pop menu section, 88–89
PostScript
compiling fonts, 43
customizing output, 69–70
defining Type 1 fonts, 65
documentation, 72
embedding in Encapsulated PostScript files, 66
encoding structure from AutoCAD drawings, 69–70
exporting images, 64, 66–68
.acad.p5f support file, 64–69
advanced PostScript processing, 69–70
color changes, 69
creating custom prolog section, 69
defining fill patterns, 67–68
embedding EPS files, 66
filling polylines, 66
Latin/1 font re-encoding, 66
layer changes, 69
linetype changes, 70
mapping fonts, 65
PSPROLOG, 69
special processing, 69–70
importing images, 70–72
AutoCAD font map file, 71
AutoCAD interpreter, 70–71
errors, 71
interpreter initialization files, 71
PSFILL command, 64
PSOUT command, 64
rendering images into AutoCAD objects, 70–71
princ AutoLISP function, 159
program parameters file, defining external commands, 16
programming interfaces
ActiveX Automation, 146–149
AutoCAD VBA, 149–153
AutoLISP and Visual LISP, 153–161
command scripts, 144–146
ObjectARX, 161–163
prompts
in menu macros, 91
suppressing, 86
properties, viewing and modifying, 6
pull-down menus. See menus, pull-down
R
referencing pull-down and shortcut menus, 103–105
RTF (Rich Text Format) files, for creating WinHelp, 21
rots DIESEL string function, 141
S
S::STARTUP AutoLISP function, 160
sample directory structure, 11
.scr file name extension, 144
SCR files, 144
screen menus. See menus, screen scripts. See command scripts
SDI system variable, 6
SELECT command, in menu macros, 85
selection set enhancements, 6
separator lines for menu item labels, 101
setfunhelp AutoLISP function, 24
shape definition files
Big Font descriptions, 54–61
compiling, 42
creating, 43
creating additional forms, 46–51
defining vector length and direction code, 45–46
Draw mode, 47
file name extensions, 43
header lines, 44
sample, 43
shape descriptions, 44
specification byte codes, 46–51
subscript and superscript, 63–64
text font descriptions, 52–53
Unicode font descriptions, 61–62
using, 42
vector length and direction code, 45–46
shapes
creating and loading shape files, 42
defining in complex linetypes, 33–34
describerning, 44
sample shape files in AutoCAD, 43
shortcut menus. See menus, shortcut
.shp file name extension, 42
SHP files, 42
.shx file name extension, 33, 65
SHX files, 33, 65
size control, in shape definition files, 47
slides
combining multiple, 112–113
preparing for Image tile menus, 115
source menu file, 75
special characters
accented, 61–62
ASCII codes, 165
extended, 165

175
special characters (continued)
length enhancement, 6
limits in menu macros, 82
non-ASCII, 54
in shape definition files, 44
used in menu macros, 82
specification byte codes, in shape definition files, 46–51
spreadsheets, invoking, 16
start command, Windows system, 18
startapp AutoLISP function, 24, 26, 148
startup routines in AutoLISP, 150–160
status line
configuring, 128–131
defining and loading strings, 130–131
help, 122–123
user-defined, 128–131
strcat AutoLISP function, 130
strlen DIESEL string function, 141
submenus
items allowed, 117
menuname string, 117
referring in menu macros, 89
sample screen menu section, 118
submenu section label, 117
subscript fonts, 63–64
subshapes, in shape definition files, 48
substr DIESEL string function, 141
superscript fonts, 63–64
support files
backing up, 10
customizable, 14
list of, 14
PostScript, 64–70
search order, 10
swapping menus
button and auxiliary, 94–95
syntax for, 88–89
symbol table objects, changes to, 6
system commands, 18
system variables
GRIPSIZe, 91
MENUCTL, 121
MODEMACRO, 128–131
PICKADD, 82
PICKAUTO, 82
PSPROLOG, 69
text (continued)
using DIESEL to change, 134
vertical, 51
text font descriptions
line feed control character, 53
in shape definition files, 52–53
syntax for, 53–54
text objects, converting to lines, 19
thumbnail images, creating, 6
toolbars
bitmap resource DLLs, 75
button, 108–109
creating, 107–111
enhancements, 7
flyout control options, 109–110
name, 108
special control element, 110–111
syntax for referencing, 111
user-defined bitmaps, 111
Type 1 fonts, referencing, 65
typographical conventions, 5
U
UNDO command, in scripts, 144
Unicode font descriptions, syntax for, 62
unloading partial menus, 78
upper DIESEL string function, 142
user input, menu macros, 84–85
user-definable documentation, 20–21
user-supplied programs, 16
V
VBA. See AutoCAD VBA
vectors, length and direction codes, 45–46
virtual keys, 124
Visual Basic for Applications (VBA). See AutoCAD VBA
Visual LISP and AutoLISP, 153–155
documentation, 2
enhancements, 6
Visual LISP Developer’s Guide, 2
Visual LISP Tutorial, 2
VSLIDE command, 112–113
W
Web browser, starting from HTML file, 26
Web pages, adding links to Contents tab, 25
What’s New, 2
Windows, program utilities, 16
WinHelp files
components, 21–22
Contents tab, 21
running from AutoLISP, 24
running from command line, 24
using with AutoCAD, 21, 23
X
xor DIESEL string function, 142