Chapter 1

Writing Your First Program

Your First Program

This chapter presents the first Dialog Box program that you will create. Designing and implementing Dialog Box programs involves two steps: The visual DCL programming step and the AutoLISP code programming step.

Visual DCL Programming Step

Dialog boxes are written in the Dialog Control Language (DCL). In this step, you use a text editor to create an ASCII text file with the .DCL extension. This file defines the appearance and content of the dialog box.

Size and positioning of the information contained within the dialog box are controlled automatically. A minimum amount of positioning information is required.

AutoLISP Code Programming Step

The DCL file defines the parts of the dialog box. An application is required to control the use and behavior of the dialog box. Both AutoLISP and ADS provide functions to program and control dialog boxes. This book focuses on using AutoLISP to control the dialog boxes. AutoLISP provides a package of functions known as the Programmable Dialogue Box (PDB) facility. In this step, you use a text editor to create an ASCII text file with the .LSP extension which contains the PDB functions.

Creating a Working Directory

Before writing the actual programs, create a subdirectory to store the .DCL and .LSP files. From this point, we assume that you have created a directory called C:\PROGRAMS on your local drive. All files will be saved in this directory.

To create the working directory:

- ☐ At the DOS prompt C:\> type **CD\ <return>**

- At the DOS prompt C:\> type **MD PROGRAMS <return>**
- Or use the File Manager program of windows to create the working directory.

The Hello Program

A DCL file contains the description and layout of the dialog box on all platforms. Differences in appearance depend on the platform's graphical user interface (GUI). Executing the Hello program does the following:

- Typing Hello at the AutoCAD Command prompt displays the dialog box as shown in Figure 1.1. This dialog contains three command buttons (Display Hello, Clear, and OK) and an empty text tile.

Figure 1.1. The Hello program.

- Selecting the *Display Hello* button, displays the text Hello World! in the text tile (see Figure 1.2).

Figure 1.2. Displaying the text inside the text tile.

- To clear the text, select the *Clear* button.
- Select *OK*, to exit the program.

Creating the Program Files

Now that you know what the Hello program is supposed to do, let's write the program.

The very first thing you must do is write the code for the DCL file and the program code for the AutoLISP file. This is accomplished in several ways. You may choose to use the DOS editor outside of AutoCAD or from within AutoCAD. If you are using the windows version of AutoCAD, you may use the Notepad, Write, Word or another word processor or text editor.

Regardless of the editor you choose, be sure to save the program files in an ASCII or non-document mode. Do not convert any program or DCL file to a document or word processor format. Always save the files in a text format.

From within AutoCAD Release 12 and Release 13 you can type EDIT to start the text editor. If AutoCAD returns Unknown command after typing EDIT, refer to the AutoCAD documentation for adding the EDIT command to your ACAD.PGP file.

When the text refers to starting the text editor, we are referring to starting the editor from within AutoCAD.

When writing the program you will always have at least two files: the dialog .DCL definition file and the AutoLISP .LSP program file. Be sure to save these files in the C:\PROGRAMS directory, whether you are using a DOS test editor or a windows test editor. From this point on we assume that these files are located in this directory and will instruct AutoLISP to look for these files in this location using a path statement `C:/PROGRAMS/` for loading the DCL files and other external data files.

The Visual Implementation of the Hello Program

Use the following steps to create the HELLO.DCL file.

- ☐ Start AutoCAD in the normal manner with a new drawing. This places you inside the AutoCAD drawing editor with a new drawing.
- ☐ Start the text editor.
- ☐ Save the file as HELLO.DCL to the C:\PROGRAMS directory.
- ☐ Write code inside the HELLO.DCL dialog box.
- ☐ Save the .DCL file.
- ☐ Start a new file from within the text editor.
- ☐ Write code inside the LOADDCL.LSP AutoLISP file.
- ☐ Save the file a LOADDCL.LSP and exit the text editor to return to AutoCAD.

- ☐ Load the AutoLISP program LOADDCL using *APPLOAD*.
- ☐ Execute LOADDCL to check the dialog box layout.

This chapter is not a typical chapter. This chapter teaches you the two steps involved in creating and controlling a dialog box. It is the purpose of the following chapters to teach you how to write the code for the DCL and AutoLISP files for specific tiles. Each chapter gives examples of how the DCL code is written for the specific tile, and how the tile is controlled from within an AutoLISP program.

When the dialog box is complete, it should look like the one shown in Figure 1.3.

Figure 1.3. The Hello program dialog box.

The Code Inside the HELLO.DCL File

You may type the code for the DCL file as each step is presented here or type the entire code from the figure shown at the end of this section. The complete code is shown in Figure 1.5.

Type these two comment lines to identify the DCL file and its purpose:

```
// HELLO.DCL
// Display Hello World!
```

Always use comments to identify the DCL file name and how the file operates. This is useful for printing the code and modifying the file at a later date. Comments are preceded by two forward slashes (`//`). Anything that appears after the `//` and the end of the line is ignored. You may also use C language style comments of the form `/* comment text */`. The starting `/*` and ending `*/` can be on separate lines. An example of this using the comments above is:

```
/* HELLO.DCL
   Display Hello World! */
```

Add a blank line after the comments above and type the next line to set the DCL audit level to three (3):

```
dcl_settings : default_dcl_settings { audit_level = 3; }
```

Syntax errors, misuse of attributes or other errors are checked the first time a DCL file is loaded. Errors encountered are written to the file *ACAD.DCE*, an alert box displays, and the DCL file does not load. To correct the error, edit the ACAD.DCE file for possible warnings, error messages, and redundant attribute keys.

Before actual coding of the dialog box, prepare a sketch or an AutoCAD drawing of the dialog to get the basic design, layout and appearance you are looking for. This allows you to layout and revise your design before writing the actual code, which saves considerable time. It is not unusual to spend one to three hours on the design and coding of a dialog.

During development of the DCL file, keep the audit level at **3** and remove the *`dcl_settings`* line before shipping to users. See the Table 1.1 for a definition of the audit levels.

Table 1.1. Semantic auditing levels.

Level	Description
0	No checking. Use if the DCL files have been audited.
1	Errors. Finds DCL bugs that may cause AutoCAD to terminate.
2	Warnings. Finds DCL bugs that result in dialogs with udesired layout and behaviour.
3	Hints. Finds redundant attribute definitions.

Add a blank line then type the next two lines to declare the dialog tile and add a label for the dialog title:

```
hello : dialog {
  label = "The Hello Dialog";
```

This code names the dialog and begins the dialog definition. A colon `:` indicates the beginning of a tile definition. The name of the dialog tile is next after the colon **:** separated by a space. A curly brace indicates the beginning of the dialog definition section. After defining all the tiles within the dialog definition, a closing curly brace ends the dialog definition section. The closing curly brace for the code above will be added after defining the other tiles.

Attributes, such as the `label` attribute above, are separated from their values by an equal sign `(=)` with a space on each side of the equal sign. Attribute lines are always terminated with a semicolon. A missing semicolon or curly brace will cause an error.

DCL Syntax and Format

Proper syntax and format of a DCL file is important to prevent errors and enable readability of the DCL file. This is similar to the formatting of an AutoLISP file. Use indentation to indicate a tile definition indenting the tile's attributes an additional two spaces. Try to format your DCL files as the examples shown in this chapter. Remember to add a semicolon at the end of each attribute and a curly brace at the end of a tile definition.

For additional reference, open some of the AutoCAD DCL files. Do not alter or change these files as this can cause AutoCAD to run incorrectly.

Dialog Tile

A *dialog* tile is the primary tile that defines the dialog box. The dialog box name appears before the dialog tile is declared separated by a space, a colon :, a space, dialog, a space, and a curly brace. A dialog's name is case sensitive and must be called from the LISP program exactly as it appears in the DCL file. The name *hello* is different from the name *Hello* or *HELLO*. All other tile definitions will occur within the curly braces of the *dialog* tile, laid out within a column format.

DCL Syntax Example:

```
hello : dialog {
  label = "The Hello Dialog";
  initial_focus = "cmd_hello";
  tile definitions....
}
```

A dialog box named **hello** is declared, labeled, with the initial focus set to the *cmd_hello* key, and all tiles are listed between the curly braces of the dialog tile definition section. Attributes may be associated with each tile depending on the specific needs of the individual tile. Attribute names are case-sensitive, so attributes named *Label* or *LABEL* will not have an effect on the `label` attribute of the tile.

Any combination of attributes may be included in the tile definition. Attributes specific to a tile are also described after the DCL syntax example. Attributes for the *dialog* tile are described as shown below:

Attributes specific to the dialog tile:

initial_focus *Specifies the key of the tile that receives the initial keyboard focus.*

label *An optional label displayed as a title in the top border of the dialog. The title may be changed at runtime using the* ***set_tile*** *function.*

value *This attribute overrides that of the label and specifics a string to display as the optional dialog box title. Both a label and a value attribute should not be specified.*

A dialog tile is laid out in a column format. Additional tiles within the dialog box description are placed below the previous tile. All other tile descriptions occur within the curly braces of the dialog definition tile section.

Type the next five lines to declare a text tile:

Next, type the following code below the label attribute of the dialog tile, to add the text tile definition. The text tile is considered a child tile definition of the dialog. Children tiles are not named as the dialog tile above, but are preceded by a colon : and the tile attributes contained within the curly braces.

```
: text {
  label = “”;
  key = “txt_hello”;
  alignment = centered;
}
```

A text tile begins with a colon : followed by a space, the tile name *`text`*, followed by a space and the opening curly brace. The `text` tile attributes are added next, followed by the closing curly brace to end the tile definition. The `label` attribute value is left blank so the program can add the required text within the tile. A unique name, *`“txt_hello”`* is assigned to the text tile `key` attribute. The tiles alignment is set to centered.

A *text* tile is one of the informative tiles used to display a text string in the dialog box. Text tiles may be used in combination with the other text tiles *text_part*, concatenation, and paragraph, to display large blocks of text within paragraphs. These tiles are described in later chapters.

TEXT Tile

A *text* tile is a useful way to display feedback about user actions. A static message may be defined in the label attribute or the message may change during runtime by leaving the label blank. A *text* tile's width is determined by the larger of the label attribute or the width attribute. At lease one of these attributes must be specified for the tile width.

DCL Syntax Example:

```
: text {
  label = “This is a text tile”;
  key = “txt_text1”
  alignment = centered;
}
```

If the alignment tile is not specified the default text alignment is left. The value for the key attribute is a quoted string that must be unique for each tile definition within the dialog. Key values used in one dialog definition file may be reused in another dialog definition file.

Text Tile Attributes:

alignment *Values are left, right or centered.*

fixed_height *Values or true or false. If true, the tile does not fill the extra space available in the layout/alignment process.*

fixed_width	*Values or true or false. If true, the tile does not fill the extra space available in the layout/alignment process.*
height	*Specifies the height of the tile in character height units.*
is_bold	*Values are true or false. If true, the text is displayed in bold characters. This attribute is not supported on all platforms.*
key	*A key is a quoted string (a unique name) that the application program uses to reference the tile.*
label	*Optional text displayed within the text tile.*
value	*A string to display in the text tile.*
width	*Specifies the width of the tile in character width units. This is a minimum width and can be expanded during the layout/align ment process.*

Type the next seven lines of code to declare a button within a row:

Tiles within a `row` are placed horizontally as they are encountered within the DCL file between the curly braces, from left to right, of the `row` tile definition. Tiles within the curly braces of the `row` are considered to be the children of the `row` tile. Attributes for the `row` tile apply to the tiles defined within the `row` tile section.

```
: row {
  : button {
   label = “Display Hello”;
   key = “cmd_hello”;
   width = 18;
   mnemonic = “D”;
  }//button
```

A closing curly brace for the *row* tile is added after the next button is declared. A *row* is one of the predefined tile clusters which provides a way of grouping related tiles together. The row tile cluster is described before the next button tile is added.

ROW Tile Cluster

Tile clusters may not be selected themselves, only tiles within the clusters. Clusters do not have assigned actions except *radio_rows* and *radio_columns*. Row attributes apply to the children tiles placed within the rows. Rows may contain any type of tile including columns and other rows.

Attributes described below for the *row* tile cluster also apply to all tile clusters and will not be repeated for the remaining tile cluster descriptions in the following chapters.

DCL Syntax Example:

```
:row {
  fixed_width = true;
  alignment = centered;
  : retirement_button {
    label = "Apply";
    key = "accept";
    is_default = true;
  }
  : spacer { width = 2; }
  cancel_button;
  : spacer { width = 2; }
  : retirement_button {
    label = "Another";
    key = "another";
  }
  : spacer { width = 2; }
  help_button;
}//row
```

The attributes *fixed_width* and *alignment* apply to all the tiles placed within the row tile definition section.

Attributes:

alignment *Values are top, centered, and bottom. This attribute specifies the vertical positing of the child tiles within the cluster.*

children_alignment *Values are top, centered, and bottom. This attribute controlsthe horizontal and vertical positioning when the tiles do not have specific alignment values.*

children_fixed_height *Values are either true or false. Prevents a tile from growing during the layout process. Has no effect on tiles with a specific fixed_height value.*

children_fixed_width *Values are either true or false. Prevents a tile from growing during the layout process. Has no effect on tiles with a specific fixed_width value.*

fixed_height *Values are true or false. Prevents a tile from growing during the layout process. Has no effect on tiles with a specific fixed_height value.*

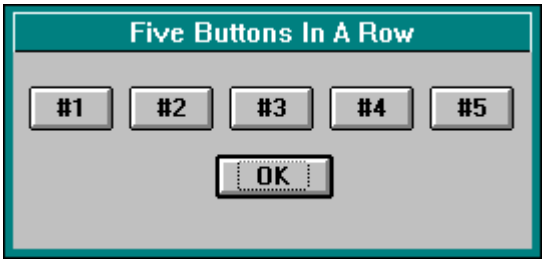
fixed_width *Values are true or false. Prevents a tile from growing during the layout process. Has no effect on tiles with a specific fixed_width value.*

height *Values are an integer or a real. Specifies the tile height in character height units (the height of the screen characters including line spacing).*

width *Values are an integer or a real. Specifies the tile width in character width units.*

Figure 1.4 shows five buttons defined within a row.

Figure 1.4. Buttons within a row.



BUTTON Tile

A **`button`** represents a tile that the user can push to make selections such as exiting or canceling a dialog box. Actions that are immediately visible to the user can be executed using a ***`button`*** tile. All dialogs require at least one button to close or retire the dialog. This can be an OK button or a user defined button equal to an OK button. If this button is omitted, the user is trapped in the dialog and cannot exit until the computer is restarted.

When the user selects a button an associated action is executed. The action is defined in the *`action_tile`* attribute for the button tile.

DCL syntax Example:

```
: button {
  label = “Button #1”;
  key = “cmd_button1”;
}
```

Button Tile Attributes:

action	*Specifies an AutoLISP expression to execute when this tile is selected. Since all tiles can have only one action an action_tile statement in the .LSP file overrides the action attribute in the .DCL file.*
alignment	*Values include left, right and centered for buttons within a column and top, bottom or centered for rows.*
fixed_height	*Values are true or false. If true the button does not fill the extra space during the layout/alignment process.*
fixed_width	*Values are true or false. If true the button does not fill the extra space during the layout/alignment process.*
height	*Specifies the height of the button in character height units.*
is_cancel	*Values are true or false. Specifies whether this button is selected when the user presses Ctrl + C or Esc. The Cancel button is usually assigned this attribute.*
is_default	*Values are true or false. Specifies whether this button is selected when the user presses the Enter key. The OK button is usually assigned this attribute.*
is_enabled	*Values are true or false. If false the tile is initially grayed out.*
is_tab_stop	*Values are true or false. If true the tile receives keyboard focus when the user moves to the tile using the tab key.*
key	*A quoted string (a unique name) to reference the button tile.*
label	*Specifies the text that appears inside the button*

mnemonic *A mnemonic attribute designates a keyboard mnemonic for the tile. The mnemonic character is underlined in the tile's label.*

width *Specifies the width of the button in character width units.*

Type the following six lines of code to declare another button and end the row:

Add the code for the Clear button, then close the row tile cluster adding the closing curly brace.

```
  : button {
    label = "Clear";
    key = "cmd_clear";
    width = 10;
    mnemonic = "C";
  }//button
}//row
```

This ends the two buttons within the *row*. The closing curly brace for the *row* should align with the colon for the *row* tile definition added above.

Type the next line to declare the PDB predefined tile OK:

The OK button may be used alone, as shown below or in a subassembly of OK and Cancel. Most dialogs include an OK and Cancel button. An "information only" dialog generally has a single OK button.

```
ok_only;
```

This tile contains a button with the *label* "OK" and the *key* "accept," and sets an attribute called "is_default" to true. The definition for this predefined tile is stored in the base.dcl file provided by AutoCAD.

Type the next line to add the closing curly brace:

```
}//dialog hello
```

This is the final line of code in the HELLO.DCL file, which describes the layout of the hello dialog. Check your code carefully with the code shown on this page before saving. If the code is correct, save the file and proceed to enter the code for the dialog checking program.

Your code should now appear as shown in Figure 1.5.

Figure 1.5. Hello DCL code.

```
// HELLO.DCL
// Display Hello World!

dcl_settings : default_dcl_settings { audit_level = 3; }

hello : dialog {
  label = “The Hello Dialog”;
  : text   { label = “”;
    key = “txt_hello”;
    alignment = centered;
  }
  : row {
    : button { label = “Display Hello”;
      key = “cmd_hello”;
      width = 18;
      mnemonic = “D”;
    }
    : button { label = “Clear”;
      key = “cmd_clear”;
      width = 10;
      mnemonic = “C”;
    }
  }//row
ok_only;
}//dialog hello
```

After verifying the code above, save the DCL file. The above code was written in the DCL language. A program to check the appearance of the DCL file will now be written in AutoLISP. Start a new file and save it as **LOADDCL.LSP**.

This program is used to check the dialog box appearance and functionality. It is best to check the code for the dialog and its appearance before writing the AutoLISP code to control the dialog box. Use this program to load and check all dialog boxes before writing the AutoLISP program code. Be sure to type this code exactly as it appears or use the code from the diskette. If you encounter any problems, refer to the Chapter on Debugging. The AutoLISP code for this program will not be explained at this time as this program is only for the purpose of loading the DCL file.

Figure 1.6. LOADDCL.LSP Type the code exactly as follows:

```
;;; LOADDCL.LSP
;;; Loads, displays, activates and unloads a dialog box of
;;; the same name. Used to verify dialog box design.

(defun C:LOADDCL (/ DCL_ID DLBNAME)
  (setq DLBNAME (getstring "\nDCL File Name: "))
  (if (findfile (strcat "C:/PROGRAMS/" DLBNAME ".DCL"))
    (progn
    (setq DCL_ID
      (load_dialog (strcat "C:/PROGRAMS/" DLBNAME))
    (if (not (new_dialog DLBNAME DCL_ID)) (exit))

    (action_tile "accept" "(done_dialog)")
    (action_tile "cancel" "(done_dialog)")

    (start_dialog)
    (unload_dialog DCL_ID)
  );;progn
  (alert
    (strcat "Unable to display < " DLBNAME " > Dialog!"))
  );;if
)
```

Now, save the file and exit the text editor to check the dialog box code and design. You should now be returned to AutoCAD. From within AutoCAD, at the command prompt, execute the *Appload* program.

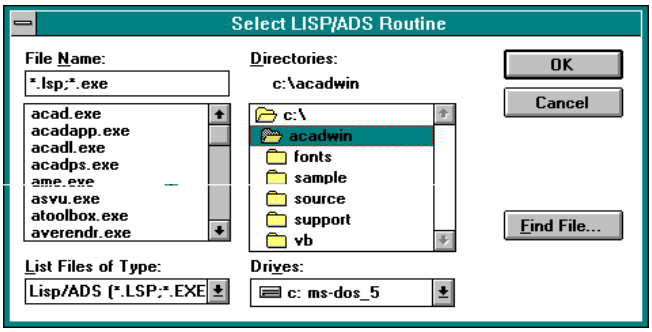
```
Command: APPLOAD <return>
```

The APPLOAD dialog box appears as shown in Figure 1.7.

Figure 1.7. Appload dialog box.

When the dialog first appears it may be empty or list some files in the Files to load: list box. Select the ***File...*** button at the top right to open the File selection dialog Figure 1.8.

Figure 1.8. Appload File Selection dialog.



This dialog displays directories in the list box on the right and files names on the left. Move to the **C:\PROGRAMS** subdirectory and select the file **LOADDCL.LSP** then click **OK**. The dialog closes and the file is now listed and highlighted in the *APP-LOAD* dialog. Select the *Load* button at the bottom left of the dialog. If the *Load* button is grayed out, click on the LOADDCL.LSP file; the *Load* button should now be accessible.

This loads the LOADDCL file and checks it for any errors. If any errors are encountered refer to the Chapter on Debugging. You are now ready to verify the layout and code for the HELLO dialog. At the AutoCAD Command: prompt type:

```
Command: LOADDCL <return>
DCL File Name: hello <return>
```

Remember to type the dialog box name in lower case letters exactly as it is typed in the HELLO.DCL file. The dialog box displays, and should look like the one shown in Figure 1.1. If the dialog box did not show and you received an alert box, correct the errors indicated or check the **ACAD.DCE** file for possible errors.

No action will occur if you click on the buttons as no functions have been assigned to the tiles.

Click on **OK** to clear the dialog box.

Always use the *LOADDCL.LSP* routine to verify dialog box design and check for possible syntax errors before writing the program to control the dialog tiles. Often, it may take several attempts to correct the DCL syntax and make the dialog box design functional and acceptable.

Building DCL files from Tables

You will be instructed to create many Dialog boxes throughout this book. However, to gain experience at learning the DCL language, try to create the DCL files from the DCL File Definition table. A DCL File Definition table is a table that contains the tile references that define the dialog box. Your job is to follow the table, line by line, and to write the code for the DCL file in the proper format. Table 1.2 is the DCL File Definition table for the HELLO.DCL dialog box.

The DCL File Definition table is made up of three sections: Tile Reference, Attribute, and Value.

Tile Reference *This shows the basic tile types as predefined by the PDB facility. This includes buttons, toggles, tile clusters, text clusters, and edit boxes that are combined to form dialog boxes. A tile reference has the form:*

```
: tile name {
  attribute = value;
  .....
}
```

The *tile name* is the name of a predefined tile as defined by the PDB facility listed in the Tile Reference part of the table. A colon `:` indicates the beginning of a tile definition. The name of the tile is next after the colon **:** separated by a space. A curly brace indicates the beginning of the tile definition section. A closing curly brace is added after the attributes.

Attribute *A tile's attribute defines its physical appearance and function. Examples include key, label, height, width, enabled, and dis abled.*

Value *The value assigned to the tile attribute must be of a specific type, integer, real, string or a reserved word.*

Table 1.2. The DCL file definition table of the HELLO dialog box.

Tile Reference	Attribute	Value
dialog	label	"The Hello Program"
text	key	"txt_hello"
	label	""
	alignment	centered
row	**{**	
button	label	"Display Hello"
	key	"cmd_hello"
	width	18
	mnemonic	"D"
button	label	"Clear"
	key	"cmd_clear"
	width	10
	mnemonic	"C"
	}//row	
ok_only		

Entering the AutoLISP Code of the Hello Program

You are now ready to write the code to control the HELLO dialog. Start the text editor and type the following code exactly as shown. Or, you may type the code as each line is explained in the next section.

- ☐ Start the text editor and type the code inside the HELLO program.
- ☐ Save the file as HELLO.LSP and exit the text editor.
- ☐ Load the HELLO program file in AutoCAD using APPLOAD.
- ☐ Execute the program at the AutoCAD Command prompt.

The HELLO Program

```
;;; HELLO.LSP
;;; Displays HELLO WORLD! in the text tile.

(defun C:HELLO (/ CMD:HELLO CMD:CLEAR DCL_ID)

  (defun CMD:HELLO ()
    (set_tile "lst_hello" "HELLO WORLD!"))

  (defun CMD:CLEAR () (set_tile "lst_hello" ""))

  (setq DCL_ID (load_dialog "HELLO.DCL"))

  (if (not (new_dialog "hello" DCL_ID)) (exit))

  (action_tile "cmd_hello" "(cmd:hello)")
  (action_tile "cmd_clear" "(cmd:clear)")

  (action_tile "accept" "(done_dialog)")

  (start_dialog)

  (unload_dialog DCL_ID)

  (princ)

)//defun of hello

(prompt "\nType < HELLO > to execute.")
(princ)
```

The AutoLISP Code Inside the HELLO Program

Type the next two lines to ass comments to identify the LSP file and its purpose:

```
;;; HELLO.LSP
;;; Displays the text HELLO WORLD! in a text tile.
```

Comments within an AutoLISP file are preceded by one or more semicolons. AutoLISP ignores any data on the same line following a semicolon. Comments may start at the beginning of a line (as shown above) or after an AutoLISP expression. Examples of this include:

```
(setq X 1) ;set X to 1
```

For AutoCAD/AutoLISP Release 13, comments may be enclosed in the string ;|...|;. The comments may start on one line and may end after several lines. Examples of this format include:

```
;| HELLO.LSP
Displays the text HELLO WORLD! in a text tile. |;

(setq X ;| set X to 1 |; 1.0)

(setq X 1) ;| set X to 1
This is the default value for X |;
```

Use caution adding comments in this manner as a missing semicolon will cause an error. Also, if your program must be compatible with Release 12, use a semicolon as shown above.

Type the next line to define the function HELLO:

```
(defun C:HELLO (/ CMD:HELLO CMD:CLEAR DCL_ID)
```

User defined functions are created using the internal AutoLISP function *DEFUN*, which is short for **DE**fine **FUN**ction. New functions and commands are added to AutoCAD by using *DEFUN* to define and implement those commands and functions. A *DEFUN* function has the syntax:

```
(defun NAME ([ARGUMENTS] / [LOCAL VARIABLES])
  ([expressions]......)
  ([expressions]......)
)
```

The *NAME* portion is simply the name of the function you are defining. The *[ARGUMENTS]* following the function name are independent variables that take the values given them when the function is called. An optional *"/"* forward slash may follow and the local variables declared. Not all functions have *[ARGUMENTS]* and may only have *[LOCAL VARIABLES]*.

Local and Global Variables

A *[LOCAL VARIABLE]* will only have a value bound to it while the function or program is executing. Once the function has finished the value of the *[LOCAL VARIABLE]* is set to nil. Variables that are not declared in this section are considered global and retain their value after the function has executed.

By retaining their values, global variables may be used by other functions. If no *[ARGUMENTS]* or *[LOCAL VARIABLES]* are declared, you must supply an empty set of parentheses *()* after the function name. For example:

Function with no arguments or local variables:

```
(defun NEWFUNC () (expressions........))
```

Function with two arguments and no local variables:

```
(defun NEWFUNC (A B) (expressions........))
```

Function with no arguments and two local variables:

```
(defun NEWFUNC (/ C D) (expressions........))
```

Function with two arguments and two local variables:

```
(defun NEWFUNC (A B / C D) (expressions........))
```

WARNING: Never use the name of an AutoCAD command built-in or external as the name of a user defined function. This makes the built-in function inaccessible.

Expressions

One or more *[EXPRESSIONS...]* may follow the *[ARGUMENTS]* and *[LOCAL VARIABLES]* after the *defun* function. These expressions are evaluated when the function is called. This function doubles any number passed to it as the *(NUM)* argument.

```
Command: (defun DOUBLE (NUM) (* NUM 2))
```

This function has one *[ARGUMENT]* and no *[LOCAL VARIABLES]*. To call the function and double the number 12, you would type:

```
Command: (DOUBLE 12)
LISP returns: 24
```

User Defined Commands

New commands are added to AutoCAD using a "C:" before the user defined function name after the *defun*. The "C:" is not a reference to a disk drive, but a special prefix that denotes a command line function. The *[ARGUMENT]* list must be nil, but

[LOCAL VARIABLES] are permitted. Here is the same program, now defined as an AutoCAD command, named DOUBLE:

```
(defun C:DOUBLE (/ NUM DNUM)
  (setq NUM (getreal "\nEnter Number to Double: "))
  (setq DNUM (* NUM 2))
  (prompt
    (strcat "\nDoubled number is < " (rtos DNUM 2 2) " >"))
)
```

This command has two local variables *NUM* and *DNUM*. The number to double is now requested from the user. Notice the program as a function before must be enclosed in parenthesis when executed. As an AutoCAD command, the program is executed from the AutoCAD Command: prompt by typing the command name *DOUBLE* and pressing return.

Type the next line of code to define the CMD:HELLO function:

You may also define a function within another function or command. This is often referred to as a subroutine. A subroutine may perform a specific task over and over again. Subroutines may be created as a stand alone function, available to other programs, and located in the ACAD.LSP file. This allows access to the subroutine from any program you are using. Subroutines that perform a specific task, or are never used by another program, should be defined within the program itself as in this case.

The *CMD:HELLO* subroutine performs a specific task as does the *CMD:CLEAR* function, and is included within the *HELLO* program. Add a blank line then type this function on the next line:

```
(defun CMD:HELLO () (set_tile "txt_hello" "HELLO WORLD!"))
```

Short, user defined functions, may be typed on a single line. This user defined function adds the text HELLO WORLD! to the *text* tile with the attribute *key* value `"txt_hello"`.

SET_TILE Function

Use the *set_tile* function for setting the initial value of a tile or for changing the value during program execution. Syntax for the *set_tile* function is:

```
(set_tile KEY VALUE)
```

The *KEY* argument is a string that represents the *KEY* value for the tiles's key attribute as specified in the DCL file. The *VALUE* argument is a string that specifies the new value assigned the tile. This value is different for different types of tiles. Refer to the *VALUE* attributes for each specific tile for this argument.

Add a blank line then type the next line of code to define the *CMD:CLEAR* function:

```
(defun CMD:CLEAR () (set_tile "txt_hello" ""))
```

The *CMD:CLEAR* function clears the `text` tile when the *Clear* button on the dialog is selected.

Displaying the dialog box involves several steps. A DCL file must first be loaded. Use the *load_dialog* function to load the DCL file into memory. Add a blank line and type the next line of code:

```
(setq DCL_ID (load_dialog "C:/PROGRAMS/HELLO.DCL"))
```

The *SETQ* function is discussed before the *LOAD_DIALOG* function.

SETQ Function

This is the basic assignment function which assigns a value to a variable or a symbol. In AutoLISP, variables and symbols store values accessed by programs. Variables are user defined and refer to program data. The values of variables may change during program execution. The syntax for the *SETQ* function is:

```
(setq VARIABLE1 VALUE1 [VARIABLE2 VALUE2]....)
```

A single *SETQ* expression may set any number of variables. A variables' value may also be set to the value of another AutoLISP expression within the *SETQ* expression. In the program code above the variable DCL_ID is assigned the DCL identification number for the loaded dialog box calling the *(LOAD_DIALOG)* function.

NOTE: For ease of reading an AutoLISP program, type AutoLISP expressions in lowercase characters and program variables in uppercase characters.

LOAD_DIALOG Function

Code within a DCL file must be loaded into memory before AutoLISP can display the dialog box. *LOAD_DIALOG* opens the specified DCL file and reads the tile descriptions into memory. Syntax for *LOAD_DIALOG* is:

```
(load_dialog DCLFILENAME)
```

The *DCLFILENAME* argument is a string containing the name of the DCL file to load. If the file is located outside the AutoCAD library path, the specific directory path must be included with the file name for the DCL file to load. Examples of this include:

```
(load_dialog "C:\\PROGRAMS\\HELLO.DCL")

(load_dialog "C:/PROGRAMS/HELLO.DCL")
```

AutoLISP adds a default file extension of .DCL to the file name if the file extension is omitted. Other file extensions may be used as with .LSP files, but AutoLISP automatically recognizes the .DCL extension.

When the DCL file is loaded successfully, a DCL identification number (a positive integer) is returned. This number is incremented by one each time a DCL file is successfully loaded. Setting this number to the variable DCL_ID allows AutoLISP to access the DCL code. If the DCL file was not loaded successfully, the DCL identification number is set to -1.

Activate the Dialog

After the dialog is loaded into memory, it must be activated. Next, type the code to call the function *new_dialog* passing the name `"hello"` and DCL identification number, `DCL_ID,` as arguments. This activates the dialog and exits if the dialog doesn't load correctly.

```
(if (not (new_dialog "hello" DCL_ID)) (exit))
```

NEW_DIALOG Function

This function activates the dialog name found within the DCL file indicated. You may have more than one dialog definition within a single .DCL file. Use the *load_dialog* function to load the main DCL file, while *new_dialog* calls the specific dialog definition within the main DCL file. The syntax for *new_dialog* is:

```
(new_dialog DLGNAME DCL_ID [ACTION [SCREEN-PT]])
```

The *DLGNAME* argument is the name of the dialog definition within the DCL file. This name must be typed exactly the same as it appears in the DCL file. If you use uppercase characters you must use uppercase characters here.

DCL_ID is the DCL identification number assigned to the main DCL file loaded. These are the two required arguments.

An optional *ACTION* argument may be added to execute a default AutoLISP expression and must be specified when the *SCREEN-PT* argument is present. The *SCREEN-PT* argument is a 2D point list that specifies the upper-left corner of the dialog in Windows. In DOS the point refers to the lower-left corner.

Add a blank line then type the next line of code to add the first *action_tile* statement:

An *action_tile* statement defines what action is taken when a certain tile in the dialog box is selected by the user. You associate an AutoLISP expression with the tile

by calling the *action_tile* function. This first expression is associated with the "Display Hello" button which has the attribute key value *"cmd_hello."*

```
(action_tile "cmd_hello" "(CMD:HELLO)")
```

When the user selects the "Display Hello" button, this executes the **(CMD:HELLO)** subroutine as described and defined previously.

ACTION_TILE Function

Assigns an action for AutoLISP to execute when the user selects the specified tile in the dialog box. The syntax for *action_tile* is:

```
(action_tile KEY ACTION-EXPRESSION)
```

The *KEY* argument is the attribute *key* value assigned to the tile that triggers the action. The KEY argument is case sensitive and must match the attribute *key* value assigned to the tile.

The ACTION-EXPRESSION argument is an AutoLISP expression presented as a string by enclosing the entire expression in double quotes. If the AutoLISP expression must contain quotation marks, precede each quotation mark with a backslash character. The following expression shows an example of this:

```
(setvar "SNAPMODE" 0);;expression with quotation marks

(action_tile "tog_snap"
     "(setvar \"SNAPMODE\" (atoi $value))")
```

This expression refers to the tiles current value as *$value*. Additional tile values may be obtained during program execution. Table 1.3 list the variable names used and their descriptions:

Table 1.3. Action expression variables.

Variable Name	Description
$data	Application specific data (as set by client_data_tile)
$key	Key value attribute for the selected tile.
$reason	Code indicating user action. Used with edit_box, list_box, image_button and slider tiles. 1 = User selected tile 2 = User exited edit_box 3 = User changed value of slider 4 = User pressed enter when a list_box item was highlighted.
$value	The current value of the tile as a string.
$x $y	Point coordinates indicating the position the user picked on an image_button tile.

Type the next line of code to add the second action_tile expression:

The second expression is associated with the *CLEAR* button using the attribute key `value` *`“cmd_clear”`*.

```
(action_tile “cmd_clear” “(CMD:CLEAR)”)
```

Selecting this tile executes the **(CMD:CLEAR)** function as defined previously.

Add a blank line then type the next line of code to add the *action_tile* expression for the OK button:

```
(action_tile “accept” “(done_dialog)”)
```

The final action expression is associated with the OK button using the key `“accept”`. Selecting this tile executes the predefined *`(done_dialog)`* expression and closes the dialog.

DONE_DIALOG function

Once the user has completed the information requested within the various tiles of the dialog, the current dialog must be dismissed. This is accomplished calling the *done_dialog* function. The *done_dialog* syntax is:

```
(done_dialog [STATUS])
```

The predefined tiles *OK* and *Cancel* automatically issue a *done_dialog* when selected. This returns the integer 1, if *OK* is selected, and 0 if *Cancel* is selected. An optional *[STATUS]* argument may be supplied as the returned value instead of the standard 0 or 1. The *[STATUS]* value is returned by the *start_dialog* function when a variable is assigned to the *start_dialog* function, as in this example:

```
(action_tile "cmd_line" "(done_dialog 5)")

(setq DO_NEXT (start_dialog))
```

This code assigns the variable *DO_NEXT* to the value of 5 when the *"cmd_line"* tile is selected by the user. Decisions can now be made by the program based on the returned value of the *DO_NEXT* variable. Examples of this are explained in later chapters.

Add a blank line and type the next line of code to display the dialog box:

```
(start_dialog)
```

This function uses no arguments. Values returned by *start_dialog* depend on the *done_dialog* expression within the *action_tile* statement as explained above. The dialog remains active until a *done_dialog* is called. Nested dialogs put the main dialog on hold and reactivate it after they are closed.

Type the next line of code to unload the DCL file:

```
(unload_dialog DCL_ID)
```

This function prevents conflicts between tiles that reference previously defined tile names.

Type the next line of code to add the princ function. Use a princ function to suppress the nil returned after program execution:

```
(princ)
```

Type the final closing parenthesis to close the main *defun* of the HELLO program:

```
)//defun of hello
```

Type the next two lines of code to add a prompt to inform the user how the program is executed:

```
(prompt "\nType < HELLO > to execute.")
(princ)
```

Always add a prompt to inform the user how to execute the program. This prompt is outside the closing parenthesis of the *defun*. When the program loads, this prompt is displayed. To suppress the returned *nil* after the prompt, add the `(princ)` function as the last line.

Save the code and exit the text editor. From within AutoCAD execute the HELLO program.

Executing the Hello Program

The Hello program is completed. Use the following steps to execute the program from within AutoCAD:

- Use Appload to load the HELLO.LSP file.
- Type HELLO at the AutoCAD command prompt to execute the program.
- Click the Display Hello and Clear buttons to display and clear the text.

- You may use the Alt-D and Alt-C keys to set the focus to these buttons. After the focus is set to the button press the enter key or space bar to execute the button.

- To exit and close the dialog press the OK button.

Note that the text displays inside the text tile when the Display Hello button is selected. The clear button displays a blank text string when selected thus clearing the text tile.

Summary

In this chapter you wrote your first custom Dialog Box program. You learned about the steps necessary to write a Dialog Box program:

- The visual DCL code programming step

- The AutoLISP code programming step

In the visual programming step you created dialog box tile definitions within a .DCL dialog definition file.

In the AutoLISP code programming step you created the program to control the dialog tiles as they are selected by the user in the dialog box.