AUTODESK. UNIVERSITY

AS322616-L

AutoCAD Customization Boot Camp: Automate Workflows and Tasks

Lee Ambrosius
Autodesk, Inc.

Learning Objectives

- Learn how to record and play back action macros
- Learn how to create and load small AutoLISP programs
- Learn how to deploy AutoLISP programs
- Learn how to manage user settings with profiles

Description

AutoCAD software offers a variety of features that allow you to automate workflows and reduce repetitive tasks. In this lab, you'll create action macros, develop simple AutoLISP programs, learn the basics to deploy AutoLISP files, and manage settings with user profiles. After this lab, you will have a broad understanding of how to implement automation and improve your productivity when you return to your office. This session features AutoCAD; AutoCAD LT doesn't support AutoLISP programming.

Speaker(s)

Lee Ambrosius is a Principal Learning Experience Designer at Autodesk, Inc., for the AutoCAD® and AutoCAD LT products on Windows and Mac. He works primarily on the customization, developer, and CAD administration documentation along with the user documentation. Lee has presented at Autodesk University for about 15 years on a range of topics, from general AutoCAD customization to programming with the ObjectARX technology. He has authored several AutoCAD-related books, with his most recent project being *AutoCAD Platform Customization: User Interface, AutoLISP, VBA, and Beyond*. When Lee isn't writing, you can find him roaming various AutoCAD community forums, posting articles on his or the AutoCAD blog, or tweeting information regarding the AutoCAD product.

Twitter: @leeambrosius
Email: lee.ambrosius@autodesk.com
Blog: http://hyperpics.blogs.com

1 Introduction

The AutoCAD software is an extensive 2D drafting and 3D modeling program that has grown in functionality since it was first introduced almost 35 years ago back in 1982. What sets AutoCAD apart from many other CAD programs is its expansive customization and automation capabilities. The customization and programming features of AutoCAD allow individuals and companies to simplify everyday workflows, such as:

- Initial drawing setup; establish drawing units and format, create layers, insert a title block and populate attribute values
- Extraction of design data for use downstream in a bill of materials or order entry system
- Consumption of project information from a data source such as a spreadsheet or database

This lab will provide you with the opportunity to roll-up your sleeves and get some hands-on experience with customizing AutoCAD which will prepare you to apply the techniques covered in your everyday workflows. While knowing how to program isn't a requirement to customize AutoCAD, learning how to program does provide you with a greater set of resources to automate tasks in AutoCAD.

2 Which Customization and Programming Options are Available

Not all customization and programming options are created equally, some options are easy to learn and are well integrated into the AutoCAD program that many don't even realize they are customizing the program. For example, creating new layers and named styles are forms of customization that are preformed frequently. There are two types for customization and programming that are available; drawing and application.

The following lists many of the customization and programming options available:

Basic

<u>Drawing</u>

- Layers
- Blocks
- Annotation styles (text, dimensions, multileaders, and tables)
- Materials and visual styles
- Drawing templates

<u>Application</u>

- Desktop shortcut
- Command aliases
- Tool palettes
- Workspaces
- User profiles
- Plot styles

Intermediate

<u>Drawing</u>

- Dynamic blocks

<u>Application</u>

- Scripts
- Action macros
- User interface (CUI Editor)
- DIESEL
- Custom linetypes and hatch patterns
- Custom shapes and text styles

Advanced (Application Only)

- AutoLISP / Visual LISP
- Visual Basic for Applications (VBA)
- ActiveX / COM (VBA, VBScript, VB.NET, C#, C++)
- Database connectivity
- Managed .NET (VB.NET, C#)
- ObjectARX (C++)
- JavaScript
- Sheet Set Manager API
- CAD Standards plug-ins
- Transmittal API
- Connectivity Automation API
- Forge Platform APIs

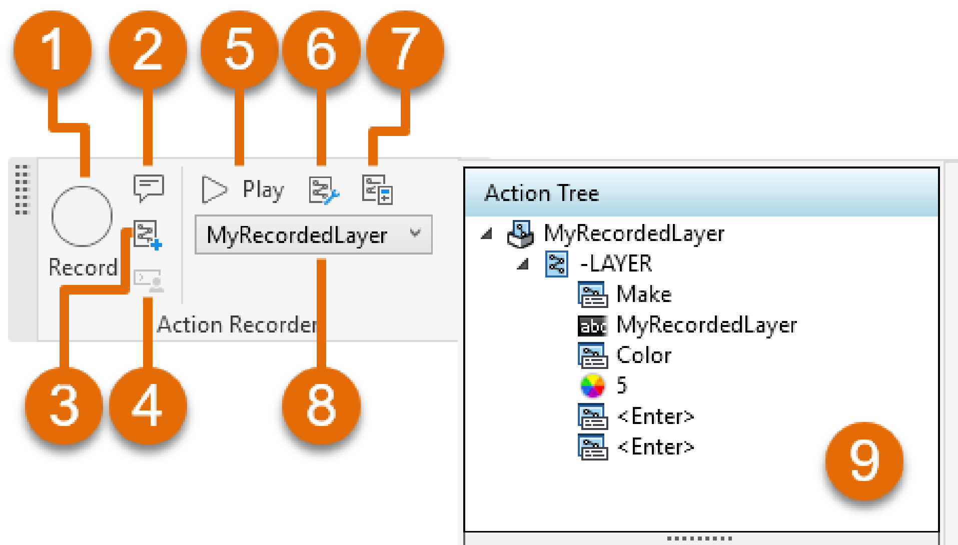
3 What You Need to Get Started

Many of the customization and programming options available for use within the AutoCAD program are supported through utilities or commands found inside the program, or applications installed with the operating system (Windows or Mac OS). It is when you want to extend the functionality of the AutoCAD program using ActiveX, Managed .NET, or ObjectARX that you will need to purchase, download, and/or install additional software.

4 Action Macros

Action macros allow for the automation of repetitive tasks without needing to know how to create a script file or learn a programming language. An action macro is recorded with the Action Recorder and stored in an action macro file which has the file extension of *.actm*.

The Action Recorder, shown in the following illustration, is used to record and modify an action macro, and it can be found on the Manage tab of the AutoCAD ribbon.



When the Action Recorder is set to Record mode, the commands and values you enter at the Command prompt or from various user interface elements are captured in real-time. Once you stop recording, captured actions can then be saved to an ACTM file. If actions are saved to an ACTM file, AutoCAD immediately loads the ACTM file into memory and defines a command from which the action macro can be played back. Playback of an action macro can be started by entering its name at the Command prompt or from the Action Recorder panel.

In addition to the tools used to start and stop the recording of an action macro, the Action Recorder panel contains tools that allow you to modify and manage ACTM files. Expanding the Action Recorder panel, by clicking its title bar, reveals the Action Tree which displays all the captured actions in real-time.

The following describes each control on the Action Recorder panel shown in the previous illustration:

1. **Record/Stop** – Starts and stops the recording of an action macro and stops the playback of an action macro.
2. **Insert Message** – Inserts a user message into the current action macro.
3. **Insert Base Point** – Inserts a request for a base point during the playback of the action macro; the base point provided is used by the proceeding coordinate entry/value in the action macro.
4. **Pause for User Input** – Inserts a pause for user input of the selected value node in the Action Tree; the user is requested to provide a value during the playback of the action macro.
5. **Play** – Starts the playback of the action macro selected in the Action Macros drop-down list.
6. **Preference** – Displays the Action Recorder Preferences dialog box which allows you to control the display of the Action Recorder panel during recording and playback.
7. **Manage Action Macros** – Displays the Action Macro Manager which allows you to copy, rename, modify, and delete previously saved ACTM files.
8. **Action Macros Drop-down List** – Displays a list of all available action macros that can be played back or modified. Selecting an action macro from the list sets it as the current action macro to be modified in the Action Tree or played back.
9. **Action Tree** – Displays the individual actions and values defined in the current action macro.

Note: If the Action Recorder panel is expanded and docked, the panel can be pinned so it remains open until you switch tabs on the ribbon or unpin the panel.

What Are Actions and What Can Be Recorded

Actions are the smallest tasks or user interactions that can be recorded with the Action Recorder. An action might be a command that prompts the user for input at the Command prompt or displays a dialog box. While commands that display dialog boxes can be recorded, it is best to avoid such commands and use those that prompt for values at the Command prompt instead. Using commands that display a dialog box will not “break” or stop the playback of an action macro during playback, like a script, but it is recommended to avoid dialog boxes to ensure consistency during playback as values and choices made in a dialog box are not captured.

As an alternative to using a command that displays a dialog box, use the command line version of a command when available which typically starts with a "–" (hyphen) or in some cases you might need to use system variables. For example, the INSERT command displays the Blocks palette, while the –INSERT command prompts for values at the Command prompt.

The Command prompt is the primary method to start commands and input values when recording an action macro with the Action Recorder. Besides the Command prompt, actions performed from the following user interface elements are also recorded:

- Application menu, Quick Access toolbar, "classic" toolbars, pull-down and shortcut menus, ribbon panels, and status bar
- Layer Properties Manager
- Properties and Quick Properties palettes
- Tool Palettes window
- AutoCAD DesignCenter

While most commands can be recorded, commands that open, create, or close a drawing can't be recorded along with a small number of other standard commands. You can find out which commands can't be recorded and other tips about recording action macros under the topic About Recording Action Macros in the AutoCAD Online Help system.

Recording Actions

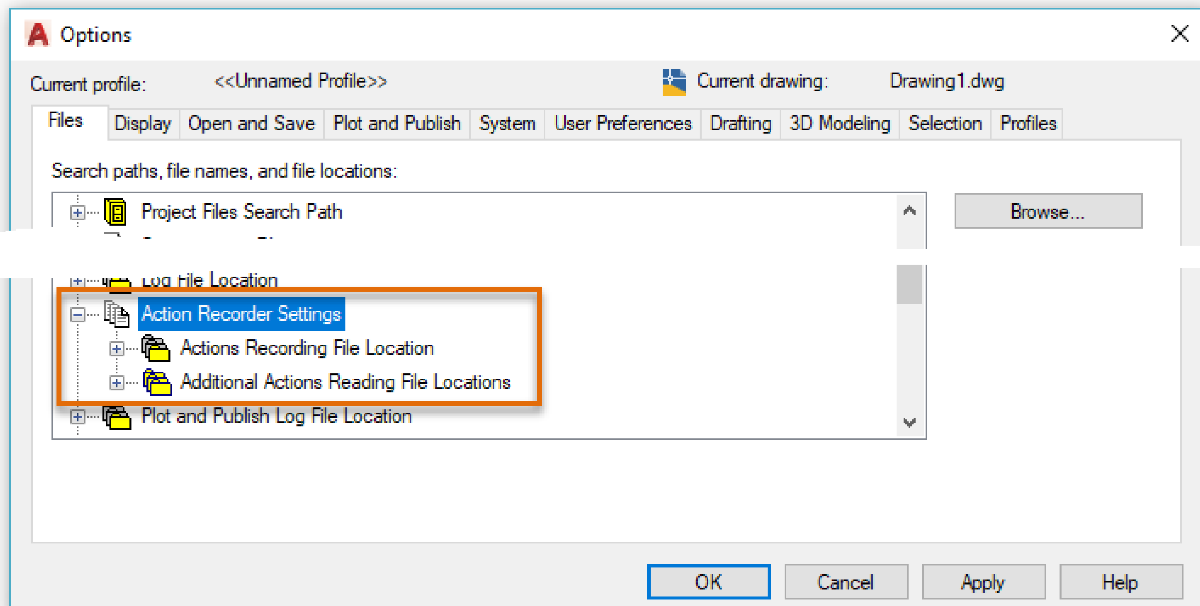
Recording actions with the Action Recorder is like operating a digital video recorder (DVR). You use the Record button (ACTRECORD command) to start recording and when you are done recording you click the Stop button (ACTSTOP command). Once the recording of an action macro is complete, you use the Action Macro dialog box to name and save the recorded actions to an ACTM file. The playback behavior and values of an action macro can be modified in the Action Tree of the Action Recorder after an action macro has been saved.

Storing of Action Macro Files

The Action Recorder stores newly recorded action macros to one location, while ACTM files can be played back from multiple locations. The locations in which the Action Recorder works with are specified under the Action Recorder Settings node on the Files tab of the Options dialog box

The following provides a description of the two location settings under the Action Recorder Settings node:

- **Actions Recording File Location** – Specifies the location used to store newly recorded ACTM files. Action macros stored in this location can be modified and played back.
- **Additional Actions Reading File Locations** – Specifies the locations in which additional ACTM files can be played back, but not modified.



Playing Back an Action Macro

ACTM files that are stored in the locations specified under the Actions Recording File Location and Additional Actions Reading File Locations nodes in the Options dialog box are automatically loaded when AutoCAD starts and when a new action macro file is saved.

To playback an action macro, you can do one of the following:

- At the Command prompt, type the name of the action macro and press Enter.
- On the ribbon, click Manage tab ➢ Action Recorder panel ➢ Action Macros drop-down list and select the action macro to playback, and then click Play.
- Right-click over the drawing area, choose Action Recorder ➢ Play submenu, and then choose the action macro to playback.

Modifying an Action Macro

During the recording of an action macro, you don't have many options for modifying the actions of an action macro except for inserting a pause for input or user message by right-clicking a node in the Action Tree. Once an action macro (ACTM) file has been saved, you can modify its recorded values and delete unnecessary actions. If an unnecessary action is deleted, it can't be restored nor, can an action later be re-inserted into an action macro. Selecting the main node of an action macro in the Action Tree allows you to make changes to and manage the file on disk that represents the current action macro, and not one of its action or value nodes.

Value nodes represent coordinates, object selection sets, and the input that was provided during the recording of the action macro. From the Action Tree, you can edit the current value of a node, change a value node so it requests a new value during playback, control object selection behavior, and more.

Tip: Action macros deleted from the Action Recorder panel are sent to the Windows Recycle Bin where they can be restored.

Additional Information

You can find additional information on action macros and ACTM files with these topics in the AutoCAD Online Help system:

- About Action Macros
- About Action Macro Files
- About Recording Action Macros
- About Managing Action Macros
- About Playing Back an Action Macro

E1.A Recording and Playing Back a Custom Action Macro

This exercise explains how to create an action macro that creates a new layer and prompts for the points to define a rectangular revision cloud.

See the Exercises section in the lab handout.

E1.B Inserting a Dynamic Block with a Custom Action Macro

This exercise explains how to create an action macro that creates a new layer and inserts a dynamic block with a set of predefined property values.

See the Exercises section in the lab handout.

5 AutoLISP Programs

AutoLISP is an interpreted programming language that can be used to extend the functionality of the AutoCAD program. Unlike other programming languages or libraries, such as Managed .NET and ObjectARX, programs developed with AutoLISP don’t need to be compiled before being loaded into the AutoCAD program. The AutoLISP programming language is made up of functions and data types that allow for a wide range of access to various parts of the AutoCAD drawing environment which can be used to create custom applications.

Custom applications developed with the AutoLISP programming language can be used to:

- Retrieve information about an open drawing
- Create/modify objects in a drawing
- Read/write information in an external file
- Access third-party applications and programming libraries using ActiveX (Windows only)

The AutoLISP programming language is based on the LISP (LISt Processing) programming language and is designed for use inside of AutoCAD. The functions in the AutoLISP programming language can be broken down into several categories of functionality:

- Math calculations
- Conditional tests
- Data conversion
- Requests for user input

- Creation and modification of drawing objects
- Among many others

AutoLISP expressions can be executed directly at the Command prompt in the AutoCAD program or by loading an LSP file into AutoCAD. When a (or an ! is typed at the Command prompt, it signals to AutoCAD that it needs to send the entered text to the AutoLISP interpreter which then processes the statement and returns a value to the Command prompt.

The following explains the significance of the two characters that are used to indicate the start of an AutoLISP expression in AutoCAD at the Command prompt.

ASCII Character	Description	Use of Character
(	Open parenthesis	An open parenthesis indicates the start of an AutoLISP expression and must always have a balancing closing parenthesis.
!	Exclamation point	Notifies the AutoLISP interpreter to return the current value assigned to the specified symbol. **Note:** Valid only from the Command prompt and not an AutoLISP program.

While AutoLISP expressions can be entered at the Command prompt in AutoCAD, they are often stored in external files with a *.lsp* extension. LSP files are ASCII text files that contain no special text formatting and can be edited with a program, such as Notepad on Windows or TextEdit on Mac OS. After AutoLISP expressions are saved to an LSP file, they can be loaded into and executed by the AutoCAD program with the Load/Unload Applications dialog box (APPLOAD command).

Learning the AutoLISP Syntax

Parentheses and exclamation point characters indicate to AutoCAD that it needs to send the text being entered at the Command prompt to the AutoLISP interpreter. The interpreter then evaluates the entered text for correct formatting and use of the functions. If the formatting is not correct or not enough values have been provided, error messages are displayed in the Command Line window to let you know something needs to be fixed.

The syntax for an AutoLISP function is as follows:

```
(function_name <argumentX…>)
```

- ***function_name*** represents the name of the function to execute
- ***argumentX*** represents the values that the function expects, some functions don't expect any values while others expect one or more values

The following are examples of AutoLISP expressions that can be entered at the Command prompt or executed from an LSP file:

- `(setq dRadius 12.0)`
- `(command "._line" "0,0" "5,5" "")`

When you enter an expression that starts with an open parenthesis, the AutoLISP interpreter expects the first value after the open parenthesis to be the name of a function. If a function name isn't provided or recognized, the AutoLISP interpreter displays the message

```
; error: no function definition
```

Optionally, a function can be defined with arguments which are placeholders for values that are provided when using the function. The name of a function and each argument provided must be separated by a space.

Note: Unlike some programming languages, AutoLISP isn't case sensitive, so the function name `alert` is the same as `Alert` or `ALERT`. However, values that represent a text string are case sensitive; "Hello" isn't the same as "HELLO".

Almost all AutoLISP functions return a value, the value returned might be `nil` or one of the supported data types. The `nil` value is AutoLISP's way of saying no value; nothing or empty. The value returned by an AutoLISP function can be used by another function.

In the previous examples, the AutoLISP expressions showed examples of the `setq` and `command` functions. The `setq` and `command` functions are the most commonly used AutoLISP functions, and they are used throughout this session.

- **`setq`** – Creates and assigns a value to a user-defined variable. The variable name and value to be assigned to the variable must follow the `setq` function name. Optionally, more than one user-defined variable name and value pairings can be passed to the `setq` function, if desired.

  Example of assigning a value of 12.0 to a user-defined variable named *`dRadius`*:

  ```
  (setq dRadius 12.0)
  ```

  Example of assigning values to multiple user-defined variables in a single statement:

  ```
  (setq dRadius 12.0 ptCen (list 5 5 0))
  ```

  The *`dRadius`* variable is assigned the value of 12.0, while the *`ptCen`* variable is assigned a list of values that represent the coordinate 5,5,0.

- **`command`** – Executes a standard AutoCAD command or one defined by a loaded ObjectARX or Managed .NET application.

  Some custom programs redefine the behavior of standard AutoCAD commands. It's important to place a **.** (period) in front of all command names executed by the `command` function to ensure they execute as expected using the standard definition of the command, and not a redefined version if one exists.

It is also recommended to place an _ (underscore) in front of all command and option names to ensure they are recognized by the AutoCAD program no matter the language pack installed; English, French, German among others. An _ (underscore) in front of a command or option name indicates that the global name, which is the same as the English language, of the command or option should be used.

As programs grow in complexity, you will encounter the need to nest multiple expressions. When expressions are nested, the innermost expression is evaluated first. The following is an example of a nested expression:

```
(+ 15 (* 2 12))
```

In the previous example, `(* 2 12)` is evaluated first because it is the innermost expression. The two numbers multiplied together returns a value of 24.

The value of 24 is then used by the next expression, in this example it is the outer expression. The outer expression after the innermost expression is evaluated looks like `(+ 15 24)` which results in the two numbers being added together for a value of 39. 39 is then returned as the final value.

The following are additional examples of nested expressions:

- `(setq mVal (* 2 12))`
- `(setq nVal (getint "\nEnter an integer: "))`
- `(setq str (strcat "Welcome " "to AU" " 2019!"))`

Defining Custom Functions

When developing programs with AutoLISP, you often want to execute several expressions at a time without needing to retype them or load them from a file each time you want to execute the expressions. The `defun` (or Define function) function is used to define a custom AutoLISP function; a custom function can be defined to mimic a standard AutoCAD command.

The following syntax is used to define a custom function:

```
(defun <function_name> (<argumentX…> / <variableX…>)
   <expressionsX…>
)
```

- ***function_name*** represents the name to be assigned to the custom function
- ***argumentX*** represents the values that the custom function expects
- ***variableX*** represents the variables to be defined locally within the custom function
- ***expressionsX*** represents the expressions to be executed by the custom function

Arguments are used as a way to pass values to a custom function and must be placed before the forward slash in ( ) after the custom function’s name. The use of arguments and variables when defining a new custom function is optional. A custom function can execute any number of AutoLISP expressions.

A custom function that can be entered directly at the Command prompt like a command must have its name prefixed with c: and not be defined with any arguments.

```
(defun c:<function_name> ( / <variableX…>)
   <expressionsX…>
)
```

- ***function_name*** represents the name to be assigned to the custom function
- ***variableX*** represents the variables to be defined locally within the custom function
- ***expressionsX*** represents the expressions to be executed by the custom function

The following statements define a function named `create-layer` that creates a new layer and sets that new layer current. The `create-layer` function is defined with two arguments that represent the name and color of the layer to be created.

```
; Usage: (create-layer "NewLayer" "45")
(defun create-layer (lay-name lay-clr / )
  (command "._-layer"
           "_m"
           lay-name
           "_c"
           lay-clr
           ""
           ""
  )
)
```

The following example defines a custom function named `Circ5`. When loaded, the name of the function can be typed directly at the Command prompt to execute the function.

```
(defun c:circ5 ( / )
   (command "._circle" PAUSE 2.5)
)
```

Upon execution, the `Circ5` function starts the AutoCAD CIRCLE command and immediately pauses and allows the user to provide a coordinate value. The pause for the user to specify a center point is indicated by the `PAUSE` variable. `PAUSE` is an internally defined variable that is used when you want the user to provide a value during the execution of the `command` function. Once a center point is provided by the user, a value of 2.5 is passed to the command and that value is used to define the radius of the circle.

Variables used in a custom function should be listed after the forward slash in **`( )`** which immediately follows the function name in a `defun` statement. Listing the variables after the forward slash specifies that the variables are only accessible within the custom function; these variables are known as *local variables*. Variables not declared using this approach are known as

global variables. Global variables are accessible by any AutoLISP expression executed in the same drawing as the global variable while the drawing remains open.

Typically, you want to declare variables locally, so they are only available while your custom function is executing. Using local variables helps to minimize the consumption of system resources and avoid conflicts with other loaded AutoLISP programs.

The following AutoLISP expressions demonstrate the effect of local and global variables:

```
; Define the var variable as global
(setq var "global")

; nil is returned as the command uses the local
; (in function) var variable
(defun c:Var2 ( / var)
  (type var)
)

; STR is returned as the command uses the global var variable
(defun c:Var1 ( / )
  (type var)
)
```

Requesting User Input and Displaying Messages

User input plays a significant role in AutoLISP programs, just like it does for standard AutoCAD commands. Most AutoCAD commands request input at the Command prompt, such as a selected object or typed value.

While most commands prompt for values at the Command prompt, some commands that are more complex request values using a dialog box. AutoLISP programs do support the ability to display and get input from a dialog box which is not covered in this session and supported on Windows only. You can learn more about dialog box creation and implementation with AutoLISP by reading the topic About Syntax and Comments in DCL Files (DCL) in the AutoCAD Online Help system.

The `command` function can accept static values or allow the user to provide a value. The `PAUSE` variable, previously mentioned, is used to interrupt the execution of a command and allow the user to provide a value. The following is an example of using the `PAUSE` variable with the LINE command:

```
(command "._line" PAUSE PAUSE "")
```

In the previous example, the LINE command is paused twice to allow the user to provide a start and endpoint for a new line object. Execution of the AutoLISP expression continues if the user provides two valid coordinates. The empty string value provided after the two pauses tells the LINE command to end. When the `PAUSE` variable is used with the `command` function, the command being executed validates the value provided by the AutoLISP program just as if it was entered at the Command prompt.

If one of the values provided is invalid, an error message might be displayed at the Command prompt and one of the following situations may happen:

- Execution of the AutoLISP statement/program just stops
- Execution continues and an unexpected result occurs

While pausing for user input can be very helpful, pausing during the `command` function doesn't always allow for the validation of the input before it is provided to the command. Input should be validated before it is passed to a command or another function. To help control and validate input entered, AutoLISP provides functions that can be used to get input from a user at the Command prompt and then determine how to proceed once the input has been validated. There are three basic types of input that you can request from a user: an object, a coordinate, or an alphanumeric/numeric value.

When using a get input function to request a value, a custom prompt can be displayed to let the user know which type of input to provide. Custom prompts should match those presented by standard AutoCAD commands.

The following special characters/syntax are used to format most prompts displayed by standard AutoCAD commands:

- **`[ / ]`** – Square brackets and forward slashes are used to present a list of options
- **`< >`** – Angle brackets are used to indicate the default value that should be accepted when the user presses Enter at the prompt without providing a value first
- **`:`** – Colon is used to indicate the end of a prompt sequence; a space is usually provided after the colon to ensure any value provided by the user is separated from the prompt

Tip: It is recommended to prefix a prompt string with the `\n` character sequence to force the prompt onto a new line.

The following AutoLISP expression displays a prompt which contains a list of three options and provides the user with a default value:

```
(getkword "\nEnter shape [Circle/Square/Hexagon] <Circle>: ")
```

When executed in AutoCAD, the previous example results in the following appearing in the Command Line window:

```
Enter shape [Circle/Square/Hexagon] <Circle>:
```

As mentioned earlier, AutoLISP allows you to request input from the user. The following table lists functions that can be used to request input from the user and the type of data that the function returns.

Function	Description	Usage of Function

`getint`	Pauses for an integer (whole) number in the range of -32,768 and 32,767 and returns the integer value provided.	Syntax:
		`(getint [prompt])`
		Example(s):
		`(getint "\nEnter a number: ")`

`getreal`	Pauses for a real (decimal or floating) number, and returns the real number provided.	Syntax:
		`(getreal [prompt])`
		Example(s):
		`(getreal "\nEnter a number: ")`

`getstring`	Pauses for a text string and returns the text string provided.	Syntax:
		`(getstring [allowspaces] [prompt])`
		Example(s):
		`(getstring "\nEnter a room label: ")`
		`(getstring T "\nEnter your name: ")`

`getpoint`	Pauses for a coordinate, and returns a list representing the coordinate provided.	Syntax:
		`(getpoint [basepoint] [prompt])`
		Example(s):
		`(getpoint "\nSpecify first point: ")`
		`(getpoint (list 2 2 0)`
		`"\nSpecify second point: ")`

`entsel`	Pauses for a single object and returns a list that contains an entity name and the coordinate picked in the drawing window.	Syntax:
		`(entsel [prompt])`
		Example(s):
		`(entsel "\nSelect an object: ")`

Function	Description	Usage of Function

`ssget`	Pauses for a set of objects and returns a pickset containing all the objects selected.	Syntax:
		`(ssget [selection-flags] [point list])`
		Example(s):
		`; User specifies object selection mode`
		`(ssget)`
		
		`; Select single object`
		`(ssget "SI")`
		
		`; Select previous object`
		`(ssget "P")`

Here are some additional functions that can be used to request input from a user:

- **`getcorner`** – Pauses for a coordinate, and returns a list representing the coordinate provided. Like the `getpoint` function, but the user is prompted for a point diagonal from a previous coordinate.
- **`getdist`** – Pauses for a real number or two coordinates. Returns the real number entered or the distance between the two coordinates specified as a real number.
- **`getangle`** and **`getorient`** – Pauses for a real number or two coordinates and returns a real number. The `getangle` function returns a real number that is a relative angle to the current value of the ANGBASE system variable, while the `getorient` function returns an absolute angle that isn't affected by the value of the ANGBASE system variable.
- **`getkword`** – Pauses for a string based on a pre-determined list of keywords. Requires the use of the `initget` function to establish a list of keywords from which the user can choose an option.
- **`nentsel`** – Pauses for a single object nested in a block or polyline. Returns a list that contains the entity name of the selected nested object and the coordinate value picked in the drawing window.

The main purpose of the `initget` function is to establish a list of supported keywords for the `getkword` function, but the `initget` function can also be used to alter the behavior of other `get*` functions except for the `getstring` function. The `initget` function can also be used to affect the behavior of the `entsel` and `nentsel` functions. See the AutoLISP Reference Guide to learn more about the `get*`, `initget`, `entsel`, `nentsel`, and `ssget` functions.

While many of the user input functions allow you to provide an optional prompt message, there are times when you might want to display information to the user without requesting input. For example, you might want to let the user know how many objects were selected or the default values your program is set to use.

The following functions can be used to provide information back to the user:

- **`prompt`** – Displays a text string at the Command prompt. Prefixing the text string with `\n` forces the message onto a new line.
- **`terpri`** – Forces a blank line in the AutoCAD Text window and Command Line history.
- **`alert`** – Displays a text string in a basic message box with an OK only button.
- **`princ`** – Displays the value of a variable, text string, or other input at the Command prompt. It can also be used to hide the return value of the most recently executed AutoLISP function.
- **`textscr`** – Displays the AutoCAD Text window.
- **`graphscr`** – Hides the AutoCAD Text window when it is displayed.

Storing Data and Working with Data Types

In the previous section, it was mentioned that AutoLISP functions return a value. The return value varies based on the function evaluated and the values provided to the function.

There are seven main data types that AutoLISP supports which are listed in the following table:

Data Type	Description

Integer (`INT`)	Any whole number; a number without a decimal point.
	Example(s):
	`12`
	`0`

Real (`REAL`)	Any number with a decimal point; sometimes referred to as a double or floating number.
	Example(s):
	`12.1250`
	`0.0000`

String (`STR`)	Any number of alphanumeric characters enclosed in double quotation marks.
	Example(s):
	`"12.125"`
	`"Welcome to AU 2019"`

Data Type	Description

List (`LIST`)	Any expression in parentheses, such as an AutoLISP expression or a coordinate value.
	Example(s):
	`(alert "An unknown error occurred.")`
	`(list 0.0000 5.0000 0.0000)`

Pickset (`PICKSET`)	A selection set of objects.
Entity Name (`ENAME`)	An object in a drawing.

Symbol (`SYM`)	Internally defined AutoLISP variable; most variables simply return the type of data that they are assigned.
	Example(s):
	`T`
	`PAUSE`

There is an additional data type which can be used to determine if a custom function is loaded or not, the Subroutine (SUBR) data type. It is possible to use custom functions defined in other AutoLISP (LSP) files, prior to using functions defined in other LSP files you should check to see if the function is defined in the current drawing. If the function isn't available, you could then attempt to load the LSP file in which the function is defined before continuing the execution of your AutoLISP program.

The `type` function can be used to determine the type of data returned by a function or that is currently assigned to a variable.

The following are some examples of the `type` function:

- `(type "Data Type")` – Returns `STR` for a string
- `(type 12.5)` – Returns `REAL` for a real number
- `(type (getint "\Enter an integer: "))` – Returns `nil` or `INT` based on the value entered
- `(type setq)` – Returns `SUBR` for subroutine
- `(type customFunction)` – Returns `nil` unless a function named `customFunction` is defined in the current drawing

Note: The `if` function can be used to branch your custom programs and determine how your program should handle an unexpected value or data type. More information on the `if` function can be found later under the "Testing Conditional Statements" section of this handout.

The value returned by an AutoLISP function can be assigned to a user-defined variable for use later during the current drawing session while the drawing file remains open. If a drawing file is closed, all user-defined variables defined while the drawing was opened are destroyed. The `setq` function is used to assign a value to a user-defined variable. The value assigned to a user-defined variable can be retrieved by entering an **!** (Exclamation point) in front of the variable's name at the Command prompt or using the variable in place of a static value when specifying values for function arguments in an AutoLISP expression.

User-defined variables are not the only variables that can be used to store values; AutoLISP also allows for the storage of values with system and environment variables. System variables are used to access common drafting settings, previously calculated values, or user provided values among others. A system variable can be drawing or application specific.

Environment variables often affect the behavior of the AutoCAD application and are largely undocumented, but you can define your own environment variables with AutoLISP unlike system variables. You can't define your own system variables with AutoLISP. While you can't define your own system variables with AutoLISP, there are 15 system variables that you can use to store values during the current drawing session; USERI1 through USERI5 can be used to store integer values, USERR1 through USERR5 can be used to store real number values, and USERS1 through USERS5 can be used to store string values. The values stored in USERS1 through USERS5 are not retained when the drawing is saved and reopened.

The `setvar` and `getvar` functions are used to work with system variables and their values, while the `setenv` and `getenv` functions are used to access environment variables.

- **`setvar`** – Assigns a value to a system variable
- **`getvar`** – Returns the current value of a system variable
- **`setenv`** – Assigns a value to an environment variable, the provided value must always be a string
- **`getenv`** – Returns the current value of an environment variable

The following are examples of the `setvar` and `getvar` functions:

- `(getvar "filletrad")` – Returns the current value of the FILLETRAD system variable
- `(setvar "filletrad" 0.125)` – Assigns the value of 0.125 to the FILLETRAD system variable

The following are examples of the `setenv` and `getenv` functions:

- `(getenv "MaxArray")` – Returns the current value of the MaxArray environment variable
- `(setenv "MaxArray" "200000")` – Assigns the value of 200000 to the MaxArray environment variable

Note: The variable name you use with the `setenv` and `getenv` functions is case specific, MaxArray is not the same as maxarray or MAXARRAY.

The values returned by a function or assigned to a variable can be manipulated and converted from one data type to another.

The following table lists several of the functions which can be used to manipulate a value or convert a value between data types.

Data Type	Description
Integer (`INT`)	**`+`**, **`-`**, **`/`**, **`*`** – Adds, subtracts, divides, or multiplies two or more numbers **`abs`** – Returns the absolute value of a number **`itoa`** – Converts an integer number to a string
Real (`REAL`)	**`+`**, **`-`**, **`/`**, **`*`** – Adds, subtracts, divides, or multiplies two or more numbers **`abs`** – Returns the absolute value of a number **`rtos`** – Converts a real number to a string
String (`STR`)	**`strcat`** – Concatenates two or more strings into a single string **`substr`** – Returns part of a string **`strlen`** – Returns the number of characters in a string **`atoi`** – Converts a string to an integer number **`atof`** – Converts a string to a real number
List (`LIST`)	**`list`** – Creates an empty list when no values are supplied or creates a list containing all supplied values **`car`** – Returns the first item in a list or `nil` if the provided list is empty **`cdr`** – Returns a list containing all items in a list except for the first item **`nth`** – Returns the item at a specified index value within a list; the first item in a list has an index of zero **`length`** – Returns the number of items in a list **`member`** – Returns `nil` or a list that contains the matching item found in a list along with all items after the matching item in the list

Testing Conditional Statements

Programs that don't request input from a user are typically referred to as being *linear*; they always follow the same path of execution each time they are executed. Linear programs are the easiest to write, debug, and troubleshoot.

Most programs request some form of input from the user, and therefore don't follow a linear path. When input is requested, the program requesting input should verify the value that is provided. While the `get*` functions are designed to request a specific type of data and perform some basic validation, it is up to you as the programmer to verify the value and that it is of the expected data type. If the Enter or Esc key is pressed by the user, a value of `nil` might be returned by a `get*` function or the AutoLISP program might suddenly terminate.

Before a value returned by a `get*` function is used, it is recommended to test the value with the assistance of what is known as a *conditional statement*. There are two functions that can be used to write conditional statements; `if` and `cond`. The `if` function is the most common conditional function and it is used to determine whether a value or expression evaluates to `True` or `False`. A value of `True` or `False` determines which execution path is to be followed, the separation of execution paths is known as *branching*. The first path of execution is followed when the conditional test evaluates to `True` (a value other than `nil`), while the second path of execution is followed when the conditional test evaluates to `False` (`nil` in AutoLISP).

The syntax of the `if` function is:

```
(if <Test/Comparison>
    <Then Expression>
    <Else Expression>
)
```

The *Test/Comparison* argument of the `if` function is a logical comparison to see if two strings match or a number is greater than, equal to, or less than another number among others. The *Then Expression* is executed when the *Test/Comparison* evaluates as being `True` and the *Else Expression* is executed when the *Test/Comparison* evaluates as being `False`.

The following is an example of the `if` function that compares the values assigned to the variables *A* and *B*, which are assigned the integer values of 7 and 12 respectively. In the example, *A* is greater than *B* so the test condition evaluates as `True` and the *Then Expression* is executed, and the *Else Expression* is ignored.

```
(setq a 7 b 12)
(if (> a b)
    (alert "A is greater than B")
    (alert "B is greater than A")
)
```

The if function can only be used to execute a single AutoLISP expression no matter if the *Test/Comparison* evaluates to True or False. To work around this limitation, the progn function can be used to nest multiple AutoLISP expressions into a single expression.

The following example demonstrates the use of the progn statement to group multiple expressions together:

```
(setq a 11 b 10 c 9)
(if (> a b)
    (progn
        (if (> a c)
            (setq msg "B and C")
            (setq msg "B and not C")
        )
       (alert (strcat "A is greater than " msg))
    )
    (if (> b c)
        (alert "B is greater than A and C")
        (alert "B is greater than A but not C")
    )
)
```

The following comparison operators can be used to test different conditions and values:

- **=** – Returns T if the compared values are equal
- **/=** – Returns T if the compared values are not equal
- **<** – Returns T if the value on the left is less than the one on the right
- **<=** – Returns T if the value on the left is less than or equal to the one on the right
- **>** – Returns T if the value on the left is greater than the one on the right
- **=>** – Returns T if the value on the left is greater than or equal to the one on the right
- **null** – Returns T if the value compared is nil
- **not** – Returns T if the value compared is nil

It is common that you might need to test multiple conditions at a time and make sure all comparisons are True before executing a group of AutoLISP expressions. You can group multiple comparisons together using these logical grouping functions:

- **or** – Returns T if one or more conditionals are True
- **and** – Returns T if all conditionals are True

The following is an example of testing two different comparisons with the `and` logical grouping function; if both evaluate to `True` then the *Then Expression* is executed:

```
(setq a 7 b 12 c 1)
(if (and (> a c) (< a b))
    (alert "A is between B and C")
    (alert "A is outside of the range of B and C")
)
```

The `cond` function is like the `if` function with the exception that you can perform more than one comparison as needed without using nested `if` functions. Each comparison can be followed by any number of AutoLISP expressions without the use of the `progn` function.

The `cond` function has the following syntax:

```
(cond
    ((<test/comparison1>) <Then expressions>)
    ((<test/comparison2>) <Else If expressions>)
    ((<test/comparisonX>) <Else If expressions>)
    (T (<Else expression>))
)
```

The *Test/Comparison* argument is a logical comparison to see if two strings match or a number is greater than, equal to, or less than another number among others. The expressions after the first logical condition that is found `True` are executed. If none of the comparisons are `True`, you can include a default group of expressions that always execute by simply providing a value of `T` for the final logical comparison. If the last logical condition is omitted, and no logical condition is `True` then nothing happens.

Repeating Statements

Repeating statements, or loops as they are commonly referred to as, can be used to execute a group of expressions while a *Test/Comparison* argument evaluates as `True` or until a specified interval is reached. For example, you might use a loop to change the layer of all objects in a selection set or maybe you want to count all the blocks in a drawing to create a basic bill of materials.

The AutoLISP programming language supports these looping functions:

- **`while`** – Executes a group of expressions while a conditional statement is `True`
- **`repeat`** – Executes a group of expressions a specified number of times
- **`foreach`** – Used to step through a list, each item in the list is assigned to a variable which can then be acted upon

The `while` function is the most versatile of the three looping functions and it continues to execute expressions until the *Test/Comparison* evaluates to `False`. The *Test/Comparison* argument of the `while` function is evaluated each time before the expressions inside the while statement are executed. Once the *Test/Comparison* argument evaluates to `False`, the next expression after the `while` function is executed.

```
(while (<test/comparison>)
    (<expressionX…>)
)
```

The following example shows a basic counting loop that uses the `while` function. Each time the `while` function is evaluated, the number assigned to the *`count`* variable is decremented by 1 and a message is displayed in the Command Line history.

```
(setq count 5)
(while (> count 0)
    (prompt (strcat "\nCounting down: " (itoa count)))
    (setq count (- count 1))
)
(prompt (strcat "\nFinal value: " (itoa count)))
```

Executing the statements in the previous example produces the following output:

```
Counting down: 5
Counting down: 4
Counting down: 3
Counting down: 2
Counting down: 1
Final value: 0
```

If you need to execute a set of expressions a specific number of times, the `repeat` function might be more efficient than using the `while` function. There is no *Test/Comparison* argument involved with the `repeat` function, but rather a counter argument that represents the number of times the expressions should be executed.

The syntax of the `repeat` function is as follows:

```
(repeat <counter>
    (<expressionX…>)
)
```

There is no limit to the number of expressions that can be placed inside the `repeat` function. The following example repeats the PURGE command three times to remove nested named objects that aren't used:

```
(repeat 3
  (command "._purge" "_all" "*" "_n")
)
```

Additional Information

You can find additional information on AutoLISP with these topics in the AutoCAD Online Help system:

- AutoLISP Developer's Guide
- Functions Reference (AutoLISP)
- About AutoLISP Applications

E2 Enter AutoLISP Expressions at the Command prompt

This exercise explains how to work with AutoLISP expressions and user-defined variables at the AutoCAD Command prompt.

See the Exercises section in the lab handout.

E3 Create Simple Custom AutoLISP Functions

This exercise explains how to create a basic AutoLISP function which can be executed from the AutoCAD Command prompt.

See the Exercises section in the lab handout.

6 Storing, Loading, and Deploying AutoLISP Files

Typing and executing AutoLISP expressions at the Command prompt in AutoCAD can be helpful when first learning AutoLISP, but it isn't something that is practical for daily use. AutoLISP expressions can be stored in a file with the `.lsp` file extension and can be loaded into AutoCAD whenever they are needed.

You don't need a fancy or expensive programming environment to create a LSP file, all you need is a basic text editor like Notepad on Windows or TextEdit on Mac OS. Microsoft Word or WordPad can be used to create LSP files, but you need to make sure to save the files as plain text without any special formatting. If you are not careful, editing LSP files with Microsoft Word or WordPad could cause problems with the file that might result in it not being loadable in AutoCAD.

The AutoCAD program for Windows does come with a specialized AutoLISP development environment called the Visual LISP Integrated Development Environment (VLIDE). While the VLIDE can make AutoLISP development easier, it has a slight learning curve to it and will require some time to get the most from it. If you want to learn more about the VLIDE, search on the keyword "vlide" in the AutoCAD Online Help system. You can also display the VLIDE by entering **vlide** at the Command prompt in AutoCAD.

In addition to AutoLISP expressions, non-executable text known as *comments* can also be stored in an LSP file. Comments are used to indicate:

- the purpose of an AutoLISP program or individual statements in a program
- when a program was last updated and who might have made the changes

A comment is defined by adding a semi-colon (;) in front of the text that shouldn't be executed. Text to the right of a semi-colon and on the same line is ignored by the AutoLISP interpreter.

The following are examples of comments you might find in an LSP file:

```
; Created on: 9/25/19 by Lee Ambrosius
; Example program created for AU 2019
```

Here is an example of an inline comment placed after an AutoLISP statement:

```
(setq dRad 5 ptCen (list 5 5))  ; Default values for the circle
```

Tip: Adding a semi-colon in front of an AutoLISP expression is a great way to temporarily disable a statement while debugging and trying to identify errors in a program.

If you have a large comment that contains several lines, you can use what is known as a *comment block*. Instead of adding a semi-colon in front of each comment, you can place a comment between the character sequences of `;|` and `|;`.

The following is an example of a comment block you might find in an LSP file:

```
;|  Created on: 9/25/19 by Lee Ambrosius
    Example program created for AU 2019   |;
```

Loading AutoLISP Files

Once you create an LSP file, it can be loaded into AutoCAD by:

- Using the Load/Unload Application dialog box (APPLOAD command) to manually load an LSP file.
- Adding an LSP file to the Startup Suite of the Load/Unload Application dialog box (APPLOAD command). The files listed in the Startup Suite are automatically loaded each time a drawing is created or opened.
- Using the AutoLISP `load` function. For example, you can load the file *Mylsp.lsp* using the statements `(load "mylsp")` or `(load "mylsp.lsp")`.
- Adding AutoLISP functions to the *acad.lsp* or *acaddoc.lsp* files; these files get loaded each time the AutoCAD application is started and/or when a drawing is created or opened. You must create these files yourself and add the files to one of the support search path folders of the AutoCAD program.
- Dragging and dropping an LSP file into the drawing area. (Windows only)
- Adding an LSP file to the LISP Files node of a CUI/CUIx file with the Customize User Interface (CUI) Editor. (Windows only)
- Creating a plug-in bundle that loads the LSP file each time the AutoCAD program starts, or a drawing is created or opened.

Deploy AutoLISP Files with Plug-in Bundles

Plug-in bundles are a relatively new concept to most individuals that extend the functionality of AutoCAD, but they have been available since AutoCAD 2013 when the AutoCAD App Store went live. A plug-in bundle is a folder structure with an XML manifest file named *PackageContents.xml*. The *PackageContents.xml* identifies the files in the plug-in bundle and how those files should be made accessible to the AutoCAD program.

Rather than use a customization (CUI/CUIx) or an *acad.lsp*/*acaddoc.lsp* file to load LSP files, a plug-in bundle can be used to load LSP files instead when a drawing is opened or created. Plug-in bundles are easier to manage and are a safer way to deploy custom programs than with the use of the *acad.lsp* or *acaddoc.lsp* files.

The following is an example of what the folder structure might look like for a plug-in bundle named *Gardenpath.bundle*:

```
Gardenpath.bundle
  |-> DCL
       |-> gpdialog.dcl
  |-> LSP
       |-> ddgpmain.lsp
       |-> gpdraw.lsp
       |-> gp-io.lsp
       |-> gpmain.lsp
       |-> utils.lsp
  |-> PackageContents.xml
```

Here is what the *PackageContents.xml* file might look like for the *Gardenpath.bundle*:

```
<?xml version="1.0" encoding="utf-8"?>
<ApplicationPackage
    SchemaVersion="1.0"
    AppVersion="1.1"
    Name="Garden Path"
    Description="AutoLISP Garden Path tutorial"
    Author="Autodesk, Inc"
    ProductCode="{3f81a8e2-2863-4caf-ba4a-3b25b2e008b6}"
>

  <CompanyDetails
     Name="Autodesk, Inc"
     Url="http://www.autodesk.com"
  />

  <Components Description="All supported operating systems">
    <RuntimeRequirements
       OS="Win32|Win64|Mac"
       SeriesMin="R19.0"
       Platform="AutoCAD*"
    />

    <ComponentEntry Description="Garden Path (Main) LSP file."
       AppName="GardenPathMain"
       Version="1.0"
       ModuleName="./LSP/gpmain.lsp">
    </ComponentEntry>
    <ComponentEntry Description="Garden Path (Draw) LSP file."
       AppName="GardenPathDraw"
```

```
      Version="1.0"
      ModuleName="./LSP/gpdraw.lsp">
    </ComponentEntry>
    <ComponentEntry Description="Garden Path (Input) LSP file."
      AppName="GardenPathInput"
      Version="1.1"
      ModuleName="./LSP/gp-io.lsp">
    </ComponentEntry>
    <ComponentEntry Description="Utility LSP file"
      AppName="UtilityFunctions"
      Version="1.0"
      ModuleName="./LSP/utils.lsp">
    </ComponentEntry>
  </Components>

  <Components Description="Windows supported operating systems">
    <RuntimeRequirements
      OS="Win32|Win64"
      SeriesMin="R19.0"
      Platform="AutoCAD*"
    />
    <ComponentEntry Description="Garden Path (Main) DCL file."
      AppName="GardenPathMainDialogFile"
      Version="1.0"
      ModuleName="./DCL/gpdialog.dcl">
    </ComponentEntry>
    <ComponentEntry Description="Garden Path (Main w/Dialog) LSP file."
      AppName="GardenPathMainDialog"
      Version="1.0"
      ModuleName="./LSP/ddgpmain.lsp">
    </ComponentEntry>
  </Components>
</ApplicationPackage>
```

Note: The `ProductCode` value must be unique for each plug-in bundle that is loaded into the AutoCAD product. You can generate a GUID value by using the Online GUID Generator website (https://www.guidgenerator.com/).

After a plug-in bundle has been defined, it needs to be copied into one of the following locations for it to be loaded into the AutoCAD product:

Product Installation folder

- **Windows 7, Windows 8, and Windows 10:** *%PROGRAMFILES%\Autodesk\ApplicationPlugins*
- **Mac OS:** */Applications/Autodesk/ApplicationAddins*

All Users Profile folder

- **Windows 7, Windows 8, and Windows 10:** *%ALLUSERSPROFILE%\Autodesk\ApplicationPlugins*
- **Mac OS:** *N/A*

User Profile folder

- **Windows 7, Windows 8, and Windows 10:** *%APPDATA%\Autodesk\ApplicationPlugins*
- **Mac OS:** *~/Library/Application Support/Autodesk/ApplicationAddins*

Note: It is recommended to place your plug-in bundles in a folder that is read-only to limit write access to the location by other software.

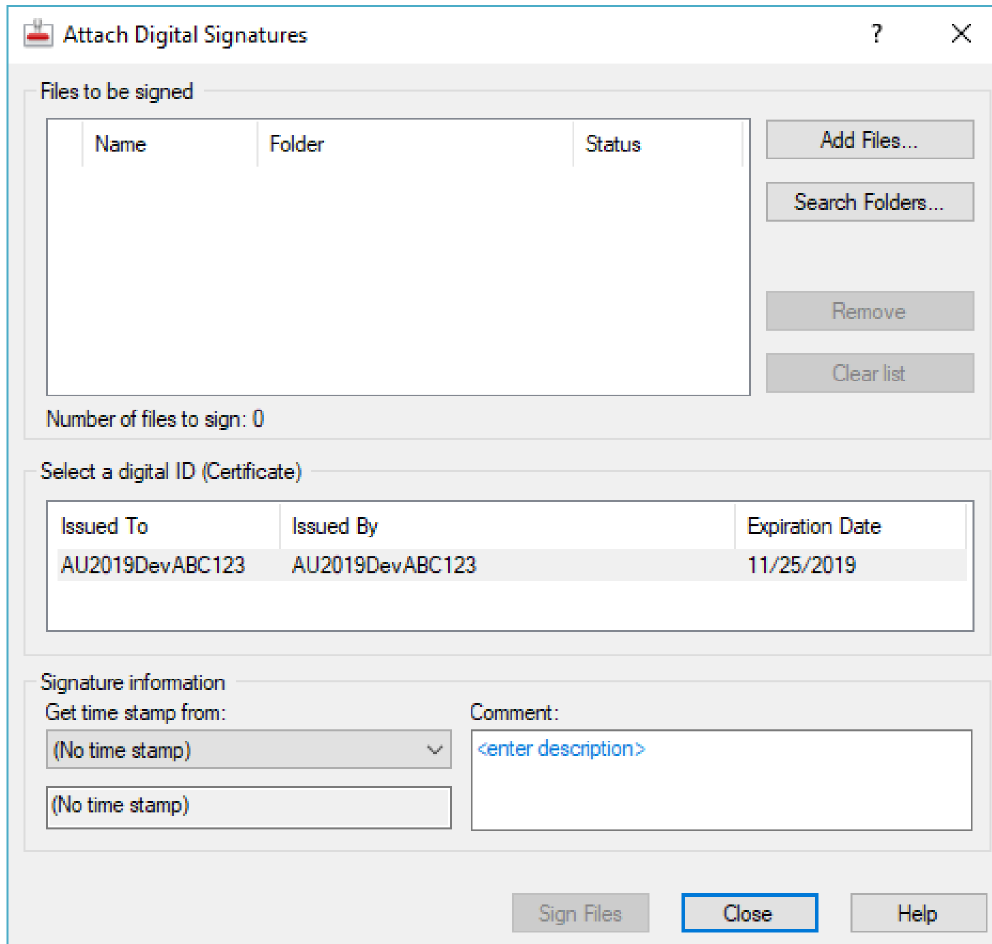
Digitally Sign AutoLISP Files

Starting with AutoCAD 2013, Autodesk has been increasing its efforts to secure AutoCAD and other Autodesk products from executing potentially harmful code. As part of this effort, Autodesk has recommended third-party developers to digitally sign any files that contain executable code, such as archive packages (ZIP and CAB) and binary executables (EXE or DLL) files. Support for and the recommendation of signing AutoLISP files came with the AutoCAD 2016 release.

During the loading of an executable or AutoLISP file, the AutoCAD program checks to see if the file has been digitally signed. Based on the existence of a digital signature, the AutoCAD program will display a warning or informational message about the file and whether it is okay to proceed with loading the file.

Note: The loading of executable files into AutoCAD is controlled by the SECURELOAD, SECUREREMOTEACCESS, and TRUSTEDPATHS system variables.

Most executable files can be signed using a utility such as *SignTool.exe* that comes with Microsoft Visual Studio, but since AutoLISP files are not compiled they need to be signed using a different approach. An LSP file can be digitally signed using the Attach Digital Signatures utility that ships with AutoCAD and can be accessed from the Windows Start menu.



Before you digitally sign an executable, AutoLISP, or even a drawing file, you must obtain a digital certificate. A digital certificate is often obtained from an entity known as a *certificate authority*. A certificate authority is a company that sells and stores digital certificates, so that a digitally signed file can be verified.

Here are some of the certificate authorities that you can use to obtain a digital certificate:

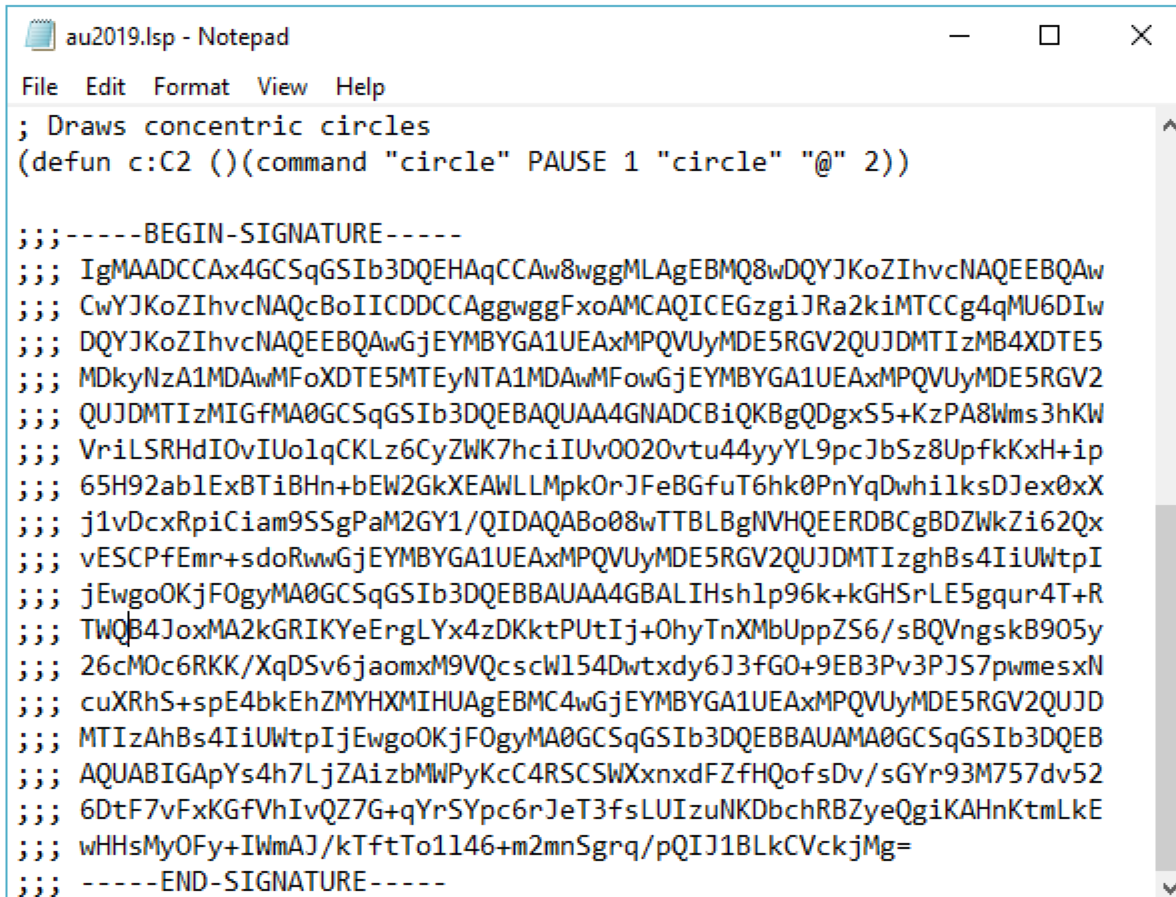
- GlobalSign – https://www.globalsign.com/
- IdenTrust – https://www.identrust.com/
- DigiCert – https://www.digicert.com/

For AutoLISP files that you don't plan to distribute beyond your company, it is possible to create a digital certificate that can be used to sign your files. Steps on creating a digital certificate can be found in the AutoCAD Online Help system under the topic About Digitally Signing Custom Program Files. However, if you plan to distribute files outside your company, it is strongly recommended to obtain a digital certificate from a certificate authority.

Note: The *importDigitalSig.bat* file in the dataset contains an example of how to create and import a digital certificate.

After an LSP file has been digitally signed, a signature block is appended to the end of the file as a comment. Since the digital signature is added as a comment, the removal of the comment will remove the digital signature from the file.

The following is an example of what a digital signature looks like in an LSP file:




```
; Draws concentric circles
(defun c:C2 ()(command "circle" PAUSE 1 "circle" "@" 2))

;;;-----BEGIN-SIGNATURE-----
;;; IgMAADCCAx4GCSqGSIb3DQEHAqCCAw8wggMLAgEBMQ8wDQYJKoZIhvcNAQEEBQAw
;;; CwYJKoZIhvcNAQcBoIICDDCCAggwggFxoAMCAQICEGzgiJRa2kiMTCCg4qMU6DIw
;;; DQYJKoZIhvcNAQEEBQAwGjEYMBYGA1UEAxMPQVUyMDE5RGV2QUJDMTIzMB4XDTE5
;;; MDkyNzA1MDAwMFoXDTE5MTEyNTA1MDAwMFowGjEYMBYGA1UEAxMPQVUyMDE5RGV2
;;; QUJDMTIzMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDgxS5+KzPA8Wms3hKW
;;; VriLSRHdIOvIUolqCKLz6CyZWK7hciIUvOO2Ovtu44yyYL9pcJbSz8UpfkKxH+ip
;;; 65H92ablExBTiBHn+bEW2GkXEAWLLMpkOrJFeBGfuT6hk0PnYqDwhilksDJex0xX
;;; j1vDcxRpiCiam9SSgPaM2GY1/QIDAQABo08wTTBLBgNVHQEERDBCgBDZWkZi62Qx
;;; vESCPfEmr+sdoRwwGjEYMBYGA1UEAxMPQVUyMDE5RGV2QUJDMTIzghBs4IiUWtpI
;;; jEwgoOKjFOgyMA0GCSqGSIb3DQEBBAUAA4GBALIHshlp96k+kGHSrLE5gqur4T+R
;;; TWQB4JoxMA2kGRIKYeErgLYx4zDKktPUtIj+OhyTnXMbUppZS6/sBQVngskB9O5y
;;; 26cMOc6RKK/XqDSv6jaomxM9VQcscWl54Dwtxdy6J3fGO+9EB3Pv3PJS7pwmesxN
;;; cuXRhS+spE4bkEhZMYHXMIHUAgEBMC4wGjEYMBYGA1UEAxMPQVUyMDE5RGV2QUJD
;;; MTIzAhBs4IiUWtpIjEwgoOKjFOgyMA0GCSqGSIb3DQEBBAUAMA0GCSqGSIb3DQEB
;;; AQUABIGApYs4h7LjZAizbMWPyKcC4RSCSWXxnxdFZfHQofsDv/sGYr93M757dv52
;;; 6DtF7vFxKGfVhIvQZ7G+qYrSYpc6rJeT3fsLUIzuNKDbchRBZyeQgiKAHnKtmLkE
;;; wHHsMyOFy+IWmAJ/kTftTo1l46+m2mnSgrq/pQIJ1BLkCVckjMg=
;;; -----END-SIGNATURE-----
```

Note: If the LSP file is changed after being digitally signed, the signature becomes invalid and the file will need to be re-signed.

Additional Information

You can find additional information on creating, loading, deploying and signing LSP files with these topics in the AutoCAD Online Help system:

- Creating, Loading, and Opening an AutoLISP File (AutoLISP)
- About Loading AutoLISP Applications
- About Auto-Loading and Running AutoLISP Routines
- About Installing and Uninstalling Plug-In Applications
- About Digitally Signing Custom Program Files

E4 Create and Load an LSP File

This exercise explains how to create an LSP file and then load it into the AutoCAD program.

See the Exercises section at the end of the lab handout.

E5 Create a Basic Plug-in Bundle to Load an AutoLISP Program

This exercise explains how to define and deploy a custom plug-in bundle.

See the Exercises section in the lab handout.

E6 Digitally Sign an LSP File

This exercise explains how to digitally sign an LSP file to help prevent the intrusion and execution of malware in your environment.

See the Exercises section in the lab handout.

7 User Profiles

AutoCAD user profiles are used to control the behavior of core features, store last known locations of dialog boxes and other user interface elements, specify the color scheme of the application window among much more. Some of the settings that user profiles contain are:

- Search paths used to locate support files,
- Trusted locations used to identify trusted executable files,
- Plot and publish, open and save file options,
- Performance settings related to layouts, zooming, and 3D,
- Colors and fonts used by grips, application, and Command window,
- and many other settings.

Most of the settings that affect the behavior of the AutoCAD program are accessed using the Options dialog box (OPTIONS command). From the Options dialog box, you can create a new user profile and set an existing user profile current on the Profiles tab.

Note: The `/p` command line switch can be used to set a user profile current during the startup of the AutoCAD program from a desktop shortcut. The following is an example of using the `/p` command line switch to set a user profile named “My Profile” current during startup:

```
"C:\Program Files\Autodesk\AutoCAD 2020\acad.exe" /p "My Profile"
```

Profiles are an important aspect of customization as they contain important settings used to identify where your blocks, linetypes, CUIx files, tool palettes, action macros, AutoLISP programs among other custom files are located. You specify the locations in which the AutoCAD program should look for custom files on the Files tab of the Options dialog box. It is recommended to store custom files in a centralized location, such as a network drive to make it easy to not only back up the files but also to share files with others in your organization. Updated files that are placed in the AutoCAD support file search paths are loaded automatically when the AutoCAD program starts, or when a drawing is created or opened.

Note: Starting with AutoCAD 2014, you must identify not only which folders the AutoCAD program can locate custom files, but also which folders are safe to load custom programs from. The designation of which folders are safe to load custom programs from are known as *trusted locations*. Trusted locations are specified under the Trusted Locations node on the Files tab of the Options dialog box. If you try to load a custom program from a folder that isn't trusted, a warning message is displayed about the safety of the file.

Tip: You can create and export a user profile for use on other workstations with the Export button on the Profiles tab of the Options dialog box (OPTIONS command). When a profile is exported, the AutoCAD program creates an ARG file that contains an exported branch of the Windows Registry. After a profile has been exported, the profile can be imported using the Import button on the Profiles tab of the Options dialog box or the `/p` command line switch.

Additional Information

You can find additional information on user profiles with these topics in the AutoCAD Online Help system:

- About Saving Program Settings as Profiles
- About Exporting and Importing Custom Settings from the Same Release
- About Resetting the Application to the Default Settings

E7 Create and Modify a New Profile

This exercise explains how to create a new profile, set the new profile current, and then make changes to the profile.

See the Exercises section in the lab handout.

E8 Reset the AutoCAD Environment

This exercise explains how to reset AutoCAD back to its default settings and remove the custom plug-in bundle.

See the Exercises section in the lab handout.

8 Exercises

See the *ClassHandout-AS322616-L-Ambrosius-AU2019.pdf* file for the exercises associated with this session.

9 Where to Get More Information

When you are first starting to learn a new skill, you will have questions and where you go to find those answers might not be clear. The following is a list of resources that can be helpful:

- **AutoCAD Help System** – The Customization Guide in the AutoCAD Online Help system contains a lot of information on creating script files, recording action macros, and working with user profiles. The Developer Home page in the AutoCAD Online Help system can be helpful in locating information about AutoLISP and other programming options that AutoCAD supports.

  To access the online help, go to: https://help.autodesk.com/view/OARX/2020/ENU/

- **Autodesk Discussion Forums** – The Autodesk discussion forums provide peer-to-peer networking that allows you to ask a question about anything related to AutoCAD and get a response from a fellow user or Autodesk employee. To access the Autodesk discussion forums, go to https://forums.autodesk.com, click Browse By Product near the upper-right of the page and then click AutoCAD. Click the appropriate subgroup link.
- **AUGI Forums** – The AUGI forums provide peer-to-peer networking where you can ask questions about virtually anything in AutoCAD and get a response from a fellow user. Visit AUGI at https://www.augi.com
- **Industry Events and Classes** – Industry events such as Midwest University and Autodesk University are great places to learn about new features in an Autodesk product. Along with industry events, you might also be able to find classes at your local technical college or Autodesk Authorized Training Center (ATC).
- **Internet** – There are tutorials on the Internet that can be helpful to learn many of the customization and programming options supported in the AutoCAD program. Use your favorite search engine, such as Google or Bing.
- **Books** – There are many general and specialized books that cover AutoCAD customization and programming. To find a book, use Amazon (amazon.com) or Barnes & Noble (bn.com) to locate a book online or visit your local Barnes and Noble store. My latest book, AutoCAD Platform Customization: User Interface, AutoLISP, VBA, and Beyond, covers all the customization options mentioned in this session and many more.