# Creo Elements/Direct Drafting Writing Macros

**Creo Elements/Direct Drafting 20.4.4.0**

# Contents

# Preface

Creo Elements/Direct Drafting is a versatile 2D design and drafting system for optimizing each stage of the design process. Using Creo Elements/Direct Drafting you can quickly and easily create and modify 2D drawings.

The purpose of this manual is to explain how to write macros for use with Creo Elements/Direct Drafting. Macros can speed up and automate many of your day-to-day tasks.

# Who Should Use This Manual

Read this manual if you are:

- A Systems Administrator

  You are an experienced systems administrator, with experience in managing UNIX-based and Windows-based operating systems. You are familiar with startup files and customization files. You will use macros to simplify customizing procedures for use by your group.

- Designer or Draftsperson

  You are an experienced Creo Elements/Direct Drafting user. You perform many repetitive tasks manually, and you want to know how to write macros to automate these tasks. You do not need previous programming experience.

# Purpose of this Manual

This manual gives information on how to do the following:

- Create geometry.
- Extract specifications from data files.
- Communicate with engineering program languages such as Pascal or C to make complex calculations. (UNIX-Based Systems Only)
- Label drawings.
- Create parts lists.
- Customize startup procedures.
- Control screen menu display and function.
- Customize the keyboard.
- Customize text fonts.

# How to Use This Manual

Systems Administrator

Here are some suggestions on how you should use this manual:

1. Read Chapters 1 through 4 for general information on writing macros.
2. Read Chapter 12 if you want to relate a macro command or function to a task that you do manually with the user interface.
3. Read Chapter 13 for information on customization.
4. Refer to Chapter 14 for a brief description of a command or function.
5. Read Appendix A for information on logical and display tables.

Designer or Draftsperson

Here are some suggestions on how you should use this manual:

1. Read Chapters 1 through 4 for general information on writing macros.
2. Read Chapter 5 to refresh your memory on vectors.
3. Read Chapters 6 through 11 for specific details on tasks that you will use in your normal work.
4. Read Chapter 12 if you want to relate a macro command or function to a task that you do manually with the user interface.
5. Read Chapter 13 if you want to try customizing your own environment.
6. Read Chapter 14 for a brief description of a command or function.

# How This Manual is Organized

| Chapter 1 | `What is a Macro?` contains an introduction to macros, and shows you how to store, run, and debug your macro. |
|---|---|
| Chapter 2 | `Using the Editor to Write a Macro` explains how to use the built-in editor to write your macros. |
| Chapter 3 | `Macro Basics` gives information on how to use syntax diagrams, local and global variables, control statements, parentheses, and defensive programming. |
| Chapter 4 | `Inquiring about the Environment and Elements` shows you how to access environment information such as current units and current linetype, and how to restore any environment information you change during the operation of your macro. |
| Chapter 5 | `Quick Review of Points and Vectors` brings you up to the level of knowledge you need for macros. |
| Chapter 6 | `Writing Geometry Macros` shows you how to do the initial vector analysis and how to incorporate the analysis into your macro. |
| Chapter 7 | `File Input/Output and Text Strings` shows you how to handle string data and how to extract strings from ASCII files. |
| Chapter 8 | `Using Dimensions Stored in a Data File` shows you how to create geometry using tabulated data from files. |
| Chapter 9 | `Useful Macros` gives a selection of macros you can use in your everyday work. |
| Chapter 10 | `Recording the System Operation` shows you how to record the activities of Creo Elements/Direct Drafting operations, and how to use the record to help you write a macro. |
| Chapter 11 | `Using the Interface to Find a Command` shows you how to relate a macro command to a task that you perform on the user interface. |
| Chapter 12 | `Customizing` shows you how to customize such things as startup, the environment, the keyboard, and text fonts. |
| Chapter 13 | `Brief List of Commands and Functions` gives a 1-line or 2-line description of each command and function. |
| Appendix A | `Logical and Display Tables` provides you with the information needed to use logical and display tables. |

# Online Help

For complete descriptions and syntax of all commands and functions, use the online help facility.

For example, to get help on the `MODIFY` command, enter at the applications command line:

```
help modify
```

The screen will clear and Creo Elements/Direct Drafting will display further information about modifying your drawing.

# Typographical Conventions

This manual uses the following typographical conventions:

**Table 1. Conventions Used in This Manual**

| Convention | Represents |
|---|---|
| Bold | Menu paths, dialog box options, buttons, and other selectable elements from the user interface. For example: **LINE GRID**, **SHOW LAYER**, **ZOOM**, and so on. |
| Courier | User input, system messages, directories, and file names. |

# 1

# What is a Macro?

This chapter gives a brief introduction to macros, and shows you how to store, run, and stop a macro.

# Why Use a Macro?

A macro is a quick and easy method of executing a sequence of commands automatically. Little or no user input is required. You should consider writing a macro for any sequence of commands that is frequently used by ME-CAD users.

For example, at least once a day you log out from your workstation. Before you log out, you store your drawing and then type `exit confirm` on the command line.

You can automate this procedure using the following macro, called `Quit`:

```
DEFINE Quit
{###############################################}
{##   This macro stores your current       ##}
{##   drawing in 'filename', then ends      ##}
{##   your ME-CAD session.                  ##}
{###############################################}

  STORE ALL DEL_OLD 'filename'
  EXIT CONFIRM
END_DEFINE
```

You can see that the macro contains the sequence of commands that you enter before you log out. The comments at the start of the macro help a user to understand the macro.

The commands are stored in a file and executed when the macro is run. You run the macro by typing the macro name on the command line, or by picking a slot on screen menu.

Some major applications for macros are as follows:

- Creating geometry.
- Making calculations.
- Labeling drawings.
- Creating parts lists.
- Controlling screen menu display and function.

Before we discuss macros in detail, we will show you how to type your macros in the built-in editor, and how to store macros.

You can create macros in two ways:

- With your favourite editor.
- With the screen editor that is built into your ME-CAD system software.

The built-in editor is similar to PC-based editors. All the examples in this chapter will be done with the built-in editor.

# Creating a File for Your Macros

Before you write a macro, you must first create a file to store it. For example, you might want to create a file called `cad_mac.m`. From within the ME-CAD environment you type on the command line:

`EDIT_FILE 'cad_mac.m'`

If this file already exists, the system displays the file and you can add your macro. If the file does not exist, the system creates an empty file. You type in your macro as described later in Using the Editor to Write a Macro on page 25.

# Storing Your Macro

To store your macro, and return to the ME-CAD screen, press `[Ctrl] [D]`. The file is stored on your system disk. This is shown by the following message at the bottom of your screen:

`writing 'cad_mac.m'`

If you list your current directory, `'cad_mac.m'` appears in the list.

If you alter your file in the editor, you might decide that you don't want to keep the changes. Hit `[ESC]` or `[Break]` to exit from the editor, and the changes will not be written to disk.

You can add as many macros as you want to a file. The sequence of the macros in the file is not important. But placing the macros in alphanumeric sequence by name makes it easy to find a macro.

When you are writing and debugging a new macro, you may want to write this new macro at the beginning of the file. The first page of a file is displayed when you enter the editor using the `EDIT_FILE` command, so you don't need to scroll through the file to find the macro. Later, when the macro is completely debugged, you can move it to its correct alphabetical position in the file.

Another way to write and debug a new macro is to create a separate file for the new macro. When the macro is fully debugged, you can append it to your normal macro file. This method has two advantages:

- When you use the `EDIT_FILE` command, your text is displayed faster.
- The `INPUT` command executes faster. (The `INPUT` command compiles and loads your macro, and is discussed in the next section).

It is advisable to keep separate files for each macro. You can then give the file the same name as the macro. If your macro calls other macros, you may have to input these other macros separately.

There is another method for storing less frequently used macros. Group similar macros in one file. For example, all bolt macros could be stored in the file `'bolt.m'`, flange macros in `'flange.m'`. Use macro names that are common to all files. For example, the first bolt macro and the first flange macro are both

called `macro1`, the second bolt macro and the second flange macro are both called `macro2`, and so on. Load each file as it is needed. Each time a file is loaded, macros of the same name are overwritten. For example, the bolt `macro1` overwrites the flange `macro1`, and so on. This means that valuable memory space is not used by infrequently-used macros.

When you use the `INPUT` command within a macro, it must be used with the qualifier `IMMEDIATE`, see also .

# Deleting Your Macro

You can delete an individual macro from RAM using the `DELETE_MACRO` command. This command can be useful if you need more memory space to load a large drawing. For example, if you want to delete the macro called `Quit`, use the following command:
`DELETE_MACRO Quit`

# Running Your Macro

A macro must be compiled and loaded in RAM before you can run it. You compile and load the macro by typing on the command line:
`INPUT 'cad_mac.m'`

Now that you have input the file, there are two ways to run a macro from this file. These two methods are described now.

## Running a Macro from the Command Line

You can run any macro by typing the macro name on the command line. If the file `cad_mac.m` contains the macro `Quit`, you can run the macro by typing on the command line:
` Quit`

The macro is executed line by line. The commands and statements are checked as they are executed. This is similar to "interpreted" versions of the Basic programming language. If an error is found, execution stops and a message is displayed.

# Debugging Your Macro

If the macro does not run, you must edit the macro with the editor. The procedure for editing a macro is the same as for creating a macro: use `EDIT_FILE` to enter the file, and then make alterations.

When the file is displayed in the editor, you will see the first page of the file. Use the arrow keys to scroll to the macro you want to alter. Examine the macro line by line until you find the error. Most errors are simple, such as missing parentheses and missing underscores.

If you cannot find the error quickly, debug your macro using the BREAKPOINT function. Insert a BREAKPOINT in desired position in macro code. This can be done either by editing the file on disk or by using EDIT_MACRO function. When the code reaches a BREAKPOINT, the system goes into debug mode.

The breakpoint function is enabled by default. If you find that it is not working, check that ENABLE_BREAKPOINT is set to ON.

To enable the BREAKPOINT function in Creo Elements/Direct Drafting:

• Click **Miscellaneous** and then, in the **System** group, click the **Breakpoints** check box.

When you use a breakpoint and they are enabled, the system will display a parameters table.

If already in debug mode and you have moved forward at least one token using debugger command, BREAKPOINT keyword will set a breakpoint in current position.

If you insert a BREAKPOINT between PARAMETER and LOCAL in a macro, the debugger will break before the first command in macro.

Note that the breakpoint will be set for the current OSD session only, it will not be added to correspondent file on disk.

Once in debug mode, use the following commands to step through your macro and isolate the area where it is not working:

• STEP_NEXT: Use this action to step through macro code one token at the time.

• STEP_OUT: Step out of current macro function.

• STEP_OVER: Step over the next token even if it is a macro function.

• STEP n: Step over several tokens in one step.

• SKIP n: Skip current token several times.

• CONTINUE: Continue with macro execution until the next breakpoint.

• GO: Execute the rest of the code disregarding all breakpoints.

• REMOVE_BREAKPOINT: Remove current breakpoint.

• ENABLE_BREAKPOINTS: Enable all breakpoints.

• LIST_BREAKPOINTS: List all breakpoints in an editor.

When you debug with BREAKPOINT, be aware of the following:

- Use the `DISPLAY` command to see variable values as you step through your macro.

- Any Creo Elements/Direct Drafting function can be executed while in break state.

- Creo Elements/Direct Drafting commands will interrupt the macro.

- When any macro in the call stack is modified, due to `INPUT, EDIT_MACRO, DELETE_MACRO, or DEFINE` commands, the debugger will also be interrupted.

- The debugger does not include a user interface. See the command prompt for additional information, such as current command, last token, next token, and next command.

- The debugger does not include a viewer for your code. To see your code as you debug, use an external viewer.

When you have found the error and corrected it, press `[Ctrl] D` to return to the ME-CAD screen. The new version of the file overwrites the old version on the disk.

The compiled version of the file in RAM remains unchanged. The only way you can alter the compiled version is to load the new version of the file into RAM using the `INPUT` command. A common error is to edit the file and then run the macro without using the `INPUT` command. This runs the old version of the macro.

Input the file again by typing:
```
INPUT 'cad_mac.m'
```

---

📋 **Note**

If you want to repeat commands you have typed previously, press the `[PgUp]` key. If you want to enter a command that is similar to a previous command, press `[PgUp]` key, and then edit the command.

---

The contents of `cad_mac.m` are reloaded from the disk and overwrite the old copy in RAM. You can now re-execute the macro. A macro can be executed as often as you like after the file is loaded.

To summarize, when you are writing and debugging a macro called `Quit` in a file called `cad_mac.m`, use three steps:
```
EDIT_FILE 'cad_mac.m '
INPUT 'cad_mac.m '
Quit
```

There is a `trace` facility that is useful for debugging macros. This is described in Using the Trace Facility on page 52.

# Stopping a Macro

If you want to stop your macro for any reason, such as an endless loop, use the `[Ctrl] [Break]` key. The effect of `[Break]` depends on your operating environment.

---

### 🗩 Note

If you press `[Break]` or `[Again]` when starting Creo Elements/Direct Drafting with your workspace menu or when you are in the graphics screen, the interrupt will not take effect until the system is ready to accept input.

For more information, refer to your Windows documentation.

---

# 2

# Using the Editor to Write a Macro

This chapter shows you how to use the built-in text editor to write your macros. You will learn how to:

- Copy blocks of lines from one part of a file to another.

- Load from other files into the current file.

- Find the next occurrence of a string.

- Replace all occurrences of a text string with another text string.

- Right justify text.

# Using the Keyboard Editing Keys

The simple editing keys on your keyboard can still be used within the editor. These keys include the following:

- `[Insert char]` or `[Ins]`
- `[Delete char]` or `[Del]`
- `[Clear line]` or `[Alt] [Del]`
- `[Clear display]` or `[Control] [Del]`
- Arrow keys: `[<] [>] [^] [v]`

# How to Enter and Leave the Editor

You can only enter the editor while you are in an ME-CAD environment. To enter the editor, type `EDIT_FILE` followed by the name of the file you want to edit, in single quotes. For example, if you want to edit a file called `'macros'`, type on the command line:

`EDIT_FILE 'macros'`

If the file exists, it will be displayed on the editor screen. If it does not exist, the editor screen will be blank.

If you want to leave the editor, and save the changes to the file, press `[Ctrl] [D]`

If you want to leave the editor, but you do not want the changes to be written to disk, press `[ESC]` or `[Ctrl] [Break]`

# Using `EDIT_PORT` to Enter the Editor Quickly

The `EDIT_PORT` command causes a small viewport to be opened on your ME-CAD screen. This viewport is used by the editor and only appears when you invoke the editor. The smaller you make the viewport, the faster you can enter and leave the editor, because screen redraw times are less.

To use the command, type `EDIT_PORT` on the command line. You will see the prompt:

`Enter background color or border width or corner of new port`

Pick the top left hand corner of your ME-CAD screen, followed by the diagonally opposite corner of the required viewport.

To test the result, type `EDIT_FILE` on the command line, followed by any file name in single quotes. If the viewport is not the correct size, try again. The viewport sizes remain in effect until you change them with another `EDIT_PORT` command, or until you terminate your ME-CAD session with `EXIT CONFIRM`. The next time you use your ME-CAD system, you must use `EDIT_PORT` again.

`EDIT_PORT` is useful when adding text to drawings. Such text is normally only a few words. If you make a tiny viewport 8 cm wide by 2 cm high in the top left hand corner of your ME-CAD screen, you will notice very fast redraw times.

Note that `HELP_PORT` works the same way. `HELP_PORT` speeds up access to the `help` file by using a small viewport to minimize screen redraws.

# How to Set and Use Markers

Many editor commands described later will not work properly without markers. For example, if you want to copy a block of text from one part of your file to another, you must mark either the beginning or the end of the block.

A marker is a character that identifies a line, but remains invisible in the file. The system sets some markers automatically.

The following table shows the characters that are used as markers:

| Markers | Use |
|---------|-----|
| ^ | Set by the system to mark the first line of a file. |
| ! | Set by the system to mark the last line of a file. |
| 0...9 | Set by the user to mark any line of a file. |

Let's try using some markers. From within your ME-CAD screen, create a new file by entering on the command line:
```
EDIT_FILE 'markers'
```

Type some lines in the file, as follows:
```
AAAAAAAAA
BBBBBBBBBBBB
CCCCCCCCCCCC
DDDDDDDDDDDD
EEEEEEEEE
FFFFFFFFF
```

We have made the lines of `B`s, `C`s, and `D`s longer than the other lines for easy identification later.

Now try using the system-defined markers to move to the beginning and end of the file. Here's how to do it.

Type $ (do not press [Enter]). The cursor moves to the bottom of the screen with the $ symbol in front of it. The $ is the default escape character, which must be typed before any of the editor commands can be typed.

Now type ^ and press [Enter]. The cursor moves to the first line of the file.

Type $! and press [Enter]. The cursor moves to the last line of the file.

You can set markers at any line in this file. Place the cursor on the line you want to mark and enter:
```
$sm n
```

The $ symbol is the default escape character. sm is the set marker command. n is a number from 0 through 9. The blank between sm and n is optional. The line will be marked with that number, but the mark will not appear on the screen.

As an example, let's set marker 1 at the line of Ds. Move the cursor within the line of Ds, then enter the following:
```
$sm1
```

Now move the cursor to another line and enter:
```
$1
```

The cursor jumps to the line of Ds.

# Copying Text

Now let's try copying a block of text. We want to copy the three lines of Bs, Cs, and Ds and place these three lines after the line of Es. Here's how to do it:

• To define the block of three lines, you can mark the line of Bs and then place the cursor on the line of Ds. Or you can mark the line of Ds and then place the cursor on the line of Bs. We already have marker 1 on the line of Ds, so let's use this marker.

• To insert the text after the line of Es, set a marker at the line of Es. Move the cursor within the line of Es, then enter the following:
```
$sm2
```

• Place the cursor anywhere on the line of Bs.

Remember that the markers are invisible. But if we could see the markers, the file might look like this:
```
AAAAAAAAA
BBBBBBBBBBBB    * (cursor line)
CCCCCCCCCCCC
DDDDDDDDDDDD    1 (marker)
EEEEEEEEE      2 (marker)
FFFFFFFFF
```

Just to remind you: we want to copy the lines of Bs, Cs, and Ds and place these three lines after the line of Es. Now enter the following:
```
$c12
```

The block of text from the current cursor position to marker 1 will be copied after marker 2. The file now looks like this:
```
AAAAAAAAA
```

```
BBBBBBBBBBB
CCCCCCCCCCC
DDDDDDDDDDD
EEEEEEEEE
BBBBBBBBBBB
CCCCCCCCCCC
DDDDDDDDDDD
FFFFFFFFF
```

# Using the Editor Commands

This section describes all the editor commands. The following terminology is used in the descriptions:

- An item enclosed in square brackets `[...]` is optional.
- An item enclosed in single quotes `'...'` means the string you type must be enclosed in single quotes.
- An item in italics is a generic description of the item to be typed. For example, the command:
  `$marker`

  means that you must type a numeric value for marker, such as `2`.

  As another example, the command to load a file is:
  `L 'filename'`

  This means you must type the name of a file (in single quotes) after the `L`. So, if the file to be loaded is `cad_mac.m`, you must type:
  `L 'cad_mac.m'`
- The current line is the line containing the cursor.

The next table shows the full set of editor commands. Before typing any editor command, you must type `$`. The cursor moves to the bottom of the screen with the symbol `$` in front of it. Remember that `$` is the default escape character.

| Command | Effect of Command |
|---|---|
| `marker` | Moves the cursor to the line containing the specified marker. For example, entering `$ 4` moves the cursor to the line containing marker number `4`. |
| `'string'` | Moves the cursor to the next occurrence of the specified string. For example, entering `$ 'SolidDesigner'` moves the cursor to the next occurrence of the string `SolidDesigner`. |
| `? ['keyword']` | This displays the system `help` file at the section describing the keyword. If you enter `?` without a keyword, the section explaining the screen editor is |

| Command | Effect of Command |
|---|---|
| | displayed. To return to your file press `[Ctrl] [D]`. |
| `AC` | Adjust Center. For each line of a paragraph, this command centralizes the text between the margins. |
| `AF` | Adjust Fill. This command flows a paragraph between the margins. The left edge is justified and the right edge is ragged. |
| `AJ` | Adjust Justify. This command flows a paragraph between the margins. Left and right edges are justified. This means that the spacing between words will vary. |

📋 **Note**

Do not use `AF` or `AJ` to edit macros as each line will be joined to the end of the previous line. These commands should be used only for editing text.

| Command | Effect of Command |
|---|---|
| `C marker1 marker2` | Copy. Copies all text lines between the current line and `marker1` (inclusive) and inserts them after `marker2`. For example, entering `$C34` will copy all the text between the current line and `marker 3` and insert it after `marker 4`. |
| `D marker` | Delete. Deletes all text lines between the current line and the marker (inclusive). |
| `H ['keyword']` | Help. This is the same as the ? command. |
| `L 'filename'` | Load. Copies all text lines from the specified file and inserts them after the current line. For example, entering $L 'cad_macros' will load the file cad_macros after the current line. |
| `M marker1 marker2` | Move. Moves all text lines between the current line and `marker1` (inclusive) and inserts them after `marker2`. Source and destination must not overlap. For example, entering `$M67` will move all text between the current |

| Command | Effect of Command |
| --- | --- |
| | line and `marker 6` and place the text after `marker 7`. |
| `N` | Next. Repeats the last string search. |
| `O 'filename' [marker]` | Overwrite. Copies all text lines between the current line and the marker (inclusive), and writes them to a file with the specified name. If no marker is specified, the cursor position is ignored and the whole file is copied. Any existing file with the specified name is overwritten. For example, entering `$O 'cad_macros' !` will copy the text between the current line and the last line (inclusive) into a file called `cad_macros`. If you omit the `!`, the whole file is copied. |

| Command | Effect of Command |
| --- | --- |
| `R [V] ['string1'] ['string2'] [marker]` | Replace. Replaces all occurrences of `string1` with `string2` between the current line and the marker (inclusive). The V lets you verify each change of `string`. If you omit the marker, only one replacement is made. If either string is omitted, the default is taken as the corresponding last-used `string`. For example, if you enter `$RV 'black' 'white' !` |
| | the cursor moves to the next occurrence of `black`. You then have the option of replacing it by `white`, or moving to the next occurrence of `black`: |
| | R replaces, |
| | '' does not replace, |
| | S aborts. |
| | Pressing R replaces the `black` by `white`. Pressing the space bar moves to the next occurrence of the `black`. Pressing S aborts the operation. |
| `SE 'character'` | Set Escape. Redefines the escape character. This character causes the cursor to jump from |

| Command | Effect of Command |
|---|---|
| | your file to the editor command line. The default escape character is $. |
| | You can use any character except `0 ... 9, A ... Z, a ... z, ... !` and `?`. A new escape character remains in force until redefined or until the system is switched off. |
| | If you wish to define the escape character as `#`, enter<br>`$ SE '#'` |
| `SL` | Set Left margin. Sets the left margin to the current cursor position. |
| `SM n` | Set Marker. Sets the marker specified at the current line. `n` can have any value from `0` through `9`. The space between `SM` and `n` is optional. For example, entering `$ SM7` sets the marker 7 at the current line. |

| Command | Effect of Command |
|---|---|
| `SR` | Set Right margin. Sets the right margin to the current cursor position. |
| `W 'filename' [marker]` | Write. Copies all text lines between the current line and the marker (inclusive) and writes them to a file with the specified name. If no marker is specified then the whole file is copied. If a file with that name already exists then the operation is aborted. For example, to copy the text between the current line and marker 6 to a file called `cad_macros`, enter the following:<br>`$ W 'cad_macros' 6` |

## 📝 Note

`SL`, `SM` and `SR` will not adjust the text until you use `AF` or `AJ`. Remember that `AF` and `AJ` are not normally used with macros.

# Using `EDIT_MACRO`

In this chapter, all our work on macros has been done using `EDIT_FILE`. The `EDIT_FILE` command is useful because you can examine any file using this command; the file need not contain a macro.

The `EDIT_MACRO` command can be used only for macros.

We will first explain the `EDIT_MACRO` command, and then compare it with `EDIT_FILE`. After that, you can decide which command you want to use with your macros.

Before you can use the `EDIT_MACRO` command, the macro must already exist in RAM. There are two ways to put a macro in RAM. We will discuss each of these methods in the next few paragraphs.

## Using the `INPUT` Command with an Existing File

You are already familiar with this method, since this is the method we have used throughout this chapter.

When you use the `INPUT` command within a macro, it must be used with the qualifier `IMMEDIATE`, see also INPUT on page 38.

## Typing the Macro on the Command Line

With this method, you type `DEFINE` followed by the name of the macro. For example, if your macro is called `Slot_mac`, then type:
```
DEFINE Slot_mac
```

The system responds with the prompt:
```
Enter macro definition
```

Now type in the remaining lines of your macro, one at a time. Each time you press `[Enter]`, the system responds with the prompt:
```
Enter macro definition
```

You can now run the macro by typing the name of the macro on the command line. Note that with this method you do not have to use `INPUT`.

When a macro is in RAM, you can edit the macro by typing `EDIT_MACRO` on the command line, followed by the name of the macro. For example, you can type:
```
EDIT_MACRO Slot_mac
```

The macro will appear in the editor, and it can be edited as if you had used the `EDIT_FILE` command. To leave the editor, press `[Ctrl] [D]`. To run the macro, type the macro name on the command line. You do not have to use the `INPUT` command.

Note that if you enter the editor using `EDIT_MACRO`, the macro is not saved by pressing `[Ctrl] [D]`. To save the macro, type `SAVE_MACRO` on the command line, followed by the name of the macro.

The system responds with the prompt:
`Enter SCREEN, DEL_OLD, APPEND or 'file name'`

Type `SCREEN` if you want to view the macro in the editor. Type `DEL_OLD` to overwrite an existing file of the same name. Type `APPEND` to add the macro to the end of an existing file. For a new file, type the 'file name'.

## Comparison of `EDIT_FILE` and `EDIT_MACRO`

`EDIT_MACRO` is superior if you need several attempts to debug a macro, because you do not have to use `INPUT` after each time that you make changes.

The disadvantage of `EDIT_MACRO` is that the macro is not automatically saved when you use `[Ctrl] [D]` to exit from the editor. If you leave your Creo Elements/Direct Drafting environment using `EXIT CONFIRM`, your macro may be lost.

# 3

# Macro Basics

This chapter describes the structure of a macro, shows the differences between local and global variables, explains when to use parentheses, and shows you how to use the `trace` facility to debug your macros.

# What Does a Macro Consist Of?

A macro can consist of six sections as follows:

- DEFINE macroname
- Parameters
- Local Variables
- User Input
- Macro Body
- END_DEFINE

A macro need not contain all of these sections. In the following example, you can see each of the six sections:

```
DEFINE Circle_mac

PARAMETER P2
LOCAL P1

  READ 'Indicate center of circle' P1

  CIRCLE CENTER P1 P2 END

END_DEFINE
```

This macro is activated by typing `Circle_mac` on the command line, followed by a value for the radius of the circle. This is obviously not a very useful macro, but in only a few lines we can show all the parts of a macro.

## DEFINE

This line always starts with the word `DEFINE`, followed by the name of the macro. In our example, the name of the macro is `Circle_mac`:
```
DEFINE Circle_mac
```

Note the use of upper and lower case when writing macros. We use upper and lower case to differentiate between macro names and variables, and commands and functions.

Keywords such as commands and functions are all upper case. For example, `DEFINE`, `LOCAL`, and `END_DEFINE` are keywords.

The system accepts a keyword typed in upper case or lower case, but not mixed case. For example, the system accepts `DEFINE` or `define`, but not `Define`, `dEFINE`, `DeFiNE`, or any other mixed case version.

The first letter of a macro name is normally upper case. So if a term such as `Circle_mac` appears in a macro listing you will know that `Circle_mac` is a macro and not a keyword.

We will always use upper and lower case correctly in this chapter, but you are free to type whatever is easier. For example, we may ask you to type on the command line:

```
EDIT_FILE 'cad_macros'
```

Here, you can type:

```
edit_file 'cad_macros'
```

---

📋 **Note**

Any string in single quotes must be typed exactly as shown.

---

## Parameters

This section defines variables that are passed to the macro as arguments. If there is more than one parameter, they must be listed in a specific sequence. For more information on parameters, refer to File Input/Output and Text Strings on page 81. There is only one parameter statement in our example macro:

```
PARAMETER P2
```

## Local Variables

Variables that exist only within the macro are defined here. Any variables that appear in the user input section of the macro, or the body of the macro, are normally defined as local variables. We will discuss local variables in more detail later in this chapter. There is only one local variable in our previous example:

```
LOCAL P1
```

## User Input

Variables that are unknown to the processor at compile time must be supplied by the user at run time. In the following example, the value for P1 is supplied by the user:

```
  READ 'Indicate center of circle' P1
```

The system displays the prompt:

```
Indicate center of circle
```

The next value entered by the user will be assigned to the variable P1. The user can type in the x, y coordinates of the point, or pick the point. In either case, the coordinates are assigned to the variable P1.

Local variables must be defined at the beginning of the macro. However, a programmer might prefer to define these variables after writing the rest of the macro.

## Macro Body

The body of the macro contains the executable code. This is where the work gets done. The start of each line contains a command or function followed by options, expressions, or operators. These terms will be discussed later in this chapter. Here is the body of our example macro:

```
CIRCLE CENTER P1 P2 END
```

## END_DEFINE

This statement marks the end of the macro.

## INPUT

When you use the `INPUT` command within a macro, it must be used with the qualifier `IMMEDIATE`, for example:

```
DEFINE xyz
  LINE 0,0 1,1 END
  INPUT IMMEDIATE 'filename'
  ....
  ....
END_DEFINE
```

# Minimum Macros

We have seen that a macro can consist of six sections. But not all macros need six sections. If the macro does not accept arguments, there will be no `PARAMETER` section. If the macro does not use variables, there will be no section defining local variables. There may be no user input section.

The absolute minimum macro cannot have less than three sections:

• DEFINE

• Body

• END_DEFINE

The macro at the beginning of this chapter is an example of a minimum macro. Here it is again:

```
DEFINE Quit

{##############################################}
{##   This macro stores your current         ##}
{##   drawing in 'filename', then ends       ##}
{##   your ME-CAD session.                   ##}
{##############################################}

  STORE ALL DEL_OLD 'filename'
  EXIT CONFIRM
```

```
END_DEFINE
```

Note that we write statements on separate lines to make them easier to understand. The computer understands the macro even if you write it as:
```
DEFINE Quit STORE ALL DEL_OLD 'filename' EXIT CONFIRM END_DEFINE
```

# Syntax Diagrams

This is a good time to introduce you to syntax diagrams. It will help you understand why the computer does not care whether you write a sequence of commands on separate lines or all on the same line.

If you study the syntax diagram for DEFINE, you can see why the computer understands the macro. The syntax diagrams for all commands and functions are in the online help. For convenience we have included the diagram for DEFINE:

```
DEFINE function
   |
   v
(DEFINE)
   |
   v
|macro name|
   |
   v
   +--------------<----------------,
   |                               |
   v                               |
   +-->(PARAMETER)-->|param. name|-->'
   |
   v
   +--------------<----------------,
   |                               |
   v                               |
   +-->(LOCAL)-->|local macro name|-->'
   |
   v
   +-----<-------,
   |             |
   v             |
   +-->|token|-->'
   |
   v
(END_DEFINE)
   |
   v
```

After the DEFINE statement, the computer expects to see:

- the keyword `PARAMETER`,

- the keyword `LOCAL`, or

- a token.

If the computer sees any of these three items, it continues to the next statement. Otherwise the macro stops.

Look under `TOKEN` in the online help, and you see that a command is an acceptable token.

Now look at the syntax diagram for `STORE`:
```
STORE command


->(STORE)-->+------>+-+--->(ALL)---->+->+-------------+->|file name|->
            |       | |               | |             |
            `-(MI)->' `->|partname|->'  `->(DEL_OLD)->'
```
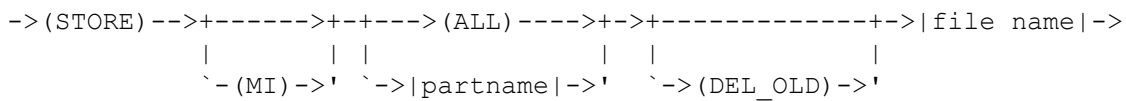
First of all, notice that `STORE` is a command, so `STORE` can be used as a token.

After `STORE`, we must have either the keyword `MI`, or the keyword `ALL`, or the name of a part. We have `ALL`.

After `ALL`, we must have `DEL_OLD` or a filename. We have `DEL_OLD` first, and then a filename in single quotes.

You can verify that `EXIT` must be followed by `CONFIRM`.

Now try the macro without `ALL`:
```
DEFINE Quit STORE DEL_OLD 'workfile' EXIT CONFIRM END_DEFINE
```

You will get the message:
```
Enter option or 'part_name'
```

The computer is confused, because it expects `MI`, `ALL`, or a part name.

So you see that writing your macros line by line helps human readers, but the computer relies on algorithms such as syntax diagrams for its understanding.

# Explaining Local Variables

Local variables are known only to the current macro, or any macro called by the current macro. Variables that are not defined as local are automatically global. Global variables are visible to all macros. In general, you should localize all variables. Global variables can be dangerous and you should avoid using them.

To understand the difference between local and global variables, type the following three macros into a file. The three macros are very similar, so type the first macro and then use block copying for the other two. You can omit any comments.
```
DEFINE Outer_macro

{############################################}
```

```
{##    This macro shows the use of            ##}
{##    global variables                       ##}
{##############################################}

  LET X 5                      {initialize the variable "X"}
  DISPLAY_NO_WAIT ('outer X = '+STR(X))
                               {display a message on the
                                command line }
  WAIT 3                       {allow time to read the message}
  Middle_macro                 {call another macro }
  DISPLAY_NO_WAIT ('outer X = '+STR(X))
  WAIT 3
END_DEFINE

DEFINE Middle_macro
  DISPLAY_NO_WAIT ('middle X = '+STR(X))
  WAIT 3
  Inner_macro
  DISPLAY_NO_WAIT ('middle X = '+STR(X))
  WAIT 3
END_DEFINE

DEFINE Inner_macro
  DISPLAY_NO_WAIT ('inner X = '+STR(X))
  WAIT 3
  LET X 20
  DISPLAY_NO_WAIT ('inner X = '+STR(X))
  WAIT 3
END_DEFINE
```

Note the use of comments enclosed in braces {}. Anything enclosed in braces is ignored by the compiler. Comments help other readers understand your macro. If your macro is very complex, comments may help you understand it—especially six months later!

Your comments should explain the purpose of the command. The user can use the syntax diagrams to understand the command. A comment such as:
```
{allow time to read the message}
```

is more revealing than:
```
{wait approximately 3 seconds}
```

📝 **Note**

Comments cannot be nested. This means that if a section of code contains comments, you cannot "comment out" this code during debugging.

You can see that `outer_macro` calls `middle_macro`, which in turn calls `inner_macro`. This is an example of nested macros.

None of the macros use local variables, so the variable `X` is a global variable. This means that `X` is available to all three macros.

Input the file containing the three macros, then type on the command line:
```
Outer_macro
```

The command line output should be, in sequence:
```
'outer X = 5'
'middle X = 5'
'inner X = 5'
'inner X = 20'
'middle X = 20'
'outer X = 20'
Enter command
```

Before discussing global variables, let's have a quick look at some of the macro statements.

`DISPLAY_NO_WAIT` is used to display a message on the command line. After the message is displayed, no user action is required. This function differs from `DISPLAY`, which requires the user to press a key in order for the macro to continue.

`DISPLAY_NO_WAIT` will first print:
```
outer X =
```

What does `STR(X)` do? `X` is a `NUMBER` with some kind of internal machine representation using bits "set" or "cleared". Only ASCII characters can be displayed using the `DISPLAY_NO_WAIT` statement. `STR(Age)` converts the internal representation of `X` to an equivalent ASCII string, so that it can be displayed.

When used with strings, the + symbol concatenates the strings. For example, let's assume `String1` is `'to'`, `String2` is `'get'`, and `String3` is `'her'`. Then `String1+String2+String3` is `together`. Any blanks inside the single quotes are also displayed.

`WAIT 3` makes the system wait for approximately three seconds after the message is displayed. This gives you time to read the message.

The `Middle_macro` statement calls another macro that must be known to the system at run time. Since our three macros are all in the same file, the system can find `Middle_macro` when required.

Now, back to global variables. You can see that the value of `X` equal to `5`, defined in `Outer_macro`, was known to the middle and inner macros. In `Inner_macro`, we changed the value of `X` to `20`. This was known to the middle and outer macros.

Now add a line to the middle macro, after the `DEFINE` statement, as shown in the following example:
```
DEFINE Middle_macro
  LOCAL X                                    {added}
```

```
  DISPLAY_NO_WAIT ('middle X = '+STR(X))
  WAIT 3
  Inner_macro
  DISPLAY_NO_WAIT ('middle X = '+STR(X))
  WAIT 3
END_DEFINE
```

Input the file, and type `Outer_macro` on the command line. The output should be:
```
'outer X = 5'
***The macro X is not defined
```

The system is indicating that it does not know the value of `X`. This is because `X` is now a local variable in `Middle_macro`. Other variables of the same name in any outer macro are not known to `Middle_macro`.

To make the macro run, add another line to the middle macro as shown:
```
DEFINE Middle_macro
  LOCAL X
  LET X 10                                {added}
  DISPLAY_NO_WAIT ('middle X = '+STR(X))
  WAIT 3
  Inner_macro
  DISPLAY_NO_WAIT ('middle X = '+STR(X))
  WAIT 3
END_DEFINE
```

Input the file, and type `Outer_macro` on the command line.

Now the output is:
```
'outer X = 5'
'middle X = 10'
'inner X = 10'
'inner X = 20'
'middle X = 20'
'outer X = 5'
Enter command
```

Here's what has happened:

- The `LOCAL X` statement in `Middle_macro` makes `Middle_macro` a watertight shell. Values of `X` cannot penetrate this shell. This means that values of `X` cannot be passed inwards from `Outer_macro` to `Middle_macro`, or outwards from `Middle_macro` to `Outer_macro`. But `X` can be passed inwards to any macro called by `Middle_macro`, unless any of these inner macros is also watertight to `X`.

- When `X` is set equal to `10` in `Middle_macro`, this value can then be passed to `Inner_macro`.

- When X is set equal to 20 in inner_macro, this value can be passed outwards to Middle_macro.

- Since X is a local variable in Middle_macro, Outer_macro does not know the value of X from Middle_macro. The value of X is the same value that was originally set in Outer_macro, which is 5.

Finally, let's make X a local variable in all three macros:

```
DEFINE Outer_macro
  LOCAL X                                        {added}
  LET X 5
  DISPLAY_NO_WAIT ('outer X = '+STR(X))
  WAIT 3
  Middle_macro
  DISPLAY_NO_WAIT ('outer X = '+STR(X))
  WAIT 3
END_DEFINE

DEFINE Middle_macro
  LOCAL X
  LET X 10
  DISPLAY_NO_WAIT ('middle X = '+STR(X))
  WAIT 3
  Inner_macro
  DISPLAY_NO_WAIT ('middle X = '+STR(X))
  WAIT 3
END_DEFINE

DEFINE Inner_macro
  LOCAL X                                             {added}
  LET X 15                                            {added}
  DISPLAY_NO_WAIT ('inner X = '+STR(X))
  WAIT 3
  LET X 20
  DISPLAY_NO_WAIT ('inner X = '+STR(X))
  WAIT 3
END_DEFINE
```

Input the file and type Outer_macro on the command line. The output is follows:

```
'outer X = 5'
'middle X = 10'
'inner X = 15'
'inner X = 20'
'middle X = 10'
'outer X = 5'
Enter command
```

Do you understand the output?

# Why Use Local Variables?

One important reason for using local variables is to make our macros as watertight as possible to avoid "side effects". Side effects occur when a variable is accidentally influenced by a variable of the same name in an inner macro.

Your office macros library may contain a useful macro that is exactly what you need to avoid writing a large block of code in your macro. You want to call this macro from your macro without worrying whether the called macro uses variable names that conflict with your variable names. The called macro may call another macro, which in turn calls a third macro, and so on. You don't want to read every line of code in these macros to find conflicting variables. If the writer of these macros has used local variables, you don't have to worry.

As an example, the following macro may not behave in the way the programmer intended:

```
DEFINE Outer_loop
  LOCAL X
  LET X 1              {initialize the loop counter}
  WHILE (X < 3)        {while X is less than 3,
                        execute the code up to the next
                        END_WHILE statement}
    DISPLAY_NO_WAIT ('outer X = '+STR(X))
    WAIT 3
    Inner_loop
    LET X (X+1)        {increment the loop counter}
  END_WHILE
END_DEFINE

DEFINE Inner_loop
  LET X 1
  WHILE (X < 4)
    DISPLAY_NO_WAIT ('inner X = '+STR(X))
    WAIT 3
    LET X (X+1)
  END_WHILE
END_DEFINE
```

Will the macro give the following output? If not, why not?

```
'outer X = 1'
'inner X = 1'
'inner X = 2'
'inner X = 3'
'outer X = 2'
'inner X = 1'
'inner X = 2'
'inner X = 3'
Enter command
```

Now that you are an expert on local variables, you know what the output should be.

Since variables such as X,I, and N are so commonly used as counters in loops, you can see how important it is for the programmer to shield these values from other macros.

# Do We Declare Variable Types?

Those of you with some knowledge of programming know that in most programming languages variables must be declared. For example, in Pascal we have variables such as integer, real, and char. In C we have int, float, char, and so on.

Declaring a variable means that a memory location of the correct size is reserved for that variable.

In our macros, we do not formally declare variables.

The nearest we come to declaring a variable is when we use a READ statement. The read statement produces a command line prompt for the user to provide some input. Here are some examples:

```
DEFINE Read_test
  READ STRING 'Enter file name' File
                        {The string 'Enter file name'
                         appears on the command line. You must
                         enter a string is single quotes. The
                         string is assigned to the variable,
                         "File"}
  READ NUMBER 'Enter your age' Age
  READ PNT 'Enter a point' P1
  READ PNT 'Digitize a point' RUBBER_LINE P1 P2
  DISPLAY_NO_WAIT ('File = '+File)
  WAIT 3
  DISPLAY_NO_WAIT ('Age = '+STR(Age))
                        {"Age" is converted to an ASCII
                         string, then printed after 'Age = ' }
  WAIT 3
  DISPLAY_NO_WAIT ('P1 = '+STR(P1))
  WAIT 3
  DISPLAY_NO_WAIT ('P2 = '+STR(P2))
  WAIT 3
END_DEFINE
```

To save typing, we have not used local variables.

Look at the first READ statement:

```
READ STRING 'Enter file name' File
```

When you run the macro (try it!), the prompt appears on the command line:

```
Enter file name
```

You must now enter a string in single quotes. If you forget the single quotes, the system again displays the same prompt. This is the advantage of stating the type of variable immediately after the READ statement: if you provide the wrong type of input, the system re-executes that macro line until the input matches the type specified.

For the second READ statement, if you try to enter a number in single quotes, or enter a point instead of a number, the prompt is again displayed. So you are alerted if you try to enter values that do not satisfy the type NUMBER.

It is possible to use READ statements without specifying the type, such as STRING or NUMBER. Then the system accepts any incorrect data while the READ statement is being executed. The system will not complain until the point in the macro where it uses the data. In our macro, this would be during the DISPLAY_NO_WAIT statement. Try removing one of the type statements after a READ statement. Run the macro, and then supply some deliberately incorrect data.

Did you notice that we used STR(Age) and STR(Point), but not STR(File)? Remember that STR(Age) converts the internal representation of Age to its ASCII equivalent, so that it can be displayed. STR(Point) does the same thing. File is already an ASCII string, exactly as entered by the user, so it does not need conversion.

The following example shows how to feedback an existing/calculated value to the screen. The DEFAULT option evaluates it's parameter and puts the result into the user input line. Pressing [Return] will then enter the (possibly edited) value to satisfy the READ request.

```
DEFINE Test
LOCAL Value
LET Value (100/2)
READ NUMBER 'Current value is:' DEFAULT Value Value
DISPLAY (STR Value)
END_DEFINE
```

# Using Control Statements

We do not always want macro statements to be executed in the sequence that they appear in the macro. We may want some statements to be executed several times. We may want some statements to be executed only if a specific condition is true. Control statements are used to alter the way a macro executes.

## WHILE ... END_WHILE

We used a WHILE statement in an earlier section, "Why Use Local Variables?" Part of the macro is repeated here:

```
DEFINE Outer_loop
  LOCAL X
  LET X 1                {initialize the loop counter}
```

```
  WHILE (X < 3)          {while X is less than 3,
                          execute the code up to the next
                          END_WHILE statement}
    DISPLAY_NO_WAIT ('outer X = '+STR(X))
    WAIT 3
    Inner_loop
    LET X (X+1)          {increment the loop counter}
  END_WHILE
END_DEFINE
```

As long as `X` is less than 3, the code between `WHILE` and `END_WHILE` is
executed. Before the `WHILE` statement, `X` is set to `1`, and `X` is incremented by one
each time the loop is executed. So this loop should execute twice (not three
times!). In fact, because of side effects caused by global variables in the inner
macro, this loop only executed once.

You can see that a `WHILE ... END_WHILE` construct is used to control the
number of times a loop executes. An important feature of this construct is that the
loop need not be executed even once. If the condition after the `WHILE` statement
is false at entry to the loop, the loop will never be executed. Later, we will look at
the `REPEAT ... UNTIL` construct, which produces a loop that must be executed
at least once.

## LOOP ... EXIT_IF ... END_LOOP

We saw that the `WHILE ... END_WHILE` construct makes a conditional test at
the beginning of the loop. In the `LOOP ... EXIT_IF ... END_LOOP`
construct, the conditional test can be done at any point in the loop.

In the following code fragment, the user is prompted for a value. If the user does
not supply the correct value, the prompt is displayed again until the correct value
is entered.
```
LOOP
  READ NUMBER 'Enter a fractional value for the split' Fract
EXIT_IF ((Fract>0)) AND (Fract< 1))
END_LOOP
```

Any number of `EXIT_IF` statements can be used in the body of the loop, as
shown in the following outline:
```
LOOP
  ...
  ...
EXIT_IF
  ...
  ...
EXIT_IF
  ...
  ...
END_LOOP
```

If no `EXIT_IF` statement is used, the macro only stops when the user enters `END` or begins a command. Here is an example:

```
DEFINE Circ_rad

{Create several circles with the same }
{radius, or several circles passing through the same peripheral}
{point                                                         }

{ local variables here }

READ 'Indicate peripheral point or enter radius or END' P2
LOOP
  READ 'Indicate center of circle' P1
  CIRCLE CENTER P1 P2
  END
END_LOOP
END_DEFINE
```

What happens if we put the first `READ` statement inside the `LOOP ... END_LOOP` construct? Here is the altered macro:

```
DEFINE Circ_rad
LOOP
  READ 'Indicate peripheral point or enter radius or END' P2
  READ 'Indicate center of circle' P1
  CIRCLE CENTER P1 P2
  END
END_LOOP
END_DEFINE
```

The `LOOP ... EXIT_IF ... END_LOOP` construct is very useful because the loop can be terminated at any point, not just the beginning or end. Also, as we have seen, the user can terminate the macro by entering `END`.

Note that if the user enters `END`, the complete macro is terminated, not just the loop. Do not expect the macro to jump out of the loop and continue executing statements after the `END_LOOP`.

## REPEAT ... UNTIL

This construct is similar to the `LOOP ... EXIT_IF ... END_LOOP` construct with the `EXIT_IF` statement placed just before the `END_LOOP`. Since the conditional test following the `UNTIL` is not made until the end of the loop, this loop will always be executed at least once.

## IF ... ELSE_IF ... ELSE ... END_IF

The `IF` family of statements are used to make decisions. Depending on the decision, some parts of the macro will be executed and other parts will not.

In the following code fragment, users are being asked whether they want a construction line to be horizontal, vertical, or perpendicular to an existing line:

```
READ "ENTER 'H' FOR HORIZ, OR 'V' FOR VERT, OR 'P' FOR PERP" Q
IF (Q='H')
  C_LINE HORIZONTAL P2       {if Q='H', only this statement}
                                        {is executed}
ELSE_IF (Q='V')
    C_LINE VERTICAL P2    {if Q<>'H', but Q='V', only these}
    LET X 3                    {two statements are executed}
ELSE
    C_LINE PERPENDICULAR P1 P2      {if Q<>'H', and Q<>'V',}
                       {but Q='P', only this statement is}
                                          {executed}

END_IF
```

The conditions are evaluated from the top downwards, in the form of a ladder. As soon as a true condition is found, the statements associated with the condition are executed, and the rest of the ladder is ignored.

You can use as many ELSE_IF statements as you need, or none at all. You can have only one ELSE statement, and it must come at the end. The final ELSE is the default condition. In other words, if all other conditional tests fail, then the last ELSE statement is executed. If the final ELSE is not present, and if the other conditions are false, then no action is taken. Sometimes the final ELSE is used in defensive programming to handle errors.

The absolute minimum IF statement has no ELSE_IF or ELSE. For example:

```
IF (X = 4)
  LET P2 P5
END_IF
```

IF statements can be nested, but each IF must have its own END_IF. We can expand the previous example:

```
IF (X < 6)
  IF (X = 4)
    LET P2 P5
  END_IF
  LINE TWO_PTS P1 P2 END
END_IF
```

# Using Parentheses

One problem with any programming language is when to use parentheses. The following guidelines may give some help.

## Boolean Expressions

Always use parentheses with boolean expressions.

A boolean expression, sometimes called a conditional expression, is any expression that has the value true or false.

Boolean expressions are used in the test part of the four major constructs:

```
IF (boolean expression) ... END_IF
LOOP...EXIT_IF (boolean expression) ... END_LOOP
REPEAT ... UNTIL (boolean expression)
WHILE (boolean expression) ... END_WHILE
```

## Arithmetic, Algebraic, and Trigonometric Expressions

Always use parentheses when an expression is used as a token.

Examples of expressions are as follows:

- X + Y
- X - 5
- X - 5.147
- X * 312
- X/312
- X DIV Y
- Point1 + 4.29,3.42
- SIN 30
- SIN (Angle3 + 30)
- SQRT 16.238
- SQRT (X + 16.238)
- LEN 'Have a nice day!'

See the online help for a complete list of possible tokens. If the syntax diagram requires a token, and any of the above expressions are used, then the expression must be enclosed in parentheses. For example:

```
LET Radius1 (X + Y)
LET Radius1 ((X + Y) * COS 60 )
LET Point2 Point1
LET Point2 (Point1 + 4.29,3.42)
DISPLAY_NO_WAIT Point2
DISPLAY_NO_WAIT (Point1 + 4.29,3.42)
```

The above examples show some interesting points. Lets look at these examples one by one.

Look at the syntax diagram for LET in the syntax chapter. You see that LET is followed by the name of a variable, and then by a token. In the first example, X + Y is an expression used as a token, so it must be enclosed in parentheses.

Now look at the second example. Here the expression is

```
(X + Y) * COS 60
```

In this expression, we need the parentheses to force the correct order of evaluation. If this expression, including parentheses, is used as a token then a further set of parentheses is necessary.

The third example shows that a simple token does not require parentheses:

```
LET Point2 Point1
```

If this simple token is modified so that it becomes an expression, then parentheses are necessary:

```
LET Point2 (Point1 + 4.29,3.42)
```

Note that the following statement is perfectly acceptable:

```
LET Point2 (Point1)
```

Look at the syntax diagram for DISPLAY_NO_WAIT in the syntax chapter and you will see that DISPLAY_NO_WAIT must be followed by a token. The examples for DISPLAY_NO_WAIT are similar to the last two examples for LET.

You can see from the previous examples that a LET statement sometimes requires the token to have parentheses, and sometimes not. A good rule in the early days of your macro writing might be: Always use parentheses with a LET statement.

# Using the Trace Facility

**T** **Trace** is a useful debugging tool. We want to introduce you to **T** **Trace** at this point because you can see how boolean expressions and other expressions are evaluated.

Here is the macro we are going to trace:

```
DEFINE Parenth
  LET P1 (0,0)                    {parentheses not necessary}
  LET P2 (10,0)                   {parentheses not necessary}
  LET X (1)                       {parentheses not necessary}
  WHILE ( X < 5 )                 {parentheses necessary}
    LET P1 (P1 + 0,10)            {parentheses necessary}
    LET P2 (P2 + 0,10)            {parentheses necessary}
    LINE TWO_PTS P1 P2 END
    LET X ( X + 1)                {parentheses necessary}
  END_WHILE
END_DEFINE
```

The macro will draw four lines. When X equals 1, a line is drawn from 0,0 to 10,0. When X equals 2, a line is drawn from 0,10 to 10,10, and so on.

We have noted where parentheses are necessary and where they are not.

This macro demonstrates an important principle. Look at the syntax diagram for LINE TWO_PTS in the *Creo Elements/Direct Drafting Programing Reference Guide*. In a macro, the recursive loop shown can be ended only by an END statement. This is a general rule: when a syntax diagram ends with a recursive loop, use an END statement to get out of the loop.

Look at the diagram for LINETYPE. This diagram does not end with a recursive loop. No END statement is required in a macro.

Now let's use  **Trace** on our macro, and store the results in a file called trace.

Before using  **Trace**, change the second line in parenth from this:
```
LET P1 (0,0)
```
to this:
```
LET P1 0,0
```
Type the following in the user input line and press ENTER after each command:
```
input 'parenth'

trace

parenth
```

When the macro ends, disable  **Trace**:

1. Click **Miscellaneous** and then, in the **System** group, click  **Trace**.

2. Click  **Off**.

    in the status bar indicates that **Trace** is inactive.

---

📝 **Note**

If your trace file is unexpectedly empty or incomplete, you may have forgotten to disable  **Trace**.

---

If you want to append the results of a  **Trace** operation to an existing file:

1. Click **Miscellaneous** and then, in the **System** group, click  **Trace**.

2. Click  **On**.

3. Click  **Append**.

    in the status bar indicates that  **Trace** is active.

If you want a clean file for each run:

1. Click **Miscellaneous** and then, in the **System** group, click **Trace**.

2. Click **On**.

3. Click **New**.

    in the status bar indicates that **Trace** is active.

To view the results of the **Trace** operation:

1. Click **Miscellaneous** and then, in the **System** group, click **Trace**.

2. Click **Edit**.

   The results of the **Trace** operation are displayed in the standard text editor.

Here is the `trace` file for `parenth`:

```
Parenth
LET P1 0,0
LET P2 ( 10,0 ) 10,0
LET X ( 1 ) 1
WHILE ( X 1 < 5 ) 1
LET P1 ( P1 0,0 + 0,10 ) 0,10
LET P2 ( P2 10,0 + 0,10 ) 10,10
LINE TWO_PTS P1 0,10 P2 10,10
END
LET X ( X 1 + 1 ) 2
END_WHILE ( X 2 < 5 ) 1
LET P1 ( P1 0,10 + 0,10 ) 0,20
LET P2 ( P2 10,10 + 0,10 ) 10,20
LINE TWO_PTS P1 0,20 P2 10,20
END
LET X ( X 2 + 1 ) 3
END_WHILE ( X 3 < 5 ) 1
LET P1 ( P1 0,20 + 0,10 ) 0,30
LET P2 ( P2 10,20 + 0,10 ) 10,30
LINE TWO_PTS P1 0,30 P2 10,30
END
LET X ( X 3 + 1 ) 4
END_WHILE ( X 4 < 5 ) 1
LET P1 ( P1 0,30 + 0,10 ) 0,40
LET P2 ( P2 10,30 + 0,10 ) 10,40
LINE TWO_PTS P1 0,40 P2 10,40
END
LET X ( X 4 + 1 ) 5
END_WHILE ( X 5 < 5 ) 0
TRACE
```

The first line of `trace`

```
LET P1 0,0
```
is the same as in our macro.

The second line is different. Here, the "expression" within the parentheses has been evaluated. The result `10,0` is shown immediately after the expression.

Now look at the boolean expression:
```
WHILE ( X 1 < 5 ) 1
```

The current value of `X` is `1`. This is shown immediately after `X`. The boolean expression `(1 < 5)` is true, so the value `1` appears immediately after the expression.

Note that `WHILE` never appears again in `trace`, but the value of the boolean expression appears after `END_WHILE`. Look at the last boolean expression after the last `END_WHILE`. Here, the current value of `X` is `5`. The boolean expression is `(5 < 5)`, which is false, so the value of the expression is `0`.

The following macro will work:
```
DEFINE Parenth_2
  LET P1 (0,0)                  {parentheses not necessary}
  LET P2 (10,0)                 {parentheses not necessary}
  WHILE (1)                     {parentheses necessary}
    LET P1 (P1 + 0,10)          {parentheses necessary}
    LET P2 (P2 + 0,10)          {parentheses necessary}
    LINE TWO_PTS P1 P2 END
  END_WHILE
END_DEFINE
```

Now the expression after the `WHILE` is always true. This macro will continue drawing lines until there is a power failure or until you press `[Break]`.

This macro was included deliberately to show that a boolean expression always needs parentheses, even if a simple token is being used. If you don't believe it, try changing `WHILE (1)` to `WHILE 1`.

Change `WHILE (1)` to `WHILE (TRUE)` and it still works.

The following macro shows that a boolean expression is true if it has any non-zero value:
```
DEFINE Parenth
  LET P1 0,0
  LET P2 (10,0)
  LET X (5)
  WHILE (X)
    LET P1 (P1 + 0,10)
    LET P2 (P2 + 0,10)
    LINE TWO_PTS (P1) (P2) END
    LET X ( X - 1)
  END_WHILE
  DISPLAY_NO_WAIT 'something has changed'
  WAIT 3
```

```
  LET X ( X - 1 )
  WHILE (X)
    LET P1 (P1 + 0,10)
    LET P2 (P2 + 0,10)
    LINE (TWO_PTS) P1 P2 END
    LET X ( X + 1 )
  END_WHILE
END_DEFINE
```

# Indenting the Lines of a Macro

You have probably noticed that some lines of a macro are indented. This indentation is ignored by the processor, but helps a reader to understand the flow of logic.

Conventionally, the lines following a boolean expression are indented. In the previous example, the lines following each `WHILE` statement are indented. The closing statement of the `WHILE` construct is `END_WHILE`, and this statement appears in the same column as the `WHILE`. The result is that the keywords of the construct act visually like markers. A reader can immediately see where each section of the code begins and ends. This makes debugging much easier.

Similarly, indentation is used for the following constructs:

*   `IF ... ELSE_IF ... ELSE ... END_IF`
*   `LOOP ... EXIT_IF ... END_LOOP`
*   `REPEAT ... UNTIL`

Each keyword of a construct starts in the same column. The statements following a keyword are indented.

Nesting of any of these constructs requires a further level of indentation.

# Defensive Programming

You can avoid many common bugs by programming defensively and trying to allow for all eventualities. Here are some guidelines:

*   Use consistent indentation. We discussed this in the previous section.
*   Use descriptive variable names. `Radius1` or `Rad1` is easier to understand than `S`. You might accidentally write `TAN(S)`, but you would immediately see the error in `TAN(Radius1)`.
*   Write simple code. The following two code fragments produce identical results. Which is easier to understand?

    Fragment 1:
    ```
    LET P3 (P1 - (PNT_RA O.5*(D2 - D1) ANG(P2 -P1)))
    ```

Fragment 2:
```
LET Shoulder (0.5*(D2 - D1)
LET Angle (ANG(P2 - P1)
LET Vect_S (PNT_RA Shoulder Angle)
LET P3 (P1 - Vect_S)
```

If there is a bug in any of the previous code fragments, which would you rather debug?

- Don't worry about efficient code while you are writing the macro. If your final macro is too slow, then you can concentrate on areas where greater speed may be possible.

- Avoid writing long macros. Prefer to use several shorter macros. Each macro can be debugged separately.

- Try to screen out bad user input. We looked at the following code fragment when discussing IF statements:

```
READ "ENTER 'H' FOR HORIZ, OR 'V' FOR VERT, OR 'P' FOR PERP"
IF (Q='H')
    C_LINE HORIZONTAL P2        {if Q='H', only this statement}
                                              {is executed}
ELSE_IF (Q='V')
      C_LINE VERTICAL P2    {if Q<>'H', but Q='V', only these}
      LET X 3                    {two statements are executed}
ELSE
      C_LINE PERPENDICULAR P1 P2     {if Q<>'H', and Q<>'V',}
                           {but Q='P', only this statement is}
                                                   {executed}
END_IF
```

This example is fine for demonstrating IF statements, but not very good for demonstrating defensive programming. What happens if the user accidentally enters, for example, 'Y'? Or enters 'v' instead of 'V'?

Here is a rewritten version, designed to trap accidental user errors:

```
LOOP
READ STRING "ENTER 'H' FOR HORIZ, OR 'V' FOR
                              VERT, OR 'P' FOR PERP" Q
EXIT_IF ((Q='H') OR (Q='h') OR (Q='V') OR
                              (Q='v') OR (Q='P') OR (Q='p'))
END_LOOP

IF ((Q='H') OR (Q='h'))
  C_LINE HORIZONTAL P2
ELSE_IF ((Q='V') OR (Q='v'))
  C_LINE VERTICAL P2
ELSE
  C_LINE PERPENDICULAR P1 P2
```

```
    END_IF
```
- If your macro fails when the user enters numbers that are too high, too low, or negative, try to trap these numbers with statements such as:
```
LOOP
   READ NUMBER 'Enter number of sides' Num_sides
EXIT_IF ((Num_sides > 0) AND (Num_sides < 20))
END_LOOP
```

# Macro Commands

All the commands listed in the online help can be used in the macro body. Some commands and statements are very useful in macros. Here are some examples:

| | |
|---|---|
| `DISPLAY P1` | Displays the X, Y coordinates of point `P1` and waits for a user response. |
| `BEEP` | Emits a short fixed-frequency/amplitude tone on the system loudspeaker. |
| `IF (M>N) ... ELSE ... END_IF` | Executes all lines before `ELSE` if M>N. Otherwise executes all lines between `ELSE` and `END_IF`. Any number of `ELSE` statements can be used. |
| `LET D (L1 + 5)` | Defines `D` as equal to the sum of `L1` and `5`. |
| `LOOP ... EXIT_IF (N > 50) ... EXIT_IF (M < 7) END_LOOP` | Executes all lines before `END_LOOP` repeatedly in sequence, until N > 50 or M < 7. Any number of `EXIT` statements can be used. If no exit statement is used, then the loop will repeat endlessly until `END` or another command is selected. |
| `READ PNT P8` | Stops system until point `P8` is entered by the user. |
| `REPEAT ... UNTIL (N>10)` | Executes all lines repeatedly in sequence until `N` is greater than `10`. |
| `TONE 256 2 0.5` | Emits a `256` Hz tone of `2` seconds duration and of relative amplitude `0.5` on the system loudspeaker. |
| `WHILE (N< 20) ... END_WHILE` | Executes all lines after `WHILE` repeatedly in sequence while `N` is less than `20`. |

Expressions are enclosed in parentheses. An expression usually contains arithmetic, trigonometric, or relational operations as in the previous examples. There are also some special operations that we can use. These are explained in the next section.

# Built-in Operations

Here are examples of some useful built-in operations that can be used within expressions:

| | |
|---|---|
| `ANG P6` | Calculates the angle between the vector `P6` and the X axis. |
| `LEN P4` | Calculates the length of the vector `P4`. |
| `PNT_XY 20 65` | Defines a vector with origin at X0, Y0 and head at X20, Y65. |
| `PNT_RA 16 38` | Defines a vector with origin at X0, Y0, of length `16` and at an angle of `38` to the X axis. |
| `ROT P2 45` | Defines a vector of the same length as the vector `P2`, at an angle of `45` degrees to `P2`, in a counterclockwise direction. |
| `SQR 15` | Calculates the square of `15`. |
| `SQRT 15` | Calculates the square root of `15`. |
| `X_OF P1` | Calculates the X coordinate of the point `P1`. |
| `Y_OF P6` | Calculates the Y coordinate of the point `P6`. |

# 4

# Inquiring about the Environment and Elements

When writing macros, it is often necessary to know about the current environment. What are the current units? Inches or millimeters? Is the current line color white or some other color? Is the scale 1:1 or 2.5:1?

As an example, you may want to use a macro to draw a yellow line using dot-center lines. It's easy to change the color and linetype. But after your macro has finished, you should change the color and linetype back to their original values. The user of your macro should not have to reset these values.

Your macro must store the current color and linetype, and then retrieve these values before the macro ends. This chapter shows you how to use the `INQ_ENV` and `INQ_ELEM` functions to store information in the system array, and how to use `INQ` to interrogate the system array. `INQ` is an abbreviation for inquire.

# Using `INQ_ENV`

The following macro demonstrates the use of `INQ_ENV` and `INQ`:

```
DEFINE Env_check
  LINE HORIZONTAL 0,0 300
  INQ_ENV 3                    {load information in system array}
  LET Col (INQ 201)                     {save color information}
  LET Ltype (INQ 301)             {save linetype information}
  INQ_ENV 2             {load window position in system array}
  LET Lower_left (INQ 101)        {save lower left point of}
                                           {current window}
  LET Upper_right (INQ 102)            {upper right point}
  YELLOW                                    {new color}
  DOT_CENTER                             {new linetype}
  LINE HORIZONTAL 0,50 300
  COLOR Col                        {reset to original color}
  LINETYPE Ltype                {reset to original linetype}
  LINE HORIZONTAL 0,100 300 END
  DISPLAY_NO_WAIT ("Look at the OLD window")
  WAIT 5
  WINDOW -10,-10 350,150             {create a new window}
  DISPLAY_NO_WAIT ("Now look at the NEW window")
  WAIT 5
  WINDOW Lower_left Upper_right
  DISPLAY_NO_WAIT ("Back to the OLD window")
  WAIT 5
END_DEFINE
```

The second line of the macro draws a line using the current color and linetype. The remaining lines of the macro are discussed in the following paragraphs.

```
INQ_ENV 3
```

If you study the `INQ_ENV` function in the online help, you see that `INQ_ENV` inserts data into a section of the system array. The section of the array is specified by the number that follows `INQ_ENV`. For example, section 0 contains information of the software version number and version string. Section 1 contains information on the current viewport, and so on.

We are interested in section 3, which contains information on catch range, geometry, construction data, text data, and so on.

Each time you use `INQ_ENV` you overwrite the previous data in that section of the system array.

```
LET Col (INQ 201)
```

`INQ` interrogates the section of the system array that was last written to using `INQ_ENV`. In our example, the last `INQ_ENV` wrote to section 3, so `INQ` interrogates section 3. The specific information depends on the number following `INQ`.

Look again at the `INQ_ENV` function in online help, and you see that `INQ 201` returns the current geometry color. This value is assigned to the variable `Col`.

```
LET Ltype (INQ 301)
```

The current geometry linetype is assigned to the variable `Ltype`.

```
INQ_ENV 2
```

Information about the current window is inserted in section 2 of the system array.

```
LET Lower_left (INQ 101)
LET Upper_right (INQ 102)
```

The lower left point and the upper right point of the current window are assigned to variables.

The next three lines of the macro change the color and linetype, and then draw a line to show the effects of the changes.

```
COLOR Col
LINETYPE Ltype
```

The original values for color and linetype are restored. The macro draws another line to show the effects of the changes.

The last few lines of the macro change the coordinates of the current window, and displays this window for approximately five seconds to allow you to see the change. The original coordinates are then restored.

# Using `INQ_ELEM`

`INQ_ELEM` is similar to `INQ_ENV`, and also places data into the system array. The data depends on the element that exists on your drawing at the specified point. For example, if the element is a circle, data about the circle is placed in the array.

If you pick an element, you can find out what type of element it is by using `INQ 403`. In your macro, you may want the user to pick a circle. You can use `INQ 403` to verify that it's actually a circle rather than, for example, a rectangle.

The following code fragment shows an example:

```
LOOP
  READ PNT 'Digitize a circle' P
  INQ_ELEM P
  EXIT_IF (INQ 403 = CIRCLE)
END_LOOP
```

In this fragment, `INQ_ELEM` reads information about the digitized point into the system array. Refer to the `INQ_ELEM` function in the online help, and you see that `INQ 403` returns the element type. If the type is a circle, the macro exits from the loop and continues with the next section of the macro.

If you want to know the radius or center point of the circle, you can then use:

```
LET Rad (INQ 3)
LET Cen_pt (INQ 101)
```

Note that if no element is found at the digitized point, `INQ 403` returns `END`. You may want to build in a check on the catch mode before prompting again.

## Using `GETENV`

`GETENV` retrieves the settings for the current environment. For example, to retrieve the setting for the variable `MEDIR`, enter the following on the command line:

```
display (getenv('MEDIR'))
```

Creo Elements/Direct Drafting returns the current setting.

## Using Other Inquiries

The use of other inquiries such as `HL_INQ_Z_VALUE` and `HL_INQ_FACE_COLOR` is similar to the use of `INQ_ENV` and `INQ_ELEM`. In each case, values are written to the system inquiry array, and retrieved using `INQ`.

For an example of a macro using `HL_INQ_Z_VALUE`, see Useful Macros on page 103.

# 5

# Quick Review of Points and Vectors

Most people reading a manual such as this will be familiar with points and vectors. However, it may be a few years since you last used them. This very brief chapter is intended to refresh your memory before you read Writing Geometry Macros on page 71.

# Points

A point is defined by an X-coordinate and a Y-coordinate. Look at the points in the next figure:

**Figure 1. Representation of Points**
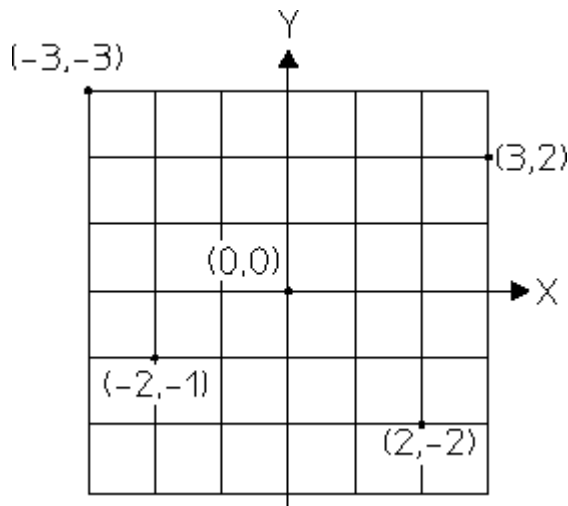


The X-axis and the Y-axis intersect at the point (0,0). This point is referred to as the origin. At the origin, the X-coordinate is 0 and the Y-coordinate is 0.

You can see that the other points are identified by writing the X-coordinate followed by the Y-coordinate.

In a macro, if you have a point and you need to know the X-coordinate or Y-coordinate of the point, use the X_OF or Y_OF operators. For example:
```
LET X1 (X_OF P1)
LET Y1 (Y_OF P1)
```

Let's assume that P1 is (3,2). Then X_OF P1 is 3 and Y_OF P1 is 2.

If you have two numbers, you can make a point from these numbers using the PNT_XY operator. For example, with the numbers 3 and 2:
```
LET P1 (PNT_XY 3 2)
```

This will produce the point (3,2).

In the general case:
```
LET P1 (PNT_XY X Y)
```

produces the point (X,Y).

# Vectors

A vector has length and direction. In the next figure, the vector A represents a line from the origin (0,0) to the point (3,2). The vector B represents a line drawn from the origin (0,0) to the point (1,4).

**Figure 2. Vectors Originating at the Origin**
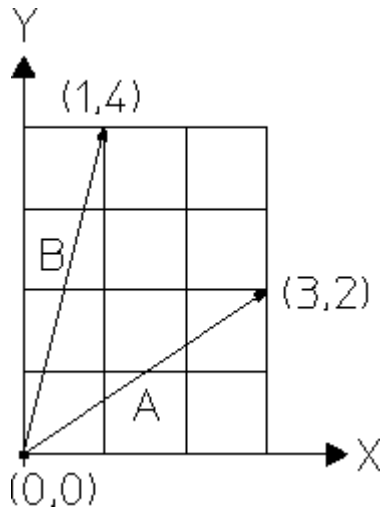


If vectors originate at the origin, as in the previous figure, then we can refer to vector A as (3,2) and vector B as (1,4). In the geometry macros described in the following chapters, all vectors originate at the origin.

## Addition of Vectors

Vector A can be added to vector B to give vector C. In the next figure, you can see that C represents a line drawn from the origin to the point (4,6). The X-coordinate represented by C is obtained by adding the X-coordinate s of (1,4) and (3,2), which gives 4. Similarly, the Y-coordinate is found by adding the Y-coordinates of these two points, which gives 6.

**Figure 3. Addition of Vectors**
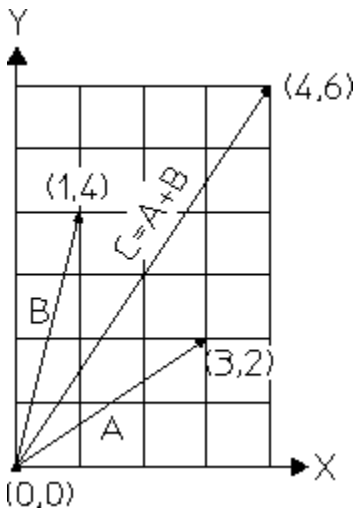


Here is another way to visualize the addition of vector A to vector B. First, move vector B to a new position B' attached to the end of vector A, as shown in the next figure. Remember that a vector is defined by a length and a direction, so the new vector B' is identical to the old vector B.

**Figure 4. Visualization of Vector Addition**
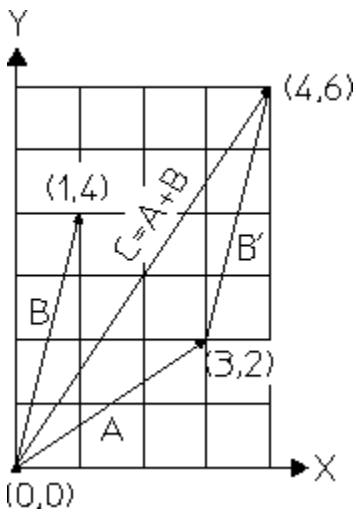


Now start at the origin and move 3 units in the X-direction and 2 units in the Y-direction Then move 1 unit in the X-direction and 4 units in Y-direction. You arrive at the same point if you start at the origin and move 4 units in the X-direction and 6 units in the Y-direction. This proves that C = A + B.

In addition of vectors, A + B is the same as B + A.

# Subtraction of Vectors

To subtract vectors, you subtract their X and Y-coordinates. The order of subtraction is important. The next figure shows the result of A − B and B − A.

**Figure 5. Subtraction of Vectors**
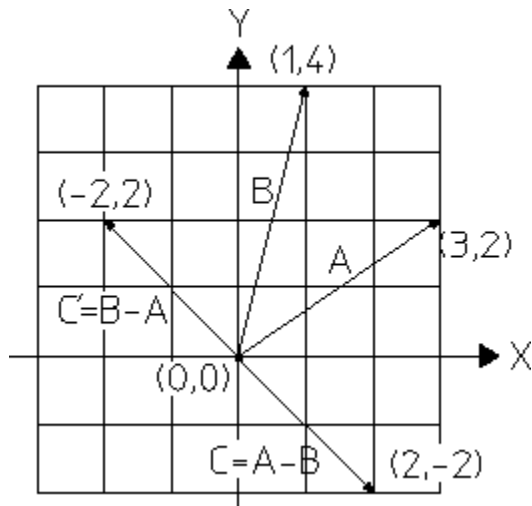


You can see that, as in normal arithmetic, the vector A − B is not the same as the vector B − A. The vectors are the same length, but in opposite directions.

As for addition, you can visualize the subtraction of vectors by moving one of the vectors and attaching it to the end of the other vector. The next figure shows the effect of A − B:

**Figure 6. Visualization of Vector Subtraction**
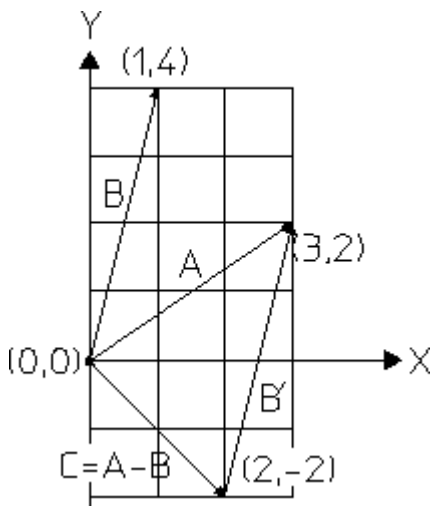
# 6

# Writing Geometry Macros

This chapter shows you how to write geometry macros. You will probably use geometry macros more often than any other type of macro. If you find that you are frequently drawing the same type of shape, using either fixed dimensions or varying dimensions, you should write a macro to do the job for you.

The chapter gives detailed instructions on how to write an arrowhead macro and a panel macro.

# The Arrowhead Macro

The macro will draw the arrowhead shown in the next figure:
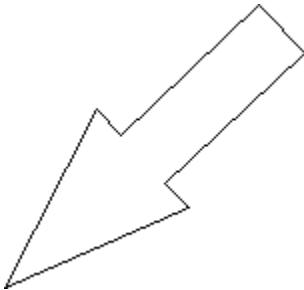


As well as drawing the basic shape, the macro must do the following:

- The arrowhead must lie between any two points picked from the screen. The first point will be the tip of the arrow. The second point will be at the center of the tail.

- The arrowhead proportions must be constant.

- The macro must repeat operation until canceled by another command.

Several ways of writing a macro will meet these requirements. There are no rules about which method to use. Choose the one that suits you best.

In the method we are going to use, points are treated as vectors. This is a good method for those who are familiar with simple vector handling.

## Writing the Arrowhead Macro

Here is the step-by-step sequence:

1. Draw the arrowhead at any angle to the axes and annotate all the points. The points to be picked from the screen are `P1` and `P2`. Draw in any point for the origin (0,0).
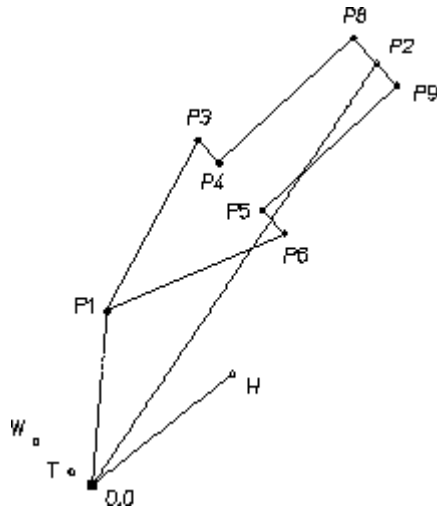
2. Work on the macro body first. For a geometry macro this is the part that does the drawing. Assume that the points `P1` and `P2` are defined and that they each represent the head of a vector drawn from the origin as shown in the previous figure.

3. The barbs of the arrowhead are a certain proportion (let's use 60%) of the total length along the centerline (from the arrowpoint). Define a vector with this length between `P1` and `P2` in the direction of `P2` and call it `H`. Like all other vectors, `H` originates at `0,0`.
   `LET H (0.6 * ( P2 - P1 ) )`

4. The barbs are a certain proportion (let's use 30%) of `H` away from the centerline. Define a vector of this length normal to `H` and call it `W`.
   `LET W ( 0.3 * ( ROT H 90 ) )`

5. The half-thickness of the tail is a certain proportion (let's use 40%) of `W`. Define a vector of this length in the direction of `W` and call it `T`.
   `LET T ( 0.4 * W )`

6. Now the points can be defined. Add `H` and `W` to `P1` to define the point `P3` as in the next figure.
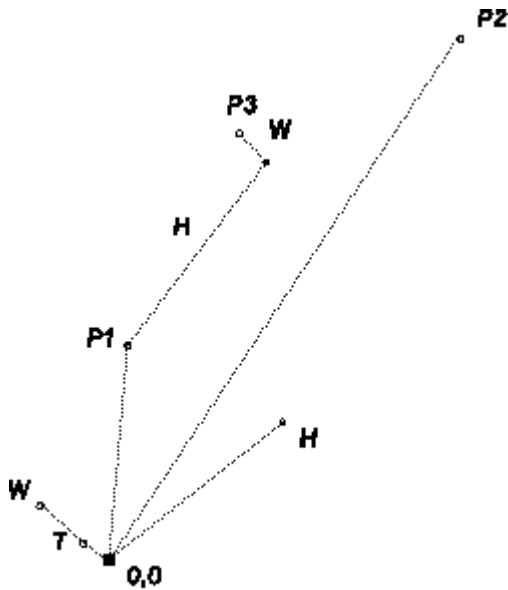
```
LET P3 ( P1 + H + W )
```

7. Define the other points in exactly the same way.
   ```
   LET P6 ( P1 + H - W )
   LET P4 ( P1 + H + T )
   LET P5 ( P1 + H - T )
   LET P8 ( P2 + T )
   LET P9 ( P2 - T )
   ```

8. Join the points to create the arrowhead shape.
   ```
   LINE POLYGON P1 P3 P4 P8 P9 P5 P6 P1
   ```

The macro body is finished, and now we can move on to handling the points `P1` and `P2`, which are picked from the screen. The `READ` command allows the user to enter the coordinates of `P1` and `P2`:

```
READ PNT 'Pick the arrow point' P1
```

`READ PNT` tells the system that a point is going to be input. The system displays the message Pick the arrow point. When the user enters a point, it is defined as `P1` and the next line of the macro is executed.

```
READ PNT 'Pick the tail centerpoint' RUBBER_LINE P1 P2
```

In this line, the RUBBER_LINE command draws a line between `P1` and the cursor as it is moved across the screen. This is useful for sizing the arrowhead. When the next point is entered it is defined as `P2` and the next line is executed. This next line is the start of the macro body.

Now enclose the `READ` statements and the macro body in a loop, using `LOOP` and `END_LOOP`. This allows the user to create several arrowheads during the same operation. Specify `LINETYPE` and `COLOR` outside the loop, so that they can be altered during operation if required. Define all the macro variables as `LOCAL`.

*Creo Elements/Direct Drafting Writing Macros*

Here is the completed macro:

```
DEFINE Arrow_head
  LOCAL P1
  LOCAL P2
  LOCAL P3
  LOCAL P4
  LOCAL P5
  LOCAL P6
  LOCAL P8
  LOCAL P9
  LOCAL H
  LOCAL W
  LOCAL T
    COLOR WHITE
    LINETYPE SOLID
  LOOP
    READ PNT 'Pick the arrow point' P1
    READ PNT 'Pick the tail centerpoint' RUBBER_LINE P1 P2
    LET H (0.6 * ( P2 - P1 ) )
    LET W ( 0.3 * ( ROT H 90 ) )
    LET T ( 0.4 * W )
    LET P3 ( P1 + H + W )
    LET P6 ( P1 + H - W )
    LET P4 ( P1 + H + T )
    LET P5 ( P1 + H - T )
    LET P8 ( P2 + T )
    LET P9 ( P2 - T )
    LINE POLYGON P1 P3 P4 P8 P9 P5 P6 P1
    END
  END_LOOP
  END
END_DEFINE
```

Remember that there are many ways of writing a macro to do the job. An alternative method is to turn the axes to the required angle before establishing the points and drawing the shape. This method would use commands such as `CS_SET`, `CS_ROTATE`, `CS_REF_PT`, `CS_AXIS`.

# The Panel Macro

The aim is to write a macro to draw a front panel like the one shown here. The panel is designed to fit a standard 19-inch rack.
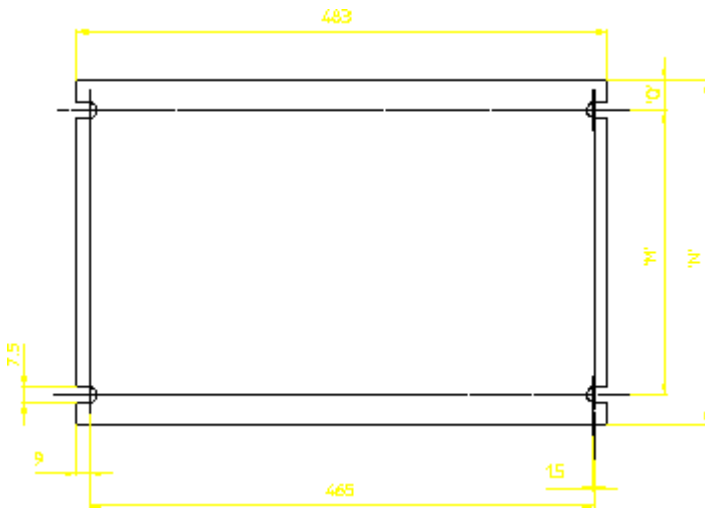
The macro must do the following:

- Position the panel by picking one of the bolt holes in the left hand upright of a rack.
- Determine the height of the panel by giving the required U value (the width is constant).
- Keep the orientation of the panel the same as that of the ruler axis.

The geometry of the front panel is mainly fixed, so it's possible to define the construction points using coordinates instead of vectors.

## Writing the Panel Macro

Draw out the front panel at a fixed angle (0 degrees is the easiest) and annotate all the coordinates involved.
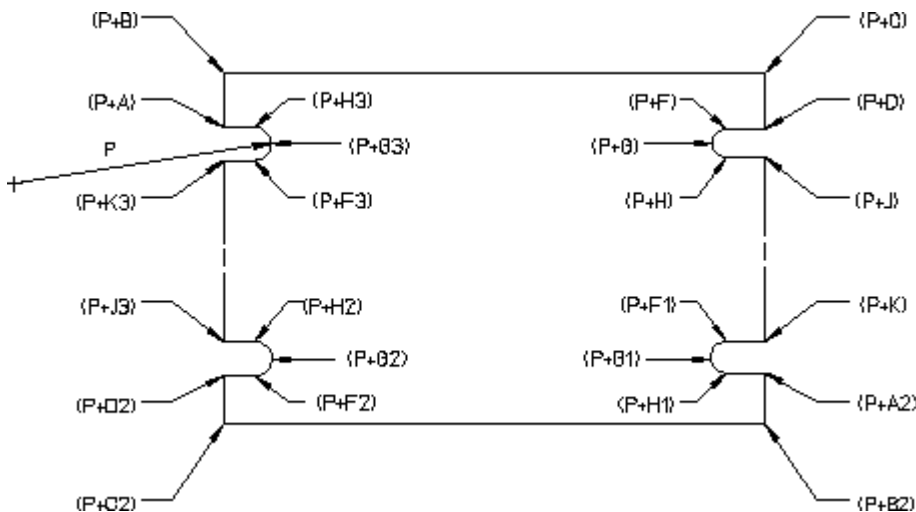
As with the arrowhead, start work on the macro body first. Assume that the vector P has been defined, and that the specifications for the front panel are as shown in the diagram. M and Q are variable parameters.

Define all points by their coordinates from the end point of the vector P. P is considered to have the coordinates (0,0). These coordinates can then be added in turn to the initial vector P to create the panel. The diagram below shows the coordinates of some of the panel features:
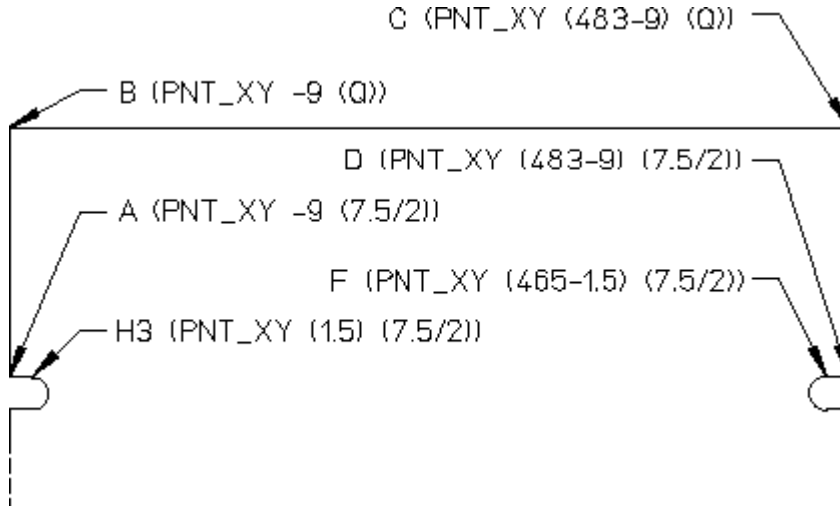


Continuing this process around the panel gives all the coordinates of the features and they can be defined as follows:

```
LET A  (PNT_XY -9 (7.5/2))
LET B  (PNT_XY -9 (Q))
LET C  (PNT_XY (483-9) (Q))
LET D  (PNT_XY (483-9)(7.5/2))
LET F  (PNT_XY (465-1.5)(7.5/2))
LET G  (PNT_XY (465-1.5-(7.5/2)) 0)
LET H  (PNT_XY (465-1.5)(-7.5/2))
LET J  (PNT_XY (483-9)(-7.5/2))
LET K  (PNT_XY (483-9)(-M+(7.5/2)))
LET F1 (PNT_XY (465-1.5)(-M+(7.5/2)))
LET G1 (PNT_XY (465-1.5-(7.5/2)) (-M))
LET H1 (PNT_XY (465-1.5)(-M-(7.5/2)))
LET A2 (PNT_XY (483-9)(-m-(7.5/2)))
LET B2 (PNT_XY (483-9)(-M-Q))
LET C2 (PNT_XY -9(-M-Q))
LET D2 (PNT_XY -9(-M-(7.5/2)))
LET F2 (PNT_XY 1.5(-M-(7.5/2)))
LET G2 (PNT_XY (1.5+(7.5/2))(-M))
LET H2 (PNT_XY 1.5(-M+(7.5/2)))
LET J2 (PNT_XY -9(-M+(7.5/2)))
LET K2 (PNT_XY -9(-7.5/2))
LET F3 (PNT_XY 1.5(-7.5/2))
LET G3 (PNT_XY (1.5+(7.5/2)) 1.5)
```

```
LET H3 (PNT_XY (1.5)(7.5/2))
```

To complete the macro body, connect all the points to create the front panel. This is done using the `LINE` and `ARC` commands, and can be done in any order. The order used here is from point `(P+H3)` to point `(P+G3)`, giving:
```
LINE POLYGON (P+H3) (P+A) (P+B) (P+C) (P+D) (P+F)
ARC THREE_PTS (P+F) (P+H) (P+G)
LINE POLYGON (P+H) (P+J) (P+K) (P+F1)
ARC_THREE_PTS (P+F1) (P+H1) (P+G1)
LINE POLYGON (P+H1) (P+A2) (P+B2) (P+C2) (P+D2) (P+F2)
ARC THREE_PTS (P+F2) (P+H2) (P+G2)
LINE POLYGON (P+H2) (P+J2) (P+K2) (P+F3)
ARC THREE_PTS (P+F3) (P+H3) (P+G3)
```

The macro body is now complete. To make the macro operational we must localize the vector point P, and allow for the input of data. `READ` statements are used for data input. The `READ` statements are implemented as shown below:
```
READ NUMBER  "Enter the required height of the front panel" U
READ "Identify the position of the top left bolt hole" P
```

Now localize the vector point P, as shown:
```
READ NUMBER  "Enter the required height of the front panel" U
READ "Identify the position of the top left bolt hole" P
```

The macro is now operational, but it is still very basic. It does not specify the linetype, line color, when the macro ends, any constraints on the input value, or how to convert this value into the actual height of the panel. These refinements determine how user-friendly the macro will be.

You can specify the linetype and color within the macro. For example, the following can be used:
```
COLOR YELLOW
LINETYPE SOLID
```

It is also simple to specify where and when the macro should complete operation. This is done by inserting `END` before `END_DEFINE` in the macro. If an `END` statement is not used, the system shows the prompt of the command that was being performed before the macro was called.

You can specify constraints for the `U` value in more than one way. In this example the `LOOP...EXIT_IF...END_LOOP` construction in conjunction with `IF...END_IF` was chosen to check the input of `U`. Three constraints are imposed upon the `U` value for this macro:

• It must be an integer value.

• It must be greater than 0.

• It must be less than 17.

The loops are used in the following manner:
```
IF ((FRACT U <> 0) OR (U>16) OR (U< 1))
  LOOP
    READ NUMBER "Please re-enter an integer value between 1 and 16"
```

```
    EXIT_IF ((FRACT U=0) AND (U>0) AND (U< 17))
  END_LOOP
END_IF
```

This loop will be executed until the value of `U` meets the conditions above.

When the value has been checked, it must then be converted into a form that is more usable to the macro. At the moment the value is integer and represents the unit height of the panel, therefore it must be converted into the actual height. This is done by the final `IF...END_IF` loop in the macro. Before the loops are defined, the `U` value is converted by `LET N (((U-1)*44.45)+43.6)` to create an `N` value that indicates the actual height of the panel. This value is then used to define the variables `M` and `Q`. The variable `M` defines the vertical distance between the slots and `Q` defines the vertical distance between each slot and a horizontal edge, where `M` is variable over the complete range of `U`, and `Q` is fixed at separate values for:

- U less than 3 but greater than 0.

- U less than 17 but greater than 2.

Here is the completed macro:
```
DEFINE Tline
LOCAL P

READ NUMBER 'Enter the required height of the front panel' U

  COLOR YELLOW
  LINETYPE SOLID

  IF ((FRACT U <> 0) OR (U>16) OR (U< 1))
    LOOP
     READ NUMBER "Please re-enter an integer value between 1 and 16" U
     EXIT_IF ((FRACT U=0) AND (U>0) AND (U< 17))
    END_LOOP
  END_IF

LET N (((U-1)*44.45)+43.6)
      IF (U< 3)
          LET M (N-11.9)
          LET Q 5.95
          LET M (N-75.4)
      ELSE
          LET Q 37.7
      END_IF

      READ "Identify the position of the top left bolt hole" P

      LET A (PNT_XY -9 (7.5/2))
      LET B (PNT_XY -9 (Q))
      LET C (PNT_XY (483-9) (Q))
      LET D (PNT_XY (483-9) (7.5/2))
```

```
LET F (PNT_XY (465-1.5) (7.5/2))
LET G (PNT_XY (465-1.5-(7.5/2)) 0)
LET H (PNT_XY (465-1.5) (-7.5/2))
LET J (PNT_XY (483-9) (-7.5/2))
LET K (PNT_XY (483-9) (-M+(7.5/2)))
LET F1 (PNT_XY (465-1.5) (-M+(7.5/2)))
LET G1 (PNT_XY (465-1.5-(7.5/2)) (-M))
LET H1 (PNT_XY (465-1.5) (-M-(7.5/2)))
LET A2 (PNT_XY (483-9) (-M-(7.5/2)))
LET B2 (PNT_XY (483-9) (-M-Q))
LET C2 (PNT_XY -9 (-M-Q))
LET D2 (PNT_XY -9 (-M-(7.5/2)))
LET F2 (PNT_XY 1.5 (-M-(7.5/2)))
LET G2 (PNT_XY (1.5+(7.5/2)) (-M))
LET H2 (PNT_XY 1.5 (-M+(7.5/2)))
LET J2 (PNT_XY -9 (-M+(7.5/2)))
LET K2 (PNT_XY -9 (-7.5/2))
LET F3 (PNT_XY 1.5 (-7.5/2))
LET G3 (PNT_XY (1.5+(7.5/2)) 1.5)
LET H3 (PNT_XY 1.5 (7.5/2))

LINE POLYGON (P+H3) (P+A) (P+B) (P+C) (P+D) (P+F)
ARC THREE_PTS (P+F) (P+H) (P+G)
LINE POLYGON (P+H) (P+J) (P+K) (P+F1)
ARC THREE_PTS (P+F1) (P+H1) (P+G1)
LINE POLYGON (P+H1) (P+A2) (P+B2) (P+C2) (P+D2) (P+F2)
ARC THREE_PTS (P+F2) (P+H2) (P+G2)
LINE POLYGON (P+H2) (P+J2) (P+K2) (P+F3)
ARC THREE_PTS (P+F3) (P+H3) (P+G3)
END
END_DEFINE
```

# 7

# File Input/Output and Text Strings

This chapter shows you how to extract data from a file for use in a macro, and how to write data to a file. In our example the data consists of text strings, but the same principles can be applied to numeric data.

# What the Macro Will Do

The aim is to write a macro that extracts the main keyword from each section of the `help` file and writes the results to a new file.

To see what we mean by a "keyword", look at the following section from the `help` file:

```
========================================================

^AUTO_NEW_SCREEN
AUTO_NEW_SCREEN function

---->(AUTO_NEW_SCREEN)---->+------>(ON)----->+---->
                           |                 |
                           `------>(OFF)---->'

The AUTO_NEW_SCREEN function allows you to select ...
.
.
========================================================
```

In this example, the section describes the `AUTO_NEW_SCREEN` function. The keyword is `AUTO_NEW_SCREEN`, and that's the string we want to write to the output file.

Note that you can search for the keyword in one of two ways:

- Look for a ^character, then take the string after the ^. Or,

- Look for the string `command` or `function`. Then take the preceding string.

From a programming point of view, the first method is better. The processor only has to examine a file in a "forwards" direction. Using the second method, the processor has to look "forwards" until it finds `command` or `function`, and then look "backwards" to find the preceding string. Let's use the first method.

Now, look at the next example:

```
========================================================

^CANCEL ESC STOP INTERRUPT BREAK
^ABORT
^Meview_cancel

CANCEL command

---->(CANCEL)---->

CANCEL  cancels  the current  activity  ...
.
.
========================================================
```

In this example, `^CANCEL`, `^ABORT`, and `^Meview_cancel` are all preceded by `^`, but we only want to include `CANCEL` in our output file. We don't want to include `ABORT` or `Meview_cancel` because these strings refer to other sections of the `help` file. So, our macro must tell the processor to extract the first string that is preceded by `^` in each section, and ignore the others.

Before discussing our text-handling macro, there is one other point to make. One of the things that our macro must do is search for blanks in lines. In the previous example, `^CANCEL` is separated from the next string, `ESC`, by a blank. We sometimes use blanks to let us know that we are at the end of strings. Now look again at this section from the `help` file:

```
========================================================

^AUTO_NEW_SCREEN
AUTO_NEW_SCREEN function

---->(AUTO_NEW_SCREEN)---->+------>(ON)----->+---->
                           |                 |
                           `------>(OFF)---->'

The AUTO_NEW_SCREEN function allows you to select ...
.
.
========================================================
```

The first line of the section is `^AUTO_NEW_SCREEN`. It is important to realize that when this line is printed on the screen, the line is padded with blanks. But in the file there are no blanks in this line. Immediately after the final `N` of `AUTO_NEW_SCREEN` there is a linefeed character, which is not printed on the screen but appears in the file. The linefeed character has a hexadecimal ASCII value of 0A. If "\n" represents the linefeed character ("\n" is used in the C programming language) then the previous example looks like:

```
^AUTO_NEW_SCREEN\nAUTO_NEW_SCREEN function\n\n ... (and so on)
```

Note that "\n" is two characters, but the actual linefeed character is only a single character.

# Analyzing the Macro

The following macro will extract the keywords from the `help` file. The macro is not necessarily the most efficient, but it is easy to understand. We are now going to analyze this macro line by line.

```
DEFINE Keywords_search
{###############################################}
{##  This macro searches for all the keywords  ##}
{##  in the help file, and lists them in    ##}
{##  an output file.                         ##}
{###############################################}
```

```
        LOCAL File1
        LOCAL File2
        LOCAL Flag
        LOCAL Filestring1
        LOCAL First_char
        LOCAL Line_pos
        LOCAL String_length
        LOCAL First_string
        LET File1 '\me10\help'
        LET File2 '\john\keywords.out'
        OPEN_INFILE 1 File1
        OPEN_OUTFILE 2 DEL_OLD File2
        LET Flag 0                    {initialize the value of Flag}


        LOOP
          READ_FILE 1 Filestring1     {read the next line of file 1}
        EXIT_IF (Filestring1='END-OF-FILE')
          LET First_char (SUBSTR Filestring1 1 1)
          IF (First_char='^')
            IF (Flag = 0)
              LET Line_pos (POS Filestring1 ' ')
                                       {find the position in the line}
                                       {of the first blank character}
              IF (Line_pos = 0)        {no blank characters}
                LET String_length (LEN Filestring1)
                                       {find the length of the complete}
                                       {line}
                LET First_string (SUBSTR Filestring1 2 (String_length-1))
                WRITE_FILE 2 First_string
                LET Flag 1
              ELSE
                LET First_string (SUBSTR Filestring1 2 (Line_pos-2))
                WRITE_FILE 2 First_string
                LET Flag 1
              END_IF
            END_IF
          ELSE
            LET Flag 0
          END_IF
        END_LOOP
        CLOSE_FILE 1
        CLOSE_FILE 2
END_DEFINE
```

Look at the first line of the macro:
```
DEFINE Keywords_search
```

Every macro must start with `DEFINE` followed by the name of the macro. Here,
the macro name is `Keywords_search`.
```
LOCAL File1
```

```
LOCAL File2
LOCAL Flag
LOCAL Filestring1
LOCAL First_char
LOCAL Line_pos
LOCAL String_length
LOCAL First_string
```

These are the local variables. If `Keywords_search` is called by another macro, there will be no danger of conflict between the names of the variables used by the two macros.
```
LET File1 '\me10\help'
LET File2 '\john\keywords.out'
```

In this macro, the variable `File1` is used to represent the full path name for the `help` file. Similarly, `File2` is the file where the list of keywords will be written.
```
OPEN_INFILE 1 File1
```

A file cannot be used in a macro until it is opened. The function `OPEN_INFILE` opens `File1` for reading. The file pointer points to the first record in the file. The numeral 1 is a file descriptor. After `File1` is opened, all further references to `File1` are made using the file descriptor 1. The use of the file descriptor will become clearer later. The file descriptor is similar to the MS-DOS "handle".

Note that the following statement is equally valid, and would have the same result as our previous statement:

## Platform Dependencies
```
OPEN_INFILE 1 '\me10\help'
```

So you might think the variable `File1` is not really necessary. The advantage of using a variable such as `File1` is that we can quickly modify our macro to read a file other than `C:\Program Files\PTC\Creo Elements\Direct Drafting [version]\help`. All we must do is alter the line:
```
LET File1 '\me10\help'
```

If we do not use a variable such as `File1`, then we must search through our macro and replace every occurrence of \me10\help with the full path name of the new file. Our macro is very short, so this would be easy. In a longer macro, this takes time.
```
OPEN_OUTFILE 2 DEL_OLD File2
```

Open `File2` for writing. The file descriptor is 2. If the file already exists, it will be overwritten because of the `DEL_OLD` instruction.
```
LET Flag 0
```

`Flag` is a variable that will have a value of either 1 or 0, depending on which of two branches is taken on each pass through the macro. A variable that shows a state or condition is often called a flag, which is why we have named it `Flag`. The use of `Flag` will become clearer later. This line initializes the value of `Flag` to 0.
```
LOOP
```

Start a loop. This loop will continue until a later `EXIT_IF` statement is satisfied, or `END_LOOP` is executed.
```
READ_FILE 1 Filestring1
```

Read the file whose file descriptor is `1`. This means read `File1`, which is `/me10/help`. If a file has been newly opened, but not read, the file pointer points to the first record in the file. The `READ_FILE` function treats each complete line as a record, so the `READ_FILE` command reads the first line of the file and then advances the file pointer so that it points to the next record. The next time the same file is read, the `READ_FILE` command reads the second line, and so on. If you are searching a file several times, and you want the macro to start the search at the beginning of the file each time, you must use the `OPEN_INFILE` statement before each `READ_FILE` statement so that the file pointer points to the first line of the file.
```
EXIT_IF (Filestring1='END-OF-FILE')
```

Every file has an end-of-file marker. The actual marker depends on the operating system, but is often the null character. If the macro detects this end of file marker, it will exit from the current loop.
```
LET First_char (SUBSTR Filestring1 1 1)
```

Find the substring that starts at position `1` in the line, and has length `1`. In other words, find the first character in the line.

As an example of substrings, let's assume `Filestring1` was:
```
Have a nice day!
```

If our program statement was:
```
LET Substring1 (SUBSTR Filestring1 3 6)
```

then `Substring1` would be `ve a n`.
```
IF (First_char='^')
```

If the first character is `^`, execute the following statements until the matching `ELSE_IF`, `ELSE`, or `END_IF`. (The matching `ELSE` occurs 14 program lines later.)

If the first character is not `^`, jump to the first statement after the matching `ELSE`.
```
IF (Flag = 0)
```

If the value of `Flag` is `0`, execute the following statements until the matching `ELSE_IF`, `ELSE`, or `END_IF`. There is no `ELSE_IF` or `ELSE`. The matching `END_IF` occurs 12 program lines later.

(We have not yet reached the point where we can explain the purpose of `Flag`.)
```
LET line_position (POS Filestring1 ' ')
```

As mentioned earlier, if a line starts with `^` and has only one keyword in the line, the line will contain no blanks. If there is more than one keyword, the keywords will be separated by blanks. (We use `' '` to indicate one blank. Two blanks would be `' '` and so on.) The purpose of the program statement is to find out if the line

contains blanks. The `POS` function gives the position in a string (first argument) of the first occurrence of a substring (second argument). As an example, the current line from the `help` file could be:

```
^CURRENT_DIRECTORY CD CURRENT DIRECTORY TM_FILE_2 SM_FILE_2
```

Here, the first argument is the complete line beginning with `^CURRENT_DIRECTORY` and ending with `SM_FILE_2`. The second argument is `' '`. So, `Line_pos` will be `19`.

The current line could be:

```
^AUTO_NEW_SCREEN
```

Here, the value of `Line_pos` is `0`. Each line ends with a linefeed character so there can be no blanks in this line.

```
LET String_length (LEN Filestring1)
```

This statement computes the length of the complete line. If the line is:

```
^AUTO_NEW_SCREEN
```

then the value of `String_length` is `16`.

When we look at the next program line, we will see why we need the length of the string.

```
LET First_string (SUBSTR Filestring1 2 (String_length-1))
```

We have seen `SUBSTR` before. This statement will extract the substring starting at position `2` in the string, and having length `String_length-1`. If the string is:

```
^AUTO_NEW_SCREEN
```

then `First_string` is `AUTO_NEW_SCREEN`. In other words, the `^` has been stripped from the beginning of the line. `AUTO_NEW_SCREEN` can now be written to a file.

```
WRITE_FILE 2 First_string
```

Append `First_string` to the file whose file descriptor is `2`. This file is `\john\keywords.out`. `First_string` will be treated as a complete line in this file.

```
LET Flag 1
```

We see that if a line is written to the output file, we set `Flag` to `1`. If you examine the rest of the code carefully, you will see that if a line is read from `File1`, but not written to `File2`, then `Flag` is set to `0`.

Why do we want to do this? Remember that we want to extract, from each section, the first string that is preceded by `^`. For example, in the next section from the `help` file, we want to extract `CANCEL`, but not `ABORT` or `Meview_cancel`:

```
=======================================================

^CANCEL ESC STOP INTERRUPT BREAK
^ABORT
^Meview_cancel

CANCEL command

.
```

```
.
.
========================================================
```

For clarity, here is a stripped-down version of our macro, containing only read and write statements, and statements referring to `Flag`:

```
LET Flag 0
LOOP
  READ_FILE 1 Filestring1
  ...
  IF (First_char='^')
    IF (Flag = 0)
      IF ...
        ...
        WRITE_FILE 2 First_string
        LET Flag 1
      ELSE
        ...
        WRITE_FILE 2 First_string
        LET Flag 1
      END_IF
    END_IF
  ELSE
    LET Flag 0
  END_IF
END_LOOP
```

After the `READ_FILE` statement, there are four possible conditions:

1. `First_char` equals ^, `Flag` equals 0.

   A keyword is written to file. `Flag` set to 1.

2. `First_char` equals ^, `Flag` equals 1.

   Nothing written to file. `Flag` set to 0.

3. `First_char` not equal to ^, `Flag` equals 0.

   Nothing written to file. `Flag` set to 0.

4. `First_char` not equal to ^, `Flag` equals 1.

   Nothing written to file. `Flag` set to 0.

So we see that if `First_char` is not ^, nothing gets written to file. This is fine. It means a string that is not a keyword cannot be written to file.

The only time anything gets written to file is when `First_char` is ^ and `Flag` is 0. Now, here is the important point: the only way `Flag` can be 0 is if the previous line did not begin with ^. If the previous line began with ^, then `Flag` would be 1. This means that if two or more consecutive lines begin with ^, only the keyword on the first of these lines will be written to file.

*Creo Elements/Direct Drafting Writing Macros*

If `Flag` is 1, the only way it can be reset to 0 is to read a line that does not begin with ^.

Have you noticed that we only need one statement `Let Flag 1`? It would be placed after the first `END_IF` statement in the macro. By using two statements, and placing each statement after a `WRITE_FILE` statement, we are emphasizing that `Flag` is only set to 1 if something is written to the file.

```
ELSE
```

If there are blanks in the line, the code after the `ELSE` is executed.

```
LET First_string (SUBSTR Filestring1 2 (Line_pos-2))
```

In this branch, the first keyword in the line will have ^ in front of it, and a blank after it. The keyword therefore starts at position 2, and its length will be `Line_pos-2`. Let's assume the line is:

```
^CURRENT_DIRECTORY CD CURRENT DIRECTORY TM_FILE_2 SM_FILE_2
```

The keyword we want to extract is `CURRENT_DIRECTORY`. The position of the first blank in the line is 19. The length of `CURRENT_DIRECTORY` is 19-2 or 17.

```
WRITE_FILE 2 First_string
LET Flag 1
```

`First_string` is written to the output file, and `Flag` is set to 1.

```
END_IF
```

The closing `END_IF` for `IF (Line_pos = 0)`.

```
END_IF
```

The closing `END_IF` for `IF (Flag = 0)`.

```
ELSE
```

If `First_char` is not ^, the code after the `ELSE` is executed.

```
LET Flag 0
```

`Flag` is set to 0 if a line is read from the help file but not printed to the output file.

We're now getting to the end of our macro. The last few lines are:

```
END_IF
```

The closing `END_IF` for `IF (First_char = '^')`.

```
END_LOOP
```

The closing statement for the loop.

```
CLOSE_FILE 1
CLOSE_FILE 2
```

The `CLOSE_FILE` statements correspond to the `OPEN_INFILE` and `OPEN_OUTFILE` statements at the start of the macro.

As a matter of good programming practice, close all files that are opened by your macro. If you don't close a file, it remains open until the next user does an `OPEN_INFILE` on this file. The operating system then closes the file before opening it for the next user. If no one uses the file, it remains open. However, each operating

system has a limit on the number of files that can be open at one time. When this limit is reached, the system is unable to open more files for other users. So it's important to close unused files.

# Calling a Macro from within a Macro

It is always possible to call a macro from within another macro. Nesting macros offers two advantages:

- The inner macros can be existing, fully debugged macros. Using existing macros saves writing time and debugging time.

- Long macros can sometimes be difficult to understand and debug. You can often split a long macro into shorter macros nested within a main macro. If you use meaningful names for the inner macros you can often produce a "self-documenting" macro—one that needs very few comment statements. As an example, it should be clear what the following macro does:

```
DEFINE Create_shell
  Calculate_internal_stress
  Calculate_flange_thickness
  Calculate_bolt_thickness
  Draw_semi_shell
  Draw_flanges
  Draw_bolts
  Draw_washers
  Draw_nuts
END_DEFINE
```

Let's replace some lines in `Keywords_search` by an inner macro. In this example, we probably do not gain very much by using an inner macro. It demonstrates a principle, and we can use the example to show you how to pass parameters to macros.

Create an inner macro using the following lines from our first macro:

```
IF (Line_pos = 0)          {no blank characters}
  LET String_length (LEN Filestring1)
                          {find the length of the complete}
                          {line}
  LET First_string (SUBSTR Filestring1 2 (String_length-1))
  WRITE_FILE 2 First_string
  LET Flag 1
ELSE
  LET First_string (SUBSTR Filestring1 2 (Line_pos-2))
  WRITE_FILE 2 First_string
  LET Flag 1
END_IF
```

To create a new macro called `Write_to_file`, all you have to do to is put `DEFINE Write_to_file` at the beginning and `END_DEFINE` at the end.

In the main macro, we must replace the ten lines with a single line containing the new macro name:

```
Write_to_file
```

Our file will now contain the following:

```
DEFINE Keywords_search
{##############################################}
{##  This macro searches for all the keywords  ##}
{##  in the help file, and lists them in       ##}
{##  an output file.                           ##}
{##############################################}

  LOCAL File1
  LOCAL File2
  LOCAL Flag
  LOCAL Filestring1
  LOCAL First_char
  LOCAL Line_pos
  LOCAL String_length
  LOCAL First_string
  LET File1 '\me10\help'
  LET File2 '\john\keywords.out'
  OPEN_INFILE 1 File1
  OPEN_OUTFILE 2 DEL_OLD File2
  LET Flag 0                       {initialize the value of Flag}

  LOOP
    READ_FILE 1 Filestring1      {read the next line of file 1}
  EXIT_IF (Filestring1='END-OF-FILE')
    LET First_char (SUBSTR Filestring1 1 1)
    IF (First_char='^')
      IF (Flag = 0)
        LET Line_pos (POS Filestring1 ' ')
                                  {find the position in the line}
                                  {of the first blank character}
        Write_to_file
      END_IF
    ELSE
      LET Flag 0
    END_IF
  END_LOOP
  CLOSE_FILE 1
  CLOSE_FILE 2
END_DEFINE

DEFINE Write_to_file
  IF (Line_pos = 0)     {no blank characters}
    LET String_length (LEN Filestring1)
                        {find the length of the complete}
                        {line}
    LET First_string (SUBSTR Filestring1 2 (String_length-1))
```

```
    WRITE_FILE 2 First_string
    LET Flag 1
  ELSE
    LET First_string (SUBSTR Filestring1 2 (Line_pos-2))
    WRITE_FILE 2 First_string
    LET Flag 1
  END_IF
END_DEFINE
```

The new file illustrates the point that variables not declared as LOCAL are automatically global. Global variables are known to both macros. For example, the value of Line_pos is evaluated by the outer macro and used by the inner macro. The value for Flag is evaluated by the inner macro and used by the outer macro.

# Passing Parameters to a Macro

Earlier in this chapter, we discussed the problem of global variables, and how these can sometimes cause harmful side effects. If macros are written as watertight compartments, then the only way to pass information to a macro is by using parameters. As an example, let's rewrite our inner macro, Write_to_file, using parameters.

The variables String_length and First_string are no longer used by Keywords_search, so they have been deleted.

```
DEFINE Keywords_search
  LOCAL File1
  LOCAL File2
  LOCAL Flag
  LOCAL Filestring1
  LOCAL First_char
  LOCAL Line_pos
{ LOCAL String_length              deleted }
{ LOCAL First_string               deleted }
  LET File1 '\me10\help'
  LET File2 '\john\keywords.out'
  OPEN_INFILE 1 File1
  OPEN_OUTFILE 2 DEL_OLD File2
  LET Flag 0

  LOOP
    READ_FILE 1 Filestring1
  EXIT_IF (Filestring1='END-OF-FILE')
    LET First_char (SUBSTR Filestring1 1 1)
    IF (First_char='^')
      IF (Flag = 0)
        LET Line_pos (POS Filestring1 ' ')
        Write_to_file Line_pos Filestring1 2
      END_IF
    ELSE
```

```
      LET Flag 0
    END_IF
  END_LOOP
  CLOSE_FILE 1
  CLOSE_FILE 2
END_DEFINE

DEFINE Write_to_file
  PARAMETER Column
  PARAMETER String
  PARAMETER File_descriptor
  LOCAL String_length
  LOCAL First_string
  IF (Column = 0)
    LET String_length (LEN String)
    LET First_string (SUBSTR String 2 (String_length-1))
    WRITE_FILE File_descriptor First_string
    LET Flag 1
  ELSE
    LET First_string (SUBSTR String 2 (Column-2))
    WRITE_FILE File_descriptor First_string
    LET Flag 1
  END_IF
END_DEFINE
```

Here are some things you should note about the new macros:

- Except for `Flag`, the variables in `Write_to_file` are different from those in `Keywords_search`. This is the most common situation if you call a macro from within a macro. The inner macro is likely to be part of a library of macros, and you have used it to avoid writing new code yourself. The specification for the macro will tell you what the macro does and what parameters you have to pass. You have no interest in the internal workings of the macro, or what variables are used.

- In `Write_to_file`, the parameters can be declared in any order. But when `Write_to_file` is called by `Keywords_search`, the arguments must be in the correct order. The statement that calls `Keywords_search` is:
  ```
  Write_to_file Line_pos Filestring1 2
  ```

  In this statement, notice the following:

  ○ `Line_pos` corresponds to `Column`.

  ○ `Filestring1` corresponds to `String`.

  ○ `2` corresponds to `File_descriptor`.

When you call the macro `Write_to_file`, the system knows that the first parameter after the macro name represents the first declared parameter in the called macro; the second parameter after the macro name represents the second declared parameter in the called macro, and so on.

Note that you can pass parameters in to a called macro, but you cannot pass parameters out to the calling macro. As in all programming languages, macros are expanded at compile time. This means that the macro code is substituted for the macro name. When one macro calls another macro, the code is substituted inline in the calling macro at compile time. The resulting code is then executed as if there were only one macro.

If `Write_to_file` were substituted inline without `PARAMETER` statements, the compiler would complain that some variables were unknown. The `PARAMETER` statements tell the compiler that some pairs of variables are equivalent. However, `PARAMETER` variables behave similarly to `LOCAL` variables: they are only visible in the macro in which they are declared. So a macro is different from functions and subroutines as used in other programming languages such as PL/I or Fortran. In these languages, the program branches to the called function or subroutine, which can then return values to the calling program.

Since you need to know the value of `Flag` after execution of `Write_to_file`, `Flag` has not been declared as a local variable or as a parameter in `Write_to_file`. We must keep `Flag` as a global variable so that it is visible to both macros.

The output file, `keywords.out`, will have the keywords listed in the order that they appear in the `help` file.

## Platform Dependencies

If you want these words to be sorted, the easiest way is to use the MS-DOS `sort` command. In a Windows environment, use the Program Manager to open a command prompt, then type:
```
sort < keywords.out > keywords.srt
```

`keywords.srt` is the name of the sorted file. If `keywords.out` is not in your current directory, you must use the full path name of the file.

# 8

# Using Dimensions Stored in a Data File

This chapter describes a macro that extracts dimensions from a file of tabulated data. The macro then draws the part at a specified location on the screen.

# What the Macro Will Do

• The user is prompted to digitize two points on the circumference of a hole face. In our simple macro, these points are the right point and the left point as you look into the hole.

• The macro calculates the diameter of the hole by using the two points.

• The macro searches a file until the calculated hole diameter matches a value in the first column. The remainder of the data in the row is then used to calculate the size of a spigot that is a friction fit in the hole.

• The macro draws the spigot in the correct position and attitude.

• The user is again prompted to digitize two points, or END.

# Describing the Spigot

The next figure shows the general dimensions of the spigot:

**Figure 13. Spigot—General Dimensions**
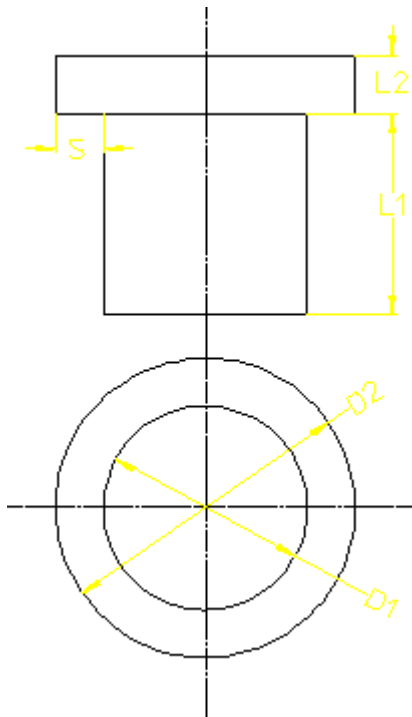


The dimension S, the width of the spigot shoulder, can be computed from 1/2(D2-D1).

You can see that the geometry of the spigot is very simple. However, the macro we discuss in this chapter can be adapted to any complex geometric figure.

To test your macro, you may want to draw a block similar to the following sectional view, which shows holes of several different diameters at several different attitudes:

**Figure 14. Sectional View of Test Block**
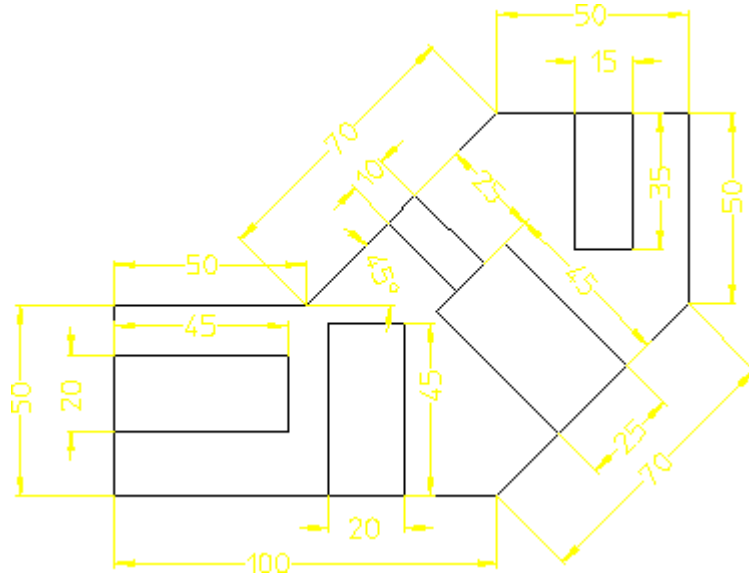


After you have used the macro to insert the correct spigot in each hole, you should see the following:

**Figure 15. Spigots in Place**
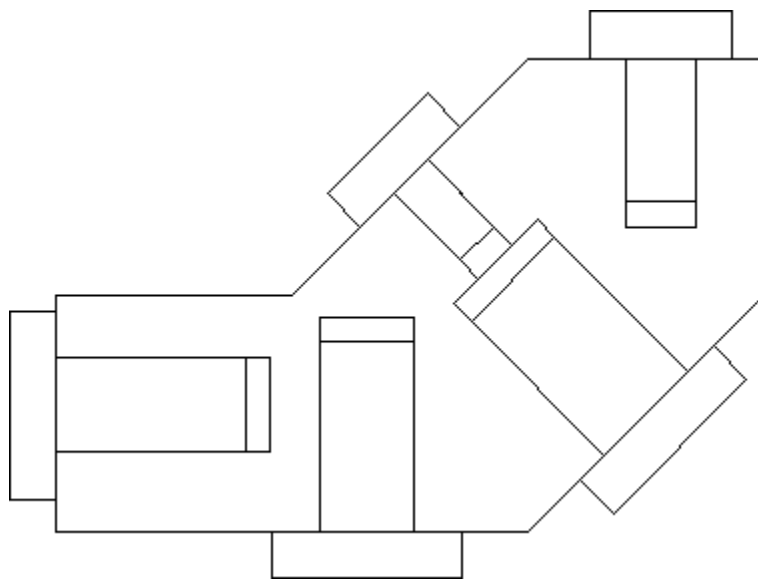
# Vector Analysis

The next figure shows the spigot in a general position. `P1` through `P8` are vectors originating at the origin. To keep the drawing simple, the line from the origin is shown only for `P1`. The vectors derived from the spigot dimensions are also shown:

**Figure 16. Spigot—General Dimensions**
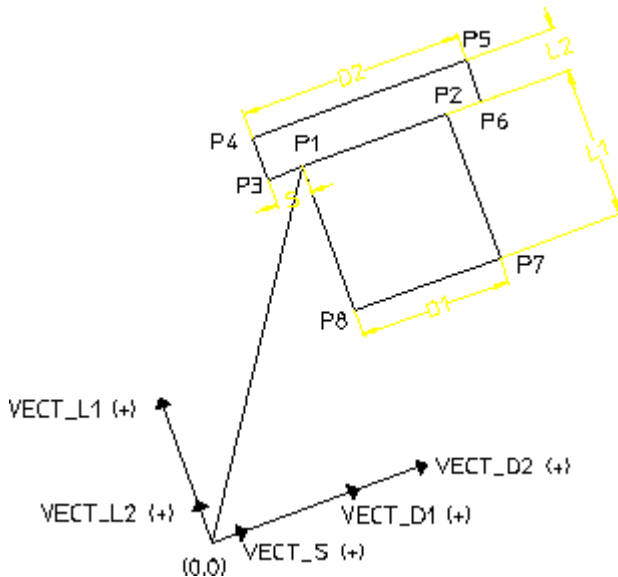


Dimensions such as `D1` and `L1` (refer to Figure 16. Spigot—General Dimensions on page 98) cannot be used in vector analysis: vectors need a direction as well as a length. However, it is simple to convert dimensions to vectors if the angles are known or can be calculated.

Let's take the dimension `S` first. The angle at which `S` acts is the same as the angle of `(P2-P1)`. Since `P2` and `P1` are vectors, the angle of the vector `(P2-P1)` can be calculated from `ANG(P2-P1)`. `ANG` is a built-in function.

If we say that the vector derived from `S` is called `Vect_S`, then:

`Vect_S = (PNT_RA S ANG(P2_P1))`

where `PNT_RA` is a built-in function that converts a length and an angle to a vector.

To convert `L1` and `L2` to vectors, note that the angle will be 90 degrees greater than the angle of `(P2-P1)`. For example:

`Vect_L1 = (PNT_RA L1 (ANG(P2-P1)+90))`

The other vectors are defined in a similar way. Note that in Figure 16. Spigot—General Dimensions on page 98, the vectors representing the dimensions are all positive in the directions shown. In Figure 16. Spigot—General Dimensions on page 98 we drew the spigot in any arbitrary position. This means

that, in many positions, the vectors of the dimensions will actually be negative. This does not matter. As long as you use a consistent "sign convention", you do not need to worry about the sign of a vector.

# Describing the Data File

The data file the macro uses is called `\anna\spigot.dat`. The data in the file is:
```
10 30 20
15 30 30
20 40 40
25 40 40
```

From left to right, the data represents the spigot dimensions `D1`, `D2`, and `L1` in millimeters. These dimensions were arbitrarily chosen so that there is no fixed relationship between the values. If there is a fixed relationship, for example `D2=3xD1`, and `L1=2xD1`, then it is easier to write the equations in your macro.

# Analyzing the Macro

The macro is shown in the following lines. You have already seen many of the commands and functions in the previous chapter, "File Input/Output and Text Strings".
```
DEFINE Spigot
  {local variables here}

  LET File1 '\anna\spigot.dat'
  OPEN_INFILE 1 File1

  LET File1 '\anna\spigot.dat'
  OPEN_INFILE 1 File1

  LET L2 10 {height of head is constant for simplicity}
{the main loop is executed once for each pair of points }
{digitized by the user}
  LOOP
    READ PNT 'Digitize first point, or END'  P1
    READ PNT 'Digitize second point' P2

    LET Hole_diam (ABS(P2 - P1)) {we don't want negative }
                                 {lengths}
    OPEN_INFILE 1 File1
    LOOP
      READ_FILE 1 Line_of_data
      LET Line_pos (POS Line_of_data ' ')
             {find the position of the first blank in the line}
      LET Data_1 (SUBSTR Line_of_data 1 (Line_pos -1))
                   {Data_1 finishes just before the first blank}
```

```
        LET D1 (VAL Data_1)            {convert from text to numeric}
    EXIT_IF (ABS(D1 - Hole_diam) < 0.0001)   {we want the spigot}
                               {to be a friction fit in the hole}
    END_LOOP
    LET Line_length (LEN Line_of_data)

{mark the start position of the next data item, then}
{step along until the next blank is found}
    LET Start_pos (Line_pos+1)
    LET Line_pos (Line_pos+1)
    LOOP
      LET Next_char (SUBSTR Line_of_data Line_pos 1)
    EXIT_IF (Next_char = ' ')
      LET Line_pos (Line_pos+1)
    END_LOOP
    LET Data_2 (SUBSTR Line_of_data Start_pos (Line_pos-1))
{the last data starts from the position after the blank, and is}
{of length (Line_length-Line_pos)  }
    LET Data_3 (SUBSTR Line_of_data (Line_pos+1)
                      (Line_length-Line_pos))
    LET D2 (VAL Data_2)                      {convert to numeric}
    LET L1 (VAL Data_3)
    LET S (0.5*(D2 - D1))
    LET Angle (ANG(P2-P1))

{convert all lengths to vectors}
    LET Vect_S (PNT_RA S Angle)
    LET Vect_L2 (PNT_RA L2 (Angle+90))
    LET Vect_D2 (PNT_RA D2 Angle)
    LET Vect_L1 (PNT_RA L1 (Angle+90))
    LET Vect_D1 (PNT_RA D1 Angle)

    LET P3 (P1 - Vect_S)
    LET P4 (P3 + Vect_L2)
    LET P5 (P4 + Vect_D2)
    LET P6 (P5 - Vect_L2)
    LET P7 (P2 - Vect_L1)
    LET P8 (P7 - Vect_D1)

{connect the points}
    LINE POLYGON P1 P3 P4 P5 P6 P2 P7 P8 P1
  END_LOOP
END_DEFINE
```

Let's start our analysis with the following line:
```
LET Hole_diam (ABS(P2 - P1))
```

When we search the data file, we will look for a value of `D1` that is equal to `Hole_diam`. Since it is possible for `(P2-P1)` to be negative, we are taking the absolute value so that we can check for equality.
```
LET D1 (VAL Data_1)
```

`Data_1` is a text string. We want to be able to compare `Data_1` with `D1`, which is a number. So we convert `Data_1` to a numeric value.
```
EXIT_IF (ABS(D1 - Hole_diam) < 0.0001)
```

We want to search the file until we find a value of `D1` that is equal to `Data_1`. You might expect us to use an equality statement such as the following:
```
EXIT IF (D1 = Hole_diam)
```

In any programming language, programmers avoid equality statements in these situations because of problems of accuracy. Let's say you digitize the 20 mm diameter hole at the left of Figure 14. Sectional View of Test Block on page 97. The computer may compute the hole diameter as 20.0000000000001 mm. When the macro is reading the third line in the data file, the statement as seen using the `trace` facility would be:
```
EXIT IF (D1 20 = Hole_diam 20.0000000000001) 0
```

The `0` at the end of the line indicates false, so there is no equality.

The actual statement we use in the macro says that if the difference between `D1` and `Hole_diam` is acceptably small, we will accept `D1`.
```
LET P3 (P1 - Vect_S)
```

Figure 16. Spigot—General Dimensions on page 98 shows the positive direction of `Vect_S`. To get from `P1` to `P3`, we have to go in the negative direction. So `P3` = `P1` – `Vect_S`. If we had defined the positive direction of `Vect_S` to be downwards to the left, then the correct equation would be `P3` = `P1` + `Vect_S`. This demonstrates that as long as you follow your own sign convention, the signs of vectors can always be determined.
```
LET P4 (P3 + Vect_L2)
```

To get from `P3` to `P4`, we go in the positive direction of `Vect_L2`, so we add `Vect_L2`. For the remaining vectors, the decision whether to add or subtract also depends on the direction that you have chosen as positive.

## Refining the Macro

This macro has been kept as simple as possible to show the principles and to give you as little typing as possible. The standard parts data that you store in text files can be for simple parts (such as washers, gaskets, and studs), or very complex parts (such as mechanical seals and piston assemblies). The principles are the same.

Even for drawing spigots, the macro is very crude. Here are some obvious improvements:

- In our macro, `P1` and `P2` must be digitized in the correct order so the macro knows the correct attitude of the spigot. If you digitize the points in the wrong order, the spigot is drawn as a mirror image of the correct position (try it!). You can rewrite the macro so that the user must digitize a third point (for

example, a point at the bottom of the hole), so that the macro knows the attitude of the hole.

• For a spigot, we need a friction fit, so the diameter of the spigot is the same as the diameter of the hole. In cases where a loose fit is required, the macro must be able to centralize the item in the hole.

• In our file `spigot.dat`, each data item consists of two digits, and the data is arranged with only one blank between each data item. Normally, tabular data items will have different lengths and each column will be right-justified. So, there will be a varying number of blanks between each data item. Your macro must allow for this.

• In the case of our simple spigot, there was only one row of data for each value of `D1`. If you have several rows of data for each value of `D1`, (for example, several values of `D2` for each value of `D1`), you must add a loop within a loop to search for the required value of `D2`.

• In the case of `P1` and `P2` it is assumed that the value of (`P1`-`P2`) corresponds to a value of `D1` in the file `spigot.dat`. If the user inputs two points `P1` and `P2` that do not meet the condition the loop will be executed until the string `END-OF-FILE` is read. Further execution will be stopped. To allow for this situation you should add a line that exits the loop if `END-OF-FILE` is read and displays a message to tell the user that the input was incorrect.

# 9

# Useful Macros

This chapter contains some macros that may be useful during your CAD operations. Even if you don't want to use a macro, you can still learn a lot by studying the macro and trying to understand it.

If you only want to try these macros, you can save typing time by omitting local variables and comments.

# Drawing Construction Lines at Angles to Existing Lines

The following macro draws a construction line to intersect an existing line at a specified angle. The existing line can be a construction line or a geometry line (real or imaginary).

```
DEFINE Ang_c_line

LOCAL P1
LOCAL P2
LOCAL Ang1
LOCAL Ang2

LOOP
  CATCH ELEM
  READ PNT
    'Pick intersection point on existing line' P1
  READ PNT
    'Pick a second point on existing line' P2
  READ NUMBER
    'Enter angle of C_line from existing line' Ang1
  LET Ang2 (ANG(P2-P1))
  C_LINE PT_ANG P1 (Ang1+Ang2)
  END
END_LOOP
END_DEFINE
```

# Splitting a Line into Equal Segments

This macro splits a line (real or imaginary) into a specified number of equal segments. The macro then draws construction lines through the split points. These construction lines can be horizontal, vertical, or perpendicular to the existing line.

```
DEFINE cline_mix

{local variables here}

  LOOP
  READ STRING
    "ENTER 'H', 'V', OR 'P' FOR C_LINE HORIZ, VERT, OR PERP" Q
  EXIT_IF ((Q='H') OR (Q='h') OR (Q='V') OR (Q='v') OR
    (Q='P') OR (Q='p')) {this line must be joined to previous line}
  END_LOOP

  READ NUMBER
    'ENTER NO. OF SECTIONS FOR SPLITTING LINES (2,3,4,..)' N
  READ PNT 'ENTER start point of line' P1
  READ PNT 'ENTER end point of line' P2
  LET Fraction (1/N)
```

```
    LOOP
        LET N (N-1)
    EXIT_IF (N=0)
        LET PN (N*Fraction)
        LET PNN (P2-(P2-P1)*PN)
        IF ((Q='H') OR (Q='h'))
          C_LINE HORIZONTAL PNN
          ELSE_IF ((Q='V') OR (Q='v'))
            C_LINE VERTICAL PNN
          ELSE
            C_COLOR BLACK
            C_LINE P1 P2
            C_COLOR RED
            C_LINE PERPENDICULAR P1 PNN
        END_IF
      END_LOOP
    IF (Q='P')
      DELETE SELECT C_LINES BLACK CONFIRM END REDRAW
      ELSE
      END
    END_IF
END_DEFINE
```

# Drawing a Round-Ended Slot

This macro will draw a round-ended slot of any size, at any angle to the axes.



```
DEFINE A_slot_macro
  LOCAL W
  LOCAL P1
  LOCAL V
  LOCAL P2
  READ NUMBER 'Enter the slot width' W
    LOOP
    FOLLOW OFF
    COLOR WHITE
    LINETYPE SOLID
    READ PNT 'Pick one center point' P1
    READ PNT 'Pick the other center point' RUBBER_LINE P1 P2
    LET V (ROT (( P2 - P1 ) * ( W /2 ) / (LEN (P2 - P1))) 90)
    ARC CEN_BEG_END P1 ( P1 + V ) ( P1 - V )
    ARC CEN_BEG_END P2 ( P2 - V ) ( P2 + V )
    LINE POLYGON ( P1 - V ) ( P2 - V )
```

```
      LINE POLYGON ( P1 + V ) ( P2 + V )

      LET V ( V * 1.5 )
      COLOR YELLOW
      LINETYPE DOT_CENTER
      LINE POLYGON ( P1 - V ) ( P1 + V )
      LINE POLYGON ( P2 - V ) ( P2 + V )
      LET V ( ROT V 90 )
      LINE POLYGON ( P1 + V ) ( P2 - V )
      END_LOOP
    COLOR WHITE
    LINETYPE SOLID
END_DEFINE
```

# Drawing Regular Polygons

This macro will draw a regular polygon with any number of sides, and with
inscribed and circumscribed circles.
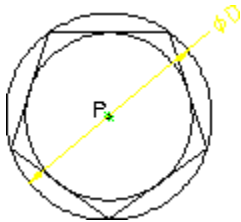


```
DEFINE A_polygon_macro
  LOCAL P
  LOCAL D
  LOCAL N
  LOCAL R
  LOCAL A
  LOCAL P1
  LOCAL P2
  LOCAL Pm
    LOOP
    READ 'Pick the center point of the polygon' P
    READ 'Enter the diameter of the circumscribing circle' D
    READ 'Enter the number of sides in the polygon ' N
    LET R (D/2)
    LET A ((180-(360/N))/2)
    LET P1 (P+PNT_XY(R*COS A)(R*SIN A))
    LET P2 (P+PNT_XY(-R*COS A)(R*SIN A))
    LET Pm ((P1 + P2) /2)
    LINE P1 P2
    MODIFY Pm ROTATE COPY (N -1) CENTER P (360/N)
    CIRCLE P (R*SIN A)
    CIRCLE P R
    WINDOW FIT
    END_LOOP
```

```
END_DEFINE
```

# Fitting text around a circular object

This macro will fit a text string around a circular object.



```
DEFINE T_rot
  LOCAL T
  LOCAL Cp
  LOCAL Sp
  LOCAL A
  LOCAL N
  LOCAL Da
  READ STRING 'Enter text' T
  READ PNT 'Enter center point' Cp
  READ PNT 'Enter start point' Sp
  READ NUMBER 'Enter angle' A
  LET N (LEN T)
  LET Da (A/N)
  LET A (ANG (Sp - Cp))
  LET N 1
  WHILE (N< =LEN T)
    TEXT_ANGLE (A - 90)
    TEXT (SUBSTR T N 1) Sp
    LET A (A - Da)
    LET Sp (Cp+PNT_RA (LEN (Sp - Cp)) A)
    LET N (N+1)
  END_WHILE
  END
  TEXT_ANGLE 0
END_DEFINE
```

# Showing the Different Z-Levels of a Hidden-Line Drawing

This macro displays each z-level in a drawing, one at a time. A logical table is created in RAM to store the necessary information.

Before you can use this macro, your hidden-line module must be activated, and you must have a drawing with assigned z-levels on your screen.
```
DEFINE Scan_z_levels
```

```
    LOCAL Range_min
    LOCAL Range_max
    LOCAL Act_line
    LOCAL Max_line
    LOCAL Act_val
    WINDOW FIT
    HL_INQ_Z_VALUE RANGE
    LET Range_min ( INQ 3 )
    LET Range_max ( INQ 4 )
    LET Act_line 0
    CREATE_LTAB "Range_drawing"     {create a logical table in RAM}
    { fill the logical table with all z values in the drawing}

    REPEAT
      LET Act_line ( Act_line + 1 )
      HL_INQ_Z_VALUE NEXT
      LET Act_val ( INQ 3 )
      WRITE_LTAB "Range_drawing" Act_line 1 Act_val
    UNTIL ( Act_val = Range_max )

    LET Max_line Act_line
    {Switch all geometry off and show only geometry having no Z-values}
    HL_REDRAW_MODE ON CURRENT
    SHOW GLOBAL ALL ON
    SHOW DIMENSIONS OFF
    SHOW HATCHING ALL OFF
    SHOW TEXTS ALL OFF
    HL_VISUALIZE GLOBAL ALL OFF
    DISPLAY "First of all, you see all elements without a z-value"
    LET Act_line 0
    {show elements on each z-value, one at a time }

    REPEAT
      LET Act_line ( Act_line + 1 )
      LET Act_val ( READ_LTAB "Range_drawing" Act_line 1)
      SHOW GLOBAL ALL OFF
      HL_VISUALIZE GLOBAL Act_val Act_val CYAN
      DISPLAY ("Now you see all elements with z-value " + (STR Act_val))
    UNTIL ( Act_line = Max_line )
    SHOW GLOBAL ALL ON
    DELETE_LTAB 'Range_drawing'
END_DEFINE
```

# 10

# Recording the System Operation

This chapter shows you how to store a record of all system inputs that are made in a certain period. This feature can be useful for:

- Creating fixed dimension geometric macros very quickly.

- Making demonstrations of system operation - the invisible draftsman!

Two functions can do this: - `ECHO` and `TECHO`. The features of each are outlined in the following sections.

# The ECHO Function

This function will write all system input to a file (or printer) in the form shown here:

```
Tm_screen_create_1
LINE POLYGON
-1.74255323667387E+002,-1.91693974675132E+000
-1.68756133685496E+002,1.64136935262188E+001
-1.64631741199077E+002,2.81847384876597E+000
-1.82402271788707E+002,3.73500551241448E+000
-1.78226960876530E+002,1.07617482670530E+001
-1.52869584848922E+002,6.79899966919450E-001

LINE PARALLEL
-1.53429687532262E+002,1.08635851185695E+001
-1.52767747997405E+002,3.88776078968923E+000
LINE HORIZONTAL
-1.67381336190023E+002,-3.44449251949884E+000
-1.50170908283734E+002,-2.27336872705908E+000

ARC THREE_PTS
-1.60100001306593E+002,7.50296901852502E+000
-1.50731010967075E+002,1.59643163056796E+000
-1.52716829571647E+002,1.09654219700860E+001
SPLITTING ON
CIRCLE CENTER
-1.54040708641361E+002,-2.06969502402607E+000
-1.51902134759515E+002,4.75337402757949E+000
END
END
```

The names of all commands and functions used are recorded, together with numerical input from the keyboard.

To start an ECHO file, type the following:
```
ECHO 'echofilename'
```

where echofilename is the chosen name for your ECHO file.

System inputs from now on will be recorded.

To close an ECHO, file type the following:
```
ECHO OFF
```

To replay an ECHO file, type the following:
```
INPUT 'echofilename'
```

where echofilename is the name previously given to the ECHO file.

# Using ECHO for Creating Macros

You can use the `ECHO` file as a basis for creating a macro. However, with some types of macro you gain nothing by using `ECHO`. For example, the spigot macro that we looked at in Using Dimensions Stored in a Data File on page 95 is almost completely involved with vector analysis and reading files. Tasks that involve a lot of digitizing and changing menus when done manually are the best type of tasks for `ECHO`.

Look at the hatched block in the next figure:

**Figure 20. Block Created Using ECHO**
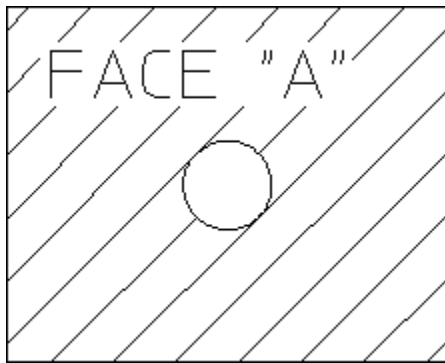


Here are the contents of the `ECHO` file recorded during creation of the block:

```
TM_CREATE_1
LINE RECTANGLE
-6.263595945324684,8.418272950516371
5.595479044490039,0.7683344359598321
TM_CONSTRUCT_1
-6.230190100282952,8.418272950516371
5.628884889531771,0.7683344359598321
-6.230190100282952,0.7683344359598321
5.562073199448307,8.418272950516371
CIRCLE CENTER
-0.350761372938188,4.610006615758968
1.252719189064929,4.409571545508578
TM_END

Tm_text_1
TEXT_SIZE
1
I_set_font_1 "hp_i3098_v"
TEXT Restore_old_text_font
'FACE "A"'
-5.161203058947541,6.71457485338806
Tm_hatch
HATCH_DIST
1
HATCH AUTO
```

```
3.991998482486922,4.409571545508578
TM_END
echo off
```

The construction lines were used to find the intersection of the diagonals of the rectangle, so that the center of the circle could be visually located. In a macro, the coordinates of the center will be known, so no construction lines are necessary.

We can edit the ECHO file, using our own coordinates for points:
```
DEFINE Hatch_block
LINE RECTANGLE
-5,4
5,-4
CIRCLE CENTER
0,0
-1,0
END

TEXT_SIZE
1
I_set_font_1 "hp_i3098_v"
TEXT Restore_old_text_font
'FACE "A"
-4,2
HATCH_DIST
1
HATCH AUTO
3,0
END
END_DEFINE
```

To create the macro, do the following:

- Put DEFINE 'filename' at the beginning and END_DEFINE at the end.

- Remove references to menus. For example, remove TM_CREATE_1, Tm_text_1, and Tm_hatch.

- Remove echo off.

- Replace TM_END by END.

Note that you use END more often in a macro than you do when working manually. When you are working manually, you can terminate a recursive command or function by starting a new command.

# 11

# Using the Interface to Find a Command

In this chapter, we "look behind" the user interface to see what commands are activated each time you pick an item on a screen menu. This helps you to relate a familiar action on the user interface to an unfamiliar macro command or function.

# What Command Will You Use?

When you start to write your macro, what commands and functions will you use? We saw in Recording the System Operation on page 109 that a good way to start a macro is to use ECHO, and then modify the ECHO file. This is fine for a "drafting" macro that saves the user from doing a lot of digitizing and menu changing. But many macros contain only a few commands that actually show visible results, such as drawing a line or creating a new viewport. The remainder of the commands perform calculations and vector analysis. The spigot macro in Using Dimensions Stored in a Data File on page 95 is a good example. For such macros, ECHO will not help you much.

If you want to create a viewport in a macro, what command will you use? Ask yourself: "How do I do this on the user interface?". If you can find out what internal command is activated when you press **CREATE**, this is the command to use in your macro.

## Screen Commands and Functions

Here's how to find the commands that lie behind the screen menu slots. Let's take the INFO menu as an example. Type on the command line:
```
EDIT_MACRO
```

The system responds with the prompt:
```
Enter macro_name or ALL
```

Now press **INFO**. The following macro is displayed on the screen:
```
DEFINE Tm_info
   Sm_info
END_DEFINE
```

Now type:
```
EDIT_MACRO Sm_info
```

The macro Sm_info is displayed on the screen. Here it is:
```
DEFINE Sm_info
  IF (I_port)
    Check_i_port
  END_IF
  IF (NOT I_port)
    CURRENT_MENU '' T_clear_menu
    MENU
    BLACK
    YELLOW '          INFO' ''  1 1
    MENU
    BLACK
    WHITE 'ADD SELECT' 'ADD_ELEM_INFO' 3 1
    MENU 'Element' 'ADD_ELEM_INFO' 3 2
    MENU
    BLACK
    CYAN 'SCREEN' 'SCREEN' 4 2
```

*Creo Elements/Direct Drafting Writing Macros*

```
            MENU
            BLACK
            WHITE 'ADD CURRNT' 'ADD_CURRENT_INFO' 5 1
            MENU
            BLACK
            WHITE 'EDIT' 'EDIT_ELEM_INFO' 7 1
            MENU 'Element' 'EDIT_ELEM_INFO' 7 2
            MENU 'Current' 'EDIT_CURRENT_INFO' 8 2
            MENU
            BLACK
            WHITE 'DELETE' 'DELETE_ELEM_INFO' 9 1
            MENU 'Element' 'DELETE_ELEM_INFO' 9 2
            MENU 'Current' 'DELETE_CURRENT_INFO' 10 2
            MENU
            BLACK
            WHITE 'CHANGE' 'CHANGE_ELEM_INFO' 11 1
            MENU 'Element' 'CHANGE_ELEM_INFO' 11 2
            MENU 'Global' 'CHANGE_GLOBAL_INFO' 12 1
            MENU 'Current' 'CHANGE_CURRENT_INFO' 12 2
            MENU
            BLACK
            WHITE 'LIST' 'LIST_GLOBAL_INFO' 13 1
    END_IF
END_DEFINE
```

You can see that the displayed text for row 1, column 1 is ADD SELECT. The corresponding command (sometimes called the "action text") is ADD_ELEM_INFO. The displayed text for row 8, column 2 is Current. The corresponding command is EDIT_CURRENT_INFO.

Use this method to find the contents of any screen menu.

## Summary

In summary, if you want to know what commands or functions to use in a macro to do a specific task:

- Decide how you would do the task using the user interface.
- Find out what command "lies behind" the user interface.
- If necessary, read the description of the command in the online help.
- Use the command, plus the correct options, in your macro.

# 12

# Customizing

When Creo Elements/Direct Drafting starts, it reads and executes a number of files which define a variety of functions (such as the screen menus, and Creo Elements/Direct Drafting environment). These files are executable programs called macros, which you can edit and store under your own filenames to be reloaded and used as required. It is therefore possible to create your own alternative definitions for the screen menus and Creo Elements/Direct Drafting environment. This process is known as customizing.

# What Is the Creo Elements/Direct Drafting Environment?

The Creo Elements/Direct Drafting environment consists of such things as the color of construction or drawing lines on the screen, the size and color of dimension text, whether drawing lines are split or not while drawing, the drawing scale and units, and so on.

All these are controlled by the settings of the environment functions, of which there are more than 70 in the system. An example of an environment function is DIM_COLOR. This can be set to produce dimensions with any of the colors available.

The current settings of the environment functions can be stored in a file. Here is a listing of part of an Creo Elements/Direct Drafting environment file:

```
MAX_FEEDBACK 100
CONFIGURE_EDITOR '$' 1 79
UNITS 1 MM
UNITS 1 DEG
CS_REF_PT 0,0
CS_AXIS 1,0  0,1
FOLLOW OFF
GRID_FACTOR 10
CURRENT_FONT 'hp_block_v'
TEXT_FRAME OFF
TEXT_ANGLE 0
TEXT_ADJUST 1
TEXT_LINESPACE 2.2
TEXT_FILL OFF
TEXT_SIZE 1
TEXT_RATIO 1
TEXT_SLANT 0
LINE WHITE SOLID END
C_LINE RED DOTTED END
TEXT WHITE END
SPLITTING ON
ARROW_FILL ON
```

At startup, the functions are set as shown in the listing to provide a standard environment. During normal use the settings are changed from the screen menu as required.

It is also possible to change the function settings by editing the Creo Elements/Direct Drafting environment. This method is useful because it gives an overview of the status of all environment functions at once.

A customized environment can be stored in a named file. This file can then be used to restore the same environment whenever it is required.

# Customizing the Creo Elements/Direct Drafting Environment

The environment will be displayed and the function settings can be changed as required. When you have finished changing the environment, press `[CTRL] [D]` to implement all the changes on the system. The current environment can be stored for future use in a named file by picking the **SAVE ENV** option from the screen menu and entering:

```
'env_filename'
```

where `env_filename` is a name of your own choice.

To execute the stored file and redefine the system environment, enter:

```
INPUT  'env_filename'
```

# How Screen Menus are Created

The screen layout consists of a menu from which commands can be given and a working area in which drawings are produced. The menu layout provides active areas (called slots) from which commands can be called directly. Names and mnemonics associated with these commands can be displayed in the slots.

A typical menu listing is shown at the beginning of Using the Interface to Find a Command on page 113. If you study this listing, you can see that, for each row and column, the "display" text appears before the "action text".

The names of the system macros that display the screen menus can by found by entering `EDIT_MACRO` and picking one of the screen menu pads. The display will become as follows:

```
  DEFINE action_text
    action
  END_DEFINE
```

In this case action is the name of the macro that displays the screen menu. Each of these macros can be edited and stored in a file for future use.

Here is how to save the macro that displays a screen menu. Let's use the CREATE 1 menu as an example:

1. Use `EDIT_MACRO` to get the macro name from the screen menu. You will find that the macro for the CREATE 1 menu is `Sm_create_1`.

2. On the command line, type:
   ```
   SAVE_MACRO Sm_create_1 'filename'
   ```

`filename` is a name of your choice.

Now you can edit the file. You may want to add the contents of the file to your `customize` file.

## Menu Variables

Here are the first few lines of the CREATE 1 menu, showing the menu variables:

```
DEFINE Sm_create_1
  LET Lastmen 'Tm_create_1'
  IF (I_port)
    Check_i_port
  END_IF
  IF (NOT I_port)
    MENU_BUFFER ON
    CURRENT_MENU Sm_create_1_layout_name
    T_clear_menu
    Menu_control_icons
    MENU Colo0 Bcol5 CENTER 'CREATE 1' '' 1 3
.
.
.
    MENU Colo0 Bcol5 CENTER 'SPLINE' 'Tm_create_3' 25 2
    Eight_menu_slots_add
  END_IF
END_DEFINE
```

The meaning of each of these variables is as follows:

- `I_port`

  The next variable `I_port` relates to the **LARGE/SMALL** pad underneath **PORT** on the formerly used tablet. When **LARGE/SMALL** is picked, the macro `Tm_port_large_slash_small` is executed and the coordinates of the current viewport are stored. The current viewport is then enlarged so that the entire screen becomes available for creating geometry. At the same time, the variable `I_port` is set to `1` to inform the system that a large viewport is on display and if a menu pad is picked, nothing will happen. When **LARGE/SMALL** is picked again, the current port is reduced to its original size, and the variable `I_port` is set to `0` so that menus can be changed in the usual way.

- `CURRENT_MENU Sm_create_1_layout_name`

  This sets the name of the current screen menu to `Sm_create_1` (`Sm_create_1_layout_name` is a macro that will be expanded to that string). All menu layouts and menu commands will be related to this current menu, until a current menu having a different name is used. To define your own menus we recommend using '' (an empty string) as layout name.

- `T_clear_menu`

  This macro clears all text from the menu so that new text can be entered.

- `Menu_control_icons`

  This macro inserts the icons for pinning, moving and removing the menu into the first line of the menu.

- `MENU Colo0 Bcol5 CENTER 'CREATE 1' 1 3`

  This line finishes the first menu row. `Colo0` and `Bcol5` will expand into a foreground and a background color, which is defined according to the `MEPELOOK` at system startup time. All system defined menus use these macros for specifying foreground and background colors. The following table gives an overview on how these macros relate to the menu color if `MEPELOOK` is set to `0`.

| Foreground | Background | COLOR (MEPELOOK=0) |
|---|---|---|
| Colo0 | Bcol0 | BLACK |
| Colo1 | Bcol1 | WHITE |
| Colo2 | Bcol2 | RED |
| Colo3 | Bcol3 | GREEN |
| Colo4 | Bcol4 | BLUE |
| Colo5 | Bcol5 | YELLOW |
| Colo6 | Bcol6 | MAGENTA |
| Colo7 | Bcol7 | CYAN |

- `Eight_menu_slots_add`

  `Eight_menu_slots_add` is a macro which updates the last 8 user customizable slots. It is used in every standard menu. You can place your own menus items in these 8 slots using the `Eight_menu_slots` macro. The syntax for this macro is

  ```
  Eight_menu_slots 'title' 'action' Row_number Column_number
  ```

  The string `title` will be displayed in the slot, `action` is the action text, which will be executed if the menu slot is pressed. `Row_number` and `Column_number` are the corresponding row and column numbers. The valid range for the row number is 26 to 27, and 1 to 4 for the column number.

## Physical Layout of Menu Slots

You have probably noticed that, with only a few exceptions, all your screen menus have a similar layout of rows and columns. This is because most of the menus use the same "template". This template is created using a macro called `Layout_body_1`. This macro is used for example to define the layout of the `Sm_create_1` menu:

```
DEFINE Layout_body_1
  Headline_height    ' | |                   | '
  Text_slot_height   '              |        '
  Text_slot_height   '              |        '
  Text_slot_height   '              |        '
  Text_slot_height   '              |        '
  Text_slot_height   '              |        '
  Text_slot_height   '              |        '
```

```
Text_slot_height   '          |          '
Text_slot_height   '          |          '
Text_slot_height   '          |          '
Text_slot_height   '          |          '
Text_slot_height   '          |          '
Text_slot_height   '          |          '
Text_slot_height   '          |          '
Text_slot_height   '          |          '
Text_slot_height   '          |          '
Text_slot_height   '          |          '
Text_slot_height   '          |          '
Text_slot_height   '          |          '
Text_slot_height   '          |          '
Text_slot_height   '          |          '
Text_slot_height   '          |          '
Text_slot_height   '          |          '
Text_slot_height   '          |          '
Text_slot_height   '          |          '
Bottom_slot_height '    |     |     |    '
Bottom_slot_height '    |     |     |    '
END_DEFINE
```

Here is an explanation of the variables used in the above macro:

- `Headline_height`

  The height of the menu title slot.

- `Text_slot_height`

  The height of the slots containing the display text.

- `Bottom_slot_height`

  The height of the small boxes at the bottom of each menu.

The slot height values are calculated during startup. This allows the menu size to be adapted to suit the window or display size on which Creo Elements/Direct Drafting was started. The values are defined in the `hp_macro.m` file. These definitions can be used as examples of how the slot sizes can be made self-adapting for different window or display sizes.

This template macro is used in the following macro, which defines the `CREATE 1` menu layout:

```
DEFINE Sm_create_1_layout
  CURRENT_MENU 'Sm_create_1'
  CURRENT_SCREEN 1
  MENU_LAYOUT Menu_position RIGHT
    Layout_body_1
    Menu_home_point_top
  END
  MENU_STATUS ENABLE_INQ
  LET  Sm_create_1_layout_name 'Sm_create_1'
END_DEFINE
```

Here is an explanation of the variables used in the above macro:

- `CURRENT_MENU 'Sm_create_1'`

  This sets the name for the `CREATE 1` menu layout to `Sm_create_1`.

- `CURRENT_SCREEN 1`

  This ensures that even in a dual screen environment the menu will be placed on the first screen.

- `MENU_LAYOUT`

  The `MENU_LAYOUT` command causes the menu to be displayed at the right of the screen. The `MENU_LAYOUT` command is terminated by `END`.

- `Menu_position`

  A macro containing the qualifier `LOWER`. This can be set to `UPPER` if required.

- `Layout_body_1`

  This line calls the macro `Layout_body_1`.

- `Menu_home_point_top`

  Moves the menu to the correct position on the screen. This position is calculated according to the `Menu_position` macro and user interface version (screen-only).

- `MENU_STATUS ENABLE_INQ`

  This removes the inquiry protection from the menu.

- `LET Sm_create_1_layout_name 'Sm_create_1'`

  The `CREATE 1` menu does not use the layout name directly. To activate its layout the `Sm_create_1_layout_name` variable is used. Therefore `Sm_create_1_layout_name` has to be defined here to the correct string. This is usefull for deferring the actual generation of the menu layout until the menu is requested the first time. Until the first request, `Sm_create_1_layout_name` will define the macro name that generates the layout.

## Using Tables for Screen Menus

Some of the screen menus are defined using the `TABLE` series of commands, such as `TABLE_LAYOUT` and `SHOW_TABLE`. Tables are used because they allow the display of system status variables. Table scrolling allows long lists to be displayed. Examples of menus using tables are:

- `HIDDEN LINE`

These menus are created using `MENU` and `TABLE` commands.

The definition of these menus can be seen in ASCII versions of the files `hp_macro.m`, and `hp_men_t.m`.

Use copies of the existing menus to make changes or additions.

Menus that are defined using tables are secured against deletion or overwriting. This avoids potential problems caused by commands such as `DELETE_TABLE_ALL`.

If you want to use your own menu tables, you should move the `SECURE_TABLE` statements in the previously mentioned files to the end of your table definitions. The logical tables for the default menus should not be changed because they are accessed by the Creo Elements/Direct Drafting code.

For further information about the use of tables, refer to the online help.

# Customizing the Screen Menus

You can customize the menus in several ways:

- New commands can be made available in the existing menus. Add the display text, and insert the new command as the action text.
- You can activate one of your macros from an empty menu slot. Add the display text, and insert the name of your macro as the action text.
- New menus can be created using the existing menu layout.
- New menus can be created using a new menu layout.
- Customized viewport arrangements can be defined and called when wanted.

# Customizing for Local Directories

When you customize Creo Elements/Direct Drafting, you will have a number of files to store on your system's hard disk (for example, files containing your own macros, customization, geometry, and bodies). These files are normally stored in your own user directory (also called your home directory). This directory is separate from the directories of other users.

## Platform Dependencies

If you start Creo Elements/Direct Drafting while you are in your home directory, all files are stored in this directory unless you specify otherwise. In a Windows environment, you can install Creo Elements/Direct Drafting so that it always starts from your home directory. See your the configuration manual that corresponds to your operating system manual.

shows a typical hierarchy of directories. Your system may be somewhat different.

**Figure 23. Directory Hierarchy Structure**

```
                      root (/)
                     /        \
                    /          \
               me10            users
                               /      \
                              /        \
                         users1         users2
                       /   |   \
                      /    |    \
                  file1  file2  sub_dir_1
                                /       \
                               /         \
                            file1        file2
```
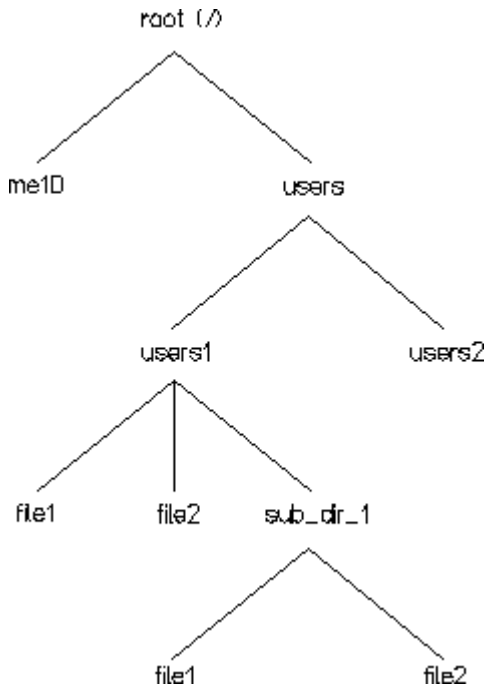
At the top of the tree is the root directory, denoted by a slash (/). Under this is a directory called `users`. Under `users` is your own user directory corresponding to your login name, together with the directories of other users. Your own directory (also called your home directory) can contain files and sub-directories. Normally you will store your files in your user directory or one of your sub-directories, but it is also possible to store files in the directories of other users, provided these directories are not write-protected.

Underneath the root directory is a directory called `me10` which contains all the Creo Elements/Direct Drafting software. Other files and directories are used by your operating system. You need not concern yourself with these.

Files and directories within the same branch of the tree structure are separated from each other with a slash, the first slash corresponding to the root directory. The full specification for a file in your user directory is:

```
/users/user_directory/filename
```

where user_directory is the directory corresponding to your own login name and filename is one of your own files.

You can use either a slash (/) or a backslash (\) to separate files and directories when specifying pathnames within Creo Elements/Direct Drafting. The following two paths specify the same file:

```
/users/user1/file2
\users\user1\file2
```

When you first log in, your own user directory is known as the current directory. If you specify a file simply by giving the filename, the system will look for the file in your current directory. If it is not there, the system will look in the `me10\me10` directory, or the directories specified in the `SEARCH` path. The `SEARCH` path is specified in the environment file.

If you wish to specify a file that is not in the current directory and not specified by the `SEARCH` path, you must specify the full path name, starting from the root directory.

You can create a sub-directory under the current directory by entering the following command at the Creo Elements/Direct Drafting command line:
```
CREATE_DIRECTORY 'subdirectory'
```

`subdirectory` is a name of your choice.

To make the subdirectory the current directory, use the CURR DIR option in the FILE screen menu and specifying the name of the new directory. If you create a file in `subdirectory`, the full specification for the file will be:
```
/users/user_directory/subdirectory/filename
```

We will now see how to produce a customized menu so that you can change from one directory to another.

## Example—Create Your Own Directory Menu

Create a file with a filename of your own choice using EDIT FILE from the FILE screen menu and enter the text of a macro. Here is an example:
```
DEFINE Directory
  IF (NOT I_port)
    CURRENT_MENU ''
    MENU Colo1 Bcol0 '' '' BOX  1 1 38 7
    MENU Colo0 Bcol5 CENTER 'DIRECTORY' 1 1
    MENU Colo1 Bcol0 'users' 'CURRENT_DIRECTORY "/users"' 3 1
    MENU 'me10' 'CURRENT_DIRECTORY "/me10"' 3 2
    MENU 'Jim' 'CURRENT_DIRECTORY "/users/jim"' 4 1
  END_IF
END_DEFINE
```

The first `MENU` statement sets the menu heading. The second `MENU` statement defines all the boxes under the heading to be blank with a black background, but to have white text as soon as some text is written into them.

The third `MENU` statement defines a slot to represent the directory of users. The one after that represents the directory containing the Creo Elements/Direct Drafting software, and so on.

Note that all the `CURRENT_DIRECTORY` statements specify the full pathname for the directories, starting with the root directory (/). This ensures that the directories can always be found, regardless of where they are in the directory tree structure.

The list of `MENU` statements can continue until the slots in the menu layout are filled.

The numbers at the end of each `MENU` statement represent the row and column number of the slot.

In this example, the last `MENU` statement defines a slot for displaying the current catalog on the screen. When the menu is displayed, you can change the current directory by picking any named slot and then display the directory by picking `CATALOG`.

If you input the file and run the macro, the menu is displayed. You can select directories by picking the named slots.

# Customizing the Keyboard

When working with Creo Elements/Direct Drafting, you may use some commands more frequently than others. This section shows how to customize the keyboard to reduce the time required to call one of these commands.

Your keyboard has eight or twelve function keys labelled f1 to f8 or f1 to f12. All of these keys can be customized.

For example, you may wish to use commands that are not available on your current screen menu. To avoid having to change the screen menu to issue a command, customize a function key so that the required command is written on the input line or directly executed whenever the function key is pressed. Refer to the explanation of `DEFINE_KEY` in the online help.

The function keys also can be used to display characters that are not available on the keyboard. Each character has a number associated with it, known as an ASCII number. The numbers range from 0 to 255, and most of them represent readable characters, although some of them are used for control functions, such as `ESC`, `RETURN`, `TAB` and `DEL`. A list of all the characters that produce useful text is given at the end of this chapter under the heading The Keyboard Input Characters.

The characters that cannot be used for text are the ones with ASCII numbers 0-31, 127-159 and 255.

Any of the text characters can be displayed on the input line by entering:
```
DISPLAY (CHR ascii_number)
```

where `ascii_number` is the ASCII number of the character.

To customize a function key so that it can be used to type a character, enter the command:
```
DEFINE_KEY key_number (CHR ascii_number)
```

where key_number is a number from 1 to 8 or from 1 to 12 representing the function key to be customized. The next time the function key is pressed, the character will appear on the user input line.

---

### 📝 Note

If a function key is customized and used to place text on a drawing from the **TEXT**, **SYMBOLS** or **DIMENSION** menus, the text on the drawing will depend on how the text font has been customized, and will not necessarily look the same as the characters that appear on the input line. See the next section on "Customizing the Text Fonts".

---

To customize a key so that it can be used to issue a command, enter:
```
DEFINE_KEY key_number 'action_text'
```

where `action_text` is the name of the required command. When the key is pressed, the command name appears on the input line. To execute it, press `[Return]`.

Quotes are needed around a command name because the system is asking for text, just as if you had typed the command on the input line. When customizing a key to produce a character, quotes are not needed because (`CHR ascii_number`) is considered by the system to be text.

The function keys also can be customized from a file. Select **EDIT FILE** from the **FILE** screen menu and enter in quotes the filename under which the file is to be stored on the disk. Type the required instructions and store the file using `[CTRL] [D]`.

Now select **INPUT** and enter the filename again. All the keys which are defined in the file will be customized.

If you customize the function keys either from the keyboard or from a file, the function key definitions will be lost whenever the system is switched off. The advantage of storing your `DEFINE_KEY` instructions in a file is that whenever you start up the system and wish to customize the function keys, all you have to do is input your file.

The following listing is an example of a file that customizes eight function keys.
```
DEFINE_KEY 1 (#14 '3' #15)
DEFINE_KEY 2 (#14 '1' #15)
```

```
DEFINE_KEY 3 (#14 '2' #15)
DEFINE_KEY 4 'LOAD'
DEFINE_KEY 5 'STORE'
DEFINE_KEY 6 'CATALOG' "" SCREEN'
DEFINE_KEY 7 'INPUT'
DEFINE_KEY 8
   'INPUT "dir_file" DEFINE Tm_macros_1 directory END_DEFINE'
```

- The first three keys are customized to produce the characters ° (degree), diameter sign, and ± (plus/minus) of the `hp_symbols` font on the drawing.

- Keys 4 and 5 are customized to load and store drawings.

- Key 6 is used to catalog the current directory on the screen. Key 7 is used to input a file.

- Key 8 is customized to input a file, and also to define the first user-defined macro pad so that when picked it executes a macro. In this example, `dir_file` is the file described in the previous section, "Customizing for Local Directories". It contains the macro called `directory`, which when executed displays the menu of user directories on the screen. Provided the file exists in the current directory, and the filename and macro name are both the same as in the `DEFINE_KEY` statement, when the function key is pressed followed by `[Return]`, all you have to do is pick the first user-defined macro pad and the menu of user directories will be displayed.

---

### 🗒 **Note**

If a function key is customized so that it executes a number of macros in series, only the last macro can contain a `READ` statement to prompt the user for input. This is because the next word in the list will be considered by the system to be the input for the `READ` statement.

---

## What is a Text Font?

A text font is a group of pre-defined geometrical patterns that can be placed on a drawing. Each pattern in a font is made up of straight lines joining points on a grid, and is associated with a keyboard character so that it can be easily called.

Although normally used for text (the standard system annotation and dimensioning texts are produced this way), text fonts also can produce special symbols. Because there can be many points in the grid, highly detailed symbols can be defined.

If you need special text or symbols, create them with your own customized text fonts and store them on disk ready for use. For example, you may want to create a text font that produces company trademarks and logos.

Text fonts are very versatile. Some of the main features:

• Characters can be filled or unfilled.

• The character width-to-height ratio can be set.

• The character slant angle can be set (forwards or backwards).

• The line angle can be set (0 to 360 degrees).

Here are some examples of text and symbols:

**Figure 24. Examples of Text and Symbols**
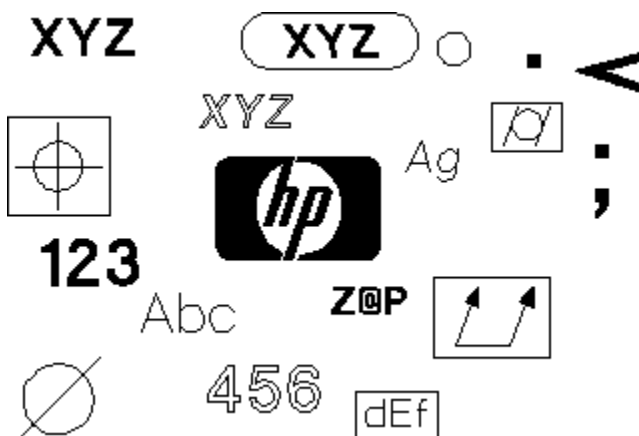


## The System Fonts

When you start Creo Elements/Direct Drafting, a number of font files are loaded into memory, but only one of them is the current font and immediately available for use. You can find out which fonts are in memory using LIST FONTS from the TEXT 2 menu and specifying the screen option.

The system displays a list of fonts loaded, fonts used, and the current font. This list it has nothing to do with the files from which they are created. The fonts `hp_Y14.5` and `hp_i3098_v` are loaded from files with the same names, but `hp_def_font` is created within the system and displays a box on the screen to represent the space that will be occupied by characters before they are drawn.

Some files available on the disk create fonts when they are loaded.

The following table gives the Creo Elements/Direct Drafting and the DOS font name, and a description for each Creo Elements/Direct Drafting drawing font:

| Drafting Font Name | DOS Filename | Description |
| --- | --- | --- |
| hp_block_c | hp_blk_c.fnt | Filled font with constant character spacing |
| hp_block_v | hp_blk_v.fnt | Filled font with variable character spacing |
| hp_d17_c | hp_d17_c.fnt | DIN17 font with constant character spacing |
| hp_d17_v | hp_d17_v.fnt | DIN17 font with variable character spacing |
| hp_jasc_c | hp_jas_c.fnt | Kanji Japanese and Greek font with constant character spacing |
| hp_jasc_v | hp_jas_v.fnt | Kanji Japanese and Greek font with variable character spacing |
| hp_kanji_c | hp_kan_c.fnt | |
| hp_symbols | hp_syms.fnt | Special characters for dimensioning: degree (°), diameter sign, plus-minus (±), minutes ('), seconds (") |
| hp_symbols2 | | Special symbols characters (tolerances) |
| hp_Y14.5 | hp_y14_5.fnt | Used for symbols only |
| hp_i3098_c | i3098_c.fnt | ISO 3098 font with constant character spacing |
| hp_i3098_v | i3098_v.fnt | ISO 3098 font with variable character spacing |

## Example—Displaying the Characters in a Font

The characters in the current font, together with their corresponding ASCII numbers, can be displayed on the screen by:

- Using the FONT EDITOR in the **TEXT 2** menu. (for further information, refer to Design and Drafting with Creo Elements/Direct Drafting.)
- Running the following macro.

(For a discussion of ASCII numbers, see the section on customizing the keyboard).

### 📋 Note

The macro uses the **WINDOW** function to set the coordinates of the current port. If the current port is not the one in that you wish the display to appear, use the CURRENT and pick the required viewport.

The macro deletes all existing 2D geometry.

```
DEFINE char_font
  LOCAL N
  LOCAL Y
  CHAR_LAYOUT
   (CHR 0) 24,27 23,21 21,17 19,15 16,13 13,12 11,12 8,13 5,15 3,17 1,21 0,27
           0,37 1,43 3,47 5,49 8,51 11,52 13,52 16,51 19,49 21,47 23,43 24,37
           24,27 36
    (CHR 1) 0,40 12,52 12,12 24
   (CHR 2) 0,46 2,49 5,51 9,52 15,52 19,51 22,49 24,46 24,42 22,38 0,12 24,12
           36
    (CHR 3) 6,36 16,36 BREAK 0,52 15,52 20,51 23,49 24,46 24,43 23,40 21,38
           19,37 16,36 20,35 22,33 23,31 24,27 24,22 23,18 21,15 18,13 15,12
           0,12 36
   (CHR 4) 18,52 0,20 24,20 BREAK 18,20 18,12 36
   (CHR 5) 22,52 0,52 0,36 15,36 20,35 23,32 24,28 24,22 23,18 21,15 18,13
           15,12 0,12 36
   (CHR 6) 0,26 1,30 3,33 6,35 9,36 15,36 18,35 21,33 23,30 24,26 24,22 23,18
           21,15 18,13 15,12 9,12 6,13 3,15 1,18 0,22 0,26 1,34 3,38 7,43
           18,52 36
   (CHR 7) 0,48 0,52 24,52 4,12 36
   (CHR 8) 10,36 5,34 2,31 0,27 0,21 2,17 5,14 9,12 15,12 19,14 22,17 24,21
           24,27 22,31 19,34 14,36 BREAK 7,51 10,52 14,52 17,51 20,49 22,46
           22,42 20,39 17,37 14,36 10,36 7,37 4,39 2,42 2,46 4,49 7,51 36
   (CHR 9) 23,30 9,30 6,31 3,33 1,36 0,40 0,42 1,46 3,49 6,51 9,52 15,52
           18,51 21,49 23,46 24,42 24,35 23,30 22,26 19,22 12,16 3,12 36
  END
  TEXT_ADJUST 1
  TEXT_ANGLE 0
  TEXT_FILL OFF
  TEXT_FRAME OFF
  TEXT_LINESPACE 2.2
  TEXT_RATIO 1
  TEXT_SIZE 3.5
  TEXT_SLANT 0
  DELETE ALL CONFIRM   (Caution! This line deletes all 2D geometry )
  WINDOW (-20,10) (80,-150)
  LET N 31
  LET Y 0
  DISPLAY_NO_WAIT 'Preparing character table'
  REPEAT
    LET N (N+1)
    IF ((N<127) OR (N>159))
      LET Y (Y-20)
      TEXT ( (CHR(N DIV 100)) + (CHR((N MOD 100) DIV 10)) + (CHR(N MOD 10))
      ( PNT_XY 0 Y ) (CHR N) ( PNT_XY 50 Y )
    END_IF
  UNTIL (N=254)
  END
END_DEFINE
```

This macro displays all the characters in the current font that can be displayed on a drawing using keyboard inputs from the TEXT, SYMBOLS, and DIMENSION menus.

It is possible to define a character within a font corresponding to any ASCII number from 0 to 255, and they can all be displayed on a drawing from within a macro, or by entering (CHR n) on the input line without quotes, where n is the ASCII number, when prompted for text to be included in your drawing.

However, you would normally be interested only in the characters that can be displayed simply by entering text when prompted for input from the **TEXT**, **SYMBOLS** and **DIMENSION** menus. For this reason the macro only lists characters that can be input using the normal keyboard characters or customized function keys. The ASCII numbers corresponding to control characters have been omitted from the list. These are the numbers `0-31, 127-159` and `255`.

The macro begins by defining local variables and then defining the ASCII numbers `0` to `9` to produce the characters `0` to `9`. These numbers would not normally be used to produce characters but have been used within the macro to avoid interfering with the characters in the current font. The characters `0` to `9` are used to produce the list of ASCII numbers in the display.

The text parameters are set to their default values and all 2D geometry is deleted. A window is defined in the current viewport and the characters are displayed, together with their corresponding ASCII numbers. When the macro is run with `hp_i3098_v` (the default text font) as the current font, the display will become as follows:

```
032    SPACE

033      !

034      "

035      #

036      $
```

# How to Create a Text Font

Each character in a font has its own cell. The cell size (in grid points) may be up to 250 × 250. The decision on how many points to use in the cell depends on the degree of detail needed: greater detail needs more points. The next figure shows an example of how the letters `A` and `g` could be defined in a text font. The same principles are applied for the definition of symbols.

## Figure 27. Creating a Font
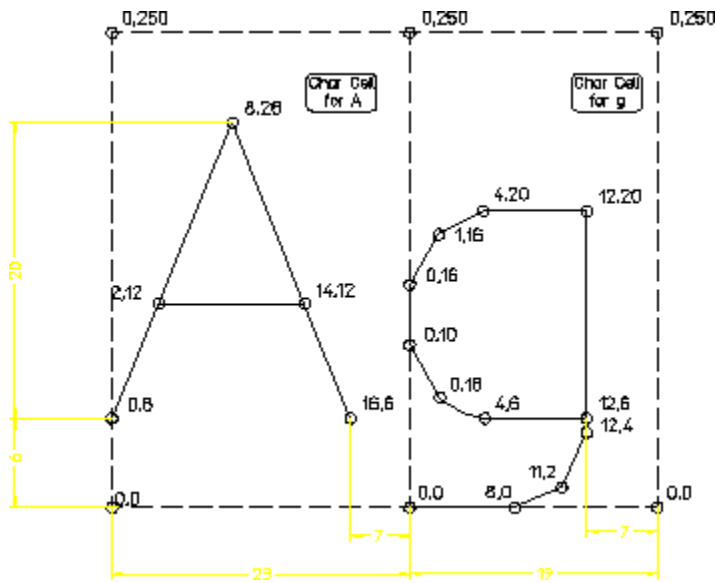


The characters are laid out in cells on a grid according to the detail required. The vertical distance between grid points is determined by the TEXT_SIZE function and the number of grid points that are fitted into the specified size. The horizontal distance between grid points is determined in the same way, except that it also can be altered using the TEXT_RATIO function, which affects the width to height ratio of points on the grid and is normally set to 1. Details of these functions are in the help file.

The command that creates a text font is DEFINE_FONT. This is how the font above is defined:

```
DEFINE_FONT 'example' 6 20 20
  CHAR_LAYOUT
  'A' 0,6 2,12 8,26 14,12 16,6 BREAK 14,12 2,12 23
  'g' 12,6 4,6 1,8 0,10 0,16 1,18 4,20 12,20 12,4 11,2 8,0 0,0 19
END
```

The name of the font is Example. The first number following this (6) is called the underlength. This represents the number of grid points that lie below the cursor when a character or symbol is placed on the screen. In text applications, this can be used as the maximum distance required for the tail of a character such as g or p.

The second number (20) is called the height. The vertical distance between grid points is the value set with the TEXT_SIZE function divided by the value of height. In text applications the value of height can represent the number of grid points corresponding to the height of a capital letter.

In the above example, if the TEXT_SIZE is left at its default value of 3.5 mm, the height of the letter A will be 3.5 mm because 20 grid points represent the height of the letter, and the value of height in the DEFINE_FONT function is also 20.

The third number in this line (`20`) is called the width. This gives the number of grid points in the horizontal direction that are equivalent to the value set with the `TEXT_SIZE` function. In this example, the width is set to the same value as the height to maintain the same character proportions as defined on the grid. If the width specification was reduced to `10`, the characters would be elongated in the horizontal direction to twice their width.

The `CHAR_LAYOUT` command starts the character definition. Each character is defined in turn with a set of grid point coordinates. A line will be drawn from point to point to form the character outline. The `BREAK` function is used to put a break in the line being drawn.

The number at the end of each character definition is the width of that character cell in grid points. This controls the space between characters.

The command `END` is used to end the font definition.

A new text font can be defined by entering the instructions from the keyboard, but it is more convenient to create a font containing all the required character definitions from within the editor, and store it as a file that can be INPUT when required. When a new font is defined, from the keyboard or from a file that is INPUT, it will become the current font and only the characters contained in the new current font will be available for use. To use any of the characters in the system fonts, or in another of your own fonts, the current font will have to be re-specified.

A number of fonts are available within the system and can be set as the current font using **SET FONT** from the screen menu **TEXT 1**. If you have defined some of your own fonts, you can select which one is to be the current font by entering the command:
```
CURRENT_FONT 'fontname'
```

where `fontname` is the name of your font.

## Storing your Fonts—ASCII Files and BINARY Files

To create a text file containing all the instructions within one of the system fonts you first must see if the font has been loaded into the system. Select LIST FONTS from the TEXT screen menu to see if the font is in the list, then press `[ESC]` to get back to the menu.

If the font is not in the list, it needs to be loaded by entering:
```
LOAD_FONT 'bin_filename'
```

where `bin_filename` is the name of the binary file containing the font.

The font can then be written to the disk as an ASCII file by entering:
```
SAVE_FONT 'fontname' 'ascii_filename'
```

where `fontname` is the name of the font, and `ascii_filename` is the name you choose for the text file containing the font. The file can then be examined and edited in the usual way.

A text file can contain more than one font. When a text file is input, all the fonts within it are loaded and the last one in the list becomes the current font.

To create a binary file containing one of your fonts, proceed as follows:

1. Create a text file using the methods described above.

2. INPUT the file so that all the fonts within it are loaded into the system.

3. Store each font on the disk in a separate binary file by entering:
   ```
   STORE_FONT 'fontname' 'bin_filename'
   ```

   where `fontname` is the name of the font and `bin_filename` is the name under which you choose to store the binary file on the disk.

The font can then be loaded back into the system using `LOAD_FONT` as described earlier in this section.

Binary files cannot be examined from within the editor. The advantage of using them is speed of access.

# Customizing the Startup Procedure

Before implementing any of the procedures described here, read the section that describes the Creo Elements/Direct Drafting startup file in the configuration manual that corresponds to your operating system.

The `custom.m` file is only input during startup if the braces { } have been removed from the following line at the end of the `\me10\startup` file:
```
{ INPUT 'custom.m' }
```

If this is done, the system looks for a file called `custom.m` according to the `SEARCH` path.

This normally means that it looks in the current directory first and then in the `\me10` directory. Throughout this discussion it is assumed that the `SEARCH` path is set up this way. After startup the `SEARCH` path can be found by selecting EDIT ENV from the SET UP screen menu.

During startup, the current directory is the home directory of the user who has logged on. You can choose one of the following options for a customized startup:

## Platform Dependencies

- Create a file called `custom.m` in the directory from which you normally start Creo Elements/Direct Drafting and edit the file so that it contains your own customizing requirements. This allows you to control the customizing procedures that are implemented during startup.

- Use the group customization file. To do this, you cannot have a file called `custom.m` in your Creo Elements/Direct Drafting startup directory. In this

case the system looks for a `\me10\custom.m` file that should have been created by the system administrator. The customization specified in this file affects all users, regardless of the startup directory.

## Customized Startup for the Whole Group

The file `\me10\custom.m` can be created, usually by a system administrator. This file can be edited so that it contains customized startup procedures that apply to any member of your group who has not created a `custom.m` file in their Creo Elements/Direct Drafting startup directory.

The `\me10\custom.m` file can be edited with any text editor, although it is more likely that the system administrator will want to develop and test the customizing procedures while running the Creo Elements/Direct Drafting software as an ordinary user and then copy or append the files into `\me10\custom.m`.

## Customized Startup for the Group and Individual Users

We have already seen how users can implement their own startup procedures by creating a file called `custom.m` in their Creo Elements/Direct Drafting startup directories. The following choices are available to users who have created this file.

*   Include the following line as the first line in your `custom.m` file:
    `INPUT '\me10\custom.m'`

    In this case the customizing procedures that have been developed by the system administrator for the whole group will be implemented first, and then your own procedures will be implemented.

*   If you wish to customize the system startup according to your own requirements without using any of the group procedures, do not include a line shown above in your customize file.

*   If you want the system to operate according to the normal defaults without any customizing at all, create a `custom.m` file in your home directory but leave it empty.

## Creating a Menu for Customized Startup

If you have created a file called `custom.m` in your home/startup directory, you can place any functions or commands in the file that you would like to have implemented during startup. In addition, you can use the file to define function keys so that other files can be input as required.

Here is an example of how you can customize the system so that a menu is displayed after startup, offering a variety of customizing procedures that may be required in different circumstances. Edit your `custom.m` file so that it contains the following lines:

```
DEFINE_KEY 1 ((CHR 4)+'INPUT"project1"'+(CHR 13))
DEFINE_KEY 2 ((CHR 4)+'INPUT"project2"'+(CHR 13))
DEFINE_KEY 3 ((CHR 4)+'INPUT"project3"'+(CHR 13))
DEFINE_KEY 4 ((CHR 4)+'INPUT"project4"'+(CHR 13))
DEFINE_KEY 5 ((CHR 4)+'INPUT"project5"'+(CHR 13))
DEFINE_KEY 6 ((CHR 4)+'INPUT"project6"'+(CHR 13))
DEFINE_KEY 7 ((CHR 4)+'INPUT"project7"'+(CHR 13))
DEFINE_KEY 8 ((CHR 4)+'INPUT"project8"'+(CHR 13))
EDIT_FILE"menu" {Must be last line in file}
```

In this example the names `project1` through `project7` are the names of files containing customizing procedures for work related to different projects. You can choose whatever names you like for your files.

The function keys have been set up so that they can be used while a menu is displayed on the alpha screen, but they also can be used when graphics are displayed. Here is a description of how the function keys are defined:

- `(CHR 4)` means `[CTRL] [D]`. This exits the alpha display so that the input statement can be executed. If graphics are displayed and the system is waiting for a command, `[CTRL] [D]` will not do anything.

- `(CHR 13)` means `RETURN`. This avoids having to press `[RETURN]` after pressing the function key.

- The first seven keys are used to input the required files, while function key `[f8]` is used to re-display the menu. The name `menu` is only an example. You can choose your own name for the file containing your screen menu.

The `EDIT_FILE` statement at the end of the customize file will display the screen menu. This should be the last line because at this point the user will be given a choice of whether or not to input other files by pressing function keys.

Here is an example of what the menu file may look like:

```
Press function key f1 to customize for project 1
Press function key f2 to customize for project 2
Press function key f3 to customize for project 3
Press function key f4 to customize for project 4
Press function key f5 to customize for project 5
Press function key f6 to customize for project 6
Press function key f7 to customize for project 7

Press ESC to display graphics
```

When a function key is pressed, the menu will disappear and the graphics will be displayed. The file that has been defined within the function key will be `INPUT` so that customizing procedures are implemented. The files that contain customizing procedures should contain the following line as the last statement:

```
DISPLAY 'Press function key f8 to display customizing menu'
```
You can re-display the menu or continue with other operations.

---

### 📋 Note

Do not type anything onto the screen while the customizing menu is displayed. `(CHR 4)` in the function key definitions will save a copy of the menu file and overwrite the old one.

---

# Customizing the Hatch Patterns

The `defaults.m` file contains the hatch patterns for the options iron, steel and copper on the HATCH screen menu. For a more detailed description of the `defaults.m` file, refer to the configuration manual that corresponds to your operating system.

The hatch patterns are produced with the following macros.
```
DEFINE I_hatch_iron
  HATCH_ANGLE 45
  HATCH_DIST 5
  CURRENT_HATCH PATTERN 0 1 0 CYAN SOLID CONFIRM
END_DEFINE

DEFINE I_hatch_steel
  HATCH_ANGLE 45
  HATCH_DIST 15
  CURRENT_HATCH PATTERN 0 1 0 CYAN SOLID (1/3 ) 1 0 CYAN SOLID
  CONFIRM
END_DEFINE

DEFINE I_hatch_copper
  HATCH_ANGLE 45
  HATCH_DIST 5
  CURRENT_HATCH PATTERN 0 1 0 CYAN 0.5 1 0 CYAN DASHED CONFIRM
END_DEFINE
```
It is very easy to alter these macros, or to write new macros of your own to produce other hatch patterns.

Your system administrator can edit the `\me10\defaults.m` file with any text editor. However, any changes made to the `\me10\defaults.m` file will be overwritten when you re-install or update your Creo Elements/Direct Drafting software. Here is a better method:

1. Start Creo Elements/Direct Drafting.

2. Use the `COPY` command in the **FILE** screen menu to copy the `\me10\defaults.m` file to the current directory.

3. Edit your file using the Creo Elements/Direct Drafting editor. Delete all the lines that do not need to be changed, and alter the remaining lines to match your current environment.

4. `INPUT` the file so that you can verify your changes.

5. If the results are satisfactory, include the following line in the `\me10\custom.m` file:
   ```
   INPUT 'filename'
   ```

   where `filename` is the full pathname and filename of your file.

   If necessary, edit the `\me10\startup` to remove the braces `{ }` from the line:
   ```
   {INPUT 'custom.m'}
   ```

   so that it reads:
   ```
   INPUT 'custom.m'
   ```

When you start Creo Elements/Direct Drafting, the `startup` file will input the contents of the `custom.m` file. This will in turn input your file.

For further information about hatching, refer to Design and Drafting with Creo Elements/Direct Drafting.

# The Keyboard Input Characters

```
CHAR 32  =   SPACE      CHAR 64  =   @       CHAR 96   =  `
CHAR 33  =   !          CHAR 65  =   A       CHAR 97   =  a
CHAR 34  =   "          CHAR 66  =   B       CHAR 98   =  b
CHAR 35  =   #          CHAR 67  =   C       CHAR 99   =  c
CHAR 36  =   $          CHAR 68  =   D       CHAR 100  =  d
CHAR 37  =   %          CHAR 69  =   E       CHAR 101  =  e
CHAR 38  =   &          CHAR 70  =   F       CHAR 102  =  f
CHAR 39  =   '          CHAR 71  =   G       CHAR 103  =  g
CHAR 40  =   (          CHAR 72  =   H       CHAR 104  =  h
CHAR 41  =   )          CHAR 73  =   I       CHAR 105  =  i
CHAR 42  =   *          CHAR 74  =   J       CHAR 106  =  j
CHAR 43  =   +          CHAR 75  =   K       CHAR 107  =  k
CHAR 44  =   ,          CHAR 76  =   L       CHAR 108  =  l
CHAR 45  =   -          CHAR 77  =   M       CHAR 109  =  m
CHAR 46  =   .          CHAR 78  =   N       CHAR 110  =  n
CHAR 47  =   /          CHAR 79  =   O       CHAR 111  =  o
CHAR 48  =   0          CHAR 80  =   P       CHAR 112  =  p
CHAR 49  =   1          CHAR 81  =   Q       CHAR 113  =  q
CHAR 50  =   2          CHAR 82  =   R       CHAR 114  =  r
CHAR 51  =   3          CHAR 83  =   S       CHAR 115  =  s
CHAR 52  =   4          CHAR 84  =   T       CHAR 116  =  t
CHAR 53  =   5          CHAR 85  =   U       CHAR 117  =  u
CHAR 54  =   6          CHAR 86  =   V       CHAR 118  =  v
CHAR 55  =   7          CHAR 87  =   W       CHAR 119  =  w
CHAR 56  =   8          CHAR 88  =   X       CHAR 120  =  x
CHAR 57  =   9          CHAR 89  =   Y       CHAR 121  =  y
CHAR 58  =   :          CHAR 90  =   Z       CHAR 122  =  z
CHAR 59  =   ;          CHAR 91  =   [       CHAR 123  =  {
CHAR 60  =   <          CHAR 92  =   \       CHAR 124  =  |
CHAR 61  =   =          CHAR 93  =   ]       CHAR 125  =  }
CHAR 62  =   >          CHAR 94  =   ^       CHAR 126  =  ~
CHAR 63  =   ?          CHAR 95  =   _
```

# The Keyboard Input Characters

```
CHAR  160  =  SPACE       CHAR  192  =  â         CHAR  224  =  Á
CHAR  161  =  À           CHAR  193  =  ê         CHAR  225  =  Ã
CHAR  162  =  Â           CHAR  194  =  ô         CHAR  226  =  ã
CHAR  163  =  È           CHAR  195  =  û         CHAR  227  =  Ð
CHAR  164  =  Ê           CHAR  196  =  á         CHAR  228  =  đ
CHAR  165  =  Ë           CHAR  197  =  é         CHAR  229  =  Í
CHAR  166  =  Î           CHAR  198  =  ó         CHAR  230  =  Ì
CHAR  167  =  Ï           CHAR  199  =  ú         CHAR  231  =  Ó
CHAR  168  =  ´           CHAR  200  =  à         CHAR  232  =  Ò
CHAR  169  =  `           CHAR  201  =  è         CHAR  233  =  Õ
CHAR  170  =  ^           CHAR  202  =  ò         CHAR  234  =  õ
CHAR  171  =  ¨           CHAR  203  =  ù         CHAR  235  =  Š
CHAR  172  =  ~           CHAR  204  =  ä         CHAR  236  =  š
CHAR  173  =  Ù           CHAR  205  =  ë         CHAR  237  =  Ú
CHAR  174  =  Û           CHAR  206  =  ö         CHAR  238  =  Ÿ
CHAR  175  =  £           CHAR  207  =  ü         CHAR  239  =  ÿ
CHAR  176  =  ‾           CHAR  208  =  Å         CHAR  240  =  Þ
CHAR  177  =  ⁋           CHAR  209  =  î         CHAR  241  =  þ
CHAR  178  =  ₤           CHAR  210  =  Ø         CHAR  242  =  °
CHAR  179  =  °           CHAR  211  =  Æ         CHAR  243  =  ø
CHAR  180  =  Ç           CHAR  212  =  å         CHAR  244  =  ŧ
CHAR  181  =  ç           CHAR  213  =  í         CHAR  245  =  ƀ
CHAR  182  =  Ñ           CHAR  214  =  ø         CHAR  246  =  -
CHAR  183  =  ñ           CHAR  215  =  œ         CHAR  247  =  ɨ
CHAR  184  =  ¡           CHAR  216  =  Ä         CHAR  248  =  ɨ
CHAR  185  =  ¿           CHAR  217  =  ì         CHAR  249  =  ɑ
CHAR  186  =  ¤           CHAR  218  =  Ö         CHAR  250  =  ǫ
CHAR  187  =  £           CHAR  219  =  Ü         CHAR  251  =  «
CHAR  188  =  ¥           CHAR  220  =  É         CHAR  252  =  □
CHAR  189  =  §           CHAR  221  =  ï         CHAR  253  =  »
CHAR  190  =  ƒ           CHAR  222  =  ß         CHAR  254  =  ±
CHAR  191  =  ¢           CHAR  223  =  Ô         CHAR  255  =  ⊠
```

# 13

# Brief Description of Commands and Functions

This chapter gives a very brief description of each command and function, arranged in alphanumeric sequence. For more information on any command or function refer to the Creo Elements/Direct Drafting help system. For example, enter:

```
help catch
```

The screen will clear and Creo Elements/Direct Drafting will display further information about the CATCH function.

`ABS` (arithmetic function)

For a number argument: returns the absolute value of the argument.For a vector argument: returns the length of the vector from the origin to the argument point.

`ADD_CURRENT_INFO` (command)

Adds the specified text to the current info.

`ADD_DIM_POSTFIX` (command)

Adds a postfix to an existing dimension.

`ADD_DIM_PREFIX` (command)

Adds a prefix to an existing dimension.

`ADD_DIM_SUBFIX` (command)

Adds a subfix to an existing dimension.

`ADD_DIM_SUPERFIX` (command)

Adds a superfix to an existing dimension.

`ADD_DIM_TOLERANCE` (command)

Allows you to add a tolerance to the selected dimension.

`ADD_ELEM_INFO` (command)

Adds the specified text to the info of the specified elements.

`ADU_ACCURACY` (function)

Specifys the accuracy when comparing two layouts.

`ADU_CHECK` (command)

Compares two layouts and attempts to re-annotate the new layout.

`ADU_CONFIRM_ANNOS` (command)

After an automatic update of annotations, the system marks every annotation as 'transffered' (default in green), 'regenerated' (default in red) or 'updated' (default in blue). With `ADU_CONFIRM_ANNOS` you can accept the proposals of the system.

`ADU_UPDATE_ANNOS` (command)

Allows you to exchange the elements, to which annotations refer. For example, you can exchange a reference element of a dimension, without deleting the dimension.

`ANALYZE_BSPLINE` (function)

Allows you to analyze specific elements and values that describe the selected b-spline.

`AND` (arithmetic function)

Returns 1 if both arguments are different from 0. Otherwise returns zero.

`ANG` (arithmetic function)

Returns the angle between the X-axis and the vector from the origin to the argument point.

`ARC` (command)

Creates an arc.

`ARCCOS` (arithmetic function)

Returns the principal value of the angle that has a cosine equal to the argument.

`ARCSIN` (arithmetic function)

Returns the principal value of the angle that has a sine equal to the argument.

`ARCTAN` (arithmetic function)

Returns the principal value of the angle that has a tangent equal to the argument.

`ARC_RESOLUTION` (function)

Specifies the precision of arcs on the screen (if display lists are used).

`AREA_PROPERTY` (command)

Calculates physical properties of selected areas.

`ARROW_CURSOR` (function)

Controls the cursor shape that you see in table and menu areas.

`ARROW_FILL` (function)

Controls the filling of arrows in dimensions and leader lines.

`ASSIST` (command)

Sets user assistance (Copilot).

`AUTO_NEW_SCREEN` (function)

Specifies whether `NEW_SCREEN` should operate after exposing the Creo Elements/Direct Drafting window.

`AUTO_STORE_TIME` (function)

Performs an autosave of the drawing. The frequency is user-defined.

`BEEP` (function)

Produces a beep on the loudspeaker.

`BLACK` (function)

Specifies the line color.

`BLUE` (function)

Specifies the line color.

`BSPLINE` (command)

Creates a B-spline curve.

`BSPL_ADD_C_PNT` (command)

Adds a control point between two neighboring control points.

`BSPL_ADD_I_PNT` (command)

Adds an interpolation point between two neighboring control points.

`BSPL_DEL_C_PNT` (command)

Deletes control points from a spline.

`BSPL_DEL_I_PNT` (command)

Deletes interpolation points from a spline.

`BSPL_DEL_TANGENT` (command)

Lets you delete a slope.

`BSPL_MOVE_C_PNT` (command)

Lets you modify the position of a control point.

`BSPL_MOVE_I_PNT` (command)

Lets you modify the position of an interpolation point.

`BSPL_MOVE_PNT` (command)

Lets you modify the position of a B-spline.

`BSPL_POINT_LENGTH` (command)

Allows you to subdivide a bspline in pieces of same length.

`BSPL_POLYGON_FEEDBACK` (function)

Used to determine your feedback type while creating bsplines.

`CANCEL` (command)

Cancels current system activity. Returns to Enter command.

`CANCEL_EDIT_DIM_TEXT` (command)

Cancels the effects of `EDIT_DIM_TEXT`. Restores an edited dimension to its original state.

`CATALOG` (function)

Outputs a listing of the named directory to the specified destination.

`CATALOG_LAYOUT` (command)

Specifies the layout of the directory information given by CATALOG.

`CATCH` (function)

Allows the user to select a point on the screen without having to put the cursor directly on that point.

`CENTER` (command)

Centers menu text.

`CENTERLINE` (command)

Allows you to create a centerline for a circular element. The centerline becomes part of the circular element and is moved or deleted if the associated circular element is moved or deleted.

`CENTER_DASH_DASH` (function)

Defines the linetype.

`CHAMFER` (command)

Creates a chamfer.

`CHANGE_COLOR` (command)

Changes the color of the selected elements.

`CHANGE_CURRENT_INFO` (function)

Global search and replace on current info text.

`CHANGE_DIM_ARROW` (command)

Changes the current dimension line termination.

`CHANGE_DIM_COLOR` (command)

For the selected dimensions, changes the color of the extension and dimension lines.

`CHANGE_DIM_FORMAT` (command)

Allows you to change the format of an existing dimension.

`CHANGE_DIM_FRAME` (command)

Changes the type of the selected frame for dimension text.

`CHANGE_DIM_LINEWIDTH` (command)

Changes the pensize of the extension and dimension lines for selected dimensions. (Same action as `CHANGE_DIM_PENSIZE`.)

`CHANGE_DIM_PENSIZE` (command)

Changes the pensize of the extension and dimension lines for selected dimensions.

`CHANGE_DIM_TEXTS` (command)

Allows you to change the attributes for existing dimension texts.

`CHANGE_DIM_TEXT_COLOR` (command)

Defines the dimension text color.

`CHANGE_DIM_TEXT_LOCATION` (command)

Changes the location of dimension text: above, on, or below the line.

`CHANGE_DIM_TEXT_ORIENTATION` (command)

Changes the orientation of the diimension text.

`CHANGE_DIM_VERTEX` (command)

Moves the dimension extension line to another vertex point.

`CHANGE_ELEM_INFO` (command)

Global search and replace on specified info text.

`CHANGE_FILLET` (command)

Changes the radius of the selected fillets.

`CHANGE_GLOBAL_INFO` (function)

Global search and replace on info text for every element in memory.

`CHANGE_HATCH_ANGLE` (command)

Changes the angle of the selected hatch.

CHANGE_HATCH_COLOR (command)

Changes the color of the selected hatch.

CHANGE_HATCH_DIST (command)

Changes the separation between hatch lines.

CHANGE_HATCH_LINETYPE (command)

Changes the linetype of the selected hatch.

CHANGE_HATCH_PATTERN (command)

Changes the pattern of the selected hatch.

CHANGE_HATCH_REF_PT (command)

Changes the reference point of the selected hatch.

CHANGE_LEADER_ARROW (command)

Changes the terminators of the selected leader lines.

CHANGE_LEADER_ARROW_SIZE (command)

Changes the size of the selected leader line terminators.

CHANGE_LINETYPE (command)

Changes the linetype of the selected elements.

CHANGE_LINEWIDTH (command)

Changes the pensize of selected elements, which may be real geometry not construction geometry or POINT. (Same action as CHANGE_DIM_PENSIZE.)

CHANGE_LINESIZE (command)

Changes the linesize of selected elements, which may be real geometry not construction geometry or POINT.

CHANGE_DIM_PENSIZE (command)

Changes the pensize of the extension and dimension lines for selected dimensions.

CHANGE_PART_REF_PT (function)

Changes the reference point of the active part.

CHANGE_TABLE_SIZE (function)

Changes the size of existing tables that are not secured against size change.

CHANGE_TEXT (command)

Changes one or more existing texts to new text.

CHANGE_TEXT_ADJUST (command)

Changes the adjust parameter of the selected text.

CHANGE_TEXT_ANGLE (command)

Changes the angle of the selected text.

CHANGE_TEXT_FILL (command)

Toggles filling of the selected text to off/on.

CHANGE_TEXT_FONTNAME (command)

Changes the font of the selected text.

CHANGE_TEXT_FRAME (command)

Changes the frame of the selected text.

CHANGE_TEXT_LINESPACE (command)

Changes the inter-line spacing of multi-line text.

CHANGE_TEXT_RATIO (command)

For the selected text, changes the ratio between character width and height.

CHANGE_TEXT_SIZE (command)

Changes the character size of the selected text.

CHANGE_TEXT_SLANT (command)

Changes the slant of the selected text.

CHANGE_VIEWPORT_COLOR (function)

Changes the background color of the current viewport.

CHANGE_VIEWPORT_SIZE (function)

Changes the size of the current viewport.

CHAR_LAYOUT (command)

Defines the characters of the current font.

CHECK_3D_GEO_MODIFY (function)

Specifies whether or not to issue a warning when modifying a Creo Elements/ Direct Modeling layout (ADU).

CHECK_BREAK (arithmetic function)

Returns 1 if BREAK key was pressed and IGNORE_BREAK was active.

CHECK_DIM_DETAIL (function)

Enables/disables the built-in dimension checking mechanism.

CHECK_ERROR (arithmetic function)

Returns 1 if one or more errors were trapped since the last TRAP_ERROR function call, otherwise returns 0.

CHECK_FONT_FILLABLE (function)

Characters of the current font are written to the selected output specification which cannot be filled.

CHECK_WINDOW (function)

Enables/disables the built-in window check mechanism which (by default) rejects extremely large or small window settings.

CHG_PIXEL_COLOR (command)

Changes the color of pixels to the color specified in SET_COLOR.

CHR (arithmetic function)

Converts a decimal number to the equivalent ASCII character. For example, CHR(35) is '#'.

CIRCLE (command)

Creates a circle.

CL_ABS_OFFSET (command)

Sets the centerline offset.

CL_COLOR (command)

Sets the centerline color.

CL_LINETYPE (command)

Sets the centerline linetype.

CL_LINEWIDTH (command)

Sets the centerline pensize. (Same action as CL_PENSIZE.)

CL_PENSIZE (command)

Sets the centerline pensize.

CL_REL_OFFSET (command)

Sets the centerline offset (relative to radius dimension).

CLEAN_DRAWING (command)

Cleans the drawing of duplicated geometry (for example, overlaps).

CLIPBOARD_SIZE

Sets the Windows Clipboard size (plot area) for subsequent plots to the Clipboard.

CLOSE_FILE (function)

Closes the specified file.

CMD_BG_COLOR (function)

Changes the background color of the command line.

`CMD_TXT_COLOR` (function)

Changes the color of the command-line text.

`COLOR` (function)

Specifies the current geometry and text color.

`COLOR_LTAB` (function)

Changes the color of the indicated position in the title and the data area of a logical table.

`CONFIGURE_EDITOR` (function)

Configures the built-in screen editor.

`CONNECT_TABLE` (function)

Connects a display table to a logical table.

`CONTOUR` (command)

Trims all selected elements with other selected elements to create a closed contour.

`CONTROLZ_IS_EOF` (function)

Specifies whether a `Ctrl-Z` character should be interpreted as an end-of-file character or a normal data byte.

`CONVERT_C_TO_B_SPLINE` (command)

Converts a selected spline of the old spline type ("Cspline") to a spline of the new spline type ("Bspline").

`CONVERT_DIM_TOLERANCE` (command)

Changes an existing tolerance to a different type.

`CONVERT_DIM_UNIT` (command)

Changes the current units for linear and angular dimensions.

`CONVERT_SPLINE` (command)

Converts the selected spline to a series of arcs and lines.

`COPY_FILE` (function)

Copies the specified source files to the destination.

`COS` (arithmetic function)

Returns the cosine of the argument.

`CREATE_DETAIL` (command)

Creates a detail view of existing geometry, magnified by the specified factor.

`CREATE_DIRECTORY` (function)

Creates a new directory.

CREATE_LTAB (function)

Creates a new logical table. If the table already exists, no error is given and the command is ignored.

CREATE_POLY (command)

Allows you to create a polyline from existing elements. You can then treat the polyline as a single element and modify it accordingly.

CREATE_SUBPART (command)

Creates a new part within the active part, using existing elements.

CREATE_VIEWPORT (function)

Creates a new viewport.

CS_AXIS (function)

Redefines the axes of the input coordinate system, leaving the origin fixed.

CS_MIRROR (function)

The specified axis of the input coordinate system is reflected about the other axis.

CS_REF_PT (function)

Changes the origin of the input reference system.

CS_ROTATE (function)

Rotates the input coordinate system about its origin.

CS_SET (function)

For the input coordinate system, changes the origin and the angle of the axes.

CURRENT_DIM_TEXTS (function)

Changes the current attributes for all dimension texts.

CURRENT_DIM_UNITS (function)

Sets the current units for linear and angular dimensions.

CURRENT_DIRECTORY (function)

Sets the current directory to the specified directory.

CURRENT_FONT (function)

Selects the font used for new texts.

CURRENT_HATCH_PATTERN (function)

Defines the current hatch pattern used for newly created hatches.

CURRENT_MENU (function)

Specifys the name of the current screen menu.

CURRENT_SPOTLIGHT_ATTR (function)

Allows you to change the default color and linetype that are used with SPOTLIGHTON.

CURRENT_VERTEX_COLOR (function)

Defines the default color of vertices used in SHOW VERTEX ON and

CURRENT_VIEWPORT (function)

Sets the current viewport.

CURSOR (function)

Selects the small or large cursor.

CURSOR_COORDINATES (function)

Allows you to display the current coordinates of the cursor above the input area.

CUT_MIDDLE

Cuts and deletes a piece out of the middle of an element at the intersection points with two other elements.

CYAN (function)

Switches the default color to cyan.

C_CIRCLE (command)

Creates a construction circle.

C_COLOR (function)

Sets the current color for construction geometry.

C_LINE (command)

Creates construction lines.

C_LINETYPE (function)

Specifies the current linetype for construction geometry, that is, c_circles and c_lines.

DASHED (function)

Specifies the linetype.

DASH_CENTER (function)

Specifies the linetype.

DATE (arithmetic function)

Returns date and time. For example, '26-Oct-99 14:26:58'.

DA_DB_ADD (command)

Adds a new part to the dimension database.

`DA_DB_DELETE` (command)

Deletes a part/entry from the dimension database.

`DA_DB_EXPORT` (command)

Extracts an entry from the dimension database and copies it into your current drawing. The new entry becomes a subpart of the top part.

`DA_DB_FILL_TABLE` (command)

Clears and refills the dimension database table. Should not be used on the command line.

`DA_DB_INQ` (function)

Returns information about the dimensioning database. This function supports the Dimensioning Acceleration Module screen interface. It should generally not be used from the command line.

`DA_DB_LOAD` (command)

Loads a dimensioning database file into memory.

`DA_DB_MATCH` (command)

Transfers dimensioning from a dimension database entry to drawing elements.

`DA_DB_STORE` (command)

Stores the dimension database in MI format in a file.

`DA_DB_UNLOAD` (command)

Unloads the current dimensioning database from system memory.

`DA_DB_WIN_CREATE` (function)

Creates a window to display the dimensioning database. Should not be used from the command line.

`DA_DB_WIN_LOC` (function)

Sets the location for the dimension database window. Should not be used from the command line.

`DA_DIM_ANGLE` (command)

Creates one or more angle dimensions.

`DA_DIM_ARC` (command)

Creates one or more arc dimensions.

`DA_DIM_AUTO_LOC` (function)

Sets the placement parameters for automatically positioned dimensioning.

`DA_DIM_AUTO_STRATEGY` (function)

Sets the level of intersection checking that is done when automatically positioning dimensions.

`DA_DIM_CHAIN` (command)

Creates one or more chain dimensions.

`DA_DIM_CHAMFER` (command)

Creates one or more chamfer dimensions.

`DA_DIM_COORD` (command)

Creates one or more instances fo coordinate dimensioning.

`DA_DIM_DATUM_LONG` (command)

Creates one or more instances of datum dimensioning with long base lines.

`DA_DIM_DATUM_LONG_SYM` (command)

Creates one or more instances of a symmetric dimensioning with long baselines.

`DA_DIM_DATUM_SHORT` (command)

Creates one or more instances of datum dimensioning with short base lines.

`DA_DIM_DELETE` (command)

Deletes one or more segments from a datum dimension stack. Also deletes entire dimensions.

`DA_DIM_DIAMETER` (command)

Creates one or more diameter dimensions.

`DA_DIM_GEO_SENSE` (command)

Creates dimensioning for selected elements based on their geometry type.

`DA_DIM_HOLE_INSERTION` (function)

Controls behavior if new dimensions are located inside a hatched area.

`DA_DIM_INCLINE` (command)

Allows dimensions with horizontal or vertical attribute to be inclined.

`DA_DIM_INSERT` (command)

Inserts one or more new segments into a datum dimension stack.

`DA_DIM_LINE` (command)

Creates one or more single-line dimensions.

`DA_DIM_LINE_SYM` (command)

Creates one or more symmetric single-line dimensions.

`DA_DIM_PD_SCAN` (command)

Creates dimensions for selected elements based on Parametric Design constraints.

`DA_DIM_RADIUS` (command)

Creates one or more radius dimensions.

`DA_DIM_SHORT_SPACE` (function)

Sets the spacing between items in a datum dimension stack.

`DA_FILTER_ACTIVATE` (function)

Enables selection filtering for dimensioning assignments.

`DA_FILTER_ADD` (function)

Adds a new selection criteria to the current dimensioning selection filter. This function is best called from the dimensioning acceleration module's screen interface.

`DA_FILTER_CLEAR_GEOTYPES` (function)

Clears all geometry-type criteria from the current dimensioning selection filter. This function is best called from the dimensioning acceleration module's screen interface.

`DA_FILTER_CLEAR_LINETYPES` (function)

Clears all linetype criteria from the current dimensioning selection filter. This function is best called from the dimensioning acceleration module's screen interface.

`DA_FILTER_DEL_COLOR` (function)

Clears color criteria from the current dimensioning selection filter.

`DA_FILTER_DEL_ORIENT` (function)

Clears line orientation criteria from the current dimensioning selection filter.

`DA_FILTER_DEL_WIDTH` (function)

Clears width criteria from the current dimensioning selection filter.

`DA_FILTER_DEL_LINESIZE` (function)

Clears linesize criteria from the current dimensioning selection filter.

`DA_FILTER_DEL_PENSIZE` (function)

Clears pensize criteria from the current dimensioning selection filter. (Same action as `DA_FILTER_DEL_WIDTH`.)

`DA_FILTER_INQ` (function)

Returns information about the dimensioning selection filter. This function supports the Dimensioning Acceleration Module screen interface. It should generally not be used from the command line.

`DA_FILTER_REFRESH_LINEWIDTH` (function)

Updates the contents of the pensize selection filter table. Should not be used from the command line. (Same Action as `DA_FILTER_REFRESH_PENSIZE`.)

`DA_FILTER_REFRESH_LINEWIDTH` (function)

Updates the contents of the pensize selection filter table. Should not be used from the command line. (Same Action as `DA_FILTER_REFRESH_PENSIZE`.)

`DA_FILTER_REFRESH_LINESIZE` (function)

Updates the contents of the linesize selection filter table. Should not be used from the command line.

`DA_FILTER_REFRESH_PENSIZE` (function)

Updates the contents of the pensize selection filter table. Should not be used from the command line.

`DA_FILTER_REFRESH_ORIENT` (function)

Updates the contents of the orientation selection filter table. Should not be used from the command line.

`DA_FILTER_SET_NAME` (function)

Sets an identifying name for the current dimensioning selection filter.

`DA_FILTER_STORE` (function)

Saves the current dimensioning selection filter to a file.

`DA_LINESIZE` (function)

Sets the linesize of the dimension.

`DA_PENSIZE` (function)

Sets the pensize of the dimension.

`DA_MOVE_DIMENSION` (command)

Moves one or more dimensions.

`DA_NULL` (function)

Support function for the Dimensioning Acceleration Module screen interface. Does nothing.

`DA_STYLE_APPLY` (function)

Applies the current style parameters to the selected dimensions.

`DA_STYLE_DEFER_UPDATE` (function)

Prevents updates of the style display viewport.

`DA_STYLE_ENABLE_UPDATE` (function)

Enables updates of the style display viewport.

`DA_STYLE_GET` (function)

Makes the dimensioning style parameters of a selected dimension the current dimensioning style.

DA_STYLE_INQ (function)

Returns information about the current dimensioning style. This function supports the Dimensioning Acceleration Module screen interface. It should generally not be used from the command line. Should not be used from the command line.

DA_STYLE_TYPE (function)

Selects a new dimensioning style.

DA_STYLE_UPDATE (function)

Updates the style display viewport.

DA_STYLE_WIN_CREATE (function)

Creates a window to display the current dimension style.

DA_STYLE_WIN_LOC (function)

Sets the location for the style display viewport. Support function for the Dimensioning Acceleration Module screen interface. Should not be used from the command line.

DA_STYLE_WIN_RAISE (function)

Raises the style display viewport.

DA_WRITE_DIM_SETTINGS_MACRO (function)

Creates a macro file that contains all current dimensioning style settings.

DDE_ADD_TOPIC (function)

Adds a new topic string to the list of recognized DDE topics.

DDE_CLOSE (arithmetic function)

Closes a named DDE conversation.

DDE_ENABLE (function)

Enables Creo Elements/Direct Drafting to act as a DDE server and act on requests from DDE clients.

DDE_EXECUTE (arithmetic function)

Sends a command string to the application indicated by the given conversation handle.

DDE_INITIATE (arithmetic function)

Initiates a DDE conversation with a given application on a given topic. Returns a conversation handle for use in subsequent DDE commands.

DDE_REMOVE_TOPIC (function)

Removes a specified topic string from the list of recognized DDE topics.

`DDE_REQUEST` (arithmetic function)

Asks a remote DDE application for the string value of the indicated data item.

`DDE_SEND_ACK` (arithmetic function)

Sends a DDE acknowledgement message for a given conversation handle.

`DDE_WITHHOLD_ACK` (arithmetic function)

Prevents the sending of a DDE acknowledgement message following execution of an action text. The active DDE conversation handle is returned instead.

`DEFINE` (function)

Defines a macro having the specified name.

`DEFINE_CATALOG` (function)

Defines the information to be printed by the CATALOG function.

`DEFINE_FONT` (function)

Defines a new font.

`DEFINE_KEY` (function)

Defines the function of the specified function key. Such a key is called a "Hot" key, "Smart" key, or "Soft" key.

`DEFINE_MOUSE_KEY` (function)

Defines mouse keys 1, 2, and 3.

`DELETE` (command)

Deletes the selected elements.

`DELETE_CURRENT_INFO` (function)

Deletes the current info.

`DELETE_DIMENSION` (command)

Deletes dimensions.

`DELETE_DIM_POSTFIX` (command)

Allows you to delete an identified postfix.

`DELETE_DIM_PREFIX` (command)

Allows you to delete an identified dimension prefix.

`DELETE_DIM_SUBFIX` (command)

Deletes a dimension subfix.

`DELETE_DIM_SUPERFIX` (command)

Deletes a dimension superfix.

`DELETE_DIM_TOLERANCE` (command)

Allows you to delete the tolerance of the selected dimension.

`DELETE_ELEM_INFO` (command)

Deletes all info for the selected elements.

`DELETE_FONT` (function)

Deletes the definition of the specified font.

`DELETE_HATCH` (command)

Deletes the selected hatch.

`DELETE_LABEL` (command)

Deletes, from the active part, all label information generated by the

`DELETE_LTAB_ROW` (function)

Deletes the indicated row from the named user table.

`DELETE_MACRO` (function)

Deletes a specified macro, or all macros.

`DELETE_MENU` (function)

Deletes the screen menu definition.

`DELETE_TABLE` (command)

Delete an existing unsecured table.

`DELETE_VIEWPORT` (function)

Deletes the specified viewport.

`DIM_ANGLE` (command)

Creates an angle dimension.

`DIM_ARC` (command)

Creates an arc dimension.

`DIM_ARROW` (function)

Specifys the type of dimension arrow.

`DIM_BREAK_RESTORE` (function)

Controls the dimension and extension lines after modification.

`DIM_BROKEN` (function)

Specifies how the dimension lines are to be drawn if the dimension text does not fit inside the dimension extension lines.

`DIM_CATCH_LINES` (function)

Controls the dimension elements that can be picked.

`DIM_CATCH_RANGE` (function)

Sets the catch range of dimension text to the middle of the extension lines.

`DIM_CHAIN` (command)

Creates chain dimensioning.

`DIM_CHAMFER` (function)

Allows you to create a JIS chamfer dimension.

`DIM_COLOR` (function)

Sets the color for the extension and dimension lines.

`DIM_CONVERT_UNIT` (command)

Specifys dimensional units.

`DIM_COORD` (command)

Creates coordinate dimensioning.

`DIM_CURSOR_POSITION` (function)

Specifys the position of the cursor when locating dimensions.

`DIM_DATUM` (command)

Specifys dimension position.

`DIM_DATUM_LONG` (command)

Creates datum dimensioning with long base lines.

`DIM_DATUM_SHORT` (command)

Creates datum dimensioning with short base lines.

`DIM_DATUM_STEP` (function)

Specifies the space between the dimension lines of a datum dimension.

`DIM_DEC_PLACE` (function)

Specifies number of decimal places for dimensions.

`DIM_DEG_MIN_SEC` (function)

Sets degrees minutes and seconds for dimensions.

`DIM_DIAMETER` (command)

Creates a diameter dimension.

`DIM_DIAMETER_LINE` (function)

Sets diameter lines.

`DIM_EXTENSION_LENGTH` (function)

Sets the current length between the arrow and the end of the extension line.

`DIM_FONT` (function)

Specifies dimension fonts.

`DIM_FORMAT` (function)

Allows you to specify the units for line and angular dimensions.

`DIM_FRAME` (function)

Specifies whether dimension text should have a frame.

`DIM_FT_INCH_SIGN` (function)

Specifies feet/inch options.

`DIM_LINE` (command)

Creates a single line dimension.

`DIM_LINEWIDTH` (function)

Sets the pensize of the extension and the dimension lines. (Same action as DIM_PENSIZE.)

`DIM_PENSIZE` (function)

Sets the pensize of the extension and the dimension lines.

`DIM_LINES_COLOR` (function)

Specifies the color of dimension lines.

`DIM_MIN_SPACE` (function)

Sets the current minimum space between the geometry and the dimension lines.

`DIM_NUMBER_FORMAT` (function)

Specifies the number format for dimensions.

`DIM_OFFSET_LINE` (function)

Specifies the offset of the dimension extension line from the dimension line.

`DIM_OFFSET_POINT` (function)

Specifies the offset of the dimension extension line from the geometry.

`DIM_POSTFIX` (function)

Allows you to add a postfix string to a dimension text.

`DIM_PREFIX` (function)

Allows you to add a postfix string to a dimension text.

`DIM_RADIUS` (command)

Creates a radius dimension.

`DIM_RADIUS_LINE` (function)

Sets the dimension radius line ON/OFF.

`DIM_SCALE` (function)

Specifies the dimension scale.

`DIM_SELECT_BY_TEXTBOX` (command)

Controls how the dimension is selected when box-selection is used.

`DIM_STAGGER_RESTORE` (function)

Controls the dimension and extension lines after modification.

`DIM_SUBFIX` (function)

Allows you to add a subfix string to dimension text.

`DIM_SUPERFIX` (function)

Allows you to add a superfix string to dimension text.

`DIM_TEXT_COLOR` (function)

Specifies the color of dimension text.

`DIM_TEXT_FRAME_COLOR_MODE` (function)

Specifies the dependency of the dimension text frame color on either the dimension main text color or the dimension line color.

`DIM_TEXT_GAP` (function)

When a dimension line is broken to allow a dimension to be inserted, sets the current gap between dimension text and dimension lines.

`DIM_TEXT_HOLE` (command)

Makes holes in an existing hatch for the selected dimension text.

`DIM_TEXT_LOCATION` (function)

Specifies whether current text dimensions should be above, on, or below the dimension line. For vertical dimension lines, "below" means nearer the geometry.

`DIM_TEXT_ORIENTATION` (function)

Specifies the current orientation for dimension text.

`DIM_TEXT_RATIO` (function)

Specifies the dimension text ratio.

`DIM_TEXT_SIZE` (function)

Specifies the dimension text size.

`DIM_TEXT_SPACE` (function)

When dimension text is placed above or below a dimension line, specifies the space the text and line.

`DIM_TOLERANCE` (function)

Specifies the current tolerance type.

`DIM_UNDERLINE_EDITED` (command)

Puts an underline on a dimension that has been edited.

`DIM_UNITS` (function)

Specifies dimension units.

`DIM_UPDATE` (command)

Used to find, mark and process dimension components, whose appearance would change after recalculation.

`DISPLAY` (function)

Evaluates a token and displays the result on the command line. You must then press any key or digitize any point.

`DISPLAY_LIST` (function)

Enables/disables the use of display list for the selected viewports.

`DISPLAY_NO_WAIT` (function)

Evaluates a token and displays the result on the command line. No user action is required.

`DIV` (arithmetic function)

Integer division with any fractional part truncated.

`DOTTED` (function)

Switches the default linetype to dotted.

`DOT_CENTER` (function)

Switches the default linetype to dot_center.

`DOT_GRID` (function)

Turns the dot grid on or off in the specified viewports.

`DRAWING_SCALE` (command)

Scales the geometry, by the specified factor, with respect to the plotter paper.

`DRAW_CURR_PART_ON_TOP` (function)

Controls the redraw of the current part (on top or z-level).

`DUMP_SCREEN` (function)

Dumps the contents of the graphics to the specified file.

`DUMP_SCREEN_DEFAULTS` (function)

Sets the parameters for the `DUMP_SCREEN` function.

`DUMP_SCREEN_LANG` (function)

Allows you to select the printer language for the `DUMP_SCREEN` function (see `DUMP_SCREEN`).

`ECHO` (function)

Opens or closes the `ECHO` file.

`EDIT_CURRENT_INFO` (function)

Allows you to edit the current info, which is the info assigned to every new element.

`EDIT_DIM_POSTFIX` (command)

For the selected dimension text, allows you to edit the postfix.

`EDIT_DIM_PREFIX` (command)

For the selected dimension text, allows you to edit the prefix.

`EDIT_DIM_SUBFIX` (command)

Allows you to edit the subfix of the identified dimension text.

`EDIT_DIM_SUPERFIX` (command)

Allows you to edit the superfix of the identified dimension text.

`EDIT_DIM_TEXT` (command)

Allows you to edit the selected dimension text.

`EDIT_DIM_TOLERANCE` (command)

Allows you to edit the tolerance of the selected dimension text.

`EDIT_ELEM_INFO` (command)

Allows you to edit the info of the specified element.

`EDIT_ENVIRONMENT` (function)

Allows you to edit a list of commands that establish the current environment. The environment consists of units, dimension defaults, hatch pattern, and so on.

`EDIT_FILE` (function)

Allows you to edit the specified file with the built-in text editor.

`EDIT_MACRO` (function)

Allows you to edit the specified macro with the built-in text editor.

`EDIT_PART` (command)

Makes the specified part the active part.

`EDIT_PORT` (function)

Specifies the viewport used by the built-in screen editor.

`EDIT_TEXT` (command)

Allows you to edit the specified text.

`ELLIPSE` (macro)

Draws splines that approximate ellipses.

`ELSE` (pseudo-command)

Conditional operator.

`ELSE_IF` (pseudo-command)

Conditional operator.

`ENABLE_BREAK` (function)

Resets the break handling back to the default setting (macros are interrupted by pressing the BREAK key).

`END` (command)

Terminates the current command or function.

`END_DEFINE` (pseudo-command)

Indicates the end of a macro definition.

`END_IF` (pseudo-command)

Indicates the end of an "IF - statement".

`END_LOOP` (pseudo-command)

Indicates the end of a loop in a macro.

`END_PART` (command)

Same as `EDIT_PART PARENT`.

`ENTER` (function)

Evaluates a token. The result is displayed on the command line.

`EQUIDISTANCE` (command)

Creates an equidistant contour.

`ERROR_LOG` (function)

Saves warnings and error messages generated by the system in internal memory.

`ERROR_STR` (arithmetic function)

Displays the first error message issued by the system after error trapping was enabled by `TRAP_ERROR`.

EXIT (command)

Terminates the session. Control is returned to the host operating system. You must confirm with CONFIRM to avoid accidental exit.

EXIT_IF (pseudo-command)

Condition statement that indicates when a loop should be terminated.

EXOR (arithmetic function)

Exclusive OR. Returns 1 if exactly one argument (of two) is 0. Otherwise returns 1.

EXP (arithmetic function)

Returns e (2.718...) raised to the power of the argument.

FALSE (arithmetic function)

Returns 0.

FBROWSER (function)

File browser functionality.

FILLET (command)

Produces a fillet having the specified radius.

FOLLOW (function)

Causes the origin of the coordinate system to be at the most recently input point.

FONT_EDITOR (Command)

Creates and modifies Creo Elements/Direct Drafting fonts.

FRACT (arithmetic function)

Returns the fractional part of the argument.

GATHER (command)

Brings existing elements into the active part.

GET_ELEM_INFO (function)

Changes the current info to be the same as the info of the selected element.

GET_PID (arithmetic function)

Displays the process ID of the running program.

GET_PROPERTIES (function)

Changes the current properties to be the same as the properties of the selected element.

GET_TYPE (function)

Returns the type of token specified. It is an enhancement of the TYPE command.

`GREEN` (function)

Switches default color to green.

`GRID_FACTOR` (function)

Specifies the distance between grid points or ruler ticks.

`HATCH` (command)

Creates hatches.

`HATCH_ANGLE` (function)

Sets the current hatch angle.

`HATCH_COLOR` (function)

Sets the current hatch color.

`HATCH_DIST` (function)

Sets the current hatch distance.

`HATCH_LINETYPE` (function)

Sets the current hatch linetype.

`HATCH_REF_PT` (function)

Sets the current hatch reference point.

`HELP` (function)

Displays the section of the HELP file that describes the required keyword.

`HELP_PORT` (function)

Defines the viewport used by the HELP function.

`HIGHLIGHT_LTAB` (function)

Changes the highlight state of the indicated position in the data area of the named user table.

`HL_CHANGE_COLOR` (macro)

Changes the color for all hidden lines computed by the `HL_GENERATE_HIDDEN` command.

`HL_CHANGE_LTYPE` (macro)

Changes the linetype for all hidden lines computed by the `HL_GENERATE_ HIDDEN` command.

`HL_DEFAULT_FACE_COLOR` (function)

Specifies the default face color.

`HL_DELETE_FACE` (command)

Deletes covering faces. The faces selected must be in the current part.

`HL_GENERATE_FACE` (command)

Specifies covering faces. A covering face covers all geometry with a lower z-value.

`HL_GENERATE_HIDDEN` (command)

Causes transition to the hidden-line-generation mode.

`HL_GEN_ALL_PART` (command)

Specifies z-value settings and face generation for a whole part.

`HL_INQ_CURR_Z_VALUE` (function)

Returns the current z-value assigned to each newly created component by default.

`HL_INQ_FACE_COLOR` (function)

Returns the rgb-color-value of the specified face.

`HL_INQ_LOAD_OFFSET` (function)

Returns the value of the load offset in the z-direction.

`HL_INQ_LOAD_VALUE` (function)

Returns the mode in which the load value / offset was specified, and the value itself.

`HL_INQ_RELATION_OFFSET` (function)

Displays the current relation offset in the z-direction.

`HL_INQ_Z_VALUE` (function)

Returns the z-value of the specified element, or the minimum or maximum z-value of the entire assembly.

`HL_REDRAW_MODE` (function)

Toggles between the Creo Elements/Direct Drafting normal redraw mode and the hidden-line redraw mode.

`HL_SET_COLOR` (function)

Sets the color for the hidden lines.

`HL_SET_CURR_Z_VALUE` (function)

Defines a z-value that is assigned to each newly created component by default.

`HL_SET_FACE_COLOR` (command)

Sets or changes the background color of covering faces.

`HL_SET_KEEP_COLOR` (function)

Defines whether elements keep their color during the hidden line generation process.

`HL_SET_LINETYPE` (function)

Sets the linetype for the hidden lines.

`HL_SET_LOAD_VALUE` (function)

Defines a load offset in the z-direction.

`HL_SET_RELATION_OFFSET` (function)

Defines an offset in the z-direction that is used to compute a z-value specified as an ABOVE/BELOW/BETWEEN relation to another element.

`HL_SET_Z_VALUE` (command)

Assigns or changes z-values.

`HL_SHOW_HIDDEN` (macro)

Makes hidden lines visible or invisible.

`HL_VISUALIZE` (function)

Allows you to view all elements selectively on certain z-values or to show faces on certain levels in different colors.

`HSL_COLOR` (function)

Allows you to set a new color (hue,saturation,luminosity).

`ICONIFY_WINDOW` (function)

Iconifies the Creo Elements/Direct Drafting window.

`IF` (pseudo-command)

Boolean expression.

`IGNORE_BREAK` (function)

When enabled from a macro the BREAK key will have no effect.

`INIT_PART` (command)

Creates a new, empty part immediately below the active part. Makes this new part active.

`INIT_SUBPART` (command)

Creates a new, empty part immediately below the active part. Makes this new part active.

`INPUT` (function)

Temporarily redirects the input stream from the keyboard, or mouse to the specified text file. (When you use the `INPUT` command within a macro, it must be used with the qualifier `IMMEDIATE`, see also INPUT on page 38.)

`INQ` (arithmetic function)

Returns an element of a system array. These values can be set using

`INQ_ELEM` (function)

Writes information about the specified element to the system array. This information can be retrieved using `INQ`.

`INQ_ENV` (function)

Writes information about the system environment to the system array. This information can be retrieved using `INQ`.

`INQ_PART` (function)

Writes information about the magnification factor of a part to the system array. This information can be retrieved using `INQ`.

`INQ_SELECTED_ELEM` (function)

Writes information about the identified element into the system inquiry array. It can then be retrieved with `INQ` (see `INQ`).

`INQ_TABLE` (function)

Allows you to obtain information about the specified table.

`INSERT_LTAB_ROW` (function)

Inserts rows into the named user table.

`INT` (arithmetic function)

Returns the integer part of the argument. For example, INT(PI) is equal to 3.

`ISOMETRIC` (command)

Allows you to create an isometric view.

`KEEP_CORNER` (function)

With OFF, the lines or arcs between the corner and the fillet or chamfer are deleted. With ON, the corner remains a corner.

`KNOB_BOX_FACTOR` (function)

Specifies the sensitivity of the knobs.

`LABEL` (command)

Generates label information for the selected points, lines, arcs, and circles in the active part.

`LAST_POSTFIX` (function)

Allows you to use the previous postfix as the current postfix.

`LAST_PREFIX` (function)

Allows you to use the previous dimension prefix as the current dimension prefix.

`LAST_SUBFIX` (function)

Allows you to use the last subfix as current subfix.

`LAST_SUPERFIX` (function)

Allows you to use the last superfix as current superfix.

`LAST_TOLERANCE` (function)

Sets the previous tolerance to the current tolerance.

`LAST_WINDOW` (function)

Restores the previous window of the current viewport.

`LEADER_ARROW` (function)

Specifies the terminator used for new leader lines.

`LEADER_LINE` (command)

Creates a leader line.

`LEN` (arithmetic function)

For a string argument, returns the length of the string.For a vector argument, returns the length of the vector from the origin to the argument point.

`LET` (function)

Defines a macro or a variable.

`LG` (arithmetic function)

Returns the decimal logarithm (base 10) of the argument.

`LINE` (command)

Creates a line.

`LINEPATTERN` (function)

Specifies the current linetype for the components that are to be created with the current active command.

`LINESIZE` (function)

Specifies the current linesize used for all new real geometry except construction geometry and POINT.

`LINETYPE` (function)

Specifies the current linetype.

`LINEWIDTH` (function)

Specifies the current pensize used for all new real geometry except construction geometry and POINT. (Same action as `PENSIZE`.)

`LINE_GRID` (function)

Turns the line grid on or off.

`LIST_FONTS` (function)

Gives a list of all fonts defined and all fonts used. Also gives the name of the current font.

`LIST_GLOBAL_INFO` (function)

Lists all info used by all elements in memory.

`LIST_MACRO_NAMES` (function)

Outputs the names of all currently defined macros to the specified destination.

`LN` (arithmetic function)

Returns the natural logarithm (base e) of its argument.

`LOAD` (command)

Loads into memory a part from the specified file.

`LOAD_FONT` (command)

Loads into memory all text fonts sorted in the specified file.

`LOAD_MACRO` (function)

Loads all macros stored in the specified file into memory.

`LOAD_MODULE` (command)

Activates the specified application module.

`LOCAL` (pseudo-command)

Defines a local variable within a macro.

`LONG_DASHED` (function)

Switches the default linetype to long_dashed.

`LOOP` (pseudo-command)

Defines a loop that is repeated until the boolean expression in an EXIT_IF clause evaluates to be logical true.

`LOWER_WINDOW` (function)

Causes the Creo Elements/Direct Drafting window to be lowered beneath all other windows on the Windows® desktop.

`LTAB_COLUMNS` (arithmetic function)

Displays the number of columns in the named logical table.

`LTAB_ROWS` (arithmetic function)

Displays the number of rows in the named logical table.

`LTAB_TITLES` (arithmetic function)

Displays the number of title strings in the named logical table.

`LWC` (arithmetic function)

Lowercase. Converts uppercase characters to lowercase.

`MAGENTA` (function)

Sets the default line color to magenta.

`MAKE_TMP_NAME` (arithmetic function)

Returns a unique filename for temporary use.

`MATCH` (arithmetic function)

Returns 1 if the first string is matched by the pattern specified by the second string, otherwise returns 0.

`MAX_FEEDBACK` (function)

Specifies the amount of element tracking in `MODIFY` and `STRETCH` commands.

`MEASURE_ANGLE` (function)

Measures the angle between two specified elements.

`MEASURE_AREA` (function)

Measures the area enclosed by the specified circle, arc, spline, or fillet.

`MEASURE_COORDINATE` (function)

Measures the coordinates of the selected point.

`MEASURE_DISTANCE` (function)

Measures the distance between two specified points.

`MEASURE_LENGTH` (function)

Measures the length of the specified line, circle, arc, fillet, or spline.

`MEASURE_RADIUS` (function)

Measures the radius of the specified circle, arc, or fillet.

`MENU` (function)

Defines the appearance and function of the screen menu slots.

`MENU_LAYOUT` (function)

Defines the shape and location of a screen menu.

`MENU_STATUS` (function)

Defines the status of a screen menu.

`MERGE` (command)

Causes two elements to be merged into a single element.

`MIRR` (arithmetic function)

Returns the mirror image of a vector about a virtual mirror line.

`MOD` (arithmetic function)

Returns the remainder of a division.

`MODIFY` (command)

Modifies elements with the options MOVE, MIRROR.

`MODIFY_DIM_LINES` (command)

Puts breaks in existing dimension and extension lines.

`MOVE_DIMENSION` (command)

Moves the selected dimension.

`MOVE_TABLE` (function)

Moves an existing table to a specified screen location. Only those tables not secured against move can be relocated.

`NEW_SCREEN` (function)

Redraws the entire screen.

`NOT` (arithmetic function)

Returns 1 if the argument equals 0. Otherwise returns 0.

`NUM` (arithmetic function)

Returns the decimal equivalent of the first ASCII character in a string. For example, NUM('hullo') is `104`.

`ON_ERROR` (function)

The specified string is used as input at the occurrence of the next error.

`OPEN_INFILE` (function)

Opens the named file for reading, using the `READ_FILE` function.

`OPEN_OUTFILE` (function)

Opens the named file for writing, using the `WRITE_FILE` function.

`OR` (arithmetic function)

Returns 1 if both arguments are non-zero. Otherwise, returns 0.

`ORIGIN` (function)

In the specified viewport, the symbol for the the origin of the input coordinate system is turned off or on.

`OUTPUT_HP15` (function)

Switches the output mode for kanji text to HP15 code.

`OUTPUT_HP16` (function)

`OUTPUT_HP16` switches the output mode for kanji text to HP16 code.

`OUTPUT_STRING` (function)

Writes the contents of |string| to its output device.

`OVERDRAW` (command)

Used to overdraw construction geometry. This command replaces the contents of the overdraw macro.

`PARAMETER` (pseudo-command)

Used in macro definition to indicate macro parameters.

`PART_DRW_SCALE` (command)

Allows you to define a drawing scale for a specified part.

`PART_DRW_SCALE_REF` (function)

Allows you to specify the reference point, from where the part is scaled.

`PARTS_LIST` (function)

Shows the parts in the active part whose names do not begin with ".". The number of occurrences of each part is shown.

`PASSWORD` (function)

Used to enter the enabling password into the system.

`PENSIZE` (function)

Specifies the current pensize used for all new real geometry except construction geometry and POINT.

`PHANTOM` (function)

Sets the current hatch linetype to phantom.

`PI` (arithmetic function)

Returns a value for pi of 3.14159265358979.

`PICK_UTAB_ROW_BY_NAME` (function)

Support function for the Dimensioning Acceleration Module screen interface. Should not be used from the command line.

`PICK_VP_PNT` (function)

Emulate interactive user pick in a specific viewport, defined by number and viewportname.

`PICTURE_BROWSER` (function)

Displays the PICTURE BROWSER window.

`PICTURE_LIST` (function)

Displays a list of pixmaps loaded in the current session.

`PLOT` (command)

Plots a drawing.

`PLOTTER_TYPE` (function)

Specifies the type of plotter currently connected.

`PLOT_AUTO_ROTATE` (function)

Allows you to disable the auto-rotate feature that is part of the firmware on some raster plotters.

`PLOT_CENTER` (function)

Specifies whether the drawing should be centered in the plotter viewport.

`PLOT_DESTINATION` (function)

Specifies the destination of the plot.

`PLOT_FORMAT` (function)

Defines the maximum plotting area (hard limits).

`PLOT_IMAGE_QUALITY` (function)

Defines such things as the color palette, resolution, and scaling.

`PLOT_LINETYPE_LENGTH` (function)

Specifies the length of the pattern for each linetype.

`PLOT_PEN_TABLE` (function)

Controls the mapping of linetypes and colors to plotter pens and plotter linetypes.

`PLOT_SCALE` (function)

Specifies the scale factor for the drawing before being plotted.

`PLOT_STOP_ON_ERROR` (function)

Controls the plot behavior when a drawing does not fit in the plot area.

`PLOT_TRANSFORMATION` (function)

Defines a mapping for certain elements during plot.

`PLOT_VIEWPORT` (function)

Locates the plotter viewport within the maximum plotting area specified by `PLOT_FORMAT`.

`PNT_RA` (arithmetic function)

Given a length and an angle, returns a 2D point.

`PNT_XY` (arithmetic function)

Given an x and a y coordinate, returns a 2D point (vector).

`PNT_XYZ` (arithmetic function)

Given an x coordinate, a y coordinate, and a z coordinate, returns a 3D point (vector).

`POINT` (command)

Creates point elements.

`POLYELEM` (command)

Allows you to create a polyline from existing elements. You can then treat the polyline as a single element and modify it accordingly.

`POP_DOWN_LTAB` (function)

Causes the named logical table to "pop down".

`POP_UP_LTAB` (function)

Causes the named logical table to "pop up".

`POS` (arithmetic function)

Returns the first position of a substring within a string.

`PRE_VIEW` (command)

Lets you preview a drawing before loading it.

`PRINT_TABLE` (function)

Prints a display table in a file.

`PROMPT_LIST` (function)

Saves up to 500 system prompts in system memory.

`PURGE_FILE` (function)

Deletes the specified files from the disk.

`PUT_PROPERTIES` (command)

The properties of the selected elements are changed to the current properties.

`RAC_CHECK` (command)

Updates new layout data coming from Creo Elements/Direct Modeling with documentation data added to an older version of this layout.

`RAISE_WINDOW` (function)

Causes the Creo Elements/Direct Drafting window to rise to the top of all other windows on the Windows desktop.

`RC_ACCURACY` (function)

Specifies the accuracy when comparing two MI files.

`RC_CHECK` (command)

Compares two parts and all their subparts and stores the result of the comparison as info texts for the parts elements.

`READ` (function)

Accepts user input from the command line, and assigns the data to a variable.

`READ_FILE` (function)

Reads one line of text from the specified file and assigns the string to the specified variable.

`READ_LTAB` (arithmetic function)

Returns the value in the named logical table.

`RECALL_BUFFER` (function)

Saves up to 63 input lines. These lines can be recalled using the `[Prev]` and `[Next]` keys.

`RECALL_WINDOW` (function)

The window of the current viewport is changed to the window stored using `STORE_WINDOW`.

`RED` (function)

Sets the default line color to red.

`REDRAW` (function)

Redraws the contents of the current viewport.

`REDRAW_SCENE` (function)

Redraws a scene viewport.

`RENAME_ELEMENT` (command)

Renames or copies an element, including ALL revisions and versions of that element.

`RENAME_PART` (command)

Renames the active part.

`RENOVATE` (function)

Restores the contents of selected viewports.

`REPEAT` (pseudo-command)

Defines a loop which is repeated until the boolean expression in the `UNTIL` clause evaluates to be logical true.

`REQUEST_PRINT_SETUP` (function)

Sets whether or not the Windows Print Manager displays its own dialog whenever plots or screendumps are sent to the Print Manager.

`RESET_PART_NUMBER` (command)

Renumbers all "unique" part numbers beginning at the TOP part.

`RESTORE` (command)

Recovers archived elements and their related files from 'source' to the database.

`RGB_COLOR` (function)

Allows you to specify a new color (red,green,blue).

`RND` (arithmetic function)

Returns a pseudo-random Xnumber in the range $0 <= X < 1$.

`ROT` (arithmetic function)

Returns the point (vector) rotated about the origin by the given angle.

`ROTATE_DIM_TEXT` (command)

Rotates the selected dimension text by the specified angle.

`ROUND` (arithmetic function)

Returns the argument, rounded to the nearest integer. For example, ROUND (4.4999) equals `4`.

`RPT` (arithmetic function)

Returns the required number of copies of a string.

`RTL_COLOR` (command)

Sets reference-line color.

`RTL_DST_GAP` (command)

Sets reference-line gap (at destination).

`RTL_LINETYPE` (command)

Sets reference-line linetype.

`RTL_LINEWIDTH` (command)

Sets reference-line pensize. (Same action as `RTL_PENSIZE`.)

`RTL_PENSIZE` (command)

Sets text reference-line pensize.

`RTL_SRC_GAP` (command)

Sets reference-line gap (source).

`RULER` (function)

Turns the ruler on or off in the specified viewports.

`RUN` (function)

Returns to the operating system shell prompt.

`SAVE` (command)

The drawing in memory is saved to the specified file.

`SAVE_ENVIRONMENT` (function)

Outputs the environment to the specified destination.

`SAVE_FONT` (function)

Saves all fonts, or a specified font, to the specified file.

`SAVE_LTAB` (function)

Saves the named logical table to "output spec" (see help for `OUTPUT_SPEC` for details).

`SAVE_MACRO` (function)

Outputs the named macro, or all macros, to the specified destination.

`SAVE_MENU` (function)

Outputs the current screen menu definition to the specified destination.

`SAVE_TABLE` (function)

Saves a display table set up in a file.

`SAVE_VIEWPORT` (function)

Outputs the current viewport definitions to the specified destination.

`SCREEN_TRANSFORMATION` (function)

Defines a mapping for certain elements during redraw.

`SCROLL_LTAB` (function)

Scrolls display tables connected to the named logical table so the specified row is at the top of the display.

`SEARCH` (command)

Specifies the directories in the search list.

`SECURE_LTAB` (function)

Secures the user table named. A secured user table cannot be deleted

`SECURE_MACRO` (function)

Prevents macros from being listed, edited, saved, or traced.

`SECURE_TABLE` (function)

Secures a display table against deletion. After securing, the table cannot be deleted or redefined.

`SELECT_DIM_ARROW` (function)

Specifies the current dimension line terminator.

`SELECT_FROM_LTAB` (function)

Performs a select operation on the source table.

`SGN` (arithmetic function)

Returns -1 if the argument is negative, 0 if the argument is 0, and 1 if the argument is positive.

`SHARE_PART` (command)

Causes the specified part to become shared.

`SHOW` (function)

Selectively turns elements on or off. Shows elements in different colors, Shows only the outline as a box.

`SHOW_CPOLY` (function)

Shows the control polygon of B-splines.

`SHOW_PART` (function)

Changes the way parts are displayed in the current viewport.

`SHOW_TABLE` (function)

Displays or erases an existing table.

`SHOW_TABLE_PAGE` (function)

Displays a query-page that has been attached to the specified display table.

`SIN` (arithmetic function)

Returns the sine of the argument.

`SL_COLOR` (command)

Sets the symmetry-line color.

`SL_LINETYPE` (command)

Sets the symmetry-line linetype.

`SL_LINEWIDTH` (command)

Sets the symmetry-line pensize. (Same action as `SL_PENSIZE`.)

`SL_PENSIZE` (command)

Sets the symmetry-line pensize.

`SL_OFFSET` (command)

Sets the symmetry-line offset.

`SMASH_POLY` (command)

Allows you to smash a polygon into its single elements.

`SMASH_SUBPART` (command)

Brings all the elements of the selected part into the active part, and deletes the subpart. The selected part must be a subpart of the active part.

`SNID` (arithmetic function)

Returns the product number and serial number of an HP-HIL security device. If no security device is present, returns the product number and serial number of the computer.

`SOLID` (function)

Sets the default linetype to solid.

`SORT_LTAB` (function)

Sorts a user table based upon the columns specified.

`SPLINE` (command)

Creates a spline.

`SPLINE_CONVERSION` (function)

Converts splines created with an earlier version of the program into B-splines.

`SPLIT` (command)

Splits lines, circles, arcs, fillets, and splines.

`SPLITTING` (function)

Automatic splitting and merging is turned on or off.

`SPOTLIGHT` (function)

The active part remains the same. All other geometry is redrawn in magenta color and with dashed lines.

`SQR` (arithmetic function)

Returns the square of the argument.

`SQRT` (arithmetic function)

Returns the square root of the argument.

`STATLINE_RESET` (function)

Re-activates the status line if it has been automatically disabled.

`STORE` (command)

Stores the drawing to the specified file. (Uses MI format 2.30.)

`STORE_202` (command)

Stores the drawing with MI format 2.02.

`STORE_210` (command)

Stores the drawing with MI format 2.10.

`STORE_211` (command)

Stores the drawing with MI format 2.11.

`STORE_221` (command)

Stores the drawing with MI format 2.21.

`STORE_230` (command)

Stores the drawing with MI format 2.30.

`STORE_231` (command)

Stores the drawing with MI format 2.31.

`STORE_FONT` (function)

Stores all fonts, or the specified font, to the named file.

`STORE_IN_RECALL_BUFFER` (function)

Stores the given string into the recall buffer for further use.

`STORE_MACRO` (function)

Stores the named macro, or all macros, to the specified destination.

`STORE_WINDOW` (function)

Stores the window coordinates of the current viewport.

`STR` (arithmetic function)

Returns the ASCII representation of the argument. For example,

`STRETCH` (command)

Stretches lines, circles, arcs, splines, and leader lines.

`STRUCTURE` (function)

Query to display hierarchical structures.

`SUBSTR` (arithmetic function)

Returns a substring of a string.

`SYMBOL_PART` (command)

Causes the selected part to become a symbol.

`SYMLINE` (command)

Creates a symmetry line.

`TABLE_COLUMN` (function)

Sets the cell attributes in the data columns of an existing unsecured table.

`TABLE_LAYOUT` (command)

Allows you to create a new table of a specified shape at a specified location.

`TABLE_SCROLL_STEP` (function)

Sets the scroll step of a display table while using the scroll bar.

`TABLE_TITLE` (function)

Sets the attributes of title slots for an existing table that is not secured against changes in title.

`TAN` (arithmetic function)

Returns the tangent of the argument.

`TEXT` (command)

Creates texts.

`TEXT_ADJUST` (function)

Specifies the position of the text origin for new texts.

`TEXT_ANGLE` (function)

Specifies the current text angle.

`TEXT_FILL` (function)

Turns the filling of characters on or off.

`TEXT_HOLE_INSERTION` (function)

Inserts a text window in a hatched area.

`TEXT_FRAME` (function)

Sets the type of the current text frame.

`TEXT_LINESPACE` (function)

Sets the current linespacing for new texts.

`TEXT_RATIO` (function)

Sets the current ratio of character width to height.

`TEXT_SIZE` (function)

Sets the current text height.

`TEXT_SLANT` (function)

Sets the current text slant.

`TEXT_TO_GEO` (command)

Converts text to geometry.

`TIME` (arithmetic function)

Returns the number of seconds since midnight.

TONE (function)

Generates an audible tone having specified frequency, duration, and amplitude.

TRACE (function)

Opens or closes the TRACE file.

TRAP_ERROR (function)

Defines the behavior in case of an error. Without TRAP_ERROR all execution will be stopped when an error occurs. After TRAP_ERROR is enabled the fact that an error has happened is just noted, but execution does not stop.

TRIM_ONE (command)

Allows you to trim or extend an element to intersect precisely with another element.

TRIM_TWO (command)

Allows you to trim or extend two elements to intersect with each other.

TRIM (arithmetic function)

Returns a string formed by stripping all leading and trailing blanks from the argument.

TRIMMING (command)

Automatic Trimming (removal of non-visible parts of splines after splitting) is switched ON/OFF.

TRUE (arithmetic function)

Returns 1.

TRUE_COLOR_PLOTTING (command)

Enables true color plotting and turns off plot transformation.

TRUNC (arithmetic function)

Returns the integer portion of a real number.

TYPE (arithmetic function)

Returns the type of the specified token.

TXT_WINDOW (command)

Specifies a text window in a hatched area.

UA_ANGLE_GRID (function)

Sets the angle increment used by COPILOT.

UA_CENTER_CATCH_RANGE (function)

Specifies the portion of each line's length that COPILOT will treat as the center for catching.

`UA_DISTANCE_GRID` (function)

Sets the length increment used by COPILOT.

`UA_GET_DESIGN_INTENT` (arithmetic function)

Responds with the on/off status of Design Intent.

`UA_PERPENDICULAR_CATCH_RANGE` (function)

Specifies the portion of lines, circles, and arcs that COPILOT will treat as perpendicular for catching.

`UA_TANGENT_CATCH_RANGE` (function)

Specifies the portion of circles and arcs that COPILOT will treat as tangent for catching.

`UA_SET_CATCH_DELAY` (function)

Sets the amount of time that the cursor must be motionless before COPILOT catch information is displayed.

`UNITS` (function)

Specifies the current units for distances and angle.

`UNLOAD_MODULE` (command)

Unloads a module.

`UNSHARE_PART` (command)

Causes a shared part to be unshared.

`UNTIL` (pseudo-command)

Boolean expression

`UPC` (arithmetic function)

Converts lowercase characters to uppercase.

`UPDATE_SCREEN` (function)

All device or driver buffers are flushed to the screen. The status line and menu are updated.

`USE_MULTILINE_HATCH` (function)

Allows you to select two different hatching processes for hatches with a distance of 0.

`VAL` (arithmetic function)

Converts a numeric string to a number.

`VERSION` (function)

Displays information on the CAD software version.

`VIEW` (function)

Causes the selected part to be viewed in the current viewport.

`WAIT` (function)

Causes the system to do nothing for the number of seconds specified.

`WHILE` (function)

The next code section is executed as long as the boolean expression following the WHILE statement is true.

`WHITE` (function)

Sets the default line color to white.

`WINDOW` (function)

Allows you to control what part of the drawing is shown in the current viewport.

`WINEXEC` (arithmetic function)

Launches the Windows application named in the given command string.

`WRITE_FILE` (function)

Writes a line of text to a specified file.

`WRITE_LTAB` (function)

Writes new values to the named user table.

`X_OF` (arithmetic function)

Returns the X coordinate of a vector.

`YELLOW` (function)

`Y_OF` (arithmetic function)

Returns the Y coordinate of a vector.

`Z_OF` (arithmetic function)

Returns the Z coordinate of a vector.

# A

# Logical and Display Tables

This appendix provides you with the information needed to use logical and display tables.

It describes the structures of a logical table and a display table, and explains the concept of connecting a display table to a logical table.

Then it describes the functions to access logical tables, and the commands and functions you need to define and use user tables.

Also, it describes the commands and functions you need to define and use display tables.

Finally, there are two examples to show you a typical procedure for defining a logical table and a display table, mapping a display table to a logical table, and using these tables.

---

### 🗒 Note

The description and examples in this appendix assume that you are already familiar with macro programming in Creo Elements/Direct Drafting.

---

# What are Logical and Display Tables?

Logical and display tables are techniques used to present a large amount of data in tabular format. It helps the user read and understand the data. Another advantage of using these techniques is that the user can select items from these tables as input to a command or function, thus enhancing the user-interface of the system.

You can use logical and display tables where you often need to manipulate and display lists of data. Refer to "Using Logical and Display Tables - Example 1" and "Using Logical and Display Tables - Example 2" to see the benefits of using them.

The following sections describe the structures of a logical table and a display table, and explain the concept of connecting a display table to a logical table.

## Logical Tables

A logical table is essentially an internal data-structure defined by the system or the user.

The system-defined logical tables contain system data. Although they are both readable and writable, you must not change the layout or data in system-defined logical tables. You are only allowed to read data from system-defined logical tables.

The user-defined logical tables, also called user tables, contain user data, and they are both readable and writable. You may create and change the layout of user-defined logical tables. Also, you can both read data from and write data to user-defined logical tables.

Generally, a logical table consists of three main components:

1. An array of N records identified by a number from 1 to N, where each record contains M values also identified by a number from 1 to M. In other words, it is similar to a two-dimensional N x M matrix of values. The number of records in an array can vary, but the number of values in each record must remain the same for all records.

   📝 **Note**

   Values in a logical table can be either text strings or numbers.

2. L title strings identified by numbers from 1 to L. Title strings are text strings which enable you to provide additional information, typically, for the titles of a display table. Refer to the section "Display Table" for details.

3. States of values in all records. Each value has an associated state to indicate that value as selected or active. This information is particularly useful. For example, if a user has selected some data from a display table as input for a command, this data can be highlighted to indicate the user's choices.

4. Foreground (display) and background colors of each cell in the table.

The following shows a schematic diagram of a logical table:

**Figure 31. Logical Table**
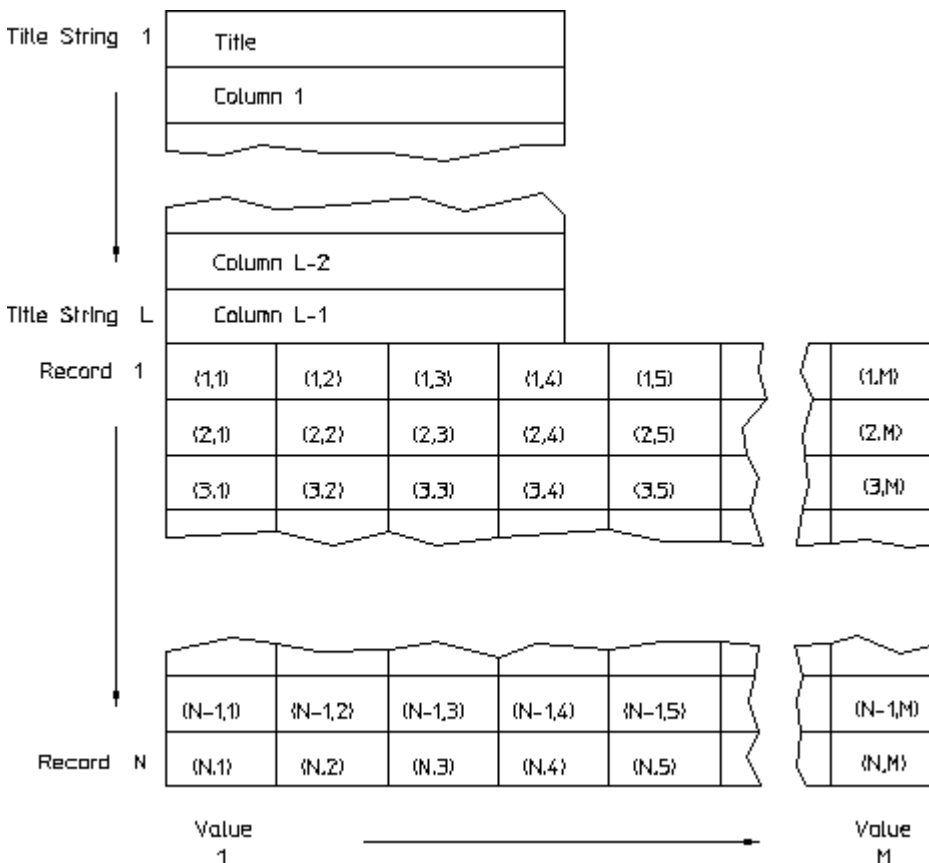
## Display Tables

A display table is a table used to display the contents of a logical table on the screen. A display table can be defined by the system or the user. Generally, it consists of the following components:

- Titles such as column headings and sub-headings

- A data area containing columns of values

- An optional VERTICAL scroll bar, which appears on the right-hand side of the table. No horizontal scroll bar is allowed.

An example of a display table is shown below:

**Figure 32. Display Table**



## Concept of Connecting Display to Logical Table

As described in the previous sections, there are two types of tables: logical tables and display tables. A logical table is an internal data-structure which stores the actual data, whereas a display table acts as a window for the user to view the data in a logical table. A display table also enables the user to interact with a logical table to change its data.

Each logical or display table is given a unique name when it is defined, and you can define as many logical or display tables as you need.

A display table can access and view a logical table only if it has been connected to that logical table. You can connect more than one display table to a logical table, so that if you have a logical table with a large amount of data, you can use display tables to view different parts of the logical table.

The advantage of this technique is that data stored in a logical table is presented consistently to the user, and the display tables are automatically updated when the data in the logical table is changed. It also saves you the work of defining the same display tables again.

The following diagram shows the mapping of a display table to a logical table:

**Figure 33. Mapping Display to Logical Table**
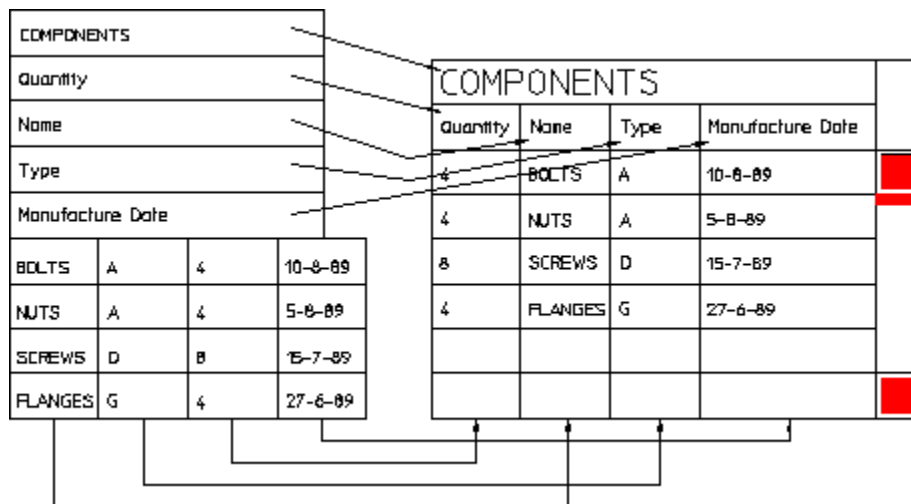


The arrows in the previous diagram indicate the way in which the items in the logical table are mapped to the items in the display table.

The title strings in the logical table are usually mapped to the table-heading and column-headings in the display table. You can map the data columns in the logical table in any order to the data columns in the display table. You can also map only some of the data columns in the logical table to the display table.

# Logical Table Access Functions

Logical table access functions enable you to access any logical tables, whether they are system-defined or user-defined. These functions are:

* `LTAB_COLUMNS`
* `LTAB_ROWS`
* `LTAB_TITLES`
* `POP_DOWN_LTAB`
* `POP_UP_LTAB`
* `READ_LTAB`
* `SAVE_LTAB`
* `SCROLL_LTAB`
* `SELECT_FROM_LTAB`

The following sections describe these functions in detail:

## LTAB_COLUMNS

This command enables you to enquire and return the number of columns in the specified logical table.

The format of the command is as follows:
```
LTAB_COLUMNS 'Logical table name'
```

An example of the command is given below:
```
LTAB_COLUMNS  'logtable1'
```

`LTAB_COLUMNS` requires a parameter, which is the name of the logical table to enquire. In this example, the logical table is `logtable1`

As the `LTAB_COLUMNS` command returns a value, this value has to be assigned to a variable, as shown in the following example. A typical use of the `LTAB_COLUMNS` command is:
```
LET  num_columns  (LTAB_COLUMNS  'logtable1')
```

which means the `LTAB_COLUMNS` command returns the number of columns in the logical table `logtable1` and then `LET` assigns it to the macro variable `num_columns`.

The returned value is a number.

## LTAB_ROWS

This command enables you to enquire and return the number of rows in the specified logical table.

The format of the command is as follows:
```
LTAB_ROWS 'Logical table name'
```

An example of the command is given below:
```
LTAB_ROWS  'logtable1'
```

`LTAB_ROWS` requires a parameter, which is the name of the logical table to enquire. In this example, the logical table is `logtable1`

As the `LTAB_ROWS` command returns a value, this value has to be assigned to a variable somewhere. A typical use of the `LTAB_ROWS` command is:
```
LET  num_rows  (LTAB_ROWS  'logtable')
```

which means the `LTAB_ROWS` command returns the number of rows in the logical table `logtable1` and then `LET` assigns it to the macro variable `num_rows`.

The returned value is a number.

## LTAB_TITLES

This command enables you to enquire and return the number of title strings in the specified logical table.

The format of the command is as follows:
```
LTAB_TITLES 'Logical table name'
```

An example of the command is given below:
```
LTAB_TITLES  'logtable1'
```

`LTAB_TITLES` requires a parameter, which is the name of the logical table to enquire. In this example, the logical table is `logtable1`

As the `LTAB_TITLES` command returns a value, this value has to be assigned to a variable somewhere. A typical use of the `LTAB_TITLES` command is:

```
LET  num_titles  (LTAB_TITLES  'logtable1')
```

which means the `LTAB_TITLES` command returns the number of titles in the logical table `logtable1` and then `LET` assigns it to the macro variable `num_titles`.

The returned value is a number.

## POP_DOWN_LTAB

This command enables you to make the specified logical table pop down at the next available opportunity. When a logical table pops down, all display tables connected to it are removed from the screen.

Pop down requests are first held in a buffer until the system is ready to accept interactive input from the user or `UPDATE_SCREEN` is called. Only then does it start processing the pop down requests.

The format of the command is as follows:

```
POP_DOWN_LTAB 'Logical table name'
```

An example of the command is given below:

```
POP_DOWN_LTAB  'logtable1'
```

`POP_DOWN_LTAB` requires a parameter, which is the name of the logical table to pop down. In this example, the logical table is `logtable1`

---

### Note

- When the `POP_DOWN_LTAB` command pops down a logical table, all display tables connected to this logical table are removed from the screen. If you want to remove only one of the display tables, you can use the `SHOW_TABLE` command with option `OFF` to do it.

- When the `POP_DOWN_LTAB` command removes a display table from the screen, it restores the saved bit-map image of the screen area under the removed display table, so that it looks the same as before. However, a special situation may arise, when you have overlapping display tables. For example, the saved bit-map image of a screen area contains an entire or a part of a display table which has already been removed from the screen since it was saved in the bit-map image. When the bit-map image is restored, this display table appears again on the screen. However, it does not exist as far as the system is concerned, so you may see on the screen a partial or an entire table which you cannot remove with the `TABLES OFF` command. If this situation occurs, you can use the `NEW_SCREEN` command to remove it from the screen. The `NEW_SCREEN` command regenerates the actual contents of the entire screen.

---

## POP_UP_LTAB

This command enables you to make the specified logical table pop up at the next available opportunity. When a table pops up, all display tables connected to it are displayed on the screen.

Pop up requests are first held in a buffer until the system is ready to accept interactive input from the user. Only then does it start processing the pop up requests.

The format of the command is as follows:
```
POP_UP_LTAB 'Logical table name'
```

An example of the command is given as below:
```
POP_UP_LTAB  'logtable1'
```

`POP_UP_LTAB` requires a parameter, which is the name of the logical table to pop up. In this example, the logical table is `logtable1`.

> **Note**
>
> - When you use the POP_UP_LTAB command on a logical table, all display tables connected to this logical table are displayed on the screen. If you want to display only one of the display tables, you can use the SHOW_TABLE command with option ON to do it.
> - When the POP_UP_LTAB command displays a display table on the screen, it saves a bit-map image of the screen area under the display table, When the display table is removed from the screen using the POP_DOWN_LTAB, the area is restored to its original image.

## READ_LTAB

This command enables you to read the values from a certain area in the specified logical table.

The format of the command is as follows:
```
READ_LTAB 'Logical table name'
(Row-number Column-number) or (TITLE Title-string-number)
```

An example of the command is given below:
```
READ_LTAB  'logtable1'
2  3
```

READ_LTAB requires two parameters. The first parameter is the name of the logical table to read from. In this example, the logical table is logtable1. The second parameter is the position of the data to be read from the specified logical table, which is 2 3 in this example, that is, row 2 and column 3 of the specified logical table.

If you want to read a value from the title string of the specified logical table, you specify TITLE 5 instead of 2 3 to indicate title string 5.

As the READ_LTAB command returns a value, this value has to be assigned to a variable somewhere. A typical use of the READ_LTAB command is:
```
LET  my_value  (READ_LTAB  'logtable1' 1  4)
```

which means the READ_LTAB command reads the value from the fourth column of the first row in the logical table logtable1, and then LET assigns it to the macro variable my_value.

The returned value may be a string or number. If the specified position contains no value, an empty string is returned instead.

*Creo Elements/Direct Drafting Writing Macros*

## SAVE_LTAB

This command enables you to save the contents of the specified logical table.

The format of the command is as follows:
```
SAVE_LTAB 'Logical table name'
SCREEN or (( or DEL_OLD or APPEND) 'filename')
```

An example of the command is given below:
```
SAVE_LTAB  'logtable1'
DEL_OLD  'file1.tbl'
```

`SAVE_LTAB` requires two parameters. The first parameter is the name of the logical table to save. In this example, the logical table is `logtable1`. The second parameter indicates the output destination to which the specified logical table is saved. In this example, the output destination is the file `file1.tbl` with the option `DEL_OLD`, which means if `file1.tbl` already exists, it is overwritten.

If you specify a file as the output destination, you can alternatively use option `APPEND` or no option at all. `APPEND` means the specified logical table is appended to the end of the specified file if it exists. If it does not exist, the specified file is created. If you do not specify any option, the specified file is created provided no other file of the same name exists. If the specified file already exists, an error message is displayed and the command is aborted.

Alternatively, you can specify the screen as the output destination by specifying `SCREEN` instead of `DEL_OLD` and `file1.tbl` in the above example. But, the advantage of saving a table in a file is that you can subsequently use the `INPUT` command to input this file to redefine the table.


## SCROLL_LTAB

This command enables you to scroll display tables connected to the specified logical table, so that the specified row is at the top of the display.

The format of the command is as follows:
```
SCROLL_LTAB 'Logical table name'
Row-number
```

An example of the command is given below:
```
SCROLL_LTAB  'logtable1'
3
```

`SCROLL_LTAB` requires two parameters. The first parameter is the name of the logical table to scroll. In this example, the logical table is `logtable1`. The second parameter indicates the row to be scrolled to the top of the display table, which is `3` in this example.

# SELECT_FROM_LTAB

This command enables you to select rows from a logical table, called the source logical table, according to certain user-specified criteria, and to write the positions of matching rows to the logical table sys_select. Optionally, the command also enables you to copy the matching rows to another logical table, called the destination logical table.

The format of the command is as follows:
```
SELECT_FROM_LTAB 'Source Logical table name'
COLUMN Column-number
or ( = or <> or > or < or >= or <= )
'Selection-string' or Selection-number
END or ( or APPEND 'Destination Logical table name')
```

An example of the command is given below:
```
SELECT_FROM_LTAB  'logtable1'
COLUMN  1
 >=
10
END
```

SELECT_FROM_LTAB requires five parameters. The first parameter is the name of the source logical table, from which you want to select data. In the above example, the logical table is logtable1. The second parameter is the position of the column used for selection which is COLUMN 1, that is, the first column, in the above example. The third parameter is the selection operator which is >=, that is, greater than or equal to, in the above example. Alternative operators you can specify are =, <>, >, < and <=. The fourth parameter is the selection string or value which is 10, that is, the number 10 and not string '10' , in the above example. The fifth parameter is either END or the name of the destination logical table to which the selected rows are copied. In the above example, END is specified and it means the end of the command.

In short, the above example means it selects rows from the source logical table logtable1 if COLUMN 1 of a row contains a value = 10, and writes the positions of the selected rows to the logical table sys_select.

If, for example, the logical table logtable1 contains the following data:
```
1     20
5     25
10    11
22    17
43    13
```

and you run the macro program in the above example. The result is that the logical table sys_select contains the following data:
```
3
4
5
```

because only rows 3, 4 and 5 of logtable1 satisfy the selection criteria.

Another example of the command is:

```
SELECT_FROM_LTAB  'logtable1'
COLUMN  2
=
'PARTS'
APPEND  'logtable2'
```

In short, the above example means it selects rows from the source logical table `logtable1` if COLUMN 2 of a row contains a string = `'PARTS'`, and it writes the positions of the selected rows to the logical table `sys_select`. In addition, it will APPEND, that is, copy the selected rows to the end of the destination logical table `logtable2`.

When you specify the option APPEND, the command assumes the specified logical table already exists. If you do not specify the option APPEND, the command assumes the specified logical table does not exist and a new logical table with the specified name is created.

If, for example, the logical tables `logtable1` and `logtable2` contain respectively the following data:

```
PISTON        DRAWING     4
BOLTS         PARTS       10
NUTS          PARTS       10
CRANKSHAFT    DRAWING     1
WASHERS       PARTS       10
```

and

```
WHEEL         ASSEMBLY    2
FRAME         ASSEMBLY    1
HANDLE        ASSEMBLY    1
```

and you run the macro program in the above example. The result is that the logical tables `sys_select` and `logtable2` contain respectively the following data:

```
2
3
5
```

and

```
WHEEL         ASSEMBLY    2
FRAME         ASSEMBLY    1
HANDLE        ASSEMBLY    1
BOLTS         PARTS       10
NUTS          PARTS       10
WASHERS       PARTS       10
```

because only rows 2, 3 and 5 of `logtable1` satisfy the selection criterion. You can see the contents of rows 2, 3 and 5 of `logtable1` are copied and appended to the end of `logtable2`.

# Display Table Functions

A display table provides a means to display all or selected contents of a logical table. Refer to the section "Display Tables" for details.

The following sections describe the commands and functions that you need to define the layout of a display table and to use the display table. There is also an example of defining a display table.

The commands for defining a display table are:

- `TABLE_COLUMN`
- `TABLE_LAYOUT`
- `TABLE_TITLE`

Refer to the section "Defining a Display Table" in the section "Using Logical and Display Tables - Example 1" for details of how to define a display table.

The following sections describe each of the above commands in detail.

## TABLE_COLUMN

This command enables you to set the attributes of the slots in the data columns of an existing display table.

The format of the command is as follows:
```
TABLE_COLUMN 'table name'
COLUMN column-number
display color
background color
Logical-table-column-number
FORMAT Format-precision or 'Numeric string'
CENTER or LEFT or RIGHT
'Action Text'
END
```

An example of the command is given below:

```
TABLE_COLUMN 'table1'
  COLUMN 1
  blue
  yellow
  1
  FORMAT 2
  LEFT
  'LINE'

  COLUMN 2
  red
  green
  2
  FORMAT 3
  RIGHT
  'CIRCLE'
END
```

📝 **Note**

In the above example, the numbers on the left hand side are NOT part of the format. They are there to help you refer to each line in the example. The empty lines are there to help you read the example and understand its structure.

Line 1 contains `TABLE_COLUMN` and the name of the display table whose columns are to be defined.

Line 2 specifies `COLUMN 1`, which means column 1 of the display table is to be defined.

Line 3 specifies the display color for text in the column. In this example, `blue` is specified. If you need to use a different color, you can follow the description in section Line 3 of `TABLE_LAYOUT` for specifying a color.

Line 4 specifies the background color of the column. In this example, `yellow` is specified. If you need to use a different color, you can follow the description in section of `TABLE_LAYOUT` for specifying a color.

Line 5 specifies the position of the column in the connected logical table to be shown in this column in the display table.

Line 6 specifies `FORMAT 2`, which means the format-precision for floating-point numbers is two significant figures. For example, if the floating-point number is 123.0, the actual number will be 120. If the number is 12.6, the actual number will be 13. The default format-precision is half the width of the column.

Or, you can specify a numeric string such as `'+1.2345'`, which means the number will be signed, four decimal places, left-zero and right-zero suppressed. Use the `HELP` command to refer to the `DIM_FORMAT` function for details about the format of this numeric string.

Line 7 specifies `LEFT`, which means the data in the column is to be left-justified. You can specify one of two other possible values, `CENTER` and `RIGHT`, which mean center-justified and right-justified respectively.

Line 8 specifies the action text `LINE` in the column. You can specify any text string that is a valid system command.

You can specify a string of the format `@s#`, which means the text string comes from the title string of number # in the logical table connected with this display table. For example, if `@s3` is specified, it means that the extra string 3 of the connected logical table is the actual text string specified. If you specify `@t3`, it means the extra string 3 of the connected logical table in single quotes is the actual text string specified. `@s3` and `@t3` are the same except that `@t3` has two single quotes, one before and one after the extra string 3.

You can also specify a string of the format `@v#`, which means the text string comes from the data in column # of the logical table connected with this display table. For example, if `@v4` is specified, it means that the data in column 4 of the connected logical table is the actual text string specified. If you specify `@q4`, it means the data in column 4 of the connected logical table in single quotes is the actual text string specified. `@v4` and `@q4` are the same except that `@q4` has two single quotes, one before and one after the data in column 4.

You can even specify a text string of the form `LINE @s4 @v2`. If the extra string 4 and data in column 2 of the connected logical table are `TWO_PNTS` and `33,33` respectively, the actual text string is `LINE TWO_PNTS 33,33`.

Lines 10 to 16 is another block of parameters similar to Lines 2 to 8 to define another column in the title of the display table.

Line 17 specifies `END` to indicate the end of the `TABLE_COLUMN` command.

You can include as many blocks of parameters similar to Lines 2 to 8 as required, provided they are within the commands `TABLE_COLUMN` and `END`.

The above example can actually be rearranged as:

```
TABLE_COLUMN 'table1'
  COLUMN 1 blue yellow 1 FORMAT 2 LEFT  'LINE'
  COLUMN 2 red  green  2 FORMAT 3 RIGHT 'CIRCLE'
END
```

If a block of parameters can fit into a line, this format may be easier to read and compare. The important point is that parameters must be separated by at least one space or tab character.

## TABLE_LAYOUT

This command enables you to create a new table with the required layout.

The format of the command is as follows:

```
TABLE_LAYOUT 'table name'
'Logical table name'
display color
background color
WIDTH width
HEIGHT height
ROWS row
FRAME_WIDTH frame-width
HORIZONTAL Line-color Linetype
VERTICAL Line-color Linetype
SCROLL_BAR Foreground Color background color width
Point 1 and/or Point 2

TITLE_LAYOUT
Height of Row 1 'Layout String'
Height of Row 2 'Layout String'
:
END

COLUMN_LAYOUT
Height of Each Data Row 'Layout String'

END
```

An example of the command is given below:

```
TABLE_LAYOUT 'table1'
  'query_results'
  white
```

```
   black
   WIDTH   30.0
   HEIGHT  28.0
   ROWS    10
   FRAME_WIDTH 1
   HORIZONTAL white solid
   VERTICAL white solid
   SCROLL_BAR blue white 30
   0,0

   TITLE_LAYOUT
      40 '                                                  '
      20 '                    |        |    |                |          '

END

   COLUMN_LAYOUT
      20 '                    |        |    |                |          '

END
```

---

## 📝 Note

In the above example, the numbers on the left hand side are NOT part of the format. They are there to help you refer to each line in the example. The empty lines are there to help you read the example and understand its structure.

---

Line 1 contains the command `TABLE_LAYOUT` and the name you specify for this display table.

Line 2 specifies the name of the logical table for this display table. In this example, `query_results` is specified. By default, the logical table name is the same as the display table name.

Line 3 specifies the display color for the frame and text in the display table. In this example, `white` is specified. However, you can specify any of the other colors already defined in the system such as `red, yellow, green, cyan, magenta, blue` or `black`.

If you need to use a color that is not one of the above eight defined in the system, you can specify the command `rgb_color` followed by the decimal values of the colors red, green and blue in that order. For example, if you want pure red, green or blue, you specify `rgb_color 1 0 0`, `rgb_color 0 1 0` or `rgb_color 0 0 1` respectively. By combining these three colors, you can define a certain required color such as `rgb_color 0.5 0.42 0.8`, which means `0.5` of red, `0.42` of green and `0.8` of blue.

Line 4 specifies the background color of the display table. In this example, `black` is specified.

If you need to use a different color, you can follow the description in section Line 3 for specifying a color.

Line 5 contains `WIDTH` and the table width `30.0` characters. This value can be fractional, for example `35.5` and must be greater than a certain minimum width. If you do not specify this parameter, the minimum width for fitting the table is used.

Line 6 contains `HEIGHT` and the table height `28.0` characters. This value can be fractional, for example `35.5` and must be greater than a certain minimum height. If you do not specify this parameter, the minimum height for fitting the table is used.

Line 7 contains `ROWS` and the number of data rows `10` required in the display table. This parameter can be fractional, for example `10.5`. It is useful to use this parameter only when the width and height of the table are not explicitly specified.

Line 8 contains `FRAME_WIDTH` and the frame-width value `1`. This parameter can take a value of either 0, 1 or 2.

Line 9 contains `HORIZONTAL`, the horizontal-line color `white` and its type `solid`. If you need to use a different color for the horizontal lines, you can follow the description in section Line 3 for specifying a color. If you need to use a different linetype, you can specify one of the other linetypes already defined in the system such as `dotted`, `dashed`, `long dashed`, `dot center`, `dash center`, `phantom` or `long dotted`. If you specify just `OFF` after `HORIZONTAL`, there will be no horizontal lines in the display table.

Line 10 contains `VERTICAL`, the vertical-line color `white` and its type `solid`. If you need to use a different color or linetype, you can follow the description in section Line 9 for specifying a color or linetype. If you specify just `OFF` after `VERTICAL`, there will be no vertical lines in the display table.

Line 11 contains `SCROLL_BAR`, the foreground color `blue`, the background color `white`, and the scroll-bar width `30` pixels.

Line 12 specifies the coordinates (0,0) in pixels of the lower-left-hand corner of the display table. Optionally, you can also specify the coordinates of the upper-right-hand corner of the display table.

Line 14 contains `TITLE_LAYOUT`, which is the parameter to specify the layout of the title in the display table.

Line 15 specifies the height `40` pixels of the first title row and the layout string `' '`, which indicates the number of slots and the width of each slot. In this example, there is only one slot of 50 characters wide.

Line 16 specifies the height `20` pixels of the second title row and the layout string `' | | | | '`, which indicates the number of slots and their widths. In this example, there are five slots of 14, 6, 3, 14 and 9 characters wide. The bar `|` separates the slots.

---

**📋 Note**

In this example, there are only two rows in the title. If you have more, they can be specified in a similar format.

---

Line 18 contains `END` to mark the end of the `TITLE_LAYOUT` parameter.

Line 20 contains `COLUMN LAYOUT`, which is the parameter to specify the layout of the data columns and rows in the display table.

Line 21 specifies the height `20` pixels of each data row and the layout string `' | | | | '`, which indicates the number of columns and their widths. In this example, there are five columns of 14, 6, 3, 14 and 9 characters wide. The bar `|` separates the columns.

Line 23 contains `END` to mark the end of the `COLUMN_LAYOUT` parameter and the `TABLE_LAYOUT` command.

---

**📋 Note**

You can specify the size of a display table by a combination of the parameters `WIDTH`, `HEIGHT`, `Point 1` and `Point 2` in the the `TABLE_LAYOUT` command. Also, you can specify the width of the title area in `TITLE_LAYOUT` and the width of the column area in `COLUMN_LAYOUT`. If the width table specified in `TABLE_LAYOUT` is different from the width in `TITLE_LAYOUT` or `COLUMN_LAYOUT`, the width in `TABLE_LAYOUT` will take precedence. So, the width of the display table will always be the width specified in `TABLE_LAYOUT`.

> For example, if the width in `TABLE_LAYOUT` is 18 columns and the width in `TITLE_LAYOUT` or `COLUMN_LAYOUT` is 9 columns as follows:
>
> ```
> 40 '   |   | '
> ```
>
> the width of the display table will be 18 columns, and the width of the titles and columns will then change to 18 columns, but the proportion of the titles and columns will be maintained as follows:
>
> ```
> 40 '       |       | '
> ```

# TABLE_TITLE

This command enables you to set the attributes of the slots in the title of an existing display table.

---

📝 **Note**

This command can only operate on a display table that has not been secured against change by the `SECURE_TABLE` command.

---

The format of the command is as follows:
```
TABLE_TITLE 'table name'
display color
background color
'display text' 'Action Text'
(Row, Column of Slot) or (BOX Row1, Column1 to Row2, Column2 of Slots)
:
Repeat the above four lines if you need to define other title-slots
:
END
```

An example of the command is given below:
```
TABLE_TITLE 'table1'
  blue
  yellow
  'LINE' 'LINE'
  1 1

  green
  white
  '@s2' '@s1'
  BOX  1 2  2 3
END
```

---

📝 **Note**

In the above example, the numbers on the left hand side are NOT part of the format. They are there to help you refer to each line in the example. The empty lines are there to help you read the example and understand its structure.

---

Line 1 contains the command `TABLE_TITLE` and the name of the display table whose title is to be defined.

Line 2 specifies the display color for text in the slot. In this example, `blue` is specified. If you need to use a different color, you can follow the description in section Line 3 of "TABLE_LAYOUT" for specifying a color.

Line 3 specifies the background color of the slot. In this example, `yellow` is specified. If you need to use a different color, you can follow the description in section Line 3 of "TABLE_LAYOUT" for specifying a color.

Line 4 specifies the display text `LINE` in the slot and the associated action text `LINE`.

A display text is a text string shown in the display table. Its associated action text is also a text string, that contains a valid system command and is sent to the system when the corresponding display text is selected from the display table on the screen.

You can specify any valid text strings in these parameters. You can also specify a string of the format @s#, which means the text string comes from the title string of number # in the logical table connected with this display table. For example, if `@s3` is specified in one of these parameters; it means that the extra string 3 of the connected logical table is the actual text string specified.

You can even specify a text string of the form `UNITS=@s2`. If the extra string 2 of the connected logical table is `Meters`, the actual text string is `UNITS= Meters`.

If you specify a text string of the form `UNITS=@t2`. and the extra string 2 of the connected logical table is `Meters`, the actual text string is `UNITS='Meters'` .

Line 5 specifies `1 1`, that is, row 1 and column 1, as the position of the slot to be defined. You can also specify a box of slots by using the parameter BOX. For example, `BOX 1 2 3 3` as in Line 10 means that the slots from row 1, column 2 to row 3, column 3 form the box required.

Lines 7 to 10 is another block of parameters similar to Lines 2 to 5 to define slots in the title of the display table.

Line 11 specifies `END` to indicate the end of the `TABLE_TITLE` command.

---

## 📋 Note

You can include as many blocks of parameters similar to Lines 2 to 5 as required, provided they are within the commands `TABLE_TITLE` and `END`.

The above example can actually be rearranged as:

```
TABLE_TITLE 'table1'
  blue yellow 'LINE'  'LINE' 1 1
  green white '@s2' '@s1' BOX  1 2  2 3
END
```

If a block of parameters can fit into a line, this format may be easier to read and compare. The important point is that parameters must be separated by at least one space or tab character.

The commands for handling a display table are:

- `CHANGE_TABLE_SIZE`
- `CONNECT_TABLE`
- `DELETE_TABLE`
- `MOVE_TABLE`
- `PRINT_TABLE`
- `SAVE_TABLE`
- `SECURE_TABLE`
- `SHOW_TABLE`
- `TABLE_SCROLL_STEP`

and they are described in detail in the following sections.

## CHANGE_TABLE_SIZE

This command enables you to change the size of existing display tables which have not been secured internally against size-change by the program.

The format of the command is as follows:
```
CHANGE_TABLE_SIZE 'table name' Point 1 Point 2
```

An example of the command is given below:
```
CHANGE_TABLE_SIZE  'disptable1'  100,100  1000,800
```

CHANGE_TABLE_SIZE requires three parameters, which are `disptable1`, `100,100` and `1000,800` in this example.

`disptable1` is the name of the display table whose size is to be changed. `100,100` and `1000,800` are respectively the x,y coordinates of the lower-left-hand and upper-right-hand corners of the display table.

---

**📝 Note**

The width and height of the display table must be greater than certain minimum dimensions for every table.

---

## CONNECT_TABLE

This command enables you to connect a display table to a logical table.

The format of the command is as follows:
```
CONNECT_TABLE 'table name' 'Logical table name'
```

An example of the command is given below:

```
CONNECT_TABLE  'disptable1'  'logtable1'
```

CONNECT_TABLE requires two parameters, which are `disptable1` and `logtable1` in this example.

`disptable1` is the name of the display table to be connected to the logical table `logtable1`.

If this command tries to connect a display table, which is already connected to a logical table, to another logical table, the existing connection between the display table and the first logical table is automatically broken before a new connection is made with the second logical table. This means you can connect only one logical table to any display at any one time.

## DELETE_TABLE

This command enables you to delete an existing display table which has not been secured.

The format of the command is as follows:
```
DELETE_TABLE
'table name' or (ALL CONFIRM)
```

An example of the command is given below:
```
DELETE_TABLE
ALL  CONFIRM
```

or, simply,
```
DELETE_TABLE  ALL  CONFIRM
```

DELETE_TABLE requires a parameter, which is `ALL CONFIRM` in this example. `ALL` means that the command deletes all the display tables defined in the system. `CONFIRM` is to confirm that the command is correct.

Alternatively, you can specify the name of a display table as follows:
```
DELETE_TABLE  'disptable1'
```

to indicate that the display table `disptable1` is to be deleted.

## MOVE_TABLE

This command enables you to move an existing display table to a specific location on the screen. The display table must not have been secured internally against move by the program.

The format of the command is as follows:
```
MOVE_TABLE 'table name1'
(Point 1 Point 2) or
((UPPER or LOWER or RIGHT or LEFT) and/or (OF 'table name2'))
:
END
```

An example of the command is given below:
```
MOVE_TABLE  'disptable1'
UPPER  OF  'disptable2'
100,100  150,400
END
```

---

### 📝 Note

In the above example, the numbers on the left hand side are NOT part of the format. They are there to help you refer to each line in the example.

---

Line 1 contains the command `MOVE_TABLE` and the name `disptable1` of the display table to be moved.

Line 2 specifies `UPPER` to indicate a move to the top of the screen. You can alternatively specify one of the other directions such as `LOWER`, `RIGHT` or `LEFT` to indicate a move to the bottom, right or left respectively of the screen.

You can optionally specify `OF` and the name of a display table `'disptable2'` after the direction to indicate the move is relative to another display table.

Line 3 specifies `100,100` and `150,400`, which are the x,y coordinates of the reference and destination points of another move.

Line 4 specifies `END` to indicate the end of the `MOVE_TABLE` command.

---

### 📝 Note

Both Line 2 and Line 3 are valid move parameters, but only specified in different forms. In other words, there are two moves in this example.

It is possible to specify multiple moves in the `MOVE_TABLE` command, provided the parameters are included within `MOVE_TABLE` and `END`. The actual move will be the result of combining all the moves together.

---

## PRINT_TABLE

This command enables you to print the visible contents of a display table to the screen or into a file.

The format of the command is as follows:
```
PRINT_TABLE 'table name' or ALL
SCREEN or (( or DEL_OLD or APPEND) 'filename')
```

An example of the command is given below:
```
PRINT_TABLE  'disptable1'
DEL_OLD  'file1.tbl'
```

or, simply,
```
PRINT_TABLE  'disptable1'  DEL_OLD 'file1.tbl'
```

`PRINT_TABLE` requires two parameters. The first parameter is the name of the display table to print. In this example, the display table is `disptable1`. The second parameter indicates the output destination to which the specified display table is printed. In this example, the output destination is the file `file1.tbl` with the option `DEL_OLD`, which means if `file1.tbl` already exists, it is overwritten.

If you specify a file as the output destination, you can alternatively use option `APPEND` or no option at all. `APPEND` means the specified display table is appended to the end of the specified file if it exists. If it does not exist, the specified file is created. If you do not specify any option, the specified file is created provided no other file of the same name exists. If the specified file already exists, an error message is displayed and the command is aborted.

Alternatively, you can specify the screen as the output destination by specifying `SCREEN` instead of `DEL_OLD` and `file1.tbl` in the above example.

## SAVE_TABLE

This command enables you to save the set-up of a display table in a file. A display table can then be re-created by reading the file using an INPUT command.

The format of the command is as follows:
```
SAVE_TABLE 'table name' or ALL
SCREEN or (( or DEL_OLD or APPEND) 'filename')
```

An example of the command is given below:
```
SAVE_TABLE  'disptable1'
DEL_OLD  'file1.tbl'
```

or, simply,
```
SAVE_TABLE  'disptable1'  DEL_OLD  'file1.tbl'
```

`SAVE_TABLE` requires two parameters. The first parameter is the name of the display table to save. In this example, the display table is `disptable1`. The second parameter indicates the output destination to which the specified display table is saved. In this example, the output destination is the file `file1.tbl` with the option `DEL_OLD`, which means if `file1.tbl` already exists, it is overwritten.

If you specify a file as the output destination, you can alternatively use option `APPEND` or no option at all. `APPEND` means the specified display table is appended to the end of the specified file if it exists. If it does not exist, the specified file is created. If you do not specify any option, the specified file is created provided no other file of the same name exists. If the specified file already exists, an error message is displayed and the command is aborted.

Alternatively, you can specify the screen as the output destination by specifying `SCREEN` instead of `DEL_OLD` and `file1.tbl` in the above example.

## SECURE_TABLE

This command enables you to secure a display table so that it cannot be deleted or redefined.

The format of the command is as follows:
```
SECURE_TABLE
'table name' or (ALL CONFIRM)
```

An example of the command is given below:
```
SECURE_TABLE
ALL   CONFIRM
```

or, simply,
```
SECURE_TABLE ALL CONFIRM
```

`SECURE_TABLE` requires one or two parameters, which are `ALL CONFIRM` in this example. `ALL` means that the command secures all the display tables defined in the system. `CONFIRM` is to confirm that the command is correct.

Alternatively, you can specify the name of a display table as follows:
```
SECURE_TABLE  'disptable1'
```

to indicate that the display table `disptable1` is to be secured.

## SHOW_TABLE

This command enables you to show or not to show a single display table or all display tables.

The format of the command is as follows:
```
SHOW_TABLE
ON or OFF
'table name' or ALL
```

An example of the command is given below:
```
SHOW_TABLE
ON
ALL
```

or, simply,
```
SHOW_TABLE  ON  ALL
```

`SHOW_TABLE` requires two parameters, which are `ON` and `ALL` in this example.

`ON` means to show the display table or tables. You can alternatively specify `OFF` to indicate not to show it.

`ALL` means to run this command on all the display tables defined in the system. You can alternatively specify the name of a display table for the command.

> 📄 **Note**

You can also use `POP_UP_LTAB` and `POP_DOWN_LTAB` commands respectively to show and not to show a display table. Refer to the sections `POP_UP_LTAB` and `POP_DOWN_LTAB` for details.

The differences between SHOW_TABLE ON (or OFF) and POP_UP_LTAB (or POP_DOWN_LTAB) are as follows:

- `SHOW_TABLE` shows a display table you specified or all display tables, whereas `POP_UP_LTAB` shows all display tables connected to a logical table you specified.

- `POP_UP_LTAB` saves a bit-map image of the screen area under the display table it displays, whereas `SHOW_TABLE` does not.

## TABLE_SCROLL_STEP

This command enables you to set the scroll step of a display table while using the scroll boxes at the ends of the scroll bar.

The format of the command is as follows:
```
TABLE_SCROLL_STEP
'table name' or ALL
DEFAULT or number
```

An example of the command is given below:
```
TABLE_SCROLL_STEP
ALL
DEFAULT
```

or, simply,
```
TABLE_SCROLL_STEP  ALL  DEFAULT
```

`TABLE_SCROLL_STEP` requires two parameters, which are `ALL` and `DEFAULT` in this example.

`ALL` means this command changes the scroll step of all the display tables defined in the system. Alternatively, you can specify a display table such as `disptable1` to indicate a particular table whose scroll step is to be changed.

`DEFAULT` means the scroll step is to be set as the number of data rows in the display table. Alternatively, you can specify a certain number in data rows such as 5 to indicate the scroll step size.

# User Table Functions

A user table is a logical table defined by the user, as opposed to one defined by the system. It has the same data-structure as any logical table. Also, you can connect a display table to a user table in the same way as to any logical table.

The command for defining a user table is:

- `CREATE_LTAB`

The commands for handling a user table are:

- `COLOR_LTAB`
- `DELETE_LTAB`
- `DELETE_LTAB_ROW`
- `HIGHLIGHT_LTAB`
- `SECURE_LTAB`
- `SORT_LTAB`
- `WRITE_LTAB`

The following sections describe each of the above commands in detail.

### 📋 Note

In the listing of each command format in the following sections, the normal-font text indicate they are part of the format, so you must specify them as they appear here. The italic text indicate you must replace them with some suitable strings or values. The word or means there are two or more options, one of which you can choose for a parameter.

## COLOR_LTAB

This command enables you to specify the foreground (display) and background colors of any cell in a table.

### 📋 Note

The command `COLOR_LTAB` can work on user tables that have been secured with the command `SECURE_LTAB` and the option `READ_ONLY`.

The format of the command is as follows:
```
COLOR_LTAB 'User table name'
(Row-no Column-no) or (ROW Row-no) or (COLUMN Column-no) or ALL
```

```
or (TITLE title string-no or ALL)
(Foreground Color or DEFAULT)
(Background Color or DEFAULT)
```

An example of the command is given below:
```
COLOR_LTAB  'usertable1'
3  5
white
blue
```

COLOR_LTAB requires four pieces of data. The first is the name of the user table for the command. The second is the position of the area in the specified user table whose colors are to be changed, and the third and fourth are respectively the foreground and background colors of the cell.

In the above example, row 3 and column 5 in the user table usertable1 is to be assigned the foreground color white and background color blue.

You can specify the position in five other ways, for example:

• ROW 5, which means the entire row 5.

• COLUMN 3, which means the entire column 3.

• ALL, which means all rows and columns.

• TITLE 2, which means the second title string.

• TITLE ALL, which means all title strings.

Also, you can specify any valid 10 color names for the foreground and background colors.

Another example of the command is:
```
HIGHLIGHT_LTAB 'usertable1'
TITLE 2
black
yellow
```

which means assigning black as the foreground color of title string 2, and yellow as the background color.

## CREATE_LTAB

This command enables you to create a new user table, and optionally to specify an estimate of the size in rows and columns of the new user table.

Refer to the section "Defining a User Table" in "Using Logical and Display Tables - Example 1" for details of how to define a user table.

The optional parameters for the size of the new user table help the system estimate resources for the user table and make the user table run as efficiently as possible.

If the new user table to be created has the same name as an existing system-defined table which has been secured READONLY, this command is ignored and an error message is displayed.

If the new user table to be created has the same name as an existing user table which is not secured, the command simply deletes all data in the existing user table and adjusts its size according to the rows and columns specified in the command.

However, if the new user table to be created has the same name as an existing user table which is secured by the SECURE_LTAB command and the READ_ONLY option, an error message is displayed.

The format of the command is as follows:
```
CREATE_LTAB (or Rows) (or Columns)
'User table name'
```

An example of the command is given below:
```
CREATE_LTAB
20 6
'usertable1'
```

CREATE_LTAB requires two pieces of data. The first is the estimate of the size in rows and columns of the new user table. In the above example, the estimated size is 20 rows and 6 columns. This size is only an estimate, and not a limit. So, you can still write data to the table at a position beyond 20 rows and 6 columns. The second is the name of the new user table which is usertable1 in the above example.

## DELETE_LTAB

This command enables you to delete a user table.

If the specified user table is secured by the SECURE_LTAB command or is locked because it is currently being used, an error message is displayed.

The format of the command is as follows:
```
DELETE_LTAB ALL or 'User table name'
```

An example of the command is given below:
```
DELETE_LTAB  ALL
```

DELETE_LTAB requires a parameter which is ALL in this example. ALL means that the command deletes all the user tables.

Alternatively, you can specify the name of the user table as follows:
```
DELETE_LTAB  usertable1
```

to indicate the user table usertable1 to be deleted.

## DELETE_LTAB_ROW

This command enables you to delete a row from the user table.

If the specified user table is secured with the command SECURE_LTAB and the option READ_ONLY, an error message is displayed.

The format of the command is as follows:
```
DELETE_LTAB_ROW 'User table name'
ALL or Row-number
```

An example of the command is given below:
```
DELETE_LTAB_ROW  'usertable1'
ALL
```

DELETE_LTAB_ROW requires two pieces of data. The first is the name of the user table for the command. The second is the position of the row in the specified user table to be deleted.

In the above example, ALL rows in the user table usertable1 are to be deleted.

Alternatively, you can specify the position of the row as follows:
```
DELETE_LTAB_ROW  'usertable1'
5
```

to indicate that row 5 in the user table usertable1 is to be deleted.


## HIGHLIGHT_LTAB

This command enables you to switch on or off the highlight of a position in the data area of the user table.

### Note

The command HIGHLIGHT_LTAB can work on user tables that have been secured with the command SECURE_LTAB and the option READ_ONLY.

The format of the command is as follows:
```
HIGHLIGHT_LTAB 'User table name'
(Row-no Column-no) or (ROW Row-no) or (COLUMN Column-no) or ALL
or (TITLE title string-no or ALL)
ON or OFF or MARK
```

An example of the command is given below:
```
HIGHLIGHT_LTAB  'usertable1'
3  5
ON
```

`HIGHLIGHT_LTAB` requires three pieces of data. The first is the name of the user table for the command. The second is the position of the area in the specified user table whose highlight is to be changed, and the third is the ON/OFF state to switch the highlight to.

In the above example, row 3 and column 5 in the user table `usertable1` is to be highlighted, that is, `ON`.

You can specify the position in five other ways, for example:

- `ROW 5`, which means the entire row 5.
- `COLUMN 3`, which means the entire column 3.
- `ALL`, which means all rows and columns.
- `TITLE 2`, which means the second title string.
- `TITLE ALL`, which means all title strings.

Also, you can specify `OFF` to switch off the highlight or `MARK` to mark the box with an inner box.

Another example of the command is:
```
HIGHLIGHT_LTAB 'usertable1'
TITLE 2
MARK
```

which means highlighting title string 2 with a `MARK` which is an inner box.

## SECURE_LTAB

This command enables you to secure a specified user table so that it cannot be deleted, but you can still change its contents. If you want to secure a user table such that it cannot be deleted and its contents cannot be changed, you need to use the option `READ_ONLY`.
```
The format of the command is as follows:
SECURE_LTAB (or READ_ONLY) 'User table name'
```

An example of the command is given below:
```
SECURE_LTAB  'usertable1'
```

`SECURE_LTAB` requires a parameter to specify the user table to be secured. In this example, the user table is `usertable1`.

You can also specify the option `READ_ONLY` as follows:
```
SECURE_LTAB  READ_ONLY  'usertable1'
```

so that the contents of `usertable1` cannot be changed.

## SORT_LTAB

`SORT_LTAB` sorts a user table based upon the columns specified. `REVERSE_SORT` reverses the order of the sorting for the next column specified.

The format of the command is as follows:
```
SORT_LTAB 'User table name'
Column-no or REVERSE_SORT Column-no
:
(Repeat the last line if you need to specify other columns to sort)
:
CONFIRM
```

An example of the command is given below:
```
SORT_LTAB 'usertable1'
REVERSE_SORT 2
CONFIRM
```

SORT_LTAB requires three pieces of data. The first is the name of the user table for the function. The second is the sorting order, either forward (ascending) or reverse (descending). The default is forward sorting. The third is the number of the column whose data is to be sorted.

Sort order (precedence) of differing types in the same column is:

1. NULL values.

2. numbers.

3. strings.

In the above example, the data in the user table usertable1 is to be sorted in reverse order according to the data in column 2. CONFIRM indicates the end of the SORT_LTAB function.

You can also specify more than one column for this function, in which case the first specified column takes the highest precedence for the sorting. For example:
```
SORT_LTAB  'usertable1'
REVERSE_SORT 3
1
REVERSE_SORT 2
CONFIRM
```

Sorting column 3 in reverse order takes the highest precedence over column 1 or 2.

An error will be generated if the table is secured against writing.


# WRITE_LTAB

This command enables you to write new values to the specified user table.

The format of the command is as follows:
```
WRITE_LTAB 'User table name'
(Row-number Column-number) or (TITLE Title-string-number)
Value
```

An example of the command is given below:
```
WRITE_LTAB  'usertable1'
```

```
3 5
26.1
```

`WRITE_LTAB` requires three pieces of data.

The first is the name of the user table for the command. The second is the position in the specified user table to which the value is to be written, and the third is the value itself.

In the above example, the value `26.1` is to be written to the position row `3` and column `5` in the user table `usertable1`.

You can specify the position in another way, for example, `TITLE 2`, which means title string 2.

You can specify a 'text string' or a number as a value, for example, `'WHEEL'` or `26.1` which are both valid. Also, you can specify a variable name or another command as the value, for example, `RADIUS`, which means the value of variable `RADIUS` is the value specified, or `(READ_LTAB 'logtable1' 2 3)`, which means the output from this command is the value specified.

# Using Logical and Display Tables— Example 1

There are some basic steps you need to take to use logical and display tables.

In general, you have to define the required logical table if it is a user table, that is, a user-defined logical table as opposed to a system-defined logical table, which you do not need to define. Then, you also have to define the required display table so that you can map it to the required logical table to view the data in this logical table.

Once the logical and display tables are defined, you can use the commands and functions provided to interact with them.

The following sections give examples to show you how to define a user table and a display table, and how to interact with them.

---

### 📋 Note

Refer to the sections "Logical Table Access Functions", "Display Table Functions" and "User Table Functions" for details of the commands and functions used in the sections below.

---

## Defining a User Table

There are two main steps to define a user table:

- Create the user table with a specified name and an estimate of its size in rows and columns.
- Fill the user table with the required data.

The following is an example listing to define a user table:

```
{
 This is the Macro 'UTABLE1' to define a user table called 'Mach_Op'.
 It then fills the user table with data in the title string 1 and
 from rows 1 to 5 in column 1.
}

DEFINE UTABLE1
    CREATE_LTAB 5 2 'Mach_Op'

    WRITE_LTAB 'Mach_Op' 1 1 'Type of Machining Operation'
    WRITE_LTAB 'Mach_Op' 2 1 'Machine Tool Parameters'
    WRITE_LTAB 'Mach_Op' 3 1 'Cutting Tool Parameters'
    WRITE_LTAB 'Mach_Op' 4 1 'Workpart Characteristics'
    WRITE_LTAB 'Mach_Op' 5 1 'Other Operating Parameters'
    WRITE_LTAB 'Mach_Op' TITLE 1 'Characteristics of a Machining Operation'
END_DEFINE
```

### 📝 Note

The text within the { } brackets in the above and subsequent listings are only comments and not a part of the macro program.

You can use a text-editor to create the above listing and save it in a file `utable1.mac`.

## Defining a Display Table

There are three main steps to define a display table:

- Configure the layout of the display table, which includes the title and the data-column areas
- Specify the data in the title area
- Specify the data in the data-column area.

The following is an example listing to define a display table:

```
{
 This is the Macro 'DTABLE1' to define a display table called 'Mach_Op'
 which maps to the user table 'Mach_Op'. The user table and display table
 are called by the same name 'Mach_Op' in this example, but they can be
 different.

 It then specifies the layout of the title and data areas of the display
```

*Creo Elements/Direct Drafting Writing Macros*

```
 table, and also specifies the data and their formats to put into the
 title and data areas.
}

DEFINE DTABLE1
  TABLE_LAYOUT  'Mach_Op'
    'Mach_Op'
    WHITE BLACK
    width 60.250000 rows 10.058824
    FRAME_WIDTH 2
    HORIZONTAL WHITE SOLID
    VERTICAL WHITE SOLID
    SCROLL_BAR WHITE BLUE 32
    TITLE_LAYOUT
    18 '                                                           '
    1  '                                                           '
    END
    COLUMN_LAYOUT
        '                                                         '
  END
  TABLE_TITLE 'Mach_Op'
    BLACK YELLOW '@s1' ''  1 1
    WHITE WHITE '' ''  2 1
  END
  TABLE_COLUMN 'Mach_Op'
    COLUMN 1 GREEN BLACK 1 FORMAT 10 LEFT '@v1'

  END
END_DEFINE
```

You can use a text-editor to create the above listing in a file. However, if you already have a display table with a similar layout to that required, it is easier to save the listing of that display table in a file using the `SAVE_TABLE` command. Then you use a text-editor to edit the file to the required layout.

Once you have created this listing, you save it in a file `dtable1.mac`.

## Interacting with Display Table

You first have to run Creo Elements/Direct Drafting on your system. Then, before you can interact with the user and display tables, you have to load their definitions by entering the following commands at the keyboard in response to the ENTER COMMAND prompt in Creo Elements/Direct Drafting:

```
INPUT 'utable1.mac' [Return]
UTABLE1 [Return]
INPUT 'dtable1.mac' [Return]
DTABLE1 [Return]
```

The first two commands load the definition of the user table, and the second two load that of the display table.

You can now start to interact with the user and display tables. First, you can show the display table `Mach_Op` on the screen by:

`SHOW_TABLE ON 'Mach_Op' [Return]`

Then, you move the display table `Mach_Op` to somewhere in the middle of the screen by:

`MOVE_TABLE 'Mach_Op' LOWER LEFT 0,0 300,300 END [Return]`

## 📝 Note

It is important to use the options `LOWER LEFT` in the command to make sure that the move starts from the lower-left-hand corner of the screen. Otherwise, the move starts from the current position of the table. This means that the table can possibly move outside the screen.

Then, you print the display table `Mach_Op` as it appears on the screen in a file `dtable1.prt` by:

`PRINT_TABLE 'Mach_Op' 'dtable1.prt' [Return]`

You can then print that file on a printer.

## 📝 Note

This is a very useful command if you want to keep a hardcopy of the display table as it appears on the screen.

Then, you save the display table `Mach_Op` in a file `dtable1.sav` by:

`SAVE_TABLE 'Mach_Op' 'dtable1.sav' [Return]`

## 📝 Note

This is a very useful command when you want to create a layout for another display table. If there is already a display table with a similar layout to that required, you can use this command to save that layout in a file and then edit it with a text-editor.

Then, you remove the display table `Mach_Op` from the screen by:

`SHOW_TABLE OFF 'Mach_Op' [Return]`

# Using Logical and Display Tables—Example 2

This section is similar to the section "Using Logical and Display Tables - Example 1", except the example here is more sophisticated. This example uses two user tables and two display tables. The first user and display tables come from the example in "Using Logical and Display Tables - Example 1" The second user and display tables are new and defined in this section.

The following sections give examples to show you how to define the required user and display tables, and how to interact with them.

## 🗐 Note

Refer to the sections "Logical Table Access Functions", "Display Table Functionalities" and "User Table and Its Functionalities" for details of the commands and functions used in the sections below.

## Defining the First User and Display Tables

The first user and display tables are the same as those used in "Using Logical and Display Tables—Example 1" with some minor changes, so you can copy those files `utable1.mac` and `dtable1.mac` to `utable21.mac` and `dtable21.mac` respectively. Then, you can edit them to the required layout with a text-editor.

The following listing is the first user table in the file `utable21.mac`:

```
{
 This is the Macro 'UTABLE21' to define the first user table called
 'Mach_Op2'. It then fills the user table with data in the title
 string 1 and from rows 1 to 5 in column 1.
}

DEFINE UTABLE21
    CREATE_LTAB 5 2 'Mach_Op2'

    WRITE_LTAB 'Mach_Op2' 1 1 'Type of Machining Operation'
    WRITE_LTAB 'Mach_Op2' 2 1 'Machine Tool Parameters'
    WRITE_LTAB 'Mach_Op2' 3 1 'Cutting Tool Parameters'
    WRITE_LTAB 'Mach_Op2' 4 1 'Workpart Characteristics'
    WRITE_LTAB 'Mach_Op2' 5 1 'Other Operating Parameters'
    WRITE_LTAB 'Mach_Op2' TITLE 1 'Characteristics of a Machining Operation'
END_DEFINE
```

This macro UTABLE21 is the same as the macro UTABLE1, except the user table name is called Mach_Op2.

The following listing is the first user table in the file dtable21.mac:

```
{
 This is the Macro 'DTABLE21' to define the first display table called
 'Mach_Op2' which maps to the user table 'Mach_Op2'. The first user and
 display tables are called by the same name 'Mach_Op2' in this example,
 but they can be different.

 It then specifies the layout of the title and data areas of the display
 table, and also specifies the data and their formats to put into the
 title and data areas.
}

DEFINE DTABLE21
  TABLE_LAYOUT  'Mach_Op2'
    'Mach_Op2'
    WHITE BLACK
    width 60.250000 rows 10.058824
    FRAME_WIDTH 2
    HORIZONTAL WHITE SOLID
    VERTICAL WHITE SOLID
    SCROLL_BAR WHITE BLUE 32
    TITLE_LAYOUT
    18 '                                                  |     '
    1  '                                                        '
    END
    COLUMN_LAYOUT
       '                                                        '
  END
  TABLE_TITLE 'Mach_Op2'
    BLACK YELLOW '@s1' ''  1 1
    BLACK YELLOW 'END' 'SHOW_TABLE OFF 'Mach_Op2' SHOW_TABLE OFF 'Op_Para' END'  1 2
    WHITE WHITE '' ''   2 1
  END
  TABLE_COLUMN 'Mach_Op2'
    COLUMN 1 GREEN BLACK 1 FORMAT 10 LEFT '@v1'
  END
END_DEFINE
```

This macro DTABLE21 is the same as DTABLE1 except:

- this display table is called `Mach_Op2` and mapped to a user table called `Mach_Op2`
- there are two columns in the first title row
- the second column in this title row contains the text `END` and the associated action text is:
  ```
  SHOW_TABLE OFF 'Mach_Op2' SHOW_TABLE OFF 'Op_Para' END
  ```

  which means removing the display tables `Mach_Op2` and `Op_Para` from the screen, and aborting the current macro program.

## Defining the Second User and Display Tables

The second user and display tables are new, so you have to define them. The following listing defines the second display table:

```
{
 This is the Macro 'DTABLE2' to define the second display table called
 'Op_Para' which maps to an unknown user table ''. Mapping
 this display table to a user table is done later when the second user
 table is defined.

 It then specifies the layout of the title and data areas of the display
 table, and also specifies the data and their formats to put into the
 title and data areas.
}

DEFINE DTABLE2
  TABLE_LAYOUT  'Op_Para'
    ''
    WHITE BLACK
    width 60.250000 rows 10.058824
    FRAME_WIDTH 2
    HORIZONTAL WHITE SOLID
    VERTICAL WHITE SOLID
    SCROLL_BAR WHITE BLUE 32
    TITLE_LAYOUT
    18 '                                  |          '
    1  '                                             '
    END
    COLUMN_LAYOUT
        '                                            '
  END
  TABLE_TITLE 'Op_Para'
    BLACK YELLOW '@s1' ''  1 1
    BLACK YELLOW 'OFF' 'SHOW_TABLE OFF' 'Op_Para'  1 2
    WHITE WHITE '' ''  2 1
  END
  TABLE_COLUMN 'Op_Para'
    COLUMN 1 GREEN BLACK 1 FORMAT 10 LEFT '@v1'
  END
```

```
END_DEFINE
```

This macro `DTABLE2` is the same as `DTABLE21` except:

- this display table is called `Op_Para` and is not mapped to a user table until later when the second user table is defined.

- the second column in first title row contains the text `OFF` and the associated action text is:
  ```
  SHOW_TABLE OFF 'Op_Para'
  ```

  which means removing the display table `Op_Para` from the screen.

Once you have created this listing, you save it in a file `dtable2.mac`.

The following listing defines the second user table and contains macro commands to interact with all four tables:

```
{
 This is the macro 'UTABLE2' to define the second user table called 'Op_Para'
 and to provide the Macro commands to interact with all four tables.
}

DEFINE UTABLE2
LOCAL OPARA

MOVE_TABLE 'Mach_Op2' LOWER LEFT 0,0 300,300 END
POP_UP_LTAB 'Mach_Op2'
CREATE_LTAB  6 2 'Op_Para'
CONNECT_TABLE 'Op_Para' 'Op_Para'

LOOP

READ STRING 'Select option from CHARACTERISTICS OF A MACHINING OPERATION table:' OPARA

LET OPARA (UPC OPARA)

IF(OPARA="TYPE OF MACHINING OPERATION")
  WRITE_LTAB 'Op_Para' TITLE 1 'Type of Machining Operation:'
  WRITE_LTAB 'Op_Para' 1 1 'Turning'
  WRITE_LTAB 'Op_Para' 2 1 'Drilling'
  WRITE_LTAB 'Op_Para' 3 1 'Tapping'
  WRITE_LTAB 'Op_Para' 4 1 'Milling'
  WRITE_LTAB 'Op_Para' 5 1 'Boring'
  WRITE_LTAB 'Op_Para' 6 1 'Grinding
ELSE_IF(OPARA="MACHINE TOOL PARAMETERS")
  WRITE_LTAB 'Op_Para' TITLE 1 'Machine Tool Parameters:'
  WRITE_LTAB 'Op_Para' 1 1 'Size and Rigidity'
  WRITE_LTAB 'Op_Para' 2 1 'Horsepower'
  WRITE_LTAB 'Op_Para' 3 1 'Spindle Speed and Feedrate Levels'
  WRITE_LTAB 'Op_Para' 4 1 'Conventional or NC'
  WRITE_LTAB 'Op_Para' 5 1 'Accuracy and Precision Capabilities'
  WRITE_LTAB 'Op_Para' 6 1 'Operating Time Data'
```

```
ELSE_IF(OPARA="CUTTING TOOL PARAMETERS")
  WRITE_LTAB 'Op_Para' TITLE 1 'Cutting Tool Parameters:'
  WRITE_LTAB 'Op_Para' 1 1 'Material Type'
  WRITE_LTAB 'Op_Para' 2 1 'Material Composition'
  WRITE_LTAB 'Op_Para' 3 1 'Physical and Mechanical Properties'
  WRITE_LTAB 'Op_Para' 4 1 'Type'
  WRITE_LTAB 'Op_Para' 5 1 'Geometry'
  WRITE_LTAB 'Op_Para' 6 1 'Cost'
ELSE_IF(OPARA="WORKPART CHARACTERISTICS")
  WRITE_LTAB 'Op_Para' TITLE 1 'Workpart Characteristics:'
  WRITE_LTAB 'Op_Para' 1 1 'Material Type'
  WRITE_LTAB 'Op_Para' 2 1 'Hardness of Material'
  WRITE_LTAB 'Op_Para' 3 1 'Geometric Size and Shape'
  WRITE_LTAB 'Op_Para' 4 1 'Tolerances'
  WRITE_LTAB 'Op_Para' 5 1 'Surface Finish'
  WRITE_LTAB 'Op_Para' 6 1 'Initial Surface Condition'
ELSE_IF(OPARA="OTHER OPERATING PARAMETERS")
  WRITE_LTAB 'Op_Para' TITLE 1 'Other Operating Parameters:'
  WRITE_LTAB 'Op_Para' 1 1 'Depth of Cut'
  WRITE_LTAB 'Op_Para' 2 1 'Cutting Fluid'
  WRITE_LTAB 'Op_Para' 3 1 'Workpart Rigidity'
  WRITE_LTAB 'Op_Para' 4 1 'Fixtures and Jigs'
  DELETE_LTAB_ROW 'Op_Para' 6
  DELETE_LTAB_ROW 'Op_Para' 5
ELSE
  BEEP
  DISPLAY "UNKNOWN OPTION"
END_IF
SHOW_TABLE ON 'Op_Para'
END_LOOP
END_DEFINE
```

Once you have created this listing, you save it in a file `utable2.mac`.

Although the above listing is essentially to define the second user table, it also provides macro commands to interact with all four tables.

Below is an explanation of the macro program:

- `DEFINE UTABLE2` marks the beginning of this macro `UTABLE2`.

- `LOCAL OPARA` declares a local variable `OPARA`.

- `MOVE_TABLE 'Mach_Op2' LOWER LEFT 0,0 300,300 END` moves the display table `Mach_Op2` to position 300,300 on the screen.

- `POP_UP_LTAB 'Mach_Op2'` displays the user table `Mach_Op2`.

- `CREATE_LTAB 6 2 'Op_Para'` defines a user table `Op_Para` of six rows and two columns in size.

- `CONNECT_LTAB 'Op_Para' 'Op_Para'` connects the display table `Op_Para` to the user table `Op_Para`.

- `LOOP` marks the beginning of the loop whose end is marked by `END_LOOP`.

- Within this loop, it first asks the user to select an option from the table by:
  ```
  READ STRING 'Select option from CHARACTERISTICS OF A MACHINING
  OPERATION table:' OPARA
  ```
  The user can select an option from the specified table and the associated action text is taken as the input to the variable `OPARA`.

- `LET OPARA (UPC OPARA)` converts the input text to upper case.

- Then, there are `IF ELSE_IF` and `END_IF` statements to check the contents of the variable `OPARA`, which can be one of five possible values. Depending on its value, the appropriate data is written to the user table `Op_Para`.

- You notice the two `DELETE_LTAB_ROW` statements which you need to delete the data left from the previous selection in rows 5 and 6 of the user table `Op_Para`. Otherwise, this data will also be displayed in the display table `Op_Para`.

- If `OPARA` contains none of the possible values, a message `UNKNOWN OPTION` is displayed.

- END_IF marks the end of the IF statement.

- SHOW_TABLE ON 'Op_Para' displays the display table Op_Para.

- END_LOOP marks the end of the LOOP statement.

- END_DEFINE marks the end of the DEFINE statement.

## Interacting with the User and Display Tables

You first have to run Creo Elements/Direct Drafting on your system. Then, before you can interact with the user and display tables, you have to load their definitions by entering the following commands at the keyboard in response to the `ENTER COMMAND` prompt in Creo Elements/Direct Drafting:

```
INPUT 'utable21.mac' [Return]
UTABLE21 [Return]
INPUT 'dtable21.mac' [Return]
DTABLE21 [Return]
INPUT 'dtable2.mac' [Return]
DTABLE2 [Return]
INPUT 'utable2.mac' [Return]
```

The first four commands load the definition of the first user and display tables, and the second three load that of the second display and user tables.

You can now run the user and display tables using the command:

```
UTABLE2 [Return]
```

The macro program `UTABLE2` first displays the `Characteristics of a Machining Operation` table and the prompt `Select option from CHARACTERISTICS OF A MACHINING OPERATION table:`. You can select any of the five options in that table, and a second table is displayed to show the options under the option selected.

Now, there are two display tables on the screen and the macro program loops back to prompt for another input. You can try all five options and see how the second display table changes its contents accordingly. You can also try an empty option, and a message `UNKNOWN OPTION` is displayed.

You notice there is a text `OFF` on the upper-right-hand corner of the second display table. If you select that, the second display is removed from the screen. However, the macro program is still running and prompts for another input. If you select an option from the first display table, the second display table appears again with the appropriate contents as selected.

You also notice there is a text `END` on the upper-right-hand corner of the first display table. If you select that, both the first and second display tables are removed from the screen, and the macro program ends.

## Comments

This example provides an indication of what you can do with logical and display tables. You can take the idea further to develop more sophisticated display and logical tables, and use more than two levels of tables.

# Index

connecting display to logical table, 195

CONNECT_TABLE command, 213

connecting display to logical table concept, 195

construction lines, macro, 104

control statements, 47

coordinates, 66

copying text, 28

CREATE_LTAB, 107

CREATE_LTAB command, 220

current environment, 61

current line, 29

## D

data files, reading, 95

debugging a macro, 20

defensive programming, 56

defining display table commands, 203

defining user table commands, 219

definition, macro, 36

Delete key, 26

DELETE_LTAB command, 221

DELETE_LTAB_ROW command, 222

DELETE_MACRO, 20

DELETE_TABLE command, 214

descriptor, file, 83, 85

diagrams, syntax, 39

display table, 195, 203
   components, 195
   defining (example 1), 226
   functions, 203
   interacting with (example 1), 227
   introduction, 193

DISPLAY_NO_WAIT function, 40

drawing, saving, 18

## E

ECHO for creating macros, 111

ECHO function, 109-110

EDIT_FILE command, 19

EDIT_PORT command, 26

editing keys, 26

editor, 25
   for writing macros, 25
   keyword, 29
   leave without saving, 19
   marker, 29
   string, 29

editor commands, 25, 29
   Adjust Center, 30
   Adjust Fill, 30
   Adjust Justify, 30
   Copy, 30
   Delete, 30
   Load, 30
   Move, 30
   Next, 31
   Overwrite, 31
   Replace, 31
   Set Escape, 31
   Set Left Margin, 32
   Set Marker, 32
   Set Right Margin, 32
   Write, 32

efficient code, 57

END command, 76
   and recursive loop, 52

End key, 26

END_DEFINE function, 38

end-of-file marker, 85

environment, current, 61

ESC key, leave editor without saving, 19

example 1
   defining display table, 226
   defining user table, 225
   interacting with display table, 227
   using logical and display table, 225

example 2

## S

save macro, Ctrl-D, 19
SAVE_LTAB command, 201
SAVE_TABLE command, 216
saving a drawing, 18
screening bad input, 57
SCROLL_LTAB command, 201
SECURE_LTAB command, 223
SECURE_TABLE command, 217
SELECT_FROM_LTAB command, 202
sequence of execution, 47
set marker command, 27
setting markers, 27
SHOW_TABLE command, 217
simple code
    defensive programming, 56
single quotes
    for strings, 36
SORT_LTAB command, 223
splitting a line, macro, 104
stopping a macro, 23
STORE command, 39
store macro, Ctrl-D, 19
storing a macro, 19
STR function, 40
string concatenation, 40
SUBSTR function, 85
subtraction of vectors, 69
syntax diagrams, 39
system array, 62

## T

TABLE_COLUMN
    format, 204
TABLE_COLUMN command, 204
TABLE_LAYOUT
    format, 207
TABLE_LAYOUT command, 207
TABLE_SCROLL_STEP command, 218

TABLE_TITLE
    format, 211
TABLE_TITLE command, 211
TECHO function, 109
title string, 193
token, parentheses with, 51
trace
    file, 53
    program, 52
true, boolean expression, 50

## U

upper case, 36
user input, 37
user table, 193
    defining (example 1), 225
    functions, 219
using display table
    commands, 211
using logical and display table
    example 1, 225
using user table
    commands, 219

## V

VAL function, 99
variable
    names, conflicting, 45
    types, 46
variable names
    defensive programming, 56
variables
    global, 40, 90
    local, 37, 40, 45
vectors, 67
    addition, 67
    subtraction, 69
vectors, for a spigot, 98

## W

WAIT function, 40
WHILE ... END_WHILE construct, 47
WRITE_FILE function, 85
WRITE_LTAB, 107
WRITE_LTAB command, 224

## X

X_OF operator, 66

## Y

Y_OF operator, 66

## Z

Z-levels, viewing, 107