Creo Elements/Direct Drafting 写入宏

Creo Elements/Direct Drafting 20.3.0.0

© 2020 PTC Inc. 和/或其子公司版权所有。保留所有权利。

PTC Inc. 及其子公司 (通称 "PTC") 的用户和培训文档受美国和其他国家/地区版权法的保护，并受许可协议的约束，复制、公开发行和使用此文档受到严格限制。PTC 在此同意，依据适用软件的许可协议规定，允许拥有软件使用权的用户以印刷形式复制本文档 (如果在软件媒介中提供)，但仅限内部/个人使用。任何复印件都应包括 PTC 版权通告和由 PTC 提供的其他专利通告。未经 PTC 明确书面许可，不得复制培训材料。未经 PTC 事先书面许可，本文档不得公开、转让、修改或简化为任何形式 (包括电子媒介)，也不允许以任何手段传播、公开发行或出于此目的进行复制。此处所描述的信息仅作为一般信息提供，如有更改恕不另行通知，并且不能将其解释为 PTC 的担保或承诺。本文档中如有错误或不确切之处，PTC 概不负责。

本文档中所述软件在有书面许可协议的条件下提供，其中包括重要的商业秘密和专利信息，并受美国和其它国家/地区版权法的保护。未经 PTC 事先书面许可，本软件不能以任何形式在任何媒介中复制或分发、公开至第三方，或者以任何软件许可证协议所不允许的方式使用。

未经授权使用软件或其文档，将会引起民事赔偿和刑事诉讼。

PTC 将软件盗版视为犯罪，而且我们据此来对待盗版者。我们不会容忍对 PTC 软件产品的盗版行为，我们会使用包括公私两种监督资源在内的一切可用法律手段来追查盗版者的 (民事和刑事) 责任。作为其中的一项防盗版举措，PTC 使用数据监控及净化技术来获取和传送对我们的软件进行非法复制的用户的数据。对于从 PTC 及其授权分销商处获取了合法许可软件的用户，我们不会收集他们的数据。如果您在使用我们软件的非法副本，但不同意我们收集和传送此类数据 (包括美国)，请停止使用此非法版本，然后与 PTC 联系以获取合法的许可版本。

若需了解重要的版权、商标、专利和许可信息：请参阅您的 PTC 软件的“关于”对话框或版权通告。

美国政府的权利

PTC 软件产品和软件文档属于 48 C.F.R. 2.101 中规定的“商品”。依照针对非军事机构的美国联邦采购条例 (FAR) 12.212 (a)-(b) (计算机软件) (2014 年 5 月) 以及针对国防部的美国国防联邦采购条例补充 (DFARS) 227.7202-1(a) (政策) 和 227.7202-3 (a) (商业计算机软件或商业计算机软件文档中的权利) (2014 年 2 月)，PTC 软件产品和软件文档根据 PTC 商业许可协议提供给美国政府。美国政府只能按照适用的 PTC 软件许可协议中规定的条款和条件来使用、复制或公开发布 PTC 软件产品和软件文档。

PTC Inc., 121 Seaport Blvd, Boston, MA 02210 USA

目录

前言 6

什么是宏？ 13

为什么要使用宏？ 14

为宏创建文件 15

存储宏 15

删除宏 16

运行宏 16

调试宏 16

停止宏 18

使用编辑器编写宏 19

使用键盘编辑键 20

如何进入和离开编辑器 20

使用 EDIT_PORT 快速进入编辑器......20

如何设置和使用标记 21

复制文本 22

使用编辑器命令 23

使用 EDIT_MACRO......26

宏基础知识 28

宏由哪些部分组成？ 29

最小宏 31

语法图 31

解释局部变量 33

为什么使用局部变量？ 37

我们是否声明了变量类型？ 38

使用控制语句 39

使用括号 42

使用追踪工具 44

缩进宏的行......48

防御性程序设计 48

宏命令 49

内置运算 50

查询环境和元素......51

使用 INQ_ENV......52

使用 INQ_ELEM......53

使用 GETENV 54
使用其他查询 54

快速查看点和矢量 55
点 56
矢量 56

编写几何宏 60
箭头宏 61
面板宏 64

文件输入/输出和文本字符串 69
宏的功能 70
分析宏 71
从宏的内部调用宏 77
向宏传递参数 79

使用数据文件中存储的尺寸 82
宏的功能 83
介绍栓 83
矢量分析 84
介绍数据文件 85
分析宏 86
细化宏 88

有用的宏 89
绘制与现有线成角度的构造线 90
将线分割为相等段 90
绘制圆端槽 91
绘制多边形 92
围绕圆形对象调整文本 92
显示隐藏线绘图的不同 Z 级 93

记录系统操作 95
ECHO 函数 96
使用 ECHO 创建宏 96

使用界面查找命令 99
您会使用哪些命令？ 100

自定义 102
什么是 Creo Elements/Direct Drafting 环境？ 103
自定义 Creo Elements/Direct Drafting 环境 104
屏幕菜单的创建方式 104
自定义屏幕菜单 108
自定义本地目录 109
自定义键盘 112

什么是文本字体? 114
如何创建文本字体 118
自定义启动过程 121
自定义剖面线图案 124
键盘输入字符 125
键盘输入字符 126
命令和函数的简短描述 127
附录 A.逻辑表和显示表 177
什么是逻辑表和显示表？179
逻辑表访问函数 182
显示表函数 189
使用表函数 203
使用逻辑表和显示表 - 示例 1 210
使用逻辑表和显示表 - 示例 2 213
索引 220

前言

Creo Elements/Direct Drafting 是通用的 2D 设计和绘制系统，用于优化设计过程的每个阶段。借助 Creo Elements/Direct Drafting，可以轻松快捷地创建和修改 2D 绘图。

本手册旨在介绍如何编写用于 Creo Elements/Direct Drafting 的宏。宏可以加速日常任务并使其实现自动化。

本手册的读者

如果您为以下角色，请阅读本手册：

- 系统管理员

  您是一名经验丰富的系统管理员，具有管理基于 UNIX 和 Windows 的操作系统的经验。非常熟悉启动文件和自定义文件。将使用宏来简化自定义过程以供组使用。

- 设计者或绘图者

  您是一位经验丰富的 Creo Elements/Direct Drafting 用户。需要手动执行许多重复任务，想要了解如何通过编写宏来自动化这些任务。之前可以没有编程经验。

本手册的用途

本手册将介绍如何执行以下操作：

- 创建几何。
- 从数据文件中提取规范。
- 使用工程程序语言 (例如 Pascal 或 C) 来进行复杂的计算。(仅适用于基于 UNIX 的系统)
- 标注绘图。
- 创建零件列表。
- 自定义启动过程。
- 控制屏幕菜单显示和功能。
- 自定义键盘。
- 自定义文本字体。

本手册的使用说明

系统管理员

这里是一些如何使用本手册的建议：

1. 有关编写宏的一般信息，请阅读第 1 至 4 章。
2. 如果要使用用户界面将宏命令或函数与您手动执行的任务关联，请阅读第 12 章。
3. 有关自定义信息，请阅读第 13 章。
4. 有关命令或功能的简要说明，请参阅第 14 章。
5. 有关逻辑表和显示表信息，请阅读附录 A。

设计者或绘图者

这里是一些如何使用本手册的建议：

1. 有关编写宏的一般信息，请阅读第 1 至 4 章。
2. 要刷新矢量内存，请阅读第 5 章。
3. 有关正常工作中使用的任务的特定详细信息，请阅读第 6 章至 11 章。
4. 如果要使用用户界面将宏命令或函数与您手动执行的任务关联，请阅读第 12 章。
5. 如果要自定义自己的环境，请阅读第 13 章。
6. 有关命令或功能的简要说明，请阅读第 14 章。

本手册的编排方式

第 1 章	What is a Macro?包含宏简介，并介绍如何存储、运行和调试宏。
第 2 章	Using the Editor to Write a Macro介绍如何使用内置编辑器编写宏。
第 3 章	Macro Basics介绍如何使用语法图、局部与全局变量、控制语句、括号和防御式编程。
第 4 章	Inquiring about the Environment and Elements 介绍如何访问环境信息 (例如当前单位和当前线类型)，以及如何恢复在操作宏期间发生更改的任何环境信息。
第 5 章	Quick Review of Points and Vectors介绍您所需的宏。
第 6 章	Writing Geometry Macros介绍如何执行初始矢量分析以及如何将分析融入宏中。
第 7 章	File Input/Output and Text Strings介绍如何处理字符串数据以及如何从 ASCII 文件提取字符串。
第 8 章	Using Dimensions Stored in a Data File介绍如何使用文件中的表格数据创建几何。
第 9 章	Useful Macros介绍可在日常工作中使用的宏。
第 10 章	Recording the System Operation 介绍如何记录 Creo Elements/Direct Drafting 操作的活动，以及如何利用记录编写宏。
第 11 章	Using the Interface to Find a Command介绍如何将宏命令与用户界面上执行的任务关联。
第 12 章	Customizing介绍如何自定义启动、环境、键盘和文本字体等内容。
第 13 章	Brief List of Commands and Functions为每个命令和函数提供 1 至 2 行介绍。
附录 A	Logical and Display Tables提供使用逻辑表和显示表所需的信息。

在线帮助

有关所有命令和功能的完整说明和语法，请使用在线帮助功能。

例如，要获得 `MODIFY` 命令的帮助，请在应用程序命令行中输入：

```
help modify
```

将会清屏，同时 Creo Elements/Direct Drafting 将显示有关修改绘图的详细信息。

印刷规定

本手册采用以下印刷规定：

表 **1**。本手册中采用的规定

规定	表示
加粗	用户界面中的菜单路径、对话框选项、按钮以及其他可选择的元素。 例如：线栅格、显示层以及缩放等等。
`Courier`	用户输入、系统消息、目录和文件名。

1

什么是宏？

为什么要使用宏？ 14
为宏创建文件 15
存储宏 15
删除宏 16
运行宏 16
调试宏 16
停止宏 18

本章将简要介绍宏及如何存储、运行和停止宏。

为什么要使用宏？

宏是用于自动执行一系列命令的快速且容易的方法。只需少量或不需要用户输入。您应该考虑为 ME-CAD 用户常用的任何命令序列编写宏。

例如，您每天至少从工作站注销一次。在注销前，您需要存储绘图，然后在命令行中键入 `exit confirm`。

您可以使用以下宏 (称为 `Quit`) 自动执行此过程：

```
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
```

您可以看到宏包含您注销前输入的一系列命令。宏开头的备注帮助用户理解宏。

命令存储于文件中并在运行宏时执行。通过在命令行中键入宏名称，或者在屏幕菜单中拾取某个槽来运行宏。

宏的一些主要应用如下所示：

- 创建几何。
- 执行计算。
- 标注绘图。
- 创建零件列表。
- 控制屏幕菜单显示和功能。

在详细讨论宏前，我们将介绍如何在内置编辑器中键入宏以及如何存储宏。

可通过两种方式创建宏：

- 使用偏好的编辑器。
- 使用内置于 ME-CAD 系统软件中的屏幕编辑器。

内置编辑器与基于 PC 的编辑器类似。本章中的所有示例将使用内置编辑器来完成。

为宏创建文件

在编写宏前，必须首先创建存储宏的文件。例如，您可能想要创建称为 cad_mac.m 的文件。从 ME-CAD 环境中，在命令行中键入以下内容：

```
EDIT_FILE 'cad_mac.m'
```

如果此文件已经存在，系统将显示该文件并且您可以添加宏。如果文件不存在，系统将创建一个空文件。如稍后第19页上的“使用编辑器编写宏”中所述键入宏。

存储宏

要存储宏，并返回到 ME-CAD 屏幕，请按 [Ctrl] [D]。文件将存储在系统磁盘中。该操作将由屏幕底部的以下消息显示：

```
writing 'cad_mac.m'
```

如果列出当前目录，'cad_mac.m' 将显示在列表中。

如果在编辑器中更改文件，您可能会决定不希望保存更改。点击 [ESC] 或 [Break] 退出编辑器，更改将不写入磁盘。

可以在文件中添加所需的任意数量的宏。宏在文件中的顺序不重要。但是按名称的字母数字顺序放置宏可简化宏的查找。

写入和调试新宏时，您可能想要在文件开头写入此新宏。使用 EDIT_FILE 命令进入编辑器时将显示文件的首页，因此您无需滚动查看文件来查找宏。稍后，在完成宏调试后，可将其移动到文件中正确的字母排序位置。

另一种写入和调试新宏的方式是为新宏创建单独的文件。当完全调试好宏后，可将其附加到正常的宏文件。此方法具有两个优点：

- 当使用 EDIT_FILE 命令时，文本的显示速度将更快。
- INPUT 命令的执行速度更快。(INPUT 命令用于编译和加载宏，此内容将在下一节讨论)。

建议在单独的文件中保存每个宏。随后可以将该文件命名为与宏同名。如果宏调用其他宏，则必须单独输入涉及的其他宏。

还有另一种存储不常用宏的方法。将类似宏组合到一个文件中。例如，所有螺栓宏可存储在文件 'bolt.m' 中，而法兰宏可存储在 'flange.m' 中。使用所有文件通用的宏名称。例如，第一个螺栓宏和第一个法兰宏都称为 macro1，第二个螺栓宏和第二个法兰宏都称为 macro2 等等。根据需要加载每个文件。每次加载文件时，同名的宏将被覆盖。例如，螺栓 macro1 会覆盖法兰 macro1 等等。这表示不常用的宏将不会占用宝贵的内存空间。

如果使用宏中的 INPUT 命令，则它必须结合限定符 IMMEDIATE 使用，另请参阅第31页上的“INPUT”。

删除宏

可以使用 DELETE_MACRO 命令从内存删除单个宏。如果需要更多内存空间来加载大绘图，则此命令非常有用。例如，如果要删除称为 Quit 的宏，可使用以下命令：

```
DELETE_MACRO Quit
```

运行宏

宏必须经过编译并加载到内存中，然后才能运行。在命令行中键入以下内容可编译和加载宏：

```
INPUT 'cad_mac.m'
```

现在您已在文件中输入宏，有两种方法可从此文件运行宏。下面介绍这两种方法。

从命令行运行宏

在命令行中键入宏名称可运行任意宏。如果文件 cad_mac.m 包含宏 Quit，可在命令行中键入以下内容来运行宏：

```
Quit
```

宏是逐行执行的。命令和语句会在执行过程中接受检查。这与 Basic 编程语言的“解释”版本类似。如果发现错误，执行将停止并将显示一条消息。

调试宏

如果宏未运行，必须使用编辑器来编辑宏。编辑宏的过程与创建宏的过程相同：使用 EDIT_FILE 进入文件，然后进行更改。

当文件在编辑器中显示时，您将看到文件的首页。使用箭头键滚动到要更改的宏。逐行检查宏直到找到错误。大多数错误都是简单错误，如缺失括号和缺失下划线。

如果无法快速找到错误，可使用 BREAKPOINT 功能调试宏。在宏代码的所需位置中插入 BREAKPOINT。编辑磁盘中的文件或者使用 EDIT_MACRO 功能均可执行此操作。当代码到达 BREAKPOINT 时，系统会转入调试模式。

默认情况下已启用断点功能。如果发现其未处于工作状态，检查是否已将 ENABLE_BREAKPOINT 设置为 ON。

要在 Creo Elements/Direct Drafting 中启用 BREAKPOINT 功能：

- 单击杂项，然后在系统组中单击断点复选框。

使用断点时如果已启用断点功能，系统将显示参数表。

如果已处于调试模式并且使用调试程序命令至少向前移动了一个令牌，BREAKPOINT 关键字将在当前位置设置断点。

如果在宏的 PARAMETER 和 LOCAL 之间插入 BREAKPOINT，调试程序将在宏的第一个命令前中断。

请注意，断点将仅针对当前的 OSD 会话设置，将不会添加到磁盘上的对应文件。

进入调试模式后，使用以下命令分步执行宏，然后隔离不执行宏的区域：

- STEP_NEXT：使用此操作分步执行宏代码，一次一个令牌。
- STEP_OUT：离开当前宏函数。
- STEP_OVER：跳过下一个令牌即使它是宏函数。
- STEP n：一步跳过几个令牌。
- SKIP n：跳过当前令牌数次。
- CONTINUE：继续执行宏直到下一个断点。
- GO：执行剩余代码，而不理会所有断点。
- REMOVE_BREAKPOINT：移除当前断点。
- ENABLE_BREAKPOINTS：启用所有断点。
- LIST_BREAKPOINTS：在编辑器中列出所有断点。

通过 BREAKPOINT 进行调试时，应注意以下事项：

- 在分步执行宏时，使用 DISPLAY 命令查看变量值。
- 所有 Creo Elements/Direct Drafting 函数都可在中断状态下执行。
- Creo Elements/Direct Drafting 命令将中断宏。
- 在调用堆栈中的任一宏修改后，由于 INPUT, EDIT_MACRO, DELETE_MACRO, or DEFINE 命令的影响，调试程序也将中断。
- 调试程序不包括用户界面。诸如当前命令、上一个令牌、下一个令牌及下一个命令等附加信息，请参阅命令提示。
- 调试程序不包括代码查看器。要在调试时查看代码，请使用外部查看器。

在找到错误并更正后，按 [Ctrl] D 返回到 ME-CAD 屏幕。文件新版本将覆盖磁盘上的旧版本。

内存中的文件编译版本保持不变。更改编译版本的唯一方法是使用 INPUT 命令将文件的新版本加载到内存中。常见错误做法是编辑文件，然后运行宏，而不使用 INPUT 命令。这将运行宏的旧版本。

键入以下命令再次输入文件：

```
INPUT 'cad_mac.m'
```

注意

如果要重复之前键入的命令，按 [PgUp] 键。如果要输入与之前命令类似的命令，按 [PgUp] 键，然后编辑该命令。

将从磁盘重新加载 cad_mac.m 的内容并覆盖内存中的旧副本。现在可以重新执行宏。在加载文件后，可以所需的频率来执行宏。

为进行汇总，在称为 cad_mac.m 的文件中编写并调试称为 Quit 的宏时，请使用三个步骤：

```
EDIT_FILE 'cad_mac.m '
INPUT 'cad_mac.m '
Quit
```

有一个用于调试宏的 trace 功能。将在第44页上的“使用追踪工具”中进行介绍。

停止宏

如果由于任意原因 (如死循环) 要停止宏，请使用 [Ctrl] [Break] 键。[Break] 的效果取决于操作环境。

注意

如果在使用工作区菜单启动 Creo Elements/Direct Drafting 或者处于图形屏幕中时按 [Break] 或 [Again]，则中断不会生效，直到系统已准备好接受输入。

有关详细信息，请参阅 Windows 文档。

2

使用编辑器编写宏

使用键盘编辑键..20
如何进入和离开编辑器..20
使用 EDIT_PORT 快速进入编辑器 20
如何设置和使用标记...21
复制文本...22
使用编辑器命令..23
使用 EDIT_MACRO...26

本章介绍如何使用内置文本编辑器编写宏。您将了解到如何：

- 从文件的一个部分复制行块到另一个部分。
- 从其他文件向当前文件加载内容。
- 查找字符串下次出现的位置。
- 用另一个文本字符串替换所有位置出现的某个文本字符串。
- 右侧对齐文本。

使用键盘编辑键

键盘上的简单编辑键在该编辑器中仍然可用。包括以下键：

- [Insert char] 或 [Ins]
- [Delete char] 或 [Del]
- [Clear line] 或 [Alt] [Del]
- [Clear display] 或 [Control] [Del]
- 箭头键：[<] [>] [^] [v]

如何进入和离开编辑器

只能在处于 ME-CAD 环境时进入编辑器。要进入编辑器，键入 EDIT_FILE 后接单引号包围的要编辑的文件的名称。例如，如果要编辑称为 'macros' 的文件，在命令行中键入：

```
EDIT_FILE 'macros'
```

如果文件存在，它将显示在编辑器屏幕上。如果不存在，编辑器屏幕将为空白。

如果要离开编辑器，并将更改保存到文件，按 [Ctrl] [D]

如果要离开编辑器，但不想将更改写入磁盘，按 [ESC] 或 [Ctrl] [Break]

使用 **EDIT_PORT** 快速进入编辑器

EDIT_PORT 命令将在 ME-CAD 屏幕上打开小视区。此视区由编辑器使用并且只在调用编辑器时显示。视区越小，进入和离开编辑器的速度就越快，因为屏幕重绘时间会缩短。

要使用命令，在命令行中键入 EDIT_PORT。您将看到提示：

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

拾取 ME-CAD 屏幕的左上角，然后拾取所需视区的斜对角。

要测试结果，在命令行中键入 EDIT_FILE 后接单引号包围的任意文件名。如果视区的大小不正确，请重试。视区大小保持有效，直到用另一个 EDIT_PORT 命令更改大小，或者直到使用 EXIT CONFIRM 终止 ME-CAD 会话。下次使用 ME-CAD 系统时，必须再次使用 EDIT_PORT。

EDIT_PORT 可用于向绘图中添加文本。此类文本通常只有几个字。如果在 ME-CAD 屏幕的左上角创建 8 cm 宽 2 cm 高的小视区，您将注意到重绘速度会非常快。

请注意 HELP_PORT 的工作方式相同。HELP_PORT 使用小视区尽量缩短屏幕重绘时间，从而加速对 help 文件的访问。

如何设置和使用标记

如果不作标记，稍后介绍的多个编辑器命令将无法正常运行。例如，如果要从文件的某个部分将文本块复制到另一部分，必须标记块的开头或结尾。

标记是用于标识行的字符，但在文件中保持不可见。系统会自动设置一些标记。

下表显示了用作标记的字符：

标记	用途
^	由系统设置来标记文件的第一行。
!	由系统设置来标记文件的最后一行。
0 . . . 9	由用户设置来标记文件的任一行。

让我们试着使用一些标记。在 ME-CAD 屏幕中，通过在命令行中输入以下内容来创建新文件：

```
EDIT_FILE 'markers'
```

在文件中键入一些行，如下所示：

```
AAAAAAAAA
BBBBBBBBBBBB
CCCCCCCCCCCC
DDDDDDDDDDDD
EEEEEEEEE
FFFFFFFFF
```

我们使行 B、C 和 D 的长度长于其他行，以便稍后能够轻松标识出它们。

现在，尝试使用系统定义的标记移动到文件的开头和结尾。操作方式如下所示。

键入 $ (不按 [Enter])。光标将移动到屏幕底部，前接 $ 符号。$ 是默认转义字符，必须在键入任意编辑器命令前键入它。

现在键入 ^ 并按 [Enter]。光标会移动到文件的第一行。

键入 $! 并按 [Enter]。光标会移动到文件的最后一行。

可以在此文件中的任意行设置标记。将光标放置在要标记的行上，然后输入：

```
$sm n
```

$ 符号是默认转义字符。sm 为设置标记命令。n 是从 0 到 9 的数字。sm 和 n 之间的空格是可选的。行将由该数字标记，但是标记不会显示在屏幕上。

例如，让我们在 D 行设置标记 1。在 D 行中移动光标，然后输入以下命令：

```
$sm1
```

现在，将光标移动到另一行，然后输入：

```
$1
```

光标将跳到 D 行。

复制文本

现在，让我们尝试复制文本块。我们要复制 B、C 和 D 三行，并将这三行放置在 E 行后。操作方法如下：

- 要定义三行的块，可以标记 B 行，然后将光标放置在 D 行上。也可以标记 D 行，然后将光标放置在 B 行上。我们已在 D 行上添加标记 1，因此让我们使用此标记。
- 要在 E 行后插入文本，请在 E 行设置标记。在 E 行中移动光标，然后输入以下命令：
  ```
  $sm2
  ```
- 将光标放置在 B 行中的任意位置。

请记住，标记是不可见的。但是如果我们能够看到标记，文件可能具有如下外观：

```
AAAAAAAAA
BBBBBBBBBBBB    * (cursor line)
CCCCCCCCCCCC
DDDDDDDDDDDD    1 (marker)
EEEEEEEEE       2 (marker)
FFFFFFFFF
```

只是为了提醒您：我们要复制 B、C 和 D 行并将这三行放置在 E 行后。现在输入以下命令：

```
$c12
```

从当前光标位置到标记 1 之间的文本块将复制到标记 2 后。文件现在具有如下外观：

```
AAAAAAAAA
BBBBBBBBBBBB
CCCCCCCCCCCC
DDDDDDDDDDDD
EEEEEEEEE
BBBBBBBBBBBB
CCCCCCCCCCCC
```

DDDDDDDDDDDDD
FFFFFFFFF

使用编辑器命令

本部分说明所有编辑器命令。说明中将用到以下术语：

- 方括号 [. . .] 中包含的项是可选的。
- 单引号 '. . .' 中包含的项表示键入的字符串必须包含在单引号中。
- 斜体项是要键入项的通用说明。例如，命令：

  ```
  $marker
  ```

  表示必须为标记键入数值，如 2。

  如另一个示例，用于加载文件的命令为：

  ```
  L 'filename'
  ```

  这表示必须在 L 后键入文件名 (在单引号中)。因此，如果要加载的文件为 cad_mac.m，则必须键入：

  ```
  L 'cad_mac.m'
  ```

- 当前行为包含光标的行。

下一个表显示编辑器命令的完整集合。在键入任何编辑器命令前，必须键入 $。光标将移动到屏幕底部，前接符号 $。请记住，$ 是默认转义字符。

命令	命令的作用结果
marker	将光标移动到包含指定标记的行。例如，输入 $ 4 可将光标移至包含标记编号 4 的行。
'string'	将光标移动到指定字符串出现的下一位置。例如，输入 $ 'SolidDesigner' 可将光标移至字符串 SolidDesigner 出现的下一个位置。
? ['keyword']	这将在说明关键字的部分显示系统 help 文件。如果输入 ? 而不带关键字，将显示介绍屏幕编辑器的部分。要返回到文件，请按 [Ctrl] [D]。
AC	调整中心。对于段落中的每一行，此命令会将文本放置在边距间的中心位置。
AF	调整填充。此命令可使段落在边距之间流动。左边将对齐，右边将参差不齐。
AJ	调整对齐。此命令可使段落在边距之间流动。左右两边将对齐。这表示字间距将不同。

注意

请勿使用 AF 或 AJ 编辑宏，因为这样将使每行与前一行的结尾连接。这些命令应仅用于编辑文本。

命令	命令的作用结果
C marker1 marker2	复制。复制当前行与 marker1 (包括) 间的所有文本行并将其插入到 marker2 后。例如，输入 $C34 将复制当前行和 marker 3 间的所有文本并将其插入 marker 4 后。
D marker	删除。删除当前行和标记 (包括) 间的所有文本行。
H ['keyword']	帮助。这与 ? 命令相同。
L 'filename'	加载。从指定文件复制所有文本行并将其插入当前行的后面。例如，输入 $L 'cad_macros' 会将文件 cad_macros 加载到当前行的后面。
M marker1 marker2	移动。移动当前行和 marker1 (包括) 间的所有文本行并将其插入 marker2 后。源和目标不得重叠。例如，输入 $M67 将移动当前行与 marker 6 间的所有文本并将这些文本放置在 marker 7 后。
N	下一个。重复上一个字符串搜索。
O 'filename' [marker]	覆盖。复制当前行和标记 (包括) 间的所有文本行，并将其写入具有指定名称的文件。如果未指定标记，将忽略光标位置并复制整个文件。任何具有指定名称的现有文件都将被覆盖。例如，输入 $O 'cad_macros' ! 会将当前行与最后一行 (包含) 间的文本复制到称为 cad_macros 的文件中。如果省略 !，将复制整个文件。

命令	命令的作用结果
R [V] ['string1'] ['string2'] [marker]	替换。用当前行与标记 (包括) 间的 string2 替换所有位置出现的 string1。V 允许您验证 string 的每处更改。如果省略标记，只会执行一处替换。如果省略任一字符串，将使用其上次使用的 string 作为默认值。例如，如果输入 $RV 'black' 'white' ! 光标将移动到 black 出现的下一个位置。然后，您可以选择用 white 替换它，或者移动到 black 出现的下一个位置。 R 替换， '' 不替换， S 中止。 按 R 用 white 替换 black。按空格键移动到 black 出现的下一个位置。按 S 中止操作。
SE 'character'	设置转义。重新定义转义字符。此字符会使光标从您的文件跳到编辑器命令行。默认转义字符为 $。 您可以使用除 0 ... 9, A ... Z, a ... z, ... ! 和 ? 以外的任何字符。新转义字符保持有效，除非重新定义转义字符或者系统关闭。 如果希望将转义字符定义为 #，则输入 $ SE '#'
SL	设置左边距。将左边距设置为当前光标位置。
SM n	设置标记。在当前行设置指定的标记。n 可以具有从 0 到 9 的任何值。SM 和 n 之间的空格是可选的。例如，输入 $ SM7 将在当前行设置标记 7。

命令	命令的作用结果
SR	设置右边距。将右边距设置为当前光标位置。
W 'filename' [marker]	写入。复制当前行和标记 (包括) 间的所有文本行，并将其写入具有指定名称的文件。如果未指定标记，则会复制整个文件。如果具有该名称的文件已存在，则将中止操作。例如，要将当前行与标记 6 间的文本复制到称为 cad_macros 的文件中，输入以下命令：`$ W 'cad_macros' 6`

注意

SL、SM 和 SR 将不会调整文本，直到使用 AF 或 AJ。请记住，AF 和 AJ 通常不与宏一起使用。

使用 **EDIT_MACRO**

在本章中，所有宏操作都是使用 EDIT_FILE 完成的。EDIT_FILE 命令很有用，因为您可以使用此命令检查任何文件；文件不需要包含宏。

EDIT_MACRO 命令只能用于宏。

我们将首先介绍 EDIT_MACRO 命令，然后将它与 EDIT_FILE 作比较。之后您便可以确定要在宏中使用哪个命令。

宏必须已经存在于内存中后，才能够使用 EDIT_MACRO 命令。将宏放到内存中有两种方式。我们将在下几段中讨论每一种方法。

对现有文件使用 **INPUT** 命令

您已熟悉此方法，因为我们在本章中使用的都是该方法。

如果使用宏中的 INPUT 命令，则它必须结合限定符 IMMEDIATE 使用，另请参阅第31页上的“INPUT”。

在命令行中键入宏

使用此方法时，您将键入 DEFINE 后接宏的名称。例如，如果宏称为 Slot_mac，则键入：

```
DEFINE Slot_mac
```

系统对提示做出响应：

```
Enter macro definition
```

现在，键入宏的其余行，一次一行。每次按 [Enter] 时，系统将通过以下提示响应：

```
Enter macro definition
```

现在，可通过在命令行中键入宏的名称来运行宏。请注意，使用此方法时不必使用 INPUT。

当宏处于内存中时，可在命令行中键入 EDIT_MACRO 后接宏的名称来编辑宏。例如，您可以键入：

```
EDIT_MACRO Slot_mac
```

宏将显示在编辑器中，编辑方法与使用 EDIT_FILE 命令时相同。要离开编辑器，请按 [Ctrl] [D]。要运行宏，请在命令行中键入宏名称。您不必使用 INPUT 命令。

请注意，如果使用 EDIT_MACRO 进入编辑器，按 [Ctrl] [D] 则不会保存宏。要保存宏，请在命令行中键入 SAVE_MACRO 后接宏的名称。

系统对提示做出响应：

```
Enter SCREEN, DEL_OLD, APPEND or 'file name'
```

如果要在编辑器中查看宏，则键入 SCREEN。键入 DEL_OLD 将覆盖同名的现有文件。键入 APPEND 宏将添加到现有文件的末尾。对于新文件，键入“文件名”。

比较 **EDIT_FILE** 和 **EDIT_MACRO**

如果需要几次尝试来调试宏，EDIT_MACRO 是最佳选择，因为这样不必在每次更改后使用 INPUT。

EDIT_MACRO 的缺点在于，当使用 [Ctrl] [D] 退出编辑器时不会自动保存宏。如果使用 EXIT CONFIRM 离开 Creo Elements/Direct Drafting 环境，则宏可能会丢失。

3

宏基础知识

宏由哪些部分组成？ 29
最小宏 31
语法图 31
解释局部变量 33
为什么使用局部变量？ 37
我们是否声明了变量类型？ 38
使用控制语句 39
使用括号 42
使用追踪工具 44
缩进宏的行 48
防御性程序设计 48
宏命令 49
内置运算 50

本章将介绍宏的结构、展示局部变量与全局变量间的差异、说明何时使用括号以及展示如何使用`trace`工具调试宏。

宏由哪些部分组成？

宏可由以下六部分组成：

- DEFINE 宏名称
- 参数
- 局部变量
- 用户输入
- 宏主体
- END_DEFINE

宏不必包含所有这些部分。在以下示例中，可看到六部分中的每一部分：

```
DEFINE Circle_mac
PARAMETER P2
LOCAL P1
  READ 'Indicate center of circle' P1
  CIRCLE CENTER P1 P2 END
END_DEFINE
```

在命令行中键入 Circle_mac，后接圆的半径值，可激活此宏。显然此宏的用途并不大，但仅在短短几行中便可显示出宏的所有部分。

DEFINE

此行始终以单词 DEFINE 开头，后接宏的名称。在我们的示例中，宏的名称为 Circle_mac：

```
DEFINE Circle_mac
```

编写宏时请注意大小写的使用。我们使用大小写来区分宏名称和变量，以及命令和函数。

关键字 (例如命令和函数) 全部大写。例如，DEFINE、LOCAL 和 END_DEFINE 就是关键字。

系统接受以大写或小写形式键入的关键字，但不接受以大小写混合形式键入的关键字。例如，系统接受 DEFINE 或 define，但不接受 Define、dEFINE、DeFiNE 或其他任何大小写混合形式的版本。

宏名称的首字母通常大写。因此，如果诸如 Circle_mac 的术语出现在宏列表中，您将知道 Circle_mac 是宏而不是关键字。

本章中，我们将始终正确使用大小写，但您可随意地以任何方便的形式键入。例如，我们可能会要求您在命令行中键入：

```
EDIT_FILE 'cad_macros'
```

此处，您可以键入：

```
edit_file 'cad_macros'
```

注意

单引号内任何字符串均必须完全按照如下所示进行键入。

参数

本部分将对作为参数传递至宏的变量进行定义。如果有多个参数，必须按照特定的序列列出。有关参数的详细信息，请参阅第69页上的“文件输入/输出和文本字符串”。我们的示例宏中只有一个参数语句：

```
PARAMETER P2
```

局部变量

此处定义的是仅存在于宏内的变量。在宏的用户输入部分或宏的主体中出现的任何变量，通常均定义为局部变量。我们将在本章后面部分，对局部变量进行更详细的讨论。我们之前的示例中只有一个局部变量：

```
LOCAL P1
```

用户输入

处理器编译期间未知的变量，用户一定要在运行时提供。以下示例中，P1 的值由用户提供：

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

系统将显示提示：

```
Indicate center of circle
```

用户输入的下一个值将分配给变量 P1。用户可键入点的 x,y 坐标，或者拾取点。两种情况下，均会将坐标分配给变量 P1。

局部变量必须在宏的开头定义。但是，程序员可能更喜欢在编写完其余宏之后再定义这些变量。

宏主体

宏的主体包含可执行代码。工作将在此完成。每行的开头均包含一个后接选项、表达式或运算符的命令或函数。这些术语将在本章的后面部分进行讨论。下面是我们的示例宏的主体：

```
CIRCLE CENTER P1 P2 END
```

END_DEFINE

此语句标记宏的结尾。

INPUT

使用宏中的 INPUT 命令时，必须结合 IMMEDIATE 限定符使用，例如：

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

最小宏

我们已经看到了，宏可由六个部分组成。但并不是所有宏均需要六个部分。如果宏不接受参数，则将没有 PARAMETER 部分。如果宏不使用变量，则将没有定义局部变量部分。宏可以没有用户输入部分。

绝对最小宏包含的部分不得少于三个：

- DEFINE
- 主体
- END_DEFINE

本章开头处的宏就是一个最小宏的示例。下面再举一个示例：

```
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
```

请注意，我们在单独的行中编写语句，使其易于理解。即使将宏编写为以下形式，计算机仍可理解：

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

语法图

现在正好向您介绍语法图。它将帮助您理解计算机为什么不在意您是将一系列命令编写在单独的行中，还是将所有命令编写在同一行。

如果学习了 DEFINE 的语法图，您便会明白计算机为什么能够理解该宏。在线帮助中提供了所有命令和函数的语法图。为了方便起见，我们已经包含了 DEFINE 的语法图：

```
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
```

在 DEFINE 语句之后，计算机会希望看到：

- 关键字 PARAMETER、
- 关键字 LOCAL 或
- 令牌。

计算机如果看到这三项中的任何一项，则会继续下一个语句。否则宏将停止。

在在线帮助的TOKEN下查看，您会看到命令是可接受的令牌。

现在请看 STORE 的语法图：

```
STORE command
->(STORE)-->+------>+-+--->(ALL)---->+->+-------------+->|file name|->
            |       | |              |  |             |
            `-(MI)->' `->|partname|->'  `->(DEL_OLD)->'
```

首先，请注意 STORE 是一个命令，所以 STORE 可用作令牌。

STORE 的后面必须带有关键字 MI，或关键字 ALL，或零件的名称。本例为 ALL。

ALL 的后面必须带有 DEL_OLD 或文件名。本例中首先为 DEL_OLD，然后是以单引号括起的文件名。

可以验证，EXIT 后一定接着 CONFIRM。

现在，尝试不含 ALL 的宏：

```
DEFINE Quit STORE DEL_OLD 'workfile' EXIT CONFIRM END_DEFINE
```

您将收到以下消息：

```
Enter option or 'part_name'
```

计算机无法理解该宏，因为它希望会出现 MI、ALL 或零件名称。

所以您将明白逐行编写宏有助于人类读取，但计算机则依赖诸如语法图这类算法来进行理解。

解释局部变量

局部变量仅对当前宏或任何由当前宏调用的宏是已知的。如果变量没有定义为局部变量，则将自动定义为全局变量。全局变量对所有宏均可见。通常，应该将所有变量均设为局部变量。全局变量可能存在危险，应避免使用。

要理解局部变量和全局变量间的差异，请在文件中键入以下三个宏。这三个宏非常相似，所以请键入第一个宏，然后对另外两个宏使用区块复制。可以省略任何备注。

```
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
```

请注意备注的使用，需用大括号 {} 括起来。编译器会忽略大括号括起的任何内容。备注可帮助其他读者理解宏。如果宏非常复杂，备注会帮助您理解该宏 - 特别是在六个月以后！

您的备注应解释命令的用途。用户可以使用语法图来理解命令。诸如下面的备注：

```
{allow time to read the message}
```

比以下备注更具启发性：

```
{wait approximately 3 seconds}
```

注意

备注不能嵌套。这意味着，如果代码段含有备注，则调试期间无法对此代码添加备注。

您会看到 outer_macro 调用 middle_macro，后者又会调用 inner_macro。这就是嵌套宏的示例。

没有宏使用局部变量，所以变量 X 是全局变量。这意味着 X 对所有这三个宏均可用。

输入包含这三个宏的文件，然后在命令行中键入：

```
Outer_macro
```

命令行输出应依次为：

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

讨论全局变量之前，让我们快速浏览一些宏语句。

DISPLAY_NO_WAIT 用于在命令行上显示消息。显示消息后，无需用户操作。此函数有别于 DISPLAY，后者需要用户按某个键来使宏继续。

DISPLAY_NO_WAIT 将首先打印：

```
outer X =
```

STR(X) 有何功能？X 是一个 NUMBER，其中带有某种使用位“设置”或“清除”的内部计算机表示。使用 DISPLAY_NO_WAIT 语句只能显示 ASCII 字符。STR(Age) 会将 X 的内部表示转换为等效的 ASCII 字符串，以便可以显示。

与字符串结合使用时，+ 符号可连接字符串。例如，假设 String1 为 'to'、String2 为 'get' 而 String3 为 'her'。那么 String1+String2+String3 便为 together。单引号内的任何空格也都会显示出来。

WAIT 3 可使系统在显示消息后等待大约三秒钟。这样可使您有时间阅读消息。

Middle_macro 语句可在系统运行时调用系统一定已知的另一个宏。因为示例中的三个宏均在同一文件中，所以需要时，系统会查找 Middle_macro。

现在，回到全局变量。您会发现在 Outer_macro 中定义的 X 的值等于 5 对中间宏和内部宏均为已知。在 Inner_macro 中，我们将 X 的值更改为 20。该值对中间宏和外部宏已是已知值。

现在按照下例所示，在中间宏 DEFINE 语句后添加一行。

```
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
```

输入文件，然后在命令行中键入 Outer_macro。输出应为：

```
'outer X = 5'
***The macro X is not defined
```

系统指示其不识别 X 的值。这是因为现在 X 是 Middle_macro 中的局部变量。任何外部宏中同名的其他变量对 Middle_macro 均不是已知的。

要使宏运行，请向中间宏中再添加一行，如下所示：

```
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
```

输入文件，然后在命令行中键入 Outer_macro。

现在输出为：

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

这次发生了以下变化：

- Middle_macro 中的 LOCAL X 语句使 Middle_macro 成为一个屏蔽外壳。X 的值无法透过这一外壳。这意味着 X 的值无法从 Outer_macro 向内传递至 Middle_macro，或从 Middle_macro 向外传递至 Outer_macro。但是 X 可以向内传递至任何由 Middle_macro 调用的宏，除非这些内部宏中的某些也对 X 进行了屏蔽。
- 在 Middle_macro 中，当 X 设置为等于 10 时，该值随后便可传递至 Inner_macro。
- 在 inner_macro 中，当 X 设置为等于 20 时，此值可向外传递至 Middle_macro。
- 因为 X 是 Middle_macro 中的局部变量，所以 Outer_macro 不识别 Middle_macro 中 X 的值。X 的值与 Outer_macro 中初始设置的值相同，即 5。

最后，让我们将 X 成为所有这三个宏中的局部变量：

```
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
```

输入文件，然后在命令行中键入 Outer_macro。输出如下：

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

您是否理解该输出？

为什么使用局部变量？

使用局部变量的一个主要原因就是要使宏尽可能地屏蔽以避免“副作用”。当变量意外受到内部宏中同名变量的影响时，会产生副作用。

您的办公宏库中可能包含了一个有用的宏，它正是为免去在宏中编写大片的代码所需的宏。要从您的宏调用此宏，不必担心调用的宏使用的变量名称是否会与您的变量名称相冲突。调用的宏可能会调用另一个宏，这个宏转而又会调用第三个宏，依此类推。您不会想读取这些宏中的每一行来查找冲突的变量。如果编写这些宏的人员使用了局部变量，您便不必担心。

例如，以下的宏可能不会按照程序员所预计的方式运行：

```
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
```

该宏是否会给出以下输出？如果不会，原因是什么？

```
'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
```

既然您已熟练掌握局部变量，那么您会知道输出应该是什么。

由于 X、I 和 N 等变量常用作循环中的计数器，所以您会看到程序员对其他宏中的这些值进行屏蔽有多么重要。

我们是否声明了变量类型？

如果您有一些程序设计知识，您会知道在大部分程序设计语言中，必须声明变量。例如，Pascal 中有诸如整型、实型和字符型的变量。C 语言中有整型、浮点型、字符型等。

声明变量意味着会为该变量保留正确大小的内存位置。

在宏中，我们不正式声明变量。

我们最近一次谈到声明变量是在使用 READ 语句时。读取语句会生成一个命令行，提示用户进行输入。下面是一些示例：

```
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
```

为了保存键入内容，我们没有使用局部变量。

请看第一个 READ 语句：

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

运行宏 (请尝试！) 时，命令行会出现提示：

```
Enter file name
```

现在必须输入以单引号括起的字符串。如果忘记加单引号，系统将再次显示相同的提示。这是紧跟在 READ 语句后声明变量类型的优点：如果提供了错误类型的输入，系统将重新执行该宏行直到输入与指定的类型相匹配。

对于第二个 READ 语句，如果尝试输入以单引号括起的数字，或输入一个点而不是数字，将会重新显示提示。所以当您尝试输入不满足类型 NUMBER 的值时，系统会向您发出警告。

可以使用 READ 语句而不指定类型，例如 STRING 或 NUMBER。然后在执行 READ 语句时，系统会接受所有不正确的数据。系统不会发出错误消息，直到宏中使用数据时。在示例宏中，这会是在执行 DISPLAY_NO_WAIT 语句过程中。请尝试移除 READ 语句后的某个类型语句。请运行宏，然后故意提供一些不正确的数据。

您是否注意到我们使用了 STR(Age) 和 STR(Point)，但没有使用 STR(File)？请记住，STR(Age) 会将 Age 的内部表示转换为等效的 ASCII，以便可以显示。STR(Point) 作用相同。File 已经是 ASCII 字符串，与用户输入的完全一样，所以不需要转换。

以下示例显示的是如何将现有/计算得到的值回输到屏幕。“默认”选项可评估其参数，并将结果输入到输入行中。按下 [Return] 随后将输入 (可能已编辑的) 值来满足 READ 要求。

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

使用控制语句

我们不会总是想要按照语句在宏中出现的顺序来执行宏语句。可能想要某些语句执行多次。也可能想要某些语句仅在特定条件为真时才执行。控制语句用于更改宏的执行方式。

WHILE ... END_WHILE

我们在之前的“为什么使用局部变量？”一节中使用了 WHILE 语句。此处重新列出了该宏的一部分：

```
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
```

只要 X 小于 3，就会执行 WHILE 和 END_WHILE 之间的代码。在 WHILE 语句之前，X 设置为 1，而每执行一次循环 X 便会增加一。所以此循环应执行两次 (而非三次！)。实际上，由于内部宏中全局变量引起的副作用，此循环只执行一次。

您会看到 WHILE ... END_WHILE 构造用于控制循环执行的次数。此构造的一个重要的特征是循环一次都不需要执行。如果循环入口处的 WHILE 语句后的条件为假，则循环将永远不会执行。稍后，我们将介绍 REPEAT ... UNTIL 构造，该构造可生成一个必须至少执行一次的循环。

LOOP ... EXIT_IF ... END_LOOP

我们看到了 WHILE ... END_WHILE 构造在循环起点处设置了条件测试。在 LOOP ... EXIT_IF ... END_LOOP 构造中，条件测试在循环中的任一点均可进行。

在以下代码片段中，系统提示用户输入一个值。如果用户没有提供正确的值，提示将会再次显示，直到输入了正确的值为止。

```
LOOP
  READ NUMBER 'Enter a fractional value for the split' Fract
EXIT_IF ((Fract>0)) AND (Fract< 1))
END_LOOP
```

在循环的主体中可使用任意数量的 EXIT_IF 语句，如以下略图所示：

```
LOOP
  ...
  ...
EXIT_IF
  ...
  ...
EXIT_IF
  ...
```

```
  ...
END_LOOP
```

如果没有使用 EXIT_IF 语句，则只有用户输入 END 或开始一个命令时，宏才会停止。下面是一个示例：

```
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
```

如果我们把第一个 READ 语句放在 LOOP ... END_LOOP 构造中会发生什么？这是更改后的宏：

```
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
```

LOOP ... EXIT_IF ... END_LOOP 构造非常有用，因为循环可在任意点终止，而不仅仅是起点或终点。而且正如我们所看到的，用户也可以输入 END 来终止宏。

请注意，如果用户输入了 END，则整个宏都会终止，而不仅仅是循环。不要期望宏能跳出循环并继续执行 END_LOOP 之后的语句。

REPEAT ... UNTIL

此构造与 LOOP ... EXIT_IF ... END_LOOP 构造类似，唯一差别是 EXIT_IF 语句就放在 END_LOOP 语句之前。因为直到循环结束后，才会执行 UNTIL 之后的条件测试，因此此循环始终都将至少执行一次。

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

IF 系列语句用于作出决定。然后根据所作决定来执行宏的某些部分，而不执行其他部分。

在以下代码片段中，将会询问用户希望构造线呈水平、竖直还是垂直于现有线：

```
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
```

条件从顶部向下，以阶梯形式进行评估。一旦发现条件为真，便会执行与该条件关联的语句，并忽略其余阶梯。

可以根据需要多次使用 ELSE_IF 语句，或者一次也不使用。只能具有一个 ELSE 语句，且必须在结尾出现。最后的 ELSE 为默认条件。也就是说，如果其他所有条件测试均失败，那么便会执行 ELSE 语句。如果不存在最后的 ELSE，且其他条件均失败，则不会执行任何操作。有时会在防御式程序设计中使用最后的 ELSE 来处理错误。

绝对最小 IF 语句不具有 ELSE_IF 或 ELSE。例如：

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

IF 语句可以进行嵌套，但是每个 IF 都必须具有自己的 END_IF。我们可以展开之前的示例：

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

使用括号

任何程序设计语言都有一个问题，就是何时使用括号。以下指导方针可能会对您有所帮助。

布尔表达式

布尔表达式始终使用括号括起。

布尔表达式 (有时称为条件表达式) 是值为真或假的任意表达式。

布尔表达式在以下四个主要构造的测试部分中使用：

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

算术、代数和三角函数表达式

表达式用作令牌时，始终用括号括起。

表达式示例如下：

- 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!'

有关可用令牌的完整列表，请参阅在线帮助。如果语法图需要令牌，并且使用以上任意表达式，则必须使用括号将表达式括起。例如：

```
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)
```

上例显示了几个关键点。让我们来逐个地查看这些示例。

请查看语法章节中的 LET 语法图。您会看到 LET 后接变量的名称，然后是一个令牌。在第一个示例中，X + Y 是用作令牌的表达式，所以必须用括号括起。

现在请看第二个示例。这里表达式为

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

在此表达式中，需要使用括号来强制计算顺序正确。如果将此表达式 (包括括号) 用作令牌，则需要一组更深一层的括号。

第三个示例显示的是简单的不需要括号的令牌：

```
LET Point2 Point1
```

如果对这一简单的令牌进行修改使其变为一个表达式，则需要使用括号。

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

请注意，以下语句完全可接受：

```
LET Point2 (Point1)
```

请查看语法章节中的 DISPLAY_NO_WAIT 语法图，您将会看到 DISPLAY_NO_WAIT 后必须接一个令牌。DISPLAY_NO_WAIT 的示例与 LET 的后两个示例类似。

通过之前的示例，您会发现 LET 语句有时需要令牌带有括号，而有时则不需要。早期编写宏时的一个较好规则可能为：始终对 LET 语句使用括号。

使用追踪工具

T 追踪是一种有用的调试工具。我们要在这为您介绍 T 追踪，因为您会看到布尔表达式和其他表达式的计算方式。

下面是我们要追踪的宏：

```
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
```

该宏将绘制四条线。当 X 等于 1 时，将从 0,0 到 10,0 绘制一条线。当 X 等于 2 时，将从 0,10 到 10,10 绘制一条线，依此类推。

我们已经了解了何处需要括号何处不需要括号。

此宏展示了一项重要原理。查看《Creo Elements/Direct Drafting 程序设计参考指南》中 LINE TWO_PTS 的语法图。在宏中，只有 END 语句可结束显示的递归循环。这是常规规则：语法图以递归循环结束时，请使用 END 语句退出循环。

请查看 LINETYPE 的语法图。此图不以递归循环结束。宏中不需要 END 语句。

现在，让我们在宏中使用 T 追踪，并将结果存储到名为 trace 的文件中。

使用 T 追踪之前，将 parenth 中的第二行从：

```
LET P1 (0,0)
```

更改为：

```
LET P1 0,0
```

在用户输入行中键入下列内容，并在每个命令之后都按 ENTER 键：

```
input 'parenth'

trace

parenth
```

宏结束后，禁用 T 追踪：

1. 单击杂项，然后在系统组中单击 T 追踪。
2. 单击 ☒ 关。

   状态栏中的 T 指示追踪处于非活动状态。

注意

如果 trace 文件意外为空或不完整，则可能是您忘了禁用 T 追踪。

如果想要将 T 追踪操作的结果附加到现有文件：

1. 单击杂项，然后在系统组中单击 T 追踪。
2. 单击 ☑ 开。
3. 单击 T 附加。

   状态栏中的 T 指示 T 追踪处于活动状态。

如果每次运行都需要清除文件：

1. 单击杂项，然后在系统组中单击 T 追踪。
2. 单击 ☑ 开。

3. 单击 新建。

状态栏中的 指示 追踪处于活动状态。

要查看 追踪操作的结果：

1. 单击杂项，然后在系统组中单击 追踪。
2. 单击 编辑。

追踪操作的结果显示在标准文本编辑器中。

这是 parenth 的 trace 文件：

```
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
```

trace 的第一行

```
LET P1 0,0
```

与我们的宏中的第一行相同。

第二行不同。这里，已对括号内的"表达式"进行了计算。结果 10,0 紧跟表达式后面显示。

现在请看布尔表达式：

```
WHILE ( X 1 < 5 ) 1
```

X 的当前值为 1。该值紧跟 X 后面显示。布尔表达式 (1 < 5) 为真，所以值 1 紧跟表达式后面出现。

请注意，trace 中将不再出现 WHILE，但是 END_WHILE 后面会出现布尔表达式的值。请看最后的 END_WHILE 后面的最后一个布尔表达式。这里，X 的当前值为 5。布尔表达式为 (5 < 5) (为假)，所以该表达式的值为 0。

以下宏将可执行：

```
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
```

现在，WHILE 后面的表达式始终为真。此宏将继续绘制线，直到出现断电情况或直到按 [Break] 键为止。

这里故意包括了此宏以显示布尔表达式始终需要括号，即使使用的是简单的令牌。如果您不相信，请尝试将 WHILE (1) 更改为 WHILE 1。

将 WHILE (1) 更改为 WHILE (TRUE)，它仍可执行。

以下宏表明如果值为任意非零值，则布尔表达式将为真。

```
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
```

缩进宏的行

您可能已经注意到，宏的某些行进行了缩进。处理器会忽略这一缩进，但缩进有助于读者理解逻辑流程。

按照惯例，布尔表达式下面的行会进行缩进。在之前的示例中，每个 WHILE 语句下面的行均进行了缩进。WHILE 构造的结束语句为 END_WHILE，此语句与 WHILE 出现在同一列中。结果是，该构造的关键字在视觉上起到了标记作用。读者可立即看到每部分代码的开始和结束位置。这样使调试更加容易。

同样，缩进可用于以下构造：

- IF ... ELSE_IF ... ELSE ... END_IF
- LOOP ... EXIT_IF ... END_LOOP
- REPEAT ... UNTIL

构造的每个关键字均开始于同一列。关键字下面的语句将进行缩进。

其中任一构造的嵌套都需要进一级的缩进。

防御性程序设计

可以通过防御性程序设计并尽力考虑到所有可能发生的事件，来避免出现很多常见缺陷。以下是一些指导方针：

- 使用一致的缩进。我们在上一节中对此进行了讨论。
- 使用描述性变量名称。Radius1 或 Rad1 比 S 更易于理解。您可能无意中编写为 TAN(S)，但是您会立即看到 TAN(Radius1) 中的错误。
- 编写简单的代码。以下两个代码片段会生成相同结果。哪一个更易于理解？

  片段 1：

  ```
  LET P3 (P1 - (PNT_RA 0.5*(D2 - D1) ANG(P2 -P1)))
  ```

  片段 2：

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

  如果之前的任意一个代码片段中存在缺陷，您更愿意调试哪一个？
- 编写宏时不要担心代码是否高效。如果最终的宏运行过慢，则可将精力集中在可能获得更快速度的区域。

- 避免编写过长的宏。建议使用多个较短的宏。每个宏均可单独进行调试。
- 请尝试筛选出错误的用户输入。在讨论 IF 语句时，我们查看了以下代码片段：

```
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
```

  此示例很好地展示了 IF 语句，但没有较好地展示出防御性程序设计。举例来说，如果用户无意间输入了 'Y'，或者输入了 'v' 而不是 'V'，会发生什么？

  这是一个重新编写的版本，旨在捕获意外的用户错误：

```
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
```

- 如果用户输入过大、过小或负的数值时宏失败，请尝试使用类似如下语句来捕获这些数值：

```
LOOP
  READ NUMBER 'Enter number of sides' Num_sides
EXIT_IF ((Num_sides > 0) AND (Num_sides < 20))
END_LOOP
```

宏命令

在线帮助中列出的所有命令均可在宏主体中使用。某些命令和语句在宏中非常有用。下面是一些示例：

DISPLAY P1	显示点 P1 的 X、Y 坐标并等待用户响应。
BEEP	在系统扩音器上发出短的固定频率/振幅音调。
IF (M>N)	执行 ELSE if M>N 之前的所有行。否则执行

... ELSE ... END_IF	ELSE 和 END_IF 之间的所有行。可使用任意数量的 ELSE 语句。
LET D (L1 + 5)	将 D 定义为等于 L1 和 5 之和。
LOOP ... EXIT_IF (N > 50) ... EXIT_IF (M < 7) END_LOOP	依次重复执行 END_LOOP 之前的所有行，直到 N > 50 或 M < 7。可使用任意数量的 EXIT 语句。如果没有使用退出语句，则循环将无限次地重复，直到遇到 END 或者选择了其他命令为止。
READ PNT P8	停止系统，直到用户输入点 P8。
REPEAT ... UNTIL (N>10)	依次重复执行所有行，直到 N 大于 10。
TONE 256 2 0.5	在系统扩音器上发出持续时间为 2 秒，相对振幅为 0.5 的 256 Hz 的音调。
WHILE (N< 20) ... END_WHILE	当 N 小于 20 时，依次重复执行 WHILE 之后的所有行。

表达式用括号括起。表达式通常包含算术、三角函数或关系运算，如之前的示例所示。我们也可以使用某些特殊运算。在下节中将对其进行说明。

内置运算

以下是一些可在表达式内使用的实用内置运算的示例：

ANG P6	计算矢量 P6 与 X 轴之间的角度。
LEN P4	计算矢量 P4 的长度。
PNT_XY 20 65	定义一个原点位于 (X0, Y0)，矢量头位于 (X20, Y65) 的矢量。
PNT_RA 16 38	定义一个原点位于 (X0, Y0)，长度为 16 且与 X 轴间夹角角度为 38 的矢量。
ROT P2 45	定义一个与矢量 P2 长度相同，与 P2 间夹角角度为 45 度，逆时针方向的矢量。
SQR 15	计算 15 的平方。
SQRT 15	计算 15 的平方根。
X_OF P1	计算点 P1 的 X 坐标。
Y_OF P6	计算点 P6 的 Y 坐标。

4

查询环境和元素

使用 INQ_ENV...52
使用 INQ_ELEM...53
使用 GETENV...54
使用其他查询...54

编写宏时，经常需要了解当前环境。当前单位是什么？英寸还是毫米？当前线颜色是白色还是某种其他颜色？比例为 1:1 还是 2.5:1？

例如，您可能要使用宏来绘制黄色点中心线。更改颜色和线类型很容易。但完成编写宏后，应将颜色和线类型更改回其原始值。宏的用户应不必重置这些值。

宏结束之前，您的宏必须存储当前颜色和线类型，然后再检索这些值。本章为您展示了如何使用 INQ_ENV 和 INQ_ELEM 函数在系统阵列中存储信息，以及如何使用 INQ 查询系统阵列。INQ 是查询 (inquire) 的缩写。

使用 **INQ_ENV**

以下宏展示了 INQ_ENV 和 INQ 的使用：

```
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
```

宏的第二行会使用当前颜色和线类型来绘制线。宏的其余行将在以下几段中进行讨论。

```
INQ_ENV 3
```

如果在在线帮助中学习了 INQ_ENV 函数，您会发现 INQ_ENV 可将数据插入到系统阵列的某一部分中。阵列的各部分由 INQ_ENV 后接的数字指定。例如，部分 0 包含软件版本号和版本字符串的信息。部分 1 包含有关当前视区等内容的信息。

我们对部分 3 感兴趣，该部分包含有关捕捉范围、几何、构造数据、文本数据等内容的信息。

每次使用 INQ_ENV 时，都会覆盖系统阵列该部分中之前的数据。

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

INQ 用于查询上一个使用 INQ_ENV 写入的系统阵列部分。在我们的示例中，上一个 INQ_ENV 写入的是部分 3，所以 INQ 将查询部分 3。特定信息取决于 INQ 后接的数字。

在在线帮助中重新查看 INQ_ENV 函数，您会发现 INQ 201 可返回当前几何的颜色。此值将分配给变量 Col。

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

当前几何线类型将分配给变量 Ltype。

```
INQ_ENV 2
```

有关当前窗口的信息将插入到系统阵列的部分 2 中。

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

当前窗口的左下点和右上点将分配给变量。

宏的接下来的三行将更改颜色和线类型，然后绘制线以显示更改的效果。

```
COLOR Col
LINETYPE Ltype
```

颜色和线类型的原始值被恢复。宏将绘制另一条线以显示更改的效果。

宏的最后几行将更改当前窗口的坐标，并显示此窗口约五秒钟以使您可以看到更改。随后将恢复原始坐标。

使用 **INQ_ELEM**

INQ_ELEM 与 INQ_ENV 类似，也会将数据放入系统阵列中。该数据取决于绘图上指定点处存在的元素。例如，如果元素为圆，则会将关于圆的数据放入阵列中。

如果拾取了一个元素，可以使用 INQ 403 来查明元素是何种类型。在宏中，您可能想要用户拾取圆。可以使用 INQ 403 来验证实际上它是圆，而不是其他类型，例如矩形。

以下代码片段显示了一个示例：

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

在此片段中，INQ_ELEM 可将关于数字化点的信息读入系统阵列中。请参阅在线帮助中的 INQ_ELEM 函数，您会发现 INQ 403 可返回元素类型。如果类型为圆，宏将退出循环并继续宏的下一部分。

如果想要知道圆的半径或圆心，可随后使用：

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

请注意，如果在数字化点处没有找到元素，则 INQ 403 将返回 END。再次提示前，您可能需要在捕捉模式中内置检查。

使用 GETENV

GETENV 用于检索当前环境设置。例如，要检索变量 MEDIR 的设置，请在命令行中输入以下内容：

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

Creo Elements/Direct Drafting 将返回当前设置。

使用其他查询

其他查询 (例如 HL_INQ_Z_VALUE 和 HL_INQ_FACE_COLOR) 的使用与 INQ_ENV 和 INQ_ELEM 的使用类似。每种情况下，值都将写入系统查询阵列，然后使用 INQ 进行检索。

有关使用 HL_INQ_Z_VALUE 的宏示例，请参阅第89页上的“有用的宏”。

5

快速查看点和矢量

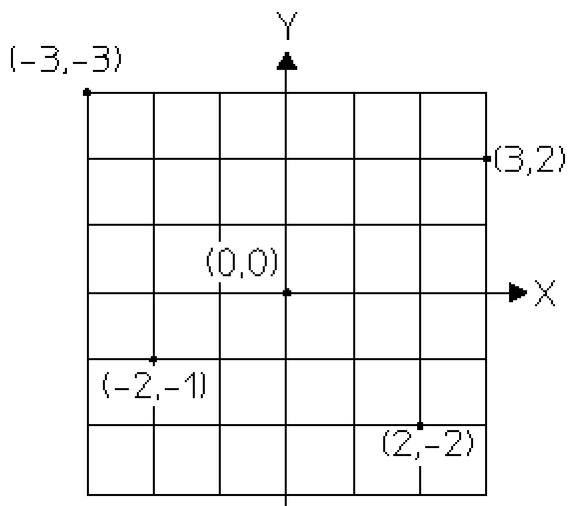
点 56
矢量 56

大多数阅读手册 (如本手册) 的人都会熟悉点和矢量。但是，您可能已有几年没有使用过它们了。本章非常简短，主要是为了在阅读第60页上的"编写几何宏"之前唤起您的记忆。

点

点通过 X 坐标和 Y 坐标进行定义。请看下图中的点：

图 **1.**点的表示



X 轴和 Y 轴相交于点 (0,0)。此点称为原点。在原点处，X 坐标为 0，Y 坐标为 0。

可以看到，其他点通过先写出 X 坐标，然后再写出 Y 坐标来进行标识。

在宏中，如果需要知道其中点的 X 坐标或 Y 坐标，请使用 X_OF 或 Y_OF 运算符。例如：

```
LET X1 (X_OF P1)
LET Y1 (Y_OF P1)
```

假设 P1 为 (3,2)。那么 X_OF P1 便为 3，而 Y_OF P1 为 2。

如果有两个数字，您就可以使用 PNT_XY 运算符来根据这些数字生成点。例如，使用数字 3 和 2：

```
LET P1 (PNT_XY 3 2)
```

将生成点 (3,2)。

一般情况下：

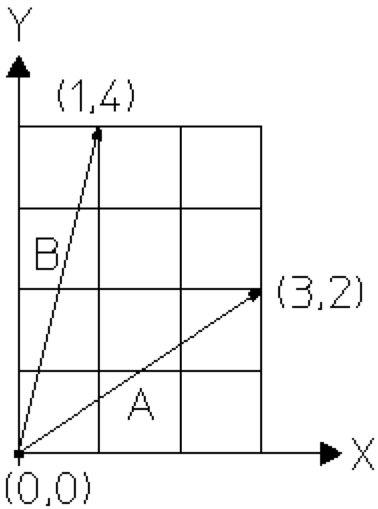
```
LET P1 (PNT_XY X Y)
```

将生成点 (X,Y)。

矢量

矢量具有长度和方向。下图中，矢量 A 代表一条从原点 (0,0) 到点 (3,2) 的线。矢量 B 代表一条从原点 (0,0) 绘制到点 (1,4) 的线。

图 **2.**起始于原点的矢量

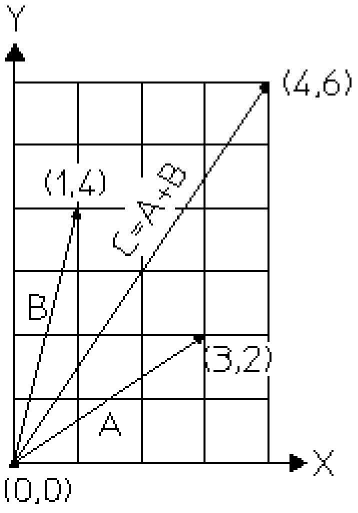


如果矢量起始于原点 (如上图所示)，则我们可以把矢量 A 称作 (3,2)，把矢量 B 称作 (1,4)。在下一章将介绍的几何宏中，所有矢量均起始于原点。

矢量相加

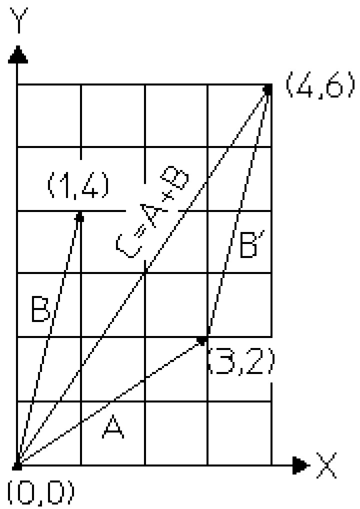
可将矢量 A 与矢量 B 相加，得到矢量 C。下图中，可以看到 C 代表一条从原点绘制到点 (4,6) 的线。C 表示的 X 坐标由 (1,4) 和 (3,2) 的 X 坐标相加而得，结果为 4。同样，Y 坐标由这两个点的 Y 坐标相加而得，结果为 6。

图 **3.**矢量相加



这是另一种形象显示 A 和 B 矢量相加的方法。首先，将矢量 B 移动到新位置 B'，与矢量 A 的终点相连，如下图所示。请记住，矢量由长度和方向定义，所以新矢量 B' 与旧矢量 B 相同。

图 **4.**形象显示矢量相加



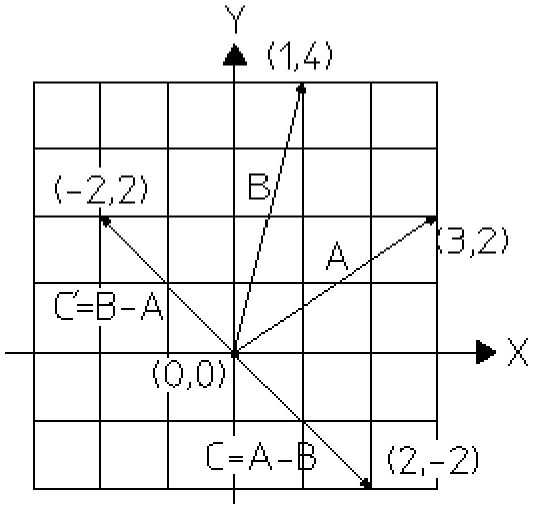
现在从原点开始，在 X 方向移动 3 个单位并在 Y 方向移动 2 个单位，然后在 X 方向移动 1 个单位并在 Y 方向移动 4 个单位。如果从原点开始，在 X 方向移动 4 个单位并在 Y 方向移动 6 个单位，将到达同一点。这证明了 C = A + B。

在矢量相加中，A + B 与 B + A 相同。

矢量相减

要减去矢量，请减去其 X 和 Y 坐标。减法的顺序很重要。下图显示了 A – B 和 B – A 的结果。

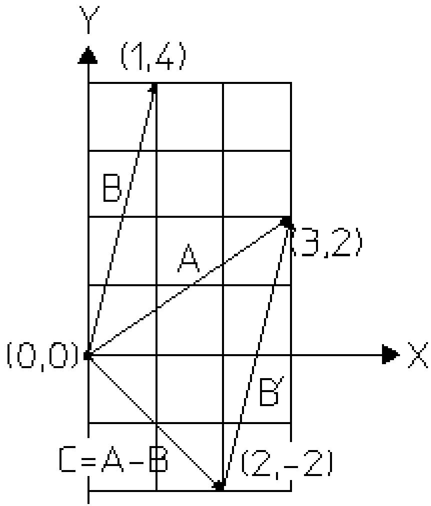
图 **5.**矢量相减



可以看到，和普通算术一样，矢量 A – B 与矢量 B – A 不同。这两个矢量长度相同，但方向相反。

与加法一样，可以通过移动其中一个矢量，将其连接到另一个矢量的终点来形象显示矢量的减法。下图显示了 A - B 的效果：

图 **6.**形象显示矢量相减



6

编写几何宏

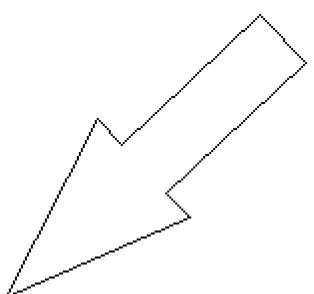
箭头宏 61
面板宏 64

本章为您展示如何编写几何宏。与其他类型的宏相比，可能几何宏更常用。如果您发现要经常使用固定的尺寸或变化的尺寸来绘制相同类型的形状，则您应为自己编写一个宏来进行该项作业。

本章将详细介绍如何编写箭头宏和面板宏。

箭头宏

该宏将绘制下图中显示的箭头：



和绘制基本形状一样，宏必须做到以下几点：

- 箭头必须位于从屏幕中拾取的任意两点之间。第一个点将是箭头的尖端。第二个点将位于尾部的中心。
- 箭头比例必须恒定。
- 宏必须重复操作，直到被其他命令取消。

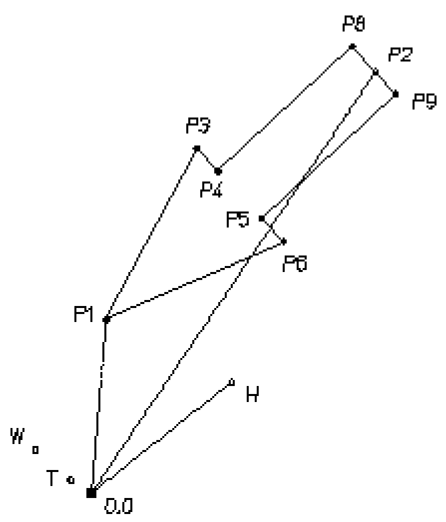
可满足这些要求的宏的编写方法有多种。没有关于使用何种方法的规则。请选取最适合您的一种。

在我们要使用的方法中，会将点视为矢量。这对于熟悉简单矢量处理的人来说，是一个很好的方法。

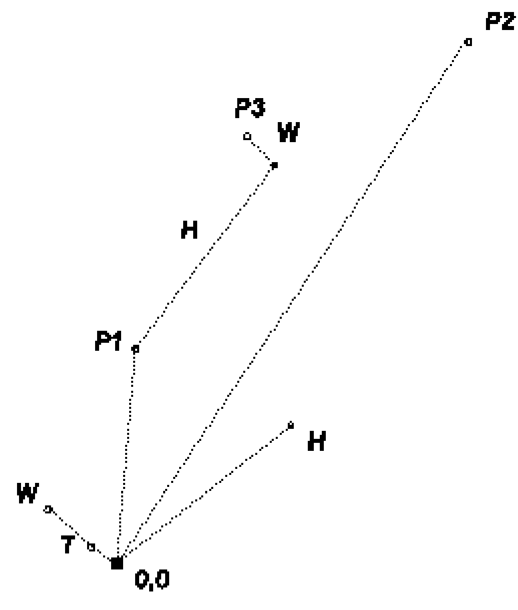
编写箭头宏

下面是一步一步的执行顺序：

1. 绘制与轴成任意角度的箭头，然后注释所有点。要从屏幕上拾取的点为 P1 和 P2。绘制任意点作为原点 (0,0)。



2. 首先处理宏主体。对于几何宏，这就是执行绘制的部分。假设定义了点 P1 和点 P2，且它们表示从原点开始绘制的两个矢量的头，如上图所示。
3. 箭头的倒钩占沿中心线的总长度 (从箭头尖) 一定的比例 (让我们使用 60%)。定义一个长度为 P1 和 P2 之间距离，方向为 P2 的方向的矢量，称之为 H。与其他所有矢量一样，H 也起始于 0,0。
```
LET H (0.6 * ( P2 - P1 ) )
```
4. 倒钩占 H 距中心线之间距离的一定比例 (让我们使用 30%)。定义一个长度为此长度，方向垂直于 H 的矢量，称之为 W。
```
LET W ( 0.3 * ( ROT H 90 ) )
```
5. 尾部厚度的一半占 W 的一定比例 (让我们使用 40%)。定义一个长度为此长度，方向为 W 方向的矢量，称之为 T。
```
LET T ( 0.4 * W )
```
6. 现在就可以定义点了。将 H 和 W 加到 P1 上以定义点 P3，如下图所示。




```
LET P3 ( P1 + H + W )
```
7. 以完全相同的方法定义其他点。
```
LET P6 ( P1 + H - W )
LET P4 ( P1 + H + T )
LET P5 ( P1 + H - T )
LET P8 ( P2 + T )
LET P9 ( P2 - T )
```
8. 连接点以创建箭头形状。
```
LINE POLYGON P1 P3 P4 P8 P9 P5 P6 P1
```

宏主体已完成，现在我们可以继续处理从屏幕中拾取的点 P1 和点 P2。通过 READ 命令，用户可输入 P1 和 P2 的坐标、

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

READ PNT 告知系统将要输入点。系统将显示消息“请拾取箭头点”。用户输入点时，会将该点定义为 P1，并执行宏的下一行。

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

在此行中，随着在屏幕上移动光标，RUBBER_LINE 命令会在 P1 和光标之间绘制一条线。这对调整箭头尺寸非常有效。输入下一个点时，会将该点定义为 P2，并执行下一行。该行是宏主体的开头。

现在，请使用 LOOP 和 END_LOOP，将 READ 语句和宏主体包含在一个循环中。这样用户就可以在同一操作过程中创建多个箭头。在循环外部指定 LINETYPE 和 COLOR，以便需要时可在操作过程中对其进行更改。将所有宏变量定义为 LOCAL。

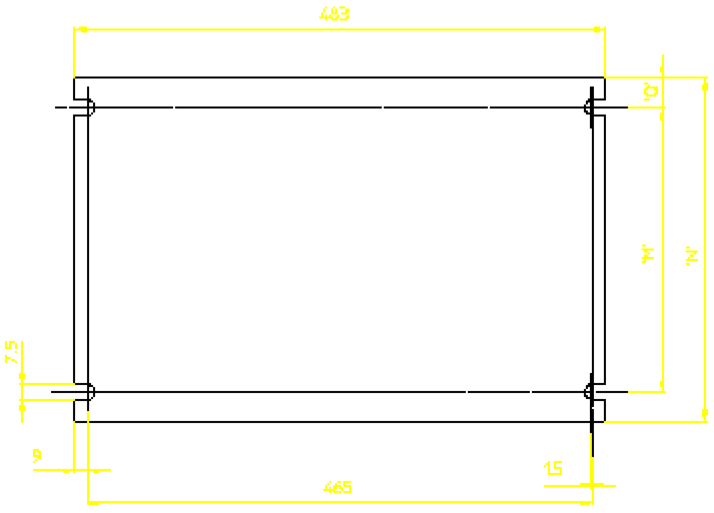
这里是完整的宏：

```
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
```

请记住，可进行该作业的宏的编写方法有很多种。另一种方法就是在建立点和绘制形状之前，将轴旋转至所需角度。此方法将使用 CS_SET、CS_ROTATE、CS_REF_PT 和 CS_AXIS 等命令。

面板宏

旨在编写一个宏来绘制类似如下所示的前面板。该面板专为适合标准的 19 英寸齿条而设计。



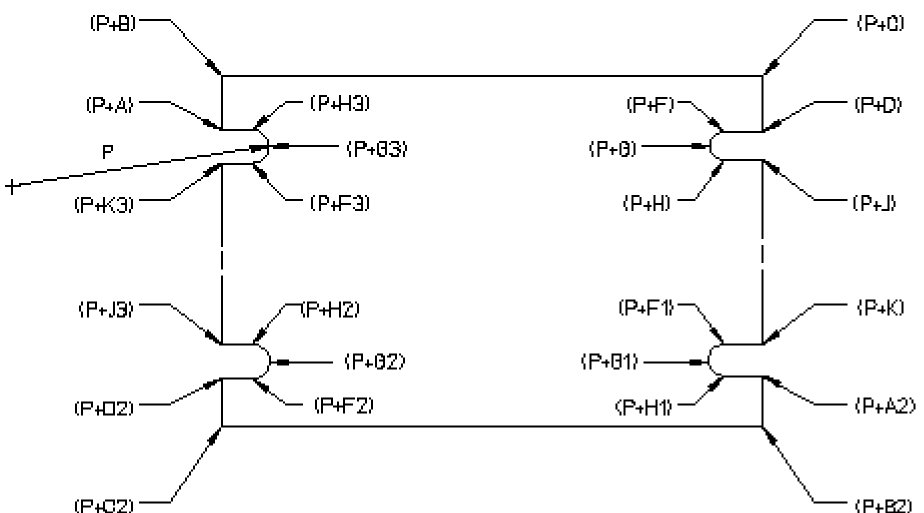
该宏必须做到以下几点：

- 通过拾取齿条左立柱上的一个螺栓孔来定位面板。
- 通过给定所需的 U 值来确定面板高度 (宽度恒定)。
- 保持面板方向与标尺轴方向相同。

前面板的几何大体上是固定的，所以可以使用坐标代替矢量来定义构造点。

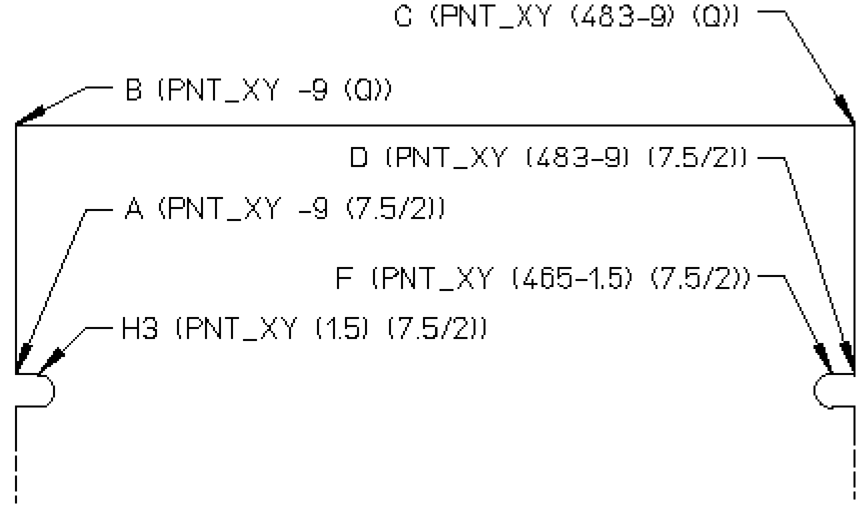
编写面板宏

以固定角度 (0 度最简单) 绘制出前面板，然后注释涉及的所有坐标。



和箭头一样，首先开始处理宏的主体。假设定义了矢量 P，且前面板的规格如图所示。M 和 Q 是可变参数。

通过根据矢量 P 的终点确定的坐标来定义所有点。将 P 的坐标看作是 (0,0)。随后可将这些坐标依次与初始矢量 P 相加来创建面板。下图显示了一些面板特征的坐标：



围绕面板继续此过程可给出所有特征的坐标，它们可按照以下方法进行定义：

```
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))
```

要完成宏主体，请连接所有点来创建前面板。使用 LINE 和 ARC 可进行此项操作，并且可以任意顺序进行。这里使用的顺序是从点 (P+H3) 到点 (P+G3)，得出：

```
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)
```

宏主体现在已完成。要使宏可操作，必须局部化矢量点 P，并允许输入数据。READ 语句可用于数据输入。READ 语句的执行方式如下所示：

```
READ NUMBER  "Enter the required height of the front panel" U
READ "Identify the position of the top left bolt hole" P
```

现在请按如下方式局部化矢量点 P：

```
READ NUMBER  "Enter the required height of the front panel" U
READ "Identify the position of the top left bolt hole" P
```

该宏现在已可操作，但它仍然是非常基本的宏。该宏没有指定线类型、线颜色、宏何时结束、输入值的任何约束，或者如何将此值转换为面板的实际高度。这些细化可确定宏的用户友好程度。

可以在宏内指定线类型和颜色。例如，可以使用：

```
COLOR YELLOW
LINETYPE SOLID
```

指定宏应该在何时何处完成操作也很简单。此操作将通过在宏中的 END_DEFINE 之前插入 END 来实现。如果没有使用 END 语句，系统将显示宏被调用之前所执行的命令的提示。

可以通过多种方法来指定对 U 值的约束。本例中，选取了 LOOP...EXIT_IF...END_LOOP 构造并配合 IF...END_IF 来检查 U 的输入。此宏中，对 U 值强加了三项约束：

- 必须为整数值。
- 必须大于 0。
- 必须小于 17。

循环按以下方式使用：

```
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
```

此循环将会一直执行，直到 U 的值满足以上条件为止。

检查完值之后，必须将其转换为更适用于宏的形式。此时值为整数且表示面板的单位高度，因此必须将其转换为实际高度。此操作将由宏中最后的 IF...END_IF 循环实现。定义循环之前，U 值必须通过 LET N (((U-1)*44.45)+43.6) 进行转换，以创建表示面板实际高度的 N 值。此值随后将用于定义变量 M 和 Q。变量 M 定义槽之间的竖直距离，Q 定义各个槽和水平边之间的竖直距离，其中 M 在 U 的整个范围内可变，而 Q 对满足以下条件的单独值是固定的：

- U 大于 0 小于 3。
- U 大于 2 小于 17。

这里是完整的宏：

```
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

文件输入/输出和文本字符串

宏的功能......70
分析宏......71
从宏的内部调用宏......77
向宏传递参数......79

本章将向您介绍如何从文件提取数据以用于宏，以及如将数据写入文件。在我们的示例中，数据由文本字符串组成，但同样的规则可应用于数字数据。

宏的功能

目标是编写一个宏，它从help文件的各个部分提取主要关键字，并将结果写入新文件。

要了解“关键字”的含义，请查看help文件中的以下部分：

```
=======================================================
^AUTO_NEW_SCREEN
AUTO_NEW_SCREEN function
---->(AUTO_NEW_SCREEN)---->+------>(ON)----->+---->
                           |                 |
                           `------>(OFF)---->'
The AUTO_NEW_SCREEN function allows you to select ...
.
.
=======================================================
```

在本例中，这一部分描述了 AUTO_NEW_SCREEN 函数。关键字为 AUTO_NEW_SCREEN，这是我们要写入到输出文件的字符串。

请注意，您可以采用以下两种方法之一搜索关键字：

- 查找 ^ 字符，然后采用 ^ 后的字符串。或者，
- 查找字符串 command 或 function。然后采用之前的字符串。

从编程角度来说，第一种方法更好。处理器只会以“向前”的方向检查文件。使用第二种方法，处理器需要“向前”查找直到找到 command 或 function，然后再“向后”查找之前的字符串。我们采用第一种方法。

现在，请看下一个示例：

```
=======================================================
^CANCEL ESC STOP INTERRUPT BREAK
^ABORT
^Meview_cancel
CANCEL command
---->(CANCEL)---->
CANCEL  cancels  the current  activity  ...
.
.
=======================================================
```

在本例中，^CANCEL、^ABORT 和 ^Meview_cancel 前面都带有一个 ^，但我们只想将 CANCEL 包含在输出文件中。不希望包括 ABORT 或 Meview_cancel，因为这些字符串引用的是help文件的其他部分。因此，宏必须告知处理器提取各个部分中前面带有 ^ 的第一个字符串，并忽略其他字符串。

讨论文本处理宏之前，还需要提到一点。宏必须做的事情之一是搜索行中的空格。在之前的示例中，^CANCEL 与下一个字符串 ESC 之间由一个空格隔开。有时我们使用空格来判断是否正处于字符串的末尾。现在请再次查看help文件的这一部分：

```
=======================================================
^AUTO_NEW_SCREEN
AUTO_NEW_SCREEN function
---->(AUTO_NEW_SCREEN)---->+------>(ON)----->+---->
                           |                 |
                           `------>(OFF)---->'
The AUTO_NEW_SCREEN function allows you to select ...
.
.
=======================================================
```

该部分的第一行是 ^AUTO_NEW_SCREEN。认识到此行何时会在屏幕上打印出来十分重要，该行用空格填补。但在文件中此行中没有空格。AUTO_NEW_SCREEN 的最后一个 N 后有一个换行字符，它不会打印在屏幕上，但是会显示在文件中。换行字符的十六进制 ASCII 值为 0A。如果 "\n" 代表换行字符 (C 编程语言中使用 "\n")，之前的示例则显示为：

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

请注意，"\n" 是两个字符，但实际换行字符是一个字符。

分析宏

以下宏将从help文件提取关键字。宏不一定最有效，但很容易理解。我们现在逐行分析该宏。

```
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
```

首先看该宏的第一行：

```
DEFINE Keywords_search
```

每个宏必须以 DEFINE 开头，后跟宏的名称。这里宏的名称是 Keywords_search。

```
LOCAL File1
LOCAL File2
LOCAL Flag
LOCAL Filestring1
LOCAL First_char
LOCAL Line_pos
LOCAL String_length
LOCAL First_string
```

这些是局部变量。如果 Keywords_search 由另一个宏调用，两个宏使用的变量的名称之间不会存在发生冲突的危险。

```
LET File1 '\me10\help'
LET File2 '\john\keywords.out'
```

在此宏中，变量 File1 用来表示help文件的完整路径名称。同样地，File2 是将在其中写入关键字列表的文件。

```
OPEN_INFILE 1 File1
```

文件打开后才能在宏中使用。函数 OPEN_INFILE 将打开 File1 以供读取。文件指针将指向文件中的第一个记录。数字 1 是文件描述符。File1 打开后，任何对 File1 的进一步引用都通过文件描述符 1 进行。稍后将会更加清楚文件描述符的使用。文件描述符类似于 MS-DOS“控制滑块”。

请注意，以下语句同样有效，并将产生与之前语句相同的结果：

平台相关性

```
OPEN_INFILE 1 '\me10\help'
```

因此您可能认为变量 File1 实际上没有必要。使用变量 (如 File1) 的好处是可以快速修改宏以读取除 C:\Program Files\PTC\Creo Elements\Direct Drafting [version]\help 外的文件。我们所必须做的是更改行：

```
LET File1 '\me10\help'
```

如果不使用变量 (如 File1)，则必须搜索整个宏，并用新文件的完整路径名称替换每个出现的 \me10\help。我们的宏很短，因此这将会很容易。在较长的宏中，这将花费一些时间。

```
OPEN_OUTFILE 2 DEL_OLD File2
```

打开 File2 以供写入。文件描述符是 2。如果该文件已存在，它将因 DEL_OLD 指令而被覆盖。

```
LET Flag 0
```

Flag 是一个变量，其值为 1 或 0，具体取决于采用两个分支中的哪一个通过宏。显示状态或条件的变量通常称为标志，因此将其命名为 Flag。稍后将会更加清楚 Flag 的使用。此行将 Flag 的值初始化为 0。

```
LOOP
```

启动循环。此循环将继续，直到满足随后的 EXIT_IF 语句或执行 END_LOOP。

```
READ_FILE 1 Filestring1
```

读取文件描述符为 1 的文件。这意味着将读取 File1，其为 /me10/help。如果新打开一个文件，但未读取，文件指针将指向文件中的第一个记录。READ_FILE 函数将每一个完整的行视为是一个记录，因此 READ_FILE 命令将读取文件的第一行，然后推进文件指针以指向下一个记录。下次将读取同一文件，READ_FILE 命令将读取第二行，依次类推。如果要多次搜索文件，并且希望宏每次在文件的开头开始搜索，则必须在每个 READ_FILE 语句之前使用 OPEN_INFILE 语句，以便文件指针指向文件的第一行。

```
EXIT_IF (Filestring1='END-OF-FILE')
```

每个文件均具有一个文件结尾标记。实际标记取决于操作系统，但通常是空字符。如果宏检测到此文件结尾标记，它将从当前循环中退出。

```
LET First_char (SUBSTR Filestring1 1 1)
```

查找在行中的位置 1 处开始、长度为 1 的子字符串。换句话说，查找行中的第一个字符。

作为子字符串的一个示例，我们假设 Filestring1 为：

```
Have a nice day!
```

如果程序语句为：

```
LET Substring1 (SUBSTR Filestring1 3 6)
```

则 Substring1 将为 ve a n。

```
IF (First_char='^')
```

如果第一个字符是 ^，则执行后续语句，直至匹配的 ELSE_IF、ELSE 或 END_IF。(稍后匹配的 ELSE 将出现在第 14 个程序行。)

如果第一个字符不是 ^，跳转到匹配 ELSE 后的第一个语句。

```
IF (Flag = 0)
```

如果 Flag 的值是 0，则执行后续语句，直至匹配的 ELSE_IF、ELSE 或 END_IF。 不存在 ELSE_IF 或 ELSE。稍后，匹配的 END_IF 将出现在第 12 个程序行。

(尚未达到能够解释 Flag 用途的程度。)

```
LET line_position (POS Filestring1 ' ')
```

如前所述，如果行以 ^ 开头，并且行中只有一个关键字，该行将不会包含空格。如果存在多个关键字，这些关键字将会由空格隔开。(我们使用 '' 来指示一个空格。两个空格将为 ''，依次类推。) 该程序语句的目的是查出行是否包含空格。POS 函数将给出子字符串 (第二个自变量) 的第一个匹配项在字符串 (第一个自变量) 中的位置。例如，help文件中的当前行可能为：

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

这里，第一个自变量是以 ^CURRENT_DIRECTORY 开始、以 SM_FILE_2 结束的完整行。第二个自变量是 ' '。因此，Line_pos 将为 19。

当前行可能为：

```
^AUTO_NEW_SCREEN
```

这里，Line_pos 的值为 0。每行均以换行字符结束，因此该行中将不存在空格。

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

此语句将计算完整行的长度。如果该行为：

```
^AUTO_NEW_SCREEN
```

则 String_length 的值为 16。

看下一个程序行时，就会明白为什么需要字符串的长度。

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

之前我们已经见到过 SUBSTR。此语句将提取字符串中起始于位置 2 且长度为 String_length-1 的子字符串。如果该字符串为：

```
^AUTO_NEW_SCREEN
```

则 First_string 为 AUTO_NEW_SCREEN。换句话说，^ 已从行的开头处剥离出来。AUTO_NEW_SCREEN 现在可以写入文件中。

```
WRITE_FILE 2 First_string
```

将 First_string 附加到文件描述符为 2 的文件。此文件为 \john\keywords.out。 将 First_string 视为此文件中的一个完整行。

```
LET Flag 1
```

我们看到如果将行写入输出文件，会将 Flag 设置为 1。 如果您仔细检查代码的其余部分，您将看到，如果从 File1 读取行但不写入到 File2，则会将 Flag 设置为 0。

为什么要这样操作？请记住，我们希望从各个部分提取前面带有 ^ 的第一个字符串。 例如，在help文件的以下部分中，要提取 CANCEL，而不是 ABORT 或 Meview_cancel：

```
=======================================================
^CANCEL ESC STOP INTERRUPT BREAK
^ABORT
^Meview_cancel
CANCEL command
.
.
.
=======================================================
```

为简洁起见，这里给出宏的精简版本，只包含读取和写入语句，并且这些语句引用 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
```

执行完 READ_FILE 语句后，有四种可能的情况：

1. First_char 等于 ^，Flag 等于 0。
   关键字将写入文件。 Flag 设置为 1。
2. First_char 等于 ^，Flag 等于 1。
   未向文件写入任何内容。Flag 设置为 0。
3. First_char 不等于 ^，Flag 等于 0。
   未向文件写入任何内容。Flag 设置为 0。
4. First_char 不等于 ^，Flag 等于 1。
   未向文件写入任何内容。Flag 设置为 0。

因此我们会看到，如果 First_char 不是 ^，将不会向文件写入任何内容。这样很好。这意味着不会向文件写入不是关键字的字符串。

仅当 First_char 是 ^ 且 Flag 是 0 时，才会向文件写入内容。现在，重点是：只有上一行不以 ^ 开头，Flag 才可以为 0。如果上一行以 ^ 开头，Flag 将会为 1。这意味着如果两个或更多连续行以 ^ 开头,将只会将这些行中的第一行上的关键字写入文件。

如果 Flag 为 1，将其重置为 0 的唯一方法是读取不以 ^ 开头的行。

是否注意到只需要一个 Let Flag 1 语句？会将其放在宏中第一个 END_IF 语句之后。通过使用两个语句，并将它们分别放在 WRITE_FILE 语句之后，强调仅当向文件写入内容时，Flag 才会设置为 1。

```
ELSE
```

如果行中有空格，将执行 ELSE 后的代码。

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

在本分支中，行中的第一个关键字前面有 ^，后面有一个空格。因此关键字将从位置 2 处开始，其长度将为 Line_pos-2。我们假设行是：

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

我们要提取的关键字是 CURRENT_DIRECTORY。行中第一个空格的位置是 19。CURRENT_DIRECTORY 的长度是 19-2 或 17。

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

First_string 将写入到输出文件，并且 Flag 将设置为 1。

```
END_IF
```

IF (Line_pos = 0) 的结束 END_IF。

```
END_IF
```

IF (Flag = 0) 的结束 END_IF。

```
ELSE
```

如果 First_char 不是 ^，将执行 ELSE 后的代码。

```
LET Flag 0
```

如果从帮助文件中读取行但未将其打印到输出文件，Flag 将设置为 0。

现在我们接近宏的末尾。最后几行是：

```
END_IF
```

IF (First_char = '^') 的结束 END_IF。

```
END_LOOP
```

循环的结束语句。

```
CLOSE_FILE 1
CLOSE_FILE 2
```

CLOSE_FILE 语句与宏开始处的 OPEN_INFILE 和 OPEN_OUTFILE 语句相对应。

好的编程习惯是关闭所有宏打开的文件。如果不关闭文件，它将保持打开状态，直至下一用户对此文件执行 OPEN_INFILE。操作系统随后将在为下一用户打开文件之前关闭该文件。如果无人使用该文件，它将保持打开状态。但每个操作系统对可同时打开的文件数都有一个限制。达到这一限制时，系统将无法再为其他用户打开更多文件。因此，关闭不使用的文件至关重要。

从宏的内部调用宏

经常会从其他宏的内部调用某个宏。嵌套宏有两个优点：

- 内部宏可以是经充分调试的现有宏。使用现有的宏保存写入时间和调试时间。
- 有时，长的宏理解和调试起来会十分困难。通常可以将长的宏拆分成嵌套在主宏中的较短的宏。如果对内部宏使用有意义的名称，通常可以生成一个“自文档化”宏 - 需要极少备注语句的宏。例如，应清楚以下宏的功能：

```
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
```

让我们通过内部宏替换 Keywords_search 中的某些行。在本例中，通过使用内部宏可能不会获得很多知识。它将解释原理，而我们可以使用该示例来说明如何向宏传递参数。

使用第一个宏中的下列行创建一个内部宏：

```
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
```

要创建称为 Write_to_file 的新宏，您需要做的是将 DEFINE Write_to_file 放在开头，将 END_DEFINE 放在结尾。

在主宏中，必须用包含新宏名称的一个行替换中间的这十行：

```
Write_to_file
```

文件现在将包含以下内容：

```
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
```

新文件说明了这一点：未声明为 LOCAL 的变量将自动成为全局变量。全局变量对这两个宏来说都是已知的。例如，Line_pos 的值由外部宏来计算，由内部宏来使用。Flag 的值由内部宏来计算，由外部宏来使用。

向宏传递参数

在本章中的先前部分，我们讨论了全局变量的问题，以及这些问题有时如何导致有害的副作用。如果编写的宏像水密舱一样封闭，则向宏传递信息的唯一方法是使用参数。例如，我们使用参数重新编写内部宏 (Write_to_file)。

由于 Keywords_search 将不再使用变量 String_length 和 First_string，因此已将其删除。

```
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
```

有关新宏应注意以下方面：

- 除 Flag 外，Write_to_file 中的变量与 Keywords_search 中的变量有所不同。这种情况在从宏内部调用宏时最为常见。内部宏很可能是宏库的一部分，使用它可避免您自己编写新代码。宏的规范将指明该宏的功能及必须传递的参数。您将不必关注宏的内部运作或使用的变量。
- 在 Write_to_file 中，参数可以任何顺序进行声明。但当 Keywords_search 调用 Write_to_file 时，必须以正确的顺序放置自变量。调用 Keywords_search 的语句是：

  ```
  Write_to_file Line_pos Filestring1 2
  ```

  在本语句中，请注意以下几点：

  - Line_pos 与 Column 对应。

- Filestring1 与 String 对应。
- 2 与 File_descriptor 对应。

调用宏 Write_to_file 时，系统知道宏名称后的第一个参数代表被调用宏中第一个声明的参数，宏名称后的第二个参数代表被调用宏中第二个声明的参数，以此类推。

请注意，可以到向被调用宏传递参数，但不能向调用宏传递参数。在所有编程语言中，宏将在编译时得到扩展。这意味着将替换宏名称的宏代码。某个宏调用另一宏时，在编译时将在调用宏中成行替换代码。随后就像只有一个宏一样执行生成的代码。

如果成行替换 Write_to_file 而无 PARAMETER 语句，编译器可能会抱怨某些变量未知。PARAMETER 语句会告知编译器一些对变量对是等效的。但 PARAMETER 变量的行为与 LOCAL 变量类似：它们只在对其进行声明的宏中可见。因此，宏不同于在其他编程语言 (例如 PL/I 或 Fortran) 中使用的函数和子例程。在这些语言中，程序将转移到被调用的函数或子例程，这些函数或子例程随后可将值返回到调用程序。

由于执行 Write_to_file 后需要知道 Flag 的值，因此尚未将 Flag 声明为本地变量或 Write_to_file 中的参数。必须保持 Flag 作为全局变量，使其对两个宏均可见。

输出文件 keywords.out 将按在help文件中出现的顺序列出关键字。

平台相关性

如果要排序这些字，最简单的方法是使用 MS-DOS sort 命令。在 Windows 环境中，使用“程序管理器”打开命令提示符，然后键入：

```
sort < keywords.out > keywords.srt
```

keywords.srt 是排序文件的名称。如果 keywords.out 不在当前目录中，必须使用文件的完整路径名称。

8

使用数据文件中存储的尺寸

宏的功能 83
介绍栓 83
矢量分析 84
介绍数据文件 85
分析宏 86
细化宏 88

本章介绍从列表数据文件中提取尺寸的宏。该宏随后将会在屏幕上的指定位置绘制零件。

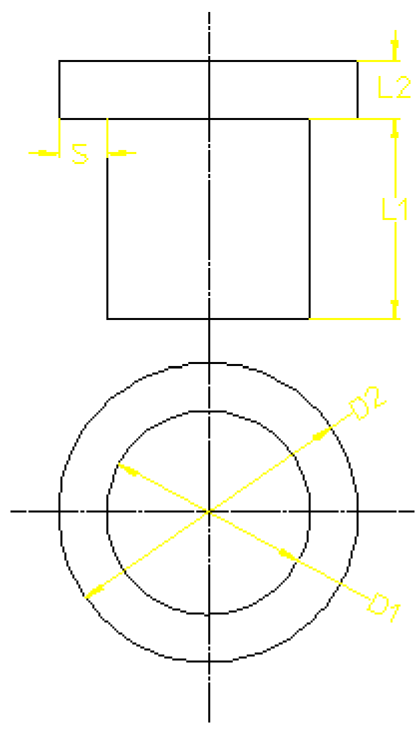
宏的功能

- 系统会提示用户在孔面的圆周上数字化两个点。在简单宏中，这两个点是您查看孔时右侧和左侧的点。
- 通过使用这两个点，宏计算该孔的直径。
- 宏会搜索文件，直至计算出的孔直径与第一列中的值相匹配。行中其余的数据随后用于计算栓 (在孔中摩擦配合) 的尺寸。
- 该宏将在正确的位置并以正确的姿态绘制栓。
- 系统再次提示用户数字化两个点，或 END。

介绍栓

下图显示栓的常规尺寸：

图 **13.** 栓 **-** 常规尺寸

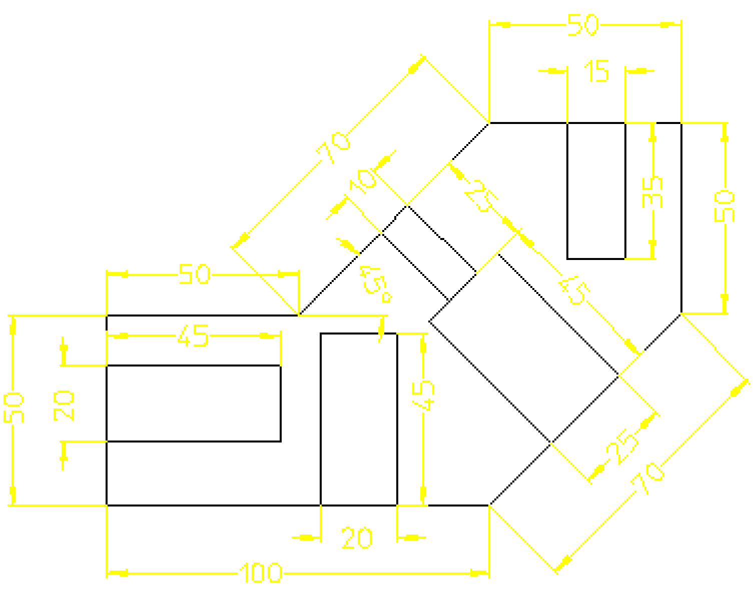


尺寸 S (栓肩部的宽度) 可通过 1/2(D2-D1) 计算得出。

您可以看到栓的几何十分简单。但是，我们在本章中讨论的宏可适用于任何复杂的几何图形。

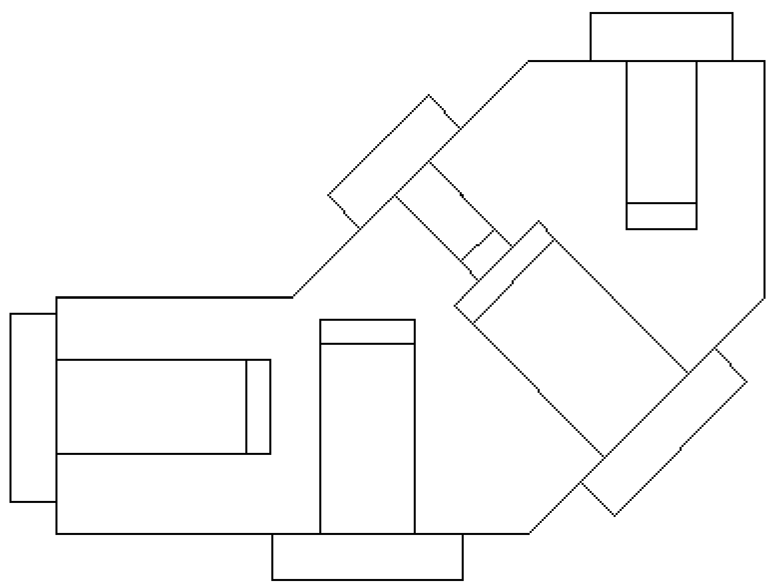
要测试宏，可能需要绘制一个类似下面剖视图的块，该剖视图显示处于多个不同姿态的多个不同直径的孔。

图 **14.** 测试块的剖视图



使用宏在各个孔插入合适的栓后，应看到下图：

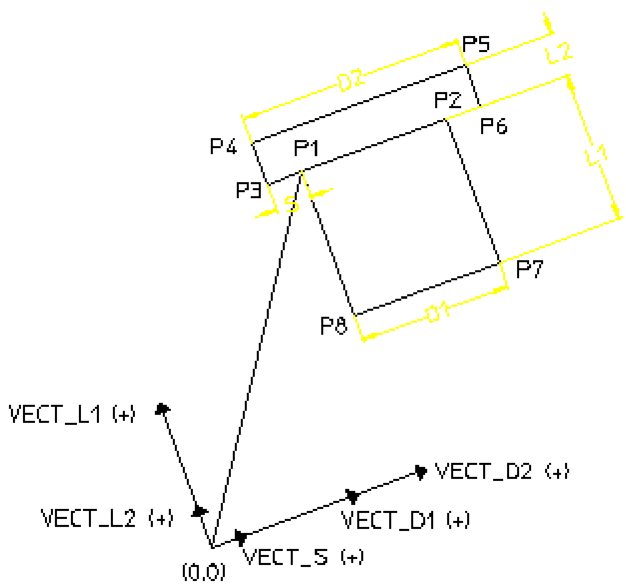
图 **15.** 在适当位置上的栓



矢量分析

下图显示了常规位置上的栓。P1 到 P8 均为起点在原点的矢量。为保持绘图简洁，只显示 P1 的源自原点的线。也显示了源于栓尺寸的矢量：

图 **16.** 栓 **-** 常规尺寸



D1 和 L1 (请参阅第85页上的“图 16. 栓 - 常规尺寸”) 之类的尺寸无法用于矢量分析：矢量需要方向和长度。但是，如果角度已知或可以通过计算得出，将尺寸转换为矢量则十分简单。

让我们首先采用尺寸 S。S 作用的角度与 (P2-P1) 的角度相同。P2 和 P1 是矢量，因此矢量 (P2-P1) 的角度可通过 ANG(P2-P1) 计算得出。ANG 是内置函数。

如果将源于 S 的矢量称为 Vect_S，则：

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

其中，PNT_RA 是将长度和角度转换为矢量的内置函数。

要将 L1 和 L2 转换为矢量，请注意角度将比 (P2-P1) 的角度大 90 度。例如：

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

以类似方式定义另一矢量。请注意，在第85页上的“图 16. 栓 - 常规尺寸”中，表示尺寸的这些矢量在所示方向上都将是正的。在第85页上的“图 16. 栓 - 常规尺寸”中，我们在任意位置绘制栓。这意味着在许多位置，尺寸的矢量实际上将会是负的。但这并不重要。只要使用一致的“符号约定”，则无需担心矢量的符号。

介绍数据文件

宏使用的数据文件称为 \anna\spigot.dat。 文件中的数据为；

```
10 30 20
15 30 30
20 40 40
```

```
25 40 40
```

从左到右，数据分别代表尺寸 D1、D2 和 L1 (以毫米为单位)。这些尺寸是任意选择的，因此值之间并不存在固定关系。如果存在固定关系，例如 D2=3xD1 和 L1=2xD1，在宏中编写方程将更加容易。

分析宏

宏显示在下面的行中。在上一章“文件输入/输出和文本字符串”中，您已经看到过许多命令和函数。

```
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
```

让我们从下面的行开始分析：

```
LET Hole_diam (ABS(P2 - P1))
```

我们搜索数据文件时，将会查找等于 Hole_diam 的 D1 值。由于 (P2-P1) 可能为负，因此我们取绝对值以便能够检查是否相等。

```
LET D1 (VAL Data_1)
```

Data_1 是文本字符串。我们希望可以将 Data_1 与 D1 (其为数字) 进行比较。因此将 Data_1 转换为数值。

```
EXIT_IF (ABS(D1 - Hole_diam) < 0.0001)
```

我们要搜索文件，直至找到等于 Data_1 的 D1 值。您可能认为我们会使用如下所示的等式语句：

```
EXIT IF (D1 = Hole_diam)
```

在任何编程语言中，由于精确度问题，在这种情况下程序员会避免使用相等语句。比如在第84页上的“图 14. 测试块的剖视图”的左侧以数字化方式创建 20 mm 直径的孔。计算机可能会将孔直径计算为 20.0000000000001 mm。当宏读取数字文件中的第三行时，使用trace工具看到的语句将是：

```
EXIT IF (D1 20 = Hole_diam 20.0000000000001) 0
```

该行末尾的 0 指示“假”，因此不相等。

根据在宏中使用的实际语句，如果 D1 与 Hole_diam 之间的差异可以接受，我们将接受 D1。

```
LET P3 (P1 - Vect_S)
```

第85页上的“图 16. 栓 - 常规尺寸”显示 Vect_S 的正方向。要获得 P1 到 P3 的矢量，必须朝向负方向。 因此，P3 = P1 – Vect_S。 如果我们将 Vect_S 的正方向定义为左侧向下，则正确的方程应为 P3 = P1 + Vect_S。 这表明只要遵循您自己的符号约定，将始终能够确定矢量的符号。

```
LET P4 (P3 + Vect_L2)
```

要获得 P3 到 P4 的矢量，需朝向 Vect_L2 的正方向，因此加上 Vect_L2。 对于其余矢量，加还是减的决定还取决于已选择作为正方向的方向。

细化宏

该宏尽可能地保持简单以说明原理，并尽可能减少您的打字。存储在文本文件中的标准零件数据可以针对简单零件 (例如垫圈、垫片和双头螺栓) 或非常复杂的零件 (例如机械密封和活塞装配)。原理是相同的。

即使用于绘制栓，该宏都很粗糙。以下是一些明显的改进：

- 在我们的宏中，必须以正确的顺序数字化 P1 和 P2，因此宏将知道栓的正确姿态。如果以错误的顺序数字化这两点，则会将该栓绘制为正确位置的镜像图像 (请尝试！)。您可以重新编写宏，以使用户必须数值化第三个点 (例如，孔底部的点)，以便宏知道孔的姿态。
- 对于栓来说，我们需要摩擦配合，因此栓的直径将会与孔的直径相同。在需要松配合的情况下，宏必须能够集中孔中的项。
- 在文件 spigot.dat 中，每个数据项由两个数字组成，并且该数据是以每个数据项之间只有一个空格的形式安排的。通常情况下，表格数据项的长度不同，各列将右对齐。因此，每个数据项之间会有不同数量的空格。您的宏必须考虑到这一点。
- 就简单栓来说，针对 D1 的每个值只有一行数据。如果针对 D1 的每个值有多行数据 (例如，针对 D1 的每个值的多个 D2 值)，必须在循环内添加一个循环以搜索所需的 D2 值。
- 就 P1 和 P2 来说，假定 (P1-P2) 的值对应于文件 spigot.dat 中 D1 的值。如果用户输入两个不满足条件的点 P1 和 P2，则会执行循环，直至读取到字符串 END-OF-FILE。后续执行将停止。针对此情况，应添加一行，可在读取到 END-OF-FILE 时退出循环，并显示消息以告知用户该输入不正确。

9

有用的宏

绘制与现有线成角度的构造线......90
将线分割为相等段......90
绘制圆端槽......91
绘制多边形......92
围绕圆形对象调整文本......92
显示隐藏线绘图的不同 Z 级......93

本章包含在 CAD 操作期间可能有用的一些宏。即使不想使用宏，通过学习宏并尝试理解它，仍可以学到很多相关知识。

如果只希想尝试这些宏，可以省略本地变量和备注，从而节省打字时间。

绘制与现有线成角度的构造线

下面的宏将绘制一条以指定角度与现有线相交的构造线。现有线可以是构造线或几何线 (实线或虚线)。

```
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
```

将线分割为相等段

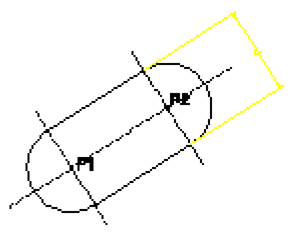
该宏会将一条线 (实线或虚线) 分割为指定数量的相等段。然后宏会绘制通过分割点的构造线。这些构造线可以是水平的、竖直的或垂直于现有线。

```
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
```

绘制圆端槽

该宏将绘制任意尺寸、与轴成任意角度的圆端槽。




```
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
```

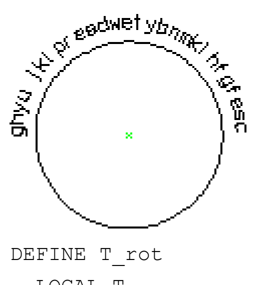
绘制多边形

该宏将绘制带有任意数量边及带有内切和外接圆的正多边形。

```
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
```

围绕圆形对象调整文本

该宏将围绕圆形对象调整文本字符串。



```
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
```

显示隐藏线绘图的不同 Z 级

该宏将显示绘图中的各个 z 级，一次显示一个。逻辑表是在内存中创建的，用于存储必需的信息。

使用该宏之前，必须激活隐藏线模块，并且必须使带有所分配 z 级的绘图显示在屏幕上。

```
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

记录系统操作

ECHO 函数..96
使用 ECHO 创建宏 96

本章介绍如何存储某个时间段内产生的所有系统输入的记录。此功能可用于：

- 快速创建固定尺寸几何宏。
- 演示系统操作 - 看不见的绘图员！

可执行此操作的两个函数：ECHO 和 TECHO。 我们将在以下部分概述每个函数的功能。

ECHO 函数

此函数会将所有系统输入以下面所示的形式写入文件 (或打印机)：

```
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
```

会记录使用的所有命令和函数的名称，以及来自键盘的数字输入。

要启动 ECHO 文件，键入以下内容：

```
ECHO 'echofilename'
```

其中 echofilename 是为 ECHO 文件选取的名称。

此后将记录系统输入。

要关闭 ECHO 文件，键入以下内容：

```
ECHO OFF
```

要重播 ECHO 文件，键入以下内容：

```
INPUT 'echofilename'
```

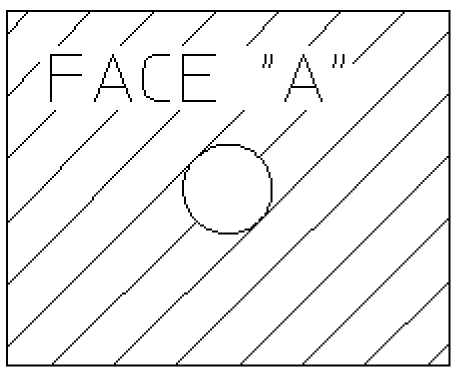
其中 echofilename 是先前指定给 ECHO 文件的名称。

使用 ECHO 创建宏

您可以使用 ECHO 文件作为创建宏的基础。但对于某些类型的宏，使用 ECHO 您会一无所获。例如，我们在第82页上的“使用数据文件中存储的尺寸”中看到的栓宏几乎完全与矢量分析和读取文件有关。手动执行时涉及到许多数字化和更改菜单操作的任务最适合 ECHO。

请看下图中的剖面线块：

图 **20.** 使用 **ECHO** 创建的块



以下是创建块期间记录的 ECHO 文件的内容：

```
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
```

构造线用于查找矩形对角线交点，以便可以直观地定位圆的中心。在宏中，中心的坐标将已知，因此不需要构造线。

可以编辑 ECHO 文件，对点使用我们自己的坐标。

```
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
```

要创建宏，执行以下操作：

- 将 DEFINE 'filename' 置于开头，将 END_DEFINE 置于结尾。
- 移除对菜单的引用。例如，移除 TM_CREATE_1、Tm_text_1 和 Tm_hatch。
- 移除 echo off。
- 用 END 替换 TM_END。

请注意，与手动处理时相比，在宏中更多地使用 END。手动处理时，您可以通过启动新命令来终止递归命令或函数。

11

使用界面查找命令

您会使用哪些命令？ 100

在本章中，我们“回顾”用户界面以查看每次在屏幕菜单中拾取项时激活的命令。这会帮助您将在用户界面上熟悉的操作与不熟悉的宏命令或函数联系起来。

您会使用哪些命令？

开始编写宏时，您会使用哪些命令和函数？我们在第95页上的“记录系统操作”中看到，启动宏的好方法是使用 ECHO，然后修改 ECHO 文件。 这适用于“绘制”宏，使用户能够免于执行许多数字化和菜单更改操作。但许多宏只包含少量实际显示可见结果的命令，例如绘制线或创建新视区。其余命令将执行计算和矢量分析。第82页上的“使用数据文件中存储的尺寸”中的宏栓就是一个很好的例子。对于此类宏而言，ECHO 对您没有多大帮助。

如果要在宏中创建视区，您会使用哪些命令？问问自己：“在用户界面上如何执行此操作？”。如果能够查出按创建时激活的内部命令，该命令即为要在宏中使用的命令。

屏幕命令和函数

以下是查找屏幕菜单槽背后的命令的方法。我们以“信息”菜单为例。在命令行键入：

```
EDIT_MACRO
```

系统对提示做出响应：

```
Enter macro_name or ALL
```

现在按信息。以下宏会显示在屏幕中：

```
DEFINE Tm_info
  Sm_info
END_DEFINE
```

现在键入：

```
EDIT_MACRO Sm_info
```

宏 Sm_info 会显示在屏幕中。如下所示：

```
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
    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
```

您可以看到行 1，列 1 的显示文本为 ADD SELECT。 相应的命令 (有时称为“操作文本”) 为 ADD_ELEM_INFO。 行 8，列 2 的显示文本为 Current。相应的命令为 EDIT_CURRENT_INFO。

使用此方法查找任意屏幕菜单的内容。

摘要

总之，如果要知道要在宏中使用哪些命令和函数以执行特定任务：

- 决定如何使用用户界面执行任务。
- 查出用户界面“背后”的命令。
- 如有必要，阅读在线帮助中该命令的说明。
- 在宏中使用该命令以及正确的选项。

12

自定义

什么是 Creo Elements/Direct Drafting 环境？ 103
自定义 Creo Elements/Direct Drafting 环境.......... 104
屏幕菜单的创建方式.......... 104
自定义屏幕菜单.......... 108
自定义本地目录.......... 109
自定义键盘 112
什么是文本字体?.......... 114
如何创建文本字体 118
自定义启动过程.......... 121
自定义剖面线图案 124
键盘输入字符 125
键盘输入字符 126

Creo Elements/Direct Drafting 启动时会读取并执行一些定义了各种函数 (如屏幕菜单和 Creo Elements/Direct Drafting 环境) 的文件。这些文件是称作宏的可执行程序，您可以对其进行编辑并以自己的文件名存储，以便根据需要重新加载和使用。因此可以为屏幕菜单和 Creo Elements/Direct Drafting 环境创建自己的替代定义。此过程称为自定义。

什么是 Creo Elements/Direct Drafting 环境？

Creo Elements/Direct Drafting 环境包含诸如屏幕上的构造线或绘图线的颜色、尺寸文本的大小和颜色、绘图时是否分割绘图线、绘图比例和单位等内容。

这一切都由环境函数设置控制，系统中有 70 多项此类设置。环境函数的一个示例为 DIM_COLOR。可对此进行设置，使用任何可用颜色生成尺寸。

环境函数的当前设置可存储在文件中。以下列出了 Creo Elements/Direct Drafting 环境文件的部分内容：

```
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
```

启动时，会如列表所示设置函数以提供标准环境。正常使用过程中，可根据需要在屏幕菜单中更改设置。

也可以通过编辑 Creo Elements/Direct Drafting 环境来更改函数设置。此方法很有用，因为它可以使您一次概览所有环境函数的状态。

自定义环境可存储在指定文件中。然后在需要时，可使用此文件恢复相同环境。

自定义 Creo Elements/Direct Drafting 环境

将显示环境，并且可根据需要更改函数设置。完成环境更改后，按 [CTRL] [D] 对系统实施所有更改。通过从屏幕菜单中拾取保存环境选项并输入以下内容可将当前环境存储在指定文件中以供今后使用：

```
'env_filename'
```

其中 env_filename 是所选项的名称。

要执行存储的文件和重新定义系统环境，请输入：

```
INPUT  'env_filename'
```

屏幕菜单的创建方式

屏幕布局由一个可从中获取命令的菜单和一个可从中生成绘图的工作区域组成。菜单布局提供了一些可从中直接调用命令的活动区域 (称为槽)。与这些命令关联的名称和记忆信息可以显示在槽中。

典型的菜单列表显示在第99页上的“使用界面查找命令”开头。研究列表就会发现：对于每行和每列，“显示”文本均出现在“操作文本”的前面。

通过输入 EDIT_MACRO 并拾取其中一个屏幕菜单小键盘，可找到用于显示屏幕菜单的系统宏的名称。将显示如下内容：

```
  DEFINE action_text
    action
  END_DEFINE
```

在本例中，操作即是用于显示屏幕菜单的宏的名称。可以编辑其中的每个宏并将其存储在文件中，以供今后使用。

此处介绍了如何保存用于显示屏幕菜单的宏。让我们使用 CREATE 1 菜单作为示例：

1. 使用 EDIT_MACRO 从屏幕菜单中获取宏名称。您将发现 CREATE 1 菜单的宏为 Sm_create_1。
2. 在命令行中键入：
   ```
   SAVE_MACRO Sm_create_1 'filename'
   ```

filename 是所选项的名称。

您现在可以对该文件进行编辑。您可能希望将该文件的内容添加到 customize 文件。

菜单变量

以下是 CREATE 1 菜单的前几行，显示了一些菜单变量：

```
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
```

其中每个变量的含义如下：

- I_port

  下一个变量 I_port 与之前使用的输入板上的端口下的大/小小键盘相关。当拾取大/小时，将执行宏 Tm_port_large_slash_small 并存储当前视区的坐标。随后会放大当前视区，以使整个屏幕都可用于创建几何。同时，将变量 I_port 设置为 1，以通知系统以下信息：当前正在显示一个较大的视区，如果某个菜单小键盘被拾取，将不执行任何操作。再次拾取大/小时，当前端口减小到原始大小，并且变量 I_port 被设置为 0，以便可以正常更改菜单。

- CURRENT_MENU Sm_create_1_layout_name

  这会将当前屏幕菜单的名称设置为 Sm_create_1 (Sm_create_1_layout_name 是将被展开为该字符串的宏)。所有菜单布局和菜单命令都将与此当前菜单相关，直到使用一个具有不同名称的当前菜单为止。要定义您自己的菜单，建议使用 '' (一个空字符串) 作为布局名称。

- T_clear_menu

  该宏会从菜单中清除所有文本，以便可以输入新文本。

- Menu_control_icons

  该宏会插入一些图标，用于将菜单锁定、移动和移除到菜单的第一行。

- MENU Colo0 Bcol5 CENTER 'CREATE 1' 1 3

  此行完成了第一个菜单行。Colo0 和 Bcol5 将展开为一个前景颜色和一个背景颜色，它们是在系统启动时根据 MEPELOOK 定义的。所有系统定义的菜单都使用这些宏来指定前景及背景颜色。下表大体介绍了当 MEPELOOK 被设置为 0 时这些宏如何与菜单颜色相关。

前景	背景	颜色 **(MEPELOOK=0)**
Colo0	Bcol0	黑色
Colo1	Bcol1	白色
Colo2	Bcol2	红色
Colo3	Bcol3	绿色
Colo4	Bcol4	蓝色
Colo5	Bcol5	黄色
Colo6	Bcol6	洋红色
Colo7	Bcol7	青色

- Eight_menu_slots_add

  Eight_menu_slots_add 是一个宏，可更新最后 8 个可用户自定义的槽。它用于每个标准菜单中。可以使用 Eight_menu_slots 宏将自己的菜单项放在这 8 个槽中。该宏的语法为

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

  字符串 title 将显示在槽中，action 是在按下菜单槽时执行的操作文本。Row_number 和 Column_number 是相应的行编号和列编号。行编号的有效范围为 26 到 27，列编号的有效范围为 1 到 4。

菜单槽的物理布局

您可能已经注意到，除几个特例以外，所有屏幕菜单都具有相似的行列布局。这是因为大多数菜单都使用了同一个“模板”。该模板是使用名为 Layout_body_1 的宏创建而成的。例如，该宏可用于定义 Sm_create_1 菜单的布局：

```
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
```

这是在上述宏中所用变量的说明：

- `Headline_height`

  菜单标题槽的高度。

- `Text_slot_height`

  包含显示文本的槽的高度。

- `Bottom_slot_height`

  位于各个菜单底部的小方框的高度。

在启动期间计算槽的高度值。这样可以调整菜单大小，使其适合启动 Creo Elements/Direct Drafting 时所在的窗口或显示内容的大小。此类值在 hp_macro.m 文件中定义。要了解如何使槽大小进行自调整以适合不同的窗口和显示内容大小，可以参考这些用作示例的定义。

此模板宏在定义 CREATE 1 菜单布局的以下宏中使用：

```
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
```

这是在上述宏中所用变量的说明：

- `CURRENT_MENU 'Sm_create_1'`

  这会将 CREATE 1 菜单布局的名称设置为 Sm_create_1。

- `CURRENT_SCREEN 1`

  这样可确保即使在双屏幕环境中也会将菜单放置在第一个屏幕中。

- `MENU_LAYOUT`

  MENU_LAYOUT 命令可使菜单显示在屏幕的右侧。MENU_LAYOUT 命令通过 END 终止。

- `Menu_position`

一个包含限定符 LOWER 的宏。可根据需要将其设置为 UPPER。

- Layout_body_1

  此行调用宏 Layout_body_1。

- Menu_home_point_top

  将菜单移至屏幕上的正确位置。此位置根据 Menu_position 宏和用户界面版本 (仅屏幕) 进行计算。

- MENU_STATUS ENABLE_INQ

  这会从菜单中移除查询保护。

- LET Sm_create_1_layout_name 'Sm_create_1'

  CREATE 1 菜单不会直接使用布局名称。要激活其布局，请使用 Sm_create_1_layout_name 变量。因此，必须在此处将 Sm_create_1_layout_name 定义为正确的字符串。当需要在首次请求菜单之前推迟菜单布局的实际生成时，这十分有用。在首次请求之前，Sm_create_1_layout_name 将定义用于生成布局的宏名称。

使用屏幕菜单表格

一些屏幕菜单是通过 TABLE 系列命令定义的，如 TABLE_LAYOUT 和 SHOW_TABLE。之所以使用表格，是因为它们可以显示系统状态变量。通过滚动表格，可以显示较长的列表。以下是使用表格的菜单的示例：

- HIDDEN LINE

这些菜单是使用 MENU 和 TABLE 命令创建而成的。

可以在文件 hp_macro.m 的 ASCII 版本和 hp_men_t.m 中查看到这些菜单的定义。

使用现有菜单的副本执行更改或其他操作。

使用表格定义的菜单受保护，无法对其进行删除或覆盖。这可避免因 DELETE_TABLE_ALL 等命令导致的潜在问题。

如果要使用自己的菜单表格，应将之前提到的文件中的 SECURE_TABLE 语句移至表格定义的末端。不应更改默认菜单的逻辑表格，因为它们是通过 Creo Elements/Direct Drafting 代码进行访问的。

有关表格使用的详细信息，请参阅在线帮助。

自定义屏幕菜单

可通过以下几种方式自定义屏幕菜单：

- 可使新命令在现有菜单中处于可用状态。添加显示文本，然后插入新命令作为操作文本。
- 可以从空菜单槽中激活一个宏。添加显示文本，然后插入宏名称作为操作文本。
- 可使用现有菜单布局创建新菜单。
- 可使用新菜单布局创建新菜单。
- 可根据需要定义和调用自定义的视区布置。

自定义本地目录

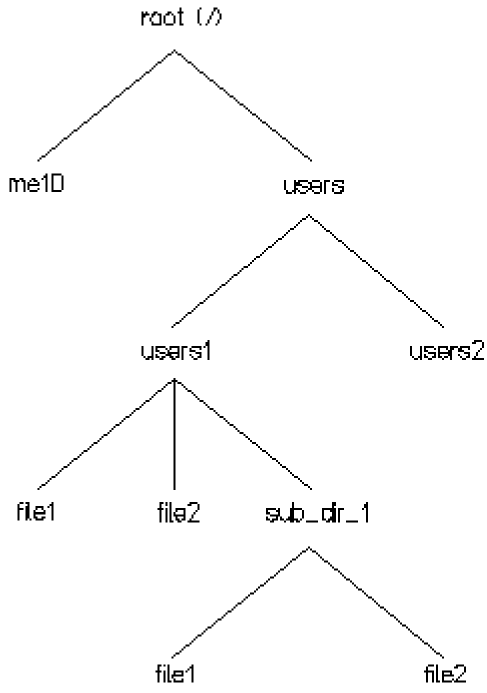
自定义 Creo Elements/Direct Drafting 时，需要将多个文件 (例如，包含您自己的宏、自定义设置、几何和主体的文件) 存储在系统硬盘上。通常将这些文件存储在您自己的用户目录 (又称为主目录) 中。此目录独立于其他用户的目录。

平台相关性

如果您在主目录中启动 Creo Elements/Direct Drafting，则除非指定其他目录，否则所有文件均将存储在此目录中。您可以在 Windows 环境中安装 Creo Elements/Direct Drafting，以便始终从主目录中启动。请参阅与操作系统手册对应的配置手册。

第110页上的“图 23. 目录层次结构”显示了一个典型的目录层次结构。您的系统可能会稍有不同。

图 **23.** 目录层次结构



位于树顶部的是根目录，用斜杠 (/) 表示。位于其下的是名为 users 的目录。位于 users 下方的是与您的登录名对应的您自己的用户目录以及其他用户的目录。您自己的目录 (又称为主目录) 可以包含文件和子目录。通常，您会将文件存储在自己的用户目录或一个子目录中，但也可以将其存储在其他用户目录中，前提是这些目录未被写保护。

位于根目录下的是名为 me10 的目录，其包含所有 Creo Elements/Direct Drafting 软件。其他文件和目录供操作系统使用。您不需要关注这些目录。

位于树结构的同一个分支中的文件和目录使用斜杠分隔开，第一个斜杠对应于根目录。用户目录中的文件的完整规范为：

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

其中 user_directory 是与您自己的登录名对应的目录，而 filename 是您自己的一个文件。

注意

在 Creo Elements/Direct Drafting 内指定路径名称时，可以使用斜杠 (/) 或反斜杠 (\) 分隔文件和目录。以下两个路径指定了同一个文件：

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

首次登录时，您自己的用户目录称之为当前目录。如果您只是简单地通过给定文件名来指定某个文件，则系统将在当前目录中查找该文件。如果当前目录中没有该文件，则系统将在me10\me10 目录或在 SEARCH 路径中指定的目录中进行查找。SEARCH 路径在环境文件中指定。

如果要指定一个不存在于当前目录中且不是通过 SEARCH 路径指定的文件，则必须指定从根目录开始的完整路径名称。

通过在 Creo Elements/Direct Drafting 命令行中输入下列命令，可以在当前目录下创建子目录：

```
CREATE_DIRECTORY 'subdirectory'
```

subdirectory 是所选项的名称。

要使子目录成为当前目录，可使用"文件"屏幕菜单中的"当前目录"选项并指定新目录的名称。如果是在 subdirectory 中创建文件，则该文件的完整规范将为：

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

接下来，我们将了解如何生成自定义菜单，以便可以在目录之间进行更改。

示例 - 创建您自己的目录菜单

使用"文件"屏幕菜单中的"编辑文件"选项创建一个具有所选文件名的文件，然后输入宏的文本。下面是一个示例：

```
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
```

第一条 MENU 语句设置菜单标题。第二条 MENU 语句使用黑色背景将标题下的所有框定义为空，但同时将它们定义为当有一些文本写入其中时立即显示白色文本。

第三条 MENU 语句定义一个表示用户目录的槽。该槽之后的槽表示包含 Creo Elements/Direct Drafting 等软件的目录。

请注意，所有 CURRENT_DIRECTORY 语句指定的都是以根目录 (/) 开头的目录的完整路径名称。这样可确保始终能够找到这些目录，无论它们处于目录树结构中的哪个位置。

MENU 语句的列表可继续延伸，直到菜单布局中的槽被填充为止。

位于每条 MENU 语句末端的编号表示槽的行编号和列编号。

在本例中，最后一条 MENU 语句定义了一个用于在屏幕上显示当前目录的槽。显示菜单时，您可以通过拾取任何指定槽来更改当前目录，然后通过拾取 CATALOG 来显示该目录。

如果您输入文件并运行宏，则会显示菜单。可以通过拾取指定槽来选择目录。

自定义键盘

使用 Creo Elements/Direct Drafting 时，某些命令的使用可能会更加频繁 (相对于其他命令而言)。本节将介绍如何自定义键盘以减少调用其中一个命令所需的时间。

您的键盘有 8 或 12 个功能键，分别标记为 f1 到 f8 或 f1 到 f12。所有这些功能键都可以进行自定义。

例如，您可能希望使用一些当前屏幕菜单中不存在的命令。要避免为发出某个命令而必须对屏幕菜单进行更改，可以自定义一个功能键，以便在每次按下此功能键时，系统都会将所需命令写入输入行或直接执行。请参阅在线帮助中有关 DEFINE_KEY 的说明。

还可以使用功能键来显示那些无法在键盘上获取的字符。每个字符都有一个与之关联的编号，称之为 ASCII 编号。编号的范围介于 0 到 255 之间，其中多数编号表示可读字符，但其中一些编号用于控制功能，如 ESC、RETURN、TAB 和 DEL。在标题“键盘输入字符”下方的本章节末尾提供了一个列表，该列表包含可生成有可用文本的所有字符。

无法用于文本的字符是指那些 ASCII 编号为 0-31、127-159 和 255 的字符。

通过输入以下内容可在输入行上显示任何文本字符：

```
DISPLAY (CHR ascii_number)
```

其中 ascii_number 是字符的 ASCII 编号。

要自定义一个功能键，以便可以通过它来键入某个字符，请输入以下命令：

```
DEFINE_KEY key_number (CHR ascii_number)
```

其中 key_number 是一个介于 1 到 8 或 1 到 12 之间的编号，用于表示要自定义的功能键。下一次按下该功能键时，对应的字符将显示在用户输入行上。

注意

如果自定义一个功能键并将其用于通过文本、符号或尺寸菜单将文本放置在绘图上，则绘图上的文本将取决于文本字体的自定义方式，其样式将不必与出现在输入行的字符样式相同。请参阅“自定义文本字体”中的下一节。

要自定义一个可用于发出命令的功能键，请输入以下内容：

```
DEFINE_KEY key_number 'action_text'
```

其中 action_text 是所需命令的名称。当按下此键时，命令名称会出现在输入行上。要执行该命令，请按 [Return]。

需要在命令名称的两侧加上引号，因为就像在输入行上键入命令一样，系统会要求您输入文本。自定义一个用于生成字符的功能键时不需要加引号，因为系统会将 (CHR ascii_number) 视为文本。

还可以从文件自定义功能键。从文件屏幕菜单中选择编辑文件，然后在引号内输入要将文件存储在其下的磁盘上的文件名。键入所需的说明，然后使用 [CTRL] [D] 存储该文件。

现在，请选择输入并再次输入文件名。将自定义在该文件中定义的所有功能键。

如果从键盘或文件自定义功能键，则功能键定义将在系统关闭时丢失。将 DEFINE_KEY 说明存储在文件中的好处在于，每次启动系统和要自定义功能键时，只需输入您的文件即可。

以下列表是自定义了 8 个功能键的文件的示例。

```
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'
```

- 前三个键被自定义为生成绘图上的 hp_symbols 字体的字符 ° (度)、直径符号和 ± (加/减)。
- 键 4 和 5 被自定义为加载和存储绘图。
- 键 6 用于编排屏幕上的当前目录。键 7 用于输入文件。

- 键 8 被自定义为输入文件，并被自定义为定义第一个用户定义的宏小键盘，以便能够在拾取该小键盘时执行宏。在本例中，dir_file 即是在上一节“自定义本地目录”中描述的文件。它包含名为 directory 的宏，执行该宏时会在屏幕上显示用户目录的菜单。假设文件存在于当前目录中，并且文件名和宏名称与 DEFINE_KEY 语句中的相同，那么，当按下此功能键之后再按下 [Return] 时，只需拾取第一个用户定义的宏小键盘，即可显示用户目录的菜单。

注意

如果将一个功能键自定义为执行多个按系列排序的宏，则最后一个宏可包含一条 READ 语句，以提示用户进行输入。这是因为系统会将列表中的下一个单词视为 READ 语句的输入。

什么是文本字体?

文本字体是一组可放在绘图上的预定义几何图案。字体中的每种图案都由连接栅格上各点的直线组成，并与键盘字符关联以便于调用。

尽管文本字体通常用于文本 (标准系统注释和标注文本以这种方式生成)，但也可用于生成特殊符号。由于栅格中可有许多个点，因此可定义极其详细的符号。

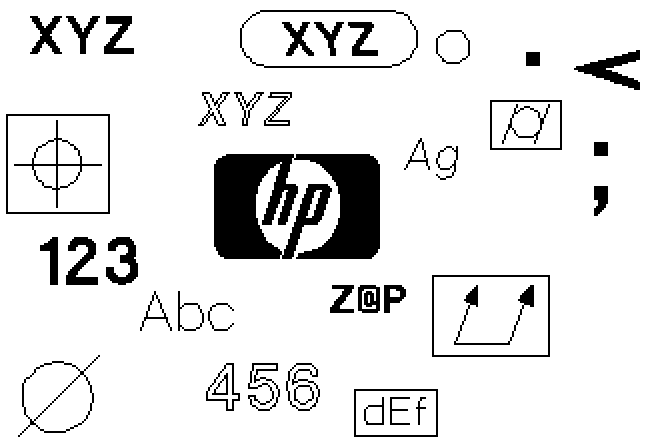
如果需要特殊文本或符号，可以使用自己自定义的文本字体创建并将其存储在磁盘上以备使用。例如，可能希望创建一个用于生成公司商标和徽标的文本。

文本字体的功能非常丰富。其中一些主要功能为：

- 可填充或不填充字符。
- 可设置字符宽高比。
- 可设置字符倾斜角度 (向前或向后)。
- 可设置线角度 (0 到 360 度)。

以下是一些文本和符号的示例：

图 **24.** 文本和符号示例



系统字体

启动 Creo Elements/Direct Drafting 时，系统会将多个字体文件加载到内存中，但作为当前字体并可立即投入使用的只有一种字体。通过使用“文本 2”菜单中的“列出字体”并指定屏幕选项，可以查找出内存中存在哪些字体。

系统随即会显示一个列表，其中列有已加载字体、已使用字体和当前字体。此列表与从中创建这些字体的文件无关。字体 `hp_Y14.5` 和 `hp_i3098_v` 加载自同名文件，但 `hp_def_font` 是在系统内创建的，而且绘制前在屏幕上显示一个框，表示字符将占用的空间。

磁盘上的某些可用文件会在加载时创建字体。

下表提供了 Creo Elements/Direct Drafting 和 DOS 字体名称，以及每个 Creo Elements/Direct Drafting 绘图字体的说明：

Drafting 字体名称	**DOS** 文件名	说明
hp_block_c	hp_blk_c.fnt	填充字体，恒定字符间距
hp_block_v	hp_blk_v.fnt	填充字体，可变字符间距
hp_d17_c	hp_d17_c.fnt	DIN17 字体，恒定字符间距
hp_d17_v	hp_d17_v.fnt	DIN17 字体，可变字符间距
hp_jasc_c	hp_jas_c.fnt	日语汉字和希腊语字体，恒定字符间距
hp_jasc_v	hp_jas_v.fnt	日语汉字和希腊语字体，可变字符间距

Drafting 字体名称	**DOS** 文件名	说明
hp_kanji_c	hp_kan_c.fnt	
hp_symbols	hp_syms.fnt	用于标注的特殊字符：度 (°)、直径符号、加-减 (±)、分 (')、秒 (")
hp_symbols2		特殊符号字符 (公差)
hp_Y14.5	hp_y14_5.fnt	仅用于符号
hp_i3098_c	i3098_c.fnt	ISO 3098 字体，恒定字符间距
hp_i3098_v	i3098_v.fnt	ISO 3098 字体，可变字符间距

示例 - 以某种字体显示字符

通过以下方式可在屏幕上以当前字体显示字符及其对应的 ASCII 编号：

- 使用文本 **2** 菜单中的“字体编辑器”。(有关详细信息，请参阅“使用 Creo Elements/Direct Drafting 设计和绘制”。)
- 运行下列宏。

(有关 ASCII 编号的说明，请参阅有关自定义键盘的部分。)

注意

宏使用窗口功能来设置当前端口的坐标。如果当前端口不是您希望显示所在的端口，请使用“当前”并选择所需视区。

该宏会删除所有现有 2D 几何。

```
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 18,28 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
```

该宏以当前字体显示所有字符，可使用“文本”、“符号”、和“尺寸”菜单中的键盘输入在绘图上显示当前字体。

可以对应于任意 ASCII 编号 (0 到 255) 的字体来定义字符，当系统提示您将文本包括在绘图中时，通过宏或在不带引号的输入行上输入 (`CHR n`)，可以将所有字符显示在绘图上，其中 n 是 ASCII 编号。

但是，您通常将只会对那些当系统提示从文本、符号和尺寸菜单中输入时通过输入文本即可显示的字符感兴趣。为此，该宏只会列出那些使用普通键盘字符或自定义功能键即可输入的字符。已从列表中省略了与控制字符对应的 ASCII 编号。它们是编号 0-31、127-159 和 255。

该宏以定义局部变量然后定义 ASCII 编号 0 到 9 生成的字符 0 到 9 开头。这些编号通常不会用于生成字符，但已在宏中使用它们，目的是为了避免与当前字体显示的字符产生冲突。字符 0 到 9 用于在屏幕中生成 ASCII 编号的列表。

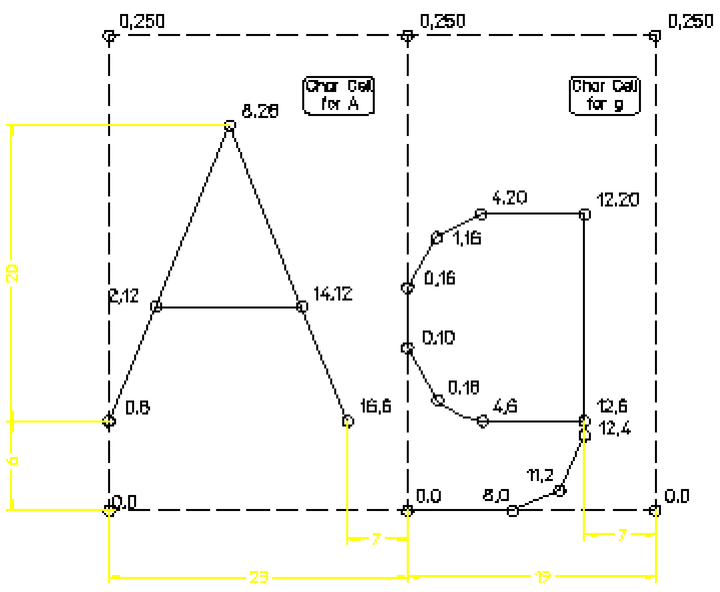
将文本参数设置为其默认值并删除所有 2D 几何。在当前视区中定义一个窗口，然后显示字符及其对应的 ASCII 编号。如果在运行宏时采用 hp_i3098_v (默认文本字体) 作为当前字体，则按以下规则显示：

```
032   SPACE

033     !

034     "

035     #

036     $
```

如何创建文本字体

字体中的每个字符都有一个属于自己的单元格。单元格大小 (以栅格点为单位) 最大可达 250 × 250。确定要在单元格中使用多少个点取决于所需的详细程度：详细程度越高，需要的点就越多。下图中显示的示例介绍了如何在文本字体中定义字母 A 和 g。相同的原则适用于符号的定义。

图 **27.** 创建字体



字符根据所需的详细程度排列在栅格的单元格中。栅格点之间的垂直距离由 TEXT_SIZE 函数确定，栅格点的数量与指定大小相符。栅格点之间的水平距离也按照相同的方式确定，除此之外，还可以使用 TEXT_RATIO 函数对其进行更高，该函数会影响栅格点的宽高比 (通常设置为 1)。有关这些功能的详细信息，请参阅 help 文件。

用于创建文本字体的命令为 DEFINE_FONT。以下是以上字体的定义方式：

```
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
```

字体的名称为 Example。(6) 后面的第一个数字称之为下长度。这表示将字符或符号放置在屏幕上时位于光标下方的栅格点的数量。在文本应用中，可将其用作 g 或 p 等字符尾部所需的最大距离。

第二个数字 (20) 称之为高度。栅格点之间的垂直距离值通过将 TEXT_SIZE 函数除以高度值来设置。在文本应用中，高度值可以表示与大写字母的高度对应的栅格点的数量。

在以上示例中，如果将 TEXT_SIZE 保留为默认值 3.5 mm，则字母 A 的高度将为 3.5 mm，因为 20 个栅格点表示字母的高度，并且 DEFINE_FONT 函数中的高度值也同样为 20。

此行 (20) 中的第三个数字称之为宽度。这提供了位于水平方向上的栅格点的数量，该数量与使用 TEXT_SIZE 函数设置的值相等。在本例中，将宽度值设置为与高度值相等，以使字符比例与栅格上定义的比例保持相同。如果将宽度规格减少为 10，将在水平方向将字符长度延长为其宽度的两倍。

CHAR_LAYOUT 命令开始字符定义。使用一组栅格点坐标依次定义各个字符。将以点到点的方式绘制线，以形成字符轮廓。BREAK 函数用于在当前绘制的线中放置一个断点。

位于每个字符定义末端的数字即是字符单元格的宽度 (以栅格点为单位)。该数字用于控制字符间距。

命令 END 用于结束字体定义。

从键盘输入指令可以定义新的文本字体，但更简便的方法如下：在编辑器中创建包含所有所需字符定义的字体，然后将其存储为文件，以便可以根据需要输入。通过"输入"的键盘或文件定义新字体后，该新字体将成为当前字体，并且将只能使用包含在新的当前字体中的字符。要使用任何以系统字体或自定义字体显示的字符，都必须重新指定当前字体。

有多种字体可在系统中使用，并且可以通过屏幕菜单文本 **1** 中的设置字体将其设置为当前字体。如果已自定义了一些字体，则可以通过输入以下命令选择要将哪种字体用作当前字体：

```
CURRENT_FONT 'fontname'
```

其中 fontname 是字体的名称。

存储字体 - ASCII 文件和二进制文件

要使用一种系统字体来创建包含所有说明的文本文件，必须先查看是否已将该字体加载到系统中。从"文本"屏幕菜单中选择"列出字体"，以查看该字体是否在列表中，然后按 [ESC] 返回菜单。

如果该字体未在列表中，则需要通过输入以下内容进行加载：

```
LOAD_FONT 'bin_filename'
```

其中 bin_filename 是包含该字体的二进制文件的名称。

通过输入以下内容可将该字体以 ASCII 文件的形式写入到磁盘中：

```
SAVE_FONT 'fontname' 'ascii_filename'
```

其中 fontname 是该字体的名称，ascii_filename 是为包含该字体的文本文件选取的名称。然后，可以常规方式检查和编辑该文件。

一个文本文件中可以包含多种字体。输入某个文本文件时，将加载其中包含的所有字体，同时列表中的最后一种字体将成为当前字体。

要创建包含一种自定义字体的二进制文件，请按以下步骤执行：

1. 使用上述方法创建文本文件。
2. “输入”该文件，以便将其中包含的所有字体加载到系统中。
3. 通过输入以下内容将磁盘上的各个字体存储在单独的二进制文件中：

   ```
   STORE_FONT 'fontname' 'bin_filename'
   ```

   其中 fontname 是字体的名称，bin_filename 是选择要将磁盘上的二进制文件存储在其下的文件的名称。

然后，可以按照本部分中之前描述的步骤使用 LOAD_FONT 将字体重新加载到系统。

无法在此编辑器中检查二进制文件。使用它们的好处在于提高了访问速度。

自定义启动过程

在实施此处描述的任何步骤之前，请阅读与操作系统对应的配置手册中介绍 Creo Elements/Direct Drafting 启动文件的部分。

如果已从 \me10\startup 文件末端的以下行中移除大括号 { }，则仅在启动过程中输入 custom.m 文件：

```
{ INPUT 'custom.m' }
```

完成此过程后，系统将根据 SEARCH 路径查找名为 custom.m 的文件。

这通常意味着，系统将首先在当前目录中查找，然后再在 \me10 目录中查找。在整个讨论过程中，假设 SEARCH 路径是按照这种方式定义的。启动过程结束后，可通过从“设置”屏幕菜单中选择“编辑环境”来查找 SEARCH 路径。

在启动过程中，当前目录是已登录用户的主目录。可以为自定义启动过程选择以下选项之一：

平台相关性

- 在通常从中启动 Creo Elements/Direct Drafting 的目录中创建一个名为 custom.m 的文件，然后对该文件进行编辑，以使其包含自己的自定义要求。这将允许控制在启动过程中实施的自定义步骤。
- 使用组自定义文件。为此，Creo Elements/Direct Drafting 启动目录中不得包含名为 custom.m 的文件。在本例中，系统查找的是本应由系统管理员创建的 \me10\custom.m 文件。无论启动目录为何，此文件中指定的自定义设置都将影响所有用户。

整个组的自定义启动

文件 \me10\custom.m 通常可以由系统管理员创建。可以编辑此文件，以使其包含自定义启动过程，该过程适用于尚未在 Creo Elements/Direct Drafting 启动目录中创建 custom.m 文件的任何组员。

可以使用任何文本编辑器对 \me10\custom.m 文件进行编辑，但系统管理员可能更希望在以普通用户身份运行 Creo Elements/Direct Drafting 软件时开发和测试自定义过程并将文件复制或附加到 \me10\custom.m 中。

组和各用户的自定义启动

我们已经了解了用户如何通过在他们的 custom.m Drafting 启动目录中创建名为 Creo Elements/Direct 的文件来实施自己的启动过程。以下选项可供已创建此文件的用户使用。

- 将以下行作为 custom.m 文件的第一行添加：

  ```
  INPUT '\me10\custom.m'
  ```

  在本例中，将首先实施由系统管理员为整个组开发的自定义过程，然后将实施您自己的过程。

- 如果希望根据自身需求来自定义系统启动而不使用任何组过程，请不要将以上显示的行包括在自定义文件中。
- 如果希望系统根据不含任何自定义设置的常规默认设置进行操作，请在主目录中创建一个 custom.m 文件并将其留空。

为自定义启动创建一个菜单

如果已在主/启动目录中创建了一个名为 custom.m 的文件，则可以将希望在启动过程中实施的任何函数或命令放置在该文件中。此外，还可以使用该文件定义功能键，以便可以根据需要输入其他文件。

以下示例介绍了为在启动后显示某个菜单而应如何对系统进行自定义设置，并提供了可能会在不同环境中使用的各种自定义过程。编辑 custom.m 文件以使其包含下列行：

```
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}
```

在本例中，名称 project1 到 project7 是包含不同项目相关工作的自定义过程的文件的名称。可以根据自身喜好选择文件名。

已经设置了功能键，以便不仅可以在字母屏幕上显示菜单时使用它们，还可以在显示图形时使用它们。以下介绍了功能键的定义方法：

- (CHR 4) 表示 [CTRL] [D]。这将退出字母显示，以便可以执行输入语句。如果当前显示的是图形且系统正在等待命令，[CTRL] [D] 将不执行任何操作。
- (CHR 13) 表示 RETURN。从而无需在按功能键之后按 [RETURN]。
- 前七个功能键用于输入所需文件，而功能键 [f8] 用于重新显示菜单。名称 menu 仅仅是一个示例。可以为包含屏幕菜单的文件选择自己的名称。

位于自定义文件末端的 EDIT_FILE 语句将显示屏幕菜单。这应是最后一行，因为此时将为用户提供是否通过按功能键来输入其他文件的选择。

以下示例显示了菜单文件可能具有的外观：

```
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
```

按某个功能键时，菜单将消失，并将显示图形。已在功能键中定义的文件将为 INPUT，因而可以实施自定义过程。包含自定义过程的文件应将以下行包含为最后一条语句。

```
DISPLAY 'Press function key f8 to display customizing menu'
```

可以重新显示菜单或继续执行其他操作。

注意

显示自定义菜单时，请不要在屏幕上键入任何内容。功能键定义中的 (CHR 4) 将保存菜单文件的副本并覆盖旧的菜单文件。

自定义剖面线图案

defaults.m 文件包含用于“剖面线”屏幕菜单上的“铁”、“钢”和“铜”选项的剖面线图案。有关 defaults.m 文件的详细信息，请参阅与操作系统对应的配置手册。

剖面线图案使用下列宏生成。

```
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
```

可以轻松更改这些宏或写入自己的新宏来生成其他剖面线图案。

系统管理员可以使用任何文本编辑器来编辑 \me10\defaults.m 文件。但是，在重新安装或更新 Creo Elements/Direct Drafting 软件时，将覆盖对 \me10\defaults.m 文件所作的任何更改。以下是一种较好的方法：

1. 启动 Creo Elements/Direct Drafting。
2. 使用文件屏幕菜单中的 COPY 命令将 \me10\defaults.m 文件复制到当前目录。
3. 使用 Creo Elements/Direct Drafting 编辑器对文件进行编辑。删除不需要更改的所有行，然后更改剩余行以匹配当前环境。
4. INPUT 文件，以便可以验证您的更改。
5. 如果结果令人满意，请将以下行包括在 \me10\custom.m 文件中：

   ```
   INPUT 'filename'
   ```

   其中 filename 是文件的完整路径名称和文件名。

   如有必要，可编辑 \me10\startup 以从以下行中移除大括号 { }：

   ```
   {INPUT 'custom.m'}
   ```

   以便读取为：

   ```
   INPUT 'custom.m'
   ```

启动 Creo Elements/Direct Drafting 时，startup 文件将输入 custom.m 文件的内容。这会依次输入您的文件。

有关剖面线的详细信息，请参阅“使用 Creo Elements/Direct Drafting 设计和绘制”。

键盘输入字符

CHAR 32 = SPACE
CHAR 33 = !
CHAR 34 = "
CHAR 35 = #
CHAR 36 = $
CHAR 37 = %
CHAR 38 = &
CHAR 39 = '
CHAR 40 = (
CHAR 41 =)
CHAR 42 = *
CHAR 43 = +
CHAR 44 = ,
CHAR 45 = -
CHAR 46 = .
CHAR 47 = /
CHAR 48 = 0
CHAR 49 = 1
CHAR 50 = 2
CHAR 51 = 3
CHAR 52 = 4
CHAR 53 = 5
CHAR 54 = 6
CHAR 55 = 7
CHAR 56 = 8
CHAR 57 = 9
CHAR 58 = :
CHAR 59 = ;
CHAR 60 = <
CHAR 61 = =
CHAR 62 = >
CHAR 63 = ?
CHAR 64 = @
CHAR 65 = A
CHAR 66 = B
CHAR 67 = C
CHAR 68 = D
CHAR 69 = E
CHAR 70 = F
CHAR 71 = G
CHAR 72 = H
CHAR 73 = I
CHAR 74 = J
CHAR 75 = K
CHAR 76 = L
CHAR 77 = M
CHAR 78 = N
CHAR 79 = O
CHAR 80 = P
CHAR 81 = Q
CHAR 82 = R
CHAR 83 = S
CHAR 84 = T
CHAR 85 = U
CHAR 86 = V
CHAR 87 = W
CHAR 88 = X
CHAR 89 = Y
CHAR 90 = Z
CHAR 91 = [
CHAR 92 = \
CHAR 93 =]
CHAR 94 = ^
CHAR 95 = _
CHAR 96 = `
CHAR 97 = a
CHAR 98 = b
CHAR 99 = c
CHAR 100 = d
CHAR 101 = e
CHAR 102 = f
CHAR 103 = g
CHAR 104 = h
CHAR 105 = i
CHAR 106 = j
CHAR 107 = k
CHAR 108 = l
CHAR 109 = m
CHAR 110 = n
CHAR 111 = o
CHAR 112 = p
CHAR 113 = q
CHAR 114 = r
CHAR 115 = s
CHAR 116 = t
CHAR 117 = u
CHAR 118 = v
CHAR 119 = w
CHAR 120 = x
CHAR 121 = y
CHAR 122 = z
CHAR 123 = {
CHAR 124 = |
CHAR 125 = }
CHAR 126 = ~

键盘输入字符

CHAR 160 = SPACE
CHAR 161 = À
CHAR 162 = Â
CHAR 163 = È
CHAR 164 = Ê
CHAR 165 = Ë
CHAR 166 = Î
CHAR 167 = Ï
CHAR 168 = ´
CHAR 169 = `
CHAR 170 = ˆ
CHAR 171 = ¨
CHAR 172 = ˜
CHAR 173 = Ù
CHAR 174 = Û
CHAR 175 = £
CHAR 176 = ¯
CHAR 177 = Ý
CHAR 178 = ý
CHAR 179 = °
CHAR 180 = Ç
CHAR 181 = ç
CHAR 182 = Ñ
CHAR 183 = ñ
CHAR 184 = ¡
CHAR 185 = ¿
CHAR 186 = ¤
CHAR 187 = £
CHAR 188 = ¥
CHAR 189 = §
CHAR 190 = ƒ
CHAR 191 = ¢
CHAR 192 = â
CHAR 193 = ê
CHAR 194 = ô
CHAR 195 = û
CHAR 196 = á
CHAR 197 = é
CHAR 198 = ó
CHAR 199 = ú
CHAR 200 = à
CHAR 201 = è
CHAR 202 = ò
CHAR 203 = ù
CHAR 204 = ä
CHAR 205 = ë
CHAR 206 = ö
CHAR 207 = ü
CHAR 208 = Å
CHAR 209 = î
CHAR 210 = Ø
CHAR 211 = Æ
CHAR 212 = å
CHAR 213 = í
CHAR 214 = ø
CHAR 215 = æ
CHAR 216 = Ä
CHAR 217 = ì
CHAR 218 = Ö
CHAR 219 = Ü
CHAR 220 = É
CHAR 221 = ï
CHAR 222 = ß
CHAR 223 = Ô
CHAR 224 = Á
CHAR 225 = Ã
CHAR 226 = ã
CHAR 227 = Ð
CHAR 228 = đ
CHAR 229 = Í
CHAR 230 = Ì
CHAR 231 = Ó
CHAR 232 = Ò
CHAR 233 = Õ
CHAR 234 = õ
CHAR 235 = Š
CHAR 236 = š
CHAR 237 = Ú
CHAR 238 = Ÿ
CHAR 239 = ÿ
CHAR 240 = Þ
CHAR 241 = þ
CHAR 242 = °
CHAR 243 = ø
CHAR 244 = ¶
CHAR 245 = ¾
CHAR 246 = -
CHAR 247 = ¼
CHAR 248 = ½
CHAR 249 = ª
CHAR 250 = º
CHAR 251 = «
CHAR 252 = □
CHAR 253 = »
CHAR 254 = ±
CHAR 255 = ⊠

13

命令和函数的简短描述

本章非常简要地介绍了各个命令和函数，并按字母数字顺序对其进行排序。有关任何命令或函数的详细信息，请参阅 Creo Elements/Direct Drafting 帮助系统。例如，输入：

```
help catch
```

将会清屏，同时 Creo Elements/Direct Drafting 将显示有关 CATCH 函数的详细信息。

`ABS` (算术函数)

对于数字自变量：返回自变量的绝对值。对于矢量自变量：返回从原点到自变量终点的矢量长度。

`ADD_CURRENT_INFO` (命令)

将指定文本添加到当前信息。

`ADD_DIM_POSTFIX` (命令)

向现有尺寸添加后缀。

`ADD_DIM_PREFIX` (命令)

向现有尺寸添加前缀。

`ADD_DIM_SUBFIX` (命令)

向现有尺寸添加下标。

`ADD_DIM_SUPERFIX` (命令)

向现有尺寸添加上标。

`ADD_DIM_TOLERANCE` (命令)

用于向选定尺寸添加公差。

ADD_ELEM_INFO (命令)

将指定文件添加到指定元素的信息中。

ADU_ACCURACY (函数)

指定比较两种布局时使用的精度。

ADU_CHECK (命令)

比较两种布局并尝试重新注释新布局。

ADU_CONFIRM_ANNOS (命令)

注释自动更新后，系统将每个注释标记为"已传递"(默认为绿色)，"已重新生成"(默认为红色) 或"已更新"(默认为蓝色)。利用 ADU_CONFIRM_ANNOS 可以接受系统提议。

ADU_UPDATE_ANNOS (命令)

用于交换注释所涉及的元素。例如，可以交换尺寸的参考元素但不删除尺寸。

ANALYZE_BSPLINE (函数)

用于分析说明选定 b 样条的特定元素和值。

AND (算术函数)

如果两个自变量均不是 0，则返回 1。否则返回零。

ANG (算术函数)

返回 X 轴与从原点到自变量终点的矢量之间的角度。

ARC (命令)

创建弧。

ARCCOS (算术函数)

返回其余弦等于自变量的角度的主值。

ARCSIN (算术函数)

返回其正弦等于自变量的角度的主值。

ARCTAN (算术函数)

返回其正切等于自变量的角度的主值。

ARC_RESOLUTION (函数)

指定屏幕上的弧精度 (如果使用显示列表)。

AREA_PROPERTY (命令)

计算选定区域的物理属性。

ARROW_CURSOR (函数)

控制在表中和菜单区域中看到的光标形状。

ARROW_FILL (函数)

控制尺寸和指引线中箭头的填充。

ASSIST (命令)

设置用户协助 (Copilot)。

AUTO_NEW_SCREEN (函数)

指定是否应在显示 Creo Elements/Direct Drafting 窗口之后运行 NEW_SCREEN。

AUTO_STORE_TIME (函数)

执行绘图的自动保存操作。频率由用户定义。

BEEP (函数)

使扩音器产生蜂鸣。

BLACK (函数)

指定线颜色。

BLUE (函数)

指定线颜色。

BSPLINE (命令)

创建 B 样条曲线。

BSPL_ADD_C_PNT (命令)

在两个相邻控制点之间添加一个控制点。

BSPL_ADD_I_PNT (命令)

在两个相邻控制点之间添加一个插值点。

BSPL_DEL_C_PNT (命令)

从样条删除控制点。

BSPL_DEL_I_PNT (命令)

从样条删除插值点。

BSPL_DEL_TANGENT (命令)

删除斜率。

BSPL_MOVE_C_PNT (命令)

可修改控制点的位置。

BSPL_MOVE_I_PNT (命令)

可修改插值点的位置。

BSPL_MOVE_PNT (命令)

可修改 B 样条的位置。

BSPL_POINT_LENGTH (命令)

用于将 b 样条分成几个长度相同的部分。

BSPL_POLYGON_FEEDBACK (函数)

在创建 b 样条时用于确定反馈类型。

CANCEL (命令)

取消当前的系统活动。返回至输入命令。

CANCEL_EDIT_DIM_TEXT (命令)

取消 EDIT_DIM_TEXT 的作用。将编辑后的尺寸恢复至其原始状态。

CATALOG (函数)

将已命名目录列表输出至指定目标。

CATALOG_LAYOUT (命令)

指定 CATALOG 给定的目录信息的布局。

CATCH (函数)

允许用户在屏幕上选择一个点，而不用直接将光标放在该点上。

CENTER (命令)

居中菜单文本。

CENTERLINE (命令)

用于创建圆形元素的中心线。中心线成为圆形元素的一部分，并且如果移动或删除了关联的圆形元素，则也会将其删除。

CENTER_DASH_DASH (函数)

定义线类型。

CHAMFER (命令)

创建倒角。

CHANGE_COLOR (命令)

更改选定元素的颜色。

CHANGE_CURRENT_INFO (函数)

在当前信息文本中全局搜索和替换。

CHANGE_DIM_ARROW (命令)

更改当前尺寸线结尾。

CHANGE_DIM_COLOR (命令)

用于选定尺寸，更改延伸线和尺寸线的颜色。

CHANGE_DIM_FORMAT (命令)

用于更改现有尺寸的格式。

CHANGE_DIM_FRAME (命令)

更改尺寸文本选定框架的类型。

CHANGE_DIM_LINEWIDTH (命令)

为选定的尺寸更改延长线和尺寸线的笔尺寸。(等同于 CHANGE_DIM_PENSIZE 操作。)

CHANGE_DIM_PENSIZE (命令)

为选定的尺寸更改延长线和尺寸线的笔尺寸。

CHANGE_DIM_TEXTS (命令)

用于更改现有尺寸文本的属性。

CHANGE_DIM_TEXT_COLOR (命令)

定义尺寸文本颜色。

CHANGE_DIM_TEXT_LOCATION (命令)

更改尺寸文本的位置：线上方、线上或者线下方。

CHANGE_DIM_TEXT_ORIENTATION (命令)

更改尺寸文本的方向。

CHANGE_DIM_VERTEX (命令)

将尺寸延伸线移至另一个顶点。

CHANGE_ELEM_INFO (命令)

在指定信息文本中全局搜索和替换。

CHANGE_FILLET (命令)

更改选定圆角的半径。

CHANGE_GLOBAL_INFO (函数)

在内存中每个元素的信息文本中全局搜索和替换。

CHANGE_HATCH_ANGLE (命令)

更改选定剖面线的角度。

CHANGE_HATCH_COLOR (命令)

更改选定剖面线的颜色。

CHANGE_HATCH_DIST (命令)

更改剖面线之间的间隔。

CHANGE_HATCH_LINETYPE (命令)

更改选定剖面线的线类型。

CHANGE_HATCH_PATTERN (命令)

更改选定剖面线的图案。

CHANGE_HATCH_REF_PT (命令)

更改选定剖面线的参考点。

CHANGE_LEADER_ARROW (命令)

更改选定指引线的终止符。

CHANGE_LEADER_ARROW_SIZE (命令)

更改选定指引线的终止符大小。

CHANGE_LINETYPE (命令)

更改选定元素的线类型。

CHANGE_LINEWIDTH (命令)

更改选定元素的笔尺寸，它可以是真实几何，但不能为构造几何或“点”。(等同于 CHANGE_DIM_PENSIZE 操作。)

CHANGE_LINESIZE (命令)

更改选定元素的线尺寸，它可以是真实几何，但不能为构造几何或“点”。

CHANGE_DIM_PENSIZE (命令)

为选定的尺寸更改延长线和尺寸线的笔尺寸。

CHANGE_PART_REF_PT (函数)

更改活动零件的参考点。

CHANGE_TABLE_SIZE (函数)

对不受大小更改保护的现有表的大小进行更改。

CHANGE_TEXT (命令)

将一个或多个现有文本更改为新文本。

CHANGE_TEXT_ADJUST (命令)

更改选定文本的调整参数。

CHANGE_TEXT_ANGLE (命令)

更改选定文本的角度。

CHANGE_TEXT_FILL (命令)

关闭或打开选定文本的填充。

CHANGE_TEXT_FONTNAME (命令)

更改选定文本的字体。

CHANGE_TEXT_FRAME (命令)

更改选定文本的框架。

CHANGE_TEXT_LINESPACE (命令)

更改多行文本的内部行间距。

CHANGE_TEXT_RATIO (命令)

用于选定文本，更改字符宽度和高度之间的比率。

CHANGE_TEXT_SIZE (命令)

更改选定文本的字符大小。

CHANGE_TEXT_SLANT (命令)

更改选定文本的倾斜。

CHANGE_VIEWPORT_COLOR (函数)

更改当前视区的背景颜色。

CHANGE_VIEWPORT_SIZE (函数)

更改当前视区的大小。

CHAR_LAYOUT (命令)

定义当前字体的字符。

CHECK_3D_GEO_MODIFY (函数)

指定修改 Creo Elements/Direct Modeling 布局 (ADU) 时是否发出警告。

CHECK_BREAK (算术函数)

如果按下 BREAK 键且 IGNORE_BREAK 活动，则返回 1。

CHECK_DIM_DETAIL (函数)

启用/禁用内置尺寸检查机制。

CHECK_ERROR (算术函数)

如果自上次调用 TRAP_ERROR 函数后捕捉到一个或多个错误，则返回 1，否则返回 0。

CHECK_FONT_FILLABLE (函数)

将当前字体的字符写入无法填充的选定输出规范。

CHECK_WINDOW (函数)

启用/禁用内置窗口检查机制，默认情况下此机制将拒绝极大或极小的窗口设置。

CHG_PIXEL_COLOR (命令)

将像素的颜色更改为 SET_COLOR 中指定的颜色。

CHR (算术函数)

将十进制数转换为等效的 ASCII 字符。例如，CHR(35) 即为 '#'。

CIRCLE (命令)

创建圆。

CL_ABS_OFFSET (命令)

设置中心线偏移。

CL_COLOR (命令)

设置中心线颜色。

CL_LINETYPE (命令)

设置中心线线类型。

CL_LINEWIDTH (命令)

设置中心线笔尺寸。(等同于 CL_PENSIZE 操作。)

CL_PENSIZE (命令)

设置中心线笔尺寸。

CL_REL_OFFSET (命令)

设置中心线偏移 (相对于半径尺寸)。

CLEAN_DRAWING (命令)

清除重复几何的绘图 (例如，重叠)。

CLIPBOARD_SIZE

设置 Windows 剪贴板大小 (出图区域)，以便后续向剪贴板出图。

CLOSE_FILE (函数)

关闭指定文件。

CMD_BG_COLOR (函数)

更改命令行的背景颜色。

CMD_TXT_COLOR (函数)

更改命令行文本的颜色。

COLOR (函数)

指定当前几何和文本颜色。

COLOR_LTAB (函数)

更改逻辑表标题和数据区域中指示位置的颜色。

CONFIGURE_EDITOR (函数)

配置内置屏幕编辑器。

CONNECT_TABLE (函数)

将显示表连接到逻辑表。

CONTOUR (命令)

修剪所有选定元素和其他选定元素以创建封闭轮廓。

CONTROLZ_IS_EOF (函数)

指定应将 Ctrl-Z 字符解释为文件结尾字符还是正常数据字节。

CONVERT_C_TO_B_SPLINE (命令)

将旧样条类型 (“C 样条”) 的选定样条转换为新样条类型 (“B 样条”) 的样条。

CONVERT_DIM_TOLERANCE (命令)

将现有公差更改为其他类型。

CONVERT_DIM_UNIT (命令)

更改线性尺寸和角度尺寸的当前单位。

CONVERT_SPLINE (命令)

将选定样条转换为一系列弧和线。

COPY_FILE (函数)

将指定源文件复制到目标。

COS (算术函数)

返回自变量的余弦。

CREATE_DETAIL (命令)

创建现有几何的详细视图并按指定因子放大。

CREATE_DIRECTORY (函数)

创建新目录。

CREATE_LTAB (函数)

创建新逻辑表。如果该表已经存在，将不会给定任何错误并忽略此命令。

CREATE_POLY (命令)

能够通过现有元素创建折线。然后可将折线视为单个元素，对其进行相应修改。

CREATE_SUBPART (命令)

使用现有元素在活动零件中创建新零件。

CREATE_VIEWPORT (函数)

创建新视区。

CS_AXIS (函数)

重新定义输入坐标系的轴，使原点保持固定。

CS_MIRROR (函数)

指定输入坐标系的轴关于其他轴反射。

CS_REF_PT (函数)

更改输入参考系统的原点。

CS_ROTATE (函数)

围绕原点旋转输入坐标系。

CS_SET (函数)

用于输入坐标系，更改原点和轴的角度。

CURRENT_DIM_TEXTS (函数)

更改所有尺寸文本的当前属性。

CURRENT_DIM_UNITS (函数)

设置线性尺寸和角度尺寸的当前单位。

CURRENT_DIRECTORY (函数)

将当前目录设置为指定目录。

CURRENT_FONT (函数)

选择新文本使用的字体。

CURRENT_HATCH_PATTERN (函数)

定义新创建的剖面线使用的当前剖面线图案。

CURRENT_MENU (函数)

指定当前屏幕菜单的名称。

CURRENT_SPOTLIGHT_ATTR (函数)

用于更改与 SPOTLIGHTON 一起使用的默认颜色和线类型。

CURRENT_VERTEX_COLOR (函数)

定义“显示顶点开”中使用的顶点的默认颜色

CURRENT_VIEWPORT (函数)

设置当前视区。

CURSOR (函数)

选择小光标或大光标。

CURSOR_COORDINATES (函数)

允许在输入区域上显示光标的当前坐标。

CUT_MIDDLE

剪切并删除元素与两个其他元素交点处的中间段。

CYAN (函数)

将默认颜色切换为青色。

C_CIRCLE (命令)

创建构造圆。

C_COLOR (函数)

设置构造几何的当前颜色。

C_LINE (命令)

设置构造线。

C_LINETYPE (函数)

指定构造几何的当前线类型，即构造圆和构造线。

DASHED (函数)

指定线类型。

DASH_CENTER (函数)

指定线类型。

DATE (算术函数)

返回日期和时间。例如，"26-Oct-99 14:26:58"。

DA_DB_ADD (命令)

向尺寸数据库添加新零件。

DA_DB_DELETE (命令)

从尺寸数据库中删除零件/条目。

DA_DB_EXPORT (命令)

从尺寸数据库中提取条目并将其复制到当前绘图中。新条目将成为 Top 零件的子零件。

DA_DB_FILL_TABLE (命令)

清除并重新填充尺寸数据库表。不应在命令行中使用。

DA_DB_INQ (函数)

返回关于标注数据库的信息。此函数支持“标注加速模块”屏幕界面。通常不应在命令行中使用它。

DA_DB_LOAD (命令)

将标注数据库文件加载至内存中。

DA_DB_MATCH (命令)

将尺寸数据库条目中的标注传递至绘图元素。

DA_DB_STORE (命令)

将尺寸数据库以 MI 格式存储至文件中。

DA_DB_UNLOAD (命令)

从系统内存中卸载当前标注数据库。

DA_DB_WIN_CREATE (函数)

创建一个窗口用于显示标注数据库。不应在命令行中使用。

DA_DB_WIN_LOC (函数)

设置尺寸数据库窗口的位置。不应在命令行中使用。

DA_DIM_ANGLE (命令)

创建一个或多个角度尺寸。

DA_DIM_ARC (命令)

创建一个或多个弧尺寸。

DA_DIM_AUTO_LOC (函数)

设置自动定位标注的放置参数。

DA_DIM_AUTO_STRATEGY (函数)

设置交点检查等级，该检查在自动定位尺寸时执行。

DA_DIM_CHAIN (命令)

创建一个或多个链尺寸。

DA_DIM_CHAMFER (命令)

创建一个或多个倒角尺寸。

DA_DIM_COORD (命令)

创建一个或多个坐标标注实例。

DA_DIM_DATUM_LONG (命令)

用长基线创建一个或多个基准标注实例。

DA_DIM_DATUM_LONG_SYM (命令)

用长基线创建一个或多个对称标注实例。

DA_DIM_DATUM_SHORT (命令)

用短基线创建一个或多个基准标注实例。

DA_DIM_DELETE (命令)

从基准尺寸堆栈中删除一段或多段。也会删除整个尺寸。

DA_DIM_DIAMETER (命令)

创建一个或者多个直径尺寸。

DA_DIM_GEO_SENSE (命令)

根据选定元素的几何类型创建其标注。

DA_DIM_HOLE_INSERTION (函数)

如果新尺寸位于剖面线区域中，则用于控制行为。

DA_DIM_INCLINE (命令)

使具有水平或竖直属性的尺寸倾斜。

DA_DIM_INSERT (命令)

向基准尺寸堆栈中插入一个或多个新段。

DA_DIM_LINE (命令)

创建一个或多个单一线尺寸。

DA_DIM_LINE_SYM (命令)

创建一个或多个对称单一线尺寸。

DA_DIM_PD_SCAN (命令)

根据选定元素的 Parametric Design 约束创建其尺寸。

DA_DIM_RADIUS (命令)

创建一个或多个半径尺寸。

DA_DIM_SHORT_SPACE (函数)

设置基准尺寸堆栈中各项之间的间距。

DA_FILTER_ACTIVATE (函数)

启用标注分配的选择过滤。

DA_FILTER_ADD (函数)

向当前标注选择过滤器添加新选择条件。最好通过"标注加速模块"屏幕界面调用此函数。

DA_FILTER_CLEAR_GEOTYPES (函数)

从当前标注选择过滤器中清除所有几何类型条件。最好通过"标注加速模块"屏幕界面调用此函数。

DA_FILTER_CLEAR_LINETYPES (函数)

从当前标注选择过滤器中清除所有线类型条件。最好通过"标注加速模块"屏幕界面调用此函数。

DA_FILTER_DEL_COLOR (函数)

从当前标注选择过滤器中清除所有颜色条件。

DA_FILTER_DEL_ORIENT (函数)

从当前标注选择过滤器中清除所有线方向条件。

DA_FILTER_DEL_WIDTH (函数)

从当前标注选择过滤器中清除所有宽度条件。

DA_FILTER_DEL_LINESIZE (函数)

从当前标注选择过滤器中清除所有线尺寸条件。

DA_FILTER_DEL_PENSIZE (函数)

从当前标注选择过滤器中清除所有笔尺寸条件。(等同于 DA_FILTER_DEL_WIDTH 操作。)

DA_FILTER_INQ (函数)

返回关于标注选择过滤器的信息。此函数支持“标注加速模块”屏幕界面。通常不应在命令行中使用它。

DA_FILTER_REFRESH_LINEWIDTH (函数)

更新笔尺寸选择过滤器表的内容。不应在命令行中使用。(等同于 DA_FILTER_REFRESH_PENSIZE 操作。)

DA_FILTER_REFRESH_LINEWIDTH (函数)

更新笔尺寸选择过滤器表的内容。不应在命令行中使用。(等同于 DA_FILTER_REFRESH_PENSIZE 操作。)

DA_FILTER_REFRESH_LINESIZE (函数)

更新线尺寸选择过滤器表的内容。不应在命令行中使用。

DA_FILTER_REFRESH_PENSIZE (函数)

更新笔尺寸选择过滤器表的内容。不应在命令行中使用。

DA_FILTER_REFRESH_ORIENT (函数)

更新方向选择过滤器表的内容。不应在命令行中使用。

DA_FILTER_SET_NAME (函数)

设置当前标注选择过滤器的标识名称。

DA_FILTER_STORE (函数)

将当前标注选择过滤器保存至文件中。

DA_LINESIZE (函数)

设置尺寸的线尺寸。

DA_PENSIZE (函数)

设置尺寸的笔尺寸。

DA_MOVE_DIMENSION (命令)

移动一个或多个尺寸。

DA_NULL (函数)

“标注加速模块”屏幕界面的支持函数。不执行任何操作。

DA_STYLE_APPLY (函数)

将当前样式参数应用到选定尺寸。

DA_STYLE_DEFER_UPDATE (函数)

防止样式显示视区更新。

DA_STYLE_ENABLE_UPDATE (函数)

启用样式显示视区更新。

DA_STYLE_GET (函数)

使选定尺寸的标注样式参数成为当前标注样式。

DA_STYLE_INQ (函数)

返回关于当前标注样式的信息。此函数支持“标注加速模块”屏幕界面。通常不应在命令行中使用它。不应在命令行中使用。

DA_STYLE_TYPE (函数)

选择新标注样式。

DA_STYLE_UPDATE (函数)

更新样式显示视区。

DA_STYLE_WIN_CREATE (函数)

创建一个窗口用于显示当前尺寸样式。

DA_STYLE_WIN_LOC (函数)

设置样式显示视区的位置。“标注加速模块”屏幕界面的支持函数。不应在命令行中使用。

DA_STYLE_WIN_RAISE (函数)

提升样式显示视区。

DA_WRITE_DIM_SETTINGS_MACRO (函数)

创建包含所有当前标注样式设置的宏文件。

DDE_ADD_TOPIC (函数)

向已识别的 DDE 主题列表添加新主题字符串。

DDE_CLOSE (算术函数)

关闭已命名的 DDE 对话。

DDE_ENABLE (函数)

启用 Creo Elements/Direct Drafting 作为 DDE 服务器并作用于 DDE 客户端的请求。

DDE_EXECUTE (算术函数)

将命令字符串发送至给定对话句柄指示的应用程序。

DDE_INITIATE (算术函数)

启动与给定应用程序关于给定主题的 DDE 对话。返回一个对话句柄供后续 DDE 命令使用。

DDE_REMOVE_TOPIC (函数)

从已识别的 DDE 主题列表中移除指定的主题字符串。

DDE_REQUEST (算术函数)

询问远程 DDE 应用程序获得指示数据项的字符串值。

DDE_SEND_ACK (算术函数)

发送给定对话句柄的 DDE 确认消息。

DDE_WITHHOLD_ACK (算术函数)

防止在执行操作文本之后发送 DDE 确认消息。而是返回活动的 DDE 对话句柄。

DEFINE (函数)

定义具有指定名称的宏。

DEFINE_CATALOG (函数)

定义 CATALOG 函数将要打印的信息。

DEFINE_FONT (函数)

定义新字体。

DEFINE_KEY (函数)

定义指定功能键的功能。这种键称为“热”键、“智能”键或“软”键。

DEFINE_MOUSE_KEY (函数)

定义鼠标键 1、2 和 3。

DELETE (命令)

删除选定元素。

DELETE_CURRENT_INFO (函数)

删除当前信息。

DELETE_DIMENSION (命令)

删除尺寸。

DELETE_DIM_POSTFIX (命令)

用于删除已标识的后缀。

DELETE_DIM_PREFIX (命令)

用于删除已标识的尺寸前缀。

DELETE_DIM_SUBFIX (命令)

删除尺寸下标。

DELETE_DIM_SUPERFIX (命令)

删除尺寸上标。

DELETE_DIM_TOLERANCE (命令)

用于删除选定尺寸的公差。

DELETE_ELEM_INFO (命令)

删除选定元素的所有信息。

DELETE_FONT (函数)

删除指定字体的定义。

DELETE_HATCH (命令)

删除选定剖面线。

DELETE_LABEL (命令)

从活动零件中删除 LABEL 命令生成的所有标签信息。

DELETE_LTAB_ROW (函数)

从命名的用户表中删除指示行。

DELETE_MACRO (函数)

删除指定宏或所有宏。

DELETE_MENU (函数)

删除屏幕菜单定义。

DELETE_TABLE (命令)

删除现有不受保护的表。

DELETE_VIEWPORT (函数)

删除指定视区。

DIM_ANGLE (命令)

创建角度尺寸。

DIM_ARC (命令)

创建弧尺寸。

DIM_ARROW (函数)

指定尺寸箭头的类型。

DIM_BREAK_RESTORE (函数)

修改后控制尺寸线和延长线。

DIM_BROKEN (函数)

当尺寸文本在尺寸延伸线内不适合时，指定如何绘制尺寸线。

DIM_CATCH_LINES (函数)

控制可以拾取的尺寸元素。

DIM_CATCH_RANGE (函数)

将尺寸文本的捕捉范围设置到延伸线的中间。

DIM_CHAIN (命令)

创建链标注。

DIM_CHAMFER (函数)

用于创建 JIS 倒角尺寸。

DIM_COLOR (函数)

设置延伸线和尺寸线的颜色。

DIM_CONVERT_UNIT (命令)

指定尺寸单位。

DIM_COORD (命令)

创建坐标标注。

DIM_CURSOR_POSITION (函数)

指定放置尺寸时光标的位置。

DIM_DATUM (命令)

指定尺寸位置。

DIM_DATUM_LONG (命令)

用长基线创建基准标注。

DIM_DATUM_SHORT (命令)

用短基线创建基准标注。

DIM_DATUM_STEP (函数)

指定基准尺寸尺寸线之间的间距。

DIM_DEC_PLACE (函数)

指定尺寸的小数位数。

DIM_DEG_MIN_SEC (函数)

设置尺寸的度分秒。

DIM_DIAMETER (命令)

创建直径尺寸。

DIM_DIAMETER_LINE (函数)

设置直径线。

DIM_EXTENSION_LENGTH (函数)

设置延伸线箭头与端点之间的当前长度。

DIM_FONT (函数)

指定尺寸字体。

DIM_FORMAT (函数)

用于指定线尺寸和角度尺寸的单位。

DIM_FRAME (函数)

指定尺寸文本是否应具有框架。

DIM_FT_INCH_SIGN (函数)

指定英尺/英寸选项。

DIM_LINE (命令)

创建单一线尺寸。

DIM_LINEWIDTH (函数)

设置延长线和尺寸线的笔尺寸。(等同于 DIM_PENSIZE 操作。)

DIM_PENSIZE (函数)

设置延长线和尺寸线的笔尺寸。

DIM_LINES_COLOR (函数)

指定尺寸线的颜色。

DIM_MIN_SPACE (函数)

设置几何与尺寸线之间的当前最小间距。

DIM_NUMBER_FORMAT (函数)

指定尺寸的编号格式。

DIM_OFFSET_LINE (函数)

指定尺寸延伸线偏移尺寸线的距离。

DIM_OFFSET_POINT (函数)

指定尺寸延伸线偏移几何的距离。

DIM_POSTFIX (函数)

能够向尺寸文本添加后缀字符串。

DIM_PREFIX (函数)

能够向尺寸文本添加后缀字符串。

DIM_RADIUS (命令)

创建半径尺寸。

DIM_RADIUS_LINE (函数)

将尺寸半径线设置为“开/关”。

DIM_SCALE (函数)

指定尺寸比例。

DIM_SELECT_BY_TEXTBOX (命令)

指定使用框选择时如何选择尺寸。

DIM_STAGGER_RESTORE (函数)

修改后控制尺寸线和延长线。

DIM_SUBFIX (函数)

用于向尺寸文本添加下标字符串。

DIM_SUPERFIX (函数)

用于向尺寸文本添加上标字符串。

DIM_TEXT_COLOR (函数)

指定尺寸文本的颜色。

DIM_TEXT_FRAME_COLOR_MODE (函数)

指定尺寸文本框架颜色与尺寸主文本颜色或尺寸线颜色的相关性。

DIM_TEXT_GAP (函数)

当破断尺寸线以插入尺寸时，设置尺寸文本和尺寸线之间的当前间隙。

DIM_TEXT_HOLE (命令)

在现有剖面线中为选定的尺寸文本创建孔。

DIM_TEXT_LOCATION (函数)

指定当前文本尺寸应在尺寸线上方、尺寸线上还是在尺寸线下方。对于竖直尺寸线而言，“下方”意味着更加接近几何。

DIM_TEXT_ORIENTATION (函数)

指定尺寸文本的当前方向。

DIM_TEXT_RATIO (函数)

指定尺寸文本比率。

DIM_TEXT_SIZE (函数)

指定尺寸文本大小。

DIM_TEXT_SPACE (函数)

将尺寸文本放置在尺寸线上方或下方时，指定文本与线之间的间距。

DIM_TOLERANCE (函数)

指定当前公差类型。

DIM_UNDERLINE_EDITED (命令)

在已编辑的尺寸上放置下划线。

DIM_UNITS (函数)

指定尺寸单位。

DIM_UPDATE (命令)

用于查找、标记和处理尺寸的组成部分，重新计算后其外观将发生更改。

DISPLAY (函数)

评估令牌和在命令行显示结果。然后，必须按下任意一键或者数字化任意点。

DISPLAY_LIST (函数)

启用/禁用选定视区显示列表的使用。

DISPLAY_NO_WAIT (函数)

评估令牌和在命令行显示结果。不需要用户操作。

DIV (算术函数)

整除，截断任意分数部分。

DOTTED (函数)

将默认线类型切换为点线。

DOT_CENTER (函数)

将默认线类型切换为点中心。

DOT_GRID (函数)

在指定视区中打开或关闭点栅格。

DRAWING_SCALE (命令)

相对于绘图仪纸张按指定因子缩放几何。

DRAW_CURR_PART_ON_TOP (函数)

控制当前零件的重绘 (顶级或 z 级)。

DUMP_SCREEN (函数)

将图形内容转储至指定文件。

DUMP_SCREEN_DEFAULTS (函数)

设置 DUMP_SCREEN 函数的参数。

DUMP_SCREEN_LANG (函数)

用于选择 DUMP_SCREEN 函数的打印机语言 (请参阅 DUMP_SCREEN)。

ECHO (函数)

打开或关闭 ECHO 文件。

EDIT_CURRENT_INFO (函数)

用于编辑当前信息，该信息会分配至每个新元素。

EDIT_DIM_POSTFIX (命令)

用于选定的尺寸文本，允许编辑后缀。

EDIT_DIM_PREFIX (命令)

用于选定的尺寸文本，允许编辑前缀。

EDIT_DIM_SUBFIX (命令)

用于编辑已标识尺寸文本的下标。

EDIT_DIM_SUPERFIX (命令)

用于编辑已标识尺寸文本的上标。

EDIT_DIM_TEXT (命令)

用于编辑选定的尺寸文本。

EDIT_DIM_TOLERANCE (命令)

用于编辑选定尺寸文本的公差。

EDIT_ELEM_INFO (命令)

用于编辑指定元素的信息。

EDIT_ENVIRONMENT (函数)

用于编辑建立当前环境的命令列表。环境由单位、尺寸默认值、剖面线图案等组成。

EDIT_FILE (函数)

允许通过内置文本编辑器编辑指定文件。

EDIT_MACRO (函数)

允许通过内置文本编辑器编辑指定宏。

EDIT_PART (命令)

使指定零件成为活动零件。

EDIT_PORT (函数)

指定内置屏幕编辑器使用的视区。

EDIT_TEXT (命令)

用于编辑指定文本。

ELLIPSE (宏)

绘制与椭圆近似的样条。

ELSE (伪命令)

条件运算符。

ELSE_IF (伪命令)

条件运算符。

ENABLE_BREAK (函数)

将中断处理重新重置为默认设置 (按下"中断"键会中断宏)。

END (命令)

终止当前命令或函数。

END_DEFINE (伪命令)

指示宏定义的结尾。

END_IF (伪命令)

指示"IF - 语句"的结尾。

END_LOOP (伪命令)

指示宏中的循环结尾。

END_PART (命令)

与 EDIT_PART PARENT 相同。

ENTER (函数)

评估令牌。结果显示在命令行中。

EQUIDISTANCE (命令)

创建等距轮廓。

ERROR_LOG (函数)

在内存中保存系统生成的警告和错误消息。

ERROR_STR (算术函数)

通过 TRAP_ERROR 启用错误捕捉后，显示系统发出的第一条错误消息。

EXIT (命令)

终止会话。控制返回至主机操作系统。必须通过 CONFIRM 确认，以免意外退出。

EXIT_IF (伪命令)

指示何时应终止循环的条件语句。

EXOR (算术函数)

异或。如果只有一个自变量 (共两个) 为 0，则返回 1。否则，返回 0。

EXP (算术函数)

返回 e (2.718...) 的自变量次幂。

FALSE (算术函数)

返回 0。

FBROWSER (函数)

文件浏览器功能。

FILLET (命令)

生成具有指定半径的圆角。

FOLLOW (函数)

使坐标系原点位于最近的输入点。

FONT_EDITOR (命令)

创建和修改 Creo Elements/Direct Drafting 字体。

FRACT (算术函数)

返回自变量的分数部分。

GATHER (命令)

将现有元素引入活动零件中。

GET_ELEM_INFO (函数)

将当前信息更改为与选定元素的信息相同。

GET_PID (算术函数)

显示运行程序的进程 ID。

GET_PROPERTIES (函数)

将当前属性更改为与选定元素的属性相同。

GET_TYPE (函数)

返回指定令牌的类型。此为 TYPE 命令的增强形式。

GREEN (函数)

将默认颜色切换为绿色。

GRID_FACTOR (函数)

指定栅格点或标尺刻度之间的距离。

HATCH (命令)

创建剖面线。

HATCH_ANGLE (函数)

设置当前剖面线角度。

HATCH_COLOR (函数)

设置当前剖面线颜色。

HATCH_DIST (函数)

设置当前剖面线距离。

HATCH_LINETYPE (函数)

设置当前剖面线线类型。

HATCH_REF_PT (函数)

设置当前剖面线参考点。

HELP (函数)

显示介绍所需关键字的 HELP 文件的部分。

HELP_PORT (函数)

定义供 HELP 函数使用的视区。

HIGHLIGHT_LTAB (函数)

更改已命名用户表数据区域内指示位置的突出显示状态。

HL_CHANGE_COLOR (宏)

更改由 HL_GENERATE_HIDDEN 命令计算的所有隐藏线的颜色。

HL_CHANGE_LTYPE (宏)

更改由 HL_GENERATE_HIDDEN 命令计算的所有隐藏线的线类型。

HL_DEFAULT_FACE_COLOR (函数)

指定默认面颜色。

HL_DELETE_FACE (命令)

删除覆盖面。这些选定面必须位于当前零件中。

HL_GENERATE_FACE (命令)

指定覆盖面。覆盖面可覆盖所有具有下限 z 值的几何。

HL_GENERATE_HIDDEN (命令)

促使转换为隐藏线生成模式。

HL_GEN_ALL_PART (命令)

指定整个零件的 z 值设置和面生成。

HL_INQ_CURR_Z_VALUE (函数)

默认情况下，返回分配给每个新创建元件的当前 z 值。

HL_INQ_FACE_COLOR (函数)

返回指定面的 rgb 颜色值。

HL_INQ_LOAD_OFFSET (函数)

返回 z 方向的加载偏移值。

HL_INQ_LOAD_VALUE (函数)

返回指定加载值/偏移的模式和值本身。

HL_INQ_RELATION_OFFSET (函数)

显示 z 方向的当前关系偏移。

HL_INQ_Z_VALUE (函数)

返回指定元素的 z 值，或整个装配的最小或最大 z 值。

HL_REDRAW_MODE (函数)

在 Creo Elements/Direct Drafting 正常重绘模式和隐藏线重绘模式间切换。

HL_SET_COLOR (函数)

设置隐藏线的颜色。

HL_SET_CURR_Z_VALUE (函数)

默认情况下，定义分配给每个新创建元件的 z 值。

HL_SET_FACE_COLOR (命令)

设置或更改覆盖面的背景颜色。

HL_SET_KEEP_COLOR (函数)

定义生成隐藏线过程中元素是否保持其颜色。

HL_SET_LINETYPE (函数)

设置隐藏线的线类型。

HL_SET_LOAD_VALUE (函数)

定义 z 方向的加载偏移。

HL_SET_RELATION_OFFSET (函数)

定义 z 方向用于计算作为与另一元素具有 ABOVE/BELOW/BETWEEN 关系而指定的 z 值的偏移。

HL_SET_Z_VALUE (命令)

分配或更改 z 值。

HL_SHOW_HIDDEN (宏)

使隐藏线可见或不可见。

HL_VISUALIZE (函数)

可以选择性查看特定 z 值的所有元素，或者以不同颜色显示特定级别的面。

HSL_COLOR (函数)

用于设置新颜色 (色调、饱和度和亮度)。

ICONIFY_WINDOW (函数)

图标化 Creo Elements/Direct Drafting 窗口。

IF (伪命令)

布尔表达式。

IGNORE_BREAK (函数)

通过宏启用时将使“中断”键无效。

INIT_PART (命令)

直接在活动零件下创建新的空零件。激活这个新零件。

INIT_SUBPART (命令)

直接在活动零件下创建新的空零件。激活这个新零件。

INPUT (函数)

将键盘或鼠标的输入流临时重新定向至指定文本文件。(使用宏中的 INPUT 命令时，必须与限定符 IMMEDIATE 一起使用，另请参阅第31页上的“INPUT”。)

INQ (算术函数)

返回系统阵列的元素。可对这些值进行设置

INQ_ELEM (函数)

将指定元素的相关信息写入系统阵列。可使用 INQ 检索此信息。

INQ_ENV (函数)

将系统环境的相关信息写入系统阵列。可使用 INQ 检索此信息。

INQ_PART (函数)

将零件的放大因子相关系统写入系统阵列。可使用 INQ 检索此信息。

INQ_SELECTED_ELEM (函数)

将已标识元素的相关信息写入系统查询阵列。这些信息可以在随后通过 INQ 检索 (请参阅 INQ)。

INQ_TABLE (函数)

用于获取指定表的相关信息。

INSERT_LTAB_ROW (函数)

向已命名用户表中插入行。

INT (算术函数)

返回自变量的整数部分。例如，INT(PI) 等于 3。

ISOMETRIC (命令)

用于创建等轴测视图。

KEEP_CORNER (函数)

设为“关”时，会删除拐角与圆角或倒角之间的线或弧。设为“开”时，拐角仍为拐角。

KNOB_BOX_FACTOR (函数)

指定旋钮的敏感度。

LABEL (命令)

生成活动零件中选定点、线、弧和圆的标签信息。

LAST_POSTFIX (函数)

允许使用上一个后缀作为当前后缀。

LAST_PREFIX (函数)

允许使用上一个尺寸前缀作为当前尺寸前缀。

LAST_SUBFIX (函数)

允许使用上一个下标作为当前下标。

LAST_SUPERFIX (函数)

允许使用上一个上标作为当前上标。

LAST_TOLERANCE (函数)

将上一个公差设置为当前公差。

LAST_WINDOW (函数)

恢复当前视区的上一个窗口。

LEADER_ARROW (函数)

指定新指引线使用的终止符。

LEADER_LINE (命令)

创建指引线。

LEN (算术函数)

对于字符串自变量，返回字符串的长度。对于矢量自变量，返回从原点到自变量终点的矢量长度。

LET (函数)

定义宏或变量。

LG (算术函数)

返回自变量的十进制对数 (底数为 10)。

LINE (命令)

创建线。

LINEPATTERN (函数)

指定要通过当前活动命令创建的元件的当前线类型。

LINESIZE (函数)

指定用于所有实际几何 (构造几何与“点”除外) 的当前线尺寸。

LINETYPE (函数)

指定当前线类型。

LINEWIDTH (函数)

指定用于所有新的真实几何 (构造几何与点除外) 的当前笔尺寸。(等同于 PENSIZE 操作。)

LINE_GRID (函数)

打开或关闭线栅格。

LIST_FONTS (函数)

给定所有定义的字体和使用的字体列表。以及给定当前字体的名称。

LIST_GLOBAL_INFO (函数)

列出内存中所有元素使用的信息。

LIST_MACRO_NAMES (函数)

将当前定义的所有宏的名称输出至指定目标。

LN (算术函数)

返回自变量的自然对数 (底数为 e)。

LOAD (命令)

将指定文件中的零件加载至内存中。

LOAD_FONT (命令)

将在指定文件中排序的所有文本字体加载至内存中。

LOAD_MACRO (函数)

将存储在指定文件中的所有宏加载至内存中。

LOAD_MODULE (命令)

激活指定应用程序模块。

LOCAL (伪命令)

定义宏中的局部变量。

LONG_DASHED (函数)

将默认线类型切换为长虚线。

LOOP (伪命令)

当“如果退出”子句中的布尔表达式评估为逻辑真时，定义重复的环。

LOWER_WINDOW (函数)

使 Creo Elements/Direct Drafting 窗口在 Windows® 桌面中位于所有其他窗口的下面。

LTAB_COLUMNS (算术函数)

显示已命名逻辑表中的列数。

LTAB_ROWS (算术函数)

显示已命名逻辑表中的行数。

LTAB_TITLES (算术函数)

显示已命名逻辑表中的标题字符串数。

LWC (算术函数)

小写。将大写字符转换为小写。

MAGENTA (函数)

将默认线颜色设置为洋红色。

MAKE_TMP_NAME (算术函数)

返回临时使用的唯一文件名。

MATCH (算术函数)

如果第一个字符串与第二个字符串指定的阵列相匹配，则返回 1，否则返回 0。

MAX_FEEDBACK (函数)

指定 MODIFY 与 STRETCH 命令中的元素跟踪量。

MEASURE_ANGLE (函数)

测量两个指定元素之间的角度。

MEASURE_AREA (函数)

测量指定圆、弧、样条或圆角封闭的面积。

MEASURE_COORDINATE (函数)

测量选定点的坐标。

MEASURE_DISTANCE (函数)

测量两个指定点之间的距离。

MEASURE_LENGTH (函数)

测量指定线、圆、弧、圆角或样条的长度。

MEASURE_RADIUS (函数)

测量指定圆、弧或圆角的半径。

MENU (函数)

定义屏幕菜单槽的外观和功能。

MENU_LAYOUT (函数)

定义屏幕菜单的形状和位置。

MENU_STATUS (函数)

定义屏幕菜单的状态。

MERGE (命令)

将两个元素合并为一个元素。

MIRR (算术函数)

返回矢量关于虚拟镜像线的镜像图像。

MOD (算术函数)

返回除法余数。

MODIFY (命令)

使用“移动”或“镜像”选项修改元素。

MODIFY_DIM_LINES (命令)

在现有尺寸和延伸线中放置断点。

MOVE_DIMENSION (命令)

移动选定尺寸。

MOVE_TABLE (函数)

将现有表移至指定屏幕位置。只能重新定位不受移动保护的表。

NEW_SCREEN (函数)

重绘整个屏幕。

NOT (算术函数)

如果自变量等于 0 则返回 1。否则返回 0。

NUM (算术函数)

返回字符串中第一个 ASCII 字符的等效十进制数。例如，NUM('hullo') 即为 104。

ON_ERROR (函数)

指定字符串在下一次出现错误时用作输入。

OPEN_INFILE (函数)

使用 READ_FILE 函数打开要读取的已命名文件。

OPEN_OUTFILE (函数)

使用 WRITE_FILE 函数打开要写入的已命名文件。

OR (算术函数)

如果两个自变量均非零，则返回 1。否则，返回 0。

ORIGIN (函数)

在指定视区中，打开或关闭输入坐标系原点的符号。

OUTPUT_HP15 (函数)

将日文汉字文本的输出模式切换为 HP15 代码。

OUTPUT_HP16 (函数)

OUTPUT_HP16 可将日文汉字文本的输出模式切换为 HP16 代码。

OUTPUT_STRING (函数)

将 |字符串| 的内容写入输出设备。

OVERDRAW (命令)

用于拉伸构造几何。该命令会替换拉伸宏的内容。

PARAMETER (伪命令)

在宏定义中用来指示宏参数。

PART_DRW_SCALE (命令)

用于定义指定零件的绘图比例。

PART_DRW_SCALE_REF (函数)

用于指定作为零件起始缩放位置的参考点。

PARTS_LIST (函数)

显示活动零件中不以 "." 开头的零件。并且显示各零件的出现次数。

PASSWORD (函数)

用于向系统中输入启用密码。

PENSIZE (函数)

指定用于所有新的真实几何 (构造几何与点除外) 的当前笔尺寸。

PHANTOM (函数)

将当前剖面线线类型设置为虚线。

PI (算术函数)

返回 3.14159265358979 的 pi 值。

PICK_UTAB_ROW_BY_NAME (函数)

"标注加速模块"屏幕界面的支持函数。不应在命令行中使用。

PICK_VP_PNT (函数)

模拟由编号和视区名称定义的特定视区中的交互式用户拾取。

PICTURE_BROWSER (函数)

显示"图片浏览器"窗口。

PICTURE_LIST (函数)

显示加载至当前会话中的像素映射列表。

PLOT (命令)

出图绘图。

PLOTTER_TYPE (函数)

指定当前连接的绘图仪的类型。

PLOT_AUTO_ROTATE (函数)

允许禁用自动旋转功能，该功能是一些光栅绘图仪的固件组成部分。

PLOT_CENTER (函数)

指定绘图是否在绘图仪视区中居中。

PLOT_DESTINATION (函数)

指定出图目标。

PLOT_FORMAT (函数)

定义最大出图区域 (硬限制)。

PLOT_IMAGE_QUALITY (函数)

定义调色板、分辨率以及缩放等内容。

PLOT_LINETYPE_LENGTH (函数)

定义每种线类型的阵列长度。

PLOT_PEN_TABLE (函数)

控制线类型和颜色到绘图仪笔和绘图仪线类型的映射。

PLOT_SCALE (函数)

指定绘图在出图前的比例因子。

PLOT_STOP_ON_ERROR (函数)

当绘图不适合出图区域时，控制出图行为。

PLOT_TRANSFORMATION (函数)

定义出图过程中特定元素的映射。

PLOT_VIEWPORT (函数)

在 PLOT_FORMAT 指定的最大出图区域中定位绘图仪视区。

PNT_RA (算术函数)

给定长度和角度时，返回 2D 点。

PNT_XY (算术函数)

给定 x 和 y 坐标时，返回 2D 点 (矢量)。

PNT_XYZ (算术函数)

给定 x 坐标、y 坐标以及 z 坐标时，返回 3D 点 (矢量)。

POINT (命令)

创建点元素。

POLYELEM (命令)

能够通过现有元素创建折线。然后可将折线视为单个元素，对其进行相应修改。

POP_DOWN_LTAB (函数)

使已命名逻辑表“压栈”。

POP_UP_LTAB (函数)

使已命名逻辑表“出栈”。

POS (算术函数)

返回字符串中子字符串的第一个位置。

PRE_VIEW (命令)

允许在加载绘图之前进行预览。

PRINT_TABLE (函数)

打印文件中的显示表。

PROMPT_LIST (函数)

在系统内存中最多保存 500 个系统提示。

PURGE_FILE (函数)

删除磁盘中的指定文件。

PUT_PROPERTIES (命令)

选定元素的属性更改为当前属性。

RAC_CHECK (命令)

更新 Creo Elements/Direct Modeling 中的新布局数据，同时将文档数据添加至该布局的旧版本。

RAISE_WINDOW (函数)

使 Creo Elements/Direct Drafting 窗口在 Windows 桌面中提升至所有其他窗口的顶部。

RC_ACCURACY (函数)

指定比较两个 MI 文件时使用的精度。

RC_CHECK (命令)

比较两个零件及其所有子零件，并将比较结果作为零件元素的信息文本存储。

READ (函数)

接受命令行的用户输入，并将数据分配给变量。

READ_FILE (函数)

从指定文件读取一行文本，并将字符串分配给指定变量。

READ_LTAB (算术函数)

返回已命名逻辑表中的值。

RECALL_BUFFER (函数)

最多保存 63 个输入行。可以使用 [Prev] 与 [Next] 键重新调用这些行。

RECALL_WINDOW (函数)

当前视区窗口更改为使用 STORE_WINDOW 存储的窗口。

RED (函数)

将默认线颜色设置为红色。

REDRAW (函数)

重绘当前视区的内容。

REDRAW_SCENE (函数)

重绘场景视区。

RENAME_ELEMENT (命令)

重命名或复制元素，包括该元素的“所有”修订版本和版本。

RENAME_PART (命令)

重命名活动零件。

RENOVATE (函数)

恢复选定视区的内容。

REPEAT (伪命令)

定义重复执行的循环，该循环在 UNTIL 子句中的布尔表达式计算出逻辑真时结束。

REQUEST_PRINT_SETUP (函数)

设置向“打印管理器”发送出图或屏幕转储时，是否使 Windows 打印管理器显示自己的对话框。

RESET_PART_NUMBER (命令)

对所有以 TOP 零件开始的“唯一”零件编号重新编号。

RESTORE (命令)

将已归档元素及其相关文件从“源”恢复到数据库。

RGB_COLOR (函数)

用于指定新颜色 (红色、绿色和蓝色)。

RND (算术函数)

返回介于 0 < = X < 1 范围内的伪随机 X 编号。

ROT (算术函数)

返回按给定角度绕原点旋转的点 (矢量)。

ROTATE_DIM_TEXT (命令)

按指定角度旋转选定尺寸文本。

ROUND (算术函数)

返回舍入为最近整数的自变量。例如，ROUND(4.4999) 等于 4。

RPT (算术函数)

返回所需的字符串份数。

RTL_COLOR (命令)

设置参考线颜色。

RTL_DST_GAP (命令)

设置参考线间隙 (目标)。

RTL_LINETYPE (命令)

设置参考线线类型。

RTL_LINEWIDTH (命令)

设置参考线笔尺寸。(等同于 RTL_PENSIZE 操作。)

RTL_PENSIZE (命令)

设置文本参考线笔尺寸。

RTL_SRC_GAP (命令)

设置参考线间隙 (源)。

RULER (函数)

在指定视区中打开或关闭标尺。

RUN (函数)

返回至操作系统 shell 提示。

SAVE (命令)

将内存中的绘图保存至指定文件。

SAVE_ENVIRONMENT (函数)

将环境输出至指定目标。

SAVE_FONT (函数)

将所有字体或指定字体保存至指定文件。

SAVE_LTAB (函数)

将已命名逻辑表保存至“输出特定值”(有关详细信息，请参阅 OUTPUT_SPEC 帮助)。

SAVE_MACRO (函数)

将已命名宏或所有宏输出至指定目标。

SAVE_MENU (函数)

将当前屏幕菜单定义输出至指定目标。

SAVE_TABLE (函数)

保存文件中设置的表。

SAVE_VIEWPORT (函数)

将当前视区定义输出至指定目标。

SCREEN_TRANSFORMATION (函数)

定义重绘过程中特定元素的映射。

SCROLL_LTAB (函数)

滚动与已命名逻辑表相连的显示表，以便指定行位于显示画面顶部。

SEARCH (命令)

指定搜索列表中的目录。

SECURE_LTAB (函数)

保护已命名的用户表。不能删除受保护的用户表。

SECURE_MACRO (函数)

防止宏被列出、编辑、保存或追踪。

SECURE_TABLE (函数)

保护显示表防止其被删除。表受到保护后不能被删除或重新定义。

SELECT_DIM_ARROW (函数)

指定当前尺寸线终止符。

SELECT_FROM_LTAB (函数)

在源表上执行选择操作。

SGN (算术函数)

如果自变量是负值，则返回 -1，如果自变量是 0，则返回 0，如果自变量是正值，则返回 1。

SHARE_PART (命令)

使指定零件变成共享零件。

SHOW (函数)

选择性打开或关闭元素。以不同的颜色显示元素，仅显示框形轮廓。

SHOW_CPOLY (函数)

显示 B 样条的控制多边形。

SHOW_PART (函数)

更改零件在当前视区中的显示方式。

SHOW_TABLE (函数)

显示或拭除现有表。

SHOW_TABLE_PAGE (函数)

显示已附加至指定显示表的查询页面。

SIN (算术函数)

返回自变量的正弦。

SL_COLOR (命令)

设置对称线颜色。

SL_LINETYPE (命令)

设置对称线线类型。

SL_LINEWIDTH (命令)

设置对称线笔尺寸。(等同于 SL_PENSIZE 操作。)

SL_PENSIZE (命令)

设置对称线笔尺寸。

SL_OFFSET (命令)

设置对称线偏移。

SMASH_POLY (命令)

用于将多边形分解为单个元素。

SMASH_SUBPART (命令)

将选定零件的所有元素引入活动零件中，并删除子零件。选定零件必须是活动零件的子零件。

SNID (算术函数)

返回 HP-HIL 安全设备的产品编号和序列号。如果不存在安全设备，则返回计算机的产品编号和序列号。

SOLID (函数)

将默认线类型设置为实线。

SORT_LTAB (函数)

根据指定列对用户表进行排序。

SPLINE (命令)

创建样条。

SPLINE_CONVERSION (函数)

将本程序早期版本创建的样条转换为 B 样条。

SPLIT (命令)

分割线、圆、弧、圆角和样条。

SPLITTING (函数)

打开或关闭自动分割和合并。

SPOTLIGHT (函数)

活动零件保持原样。所有其他几何重绘为洋红色并带有虚线。

SQR (算术函数)

返回自变量的平方。

SQRT (算术函数)

返回自变量的平方根。

STATLINE_RESET (函数)

在已自动禁用状态行的情况下重新将其激活。

STORE (命令)

将绘图存储至指定文件。(使用 MI 格式 2.30。)

STORE_202 (命令)

使用 MI 格式 2.02 存储绘图。

STORE_210 (命令)

使用 MI 格式 2.10 存储绘图。

STORE_211 (命令)

使用 MI 格式 2.11 存储绘图。

STORE_221 (命令)

使用 MI 格式 2.21 存储绘图。

STORE_230 (命令)

使用 MI 格式 2.30 存储绘图。

STORE_231 (命令)

使用 MI 格式 2.31 存储绘图。

STORE_FONT (函数)

将所有字体或指定字体存储至已命名文件。

STORE_IN_RECALL_BUFFER (函数)

将给定字符串存储至重新调用缓冲区以供后续使用。

STORE_MACRO (函数)

将已命名宏或所有宏存储至指定目标。

STORE_WINDOW (函数)

存储当前视区的窗口坐标。

STR (算术函数)

返回自变量的 ASCII 表示。例如，

STRETCH (命令)

拉伸线、圆、弧、样条和指引线。

STRUCTURE (函数)

查询以显示层次结构。

SUBSTR (算术函数)

返回字符串的子字符串。

SYMBOL_PART (命令)

使选定零件成为符号。

SYMLINE (命令)

创建对称线。

TABLE_COLUMN (函数)

设置现有未受保护表的数据列中的单元格属性。

TABLE_LAYOUT (命令)

用于在指定位置创建指定形状的新表。

TABLE_SCROLL_STEP (函数)

使用滚动条时用于设置显示表的滚动步长。

TABLE_TITLE (函数)

设置不受标题更改保护的现有表中的标题槽属性。

TAN (算术函数)

返回自变量的正切。

TEXT (命令)

创建文本。

TEXT_ADJUST (函数)

指定新文本的文本原点位置。

TEXT_ANGLE (函数)

指定当前文本角度。

TEXT_FILL (函数)

打开或关闭字符填充。

TEXT_HOLE_INSERTION (函数)

在剖面线区域中插入文本窗口。

TEXT_FRAME (函数)

设置当前文本框架的类型。

TEXT_LINESPACE (函数)

设置新文本的当前行间距。

TEXT_RATIO (函数)

设置字符宽度与高度的当前比率。

TEXT_SIZE (函数)

设置当前文本高度。

TEXT_SLANT (函数)

设置当前文本倾斜。

TEXT_TO_GEO (命令)

将文本转换为几何。

TIME (算术函数)

设置午夜过后的秒数。

TONE (函数)

生成具有指定频率、持续时间和幅度的可听音调。

TRACE (函数)

打开或关闭 TRACE 文件。

TRAP_ERROR (函数)

定义出现错误时的行为。如果没有 TRAP_ERROR，出现错误时将停止所有执行。启用 TRAP_ERROR 之后，只会发现出现错误，但执行不会停止。

TRIM_ONE (命令)

用于修剪或延伸元素以便与其他元素准确相交。

TRIM_TWO (命令)

用于修剪或延伸两个元素以便相交。

TRIM (算术函数)

返回通过剥离自变量中的前导空格和尾随空格而形成的字段串。

TRIMMING (命令)

打开或关闭自动修剪 (分割后移除样条的不可见部分)。

TRUE (算术函数)

返回 1。

TRUE_COLOR_PLOTTING (命令)

启用真彩色出图并关闭出图变换。

TRUNC (算术函数)

返回实数的整数部分。

TYPE (算术函数)

返回指定令牌的类型。

TXT_WINDOW (命令)

指定剖面线区域中的文本窗口。

UA_ANGLE_GRID (函数)

设置 COPILOT 使用的角度增量。

UA_CENTER_CATCH_RANGE (函数)

指定 COPILOT 将在捕捉时视为中心的每条线的长度部分。

UA_DISTANCE_GRID (函数)

设置 COPILOT 使用的长度增量。

UA_GET_DESIGN_INTENT (算术函数)

以“设计意图”的开/关响应。

UA_PERPENDICULAR_CATCH_RANGE (函数)

指定 COPILOT 将在捕捉时视为垂直的线、圆和弧的部分。

UA_TANGENT_CATCH_RANGE (函数)

指定 COPILOT 将在捕捉时视为相切的圆和弧的部分。

UA_SET_CATCH_DELAY (函数)

设置显示 COPILOT 捕捉信息之前光标须静止不动的时间。

UNITS (函数)

指定距离和角度的当前单位。

UNLOAD_MODULE (命令)

卸载模块。

UNSHARE_PART (命令)

使共享零件变为非共享零件。

UNTIL (伪命令)

布尔表达式

UPC (算术函数)

将小写字符转换为大写。

UPDATE_SCREEN (函数)

所有设备或驱动器缓冲内容均显示在屏幕中。状态行和菜单会进行更新。

USE_MULTILINE_HATCH (函数)

允许为距离为 0 的剖面线选择两种不同的剖面线处理。

VAL (算术函数)

将数值字符串转换为数值。

`VERSION` (函数)

显示关于 CAD 软件版本的信息。

`VIEW` (函数)

使选定零件可在当前视区中查看。

`WAIT` (函数)

使系统在指定秒数内不执行任何操作。

`WHILE` (函数)

只要 WHILE 语句后面的布尔表达式为真，便执行下一个代码部分。

`WHITE` (函数)

将默认线颜色设置为白色。

`WINDOW` (函数)

用于控制在当前视区中显示绘图的哪些部分。

`WINEXEC` (算术函数)

启动给定命令字符串中命名的 Windows 应用程序。

`WRITE_FILE` (函数)

将文本行写入指定文件。

`WRITE_LTAB` (函数)

将新值写入已命名用户表。

`X_OF` (算术函数)

返回矢量的 X 坐标。

`YELLOW` (函数)

`Y_OF` (算术函数)

返回矢量的 Y 坐标。

`Z_OF` (算术函数)

返回矢量的 Z 坐标。

逻辑表和显示表

什么是逻辑表和显示表？ 179
逻辑表 179
显示表 180
将显示表连接到逻辑表的概念 181
逻辑表访问函数 182
LTAB_COLUMNS 182
LTAB_ROWS 183
LTAB_TITLES 183
POP_DOWN_LTAB 184
POP_UP_LTAB 185
READ_LTAB 186
SAVE_LTAB 186
SCROLL_LTAB 187
SELECT_FROM_LTAB 187
显示表函数 189
TABLE_COLUMN 190
TABLE_LAYOUT 192
TABLE_TITLE 195
CHANGE_TABLE_SIZE 198
CONNECT_TABLE 198
DELETE_TABLE 199
MOVE_TABLE 199
PRINT_TABLE 200
SAVE_TABLE 201
SECURE_TABLE 202
SHOW_TABLE 202
TABLE_SCROLL_STEP 203
使用表函数 203
COLOR_LTAB 204
CREATE_LTAB 205
DELETE_LTAB 206
DELETE_LTAB_ROW 206
HIGHLIGHT_LTAB 207
SECURE_LTAB 208
SORT_LTAB 208

WRITE_LTAB......209
使用逻辑表和显示表 - 示例 1......210
定义用户表......210
定义显示表......211
与显示表交互......212
使用逻辑表和显示表 - 示例 2......213
定义第一个用户表和显示表......213
定义第二个用户表和显示表......215
与用户表和显示表交互......218
备注......219

本附录提供了使用逻辑表和显示表所需的信息。

其介绍了逻辑表和显示表的结构，并解释了将显示表连接到逻辑表的概念。

然后介绍了用于访问逻辑表的函数以及定义和使用用户表所需的命令和函数。

此外，还介绍了定义和使用显示表所需的命令和函数。

最后，提供了两个示例，用以介绍定义逻辑表和显示表、将显示表映射到逻辑表和使用这些表的常规过程。

注意

本附录中的说明和示例假定您已熟悉 Creo Elements/Direct Drafting 中的宏编程。

什么是逻辑表和显示表？

逻辑表和显示表是两种技术，用于通过表格方式提供大量数据。其可帮助用户读取和理解数据。使用这些技术的另一个优势是用户可以从这些表中选择条目作为命令或函数的输入，从而增强系统的用户界面。

您可以在通常需要处理和显示数据列表的时候使用逻辑表和显示表。有关使用这些表的好处，请参阅“使用逻辑表和显示表 - 示例 1”和“使用逻辑表和显示表 - 示例 2”。

以下部分介绍了逻辑表和显示表的结构，并解释了将显示表连接到逻辑表的概念。

逻辑表

逻辑表是系统或用户定义的基本内部数据结构。

系统定义的逻辑表包含系统数据。尽管它们都是可读和可写的，但不得更改系统定义逻辑表中的布局或数据。仅允许从系统定义逻辑表读取数据。

用户定义逻辑表 (也称为用户表) 包含用户数据，它们都是可读和可写的。您可以创建和更改用户定义逻辑表的布局。此外，也可从用户定义逻辑表读取数据和向其写入数据。

逻辑表通常由三个主要组件组成：

1. 通过编号 1 到 N 标识的 N 个记录构成的数组，其中，每个记录还包含通过编号 1 到 M 标识的 M 个值。换句话说，其与二维 N x M 数值矩阵类似。数组中的记录数可以变化，但对于所有记录，各记录中的数值数必须保持相同。

> 注意
>
> 逻辑表中的值可以是文本字符串或数字。

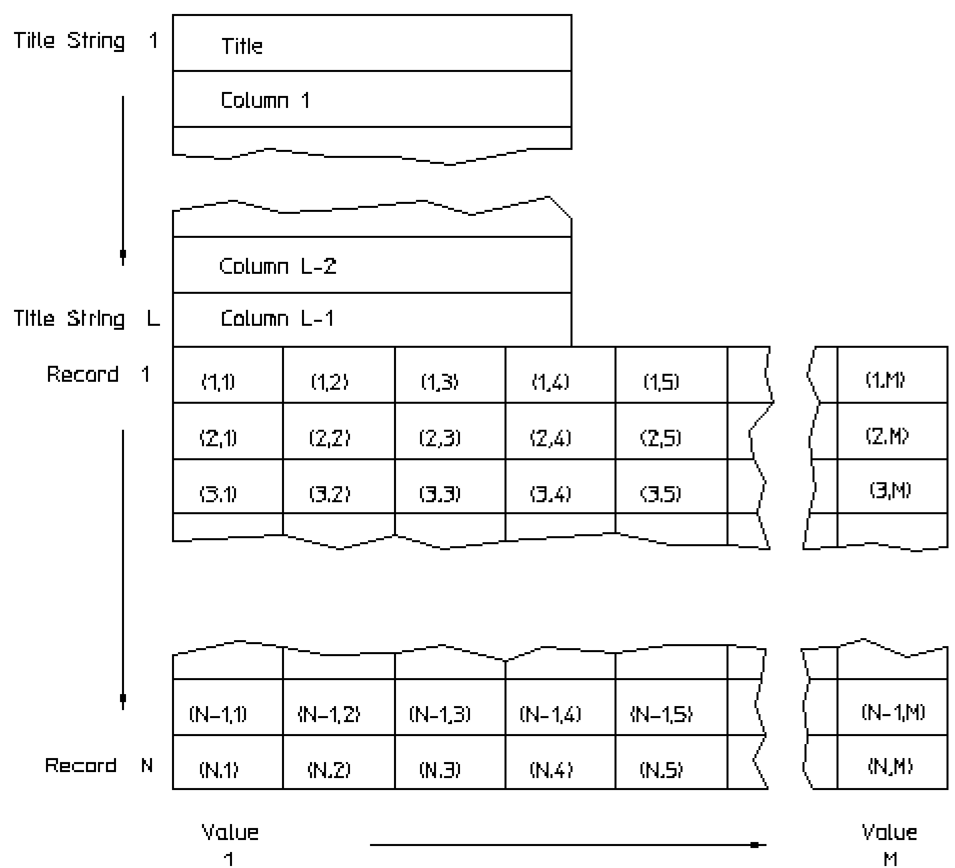
2. 通过编号 1 到 L 标识的 L 个标题字符串。标题字符串是文本字符串，可以通过它来提供附加信息，而这些信息通常是为显示表标题提供。有关详细信息，请参阅“显示表”部分。

> 注意
>
> 如果不需要任何标题字符串，则不必进行定义。此外，也可不在标题字符串中输入数据，即标题字符串为空。

3. 所有记录中值的状态。每个值都具有相关联的状态，选中或激活时指示该值。该信息非常有用。例如，如果用户已从显示表中选择某些数据作为命令的输入，则该数据将突出显示以指示用户的选择。
4. 表格中各单元格的前景 (显示) 和背景颜色。

下面给出了逻辑表的示意图：

图 **31.**逻辑表



显示表

显示表是用于在屏幕上显示逻辑表内容的表。显示表可由系统或用户定义。显示表通常由以下组件组成：

- 标题，例如列标题和子标题

- 包含值列的数据区
- 可选垂直滚动条，显示在表的右侧。不允许水平滚动条。

下面给出了显示表的示例：

图 **32.**显示表

TITLE			
Column 1	Column 2	Column 3	Column 4
(1.2)	(1.4)	(1.3)	(1.5)
(2.2)	(2.4)	(2.3)	(2.5)
(3.2)	(3.4)	(3.3)	(3.5)
(4.2)	(4.4)	(4.3)	(4.5)
(5.2)	(5.4)	(5.3)	(5.5)
(6.2)	(6.4)	(6.3)	(6.5)
(7.2)	(7.4)	(7.3)	(7.5)
(8.2)	(8.4)	(8.3)	(8.5)

Title Area
Data Area
Scroll Up Box
Scroll Bar
Scroll Down Box

将显示表连接到逻辑表的概念

如前面部分所述，共有两类表格：逻辑表和显示表。逻辑表是存储实际数据的内部数据结构，而显示表用作用户查看逻辑表中数据的窗口。通过显示表还可与逻辑表进行交互来更改其数据。

在定义时为各逻辑表或显示表赋予唯一名称，您可以根据需要数量定义逻辑表或显示表。

仅当显示表连接到逻辑表后，它才可访问和查看该逻辑表。您可以将多个显示表连接到一个逻辑表，这样当逻辑表包含大量数据时，您可以使用这些显示表查看该逻辑表的不同部分。

该技术的优势在于，存储在逻辑表中的数据一致地提供给用户，并且当逻辑表中的数据发生更改时将自动更新显示表。通过该技术还可以不必再次定义相同的显示表。

下图显示了显示表到逻辑表的映射：

图 **33.**将显示表映射到逻辑表

COMPONENTS			
Quantity			
Name			
Type			
Manufacture Date			
BOLTS	A	4	10-8-89
NUTS	A	4	5-8-89
SCREWS	D	8	15-7-89
FLANGES	G	4	27-6-89

COMPONENTS			
Quantity	Name	Type	Manufacture Date
4	BOLTS	A	10-8-89
4	NUTS	A	5-8-89
8	SCREWS	D	15-7-89
4	FLANGES	G	27-6-89

上图中的箭头指示了逻辑表中的条目映射到显示表中条目的方式。

逻辑表中的标题字符串通常映射到显示表中的表标题和列标题。您可以采用任何顺序将逻辑表中的数据列映射到显示表中的数据列。您也可仅将逻辑表中的某些数据列映射到显示表。

逻辑表访问函数

通过逻辑表访问函数可以访问任何逻辑表，无论逻辑表是系统定义还是用户定义。这些函数包括：

- LTAB_COLUMNS
- LTAB_ROWS
- LTAB_TITLES
- POP_DOWN_LTAB
- POP_UP_LTAB
- READ_LTAB
- SAVE_LTAB
- SCROLL_LTAB
- SELECT_FROM_LTAB

以下部分详细介绍了这些函数：

LTAB_COLUMNS

通过该命令可以查询并返回指定逻辑表中的列数。

命令格式如下：

```
LTAB_COLUMNS 'Logical table name'
```

下面给出了命令示例：

```
LTAB_COLUMNS  'logtable1'
```

LTAB_COLUMNS 需要一个参数，即要查询的逻辑表名称。本例中，逻辑表为 logtable1

由于 LTAB_COLUMNS 命令返回一个值，所以必须将此值分配给某一变量，如下例所示。LTAB_COLUMNS 命令的典型应用为：

```
LET  num_columns  (LTAB_COLUMNS  'logtable1')
```

其中 LTAB_COLUMNS 命令返回逻辑表 logtable1 中的列数，然后 LET 将其分配给宏变量 num_columns。

返回值为数字。

LTAB_ROWS

通过该命令可以查询并返回指定逻辑表中的行数。

命令格式如下：

```
LTAB_ROWS 'Logical table name'
```

下面给出了命令示例：

```
LTAB_ROWS  'logtable1'
```

LTAB_ROWS 需要一个参数，即要查询的逻辑表名称。本例中，逻辑表为 logtable1

由于 LTAB_ROWS 命令返回一个值，所以必须将此值分配给某一变量。LTAB_ROWS 命令的典型应用为：

```
LET  num_rows  (LTAB_ROWS  'logtable')
```

其中 LTAB_ROWS 命令返回逻辑表 logtable1 中的行数，然后 LET 将其分配给宏变量 num_rows。

返回值为数字。

LTAB_TITLES

通过该命令可以查询并返回指定逻辑表中的标题字符串数。

命令格式如下：

```
LTAB_TITLES 'Logical table name'
```

下面给出了命令示例：

```
LTAB_TITLES  'logtable1'
```

LTAB_TITLES 需要一个参数，即要查询的逻辑表名称。本例中，逻辑表为 logtable1

由于 LTAB_TITLES 命令返回一个值，所以必须将此值分配给某一变量。LTAB_TITLES 命令的典型应用为：

```
LET  num_titles  (LTAB_TITLES  'logtable1')
```

其中 LTAB_TITLES 命令返回逻辑表 logtable1 中的标题数，然后 LET 将其分配给宏变量 num_titles。

返回值为数字。

POP_DOWN_LTAB

通过该命令可以使指定逻辑表在下次条件满足时消失。逻辑表消失时，连接到该逻辑表的所有显示表都将从屏幕中移除。

消失请求先存储在缓冲区中，直到系统能够从用户接受交互输入或调用 UPDATE_SCREEN 为止。在此之后系统才会处理消失请求。

命令格式如下：

```
POP_DOWN_LTAB 'Logical table name'
```

下面给出了命令示例：

```
POP_DOWN_LTAB  'logtable1'
```

POP_DOWN_LTAB 需要一个参数，即要消失的逻辑表名称。本例中，逻辑表为 logtable1

注意

- POP_DOWN_LTAB 命令使逻辑表消失时，连接到该逻辑表的所有显示表都将从屏幕中移除。如果仅希望移除一个显示表，则可使用具有 OFF 选项的 SHOW_TABLE 命令进行该操作。
- POP_DOWN_LTAB 命令从屏幕中移除显示表时，会恢复移除显示表下的屏幕区域内保存的位图图像，因此看起来与以前一样。但是当显示表重叠时，可能会出现特殊情况。例如，屏幕区域内保存的位图图像包含整个或部分显示表，但该显示表保存到该位图图像中后又从屏幕中移除。当恢复位图图像时，该显示表再次出现在屏幕上。但是，对于系统，该显示表并不存在，因此您会在屏幕上看到无法使用 TABLES OFF 命令移除的部分或整个表。如果出现这种情况，您可以使用 NEW_SCREEN 命令将其从屏幕中移除。NEW_SCREEN 命令用于重新生成整个屏幕的实际内容。

POP_UP_LTAB

通过该命令可以使指定逻辑表在下次条件满足时弹出。逻辑表弹出时，连接到该表的所有显示表都将显示在屏幕上。

弹出请求先存储在缓冲区中，直到系统准备从用户接受交互输入为止。在此之后系统才会处理弹出请求。

命令格式如下：

```
POP_UP_LTAB 'Logical table name'
```

下面给出了命令示例：

```
POP_UP_LTAB  'logtable1'
```

POP_UP_LTAB 需要一个参数，即要弹出的逻辑表名称。本例中，逻辑表为 logtable1。

注意

- 对逻辑表使用 POP_UP_LTAB 命令时，连接到该逻辑表的所有显示表都将显示在屏幕中。如果仅希望显示一个显示表，则可使用具有 ON 选项的 SHOW_TABLE 命令进行该操作。
- POP_UP_LTAB 命令在屏幕上显示显示表时，会保存显示表下的屏幕区域内的位图图像。当使用 POP_DOWN_LTAB 命令从屏幕中移除显示表时，该区域恢复到原始图像。

READ_LTAB

通过该命令可以从指定逻辑表中的某个区域读取值。

命令格式如下：

```
READ_LTAB 'Logical table name'
(Row-number Column-number) or (TITLE Title-string-number)
```

下面给出了命令示例：

```
READ_LTAB  'logtable1'
2  3
```

READ_LTAB 需要两个参数。第一个参数是要从其读取数据的逻辑表名称。本例中，逻辑表为 logtable1。第二个参数是要从指定逻辑表读取的数据位置，在本例中为 2 3，即指定逻辑表的第 2 行第 3 列。

如果要从指定逻辑表的标题字符串读取值，则指定 TITLE 5 而不是 2 3 来指示标题字符串 5。

由于 READ_LTAB 命令返回一个值，所以必须将此值分配给某一变量。READ_LTAB 命令的典型应用为：

```
LET  my_value  (READ_LTAB  'logtable1' 1  4)
```

其中 READ_LTAB 命令从逻辑表 logtable1 的第一行第四列读取值，然后 LET 将其分配给宏变量 my_value。

返回的值可能是字符串或数字。如果指定位置不包含值，则将返回空字符串。

SAVE_LTAB

使用该命令可保存指定逻辑表中的内容。

命令格式如下：

```
SAVE_LTAB 'Logical table name'
SCREEN or (( or DEL_OLD or APPEND) 'filename')
```

下面给出了命令示例：

```
SAVE_LTAB  'logtable1'
DEL_OLD  'file1.tbl'
```

SAVE_LTAB 需要两个参数。第一个参数是要保存逻辑表的名称。本例中，逻辑表为 logtable1。第二个参数指示指定逻辑表保存到的输出目标。本例中，输出目标是具有 DEL_OLD 选项的 file1.tbl 文件，这意味着如果 file1.tbl 已存在，则会将其覆盖。

如果将某一文件指定为输出目标，那么也可以使用 APPEND 选项或根本不使用选项。APPEND 表示指定逻辑表会附加到指定文件的结尾 (如果文件存在)。如果文件不存在，则会创建指定文件。如果不指定任何选项，则只要不存在其他具有相同名称的文件，便会创建指定文件。如果指定文件已存在，则显示错误消息并中止命令。

或者，也可以通过指定 SCREEN 来替代上述示例中的 DEL_OLD 和 file1.tbl，将屏幕指定为输出目标。然而，将表保存至文件中的优点是，可以随后使用 INPUT 命令来输入此文件以重新定义该表。

SCROLL_LTAB

使用该命令可以滚动连接至指定逻辑表的显示表，这样指定行就会位于显示表的顶部。

命令格式如下：

```
SCROLL_LTAB 'Logical table name'
Row-number
```

下面给出了命令示例：

```
SCROLL_LTAB  'logtable1'
3
```

SCROLL_LTAB 需要两个参数。第一个参数是要滚动的逻辑表的名称。本例中，逻辑表为 logtable1。第二个参数指示要滚动至显示表顶部的行，在本示例中为 3。

SELECT_FROM_LTAB

使用该命令可以根据用户指定的特定条件，从称为源逻辑表的表中选择行，并将匹配行的位置写入逻辑表 sys_select。或者，使用该命令也可以复制匹配行至另外一个逻辑表，叫做目标逻辑表。

命令格式如下：

```
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')
```

下面给出了命令示例：

```
SELECT_FROM_LTAB  'logtable1'
COLUMN  1
 >=
10
END
```

SELECT_FROM_LTAB 需要五个参数。第一个参数是要从中选择数据的源逻辑表的名称。以上示例中，逻辑表为 logtable1。第二个参数是用于选择的列的位置，即以上示例中的第一列 COLUMN 1。第三个参数是选择运算符，即以上示例中的 >= (大于等于)。可以指定的替代运算符有 =、<>、>、< 以及 <=。第四个参数是选择字符串或值，即以上示例中的 10，是数字 10 而不是字符串 '10'。第五个参数是 END 或者选定行要复制到的目标逻辑表的名称。在以上示例中，指定 END 代表命令的结束。

简而言之，以上示例表示如果行的 COLUMN 1 包含值 = 10，则从源逻辑表 logtable1 中选择行，并将选定行的位置写入逻辑表 sys_select。

例如，如果逻辑表 logtable1 包含以下数据：

```
1    20
5    25
10   11
22   17
43   13
```

并在上述示例中运行宏程序。结果为逻辑表 sys_select 包含以下数据：

```
3
4
5
```

因为仅有 logtable1 的第 3、4、5 行满足选择条件。

另一命令示例为：

```
SELECT_FROM_LTAB  'logtable1'
COLUMN  2
=
'PARTS'
APPEND  'logtable2'
```

简而言之，以上示例代表如果行的 COLUMN 2 包含字符串 = 'PARTS'，则从源逻辑表 logtable1 中选择行，并将选定行的位置写入逻辑表 sys_select。此外，它将 APPEND，即复制选定行至目标逻辑表 logtable2 的结尾。

指定选项 APPEND 时，该命令假定已存在指定的逻辑表。如果不指定选项 APPEND，则该命令假定不存在指定的逻辑表，并创建具有指定名称的新逻辑表。

例如，如果逻辑表 logtable1 和 logtable2 分别包含以下数据：

```
PISTON        DRAWING     4
BOLTS         PARTS       10
NUTS          PARTS       10
CRANKSHAFT    DRAWING     1
WASHERS       PARTS       10
```

和

```
WHEEL         ASSEMBLY    2
FRAME         ASSEMBLY    1
HANDLE        ASSEMBLY    1
```

并在上述示例中运行宏程序。结果为逻辑表 sys_select 和 logtable2 分别包含以下数据：

```
2
3
5
```

和

```
WHEEL         ASSEMBLY    2
FRAME         ASSEMBLY    1
HANDLE        ASSEMBLY    1
BOLTS         PARTS       10
NUTS          PARTS       10
WASHERS       PARTS       10
```

因为仅有 logtable1 的第 2、3 和 5 行满足选择条件。可以看到 logtable1 的第 2、3 和 5 行的内容复制并附加至 logtable2 的结尾。

显示表函数

显示表提供了显示逻辑表的所有或选定内容的方式。有关详细信息，请参阅“显示表”部分。

以下部分介绍定义显示表的布局和使用显示表所需的命令和函数。还有定义显示表的示例。

注意

在以下各部分列出的每种命令格式中，正常字体文本表示它们是格式的一部分，因此必须按此处显示的内容进行指定。斜体文本表示必须将其替换为合适的字符串或值。空行用于帮助您阅读格式和理解其结构。单词 or 表示存在两个或多个选项，并可为参数选取其中一个选项。

用于定义显示表的命令包括：

- TABLE_COLUMN
- TABLE_LAYOUT
- TABLE_TITLE

有关如何定义显示表的详细信息，请参阅“使用逻辑表和显示表 - 示例 1”中的“定义显示表”部分。

以下部分详细介绍了上述每种命令。

TABLE_COLUMN

使用该命令可以设置现有显示表数据列中槽的属性。

注意

此命令只能在未通过 SECURE_TABLE 命令进行防更改保护的显示表上运行。

命令格式如下：

```
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
```

下面给出了命令示例：

```
TABLE_COLUMN 'table1'
  COLUMN 1
  blue
  yellow
  1
  FORMAT 2
  LEFT
  'LINE'

  COLUMN 2
  red
  green
  2
  FORMAT 3
  RIGHT
  'CIRCLE'
```

```
END
```

注意

上例中，左侧数字不是格式的一部分。它们用于帮助您参考示例中的每一行。空行用于帮助您阅读示例和理解其结构。

第 1 行包括 TABLE_COLUMN 和要定义列的显示表的名称。

第 2 行指定 COLUMN 1，表示将要定义显示表的第 1 列。

第 3 行指定列中文本的显示颜色。本例中指定的是 blue。如果需要使用其他颜色，可按 TABLE_LAYOUT 第 3 行部分中的说明指定颜色。

第 4 行指定列的背景颜色。本例中指定的是 yellow。如果需要使用其他颜色，可按 TABLE_LAYOUT 部分中的说明指定颜色。

第 5 行指定要显示在显示表的这一列中的列在连接逻辑表中的位置。

第 6 行指定 FORMAT 2，表示浮点数的格式精度为两位有效数字。例如，如果浮点数是 123.0，实际数值将是 120。如果浮点数是 12.6，实际数值将是 13。默认的格式精度是列宽度的一半。

或者，可以指定数字字符串，例如 '+1.2345'，表示数字将标正负号、有四位小数，且将消去左侧和右侧的零。有关此数字字符串格式的详细信息，请使用 HELP 命令来参阅 DIM_FORMAT 函数。

第 7 行指定 LEFT，表示列中的数据左对齐。还可以指定其他两个值 CENTER 和 RIGHT，分别表示居中对齐和右对齐。

第 8 行指定列中的操作文本 LINE。可以指定是有效系统命令的任意文本字符串。

可以指定 @s# 格式的字符串，表示文本字符串来自与此显示表连接的逻辑表中编号为 # 的标题字符串。例如，如果指定 @s3，表示连接逻辑表中的额外字符串 3 就是实际指定的文本字符串。如果指定 @t3，表示连接逻辑表中在单引号内的额外字符串 3 就是实际指定的文本字符串。除了 @t3 在额外字符串 3 前后各有一个单引号之外，@s3 和 @t3 相同。

也可以指定 @v# 格式的字符串，表示文本字符串来自与此显示表连接的逻辑表中第 # 列的数据。例如，如果指定 @v4，表示连接逻辑表第 4 列中的数据就是实际指定的文本字符串。如果指定 @q4，表示连接逻辑表中在单引号内的第 4 列中的数据就是实际指定的文本字符串。除了 @q4 在第 4 列中的数据前后各有一个单引号之外，@v4 和 @q4 相同。

您甚至可以指定 LINE @s4 @v2 形式的文本字符串。如果连接逻辑表中的额外字符串 4 和第 2 列中的数据分别是 TWO_PNTS 和 33,33，那么实际文本字符串是 LINE TWO_PNTS 33,33。

第 10 至 16 行是与第 2 至 8 行类似的另一个参数块，用于定义显示表标题中的另一列。

第 17 行指定 END 来指示 TABLE_COLUMN 命令的结束。

注意

您可以根据需要包括任意数量的与第 2 至 8 行类似的参数块，只要这些参数块在命令 TABLE_COLUMN 和 END 中即可。

实际上可将上例重新调整为：

```
TABLE_COLUMN 'table1'
  COLUMN 1 blue yellow 1 FORMAT 2 LEFT  'LINE'
  COLUMN 2 red  green  2 FORMAT 3 RIGHT 'CIRCLE'
END
```

如果行中能容纳参数块，则此格式可能更易于阅读和比较。重点是参数必须至少用一个空格或制表符分隔。

TABLE_LAYOUT

使用该命令可以创建具有所需布局的新表。

命令格式如下：

```
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
```

下面给出了命令示例：

```
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
```

注意

上例中，左侧数字不是格式的一部分。它们用于帮助您参考示例中的每一行。空行用于帮助您阅读示例和理解其结构。

第 1 行包含命令 TABLE_LAYOUT 和此显示表的指定名称。

第 2 行指定该显示表的逻辑表的名称。本例中指定的是 query_results。默认情况下，逻辑表的名称与显示表的名称相同。

第 3 行指定显示表中框架和文本的显示颜色。本例中指定的是 white。还可以指定已在系统中定义的任何其他颜色，例如 red, yellow, green, cyan, magenta, blue 或 black。

如果需要使用系统中定义的以上八种颜色之外的颜色，可以先指定命令 rgb_color，然后按照红、绿、蓝的顺序输入各自的小数值。例如，如果想要纯红色、绿色或者蓝色，可以分别指定 rgb_color 1 0 0、rgb_

color 0 1 0 或者 rgb_color 0 0 1。通过结合这三种颜色可以定义所需的特定颜色，例如 rgb_color 0.5 0.42 0.8，表示红色占 0.5，绿色占 0.42，而蓝色占 0.8。

第 4 行指定显示表的背景颜色。本例中指定的是 black。

如果需要使用其他颜色，可按第 3 行部分中的说明指定颜色。

第 5 行包含 WIDTH 和表宽度 30.0 字符。这个值可以是小数，例如 35.5，而且必须大于特定的最小宽度。如果未指定该参数，则使用适合此表的最小宽度。

第 6 行包含 HEIGHT 和表高度 28.0 字符。这个值可以是小数，例如 35.5，同时必须大于特定的最小高度。如果未指定该参数，则使用适合此表的最小高度。

第 7 行包含 ROWS 和显示表中所需的数据行数 10。该参数可以是小数，例如 10.5。只有在未明确指定表的宽度和高度时，使用该参数才有用。

第 8 行包含 FRAME_WIDTH 和框架宽度值 1。该参数的取值可以是 0、1 或 2。

第 9 行包含 HORIZONTAL、水平线颜色 white 及其类型 solid。如果水平线需要使用其他颜色，可按第 3 行部分中的说明指定颜色。如果需要使用其他线类型，可以指定已在系统中定义的其他线类型中之一，例如 dotted、dashed、long dashed、dot center、dash center、phantom 或 long dotted。如果在 HORIZONTAL 之后指定 OFF，在显示表中将不会出现水平线。

第 10 行包含 VERTICAL、竖直线颜色 white 及其类型 solid。如果需要使用其他颜色或者线类型，可按第 9 行部分中的说明指定颜色或者线类型。如果在 VERTICAL 之后指定 OFF，在显示表中将不会出现竖直线。

第 11 行包含 SCROLL_BAR、前景颜色 blue、背景颜色 white 以及滚动条宽度 30 像素。

第 12 行指定显示表左下角的像素坐标 (0,0)。或者也可以指定显示表右上角的坐标。

第 14 行包含 TITLE_LAYOUT，是指定显示表中标题布局的参数。

第 15 行指定第一个标题行的高度 40 像素和布局字符串 ''，后者指示槽的数量和每个槽的宽度。在该示例中只有一个 50 个字符宽度的槽。

第 16 行指定第二个标题行的高度 20 像素和布局字符串 '||||'，后者指示槽的数量及其宽度。在该示例中有五个槽，宽度分别为 14、6、3、14 和 9 个字符。条 | 将槽隔开。

注意

在该示例中，标题仅有两行。如果有更多行，可以用类似的格式指定。

第 18 行包含 END 以标记 TITLE_LAYOUT 参数的结束。

第 20 行包含 COLUMN LAYOUT，是指定显示表中数据列和行布局的参数。

第 21 行指定每个数据行的高度 20 像素和布局字符串 '||||'，后者指示列的数量及其宽度。在该示例中有五个列，宽度分别为 14、6、3、14 和 9 个字符。条 | 将列隔开。

第 23 行包含 END 以标记 COLUMN_LAYOUT 参数和 TABLE_LAYOUT 命令的结束。

注意

通过在 TABLE_LAYOUT 命令中组合参数 WIDTH、HEIGHT、Point 1 和 Point 2，可以指定显示表的尺寸。还可以在 TITLE_LAYOUT 中指定标题区域的宽度，在 COLUMN_LAYOUT 中指定列区域的宽度。如果 TABLE_LAYOUT 中指定的宽度表与 TITLE_LAYOUT 或 COLUMN_LAYOUT 中的宽度不同，将会优先使用 TABLE_LAYOUT 中的宽度。所以，显示表的宽度将始终为 TABLE_LAYOUT 中指定的宽度。

例如，如果 TABLE_LAYOUT 中的宽度是 18 列，TITLE_LAYOUT 或者 COLUMN_LAYOUT 中的宽度是 9 列，如下所示：

```
40 '   |   |   '
```

显示表的宽度将为 18 列，然后标题和列的宽度也将更改为 18 列，但是标题和列的比例将保持如下：

```
40 '      |      |      '
```

TABLE_TITLE

使用该命令可以设置现有显示表标题中槽的属性。

注意

此命令只能在未通过 SECURE_TABLE 命令进行防更改保护的显示表上运行。

命令格式如下：

```
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
```

下面给出了命令示例：

```
TABLE_TITLE 'table1'
  blue
  yellow
  'LINE' 'LINE'
  1 1

  green
  white
  '@s2' '@s1'
  BOX  1 2  2 3
END
```

注意

上例中，左侧数字不是格式的一部分。它们用于帮助您参考示例中的每一行。空行用于帮助您阅读示例和理解其结构。

第 1 行包含命令 TABLE_TITLE 和要定义标题的显示表的名称。

第 2 行指定槽中文本的显示颜色。本例中指定的是 blue。如果需要使用其他颜色，可按 "TABLE_LAYOUT" 第 3 行部分中的说明指定颜色。

第 3 行指定槽的背景颜色。本例中指定的是 yellow。如果需要使用其他颜色，可按 "TABLE_LAYOUT" 第 3 行部分中的说明指定颜色。

第 4 行指定槽中的显示文本 LINE 和相关联的操作文本 LINE。

显示文本是显示在显示表中的文本字符串。与其相关联的操作文本也是文本字符串，它包含有效的系统命令，当从屏幕的显示表中选择相应的显示文本时，它会发送至系统。

在这些参数中可以指定任何有效的文本字符串。还可以指定 @s# 格式的字符串，表示文本字符串来自与此显示表连接的逻辑表中编号为 # 的标题字符串。例如，如果在其中某一个参数中指定 @s3，表示连接逻辑表中的额外字符串 3 就是实际指定的文本字符串。

您甚至可以指定 UNITS=@s2 形式的文本字符串。如果连接逻辑表的额外字符串 2 是 Meters，则实际文本字符串是 UNITS=Meters。

如果指定 UNITS=@t2 形式的文本字符串，而且连接逻辑表的额外字符串 2 是 Meters，那么实际文本字符串是 UNITS='Meters' 。

第 5 行指定 1 1，即将第 1 行和第 1 列指定为要定义槽的位置。使用参数 BOX 也可以指定槽的框。例如，第 10 行中的 BOX 1 2 3 3 表示从第 1 行第 2 列至第 3 行第 3 列的槽形成所需的框。

第 7 至 10 行是与第 2 至 5 行类似的另一个参数块，用于定义显示表标题中的槽。

第 11 行指定 END 来指示 TABLE_TITLE 命令的结束。

注意

您可以根据需要包括任意数量的与第 2 至 5 行类似的参数块，只要这些参数块在命令 TABLE_TITLE 和 END 中即可。

实际上可将上例重新调整为：

```
TABLE_TITLE 'table1'
  blue yellow 'LINE'  'LINE' 1 1
  green white '@s2' '@s1' BOX  1 2  2 3
END
```

如果行中能容纳参数块，则此格式可能更易于阅读和比较。重点是参数必须至少用一个空格或制表符分隔。

用于处理显示表的命令包括：

- CHANGE_TABLE_SIZE
- CONNECT_TABLE
- DELETE_TABLE

- MOVE_TABLE
- PRINT_TABLE
- SAVE_TABLE
- SECURE_TABLE
- SHOW_TABLE
- TABLE_SCROLL_STEP

以下部分将对其进行详细介绍。

CHANGE_TABLE_SIZE

使用此命令可以更改现有显示表的尺寸，要求该表不受程序针对尺寸更改的内部保护。

命令格式如下：

```
CHANGE_TABLE_SIZE 'table name' Point 1 Point 2
```

下面给出了命令示例：

```
CHANGE_TABLE_SIZE  'disptable1'  100,100  1000,800
```

CHANGE_TABLE_SIZE 需要三个参数，分别是该示例中的 disptable1、100,100 和 1000,800。

disptable1 是要更改尺寸的显示表的名称。100,100 和 1000,800 分别是显示表左下角和右上角的 x,y 坐标。

注意

显示表的宽度与高度必须大于每个表的特定最小尺寸。

CONNECT_TABLE

使用该命令可以将显示表连接到逻辑表。

命令格式如下：

```
CONNECT_TABLE 'table name' 'Logical table name'
```

下面给出了命令示例：

```
CONNECT_TABLE  'disptable1'  'logtable1'
```

CONNECT_TABLE 需要两个参数，分别是该示例中的 disptable1 和 logtable1。

disptable1 是要与逻辑表 logtable1 连接的显示表的名称。

如果此命令尝试将已连接到某个逻辑表的显示表连接到另一个逻辑表，则先自动断开显示表与第一个逻辑表之间的现有连接，然后再建立与第二个逻辑表之间的新连接。这说明在任何时间显示表只能与一个逻辑表相连接。

DELETE_TABLE

使用该命令可以删除不受保护的现有显示表。

命令格式如下：

```
DELETE_TABLE
'table name' or (ALL CONFIRM)
```

下面给出了命令示例：

```
DELETE_TABLE
ALL  CONFIRM
```

或者简化为，

```
DELETE_TABLE  ALL  CONFIRM
```

DELETE_TABLE 需要一个参数，本示例中为 ALL CONFIRM。ALL 表示该命令删除系统中定义的所有显示表。CONFIRM 用于确认命令正确。

或者，也可按以下方式指定显示表名称：

```
DELETE_TABLE  'disptable1'
```

以指示将删除显示表 disptable1。

MOVE_TABLE

使用该命令可以将现有显示表移动至屏幕上的特定位置。该显示表不能受程序针对移动的内部保护。

命令格式如下：

```
MOVE_TABLE 'table name1'
(Point 1 Point 2) or
((UPPER or LOWER or RIGHT or LEFT) and/or (OF 'table name2'))
:
END
```

下面给出了命令示例：

```
MOVE_TABLE  'disptable1'
UPPER  OF  'disptable2'
100,100  150,400
END
```

注意

上例中，左侧数字不是格式的一部分。它们用于帮助您参考示例中的每一行。

第 1 行包含命令 MOVE_TABLE 和要移动的显示表的名称 disptable1。

第 2 行指定 UPPER 以指示移动至屏幕顶部。或者也可以指定其他方向，例如 LOWER、RIGHT 或 LEFT，来分别指示向屏幕的底部、右侧或者左侧移动。

或者可以在方向之后指定 OF 和显示表的名称 'disptable2'，以指示移动相对于另一个显示表。

第 3 行指定 100,100 和 150,400，它们是另一个移动的参考点和目标点的 x,y 坐标。

第 4 行指定 END 以指示 MOVE_TABLE 命令的结束。

注意

第 2 行和第 3 行都是有效的移动参数，只是指定的形式不同。换句话说，在该示例中有两个移动。

可以在 MOVE_TABLE 命令中指定多个移动，只要参数包含在 MOVE_TABLE 和 END 中即可。实际的移动将是所有移动组合的结果。

PRINT_TABLE

使用该命令可以将显示表的可视内容打印至屏幕上或者文件中。

命令格式如下：

```
PRINT_TABLE 'table name' or ALL
SCREEN or (( or DEL_OLD or APPEND) 'filename')
```

下面给出了命令示例：

```
PRINT_TABLE  'disptable1'
DEL_OLD  'file1.tbl'
```

或者简化为，

```
PRINT_TABLE  'disptable1'  DEL_OLD 'file1.tbl'
```

PRINT_TABLE 需要两个参数。第一个参数是要打印的显示表的名称。本例中，显示表为 disptable1。第二个参数指示指定显示表打印的输出目标。本例中，输出目标是具有 DEL_OLD 选项的 file1.tbl 文件，这意味着如果 file1.tbl 已存在，则会将其覆盖。

如果将某一文件指定为输出目标，那么也可以使用 APPEND 选项或根本不使用选项。APPEND 表示指定显示表会附加到指定文件的结尾 (如果文件存在)。如果文件不存在，则会创建指定文件。如果不指定任何选项，则只要不存在其他具有相同名称的文件，便会创建指定文件。如果指定文件已存在，则显示错误消息并中止命令。

或者，也可以通过指定 SCREEN 来替代上述示例中的 DEL_OLD 和 file1.tbl，将屏幕指定为输出目标。

SAVE_TABLE

使用该命令将显示表的设置保存至文件中。然后通过使用 INPUT 命令读取文件，可以重新创建显示表。

命令格式如下：

```
SAVE_TABLE 'table name' or ALL
SCREEN or (( or DEL_OLD or APPEND) 'filename')
```

下面给出了命令示例：

```
SAVE_TABLE  'disptable1'
DEL_OLD  'file1.tbl'
```

或者简化为，

```
SAVE_TABLE  'disptable1'  DEL_OLD  'file1.tbl'
```

SAVE_TABLE 需要两个参数。第一个参数是要保存的显示表的名称。本例中，显示表为 disptable1。第二个参数指示指定显示表保存的输出目标。本例中，输出目标是具有 DEL_OLD 选项的 file1.tbl 文件，这意味着如果 file1.tbl 已存在，则会将其覆盖。

如果将某一文件指定为输出目标，那么也可以使用 APPEND 选项或根本不使用选项。APPEND 表示指定显示表会附加到指定文件的结尾 (如果文件存在)。如果文件不存在，则会创建指定文件。如果不指定任何选项，则只要不存在其他具有相同名称的文件，便会创建指定文件。如果指定文件已存在，则显示错误消息并中止命令。

或者，也可以通过指定 SCREEN 来替代上述示例中的 DEL_OLD 和 file1.tbl，将屏幕指定为输出目标。

SECURE_TABLE

使用该命令可以保护显示表，以防止其删除或者重新定义。

命令格式如下：

```
SECURE_TABLE
'table name' or (ALL CONFIRM)
```

下面给出了命令示例：

```
SECURE_TABLE
ALL  CONFIRM
```

或者简化为，

```
SECURE_TABLE ALL CONFIRM
```

SECURE_TABLE 需要一个或者两个参数，本示例中为 ALL CONFIRM。ALL 表示该命令保护系统中定义的所有显示表。CONFIRM 用于确认命令正确。

或者，也可按以下方式指定显示表名称：

```
SECURE_TABLE  'disptable1'
```

以指示将保护显示表 disptable1。

SHOW_TABLE

使用该命令可以显示或者不显示单个显示表或者全部显示表。

命令格式如下：

```
SHOW_TABLE
ON or OFF
'table name' or ALL
```

下面给出了命令示例：

```
SHOW_TABLE
ON
ALL
```

或者简化为，

```
SHOW_TABLE  ON  ALL
```

SHOW_TABLE 需要两个参数，本例中为 ON 和 ALL。

ON 表示显示显示表。或者指定 OFF 以指示不显示。

ALL 表示在系统中定义的所有显示表中运行该命令。或者可以为命令指定显示表的名称。

注意

还可以分别使用 POP_UP_LTAB 和 POP_DOWN_LTAB 命令以显示或者不显示显示表。有关详细信息，请参阅 POP_UP_LTAB 和 POP_DOWN_LTAB 部分。

SHOW_TABLE ON (或者 OFF) 与 POP_UP_LTAB (或者 POP_DOWN_LTAB) 之间的差异如下所示：

- SHOW_TABLE 显示指定的显示表或者所有显示表，而 POP_UP_LTAB 显示与指定的逻辑表连接的所有显示表。
- POP_UP_LTAB 会保存其显示的显示表下屏幕区域的位图图像，而 SHOW_TABLE 不会。

TABLE_SCROLL_STEP

使用该命令可以设置使用滚动条两端的滚动框时显示表的滚动步长。

命令格式如下：

```
TABLE_SCROLL_STEP
'table name' or ALL
DEFAULT or number
```

下面给出了命令示例：

```
TABLE_SCROLL_STEP
ALL
DEFAULT
```

或者简化为，

```
TABLE_SCROLL_STEP  ALL  DEFAULT
```

TABLE_SCROLL_STEP 需要两个参数，本例中为 ALL 和 DEFAULT。

ALL 表示该命令更改系统中定义的所有显示表的滚动步长。或者可以指定显示表，例如 disptable1，以指示要更改滚动步长的特定表。

DEFAULT 表示滚动步长将设置为显示表中的数据行数。或者，可以指定特定的数据行数 (例如 5) 来指示滚动步长大小。

使用表函数

用户表是用户定义的逻辑表，与系统定义的逻辑表形成对照。它的数据结构与任意逻辑表相同。也可将显示表与用户表连接，方法与将显示表与任意逻辑表连接相同。

用于定义用户表的命令是：

- CREATE_LTAB

用于处理用户表的命令包括：

- COLOR_LTAB
- DELETE_LTAB
- DELETE_LTAB_ROW
- HIGHLIGHT_LTAB
- SECURE_LTAB
- SORT_LTAB
- WRITE_LTAB

以下部分详细介绍了上述每种命令。

注意

在以下各部分列出的每种命令格式中，正常字体文本表示它们是格式的一部分，因此必须按此处显示的内容进行指定。斜体文本表示必须将其替换为合适的字符串或值。单词 or 表示存在两个或多个选项，并可为参数选取其中一个选项。

COLOR_LTAB

使用该命令可以指定表中任意单元格的前景 (显示) 和背景颜色。

注意

命令 COLOR_LTAB 可在已使用 SECURE_LTAB 命令和 READ_ONLY 选项进行保护的用户表上运行。

命令格式如下：

```
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)
```

下面给出了命令示例：

```
COLOR_LTAB  'usertable1'
3  5
white
blue
```

COLOR_LTAB 需要四段数据。第一份是命令所需的用户表名称。第二个是要更改颜色的区域在指定用户表中的位置，第三个和第四个分别是单元格的前景和背景颜色。

在以上示例中，将为用户表 usertable1 中的行 3 和列 5 分配前景色 white 和背景色 blue。

可采用其他五种方式指定位置，例如：

- ROW 5，表示整个第 5 行。
- COLUMN 3，表示整个第 3 列。
- ALL，表示所有行和列。
- TITLE 2，表示第二个标题字符串。
- TITLE ALL，表示所有标题字符串。

也可为前景色和背景色指定任意有效的 10 个颜色名称。

另一命令示例为：

```
HIGHLIGHT_LTAB 'usertable1'
TITLE 2
black
yellow
```

表示将分配 black 作为标题字符串 2 的前景色，yellow 作为背景色。

CREATE_LTAB

使用该命令可以创建新用户表，或者指定新用户表的行和列中尺寸的估计值。

有关如何定义用户表的详细信息，请参阅“使用逻辑表和显示表 - 示例 1”中的“定义用户表”部分。

新用户表尺寸的可选参数有助于系统估计用户表的资源，从而使用户表的运行尽可能高效。

如果要创建的新用户表的名称与现有系统定义表的名称相同，而且该系统表受“只读”保护，那么将忽略此命令并显示错误消息。

如果要创建的新用户表的名称与现有用户表的名称相同，而且后者未受保护，那么该命令就会删除现有用户表中的所有数据，并根据命令中指定的行和列调整其尺寸。

但是如果要创建的新用户表的名称与现有用户表的名称相同，而且后者受 SECURE_LTAB 命令和 READ_ONLY 选项的保护，那么将显示错误消息。

命令格式如下：

```
CREATE_LTAB (or Rows) (or Columns)
'User table name'
```

下面给出了命令示例：

```
CREATE_LTAB
20 6
'usertable1'
```

CREATE_LTAB 需要两份数据。第一个是新用户表中行和列尺寸的估计值。在以上示例中，估计尺寸是 20 行和 6 列。该尺寸仅是估计值，并非限制。所以仍可以将数据写入表中在 20 行和 6 列之外的位置。第二个是新用户表的名称，即以上示例中的 usertable1。

DELETE_LTAB

使用该命令可以删除用户表。

如果指定的用户表受 SECURE_LTAB 命令的保护，或者由于当前在使用中而锁定，那么将显示错误消息。

命令格式如下：

```
DELETE_LTAB ALL or 'User table name'
```

下面给出了命令示例：

```
DELETE_LTAB  ALL
```

DELETE_LTAB 需要一个参数，本示例中为 ALL。ALL 表示该命令删除所有用户表。

或者，也可按以下方式指定用户表名称：

```
DELETE_LTAB  usertable1
```

以指示要删除的用户表 usertable1。

DELETE_LTAB_ROW

使用该命令可以删除用户表中的某一行。

如果指定的用户表受命令 SECURE_LTAB 和选项 READ_ONLY 的保护，则显示错误消息。

命令格式如下：

```
DELETE_LTAB_ROW 'User table name'
ALL or Row-number
```

下面给出了命令示例：

```
DELETE_LTAB_ROW  'usertable1'
ALL
```

DELETE_LTAB_ROW 需要两份数据。第一份是命令所需的用户表名称。第二个是要删除的行在指定用户表中的位置。

在以上示例中，将删除用户表 usertable1 中的 ALL 行。

或者，也可按以下方式指定行的位置：

```
DELETE_LTAB_ROW  'usertable1'
5
```

以指示将删除用户表 usertable1 中的行 5。

HIGHLIGHT_LTAB

使用该命令可以打开或者关闭用户表中数据区域位置的突出显示。

注意

命令 HIGHLIGHT_LTAB 可在已使用 SECURE_LTAB 命令和 READ_ONLY 选项进行保护的用户表上运行。

命令格式如下：

```
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
```

下面给出了命令示例：

```
HIGHLIGHT_LTAB  'usertable1'
3  5
ON
```

HIGHLIGHT_LTAB 需要三份数据。第一份是命令所需的用户表名称。第二个是要更改突出显示的区域在指定用户表中的位置，第三个是突出显示切换到的开/关状态。

在以上示例中，将突出显示用户表 usertable1 中的行 3 和列 5，即 ON。

可采用其他五种方式指定位置，例如：

- ROW 5，表示整个第 5 行。

- COLUMN 3，表示整个第 3 列。
- ALL，表示所有行和列。
- TITLE 2，表示第二个标题字符串。
- TITLE ALL，表示所有标题字符串。

也可指定 OFF 以关闭突出显示，或者指定 MARK 以使用内框标记框。

另一命令示例为：

```
HIGHLIGHT_LTAB 'usertable1'
TITLE 2
MARK
```

表示使用 MARK (内框) 突出显示标题字符串 2。

SECURE_LTAB

使用该命令可以保护指定的用户表以防止其删除，但是仍然可以更改其内容。如果想保护用户表以使其既不能删除也不能更改其内容，需要使用选项 READ_ONLY。

```
The format of the command is as follows:
SECURE_LTAB (or READ_ONLY) 'User table name'
```

下面给出了命令示例：

```
SECURE_LTAB  'usertable1'
```

SECURE_LTAB 需要一个参数来指定要保护的用户表。在该示例中，用户表为 usertable1。

也可按以下方式指定选项 READ_ONLY：

```
SECURE_LTAB  READ_ONLY  'usertable1'
```

这样就无法更改 usertable1 的内容。

SORT_LTAB

SORT_LTAB 根据指定的列排序用户表。REVERSE_SORT 反转指定的下一列的排序顺序。

命令格式如下：

```
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
```

下面给出了命令示例：

```
SORT_LTAB 'usertable1'
```

```
REVERSE_SORT 2
CONFIRM
```

SORT_LTAB 需要三份数据。第一个是函数所需的用户表名称。第二个是排序顺序，向前 (升序) 或者反向 (降序)。默认为向前排序。第三个是要排序数据的列的编号。

同一列中不同类型的排序顺序 (优先) 是：

1. 空值。
2. 数字。
3. 字符串。

在以上示例中，用户表 usertable1 中的数据将根据第 2 列中的数据进行逆序排序。CONFIRM 指示 SORT_LTAB 函数的结束。

也可以为该函数指定多个列，在这种情况中，最先指定的列在排序中具有最高的优先级。例如：

```
SORT_LTAB  'usertable1'
REVERSE_SORT 3
1
REVERSE_SORT 2
CONFIRM
```

逆序排序第 3 列的优先级要高于第 1 列或者第 2 列。

如果表具有写入保护，则会生成错误。

WRITE_LTAB

使用该命令可以将新值写入指定的用户表中。

命令格式如下：

```
WRITE_LTAB 'User table name'
(Row-number Column-number) or (TITLE Title-string-number)
Value
```

下面给出了命令示例：

```
WRITE_LTAB  'usertable1'
3  5
26.1
```

WRITE_LTAB 需要三份数据。

第一份是命令所需的用户表名称。第二个是指定用户表中要写入值的位置，第三个是值本身。

在以上示例中，值 26.1 将写入用户表 usertable1 中的位置行 3 和列 5。

可以使用另一种方式指定位置，例如表示标题字符串 2 的 `TITLE 2`。

可以指定“文本字符串”或者数字作为值，例如 `'WHEEL'` 与 `26.1` 均有效。也可指定变量名或者另一个命令作为值，例如 `RADIUS`，表示变量 `RADIUS` 的值就是指定的值，或者 `(READ_LTAB 'logtable1' 2 3)`，表示此命令的输出就是指定的值。

使用逻辑表和显示表 - 示例 1

要使用逻辑表和显示表，需要采取一些基本的步骤。

通常情况下，如果是用户表 (即用户定义的逻辑表)，就必须定义所需的逻辑表，而与其形成对照的系统定义逻辑表则无需定义。然后还必须定义所需的显示表，这样就可以将其映射至所需的逻辑表以查看此逻辑表中的数据。

在定义逻辑表和显示表后，可使用提供的命令和函数与其进行交互。

以下部分中的示例用于说明如何定义用户表和显示表，以及如何与其交互。

注意

有关以下部分中使用的命令和函数的详细信息，请参阅“逻辑表访问函数”、“显示表函数”以及“用户表函数”部分。

定义用户表

定义用户表主要有两步：

- 使用指定的名称以及行和列尺寸的估计值创建用户表。
- 使用所需数据填充用户表。

下面列出的是用于定义用户表的示例：

```
{
 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
```

注意

在上述以及后面的所列内容中，{ } 括号内的文本仅为注释，并不是宏程序的一部分。

可以使用文本编辑器创建上述内容并将其保存至文件 utable1.mac 中。

定义显示表

定义显示表主要有三步：

- 配置显示表的布局，包括标题和数据列区域
- 指定标题区域中的数据
- 指定数据列区域中的数据

下面列出的是用于定义显示表的示例：

```
{
 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
 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
```

可以使用文本编辑器创建上述内容。但是如果您已具有布局与所需布局类似的显示表，使用 SAVE_TABLE 命令将该显示表的上述内容保存至文件中会更加方便。然后使用文本编辑器将文件编辑为所需的布局。

创建完此列出内容后，会将其保存在文件 dtable1.mac 中。

与显示表交互

首先，必须在系统上运行 Creo Elements/Direct Drafting。然后，在可与用户表和显示表交互之前，必须通过以下方式加载其定义：用键盘输入以下命令来响应 Creo Elements/Direct Drafting 中的 ENTER COMMAND 提示：

```
INPUT 'utable1.mac' [Return]
UTABLE1 [Return]
INPUT 'dtable1.mac' [Return]
DTABLE1 [Return]
```

前两个命令加载用户表的定义，后两个命令加载显示表的定义。

现在可以开始与用户表和显示表交互。首先可以在屏幕上显示显示表 Mach_Op，通过：

```
SHOW_TABLE ON 'Mach_Op' [Return]
```

然后将显示表 Mach_Op 移动至屏幕中间的某处，通过：

```
MOVE_TABLE 'Mach_Op' LOWER LEFT 0,0 300,300 END [Return]
```

注意

使用命令中的选项 LOWER LEFT 非常重要，因为这可以确保移动从屏幕的左下角开始。否则，移动将从表的当前位置开始。这意味着表可能移动到屏幕之外。

然后，当显示表 Mach_Op 出现在屏幕上时，将其打印到文件 dtable1.prt 中，通过：

```
PRINT_TABLE 'Mach_Op' 'dtable1.prt' [Return]
```

接下来可以在打印机上打印该文件。

注意

如果要在显示表出现在屏幕上时保留该表的硬拷贝，该命令非常有用。

然后将显示表 Mach_Op 保存到文件 dtable1.sav 中，通过：

```
SAVE_TABLE 'Mach_Op' 'dtable1.sav' [Return]
```

注意

此命令在创建其他显示表的布局时非常有用。如果已存在布局与所需布局类似的显示表，可以使用此命令将该布局保存至文件中，然后通过文本编辑器进行编辑。

然后从屏幕中移除显示表 Mach_Op，通过：

```
SHOW_TABLE OFF 'Mach_Op' [Return]
```

使用逻辑表和显示表 - 示例 2

本部分与“使用逻辑表和显示表 - 示例 1”相似，区别是这里的示例更加复杂。本示例使用两个用户表和两个显示表。第一个用户表和显示表来自于“使用逻辑表和显示表 - 示例 1”中的示例。第二个用户表和显示表是新的，将在本部分中定义。

以下部分中的示例用于说明如何定义所需的用户表和显示表，以及如何与其交互。

注意

有关以下部分中使用的命令和函数的详细信息，请参阅“逻辑表访问函数”、“显示表函数”以及“用户表及其函数”部分。

定义第一个用户表和显示表

第一个用户表和显示表与“使用逻辑表和显示表 - 示例 1”中的表基本相同，仅有少许的更改，所以可以将文件 utable1.mac 和 dtable1.mac 分别复制到 utable21.mac 和 dtable21.mac。然后可以使用文本编辑器将其编辑为所需的布局。

以下列出内容为 utable21.mac 文件中的第一张用户表：

```
{
 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
```

注意

在上述以及后面的所列内容中，{ } 括号内的文本仅为注释，并不是宏程序的一部分。

除了用户表的名称为 Mach_Op2 之外，此宏 UTABLE21 与宏 UTABLE1 完全相同。

以下列出内容为 dtable21.mac 文件中的第一张用户表：

```
{
 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
```

除以下内容外，此宏 DTABLE21 与 DTABLE1 相同：

- 该显示表的名称为 Mach_Op2，映射到名称为 Mach_Op2 的用户表
- 在第一个标题行中有两列
- 该标题行中的第二列包含文本 END，相关联的操作文本是：
  `SHOW_TABLE OFF 'Mach_Op2' SHOW_TABLE OFF 'Op_Para' END`

  表示从屏幕中移除显示表 Mach_Op2 和 Op_Para，并中止当前的宏程序。

定义第二个用户表和显示表

第二个用户表和显示表是新的，所以必须进行定义。下述内容用于定义第二个显示表：

```
{
 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
```

除以下内容外，此宏 DTABLE2 与 DTABLE21 相同：

- 该显示表的名称为 Op_Para，在定义第二个用户表之前不会映射到用户表。
- 第一个标题行中的第二列包含文本 OFF，相关联的操作文本是：

  ```
  SHOW_TABLE OFF 'Op_Para'
  ```

  表示从屏幕中移除显示表 Op_Para。

创建完此列出内容后，会将其保存在文件 dtable2.mac 中。

下述内容用于定义第二个用户表，包含与所有四个表交互的宏命令：

```
  {
   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
```

创建完此列出内容后，会将其保存在文件 utable2.mac 中。

上述内容不仅在定义第二个用户表时不可或缺，而且还提供了与所有四个表交互的宏命令。

以下是对于宏程序的解释：

- DEFINE UTABLE2 标记此宏 UTABLE2 的开始。
- LOCAL OPARA 表示局部变量 OPARA。
- MOVE_TABLE 'Mach_Op2' LOWER LEFT 0,0 300,300 END 将显示表 Mach_Op2 移动至屏幕上的 300,300 位置。
- POP_UP_LTAB 'Mach_Op2' 显示用户表 Mach_Op2。
- CREATE_LTAB 6 2 'Op_Para' 定义尺寸为六行两列的用户表 Op_Para。
- CONNECT_LTAB 'Op_Para' 'Op_Para' 将显示表 Op_Para 连接至用户表 Op_Para。
- LOOP 标记循环开始，END_LOOP 标记其终点。
- 在此循环中，首先请求用户从表中选择一个选项，通过：

  ```
  READ STRING 'Select option from CHARACTERISTICS OF A MACHINING
  OPERATION table:' OPARA
  ```

  用户可以从指定表中选择一个选项，相关联的操作文本作为变量 OPARA 的输入。
- LET OPARA (UPC OPARA) 将输入文本转换为大写。

- 然后 IF、ELSE_IF 和 END_IF 语句检查变量 OPARA 的内容，该变量的值可以是五个可能值中的一个。根据其值，将相应的数据写入用户表 Op_Para。
- 您会注意到两个 DELETE_LTAB_ROW 语句，需要使用它们删除用户表 Op_Para 的第 5 行和第 6 行中上一个选择遗留下来的数据。否则，该数据也会显示在显示表 Op_Para 中。
- 如果 OPARA 不包含任何可能值，将显示消息 UNKNOWN OPTION。
- END_IF 标记 IF 语句的结束。
- SHOW_TABLE ON 'Op_Para' 显示显示表 Op_Para。
- END_LOOP 标记 LOOP 语句的结束。
- END_DEFINE 标记 DEFINE 语句的结束。

与用户表和显示表交互

首先，必须在系统上运行 Creo Elements/Direct Drafting。然后，在可与用户表和显示表交互之前，必须通过以下方式加载其定义：用键盘输入以下命令来响应 Creo Elements/Direct Drafting 中的 ENTER COMMAND 提示：

```
INPUT 'utable21.mac' [Return]
UTABLE21 [Return]
INPUT 'dtable21.mac' [Return]
DTABLE21 [Return]
INPUT 'dtable2.mac' [Return]
DTABLE2 [Return]
INPUT 'utable2.mac' [Return]
```

前四个命令加载第一个用户表和显示表的定义，后三个命令加载第二个显示表和用户表的定义。

现在可以使用该命令运行用户表和显示表：

```
UTABLE2 [Return]
```

宏程序 UTABLE2 首先显示 Characteristics of a Machining Operation 表和提示 Select option from CHARACTERISTICS OF A MACHINING OPERATION table:。可以选择该表中五个选项的任意一个，然后显示第二个表以显示选定选项下的选项。

现在屏幕上有两个显示表，宏程序环回以提示再次输入。可以尝试所有五个选项并相应查看第二个显示表如何更改其内容。还可以尝试空选项，显示消息 UNKNOWN OPTION。

您会注意到在第二个显示表的右上角处有文本 OFF。如果选择该文本，将从屏幕中移除第二个显示表。但是宏程序仍在运行，并提示再次输入。如果从第一个显示表中选择了一个选项，那么第二个显示表就会再次出现，并含有选定选项的相应内容。

您还会注意到在第一个显示表的右上角处有文本 END。如果选择该文本，第一个和第二个显示表都将从屏幕中移除，而且宏程序结束。

备注

该示例用于说明可以对逻辑表和显示表进行哪些操作。您可以进一步考虑开发更加复杂的显示表和逻辑表，以及使用两级以上的表。

索引

A

ABS 函数, 86
alpha 端子, 13
ANG 函数, 84

B

Break 键, 18
  不保存并离开编辑器, 15

C

C 程序设计语言, 38
CHANGE_TABLE_SIZE 命令, 198
CLOSE_FILE 函数, 73
COLOR_LTAB 命令, 204
CONNECT_TABLE 命令, 198
CREATE_LTAB, 93
CREATE_LTAB 命令, 205

D

Delete 键, 20
DELETE_LTAB 命令, 206
DELETE_LTAB_ROW 命令, 206
DELETE_MACRO, 16
DELETE_TABLE 命令, 199
DISPLAY_NO_WAIT 函数, 33

E

ECHO 函数, 95-96
EDIT_FILE 命令, 15
EDIT_PORT 命令, 20
End 键, 20
END 命令, 64
  和递归循环, 44
END_DEFINE 函数, 31
ESC 键，不保存并离开编辑器, 15

F

Fortran 编程语言, 79

G

GETENV 函数, 54

H

HELP_PORT 命令, 20
HIGHLIGHT_LTAB 命令, 207
HL_INQ_Z_VALUE, 93
HL_INQ_Z_VALUE 函数, 54
Home 键, 20

I

IF ... ELSE_IF ... ELSE ... END_IF 构造, 41
INPUT 命令, 15-16, 31
INQ 表达式, 52
INQ_ELEM 函数, 53
INQ_ENV 函数, 52
Insert 键, 20

L

LEN 函数, 73
LET 函数, 33
  括号括起, 43
LOCAL 伪命令, 61
LOOP ... EXIT_IF ... END_LOOP 构造, 40
LOOP 伪命令, 64
LTAB_COLUMNS 命令, 182
LTAB_ROWS 命令, 183
LTAB_TITLES 命令, 183

M

MOVE_TABLE 命令, 199

O

OPEN_INFILE 函数, 71
OPEN_OUTFILE 函数, 73

P

Pascal 程序设计语言, 38
PL/I 编程语言, 79
PNT_RA 函数, 84
PNT_XY 运算符, 56
POP_DOWN_LTAB 命令, 184
POP_UP_LTAB 命令, 185
POS 函数, 73
PRINT_TABLE 命令, 200

R

READ 函数, 38, 61, 64
READ_FILE 函数, 73
READ_LTAB 命令, 186
REPEAT ... UNTIL 构造, 41

S

SAVE_LTAB 命令, 186
SAVE_TABLE 命令, 201
SCROLL_LTAB 命令, 187
SECURE_LTAB 命令, 208
SECURE_TABLE 命令, 202
SELECT_FROM_LTAB 命令, 187
SHOW_TABLE 命令, 202
SORT_LTAB 命令, 208
STORE 命令, 31
STR 函数, 33
SUBSTR 函数, 73

T

TABLE_COLUMN
  格式, 190
TABLE_COLUMN 命令, 190
TABLE_LAYOUT
  格式, 192
TABLE_LAYOUT 命令, 192
TABLE_SCROLL_STEP 命令, 203
TABLE_TITLE
  格式, 195
TABLE_TITLE 命令, 195
TECHO 函数, 95

V

VAL 函数, 86

W

WAIT 函数, 33
WHILE ... END_WHILE 构造, 40
WRITE_FILE 函数, 73
WRITE_LTAB, 93
WRITE_LTAB 命令, 209

X

X_OF 运算符, 56

Y

Y_OF 运算符, 56

Z

Z 级，查看, 93

丶

主体，宏, 30

人

保存宏，Ctrl-D, 15
保存绘图, 14
假，布尔表达式, 42
令牌，括号括起, 43
使用逻辑表和显示表
  示例 1, 210
使用显示表
  命令, 195
使用用户表
  命令, 203
停止宏, 18

入

全局变量, 33, 77

八

关键字
  大写, 29
  示例, 70

凵

函数
  访问逻辑表, 182
  显示表, 189
  用户表, 203
函数，PL/I 中，Fortran, 79

刀

分割线，宏, 90
剖面线图案, 124

力

加载
  宏, 16

十

单引号
  对于字符串, 29

厶

参数, 30
  传递到宏, 79

又

变量
  局部, 30, 33, 37
  类型, 38
  名称，冲突, 37
  全局, 33, 77
变量名称
  防御性程序设计, 48

口

命令
  编辑器, 23
  定义显示表, 189
  定义用户表, 203
  使用显示表, 195
  使用用户表, 203
  文字处理, 19
  CHANGE_TABLE_SIZE, 198
  COLOR_LTAB, 204
  CONNECT_TABLE, 198
  CREATE_LTAB, 205
  DELETE_LTAB, 206
  DELETE_LTAB_ROW, 206
  DELETE_TABLE, 199
  HIGHLIGHT_LTAB, 207
  LTAB_COLUMNS, 182
  LTAB_ROWS, 183
  LTAB_TITLES, 183
  MOVE_TABLE, 199
  POP_DOWN_LTAB, 184
  POP_UP_LTAB, 185
  PRINT_TABLE, 200
  READ_LTAB, 186
  SAVE_LTAB, 186
  SAVE_TABLE, 201
  SCROLL_LTAB, 187
  SECURE_LTAB, 208
  SECURE_TABLE, 202
  SELECT_FROM_LTAB, 187
  SHOW_TABLE, 202
  SORT_LTAB, 208
  TABLE_COLUMN, 190
  TABLE_LAYOUT, 192
  TABLE_SCROLL_STEP, 203
  TABLE_TITLE, 195
  WRITE_LTAB, 209
只读
  SECURE_LTAB, 208

囗

图，语法, 31

土

坐标, 56

夂

备注
  不能嵌套, 34
  宏内, 33

夊

复制文本, 22

大

大写, 29

子

存储宏, 15
存储宏，Ctrl-D, 15
字符串连接, 33

宀

定义，宏, 29
定义显示表
  命令, 189
定义用户表
  命令, 203
宏

备注位于, 33
编译, 16
槽, 91
存储, 15
单独的文件用于, 15
的应用, 14
调试, 16
定义, 29
多边形, 92
几何, 60, 64
加载, 16
扩展, 77
内嵌代码的替换, 77
嵌套, 33, 77
使用命令位于, 49
缩进行, 48
它为何物？, 14
停止, 18
隐藏的行，查看, 93
运行, 16
主体, 30
组成部分, 29
宏示例
箭头, 61

寸

将显示表连接到逻辑表
概念, 181

小

小
写, 29
小写, 29

尸

局部变量, 30, 33, 37

山

嵌套备注, 34
嵌套宏, 33

巾

布尔表达式
假, 42
括号括起, 42
真, 42

彐

当前环境, 51
当前行, 23

彳

循环
递归，在语法图中, 44

手

换行字符, 70
控制语句, 39
括号
带有布尔表达式, 49
使用, 42
描述符，文件, 71, 73
执行顺序, 39
指针，文件, 73

攴

数据文件，读取, 82

文

文件
  存储宏, 15
  打开, 71, 73
  描述符, 71, 73
  写入, 73
  用于存储宏, 15
  指针, 73
文件结尾标记, 73
文件名，与宏同名, 15

日

显示表, 180, 189
  定义 (示例 1), 211
  函数, 189
  简介, 179
  交互 (示例 1), 212
  组件, 180

木

标记
  设置, 21
  系统定义的, 21
标题字符串, 179
概念
  将显示表连接到逻辑表, 181
格式
  TABLE_COLUMN, 190
  TABLE_LAYOUT, 192
  TABLE_TITLE, 195
构造线，宏, 90

火

点，定义, 56

玉

环境，当前, 51

用

用户表, 179
  定义 (示例 1), 210
  函数, 203
用户输入, 30
用于创建宏的 ECHO, 96

目

真，布尔表达式, 42

矢

矢量, 56
  相加, 57
  相减, 58
矢量，针对栓, 84
矢量相加, 57
矢量相减, 58

示

示例 1
  定义显示表, 211
  定义用户表, 210
  使用逻辑表和显示表, 210
  与显示表交互, 212
示例 2
  定义第二个用户表和显示表, 215
  定义第一个用户表和显示表, 213
  使用逻辑表和显示表, 213
  与用户和显示表交互, 218

竹

简单的代码
　防御性程序设计, 48
箭头键, 20
筛选错误的输入, 49

糸

系统阵列, 52

纟

编辑键, 20
编辑器, 19
　标记, 23
　不保存并离开, 15
　关键字, 23
　用于编写宏, 19
　字符串, 23
编辑器命令, 19, 23
　调整对齐, 23
　调整填充, 23
　调整中心, 23
　复制, 24
　覆盖, 24
　加载, 24
　删除, 24
　设置标记, 25
　设置右边距, 26
　设置转义, 25
　设置左边距, 25
　替换, 25
　下一个, 24
　写入, 26
　移动, 24
编译
　宏, 16
绘图，保存, 14
缩进, 48
　防御性程序设计, 48
线类型, 64

衣

表达式
　括号括起, 50
　内置, 50

讠

调试宏, 16
记录输入, 95
设置标记, 21
设置标记命令, 21
语法图, 31

车

输入
　用户，对于宏, 30

辶

递归循环，在语法图中, 44
连接字符串, 33
逻辑表, 179
　访问函数, 182
　简介, 179
　组件, 179
逻辑表和显示表, 178
　使用 (示例 1), 210
　使用 (示例 2), 213
运行宏, 16
追踪
　程序, 44
　文件, 45

钅

键，编辑, 20

阝

防御性程序设计, 48
隐藏的行，查看, 93

页

颜色, 64

高

高效的代码, 49