目 录

第一章 Visual FORTRAN 初步

本章介绍 Visual FORTRAN 的一些基本知识。包括对 FORTRAN 语言本身发展简史和新版本的标准特性的分析和它与其他一些工具的比较，Visual FORTRAN 简介和 5.0 版本的安装，Microsoft Developer Studio 开发环境的介绍，如何有效使用在线帮助，最后介绍在 Developer Studio 中对互联网的访问。

1.1 重新认识 FORTRAN

随着 FORTRAN 语言标准的沿革，现在的 FORTRAN 90 标准与原来的各个标准有了很大的区别：现在的 FORTRAN 已经是一种功能强大、尤其适合进行科学计算的现代语言，我们对古老的 FORTRAN 语言应该有全新的认识。

1.1.1 FORTRAN 语言的发展简介

一套特定的编译计算机指令的规则称为编程语言。FORTRAN 是世界上最早出现的高级编程语言。FORTRAN 是 FORmula TRANslate（公式翻译）的缩写，所以 FORTRAN 的主要用途是做科学计算。FORTRAN 语言的思想首先由 John Backus 于 1953 年在纽约提出，称为 FORTRAN I。第一个 FORTRAN 程序运行于 1957 年 4 月。随后，FORTRAN 的应用迅速广泛传播，形成了许多不同的版本。较为重要和流行的是 1958 年提出的 FORTRAN II，它较 FORTRAN I 增加了大量重要的扩充，例如引进子函数等概念。从 1958 年至 1963 年期间，FORTRAN 语言在许多计算机上得以实现，期间又出现了较为流行的 FORTRAN IV。但是 FORTRAN II 和 FORTRAN IV并不兼容，从而使 FORTRAN 语言的标准化工作提上日程。

1962 年 5 月，美国国家标准学会（American National Standard Institute，简称 ANSI）成立了相关机构来进行 FORTRAN 语言标准化的工作。1966 年 ANSI 正式公布了两个美国标准文本：

- 美国国家标准 FORTRAN（ANSI X3.9-1966），相当于 FORTRAN IV
- 美国国家标准基本 FORTRAN（ANSI X3.10-1966），相当于 FORTRAN II

其中美国国家标准基本 FORTRAN 是美国国家标准 FORTRAN 的一个子集，从而实现了语言的向下兼容。1972 年国际标准化组织（International Standard Organization，简称 ISO）在 FORTRAN 66 的基础上公布了 ISO FORTRAN 标准（ISO R1539）。它描述了三种级别：

- 基本级，接近于 ANSI X3.10-1966

- 中间级，介于基本级和完全级之间
- 完全级，接近于 ANSI X3.9-1966

美国国家标准 FORTRAN（通常称为 FORTRAN 66）建立之后受到了国际上的广泛接受，统治了几乎所有的数值计算领域。但是随着计算机软硬件技术的迅猛发展，尤其是在结构化程序设计（Structure Programming，简称 SP）方法提出以后，FORTRAN 66 日益显得不能满足需要。这是因为，FORTRAN 66 不是结构化的语言，尤其以 GOTO 语句为标志，没有很好地实现顺序、分支和循环这三种基本结构的语句。因此许多计算机厂商对 FORTRAN 66 进行了不同程度的扩充。针对这种情况，美国国家标准学会（ANSI）于 1976 年对 1966 年公布的 ANSI X3.9-1966 进行了修订，基本上把各厂商的有效的功能都吸收进来，并且还增加了许多新的内容。1978 年 4 月美国国家标准学会正式公布了新的美国国家标准 ANSI X3.9-1978《程序设计语言 FORTRAN》，这就是通常称为 FORTRAN 77 的 FORTRAN 语言标准。FORTRAN 77 向下兼容 FORTRAN 66。1980 年，FORTRAN 77 被 ISO 正式采纳成为国际标准 ISO 1539-1980，该标准同样分为全集和子集。

FORTRAN 77 公布之后在国际上得到了广泛应用，多数计算机系统都配备了 FORTRAN 77 编译程序，国内外关于 FORTRAN 77 的教材和参考资料也相当丰富。但是，FORTRAN 77 还不是完全结构化的语言。国际上为了推动 FORTRAN 语言的结构化和现代化作了不懈的努力。经过长时间酝酿，1991 年 5 月通过了研制期间称为 FORTRAN 8x 的 FORTRAN 90，美国编号为 ANSI X3.198-1991，ISO 编号为 ISO/IEC 1539:1991。新标准中增加了许多新的特性和功能，其中值得一提的是由我国计算机和信息处理标准化技术委员会程序设计分会提出的多字节字符集数据类型及相应的内部函数。这一数据类型的增加为非英语国家在使用计算机方面提供了极大的方便。之后不久又出现了 FORTRAN 95，并且更新的标准又在积极准备之中。本书应用的主要是 FORTRAN 90 标准。

由此可以看出，FORTRAN 语言虽然历史最为悠久但仍在不停地改进和发展。这也是 FORTRAN 语言保持旺盛的生命力的原因之一。

1.1.2 FORTRAN 90 语言标准的新特性

FORTRAN 90 采用仅第一个字母是大写，而 FORTRAN 77 及 FORTRAN 66 等语言的名称中只使用大写字母。FORTRAN 90 没有沿用这一传统的书写方式。

FORTRAN 90 对 FORTRAN 77 做了较大的扩充和完善，显著的扩充主要有如下七个方面：

（1）引入数组运算；
（2）提高了数值计算的功能；
（3）内在数据类型的参数化；
（4）用户定义的数据类型；
（5）用户定义的运算与赋值；
（6）引入模块数据及过程定义的功能；
（7）引入指针概念。

另外，还包括了其它一些扩充。例如，改进了源程序的书写形式、引入了更多的控

制构造和递归过程、新的输入/输出功能及动态可分配数组等。

FORTRAN 90 的先进性，体现在以下几个方面：

- 增加了许多具有现代特点的项目和语句，用新的控制结构实现选择分叉与重复操作，保证了程序的结构化。
- 增加了结构块、模块及过程调用的灵活性，使源程序易读易维护。
- 吸收了 C、PASCAL 语言的长处，淘汰或拟淘汰 FORTRAN 77 中过时的语句，具有现代语言的特色。
- 在 FORTRAN 77 数值计算的基础上，进一步发展了数值计算的优势。新增了许多先进的调用手段，扩展了 FORTRAN 77 的操作功能。
- 增加了多字节字符集的数据类型及相应的内在函数，允许在字符数据中选取不同种别，在源程序字符串中可以使用各国文字和各种专用符号，对非英语国家使用计算机提供了更大的支持。

FORTRAN 77 程序仍能在 FORTRAN 90 编译系统下运行，即具有对 FORTRAN 77 的向上兼容性。

1.1.3 FORTRAN 语言与其它语言的比较

前面提到，从名称就可以看出 FORTRAN 语言是一种用途非常专一的语言。而随着其它高级语言和软件的发展，许多都能够用来作科学计算，其中较为典型的是 C/C++语言和 Matlab 语言。

C 语言的诞生背景是人们希望能够找到一种集高级语言特性（可读性和可移植性好）和低级语言特性（可以直接对硬件操作）于一身的语言。C 语言是由贝尔实验室的 D.M.Ritchie 于 1972 至 1973 年间在 B 语言的基础上设计出来的。随后 C 语言又作了多次改进，但主要还是在贝尔实验室内部使用，直到 1975 年 C 语言才引起人们的普遍注意。C 语言简洁、紧凑，使用方便灵活，书写格式自由，语法限制不严格，运算符和数据类型丰富，同时还是结构化的理想语言。C 语言成功地实现了它的设计初衷，从而在系统设计及应用程序设计方面得到广泛应用。

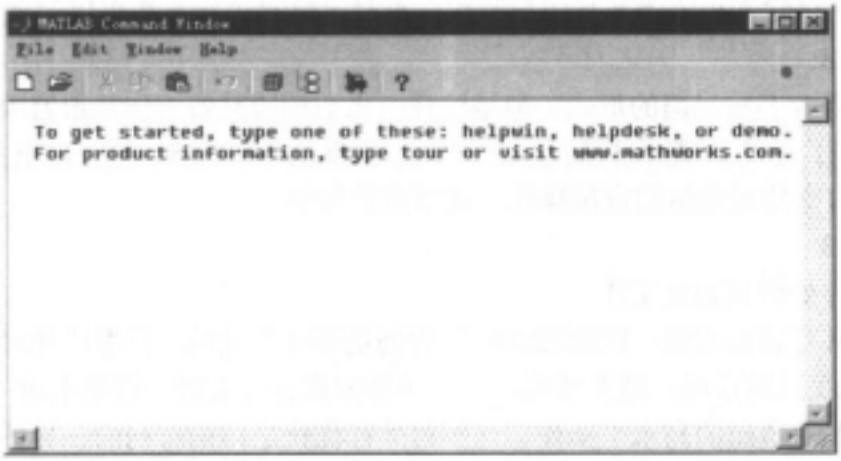


图 1.1 Matlab 5.2 的界面

Matlab 是“Matrix Laboratory”（矩阵实验室）的缩写。开始是由 Cleve Moler 于 1967 年用 FORTRAN 编写的，可以方便地调用 LINPACK EISPACK 公司所设计开发的矩阵分析软件的各种过程。现在 Windows 下的 Matlab 是 Mathworks 公司用 C 语言编写的。它是一个高度集成的系统，由于操作（尤其是对矩阵）方便，容易掌握，所以很受美国大学生的欢迎。近年我国对 Matlab 的研究也很活跃。现在 Matlab 的最高版本是 5.x 版。Matlab 5.2 的工作界面如图 1.1 所示。

与 C/C++语言和 Matlab 语言相比，FORTRAN 语言在科学计算方面有得天独厚的优越性。

首先，FORTRAN 语言悠久的历史是一笔宝贵的财富。在很长一段时间里，FORTRAN 语言是科技工作者唯一的选择。从 FORTRAN 语言诞生到目前近 40 年的历史过程中，在科学技术研究的各个领域积累了大量的 FORTRAN 语言程序，其中很多是经过长期实践检验证明是正确、可靠的，如果用其它语言重新编写其正确性和可靠性都还要再由实践或时间来验证。同时，FORTRAN 语言标准的历次修订都尽量保持向下兼容，这使得前人编写的 FORTRAN 程序可以不改动或只作很小的改动就可以供现在使用，这样就尽量避免了重复的脑力劳动。相比之下，C/C++和 Matlab 比 FORTRAN 晚出现 20 年，它们所能利用和借鉴的材料是无法和 FORTRAN 语言相比的。

其次，FORTRAN 语言书写和语法要求严格，更适合严谨的科学计算领域。例如，C 语言中的数组并不提供越界检查（实际上 C 语言中数组名和指针几乎是同一个概念），这使数组的应用更加灵活。但是在科学计算方面数组的这种“散漫”的用法是相当危险的，如果允许访问到错误的内存地址，其计算的结果是不可预测的，将会在科学和工程应用上造成不可估量的损失。

第三，FORTRAN 语言可以直接对数组和复数进行运算。矩阵（包括向量）是科学计算最重要的单元，科学计算中需要对矩阵进行大量的运算。在计算机中矩阵是以数组的形式存储的。在 C 语言中要完成两个矩阵相加（如矩阵 A 加矩阵 B）要通过两个循环语句来实现，如果编成子函数调用形式也是类似

```
add_matrix (A , B , Result , matrix_line , matrix_column );
```

的有 5 个形式（相加的矩阵 A、B、存放结果的矩阵 C 及其行数和列数）繁琐的形式。在 C++中，可以通过定义矩阵类及其成员函数，再对运算符进行重载可以实现

```
Result=A+B;
```

这样的简洁、符合书写习惯的形式。但是这样一来前期的准备工作就很繁琐，并且对矩阵的初始化变成了对矩阵类的初始化，工作量不减反增。复数的情况也是类似的。而 FORTRAN 语言支持对数组的直接操作，可以直接使用

```
Result=A+B
```

的形式而不需作如何初始化工作。

第四，在并行计算方面，FORTRAN 语言仍是不可替代的。巨型计算机的实现，包括 IBM 的深蓝，我国的银河、曙光系列，无一不是依靠并行处理。近年来对 FORTRAN 的扩展中有很多是对数组的高水平操作，这些对并行优化是特别有利的。另外还有对共用存储器并行系统的扩展(PCF)和对串并行系统的数据并行扩展(HPF)。它们都使得 FORTRAN 在并行计算领域独领风骚。

第五，FORTRAN 语言是一种编译语言，运行速度快。Matlab 语言之所以能够流行，很大程度上得益于它对矩阵运算的简化。但它是类似于 Basic 语言的一种解释语言，这使得 Matlab 语言的循环效率非常低。所以在 Matlab 中如果要大量使用循环就不得不调用 C/C++和 FORTRAN 程序。科学计算中普遍存在大量的循环，这使得 Matlab 语言的应用受到很大制约，而 FORTRAN 语言则显得得心应手。

最后，FORTRAN 语言自身仍在不断完善和发展，功能不断增强。作为历史最长的高级语言，FORTRAN 语言自身有很多早期历史的痕迹，如书写格式近乎刻板、不是结构化语言、输入输出简陋、命名的隐式规则等。尽管其中很多是受当时硬件条件制约的结果，但是如果不加以改进仍会被历史淘汰。FORTRAN 语言经过几次标准修订淘汰了部分过时的语言和特性（尽管还保持兼容）并增加了符合需要的现代特性，顺应了时代的潮流。

从这些比较中可以看出，FORTRAN 语言具有 C/C++语言编译语言的特点又具有 Matlab 语言对矩阵操作简洁直接的特点，并且本身语法严谨，底蕴丰富，包含了现代语言的特征，是科学计算的首选语言。

1.2 Visual FORTRAN 简介和安装

本节介绍 Visual FORTRAN 的由来，版本特性和安装所需的条件以及步骤。

1.2.1 Visual FORTRAN 简介

Visual FORTRAN的前身是微软的Microsoft FORTRAN PowerStation 4.0。成立于1975年的微软（Microsoft）公司是世界范围内的个人计算机软件业的领导厂商。Microsoft的产品线很宽，如Microsoft DOS、Windows、Windows NT操作系统，Microsoft Office系列，Microsoft Visual Studio系列等。其完整全面的产品线使FORTRAN PowerStation 4.0能以Microsoft Developer Studio为集成开发环境。FORTRAN PowerStation 4.0的功能十分强大，它的主要特性有：

- 全面支持 FORTRAN 90 语言标准；
- 对 FORTRAN 语言的丰富扩展；
- 丰富的在线文档；
- 完全支持与 Microsoft Visual C++的混合语言编程；
- 与 Microsoft Visual Basic 和 Office 协同工作；
- 支持 Windows 编程（QuickWin）。

为了将生产开发工具的主要精力集中在系统开发方面，1997 年 3 月，微软和 DEC（Digital Equipment Corp，即数据设备公司）达成协议，微软不再销售和开发 PowerStation 4.0，并且授权 DEC 提供其后继版本 Digital Visual FORTRAN 5.0（也可称为 Visual FORTRAN 5.0），微软推荐 PowerStation 4.0 的用户升级到 Visual FORTRAN 5.0。

DEC的高质量FORTRAN编译器在全世界享有盛誉，并在高性能科学及工程计算方面

拥有世界领先的技术。DEC和微软的这次合作使得两家公司优势互补，其第一个产品Digital Visual FORTRAN 5.0成为继FORTRAN PowerStation 4.0之后又一个功能强大的FORTRAN编译器。

1998 年 1 月 26 日，Digital 公司和 Compaq 公司同意进行当时计算机工业史上最大的一次合并。根据协议条款，Digital 公司将成为 Compaq 公司的全资子公司，创建一个年收入超过 370 亿美元的公司。于是 Digital Visual FORTRAN 更名为 Compaq Visual FORTRAN。现在，Visual FORTRAN 的最新版本是 Compaq Visual FORTRAN 6.1。本书主要介绍较常见的 Digital Visual FORTRAN 5.0。

Digital Visual FORTRAN 5.0 分为两种版本：

- Digital Visual FORTRAN 5.0 标准版

  标准版包括在 Intel CPU 和 Windows 9x、Windows NT 上运行的 Digital Visual FORTRAN （DVF）编译器，DVF 程序库和 Developer Studio 。

- Digital Visual FORTRAN 5.0 专业版

  专业版除了包含标准版的所有内容以外还包括在 Digital Alpha CPU 和 Windows NT 上运行的 Digital Visual FORTRAN （DVF）编译器，DVF 程序库和 Developer Studio。另外还包括在 Intel 和 Alpha CPU 上都可使用的 IMSL 数学和统计程序库。

其中的 FORTRAN 90 优化编译器与在 Digital UNIX 和 OpenVMS 上运行的 Digital FORTRAN 编译器是由同一产品派生出来的。除了包含了一般的 Digital FORTRAN 语言扩展之外，FORTRAN 90 编译器还与 Microsoft FORTRAN PowerStation 兼容，并且支持 FORTRAN 95 的语言特性。

1.2.2 Visual FORTRAN 5.0 的特性

- 全面支持 FORTRAN 90 语言标准，支持大部分其他平台上供应商提供的 FORTRAN 语言扩展，还包含了 FORTRAN 95 的标准特性。
- 使用 Microsoft Developer Studio 集成开发环境，用户可以应用 Windows 的丰富特性使开发速度更快、效率更高。
- 支持 COM（Component Object Model，即组件对象模型）和自动对象（模块向导）。
- 专业版包含了分析和处理科学和商业应用中统计数字的 IMSL 数值库。.
- 支持在命令行窗口中使用命令行界面，并且用户可以定制习惯的命令行工作环境。
- 提供和运行在 Digital UNIX 和 OpenVMS Alpha 系统上的 Digital FORTRAN 以及 Microsoft FORTRAN Powerstation 4.0 兼容的语言扩展。
- 完整详尽的在线帮助系统。
- 完全支持 Visual FORTRAN 和 Visual C++, Visual J++, Visual Basic, 和 Microsoft MASM （x86 系统的汇编器）语言之间的混合语言编程。
- 与 Microsoft Visual Basic 和 Office 协同工作，例如可以用 Visual Basic 创建图形用户界面程序，用 Visual FORTRAN 从已经存在的 FORTRAN 源代码创建的动

态连接库（DLL）作为后台数值计算引擎协同开发 32 位 Windows 应用程序。

1.2.3 Visual FORTRAN 5.0 的安装

Visual FORTRAN 5.0 安装的系统要求：

- 对于标准版，需要 CPU 为 Intel 486/66 MHz （或 100%兼容）以上（推荐 Intel Pentium 系列 CPU），操作系统为 Microsoft Windows NT 4.0 或 Microsoft Windows 95 以上。
- 对于专业版，需要 CPU 为 Intel 486/66 MHz （或 100%兼容）以上（推荐 Intel Pentium 系列 CPU），操作系统为 Microsoft Windows NT 4.0 或 Microsoft Windows 95 以上；或 DIGITAL 的 Alpha CPU（或 100%兼容），操作系统 Microsoft Windows NT 4.0。
- Windows 95 下的 x86 系统至少需要 16 MB 内存（推荐使用 32 MB 以上）；Windows NT 下的 x86 系统至少需要 24 MB 内存（推荐使用 32 MB 以上）；Windows NT 下的 Alpha 系统至少需要 32 MB 内存（推荐使用 48 MB）。
- 用于安装 Visual FORTRAN 的 CD-ROM 驱动器。
- 标准版（x86 系统）需要的硬盘空间为从 30 MB（从 CD-ROM 驱动器运行）至 190 MB（完全安装）；专业版（x86 系统）需要的硬盘空间为从 30 MB（从 CD-ROM 驱动器运行）至 240 MB（完全安装）；专业版（Alpha 系统）需要的硬盘空间为从 40 MB（从 CD-ROM 驱动器运行）至 260 MB（完全安装）。
- VGA 显示器（推荐 17 英寸的 SVGA 显示器）。
- 鼠标或兼容的定位设备。

用户可以根据实际情况看系统是否满足安装要求。如果满足安装要求就可以开始安装 Visual FORTRAN 了。安装需要运行 Setup 程序。一旦安装以后就可以和 Microsoft Developer Studio 一起运行 Visual FORTRAN。

如果需要查看 Visual FORTRAN 和 Developer Studio 的在线文档则必须安装微软的浏览器 Internet Explorer。

如果需要允许在 InfoViewer（信息查看器）中的主题之间跳转，必须使 Internet Explorer 运行 ActiveX Scripts 的特性生效。

可以按如下步骤运行 Setup 程序安装 Visual FORTRAN 和相关软件：

- 启动 Windows NT 或 Windows 95。
- 如果操作系统为 Windows NT，应该以一个拥有管理员特权的名称登录。
- 把 Visual FORTRAN 光盘插入 CD-ROM 驱动器。
- 如果是初次安装该版本的 Visual FORTRAN，光盘上的自动运行程序将显示 DIGITAL Visual FORTRAN 安装的主窗口，如图 1.2 所示。 这时可以直接选择“Install Visual FORTRAN”。
- 如果以前已经安装了该版本的 Visual FORTRAN，自动运行画面将不会显示。这时可以全部或部分从 CD-ROM 运行 Visual FORTRAN。如果要重新安装可以

双击光盘根目录下的 Setup 图标运行安装程序；或在控制面板中选择“添加/删除应用程序”，再选择“添加”，选择“下一步”，最后选择“完成”。

图 1.2 Visual FORTRAN 安装主窗口

- 安装程序会检查是否已经安装了 IE，若没有安装将提示是否安装 IE。
- 随后出现的是欢迎对话框，要求用户在安装过程中停止运行其它应用程序。
- 在出现的软件许可协议对话框中选择接受“Yes”。
- 在注册对话框中输入注册号。
- 这时可以选择“Next>”进入选择安装类型对话框，或选择“<Back”回到上一步，或选择“Cancel”来中止安装。
- 选择安装类型的对话框如图 1.3 所示。

用户可以选择标准安装，定制安装或由 CD-ROM 运行的安装。还可以改变系统默认的安装路径。 Digital 推荐用户安装到类似

C:\Program Files\DevStudio

的顶级目录。在 Windows NT 系统中需要选择程序组的类型：供所有用户使用的公共程序组或只供安装用户的私人程序组。

如果已经安装了类似于 Visual C++的微软可视化开发工具，注意只有相应的 Microsoft Developer Studio 版本相同时才可以把 Visual FORTRAN 安装到同一个目录下。Visual FORTRAN 不支持使用不同版本的 Microsoft Developer Studio 的可视化开发工具。

如果先安装了 Visual FORTRAN 再要安装类似于 Visual C++的微软可视化开发工具，并且在使用 Visual FORTRAN 时出现问题时，可以在 Visual FORTRAN 程序组中选择 Repair #1 来恢复 Visual FORTRAN 的注册记录。

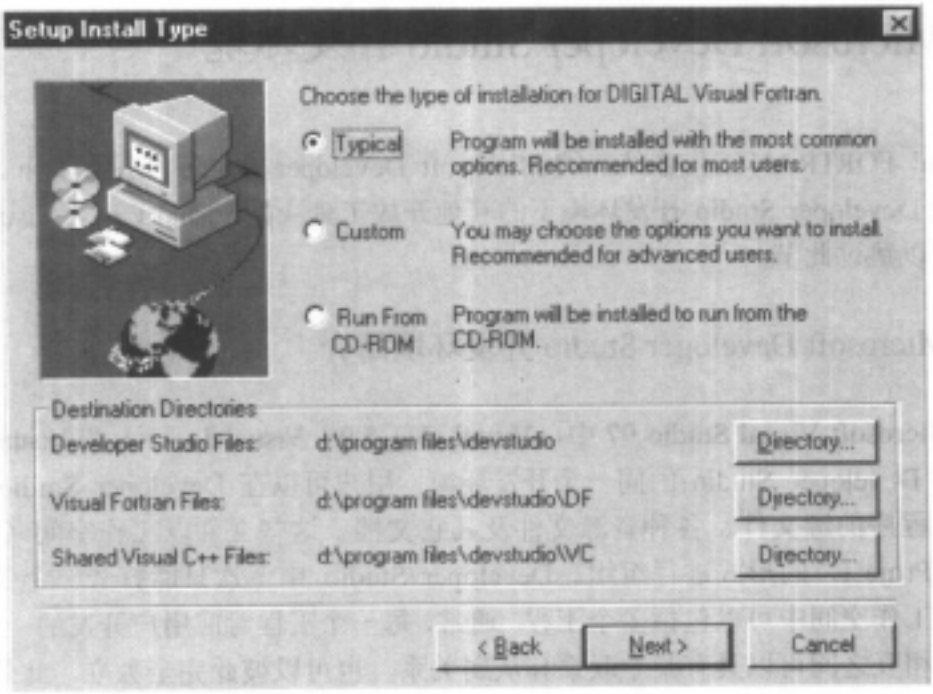


图 1.3 安装类型对话框

对于大部分用户，Digital 推荐标准安装。它安装了 Visual FORTRAN 最常用的组件，但不包括例子，SDK 平台在线文档和其它部分组件。如果选择定制安装但后来希望添加附加组件，可以选择定制安装而不需卸载 Visual FORTRAN。

对于高级用户，Digital 推荐使用定制安装。定制安装可以在选项对话框中选择需要安装的组件，并给出每个组件所需的硬盘空间和可用的硬盘空间。如果 Visual FORTRAN 安装在不同分区则可用硬盘空间是指 Developer Studio 安装的分区。

从 CD-ROM 驱动器运行选项只在硬盘上安装最小的公用 Developer Studio 集合，其它组件需要在 CD-ROM 驱动器上运行。

- 选择好安装类型和路径之后，安装程序开始拷贝文件，升级注册表和创建程序组和相应的图标。
- 之后出现提示是否升级使用命令行窗口的环境变量。如果选择“是”则将对 Windows 9x 的 AUTOEXEC.BAT 文件进行修改：在 AUTOEXEC.BAT 文件中插入一条 CALL 命令来执行 DFVARS.BAT 文件。Visual FORTRAN 支持应用命令行的一些特性但几乎不必改动环境变量。一方面，Visual FORTRAN 程序组包括一个配有合适环境变量集的 FORTRAN 90 命令行窗口，另一方面，可以通过执行 DFVARS.BAT 文件来设置这些环境变量。在专业版中，DFVARS.BAT 还可以设置 IMSL 例行子函数的合适的环境变量。
- 安装完成。

1.3 Microsoft Developer Studio 开发环境

Visual FORTRAN 工作在统一的 Microsoft Developer Studio 开发环境下。如果读者对 Microsoft Developer Studio 开发环境下的其他开放工具（如 Visual C++，Visual Basic 等）很熟悉可以跳过此节。

1.3.1 Microsoft Developer Studio 开发环境简介

在 Microsoft Visual Studio 97 中，Visual C++ 5.0、Visual J++ 1.1 和 Visual InterDev 都使用称作 Developer Studio 的同一个开发环境。用户可以在 Developer Studio 中创建所开发的应用程序的源文件、各种资源文件及其它文档。这些文件以工作空间（Workspace）和工程（Project）的形式进行组织。Developer Studio 中一次只能打开一个工作空间，但在同一个工作空间中可以包括多个工程。通常，每一个工程对应用户开发的一个应用程序。这些工程相互之间可以具有某个联系和从属关系，也可以彼此完全独立。此外，这些工程的类型还可以不同。工程中除了包括应用程序所用到的源文件、资源文件外，还可以包括其它类型的文件，如应用程序的规格说明书、流程图、开发日程等等。对于那些由 ActiveX 部件(如字处理软件和空白表格软件等)所创建的 ActiveX 文档，可以在 Developer Studio 中直接打开。而对于与其它类型的应用程序相关联的文档也可以通过 Developer Studio 在独立的窗口的打开。

Developer Studio 所包括的内容十分丰富。它集成了文本编辑器，资源编辑器，工程建立工具，增量连接器，源代码浏览器，内嵌调试器和包含一般编辑、编译、连接、调试信息的在线用户指南。但并不是所有的 Developer Studio 产品都具有以上工具，而且有些 Developer Studio 产品还增加了其它工具。通过一个由窗口、对话框、菜单、工具栏及宏组成的和谐系统，用户可以观察和控制整个开发过程。一个典型的 Developer Studio（在这里是 Visual FORTRAN）主窗口如图 1.4 所示。

当然，用户所看到的内容和工具条等也许会有少许的不同，这取决于用户的设置。下面将讲述其中的主要部分。在 Developer Studio 中，整个窗口被分成了若干个部分，需要注意的是，随着设置的不同，或者所安装的软件包的不同，或者是处于开发的不同阶段(典型的是，在输入源代码和调试程序的两个不同阶段)，用户所见到的 Developer Studio 组件和相互之间的位置也会不一样。

用户还可以使用 Developer Studio 中的信息查看器（InfoViewer）或系统本身的 Web 浏览器来浏览万维网(World Wide Web)。用户可以访问到 Developer Studio 资源窗口的 URL 所指向的 Web 页，信息查看器主题窗口超文本跳转的 Web 页，以及信息查看器工具条或帮助菜单的当前 URL 所指向的 Web 页。

本节下面的内容将要详细介绍 Visual FORTRAN 的环境外壳和开发应用程序时要碰到的界面和窗口。

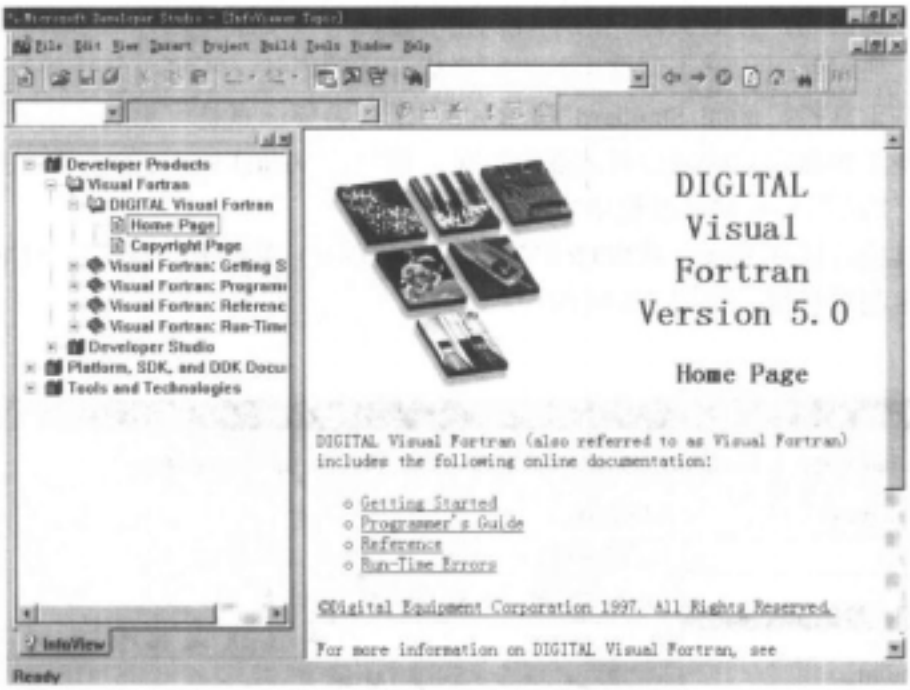


图 1.4 典型的 Developer Studio 主窗口

1.3.2 工具栏和菜单

Visual FORTRAN 带有一个已经预先定义好的工具栏集，可以通过单击来访问这些工具栏。如果找不到所需的工具可以自己定制工具栏来扩大工具栏集。每个工具栏都由工具栏的标题栏上的名字标识。图 1.5 分别显示了建立（Build）、标准（Standard）和编辑（Edit）工具栏定位在 Visual FORTRAN 主窗口顶层的样子。

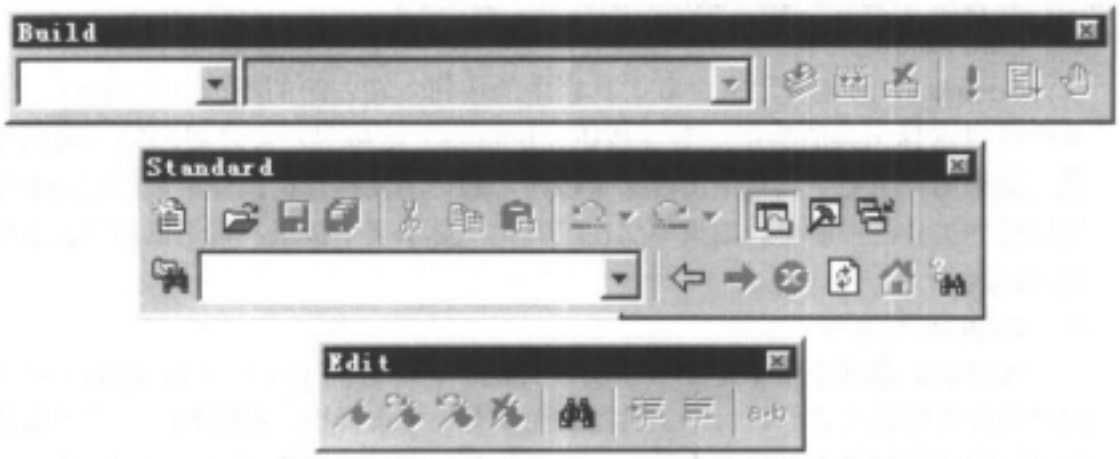


图 1.5 典型的工具栏

工具栏可以由用户自己安排。用户可以在屏幕四周移动工具栏并把它们放在自己认为合适的位置；通过拖曳边框来调整工具栏矩形成合适的形状；还可以使任何一套工具看见或隐藏。某些工具栏，例如 Standard 和 Build，用户希望一直是可见的；而其它一些工具栏通常只希望工作在它们的窗口时才变得可见。例如在 Visual FORTRAN 默认设置中，调试（Debug）工具栏只在调试过程中才是可见的。

定制工具栏可以在 Visual FORTRAN 的工具（Tools）菜单里选择定制（Customize），然后会弹出定制对话框，如图 1.6 所示。

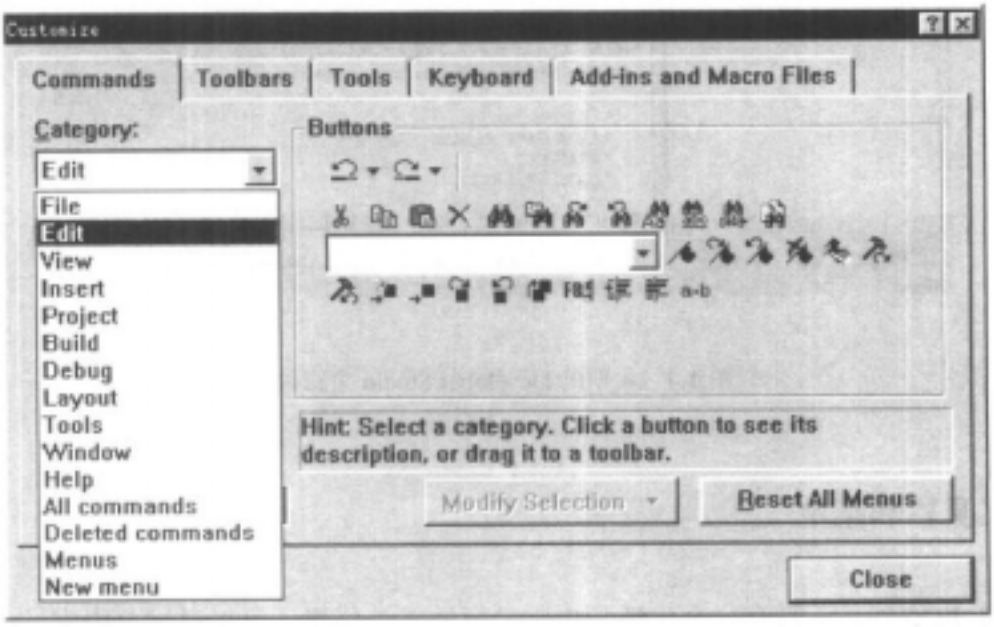


图 1.6 定制工具栏对话框

定制对话框有命令（Command）、工具栏（Toolbars）、工具（Tools）、键盘（Keyboard）、附加项和宏文件（Add-in and Macro Files）五个选项卡。

- Command 选项卡。

Command 选项卡用来定制工具栏中的按钮。在左边的类型（Category）下拉菜单中可以选择工具的类型，右侧按钮（Buttons）区域显示与之对应的工具类型的按钮。如果用户希望工具栏中出现某个按钮，则可用鼠标左键按住所选的按钮不放，再把它托到工具栏中即可。如果不想在工具栏中显示某个按钮，可用鼠标把该按钮拖回到按钮区。

- Toolbars 选项卡。

Toolbars 选项卡用来定制显示的工具栏。在左边的工具栏（Toolbars）区中显示的是所有可用的工具栏，每个工具栏前面都有一个选择框，在选择框中单击鼠标左键即可选中或取消该工具栏。右边还有三个选项：显示工具提示（Show ToolTips）、使用快捷键（With shortcut keys）和大按钮（Large buttons）。在 Visual FORTRAN 已经存在的工具栏上单击鼠标右键会也显示所有的工具栏和它们的显示/隐藏状态，用户

可以直接对显示的内容进行修改。

- Tools 选项卡。

Tools 选项卡用来定制 Tools 菜单中包含的命令。在该选项卡中可以选择添加、删除或调整命令的顺序。修改该选项卡的结果将使 Tools 菜单发生相应的的变化。

- Keyboard 选项卡。

Keyboard 选项卡用来定义命令的快捷键。先从类型（Category）下拉菜单中选择命令类型，然后在编辑器（Editor）下拉菜单中选择编辑器类型，再从命令（Commands）下拉菜单中选择命令，可以在当前快捷键（Current keys）框里看到默认的快捷键（有些可能没有定义），然后在按下新的快捷键（Press new shortcut）框中定义用户希望的快捷键了。

- Add-in and Macro Files 选项卡。

Add-in and Macro Files 选项卡用来定制工具（Tools）菜单中的宏和附加项。单击在选项卡中列出的附加项和宏左侧的选择框可以选择需要的附加项和宏。

Developer Studio 的菜单类似于标准的 Windows 应用程序，最多只增加了工程（Project）和建立（Build）菜单。Developer Studio 环境大部分情况下都响应鼠标右键单击，所显示的是一个弹出式并与位置相适应命令的上下文相关菜单。

1.3.3 环境窗口

除对话框之外，Visual FORTRAN 显示两种类型的窗口：文档窗口和停靠窗口。

文档窗口是一般的带边框子窗口，其中包含有源代码文本和图形文档。窗口（Windows）菜单中列出了三种显示文档窗口的命令：层叠（Cascade）、标题水平放置（Tile Horizontally）和标题垂直放置（Tile Vertically）。

停靠窗口是其它的 Visual FORTRAN 窗口（包括工具栏和菜单栏）。开发环境有两个主要的停靠窗口：工作空间（Workspace）窗口和输出（Output）窗口，它们通过查看（View）菜单中的命令变成可见的窗口。

停靠窗口可以固定在 Visual FORTRAN 用户区的顶端、底端或侧面，或者浮动在屏幕上任何地方。固定或浮动的停靠窗口都总是出现在文档窗口上面。所以浮动工具栏始终是可见的。但有时某个占据整个 Visual FORTRAN 用户区的停靠窗口可能覆盖了需要使用的文档窗口，这时需要关闭或拖动停靠窗口。

拖动一个停靠窗口时会出现一个移动的轮廓，这个轮廓就是释放鼠标左键后窗口的新位置。轮廓线是灰色的虚线，直到与环境用户区的边界或另一个停靠窗口的边界接触时才变成细黑线。这时释放鼠标左键后窗口会定位在边界附近的位置。工具栏在用户区的顶端或底端时定位为水平位置，而在左右边界时定位为垂直位置。拖动工具栏时可以通过按 Shift 键来改变工具栏的放置方向。

要使窗口占据整个用户区可以向上拉动边界直到与用户区的上边界接触，然后释放鼠标左键。如果要把窗口恢复成较小的尺寸可以拖动窗口，直到光标接触到用户区的左边界，这将迫使窗口成为浮动窗口，这时可以把它拖到其它位置。

在屏幕上移动一个停靠窗口有时会比较困难，窗口可能不会离开 Visual FORTRAN 主

窗口的边界或其它与之接触的定位窗口。解决这个问题有两种方法。第一种，可以在移动窗口时按下 Ctrl 键，这时窗口的停靠特征被暂时禁止。第二种，在窗口（Windows）菜单中选择停靠视图（Docking View）命令来关闭命令的复选标志，这样就禁止了窗口的停靠能力。还以在窗口内部单击鼠标右键从上下文相关菜单中来选择 Docking View。第二种方法只对窗口有效而对工具栏无效。

关掉窗口的停靠特征会在以下几个方面影响窗口：

- 窗口变成类似普通文档窗口，标题栏有最小化、最大化和关闭按钮。
- 从窗口（Windows）菜单选则层叠（Cascade）、标题水平放置（Tile Horizontally）、标题垂直放置（Tile Vertically）命令时窗口和其它打开的文档窗口一起排列。
- 窗口不能移出 Visual FORTRAN 主窗口，只有恢复窗口的停靠特征时才可以移到主窗口之外。
- 当它为当前窗口（标题栏为实颜色的窗口，也就是用户最近操作过的窗口）时可以由窗口（Windows）菜单上的关闭（Close）命令关闭。在停靠模式下，即使是当前窗口也不会受到 Close 命令的影响。

窗口或工具栏被定位后，凸起的把手（两条凸起的细线，如图 1.7 所示）出现在窗口的顶端或左边界。双击把手可以使窗口变成浮动的；双击浮动窗口或工具栏的标题栏可使它回到原来的位置。拖动把手还可以把窗口移到另外一个固定或浮动的位置。

在 Visual FORTRAN 中创建的窗口排列会在项目的整个过程中维持。而在环境中运行的实用程序的窗口并不遵守这些规则。因为它们即不是文档窗口也不是停靠窗口，其特征是由实用程序而不是 Visual FORTRAN 决定。

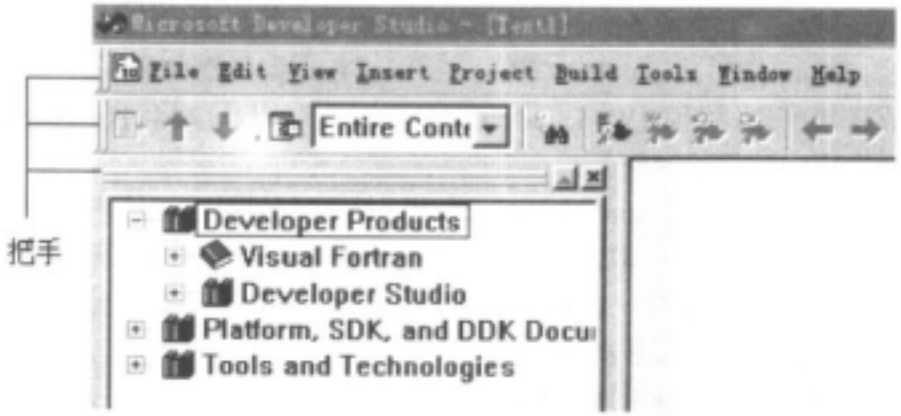


图 1.7 窗口停靠时的把手

1.3.4 工作空间（Workspace）窗口和输出（Output）窗口

Visual FORTRAN 在停靠的工作空间（Workspace）和输出（Output）窗口中显示项目的有关信息。这两个窗口，尤其是工作空间窗口，是始终要遇到的重要窗口。在查看（View）

菜单中单击 Workspace 或 Output 可以使这两个窗口变成可见。还可以通过标准（Standard）工具栏上的按钮来激活。如图 1.8 所示，右侧的两个按钮分别是 Workspace 和 Output。

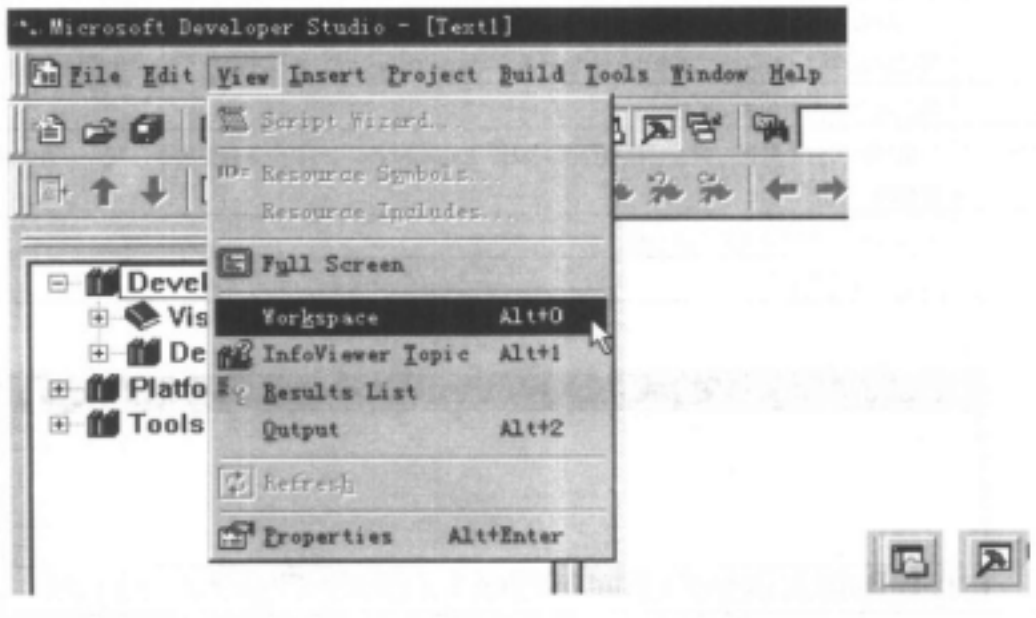


图 1.8 显示 Workspace 和 Output 的窗口和按钮

隐藏 Workspace 和 Output 窗口的方法除了使用工具栏按钮以外还有下面几种方法：

- 如果窗口是浮动的，可以单击窗口的标题栏上的关闭按钮。
- 如果窗口是固定的，可以单击把手右端或上端的关闭按钮。
- 在窗口内单击右键打开相关菜单，选择隐藏（Hide）或关闭（Close）命令。
- 如果窗口的停靠特征被禁止，可以单击窗口使它变成当前窗口，然后从窗口（Windows）菜单中选择关闭（Close）命令。

工作空间（Workspace）窗口显示了项目各个方面的信息。在窗口底端选择相应的选项卡来显示项目的类、资源、数据源和文件的列表。当选择了某一选项卡后，Workspace 的文件组织类似于 Windows 9x 中的资源管理器，在窗口中单击加号（＋）或减号（－）可以展开或折叠列表。

Workspace 窗口可以显示多至五个选项卡的信息，如表 1.1 所示：

输出（Output）窗口（如图 1.9 所示）有四个选项卡：建立（Build）、调试（Debug）、在文件 1 中查找（Find in Files 1）和在文件 2 中查找（Find in Files 2）。Build 选项卡显示编译器、链接器和其它工具的状态信息。Debug 选项卡用于通知来自调试器的提示，这些提示对例如未处理的异常和内存异常之类的情况提出警告。

另外两个查找选项卡显示从编辑（Edit）菜单中选中的在文件中查找（Find In Files）命令的执行结果。在默认情况下，Find in Files 搜索结果显示在输出（Output）窗口的 Find In Files 选项卡中，但 Find In Files 对话框中的一个复选框，允许用户把结果转移到 File In Files 2 选项卡中。Output 窗口中还可以包含其它选项卡。

表 1.1 选项卡

选项卡名称	描　述
FileView	显示项目的源文件。可以把源文件复制到项目夹中，但不会把文件添加到 FileView 窗口的列表中，必须通过在项目（Project）菜单中的添加到项目（Add To Project）命令明确地把新文件添加到项目中。
ClassView	列出项目中的类和成员函数。
Resource View	列出项目的资源数据，如对话框和位图。双击 ResourceView 列表中的数据项会打开合适的编辑器并加载资源。
Data View	显示数据库项目的数据来源信息。
InfoView	显示在线帮助文档。

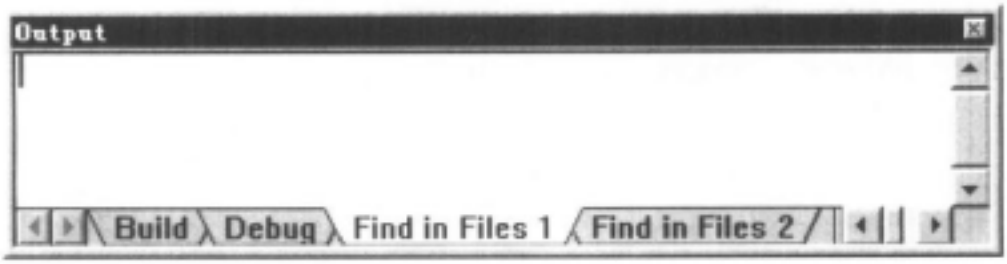


图 1.9 Output 窗口

1.4 在线帮助

随着计算机软硬件的飞速发展，应用程序越来越复杂，程序员所需要掌握和用到的信息越来越多。如果没有在线文档的帮助，开发应用程序的过程几乎无法进行。对于一个开发工具来说，在线文档做到是否完善，是否易于使用，成为衡量一个开发工具是好是坏的一个重要标准。Visual FORTRAN 延续了 Developer Studio 的一贯优良作风，其在线帮助文档参考资料全面而详尽，足以满足各个不同层次的需要。但是，如何才能最有效的利用 Visual FORTRAN 博大精深的联机帮助是一个很值得探讨的课题。从在线文档中快速地寻找到所需的各种资料的技能，和从任何一本书中学到的知识同等的重要。

因此，在学习使用 Visual FORTRAN 进行应用程序设计之前，先学习一下如何从 Visual FORTRAN 的集成开发环境 Developer Studio 中获得帮助是很有必要的。

1.4.1 使用 InfoView

在 Workspace 窗口的 InfoView 选项卡中包括了 Visual FORTRAN 中每一份在线文档的树状结点列表。很多情况下，用户都是从 InfoView 入手，来一步一步地查找到所需要的各种资料的。因此，有必要简单地了解 InfoView 中的结点的组织形式。这里列举一些用户应该经常查看的节点。

- 程序员指南（Programmer's Guide）
  如图 1.10 所示。

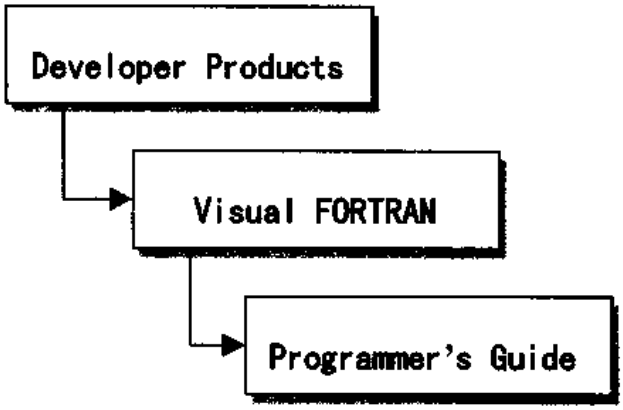


图 1.10 Programmer's Guide 节点

Visual FORTRAN Programmer's Guide 是最常访问的节点之一。这里提供了大多数的编程任务所需的知识。

- 语言参考手册（Reference）
  如图 1.11 所示。

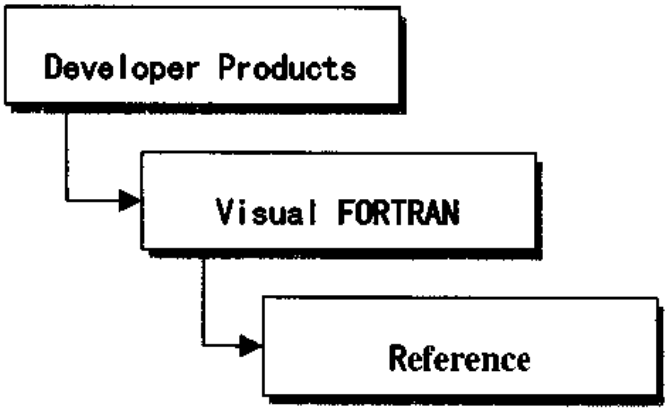


图 1.11 Reference 节点

语言参考手册前一部分以分类表格的形式给出，后一部分以字母顺序给出。用户如果有编程语言方面的问题可以根据需要查询。

- 运行错误（Run-time Errors）
  如图 1.12 所示。

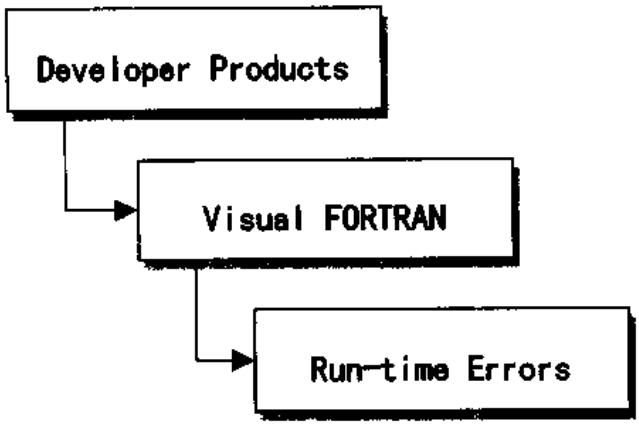


图 1.12 Run-time Error 节点

编译或建立程序时如果出现错误，在这个节点中可以根据错误信息的编号获得较详细的信息。

- Developer Studio 开发环境用户指南（Developer Studio Environment User's Guide）如图 1.13 所示。

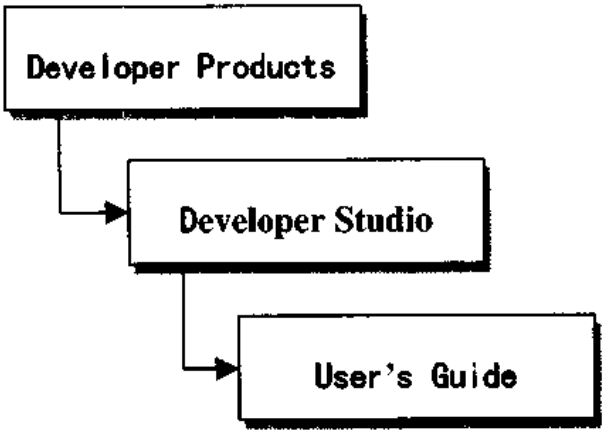


图 1.13 Developer Studio Environment User's Guide 节点

Developer Studio 是一个内容相当丰富的开发环境，在这个环境中工作需要经常查看相关信息。

1.4.2 使用上下文相关的帮助

最常用的一种方法是通过上下文相关的帮助快速的获得所需的信息。打开上下文相关的帮助的快捷键是 F1。见图 1.14。上下文相关的帮助可以用于多种场合。

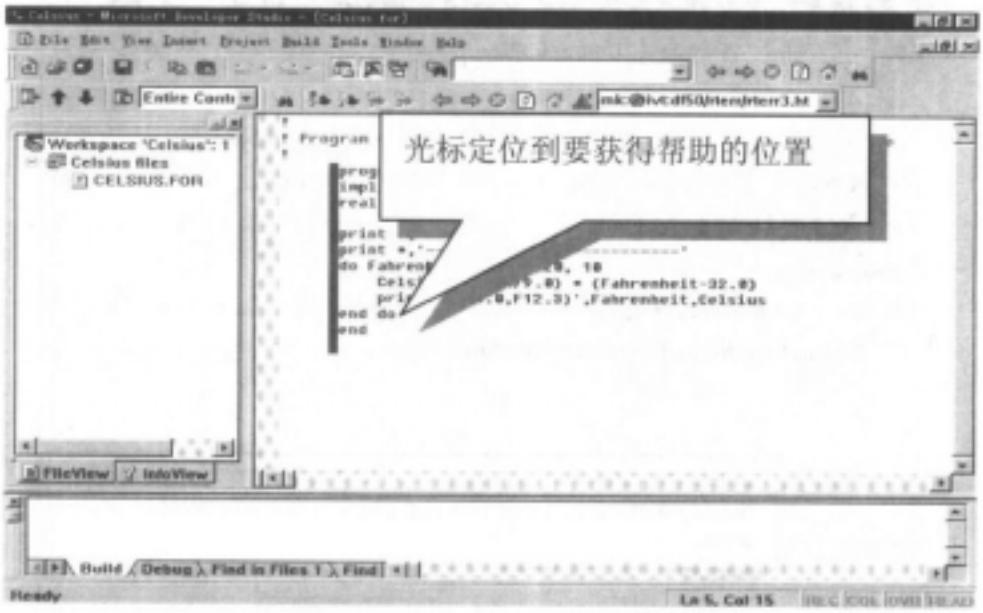


图 1.14 光标定位获得帮助

1.4.3 其它帮助途径

1. WinHlp32 查看器显示的帮助 HLP 文件

标准的 HLP 文件内容包括环境的命令和窗口，当 Visual FORTRAN 不能为帮助主题确定明确的上下文时，按 F1 键就会显示这些帮助文件。例如在文本编辑器中，如果光标停留在一个没有上下文的空行上时，按下 F1 键会提供文本编辑器窗口的有关信息，它们显示在 WinHlp 查看器中，如图 1.15 所示。

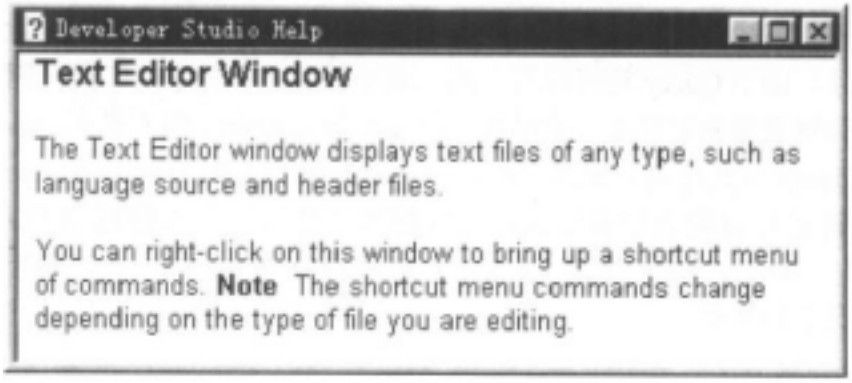


图 1.15 帮助信息

2. 对话框中弹出式帮助信息

在对话框右上有一个“？”标志，单击它之后再单击对话框中的文本或按钮就可以显示相应的帮助信息，如图 1.16 所示。

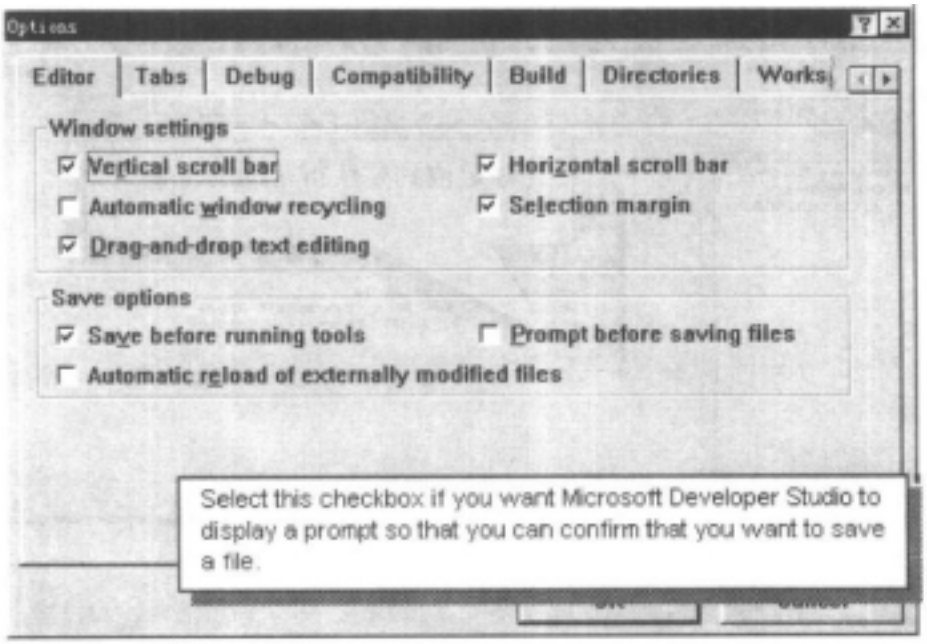


图 1.16 典型的对话框弹出帮助

3. 查询对话框

在 Help 菜单中选择 Search 命令或者单击工具条上的按钮来打开如图 1.17 所示的查询对话框。

该对话框分成两个选项卡：选项卡 Index 以索引方式来查询在线文档，文档中的每一个主题都与一个或多个索引关键字相联系；反之，一个索引关键字也可能与多个文档主题相联系。如果用户不知道所需要的资料在 InfoView 窗口的内容列表中的节点位置，那么使用 Index 方式进行查询是一个很好的办法。另一个选项卡称作 Query (查询)选项卡。该选项卡允许用户指定查询字符串，然后从在线文档中查找匹配的所有文档，并且，还可以指定多种不同的查找方式和范围。一般来说，通过 Query 得到的结果要比使用 Index 的庞大得多，并且，由于 Query 还可以对文本，而不仅仅是标题进行匹配，因此，也许会得到一些事实上和所需要的资料无关的结果。这使得使用 Query 没有使用 Index 那么方便和有效。然而，Query 方式也有其先进之处，在 Query 选项卡的 Type in the word(s) to find 处可以使用含逻辑运算符的查询表达式，可以使用的逻辑运行符包括 AND、OR、NOT 和 NEAR，其中，AND、OR、NOT 分别还可以简写为“&”、“|”、“!”。四个逻辑运算符的含义和示例如表 1.2 所示。

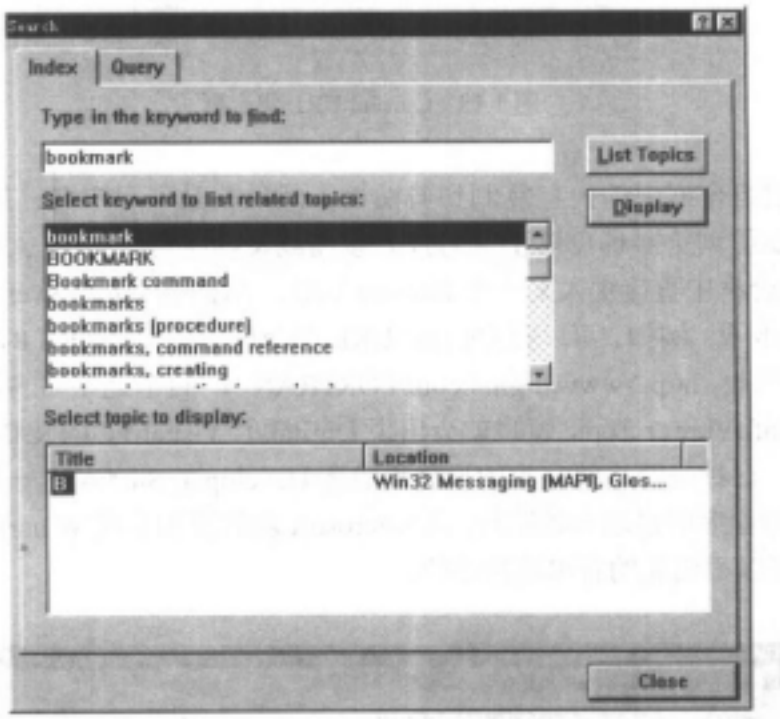


图 1.17 查询对话框

表 1.2 在 Query 方式中使用逻辑运算符

运算符	示 例	含 义
AND	PRINT AND READ	同时包括 PRINT 和 READ 的匹配项
OR	PRINT OR READ	包括 PRINT 或 READ 的匹配项
NOT	PRINT NOT READ	包括 PRINT，但不包括 READ 的匹配项
NEAR	PRINT NEAR READ	在 READ 周围 8 个字内包括 PRINT。用于 NEAR 运算符的匹配范围可以在通过 Tools 菜单的 Option 命令在 InfoViewer 选项卡中进行设置

1.5 Developer Studio 与 Web

除了用来查看在线文档的窗口外，Developer Studio 的 InfoView 窗口还可以作为一个 World Wide Web 浏览器使用。事实上，Visual FORTRAN 5.0 的在线文档就是一系列的超文本文档。为了验证这一点，用户可以在 InfoViewer Topic 窗口中打开一个在线文档主题，然后工具条上出现 Current URL 组合框，如图 1.18 所示。

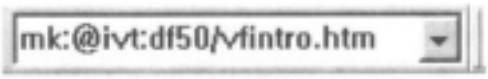


图 1.18 Current URL 组合框

用户可以把组合框中的内容复制到剪贴板，然后粘贴到用户自己的浏览器的地址框中，这时可以发现浏览器也可以正常的打开该在线文档主题。另一方面，用户也可以在 Current URL 组合框中直接键入某一个 Internet URL，从而在 InfoViewer Topic 窗口中直接打开 Internet Web 页。例如，可以在 Current URL 组合框中输入 Digital 的 Visual FORTRAN 技术支持主页地址 http://www.digital.com/FORTRAN ，回车之后如果用户已经连接到 Internet，那么 InfoViewer Topic 窗口就会出现 Digital 的 Visual FORTRAN 技术支持主页，如图 1.19 所示。这种与网络的完整的无缝集成是 Developer Studio 的一大特点，并且在新的 Windows 应用程序中也越来越流行，从 Microsoft 新的操作系统 Windows 98 和 Windows NT 5.0 中我们可以很明显的看出这种趋势。

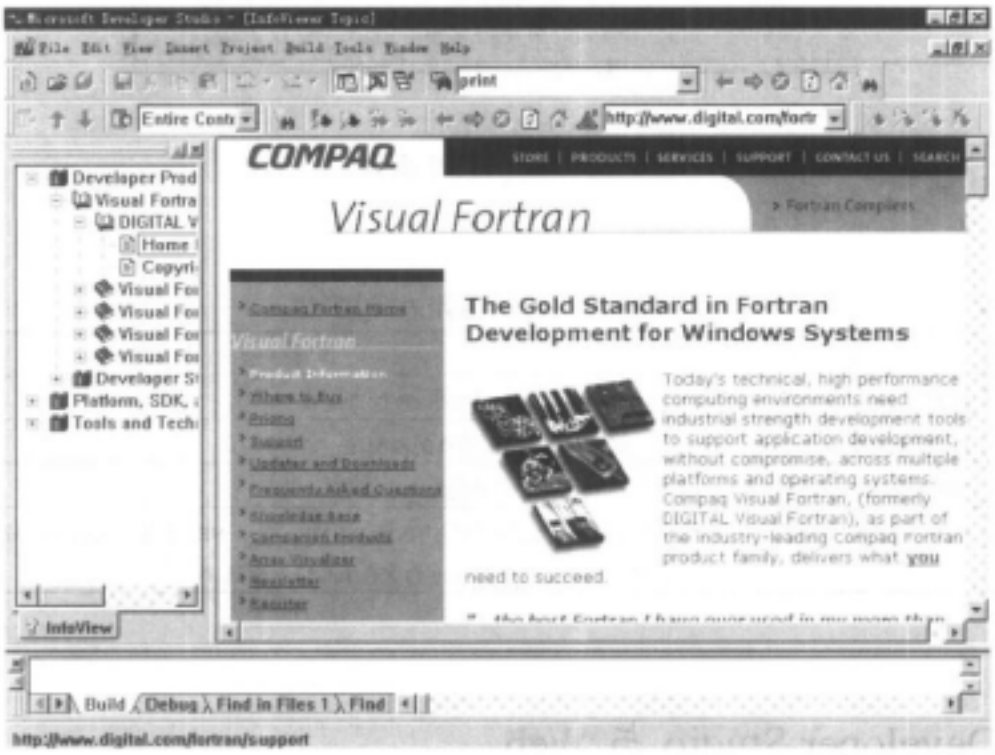


图 1.19 Developer Studio 与 Web

第二章 FORTRAN 90 基础知识

这一章主要介绍 FORTRAN 90 中最基本的一些知识，即字符集、程序结构、表达式和源程序书写的几种格式。

2.1 字符集

FORTRAN 90 字符集由下面字符组成（斜体字表示 Visual FORTRAN 的扩展）：

- 所有大写和小写英文字母：
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
  a b c d e f g h i j k l m n o p q r s t u v w x y z
  在控制程序方面不区分字母的大小写（除了字符常量和 *Hollerith 常量*）。
- 十个阿拉伯数字：
  0 1 2 3 4 5 6 7 8 9
- 下划线（_）。
- 其它特殊字符，如表 2.1 所示。

表 2.1 **特殊字符**

字 符	名 称	字 符	名 称
空格或 <Tab>	空格或 tab	:	冒号
=	等号	!	感叹号
+	加号	"	引号
-	减号	%	百分号
*	星号	&	And（与）符号
/	斜线	;	分号
(	左括号	<	小于号
)	右括号	>	大于号
,	逗号	?	问号
.	句号（小数点）	$	美元符号
'	撇号		

- 其它可打印字符。

可打印字符包括 tab 符（十六进制 ASCII 码为 09h），ASCII 码在 20h 和 7Eh 之间的字符。可打印字符只能出现在注释、字符常量、字符串编辑符和输入/输出记录中。

FORTRAN 90 中的字符集较 FORTRAN 77 中的字符有了较大的扩充和更大的灵活性，其中_（下划线）、!（叹号）、"（双引号）、%（百分号）、&（and）、;（分号）、>（大于号）、<（小于号）、?（问号）等 9 个字符是 FORTRAN 90 新增的。

2.2 程序结构

本节介绍程序单元，语句，名称和关键字等 FORTRAN 程序结构。

2.2.1 程序单元

程序单元是 FORTRAN 90 程序的基本成分。一个 FORTRAN 程序由一个或多个程序单元组成。程序单元通常是定义数据环境和实施计算所必须的一系列步骤。程序单元以 END 作为结束标志。

程序单元可以是主程序、外部子程序，模块或块数据单元。一个可执行程序包含一个主程序和任意数目的其它类型的程序单元。程序单元可以分别编译。

一个外部子程序是主程序、模块或其它子程序不包括的函数或例行子程序。它定义了一个可执行的过程并且可以由 FORTRAN 程序的其它单元进行调用。模块和块数据程序单元不是可执行的，所以不认为它们是函数（但模块可以包含模块）。

模块包含可以由其它程序单元访问的定义：数据和类型定义、过程定义（称为模块子程序）和过程接口。模块子程序可以是函数或例行子程序。它们可以被其它模块中的模块子程序或其它可访问该模块的程序单元调用。

块数据程序单元给定了一个已命名的公共块中的数据对象的初始值。

主程序、外部子程序和模块子程序可以包含内在子程序。包含内在子程序的实体称为内在子程序的宿主（Host）。内在子程序只可以被宿主和宿主内的其它内在子程序调用。内在子程序不能再包含内在子程序。

主程序单元或子程序单元它们的基本形式相似：以开始语句作为程序单元的第一条语句，然后在程序体，最后是该程序单元的结束语句。例如程序：

主程序
```
PROGRAM DEMO
REAL Result                 ! 说明部分
CALL ROOT                   ! 执行部分
CONTAINS
SUBROUTINE ROOT             ! 内部过程
      …                                  } 函数辅程序
END SUBROUTINE ROOT
END PROGRAM DEMO
```

注意这里的结束语句与 FORTRAN 77 结束语句的不同。说明部分只允许写说明语句。

2.2.2 语句

1. 语句的类型

- 说明语句

一般形式：类型关键字::变量名 1，变量名 2，…，变量名 n

类型关键字声明变量 的数据类型，如 REAL，INTEGER，COMPLEX 等等。双分隔符： ：后面的变量 名是被说明的对象，它们的类型取关键字指定的类型。例如，

INTEGER:: A，B REAL:: C，D，E

说明 A，B 为整型变量 ，C、D、E 为实型变量。

- 可执行语句

一般形式：关键字 变量名 1，变量名 2，…，变量名 n

关键字指定具体操作，变量 名是被执行操作的对象。例如，

READ ，X，6；PRINT，X+Y

指定机器要求读入 X、Y 的值，打印输出 X 与 Y 之和的值。

上面只是说明语句和执行语句的最一般形式，实际编程时，还要求指出各种附属性质，这些将在以后的章节中给予介绍。

2. 语句排序规则

在 FORTRAN 90 中对语句排序的要求与 FORTRAN 77 的要求有所不同。图 2.1 给出了语句序列的一般规则，这些规则适用于全部程序单元和辅程序的排序规则。竖行描写了各种语句所能分布的范围，横行描写了各种语句允许的出现顺序。

一个可执行程序的执行是从主程序的第一个可执行构造开始的。当引用一个过程时，从引用的入口点之后出现的第一个可执行构造处开始执行。程序执行是按可执行构造次序执行，直到执行 STOP、RETURN 或 END 语句时为止。在下列情况时，可改变执行序列：

（1） 转移语句的执行。

（2） IF 构造、CASE 构造和 DO 构造包含了一个内部语句结构，该结构中包含了隐式（即自动）的内部转移。

（3） 交错返回和 END=、ERR=及 EOR=说明符的存在。

3. 作用域

如果在某一程序单元 中说明了一个名字的类型及含意、所代表的一个变量或过程等，那么对名字的有关说明就可从该程序单元第一条语句开始到 END 语句这个范围内有意义。这就是名字的作用域。

语句可分为两类：可执行语句和不可执行语句。可执行语句指定了要被实施的动作。不可执行语句描述了程序属性，例如数据的安排和特征，编辑和数据转换信息。

关于程序单元中的语句顺序为：图 2.1 显示了 FORTRAN 程序单元中需要的语句的顺序。图中，竖直线把可以互相颠倒的语句分开。例如，可以在可执行结构中插入 DATA 语句。

水平线表示的是不能颠倒的语句。例如不能把 DATA 语句和 CONTAINS 语句颠倒。

PUBLIC 和 PRIVATE 语句只能在模块的范围单位出现。表 2.2 说明了其它由范围单元

类型限制的语句。

表 2.2 单元范围和受限制语句

单元范围	受限制语句
主程序	ENTRY 和 RETURN 语句
模块 [1]	ENTRY,FORMAT,OPTIONAL 和 INTENT 语句，语句函数和可执行语句
块数据程序单元	CONTAINS, ENTRY, 和 FORMAT 语句,接口块，语句函数和可执行语句
内在子程序	CONTAINS 和 ENTRY 语句
接口体	CONTAINS, DATA, ENTRY, SAVE, 和 FORMAT 语句语句函数和可执行语句

[1]模块的范围单元不包括该模块包含的任何任何模块子程序

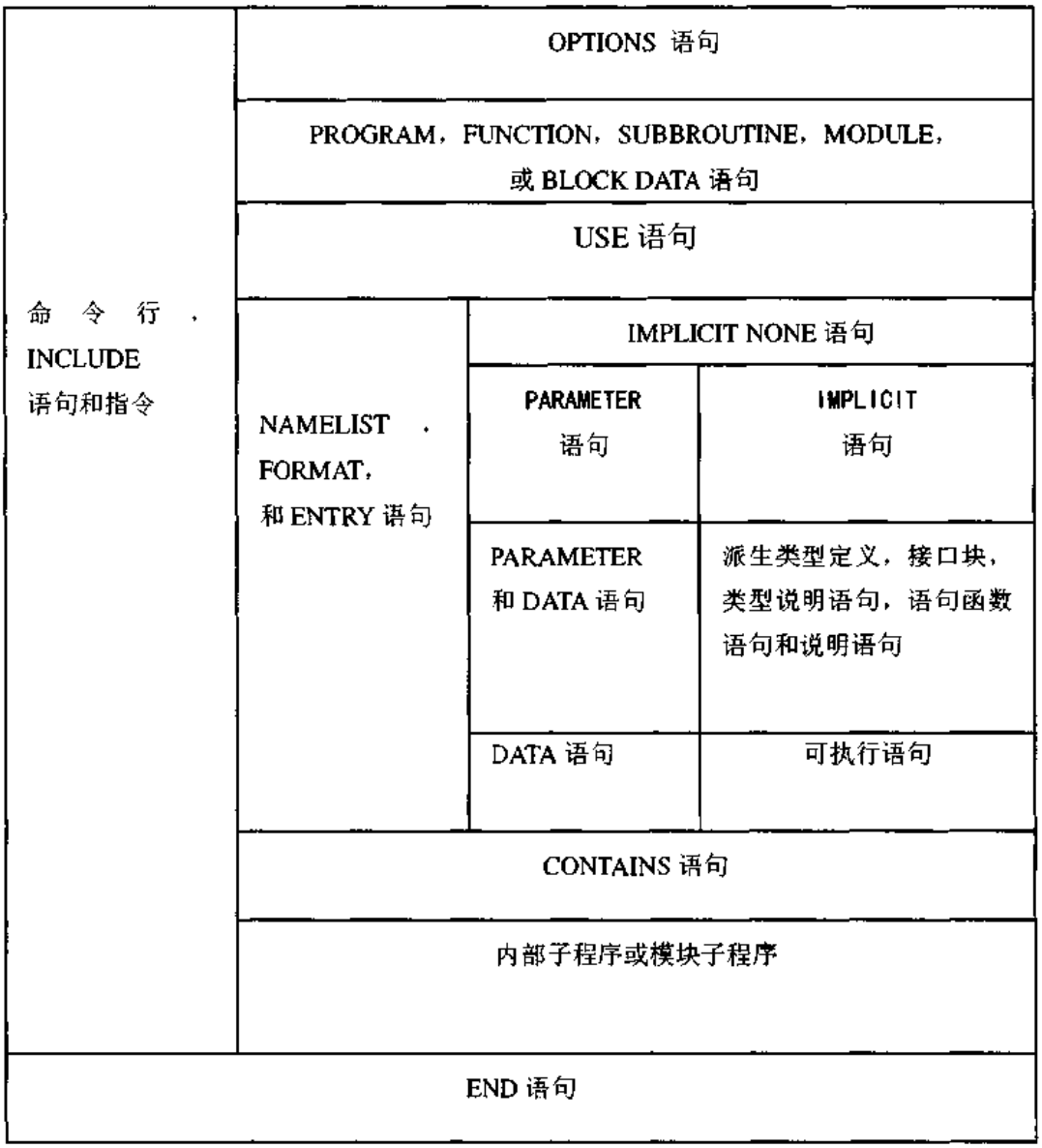


图 2.1 语句顺序

2.2.3 名称

名称在 FORTRAN 程序单元中用来区分实体（例如变量、命名常量、过程、程序单元、名称列表组和哑元）。在 FORTRAN 90 中用“名字”来代替以往 FORTRAN 术语中的“符号名”，名字沿用以往符号名的定义。

FORTRAN 90 中规定了名字的最大长度不能超过 31 个字符，名字由英文字母和数码及下划线*或美元符号$*组成，第一个字符必须是英文字母。

在 FORTRAN 90 中还引入了指定符的概念用它指定子对象。子对象指定符是一个名，其后跟着一个或多个下列内容：成为选择符、数组片段选择符、数组元素选择符以及子串选择符。例如，T（1：7）和 T（1）%VECTOR 就是子对象指定符，它由对象的名字后面跟有关的选择符组成。

在 OpenVMS 系统上，命名习惯保留了包含$的名称。在 Digital UNIX，Windows NT 和 Windows 9x 系统中，$可以作为各种外壳和命令的命令符号或替代符号。

在可执行程序中，下列实体的名称是全局名称并且在整个程序中必须唯一：

- 程序单元；
- 外部过程；
- 命令块；
- 模块。

例：表 2.3 列出了合法及非法的名称。

表 2.3 合法和不合法的名称

合 法	
NUMBER	
FIND_IT	
X	
非 法	注 释
5Q	名称开始使用数字
B.4	包含了除_ 或$之外的特殊字符
_WRONG	名称开始使用_

下面都是合法使用名称的例子：

```
INTEGER (SHORT) K          !K 命名了一个整型变量
SUBROUTINE EXAMPLE         !EXAMPLE 命名了一个子程序
LABEL: DO I = 1,N          !LABEL 命名了一个 DO 循环
```

在 FORTRAN 90 中还引入了指定符的概念，用它指定子对象。子对象指定符是一个名，其后跟着一个或多个下列内容：成为选择符、数组片段选择符、数组元素选择符以及子串选择符。例如，T（1：7）和 T（1）%VECTOR 就是子对象指定符，它由对象的名字

后面跟有关的选择符组成。

2.2.3 关键字

在 FORTRAN 90 中将术语关键字分为语句关键字和变元关键字两种。

- 语句关键字

  语句语法部分的一个词是语句关键字。例如：IF、READ、UNIT、KIND、INTEGER 等等。

- 变元关键字

  变元关键字是哑元名。如对所有的内在过程规定了变元关键字，对外部过程的变元关键字在过程接口块中做出了规定。

  例如在内在函数

  UNPACK (VECTOR, MASK, FIELD)

中，VECTOR, MASK 和 FIELD 是变元关键字。它们是哑元参数名，任何变量都可以取代它们的位置。

关键字不予保留，编译器根据上下文判断是否为关键字。例如，程序可以有一个名为 IF，read 或 Goto 的数组，但这样显然是不好的程序风格。

变元关键字是 FORTRAN 90 的特性，允许用户在单元内在过程时指定哑元参数名称，或任何定义接口（不论隐式还是显式）的位置。使用变元关键字使程序可读性更强、更容易接受。

2.3 表达式

变量、常数、函数引用和操作符的组合称为表达式。表达式是计算数值的公式。操作符指定了对操作数的操作。

2.3.1 内部操作符

FORTRAN 90 中共有 4 类操作符，如下表所示：

表 2.4　　内部操作符

种　类	内部操作符	应用场合
数值操作符	**、*、/、+、-、一元 +、一元 -	整数，实数或复数
字符操作符	//	任何长度的字符串
关系操作符	.EQ.、.NE.、= =、/ =、.GT.、.GE.、.LT.、.LE.、>、> =、<、< =	整数、实数之间，或字符型数据之间；操作的结果是逻辑量；.EQ. 和.NE. 也适用于复型数据
逻辑操作符	一元 .NOT.、.AND.、.OR.、.XOR.、.EQV.、.NEQV.	逻辑型或整型值

一元操作符只有一个操作数，二元操作符要求有两个操作数，例如：

```
-(a+b)        ! - 是一元操作符，+是二元操作符
```

除了内部操作符以外，用户还可以在函数和接口块中定义操作符。

在表达式引用操作数之前必须给操作数赋值。不能使用未定义的操作，例如下面的操作是禁止的：

- 被 0 除；
- 在指数为 0 或负数时以 0 为底数；
- 在指数为非整数时以负数为底数。

编译系统没有必要计算表达式所有部分的值。例如，如果下面的连乘的表达式含有 0，编译系统就不会计算括号内的表达式：

```
(37.8 / scale**expo + factor) * 0.0
```

类似地，如果一个逻辑与表达式中有“假”，则表达式就不必全部计算。例如下面的表达式中，左边括号内的表达式值为假，则右侧表达式就不予计算：

```
((3 .LE. 1) .AND. (switch .EQ. on))
```

如果一个表达式中有多个操作符出现但没有分隔的括号时就存在操作符的优先级的问题。操作符的优先级按递减的顺序列在表 2.5 中。

例如，下面的两个表达式是等价的，因为一元操作符比指数操作符优先级低。

表 2.5　　操作符优先级

操作符类型	操作符	相同优先级时的顺序
	用户定义的一元操作符	N/A
数值	**	从右到左
数值	* 或 /	从左到右
数值	一元 + or -	N/A
数值	二元 + or -	从左到右
字符	//	从左到右
关系	.EQ., .NE., .LT., .LE., .GT., .GE., = =, / =, <, < =, >, > =	N/A
逻辑	.NOT.	N/A
逻辑	.AND.	从左到右
逻辑	.OR.	从左到右
逻辑	.XOR., .EQV., or .NEQV.	从左到右
逻辑	用户定义的二元操作符	从左到右

```
-a**2
-(a**2)
```

2.3.2 创建表达式

表达式由操作数、操作符和括号组成。括号中包含的表达式可以认为是一个操作数，下面是一些表达式的例子：

```
a + b
(a - b) * c
a ** b
c .AND. d
f // g
```

表达式的数据类型由其操作符和操作数决定。结果的形态（例如数组各个维数的大小）由表达式中的操作符和数组的形态决定。表达式结果的数据类型即可以是内部数据类型也可以是派生数据类型。

如果表达式包含两个相同类型的操作数但种别（见第五章）不同，则结果会有较高的精度。例如，如果一个单精度的数加上另一个双精度的数，结果将是双精度。对所有二元操作，如果操作数是数组则它们的形态应该相同，或都是标量。

表达式可以是标量，例如：

```
q + 2.3 + r
```

或数组表达式，例如若 a 和 b 被声明为 a(10)和 b(10)：

```
a + b
```

如果一个操作数是标量而另一个是数组时，则认为标量是和数组形态相同的数组，其中每个元素都和标量相同，例如：

```
a + r
```

在上面的例子中，如果 r 是标量，a 是数组，则 r 将加到 a 的每个元素上。

2.3.3 数值表达式

由数值表达式可以求出整数、实数、复数或这些类型的数组。数组表达式由下列操作数组成：数值常量、命名常量、常量子对象、数组引用、数组元素引用、函数引用、派生类型引用和结构成员。

数值操作数可以是不同的数据类型和种别，结果的类型和种别由其定义决定。在操作之前所有操作数都将先转换为其中最高的精度。例如，若 a 为单精度，r 为单精度，b 为双

精度，则表达式：

```
r=a + b
```

在计算之前将先把 a 转换成双精度，而结果仍然是 r 的类型，即单精度。但相加的过程中会产生非零位，这意味着尽管精度提高了而准确度没有改进。例如下面的程序：

```
PROGRAM datconv
 ! demonstrate data precision when converting types
REAL(4) a
REAL(8) c
a = 1.11
c = a
WRITE (*,10) a
WRITE (*,20) c
10  FORMAT (' a:', F9.7)
20  FORMAT (' c:', F18.16)
END
```

输出结果为：

```
a: 1.1100000
c: 1.1100000143051147
```

在这个例子中，声明 a 为单精度（4 字节），c 为双精度（8 字节），尽管 c 有更高的精度但准确度并没有改进，如果用户对准确度要求较高则应该在声明时就设为双精度。

当一个整数被另一个整数除时，只返回商的整数部分，例如，7/3 的值为 2，(-7)/3 的值为-2，9/10 和 9/(-10) 的值都为 0。下面的表达式：

```
i = 1/4 + 1/4 + 1/4 + 1/4
```

的值为 0。

2.3.4 字符表达式

字符表达式会产生字符型的值，在字符表达式中有六种操作数：字符常量，字符变量引用，字符数组引用，字符函数引用，子字符串和字符结构成员引用。字符操作数可以是字符，可打印字符或任何长度的字符串。

字符操作符只有一个，即（//）。表达式 first // second 的结果为两个字符串的组合。例如表达式'AB '//'CDE'的结果是字符串：

```
'ABCDE'
```

若连接两个字符串变量，产生的字符串中的空格并不省略，例如：

```
CHARACTER*10 first
CHARACTER*6 second
 first = 'heaven'
 second = '  sent'
 WRITE (*, *) first//second
```

结果为：

```
 heaven      sent
```

注意在 heaven 和 sent 之间有六个空格。

函数 **LEN** 可以获得字符串的长度，**LEN_TRIM** 可以得到忽略空格后字符串的长度。

如果连接了 C 语言中的字符串，应该注意 C 语言中字符串末尾自动加入了空字符(\0)，例如表达式：

```
'hello 'C // 'world'C
```

等价于下面的 C 字符串：

```
'hello \0world'C
```

并且 C 编译器只把这个字符串当作'hello'，因为 C 语言中字符串以\0 为结束标志。

字符表达式可以不被完全赋值，例如：

```
CHARACTER (LEN = 2) c1, c2, c3, cf    ! cf 是字符函数
c1 = c2 // cf (c3)
```

其中，函数 cf 每有必要赋值，因为 c1 和 c2 长度相等，c1 可以完全由 c2 决定。

2.3.5 关系表达式

关系表达式用来比较两个数值型或字符型表达式的值。在标准 FORTRAN 90 中字符变量、数值变量和逻辑变量之间不能相互比较。而在 Visual FORTRAN 中数值表达式可以和字符表达式相比较，这时数值表达式被当作一个字符表达式（即一系列字节值）。例如，如果一个 4 字节整数 Int 被赋值为'A'，则 Int 的最低字节为 A 的 ASCII 码值（41h），其它

字节为 0。如果 A 赋值给一个 4 字节长的字符串 Chr，则因为字符变量是左对齐的，所以 Chr 的最高字节为 41h，这时显然 Chr 要大于 Int，尽管给它们赋的值是相同的。

关系表达式的运算结果是逻辑型(**.TRUE.** 或 **.FALSE.**)，FORTRAN 90 的关系操作符有了两中写法，可以直观地使用习惯的关系符号，如表 2.6 所示。它们对大小写是敏感的。

表 2.6 **关系操作符**

操作符	.LT.、<	.LE.、<=	.EQ.、==	.NE.、/=	.GT.、>	.GE.、>=
关系操作	小于	小于或等于	等于	不等于	大于	大于或等于

关系操作符都是二元操作符。关系表达式不能再包含关系表达式，下面的程序段是非法的：

```
REAL(4) a, b, c, d
IF ((a .LT. b) .NE. c) d = 12.0 ! 非法表达式
```

复数的关系只有两种：相等（**.EQ.**, ==）或不等（**.NE.**, /=）。

字符操作数的关系表达式是比较对应位置字符的 ASCII 码值，例如表达式('apple'<='banana') 返回**.TRUE.**，而表达式 ('Keith'>='Susan')返回**.FALSE.**。如果长度不同的两个字符串进行比较则较短的字符串将在右侧补上空格来扩展成和较长的字符串相同的长度。

2.3.6 逻辑表达式

逻辑表达式会产生逻辑量，在逻辑表达式中有七种操作数：逻辑常量、逻辑变量引用、逻辑数组元素引用、逻辑函数引用、关系表达式、*整型常量或变量*和逻辑结构成员引用。

逻辑操作符如图 2.7 所示。当优先级相同时按从左到右的顺序进行计算。

表 2.7 **逻辑操作符**

操 作 符	操 作	优 先 级
.NOT.	逻辑否	1(最高)
.AND.	逻辑与	2
.OR.	逻辑或	3
.XOR.	*逻辑非*	*4*
.EQV.	等	4
.NEQV.	不等	4

其中**.AND.**, **.OR.**, **.XOR.**, **.EQV.**, 和**.NEQV.**是二元操作符，**.NOT.**是一元操作符，例如若 switch 是 **.TRUE.**，则(.NOT. switch) 是 **.FALSE.**。 **.NOT.** 操作符可以在任何其它的逻辑操作符前出现，但两个**.NOT.** 不能连续使用，例如下面的语句是合法的：

```
logvar = a .AND. .NOT. b
```

标准逻辑操作符仅允许对 **LOGICAL** 型参数进行操作。Visual FORTRAN 允许整数作为参数，可以是整型常量，整型变量，整型结构成员或整型表达式。操作是“逐位”进行的，例如表达式：k .XOR. m 进行按位异或比较。异或是指相应的位相同则结果为 0（假），不同则结果为 1（真）。例如下面的程序：

```
PROGRAM exclus_OR
 ! demonstrate bit exclusive-OR
      INTEGER(2) L1,L2,L3
          L1= 0
          L2= 0
          L3= L1.XOR.L2
          WRITE (*, *) '0.XOR.0=',L3

          L1= 0
          L2= 1
          L3= L1.XOR.L2
          WRITE (*, *) '0.XOR.1=',L3

          L1= 1
          L2= 0
          L3= L1.XOR.L2
          WRITE (*, *) '1.XOR.0=',L3

          L1= 1
          L2= 1
          L3= L1.XOR.L2
          WRITE (*, *) '1.XOR.1=',L3

END
```

输出结果为：

```
0.XOR.0=       0
0.XOR.1=       1
1.XOR.0=       1
1.XOR.1=       0
```

下面的例子说明了逻辑表达式中运算的优先级：

```
LOGICAL stop, go, wait, a, b, c, d, e

!   下面两个表达式是相等的:
    stop = a .AND. b .AND. c
    stop = (a .AND. b) .AND. c

!   下面两个表达式是相等的:
    go = .NOT. a .OR. b .AND. c
    go = (.NOT. a) .OR. (b .AND. c)

!   下面两个表达式是相等的:
    wait = .NOT. a .EQV. b .OR. c .NEQV. d .AND. e
    wait = ((.NOT. a) .EQV. (b .OR. c)) .NEQV. (d .AND. e)
```

下面的例子说明了整数在逻辑表达式中实现字节屏蔽:

```
INTEGER(2) lowerbyte, dataval, mask
mask = #00FF      ! 屏蔽高为字节
dataval = #1234
lowerbyte = (dataval .AND. mask)
WRITE (*, '(1x, 2Z4)') dataval, lowerbyte
```

输出为:

```
1234  34
```

表 2.8 中列出了一些逻辑表达式的值。

表 2.8 逻辑表达式的值

a 和 b 的值	各表达式的值			
	a .AND. b	a .OR. b	a .EQV. b	a .XOR. b 或 a .NEQV. b
均为 true	True	True	True	False
一个为 true，另一个为 false	False	True	False	True
均为 false	False	False	True	False

2.4 源程序书写格式

由于早期 FORTRAN 源程序的书写受硬件限制，所以格式死板。FORTRAN 90 标准的格式灵活的多，但为了保证程序的兼容性还是有必要对源程序的书写格式进行说明。另外，

还对 Visual FORTRAN 的 Tab 格式进行了简单介绍。

2.4.1 概述

FORTRAN 90 源程序是由若干个程序单元块构成。一个程序单元或是一个主程序，就是一个辅程序。主程序单元整控制作用，各辅程序单元各自完成问题中的一个算法。

在 FORTRAN 90 中有两种源程序形式：自由源程序和固定源程序。两种形式都不区分大小写。有一些字符是源代码的标志符（除非它们出现在注释*或 Hollerith 常量*或字符常量中）。下面是全部源文件形式都适用的标志符：

- 注释标志符

  注释可以出现在程序单元的任意位置；当注释标志出现后，注释一直延续到该行末尾；如果一行都是空格也是注释行；注释对程序单元的编译没有影响。

- 语句分隔符

  源代码的一行可以有多个被语句分隔符分开的语句。分隔符为分号（;）多个连续的分号只认为是同一个分号。如果分号是一行最后一个字符或是在注释前的最后一个字符则将被忽略。

- 续行符

  一个语句可以分写为几行，中间用续行符连接；Digital FORTRAN 允许在一个源文件中最多续 511 行；注释行可以出现在续行中，注释行不能续行。

表 2.9 总结了各标志符的用法。

表 2.9　　标志符

源文件项	标志符 [1]	源文件格式	位　置
注释	!	所有格式	源代码的任何位置
注释行	!	自由格式	在一行开始处
	!, C, 或 *	固定格式	第一列
		Tab 格式	*第一列*
续行符 [2]	&	自由格式	在一行末尾处
	除了 0 或空格的其它任何字符	固定格式	第 6 列
	除了 0 的任何数字	*Tab 格式*	*在第一个 tab 后*
语句分隔符	;	所有格式	同一行的两个语句之间
声明标号	1 至 5 位十进制数	自由格式	在声明之前
		固定格式	在第 1 至第 5 列
		Tab	*在第一个 tab 之前*
调试声明 [3]	*D*	*固定格式*	*第一列*
		Tab	*第一列*

- 声明标号

声明标号（或语句标号）标记语句使其它语句可以涉及（获得信息或转移控制）到该语句。标号可以在任何完整语句的前面。声明标号长度必须是 1 到 5 位十进制数；空格和开始的 0 将被忽略。全是 0 的标号是非法的；空语句不能被标号。被标号的 FORMAT 语句和可执行语句是唯一能被其它语句涉及的语句。在同一个范围单元的两个语句不能有相同的语句。

以.FOR 为扩展名的源文件的缺省格式是固定格式。可以通过以下三种方法来选择源文件的格式：

（1） 使用.F90 为源文件的扩展名。

（2） 使用/free 编译器选项。

（3） 在源文件编译指示中使用 FREEFORM。

源文件格式和每行最大长度可以用 FREEFORM, NOFREEFORM, 或 FIXEDFORMLINESIZE 指示在任何时候改变。格式的改变直到文件末尾或再次改变之前都是有效的。

2.4.2 自由格式

在自由格式中，语句在源文件中的位置不受限制，每行可以包含 0 到 132 个字符。在自由格式中空格是有意义的。表 2.10 说明了空格的使用要求：

表 2.10 空格的使用

可选的空格	必须要的空格
BLOCK DATA	CASE DEFAULT
DOUBLE COMPLEX	DO WHILE
DOUBLE PRECISION	IMPLICIT *type- specifier*
ELSE IF	IMPLICIT NONE
END BLOCK DATA	INTERFACE ASSIGNMENT
END DO	INTERFACE OPERATOR
END FILE	MODULE PROCEDURE
END FORALL	RECURSIVE FUNCTION
END FUNCTION	RECURSIVE SUBROUTINE
END IF	RECURSIVE *type-specifier* FUNCTION
END INTERFACE	*type-specifier* FUNCTION
END MODULE	*type-specifier* RECURSIVE FUNCTION
END PROGRAM	
END SELECT	
END SUBROUTINE	
END TYPE	
END WHERE	
IN OUT	
SELECT CASE	
GO TO	

FORTRAN 90 用字符“!”开头为注释行的注解说明，注解延续到源程序行的末端。注释符“!”也可以写在语句行的右边作为对该语句的注解。

在语句行书写语句，习惯上一行只写一个语句，一个语句最长不能超过 2640 个字符，语句结尾不可有标点符号。在不妨碍程序的可读性时，也允许一行中写多个语句，语句与语句间用分隔符“;”分开，最后一语句的后面仍不需标点符号。

在 FORTRAN 90 中没有标号区、续行区、正文区等限制，字符和语句可以从行内任何一列写起，允许在任何列处中断语句，中断处写一个继续符“&”，随后在以下任何列处继续书写完成。

2.4.3 固定格式和 Tab 格式

在固定源程序形式中，在一行内一个语句从何列写起是有限的。在 FORTRAN 95 中，固定格式被认为是应废除的项目，但是一方面考虑到很多原来的 FORTRAN 程序是以固定格式编写的，为了读懂这些程序应该了解一下固定格式；另一方面为了程序的可移植性也需要了解固定格式。

在固定格式和 *tab 格式*中语句的位置是有规定的。缺省状态下，语句可以写到第 72 列。这时超过 72 列的字符将被忽略并且没有任何警告信息。*用户可以指定编译器选项使语句可以写到第 132 列。*

除了字符上下文以外，空格是没有意义的，所以可以在程序中任意使用空格以获得最清晰的可读性。

- 注释符

  在固定格式和 tab 格式中，叹号（!）表示注释。它不许出现在第六列（第六列为连接符保留）。出现在第一列的字母 C（或 c），星号（*）或叹号（!）说明该行是注释行。

- 续行符

  在固定或 tab 格式中，继续的行由下面表示：

  对固定格式：任何出现在第六列的字符（除了 0 或空格，如果 0 或空格用来作为续行符则编译器认为该行为新的一行 FORTRAN 语句）。

  对 tab 格式：在第一个 tab 后的任何位（除了 0）。

  标准 FORTRAN 90 允许续 19 行，而 Digital FORTRAN 允许续 511 行。

  续行的语句标号区必须是空白的（*除了调试语句*）。

  当一个很长的字符常量或 *Hollerith 常量*行之间连续时可能会出现问题。可以使用连接符来避免这样的问题：

```
PRINT *, 'This is a very long character constant '//
+        'which is safely continued across lines'
```

可以使用同样的方法来初始化长字符常量或 *Hollerith 常量*，例如：

```
CHARACTER*(*) LONG_CONST
```

```
 PARAMETER (LONG_CONST = 'This is a very long '//
+ 'character constant which is safely continued '//
+ 'across lines')
 CHARACTER*100 LONG_VAL
 DATA LONG_VAL /LONG_CONST/
```

在使用续行的连接方法之前Hollerith 常量必须转换成字符常量。

- 调试符

在固定格式和 tab 格式中，语句标号区除了可以包含语句标号以外，还可以包含注释符，或调试符。字母 D 在源文件第一列出现时代表调试符。

2.4.4 所有格式都适用的格式

源文件的格式可以写成所有格式都能使用的形式。为此，必须遵守表 2.11 的限制。

表 2.11 通用格式

空格	认为是有意义的
语句标号	在第 1 至第 5 列
语句	从第 7 行开始
注释符	只使用!，位置在除了第 6 列的任意一列。
续行符	只使用&，位置在开始行的 73 列和 以后各个续行的第 6 列

下面的例子对于所有格式都是合法的。

```
列号:

12345678...                                                    73
-------------------------------------------------------------------

! Define the user function MY_SIN

      DOUBLE PRECISION FUNCTION MY_SIN(X)
        MY_SIN = X - X**3/FACTOR(3) + X**5/FACTOR(5)           &
     &           - X**7/FACTOR(7)
       CONTAINS
         INTEGER FUNCTION FACTOR(N)
          FACTOR = 1
          DO 10 I = N, 1, -1
 10       FACTOR = FACTOR * I
         END FUNCTION FACTOR
       END FUNCTION MY_SIN
```

第三章 数据类型

FORTRAN 90 中的数据类型十分丰富，其中包含科学计算中必不可少的复数类型。另外，FORTRAN 90 对数组的引用和操作也很有特色。本章介绍了 FORTRAN 90 中各种内部和派生数据类型以及类型属性，并对用途很广的数组和指针专门进行了详细介绍。

3.1 概述

一种数据类型有四个性质：

- 类型的名称

内部数据类型的名称是由 FORTRAN 语言预先定义的；派生数据类型是由类型定义语句定义的。

- 允许值的集合

每种数据类型都有有效值的集合。例如逻辑型数据只有两个可能的取值：**.TRUE.** 或 **.FALSE.**。整型或实型数据都有允许的取值范围。复型数据和派生数据类型的值集是它们各自组成部分值集的组合。

- 允许值（常量）的表示方法
- 一套用来处理和解释允许值的操作

变量的数据类型决定了对它可能的操作。用户可以通过定义操作和操作符来扩展内部数据类型。

数据对象是用数据类型的名称来定义的。一个数据对象是常数，变量，或常数、变量的一部分。一旦用户定义了某种派生类型就可以定义这种类型的对象。

子对象是已命名的对象的一部分。例如子对象可以是数组元素，结构的成员，或字符串的一部分。

3.2 内部数据类型

内部数据类型包括：

- 整型

INTEGER, INTEGER(1), INTEGER(2), INTEGER(4), 和 INTEGER(8) (仅存在于 Alpha 系统上)

- 实型

REAL, DOUBLE PRECISION, REAL(4), 和 REAL(8)

- 复型

COMPLEX, COMPLEX(4), *DOUBLE COMPLEX*, and COMPLEX(8)

- 字符型

CHARACTER[*n*] ， *n* 是字符串长度

- 逻辑型

LOGICAL, LOGICAL(1), LOGICAL(2), LOGICAL(4), 和 LOGICAL(8) (仅存在于 Alpha 系统上)

每种数值类型都包括 0，0 即不认为是正值也不认为是负值。有符号的零与无符号的零完全相等。

每个内部类型都有一个种别（KIND）参数，它进一步描述了数据类型。在数值类型中，种别参数描述了精度和十进制指数范围。对于字符型只有一种种别。各种数据类型的缺省种别参数如表 3.1 所示。

表 3.1　　各种数据类型的内存需求和缺省种别

数据类型	种　别	字　节	说　明
BYTE	1	1	和 INTEGER(1)相同
INTEGER	2, 4, 或 8	2, 4, 或 8	缺省字节为 4
INTEGER(1)	1	1	
INTEGER(2)	2	2	
INTEGER(4)	4	4	
INTEGER(8)	8	8	仅存在于 Alpha 系统上
REAL	4 或 8	4 或 8	缺省字节为 4
REAL(4)	4	4	
DOUBLE PRECISION	8	8	和 REAL(8)相同
REAL(8)	8	8	
COMPLEX	4 or 8	8 or 16	缺省字节为 8
COMPLEX(4)	4	8	
DOUBLE COMPLEX	8	16	和 COMPLEX(8)相同
COMPLEX(8)	8	16	
CHARACTER	1	1	CHARACTER 和 CHARACTER(1)相同， (1) 是种别参数而不是字符串长度
CHARACTER*len	1	len	len 是字符串长度；在 Intel CPU 上从 1 到 65535，在 Alpha CPU 上从 1 到 2**31-1
LOGICAL	2, 4, or 8	2, 4, or 8	缺省字节为 4
LOGICAL(1)	1	1	
LOGICAL(2)	2	2	
LOGICAL(4)	4	4	
LOGICAL(8)	8	8	仅存在于 Alpha 系统上

3.2.1 整型数据

整型可以定义为 INTEGER, INTEGER(1), INTEGER(2), INTEGER(4), 或 INTEGER(8)(仅存在于 Alpha 系统上)。*还可以指定为 INTEGER*1, INTEGER*2, INTEGER*4, 或 INTEGER*8*。可以通过使用/integer_size:*size* 来改变整型的缺省长度。

整型定义语句的语法如下:

INTEGER [([KIND =] 种别值)] [[,属性列表] ::]实体表

内部查询函数 KIND 可以返回整型数据的种别参数; 内部查询函数 SELECTED_INT_KIND 可以找出提供指定整数范围的种别值; 内部查询函数 RANGE 可以返回指数范围。

表 3.2 列出了每种数据所占的字节和范围。

表 3.2 整型数据类型的字节和范围

数据类型(种别)	字 节	范 围
INTEGER(1)	1	有符号: -128 ~ 127 无符号: 0 ~ 255
INTEGER(2)	2	有符号: -32,768 ~ 32,767 无符号: 0 ~ 65535
INTEGER(4)	4	有符号: -2,147,483,648 ~ 2,147,483,647 无符号: 0 ~ 4,294,967,295
INTEGER(8) (仅存在于 Alpha 系统)	8	有符号: -9,223,372,036,854,775,808 ~9,223,372,036,854,775,807

下面的例子说明了整型变量的定义方式:

```
INTEGER, DIMENSION(:), POINTER :: days, hours
INTEGER(2), POINTER :: k, limit
INTEGER(1), DIMENSION(10) :: min
```

属性定义的例子:

```
INTEGER days, hours
INTEGER(2) k, limit
INTEGER(1) min
DIMENSION days(:), hours(:), min (10)
POINTER days, hours, k, limit
```

3.2.2 实型数据

实型数据以单精度和双精度来近似数学上的实数。这些数据也称为浮点数。

语句 **REAL** 可以定义实型变量或函数。单精度用 REAL(4)定义，双精度用 REAL(8)或 DOUBLE PRECISION 定义。

整型定义语句的语法如下：

REAL [([KIND =] 种别值)] [[,属性列表] ::]实体表

缺省的种别值为 4，可以通过使用/real_size:*size* 来改变实型的缺省长度。内部查询函数 KIND 可以返回实型数据的种别参数。内部查询函数 SELECTED_INT_KIND 可以找出提供指定精度和指数范围的种别值；内部查询函数 RANGE 可以返回指数范围。

下面的例子说明了实型变量的定义方式：

```
REAL (KIND = high), OPTIONAL :: testval
REAL, SAVE :: a(10), b(20,30)
```

属性定义的例子:

```
REAL (KIND = high) testval
REAL a(10), b(20,30)
OPTIONAL testval
SAVE a, b
```

表 3.3 列出了每种数据所占的字节和范围。

表 3.3　　实型数据类型的字节和范围

数据类型 (种别)	字节	精　度	范　围
REAL(4)	4	最左边的 7 位	负数大约从 -3.40282347E+38 至 -1.17549435E-38 0 正数大约从 +1.17549435E-38 至 +3.40282347E+38
REAL(8) 或 DOUBLE PRECISION	8	最左边的 15 位	负数大约从-1.7976931348623158D+308 至 -2.2250738585072013D-308 0 正数大约从+2.2250738585072013D-308 至 +1.7976931348623158D+308

下面的实型双精度常数都代表了 0.052 或 52/1000：

```
5.2D-2    +.00052E+2_8    .052D0 52D-3    .052_8    52.000E-3_dbl
```

其中参数 dbl 是在前面定义的用来指定双精度，即 PARAMETER dbl = 8。

3.2.3　复型数据

COMPLEX 或 COMPLEX(4) (COMPLEX*8)数据类型是一对有顺序的单精度实数。*DOUBLE COMPLEX* 或 COMPLEX(8) (COMPLEX*16) 数据类型则是一对有顺序的双精度实数。例如

```
COMPLEX(4) c
c = (3.0, 4.0)
```

其中前一个数是实部，后一个是虚部。实部和虚部的种别是相同的。单精度复数在内存中占 8 字节，双精度占 16 字节。由于复数是由两部分组成，所以在输入/输出时要给予特殊的注意。

复型数据的定义语法为：

COMPLEX [([KIND =]种别值)] [[, 属性列表] ::] 实体列表

复型数据定义的例子：

```
COMPLEX (4), DIMENSION (8) :: cz, cq
```

属性定义的例子:

```
COMPLEX(4) cz, cq
DIMENSION(8) cz, cq
```

复型常量的形式如下：

(c, c)

c 的取值为：对于单精度复型常量， c 是整型或 REAL(4)型常量；*对于双精度复型常量，c 是整型、REAL(4)型、或双精度(REAL(8))常量，其中至少有一个数据是双精度。*

下面是复型常量的例子：

```
(1.0, -1.0)      (4, 4.2E3)      (5.0_8, 8.3E9_8)
```

3.2.4 字符型数据

字符型数据的值集为字符。每个字符在内存中占一字节，定义字符的关键字是 CHARACTER，其定义的语法是：

CHARACTER [类型参数] [[, 属性列表] ::] 变量名

类型参数使用下面两种形式：

[LEN =] value

*character-length

value 和 character-length 可以是常量或命名常量。例如要定义字符串 last_name 的长度为 20：

```
CHARACTER (LEN=20)    last_name
```

下面的例子中，stri 的长度为 12，其它两个变量的长度保持为 8：

```
CHARACTER *8    strg, strh, stri*12
```

而下面的例子中，哑元 strh 在赋值时得到字符串的长度，其它两个变量长度保持为 8：

```
CHARACTER *8    strg, strh(*), stri
```

一般来说用户没有必要指定字符型数据的种别参数，因为只有一种字符类型。对于非英语国家的字符，可以用多字节字符集 MBCS（Several Multi-Byte Character Set）来处理。

字符串不能被分配（除非指定其固定长度为 0）。字符串数组和单一的一个字符串不同，用户可以定义元素长度一致的字符串数组。固定长度字符串数组可以被使用、访问、分配和释放。

如果定义了一个字符型的语句函数或语句函数的哑元，必须使用常量而不是星号*来说明字符串长度。星号只能用于下列情况：

- 定义过程的哑元。哑元将获得相关实际参数的长度。
- 定义一个命名常量。
- 为外部函数定义一个可变的结果变量。这种情况下，任何调用该函数的程序单元必须说明长度参数而不是星号，当函数被调用时，结果变量的长度就是长度参数指定的长度。

下面的例子说明了定义已知长度的字符串：

```
CHARACTER*32 string
CHARACTER string*32
CHARACTER string*(const+5)
```

下面的例子则说明了如何定义未知长度的字符串：

```
CHARACTER string*(*)
CHARACTER*(*) string
```

尽管单独的字符串是一个标量，但用户仍然可以使用和改变字符串的一部分，这部分称为子字符串。子字符串是字符串中的连续部分，可以用类似引用数组的方式来定义和引用。

表示子字符串的语法如下：

[first-position] : [end-position]

缺省的开始位置是 1，缺省的结束位置是字符串的长度。如果开始位置大于结束位置则子字符串的长度为 0。下面的例子说明了子字符串的使用：

```
CHARACTER(10) string
CHARACTER(5) substring
CHARACTER(1) char
string = "Jane Doe "
substring = string(:5)     ! 返回 'Jane '
substring = string(6:)     ! 返回 'Doe '
substring = string(3:7)    ! 返回 'ne Do'
substring = string(6:6)    ! 返回 'D '
n = 7
char = 'abcdefghijk'(n:n) ! 返回 'g', 第 n 个 (7) 字符
```

字符串可以是数组，子字符串也可以是数组，例如：

```
CHARACTER(8) A(5)          ! 包含 5 个元素，每个元素都
                           ! 含有 8 个字符.
CHARACTER(3) B(5)          ! 包含 5 个元素，每个元素都
                           ! 含有 8 个字符
CHARACTER(5) substring
substring = A(3)(2:6)      ! 返回 A 的第 3 个元素由
                           ! 第 2 至第 6 个字符组成的字符串
B(1:2) = A(4:5)(1:3)       ! 把 A 的第 4 个元素由
                           ! 第 1 至第 3 个字符组成的字符串
                           ! 赋给 B 的第一个元素并
                           ! 把 A 的第 5 个元素由
                           ! 第 1 至第 3 个字符组成的字符串
                           ! 赋给 B 的第二个元素
```

字符串常量以撇号为界定符。长度为 0 的字符串可以由两个连续的撇号表示。在字符串中可以有空格和 tab，并且它们是有意义的。字符串区分大小写。*用户还可以用 C 语言中的字符串来定义不可打印字符。*

表 3.4 列出了一些字符常量的例子：

表 3.4　　字符常量的值

字符串	所表示的常量
'String'	String
'1234!@#$'	1234!@#$
'Blanks count'	Blanks count
''	
'Case Is Significant'	Case Is Significant
' '	
"""Double"" quotes count as one"	"Double" quotes count as one

如果字符串有续行则将包括所有剩下的空格。下面的方法可以避免这种情况：

```
Heading (secondcolumn) = 'Acceleration of particles '//
&'from Group A'
```

在自由格式下可以写成：

```
Heading (secondcolumn) = 'Acceleration of particles ' &
& // 'from Group A'
```

字符串转换成数值类型数据（如整型，复型或实型）是一种有用的功能。可以用内部 **READ** 和 **WRITE** 语句来完成转换，例如下面的例子把字符串转换成实数：

```
PROGRAM testget
REAL x
CHARACTER (50) text
text = "12345.67"
READ (text,*) x
WRITE (*,*) x
END
```

3.2.5　逻辑型数据

LOGICAL 语句定义变量和函数为逻辑型。逻辑型变量可以是LOGICAL, LOGICAL(1), LOGICAL(2), LOGICAL(4)或 LOGICAL(8) (仅存在于 Alpha 系统)。*还可以这么定义：LOGICAL*1, LOGICAL*2, LOGICAL*4, 或 LOGICAL*8。* 缺省种别为 4。

逻辑类型数据的定义语法如下：

LOGICAL [([KIND =] 种别值)] [[, 属性列表] ::] 实体列表

定义变量的例子如下：

```
LOGICAL, ALLOCATABLE :: flag1, flag2
LOGICAL (KIND = byte), SAVE :: doit, dont
```

定义属性的例子如下：

```
LOGICAL flag1, flag2
LOGICAL (KIND = byte) doit, dont
ALLOCATABLE flag1, flag2
SAVE doit, dont
```

逻辑常量只有真和假两种值，逻辑常量的定义如下：

.TRUE.[_*k*]

.FALSE.[_*k*]

k 是种别参数，注意要有下划线(_)。

逻辑数据的范围和相同种别的整型数据的范围一致，例如 **LOGICAL**(2) 的范围和 **INTEGER**(2)的范围是一样的。

3.3 派生数据类型

在一些复杂情况下，单纯使用内部数据类型会使程序显得十分繁琐。为了便于用户更好地组织数据，FORTRAN 90 提供允许定义和使用派生数据类型。

3.3.1 派生数据类型

用户可以由内部数据类型或其它派生数据类型来创建派生数据类型。派生数据类型应该有一个类型名称，它不能和任何内部数据类型或已定义的派生数据类型重名。定义一种派生数据类型后就可以用它来定义变量、命名常量或其它派生类型了。派生数据类型的标量称为结构（structure）。

当 **TYPE** 语句后有类型名称时，**TYPE ... END TYPE** 将定义一个派生类型而不是变量。下面的例子定义了一个简单的派生类型 member，它包含一个名为 age 的 4 字节整数和名为 name 长度为 20 的字符串：

```
TYPE member
INTEGER age
CHARACTER (LEN = 20) name
END TYPE member
```

用上面例子定义的数据类型定义变量的例子如下：

```
TYPE (member) :: george
```

这样变量 george 就有 member 类型的特征，有 age 和 name 两部分。引用结构的成员可以按成员排列的顺序，中间以*%或(.*分隔。例如要引用 george 的 age 可以写成：george%age *或 george.age* 的形式。

系统不一定会按定义时的顺序储存各个成员，除非定义中包含了 **SEQUENCE** 语句。

用户还可以定义派生数据类型数组。下面的例子中，a 和 b 都是派生数据类型 pair 型的数组：

```
TYPE (pair)
 INTEGER i, j
 END TYPE
TYPE (pair), DIMENSION (2, 2) :: a, b(3)
```

派生数据类型的成员还可以是其它派生数据的组成部分，例如可以定义两个派生数据如下：

```
TYPE employee_name
  CHARACTER(25) last_name
  CHARACTER(15) first_name
END TYPE
TYPE employee_addr
  CHARACTER(20) street_name
  INTEGER(2) street_number
  INTEGER(2) apt_number
  CHARACTER(20) city
  CHARACTER(2) state
  INTEGER(4) zip
 END TYPE

 TYPE employee_data
   TYPE (employee_name) :: name
   TYPE (employee_addr) :: addr
   INTEGER(4) telephone
   INTEGER(2) date_of_birth
   INTEGER(2) date_of_hire
   INTEGER(2) social_security(3)
   LOGICAL(2) married
   INTEGER(2) dependents
 END TYPE
```

Visual FORTRAN 支持非标准 FORTRAN 90 的用户定义结构。结构和派生数据类型相似，不同的是用 **STRUCTURE** 来定义。结构中可以包含映射元素。映射指定一个或多个变量储存在连续内存空间里。变量类型是任意的，包括其它结构。映射只能在 **UNION** 块中出现，并且 **UNION** 只能出现在 **STRUCTURE** 中，例如：

```
STRUCTURE /test/
   UNION
     MAP
```

```
      INTEGER(4) j, k, m
      CHARACTER(21) name
    END MAP
    MAP
      REAL(4) a, b, c
      COMPLEX(8) z
    END MAP
  END UNION
END STRUCTURE
```

在上例中，4 字节整数 j，k 和 m 先出现在内存中，紧接着是长度为 21 的字符串变量 name。4 字节实数 a，b 和 c 与 j，k 和 m 分别共享同一段内存，8 字节复型变量 z 和 name 的前 8 个字符共享同一段内存。

当 **UNION** 中结合了映射以后，变量之间就像 **EQUIVALENCE** 语句一样可以互相覆盖。下例出自 DFLIB.F90 模块文件(在\DF\INCLUDE 子目录)，它对同一段内存定义了多种形式，每种都包含 3 个连续变量。因为各组变量的字长不同，所以较短的变量不会完全占据 **UNION** 中最长映射的空间。

```
STRUCTURE /MTH$E_INFO/
 INTEGER*4 ERRCODE          ! INPUT : One of the MTH$ values in DFLIB
 INTEGER*4 FTYPE            ! INPUT : One of the TY$ values  in DFLIB
 UNION
 MAP
    REAL*4 R4ARG1           ! INPUT: First argument
    CHARACTER*12 R4FILL1
    REAL*4 R4ARG2           ! INPUT: Second argument (if any)
    CHARACTER*12 R4FILL2
    REAL*4 R4RES            ! OUTPUT: Desired result
    CHARACTER*12 R4FILL3
END MAP
MAP
    REAL*8 R8ARG1           ! INPUT: First argument
    CHARACTER*8 R8FILL1
    REAL*8 R8ARG2           ! INPUT: Second argument (if any)
    CHARACTER*8 R8FILL2
    REAL*8 R8RES            ! OUTPUT: Desired result
    CHARACTER*8 R8FILL3
END MAP
MAP
```

```
        COMPLEX*8 C8ARG1        ! INPUT: First argument
        CHARACTER*8 C8FILL1
        COMPLEX*8 C8ARG2        ! INPUT: Second argument (if any)
        CHARACTER*8 C8FILL2
        COMPLEX*8 C8RES         ! OUTPUT: Desired result
        CHARACTER*8 C8FILL3
    END MAP
    MAP
        COMPLEX*16 C16ARG1      ! INPUT: First argument
        COMPLEX*16 C16ARG2      ! INPUT: Second argument (if any)
        COMPLEX*16 C16RES       ! OUTPUT: Desired result
    END MAP
    END UNION
    END STRUCTURE
```

3.3.2 派生数据类型的缺省初始化

如果派生数据类型的成员定义中出现了初始化则将出现缺省初始化。(这是 FORTRAN 95 的特性)。即使成员的定义为 PRIVATE 指定的初始化也将生效。派生类型的显式初始化会覆盖缺省的初始化值。

为了确定一个为数组的成员的缺省初始化值可以用包含下列两种类型之一的常量表达式：数组赋值器或一个单独的标量，它可以获得每个数组元素的值。

指针联合状态可以是“已联合（associated）”，“未联合（disassociated）”或“未定义（undefined）”。如果指定没有缺省状态，指针的状态就是“未定义”。如果要设置成“未联合”可以使用=>NULL( )。

用户没有必要为派生类型的每一个成员指定初始化值，例如：

```
TYPE REPORT
  CHARACTER (LEN=20) REPORT_NAME
  INTEGER DAY
  CHARACTER (LEN=3) MONTH
  INTEGER :: YEAR = 1995        ! Only component with default
END TYPE REPORT                 !      initialization
```

下面这条语句：

```
TYPE (REPORT), PARAMETER :: TODAYS_REPORT = REPORT (15, "NOV", 1996)
```

这时，显式的初始化声明覆盖了 TODAYS_REPORT 的 YEAR 成员。成员缺省的初始化值也可以由类型定义时指定的缺省初始化覆盖，例如：

```
TYPE MGR_REPORT
    TYPE (REPORT) :: STATUS = TODAYS_REPORT
    INTEGER NUM
  END TYPE MGR_REPORT

  TYPE (MGR_REPORT) STARTUP
```

这时，STARTUP 类型的 STATUS 成员从 TODAYS_REPORT 中获得初始值，覆盖了 YEAR 成员的初始值。

3.3.3 派生类型的值

指定的派生类型的允许取值由其最后的成员的允许取值决定。当用户定义一种派生类型时会隐式地创建相应的结构赋值器，它允许派生类型的标量值从一系列值中构造，每一个对应一个成员。相对于为每个成员分别使用赋值语句，这种方法显得更为有效和简便。例如已经定义了 n 个成员则结构赋值器可以有以下形式：

type-name (expr1, expr2, ... , expr*n*)

括号中每一个表达式都给定了一个成员值，它们必须以数据类型定义中的顺序给出，并且各自的允许值也需要相符。下面是一个用赋值器赋值的例子：

```
george = member ( 33, 'George Brown' )
```

还可以用赋值器 PARAMETER 属性或语句来创建命名的派生数据对象常量。例如：

```
TYPE pair
   INTEGER i, j
 END TYPE
 TYPE (pair) p
 PARAMETER (p = pair (9, 2))
 TYPE (pair), PARAMETER :: q = pair (7, 3)
```

如果有成员是数组，指定成员值时需要数组赋值器。

3.4 数据属性

一般来说，数据属性描述了一个数据对象如何在程序中应用。用户可以使用一个或多

个语句来规定数据对象的属性，但每个属性只能声明一次。

Visual FORTRAN 的数据属性如表 3.5 所示：

表 3.5　数据属性

属性名称	描述	适用范围
ALLOCATABLE	说明数组的大小将由 ALLOCATE 语句执行时决定	数组
AUTOMATIC	*在堆栈而不是内存中声明变量*	*变量*
DIMENSION	说明数组	常量或变量
EXTERNAL	声明外部函数的名称	函数或例行子程序
INTENT	说明函数哑元的用意	函数或例行子程序哑元
INTRINSIC	声明一个为内部函数	函数或例行子程序
OPTIONAL	允许一个函数被调用时省略这个哑元	函数或例行子程序哑元
PARAMETER	声明一个名称为常量	数据对象常量
POINTER	声明一个数据对象为指针	变量
PRIVATE	把模块中的实体访问限制在本模块内	常量、变量或模块程序
PUBLIC	把模块中的实体可以被外部模块使用	常量、变量或模块程序
SAVE	在定义变量的子程序执行之后保持变量的值，定义、联系和分配状态	变量或公共块
STATIC	说明一个变量以静态存储	变量
TARGET	声明一个数据对象为目标	变量
VOLATILE	*声明一个对象完全不可预测并使其在编译时不受优化*	*数据对象或公共块*

3.4.1　参数（PARAMETER）属性和语句

命名常量是在类型声明中用 **PARAMETER** 语句包含 PARAMETER 属性。它们在程序执行时不能被改变。

PARAMETER 可以用于属性和语句格式。定义属性的 **PARAMETER** 语句形式如下：

PARAMETER *[(]*named-constant = 初始化表达式 *[, ...] [ )]*

如果在 **PARAMETER** 语句中的命名常量以隐式出现，任何包含该常量的后来的类型声明语句要注意其隐含的值。

中括号的有无可以由/[no]altparam 编译选项来确定，默认选项是/altparam,这时的定义例如：

```
PARAMETER (xmax = 50.)
```

若选为/noaltparam 则可以写成：

```
PARAMETER y = 2.1*2.0
```

在定义实体的类型声明语句中 PARAMETER 格式为：

类型说明,PARAMETER [, 属性] :: 命名常量 = 初始化表达式

其中命名常量必须在本语句或前面的类型声明语句中定义，或者是应该是通过宿主或使用联合可以访问的。

在一条类型声明语句中可以声明多个命名常量：

```
REAL, PARAMETER :: xmax = 50., y = 2.1*2.0
```

3.4.2 公共（PUBLIC）与个别（PRIVATE）属性和语句

具有 PUBLIC 属性的实体可以被其它程序单元用 USE 语句访问。而具有 PRIVATE 属性的实体不能被外部模块访问。实体的缺省属性是 PUBLIC。使用访问说明语句格式为：

PUBLIC | PRIVATE [[::]访问识别列表]

例如：

```
MODULE EX
  PRIVATE
  PRIVATE :: x, y, z
  PUBLIC  :: a, b, c, assignment (=), operator (+)
```

类型声明语句中使用访问说明语句格式为

类型说明, PUBLIC | PRIVATE [, 属性列表] ::实体列表

实体列表可以是已命名的变量，子程序，派生类型，命名常量或名称列表组，例如：

```
REAL, PRIVATE :: x, y, z
REAL, DIMENSION (10, 10), PUBLIC :: a, b
```

如果一个对象已经被声明为 PRIVATE 则它就不能再声明为 PUBLIC。

访问属性只能在模块中使用。如果 **PUBLIC** 或 **PRIVATE** 语句后面没有实体列表则将在整个程序单元范围内设置默认访问属性。也就是说，模块中的每一个变量、常量和子程序都将拥有同样的默认访问属性，除非显式说明某个实体为非默认属性。显然，在一个程序单元中只能有一个没有实体列表的 **PUBLIC** 或 **PRIVATE** 语句。

一个模块中定义的派生类型在任何调用该模块的单元中是以缺省属性访问的。缺省属性可以改变以限制访问权限在模块内部，某种派生类型可以声明为 PRIVATE，或声明为 PUBLIC 但其中的某些成员为 PRIVATE。

3.4.3 保存（SAVE）属性和语句

SAVE 用来保留在各个过程中会改变值的变量值。有 SAVE 属性的对象在包含对它们

的声明的作用域单元中执行 **RETURN** 或 **END** 后会保留它们的联系状态、分配状态、定义状态和取值。其它任何程序使用 **USE** 语句访问模块时，其中有 SAVE 属性的对象在执行 **RETURN** 或 **END** 后会保留它们的性质。

用户可以在程序单元中用 SAVE 来保留所有的对象而不必跟实体列表，这时在同一作用域中不允许出现其它显式的 SAVE 属性或 SAVE 语句。要注意的是，SAVE 属性不能用于公共块、哑元、过程、函数结果的对象或自动数据对象。SAVE 属性在主程序说明部分中也是无效的。

在公共块中尽管不能为单独的项目指定 SAVE 属性，但可以把整个公块保存下来，其中的所有对象都将被保存。如果一个公共块在主程序之外被声明为 SAVE，则它要在每一个出现的作用域中声明具有 SAVE 属性（除了主程序之外）。当执行 **RETURN** 或 **END** 后保存过的公共块的当前值可以用于下一个使用这个块的作用域单元。

SAVE 属性可以由类型定义语句或 **SAVE** 语句来指定。

SAVE 语句格式为：

SAVE [[::]实体列表]

实体列表包括对象名称和公共块名称，它们之间用/隔开。例如：

```
SAVE a, b, c, / blocka /, d
```

指定 SAVE 属性的格式为：

类型说明, SAVE [[, 属性说明] ::] 实体列表

例如：

```
REAL, SAVE :: a, b
```

3.4.4　静态（STATIC）属性和语句

STATIC 属性指定一个变量的存储类型为静态，就是说它在程序执行的过程中始终存在于内存里，它的值在调用包含的过程中会被保持。这种属性和 FORTRAN 属性以及 C 中的静态属性是等价的。STATIC 属性即可以用类型声明语句也可以用 **STATIC** 语句来声明。**STATIC** 语句的格式为：

STATIC [::] 变量名

例如：

```
STATIC :: a, b, c
```

变量名中可以包括对象名称和公共块名称，它们之间用/隔开。

指定 STATIC 属性的类型声明语句格式：

类型说明, STATIC [[, 属性说明] ::] 实体列表

例如：

```
INTEGER, STATIC :: d
```

3.4.5 自动（AUTOMATIC）属性和语句

AUTOMATIC 属性可以由类型声明或语句来指定。它指定变量在堆栈而不是静态内存中。在 Visual FORTRAN 里，所有变量在缺省状态下都是静态的。声明为 AUTOMATIC 的变量没有固定的内存地址，但在堆栈中分配一段存储单元。过程中的 AUTOMATIC 变量在执行完毕之后将被丢弃。**AUTOMATIC** 语句的格式为：

AUTOMATIC [[::] 实体列表]

实体列表由变量名称或数组定义组成，例如：

```
AUTOMATIC :: e, f, g
```

如果实体列表不存在则整个程序单元内的变量都将认为是自动的。

指定 AUTOMATIC 属性的格式为：

类型说明, AUTOMATIC [[, 属性说明] ::] 实体列表

例如：

```
INTEGER, AUTOMATIC :: h
```

3.4.6 用编译器指令指定属性

在标准 FORTRAN 90 之外 Visual FORTRAN 的 **ATTRIBUTES** 编译指令提供了一些附近特性。例如，这条指令允许用户使用调用 Microsoft C 的习惯，通过值或引用传递参数。**ATTRIBUTES** 的语法格式为：

c**DEC$ ATTRIBUTES** 属性列表 :: 对象列表

其中，

c 是 a，c， C， !，或*。

属性列表为要指定的属性，是下面选项的一个或多个：ALIAS, C, DLLEXPORT, DLLIMPORT, EXTERN, REFERENCE, STDCALL, VALUE, 或 VARYING。

对象列表是要声明具有属性列表中属性的对象或过程。

ATTRIBUTES 选项可以用于函数和子程序定义，类型声明以及 **INTERFACE** 和 **ENTRY** 语句。下面的例子中，go 在调用 happy 时使用了接口块中指定的 C 选项：

```
MODULE mod
 INTERFACE
   SUBROUTINE happy
 !DEC$ ATTRIBUTES C :: happy
   END SUBROUTINE
 END INTERFACE
 CONTAINS
   SUBROUTINE go
   CALL happy(12)
   END SUBROUTINE
 END MODULE
```

3.5 数组和指针

在 FORTRAN 90 中变量的概念与 FORTRAN77 的不同，它包含了两种变量，一种是标量，一种是数组。就像对一个简单变量操作赋值一样，FORTRAN 90 允许把整个数组作为一个操作数进行操作，也允许在赋值语句中对整个数组进行赋值。

数组是至少有一个维数的变量，数组可以是静态的也可以是动态的。如果数组是静态的，则在编译时就被分配了固定的储存空间，并且直到程序退出时才被释放。程序运行时静态数组的大小不能改变。而动态数组的储存在程序运行当中是可以分配、改变和释放的。动态数组只有两种：可分配数组和自动数组。

所有的指针都是动态变量，在用户程序运行时被分配储存空间。如果一个指针同时也是数组，则它的每一个维数像动态数组一样在程序运行当中都是可以改变的。指针可以指向数组标量或标量变量（即没有维数的变量）。只有当指针由 **ALLOCATE** 语句分配内存或直到它被指向一个已分配的目标时，指针才会被分配存储空间。

可分配数组和指针数组十分类似，只是可分配数组不能指向其它数组。自动数组和可分配数组也很类似，区别在于当用户开始或结束程序时自动数组会自动分配和释放内存。指针、可分配数组和自动数组的共同点是它们都是 FORTRAN 90 中的动态数据对象。

3.5.1 数组的性质和定义

数组的维数称为秩，数组中所有元素的个数称为数组的大小。在某一维中元素的个数称为该维的长度。数组的形状取决于秩和每一维的长度，例如：

```
REAL A(10, 3, 2)
```

数组 A 的秩为 3，大小为 10×3×2=60，形状为 10 乘 3 乘 2，或表示成（10，3，2）。

每一维都可以由一个下界和一个上界来指定，之间以冒号分开，例如：

```
REAL B(0:9, -1:1, 4:5)
```

声明数组时下界可以省略，对于某些数组上界也可以省略。缺省状态下，下界为 1，

但可以设成零或负整数，比 C 等的固定下界灵活。在上面的例子中，数组 A 和 B 的秩、大小和形状完全相同，数组 A 三个维的下界都是 1，数组 B 的下界分别为 0，－1 和 4。

数组义可以用类型声明，DIMENSION 语句，POINTER 语句或 ALLOCATABLE 语句来定义。例如，下面的例子是合法的数组声明：

```
REAL          A(10, 2, 3)          ! 类型声明
DIMENSION     A(10, 2, 3)          ! DIMENSION 语句
ALLOCATABLE   B(:, :)              ! ALLOCATABLE 语句
POINTER       C(:, :, :)           ! POINTER 语句
REAL, DIMENSION (2, 5) :: D        ! 和 DIMENSION 属性
                                   ! 一起作类型声明
REAL, ALLOCATABLE :: E(:, :, :)    ! 和 ALLOCATABLE 属性
                                   ! 一起作类型声明
REAL, POINTER :: F(:,:)            ! 和 POINTER 属性
                                   ! 一起作类型声明
```

无论何种情况，数组的秩总是要指定的。如果数组即不是可分配的，又不是指针或哑元，那么它必须声明秩、大小和形状。数组有四种形式：

1. 显式形状数组

这种数组指定了所有特征：固定的秩、每一维的长度和形状。其中下界是可以忽略的，例如：

```
INTEGER M(10, 10, 10)
INTEGER K(-3:6, 4:13, 0:9)
```

在函数或子程序中，数组的上界和下界可以由变量或表达式指定。在调用子程序时，上下界通过变量或表达式求出，在子程序后来变量或表达式值的变化不会对数组的上下界产生影响。例如：

```
SUBROUTINE EXAMPLE (N, R1, R2)
DIMENSION A (N, 5), B(10*N)
...
N = IFIX(R1) + IFIX(R2)
```

子函数被调用时，数组 A 和 B 的上界通过传入的变量 N 来确定，而以后 N 的值的变化对 A 和 B 的维数就不会有影响。使用变量或表达式的数组必须是哑元、函数结果或自动数组。在子程序中声明的数组是显式形状数组而不是哑元，并且它的上下界是不定的表达式。上面的例子中 A 和 B 都是自动数组。

2. 假定形状数组

这种数组是在函数或子程序中的哑元，它从实际传递过来的数组获得形状参数。假定形状数组的秩由冒号的个数确定，例如：

```
REAL A(:, :, :)
```

数组 A 的秩为 3，但每一维的长度待定。当子程序被调用时 A 将从传递到子程序的数组获得形状，例如：

```
REAL X (4, 7, 9)
...
CALL ASSUMED(X)
```

于是 A 获得了维数（4，7，9）。实际数组和假定形状数组的秩必须相同。

假定形状数组可以指定下界，这时数组的形状和传递给它的数组相同，但上界将会改变，例如：

```
SUBROUTINE ASSUMED(A)
REAL A(3:, 0:, -2:)
...
```

在这个子程序中调用的是同一个数组 X(4,7,9)，则数组 A 的上下界将是：

```
A(3:6, 0:6, -2:6)
```

应用假定形状数组为哑元的子程序必须有显式的接口。

3. 假定大小数组

这种数组是在函数或子程序中的哑元，它从实际传递过来的数组获得数组大小。除了最后一维的上界以外，其它所有特征（维数，长度和边界）都必须指定。声明一个假定大小数组时，最后一个上界用星号*表示，例如：

```
SUBROUTINE ASSUME(A)
REAL A(2, 2, *)
```

假定大小数组的秩和形状可以和实际传入的数组不同，传入的数组只确定它的大小。实际数组的元素按列传递给假定大小数组，假定大小数组也按列接收。接受的过程中假定大小数组的最后一维的长度会改变来接受所有传递进来的数组元素，于是最终给出数组的大小。例如上面例子中的 ASSUME 子程序，如果以数组 X 为哑元来调用：

```
REAL X(7)
CALL ASSUME(X)
```

数组 X 的元素与数组 A 的对应顺序是：

```
X(1) = A(1, 1, 1)
X(2) = A(2, 1, 1)
X(3) = A(1, 2, 1)
X(4) = A(2, 2, 1)
X(5) = A(1, 1, 2)
X(6) = A(2, 1, 2)
X(7) = A(1, 2, 2)
```

其中数组 A 的最后一维没有必要成为完整的维，所以数组 A 始终没有确定的形状。因为假定大小数组没有形状，所以这样的数组不能仅仅通过名称来向其它子程序传递（除了不需要形状的函数和子程序，例如内部函数 LBOUND）。

假定大小数组可以分解成确定的数组片段，但有时使用起来会显得冗长。例如上面例子中的数组 A，如果要确定其中的各个元素需要三个片段：

```
A(1:2, 1:2, 1) 和 A(1:2, 1, 2) 以及 A(1, 2, 2)
```

假定大小数组的秩是完全确定的维数加一。上例中数组 A 的秩为 3，尽管 A 的第三维不是完整的。

用户还可以指定假定大小数组任意维的下界（包括最后一维），例如：

```
SUBROUTINE ASSUME(A)
REAL A(-4:-2, 4:6, 3:*)
```

4. 迟形数组

迟形数组是数组指针或可分配数组。它每一维的长度只有在分配数组或指针时，或在指针和一个已分配目标联合时才被确定。声明迟形数组时，秩由冒号确定，但维数是未知的，例如：

```
REAL, ALLOCATABLE :: A(:, :, :)
REAL, POINTER :: B(:,:)
INTEGER, ALLOCATABLE, TARGET :: K(:)
```

可分配数组和指针数组必须以延迟形状的形式来声明。可分配数组可由下列方式声明：使用 **ALLOCATABLE** 语句、**DIMENSION** 语句、**TARGET** 语句或在类型声明中使用 ALLOCATABLE 属性。

指针数组可由下列方式声明：使用 **POINTER** 语句、**DIMENSION** 语句、**TARGET** 语句或在类型声明中使用 POINTER 属性。如果迟形数组以 **DIMENSION** 语句或 **TARGET** 语句声明，则在其它语句中必须给出 ALLOCATABLE 或 POINTER 属性，例如：

```
DIMENSION P(:, :, :)
POINTER P

TARGET B(:,:)
ALLOCATABLE B
```

在迟形数组的大小、形状和上下界没有确定之前，其任何部分都不能被引用（除了查询参数是否存在、联合状态或类型属性的内部查询函数）。如果迟形数组是可分配数组，它的大小、形状和边界在 ALLOCATE 语句执行时才被确定。如果迟形数组是指针数组，它的大小、形状和边界在 ALLOCATE 语句或指针赋值语句执行时，或在和已分配目标联合时才被确定。例如：

```
REAL, POINTER :: A(:,:), B(:), C(:,:)
INTEGER, ALLOCATABLE :: I(:)
REAL, ALLOCATABLE, TARGET :: D(:, :), E(:)
      ...
ALLOCATE (A(2, 3), I(5), D(SIZE(I), 12), E(98) )
C => D              ! 指针赋值语句 Pointer assignment statement
B => E(25:56)       ! 指针赋值给一个
                    ! 目标片段
```

3.5.2 数组元素和数组片段

标量是一个单独的数据对象，数组是标量的集合。数组中单独的标量称为元素。标量的秩为 0，而数组的秩至少是 1。在 Digital FORTRAN 中数组最大的秩可以为 7。

数组片段是数组元素的一个子集。数组片段的元素可以是数组中任意的元素，它们不需连续或遵循某个规则。数组中的所有元素和片段的数据类型和种别都相同。

部分或整个数组都可以在程序中引用。引用整个数组时使用数组名，例如：

```
REAL A (10), B(10)
A = 3.0       ! 把 A 的所有元素值置为 3
B = SQRT(A)   ! 把 B 的所有元素值置为 3 的平方根
```

通过下标可以引用数组的一部分。例如上例的两个数组，A(1)指数组 A 的第 1 个元素，B(3:4)指数组 B 的第 3 和第 4 个元素。如果没有下标则指整个数组。还可以用下标指定用户需要的特定的元素或片段。例如，B(1:10:2)指的是数组 B 的第 1、3、5、7、和 9 个元素。FORTRAN 90 的这种引用是和 Matlab 非常类似的。

1. 数组元素

由于计算机的内存是一维的，所以不管数组是几维它在内存中都是按一维来存储，与C语言不同的是，FORTRAN 的数组是按列存储的，即最左侧的下标先变动，最右侧的下标最后变动。例如数组 A(2,2,2)在内存中的储存顺序如下：

```
A(1, 1, 1)
A(2, 1, 1)
A(1, 2, 1)
A(2, 2, 1)
A(1, 1, 2)
A(2, 1, 2)
A(1, 2, 2)
A(2, 2, 2)
```

数组的下标必须用逗号隔开，下标是整型常量、变量或表达式。下标可正、可负，也可以为 0，但必须在引用的数组的维数之内。引用下标的个数要和声明的数组的维数一致。可以使用函数或数组元素作为下标：

```
REAL A(3, 3)
REAL B(3, 3), C(89), R
B(2, 2) = 4.5                          !给元素 B(2, 2)赋值 4.5
R = 7.0
C(INT(R)*2 + 1) = 2.0                  !C 的第 15 个元素为 2.0
A(1,2) = B(INT(C(15)), INT(SQRT(R)))   !元素 A(1,2)和元素 B(2,2)的值相同
```

2. 数组片段

用户可以访问和使用数组的元素的子集，称为数组片段。如果用户指定了数组的所有下标则得到的是数组元素（即标量），如果只指定部分下标则结果是部分数组元素的集合，称为数组片段。

数组片段本身也是数组。前面提到过，数组片段的元素可以是数组中任意的元素，也不需连续或遵循某个规则。例如，如果数组 A 声明为：

```
REAL A(2,3,4)
```

则 A(1,2,3)是数组元素，而 A(1:2,2,2)， A(1,1,4:2:-1)，和 A(1,2:3, (/2, 4/))都是数组片段。

数组片段由下标列表确定，下标列表有两种：三元下标和向量下标。

三元下标是用三个值分别代表数组片段的下界，上界和上下界之间的增幅（或步长）。例如：

```
REAL A(10)
A(3:5:2) = 1.0    ! 用三元下标将 A(3) 和 A(5)
                  ! 的值置为 1.0
```

三元下标的语法格式为：

[下界]:[上界][:步长]

如果省略下界缺省值为数组相应维的下界；如果省略上界则缺省值为数组相应维的上界；如果省略步长则缺省值为 1。如果下标都省略了则缺省片段为这个维的全长。例如：

```
REAL A(10)
A(1:5:2) = 3.0    ! 把元素 A(1)，A(3)，A(5) 置为 3.0
A(:5:2) = 3.0     ! 也是把元素 A(1)，A(3)，A(5) 置为 3.0
                  ! 因为缺省下界为 1
A(2::3) = 3.0     ! 把元素 A(2)，A(5)，A(8) 置为 3.0
                  ! 因为上界缺省值为 10
A(7:9) = 3.0      ! 把元素 A(7)，A(8)，A(9) 置为 3.0
                  ! 因为缺省步长为 1
A(:) = 3.0        ! 和 A = 3.0 相同，将 A 的所有元素置为 3.0
```

对于一个多维数组的数组片段，它的每一维都可以用三元下标来声明。如果要在一个语句或过程中引用这个数组片段，则引用下标要和声明时的下标个数一样多。注意，三元下标只算一个下标。例如：

```
REAL A(8, 3, 5)
A(1:2, 2:3, 4) = 3.0
```

数组 A 是三维数组，数组片段 A(1:2, 2:3, 4)有 3 个下标，第一个是三元下标(1:2)，它指定了数组 A 的第一维的第一个和第二个元素；第二个是三元下标(2:3)，它指定了数组 A 的第二维的第二个和第三个元素；第三个下标(4)不是三元下标，它指数组 A 第三维的第四个位置的元素。A(1:2, 2:3, 4)的各个元素为：

```
第 1 个元素  = A(1, 2, 4)        第 3 个元素  = A(1, 3, 4)
第 2 个元素  = A(2, 2, 4)        第 4 个元素  = A(2, 3, 4)
```

所以这个数组是二维数组，形状是(2, 2)。

数组的步长一定不能是 0。当步长为负值时，数组子片段从上界开始然后递减至下界，例如声明一个数组：B(10)，则数组片段 B(9:2:-2)是由元素 B(9)， B(7)， B(5)和 B(3)组成的数组。显然下界不能比上界大，否则产生的数组大小为 0。

三元下标的值可以不在数组的边界以内，例如对于数组 B(10)，数组片段 B(3:15:6)是由 B(3)和 B(9)组成的数组。

三元下标以上升或下降的顺序指定数组元素，而向量下标可以以任何顺序来指定数组元素。向量下标是一个一维整数数组（即向量），它可以从整个数组中选择片段，例如：

```
REAL A(10), B(5, 5)
INTEGER I(4), J(3)
! 定义向量 I.
I = (/5, 3, 8, 2/)
! 定义向量 J.
J = (/3, 1, 5/)
! 设置元素 A(5), A(3), A(8) 和 A(2) 的值为 3.0
A(I) = 3.0
! 设置元素 B(2,3), B(2,1) 和 B(2,5) 的值为 3.0
B(2,J) = 3.0
```

向量下标的值应该在定义的边界之内。向量下标可以有多个重复的值，这样的数组片段称为多对一（many-one）数组片段，例如：

```
REAL A(3, 3), B(4)
INTEGER K(4)
! Vector K has repeated values
K = (/3, 1, 1, 2/)
! Sets all elements of A to 5.0
A = 5.0
B = A(3, K)
```

数组片段 A(3,K)由下列元素组成：

```
A(3, 3) A(3, 1) A(3, 1) A(3, 2)
```

因为在 A(3,K)中有重复的元素，所以它是多对一数组片段。一个多对一数组片段不能出现在赋值语句的左端，也不能作为 **READ** 语句的输入项，因为这些结果取决于下标的计算，而对于多对一数组片段是不可预料的。

带有向量下标的数组片段不能出现在内部文件和指针目标中，如果一个子程序的

哑元数组被重定义，则带有向量下标的数组片段不能作为这个子程序的实际参数。带有向量下标的数组片段可以用于传递给只能引用的哑元，即只有 INTENT(IN)属性，而且值不改变、不被重定义的子程序。

3.5.3　数组赋值

数组的赋值可以使用一般的赋值语句或数组赋值器。数组赋值器是由括号和斜线之间的一系列值或隐 DO 循环。例如：

```
INTEGER A(6)
A(1) = 1
A = (/1, 2, 3, 4, 5, 6 /)      ! 数组赋值器给数组 A 赋值
```

数组构造器给数组 A 的六个元素分别赋值：A(1)=1, A(2)=2, A(3)=3 等等。 数组构造器的格式为：

(/取值列表/)

取值列表可以是标量，隐 DO 循环或任意秩的数组。其中的所有值的类型和种别都相同，以逗号隔开。如果其中列表中出现了数组，它的值是按列来赋的。如果用户想赋给数组的值不是向量，可以用函数 RESHAPE 把数据赋给相同形状的数组，例如：

```
INTEGER B(2,3), C(8)
! 赋值给形状为 (2,3) 的数组.
B = RESHAPE((/1, 2, 3, 4, 5, 6/), (/2,3/))
!在赋值给向量 C 之前先把 B 转换成向量
C = (/ 0, RESHAPE(B,(/6/)), 7 /)
```

RESHAPE 前一个向量是要改变形状的列表，后一个是表示形状的向量，它应该和等号左侧数组的形状相同。

赋值器的取值列表还可以包含隐 DO 循环，例如：

```
INTEGER A(6)
REAL R(8)
LOGICAL L(10)
! 隐 DO 循环.
A = (/(I, I = 1, 6)/)
! 含有 COS 函数的隐 DO 循环.
R = (/(COS(REAL(K)*3.1416/180.0), K=1,8)/)
! 用来赋逻辑值的隐 DO 循环
L = (/(.TRUE., N=1,5), (.FALSE., N=6,10)/)
```

本例中，A(1)=1，A(2)=2…；R(1)=cos(1.0*3.1416/180.0)，R(2)=cos(2.0*3.1416/180.0)…。L 的前五个元素为.TRUE.，后五个元素为.FALSE.。

隐 DO 循环可以和其它取值列表混合使用，例如：

```
INTEGER A(10)
A = (/1, 0, (I, I = -1, -6, -1), -7, -8 /)
! 隐 DO 循环和其它取值列表混合使用
```

上例中 A 的元素依次为：1,0,-1,-2,-3,-4,-5,-6,-7,-8。

数组赋值器是一维的，多维数组赋值器可以用来定义多维数组，或使用内部函数 RESHAPE。例如：

```
INTEGER B(2, 3), D(3, 6)
! 用隐 DO 循环给第一行赋值.
B(1, :) = (/(K, K = 1, 3)/)
! 用标量列表给第二列赋值
B(2, :) = (/5, 81, 17/)
! 隐 DO 循环和 RESHAPE 联合使用
D = RESHAPE ( (/(I, I= 1,18)/), (/3, 6/) )
```

本例中 B 的值为：

```
 1   2   3
 5  81  17
```

D 的值为：

```
 1  4  7  10  13  16
 2  5  8  11  14  17
 3  6  9  12  15  18
 4
```

下面是一些数组赋值器的替换格式：

（1） 用方括号代替括号和斜线，例如下面两个数组赋值器是等价的：

```
INTEGER C(4)
C = (/4,8,7,6/)
C = [4,8,7,6]
```

（2） 冒号三元下标（而不是隐 DO 循环）来指定值的范围和步长，例如下面两个数组赋值器是等价的：

```
INTEGER D(3)
D = (/1:5:2/)              ! 三元下标格式
D = (/(I, I=1, 5, 2)/)     ! 隐 DO 循环格式
```

3.5.4　数组操作

Visual FORTRAN 允许把整个数组或数组片段作为一个单独的对象。例如下面的操作都是合法的：

```
REAL A (5), B(5), C(5)
A = 4.0
B = 17.0
C = A + B
```

所有的算术操作符(+, -, *, /, **)，逻辑操作符(如 **.AND.**, **.OR.**, **.NOT.**)和所有关系操作符(如 **.LT.**, **.EQ.**, **.GT.**)，还有很多内部函数都可以接受数组名称作为参数并对数组元素逐一操作。可以使用数组名作为参数的内部函数称为基本函数。例如：

```
REAL A (5), B(5), C(5)
INTEGER D(5)
DATA PI /3.14159265/
A = (/ ((REAL(I) * PI/180.0), I = 1, 5) /)
B = COS(A)
C = SQRT(A)
D = CEILING (A * 180.0)
```

其中 **COS**, **SQRT** 和 **CEILING** 都是基本内部函数。当两个以上数组出现在赋值语句或表达式中时，数组的大小和形状应该相同（称为相容）。例如：

```
REAL A (2, 3), B(2, 3), C(2:3, 6:8)   ! 这些数组都相容
REAL D (4, 5), E (5, 4), F(5, 2, 2)   ! 这些数组不相容
```

下面对数组的操作不会引起冲突：

```
REAL X (10)
X (1:10) = X (10:1:-1)   ! 使用三元下标颠倒数组 X
```

这时因为赋值是从左向右进行，右侧数组元素在赋值之前没有改变。

如果数组片段指定的部分相容，则它也可以用于表达式和赋值。这样，不相容的数组也可以互相使用。例如，一个较小的完整数组可以和一个相容的更大的数组的数组片段进行操作：

```
REAL A (5), B(4, 7)
A = 20.0              ! 整个数组赋值
B = 5.0               ! 整个数组赋值
A = A - B (2, 1:5) ! 整个数组和相容片段操作
```

3.5.5 内部数组操作函数

FORTRAN 90 中增加了许多新的内在函数，使得运算更加灵活方便。下面介绍一部分新增数组内在函数，FORTRAN 90 的全部数组操作函数列在表 3.6 中。

1. 矩阵乘积函数

函数名：MATMUL（A，B）

描述：执行数值或逻辑型矩阵 A 与 B 的矩阵乘法。

说明：矩阵 A 和 B 必须是秩为 1 或 2 的数值型或逻辑型的有值数组，且 A 与 B 的类型必须相同。若 A 秩为 1，则 B 秩必须是 2；若 B 秩为 1，则 A 秩必须是 2。A 与 B 的矩阵乘积规则和结果值与数学上的定义相同。

2. 向量点乘函数

函数名：DOT_PRODUCT(A,B)

描述：执行数值或逻辑型向量 A 与 B 的点积乘。

说明：向量 A 和 B 必须是秩为 1（即向量）的数值型或逻辑型的有值数组，且 A 与 B 必须类型相同。向量 A 与 B 点乘的结果是标量，其规则和结果值与数学上的定义相同。

示例：DOT_PRODUCT((/1,2,3/),(/2,3,4/))的值为 20。

3. 数组归约函数

数组归约函数包括 SUM、PRODUCT、MAXVAL、MINVAL、COUNT、ANY 和 ALL 函数。

（1） 元素求和函数

函数名：SUM（ARRAY，DIM，MASK）

描述：沿着维 DIM，对在 MASK 真值中的数组 ARRAY 的所有元素求和。

说明：ARRAY 是被求和的数组名，是必选项。DIM、MASK 是任选自变量。DIM 用于指明选哪一维来求函数值。如 DIM=1 分别按列求和，DIM=2 分别按行求和，等等。MASK 起屏蔽作用，它的值通常是一个条件逻辑表达式，不满足条件的元素则被屏蔽在外，不参加求函数值。

示例：

- SUM（(/4，5，6/)）的值是 15。
- SUM（C，MASK=C>0.0）形成对 C 的所有正元素值的求和。
- 若 A 是数组

2　3　4
1　2　3

则 SUM（A，DIM=1）的值是[3，5，7]，SUM（A，DIM=2）的值是[9，6]

（2）　元素连乘求积函数

函数名：PRODUCT（ARRAY，DIM，MASK）

描述：沿着维 DIM，在对 MASK 的真值中的数组 ARRAY 的元素求连乘积。DIM、MASK 任选。

说明：同前。

示例：

- PRODUCT（(/4，5，6/)）的值为 120。
- PRODUCT（C，MASK=C>0.0)形成对 C 的所有正元素求连乘。
- 若 A 是数组

2　3　4
1　2　3

则 PRODUCT（A，DIM=1）的值是[2，6，12]，PRODUCT（A，DIM=2）的值是[24，6]。

4．数组查询函数

数组查询函数包括 SIZE、SHAPE、ALLOCATED、LBOUND 和 UBOUND 函数。

（1）　求数组大小函数

函数名：SIZE（ARRAY，DIM）

描述：求数组沿着所说明维的长度或数组元素的总数目。DIM 任选。

（2）　求数组形状函数

函数名：SHAPE（ARRAY）。

描述：求数组或标量的形状。

5．数组的构造函数

数组构造函数用于从已有数组的元素构造出新数组。数组构造函数包括 MERGE、PACK、UNPACK 和 SPREAD 函数。

（1）　合并数组函数

函数名：MERGE（TSOURCE，FSOURCE，MASK）

描述：在 MASK 的控制下，合并数组 STOURCE 和 FSOURCE。

说明：数组 TSOURCE 可以是任一类型，数组 FSOURCE 必须与 TSOURCE 具有相同的类型和种别类型参数。MASK 必须是逻辑型的。若 MASK 为真（T），则结果是 TSOURCE，若为假（.），则结果是 FSOURCE。

示例：若数组 TSOURCE 为$\begin{bmatrix}7 & 8 & 5\\4 & 3 & 9\end{bmatrix}$，FSOURCE 为$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}$，MASK 是数组$\begin{bmatrix}T & . & T\\T & . & .\end{bmatrix}$，则 MERGE（TSOURCE，FSOURCE，MASK）的值为$\begin{bmatrix}7 & 2 & 5\\4 & 5 & 6\end{bmatrix}$。

若 K 等于 3，则 MERGE（3.0,1.0,K>0）值为 3.0；若 K 等于-1，则值为 1.0。

（2） 压缩数组函数

函数名：PACK（ARRAY，MASK，VECTOR）

描述：在 MASK 控制下，将数组压缩成秩为 1 的数组。

说明：ARRAY 可是任意类型的数组。MASK 必须是逻辑型的，并与 ARRAY 具有相同形状。VECTOR（可选）必须秩为 1，且与 ARRAY 具有相同的类型和类型参数。结果是秩为 1 的数组，其类型和类型参数与 ARRAY 相同；若 VECTOR 存在，结果大小等于 VECTOR 的大小，否则其大小是 MASK 中真元素的个数 t，若 MASK 为标量且为真值，这时结果的大小与 ARRAY 相同。结果的值按数组中元素排序，ARRAY 中的第 i 个元素对应于 MASK 的第 i 个真元素。若 VECTOR 存在，且大小 n>t，则结果第 i 个元素值为 VECTOR（i），i=t+1,…,n。

示例：数组 A 为$\begin{bmatrix}0 & 0 & 0\\9 & 0 & 0\\0 & 0 & 7\end{bmatrix}$其非 0 元素可由 PACK 函数收集，PACK（A，MASK=A.NE.0）的结果为[9，7]，PACK(A,A.NE.0,VECTOR=(/1,2,3,4,5,6/))的结果值为[9,7,3,4,5,6]。

6. 数组重构形函数

函数名：RESHAPE（SOURCE，SHAPE，PAD，ORDER）

描述：从给定的数组的元素来构造一个指定形状的数组。

说明：SOURCE 是任意类型的数组。若缺省 PAD 或大小为 0，则 SOURCE 的大小必须大于等于 PRODUCT（SHARP）。结果的大小是 SHAPE 的各元素值的乘积。SHAPE 必须是秩为 1，大小为常数的整型数组。其大小必须是小于 8 的正整数。其元素不能为负值。

PAD（可选）必须是与 SOURCE 同类型及同种别类型参数的数组。ORDER（可选）必须是与 SHAPE 同形状的 整型数组，其值是（1，2，…，n）。

结果是一个与 SHAPE 具有相同形状、与 SOURCE 具有相同类型及类型参数的数组。结果的元素按 ORDER 所指的下标顺序将 SOURCE 的元素按正常数组元素的顺序排列。若需要，后跟 PAD 按正常数组元素排序的元素。

示例：RESHAPE（(/1，2，3，4，5，6/)，(/2，3/)）值为。

7. 数组操作函数

数组操作函数包括 TRANSPOSE、EOSHIFT 和 CSHIFT 三个函数。

（1） 矩阵转置函数

函数名：TRANSPOSE（MATRIX）

描述：将秩为 2 的数组转置。

示例：若 A 是数组$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}$，则 TRANSPOSE（A）的值为$\begin{bmatrix}1 & 4\\2 & 5\\3 & 6\end{bmatrix}$

（2） 去端移动函数

函数名：EOSHIFT（ARRAY，SHIFT，BOUNDARY，DIM）

描述：对秩为 1 的数组表达式作去端移位，或沿着维 DIM 对秩大小等于 2 的数组表达式在所有秩为 1 的完整数组片段上作去端移位。在一个数组片段的一端被移出的元素被丢弃，并在另一端移入相同的 BOUNDARY 的值。不同的片段可以有不同的 BOUNDARY 值，并可在不同的方向上移动不同的位数。SHIFT 为正值去端左移，为负值去端右移。BOUNDARY，DIM 任选。

示例：

若 A 是数组[2，4，6，8，10]，则 EOSHIFT（A，SHIFT=3）去端左移 3 位的结果为[8，10，0，0，0]；而用 EOSHIFT（A，SHIFT=-2，BOUNDARY=36）去端右移 2 位的结果为[36，36，2，4，6]。

一个秩为 2 的数组的各行可以移动相同或不同数量的元素，边界元素也可以相同或不同。

若 B 是数组$\begin{bmatrix}A & B & C\\D & E & F\\G & H & I\end{bmatrix}$，则 EOSHIFT（B，SHIFT=-1，BOUNDARY='*'，DIM=2）的值是$\begin{bmatrix}* & A & B* & D & E* & G & H\end{bmatrix}$，而 EOSHIFT（B，SHIFT）=（/-1，1，0/），BOUNDARY=（/'*','/','?'/）,DIM=2)的值是$\begin{bmatrix}* & A & B\\E & F & /\\G & H & I\end{bmatrix}$。

8. 数组定位函数

数组定位函数有 MAXLOC 和 MINLOC 两个函数。

最大值元素定位函数

函数名：MAXLOC（ARRAY，MASK）

描述：根据 MASK 的真值条件确定 ARRAY 中第一个最大值元素的位置。MASK 任选。

示例：

（1） MAXLOC（/3，8，5，8/）的值为[2]。

（2） 若 A 为数组 $\begin{bmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & -4 \end{bmatrix}$，则 MAXLOC（A，MASK=A.LT.6）

的值为[3，2]。

注意：即使 A 声明的下界不是 1，结果也是这样。

类似地，可给出最小值元素定位函数定义。

还有许多 FORTRAN 90 内部函数提供了对数组操作的函数，如表 3.6 所示。

表 3.6 FORTRAN 90 内部数组操作函数

函数名称	描 述
ALL	ALL(mask [, dim])判断全部数组值是否都满足沿着 dim 维方向（可选）满足 mask 的条件
ANY	ANY(mask [, dim])判断是否有数组值满足沿着 dim 维方向（可选）满足 mask 的条件
COUNT	COUNT(mask [, dim])统计沿着 dim 维方向（可选）满足 mask 的条件的元素的个数
CSHIFT	CSHIFT(array, shift [, dim])进行沿着 dim 维方向（可选）的循环替换
DOT_PRODUCT	DOT_PRODUCT(vector_a, vector_b)进行两个向量的点乘
EOSHIFT	EOSHIFT(array, shift [, boundary] [, dim])沿着 dim 维方向（可选）替换掉数组末端，复制边界值（可选）到数组末尾
LBOUND	LBOUND(array [, dim])返回沿着 dim 维方向（可选）较低维的下界
MATMUL	MATMUL(matrix_a, matrix_b)完成两个矩阵（二维数组）的乘积
MAXLOC	MAXLOC(array [, mask] [, dim])返回数组的全部元素或一组满足 mask（可选）条件或指定维元素的最大值的位置
MAXVAL	MAXVAL(array [, dim] [, mask])返回沿着 dim 维方向（可选）满足 mask 条件的最大值
MERGE	MERGE(tsource, fsource, mask)按 mask 条件组合两个数组
MINLOC	MINLOC(array [, mask] [, dim])返回数组的全部元素或一组满足 mask（可选）条件或指定维元素的最小值的位置
MINVAL	MINVAL(array [, dim] [, mask])返回沿着 dim 维方向（可选）满足 mask 条件的最小值
PACK	PACK(array, mask [,vector])把一个数组中使用 mask 压缩至 vector 大小（可选）的向量
PRODUCT	PRODUCT(array [, dim] [, mask])返回沿着 dim 维方向（可选）满足 mask 条件（可选）的元素的积
RESHAPE	RESHAPE(source, shape [, pad] [, order])使用下标 order（可选）并填补 pad 数组元素（可选）来改变数组形状
SHAPE	SHAPE(source)返回数组的形状
SIZE	SIZE(array [, dim])返回数组沿着 dim 维方向（可选）的长度
SPREAD	SPREAD(source, dim, ncopies)通过增加一维来复制数组
SUM	SUM(array [, dim] [, mask])返回沿着 dim 维方向（可选）满足 mask 条件（可选）的元素的和
TRANSPOSE	TRANSPOSE(matrix)转置二维数组
UBOUND	UBOUND(array [, dim])返回沿着 dim 维方向（可选）较高维的下界
UNPACK	UNPACK(vector, mask, field)把向量在 mask 条件下填充 field 的元素解压至数组

在这些函数中，*mask* 是屏蔽选项，它通常是一个表明条件的逻辑表达式，满足条件的元素参与运算，否则将被屏蔽掉。其它参数的类型参见附录《Visual FORTRAN 语言简表》，更详细的用法可以参考在线帮助。

3.5.5 指针

指针类似于可分配数组，允许动态访问和处理数据。指针在联合内存之前没有初始化的地址空间，也不能被引用。当给它指定目标或分配内存后才和内存相联系。在程序执行过程中指针的目标是可以改变的。

指针可以是标量、数组、派生类型成员或派生类型整体，它可以指向标量、数组、派生类型成员或派生类型整体。指针还可以指向数组元素或数组片段，但是数组元素或数组片段不能是指针。指针有 POINTER 属性，并且只能指向具有 TARGET 属性的对象，或另一个以联合目标的指针。

指定指针可以通过声明 POINTER 属性，或用 **POINTER** 语句。如果指针是数组则数组必须声明为迟形数组。例如：

```
REAL, POINTER :: A (:,:)   ! 迟形数组的 POINTER
                           ! 属性类型声明
REAL B, X(:,:)
POINTER B, X               ! POINTER 语句声明标量 B 和
                           ! 迟形数组 X 为指针
```

类型地，指定目标也可以通过声明 TARGET 属性，或用 **TARGET** 语句。目标可以是显式形状或迟形数组。如果是迟形数组则它必须有 ALLOCATABLE 属性。目标不能是有向量下标的数组片段。一些指定目标的例子如下：

```
! 迟形数组的 TARGET 属性声明
REAL, ALLOCATABLE, TARGET :: C (:,:)
REAL D
REAL, ALLOCATABLE :: Y(:,:)
INTEGER K(33, 20)
! TARGET 语句
TARGET D, Y, K
```

具有 POINTER 属性的对象可以在公共块中，但包含指针对象的公共块的声明必须指定相同的标量顺序。如果一个对象有 POINTER 属性就不能有 INTENT, PARAMETER 或 TARGET 属性。如果一个对象有 TARGET 属性就不能有 PARAMETER 或 POINTER 属性。哑元不能有 TARGET 属性。

指针赋值是将一个存在的目标和指针相联系。指针还可以指向其它目标，多个指针也

可以指向同一个目标。ALLOCATE 语句可以通过创建一个目标来开辟指针的空间。指针的目标必须和指针具有相同的类型、种别和形状。指针赋值具有普通的赋值语句形式（除了赋值符号=>），例如：

```
REAL A, X(:,:), B, Y(5, 5)
POINTER A, X
TARGET B, Y
A => B     ! 标量指针赋值
X => Y     ! 数组指针赋值
```

指针必须有 POINTER 属性，而目标必须有 TARGET 或 POINTER 属性。如果目标有 TARGET 属性则指针变量将会和目标直接联系。如果目标有 POINTER 属性则指针变量将会和该指针指向的同一个目标相联系。

已经和一个目标相联合的指针可以从新指定一个新目标。这时，指针和旧目标脱离联系并和新目标建立联系。如果指针 P1 的目标是另一个指针 P2，并且被指向的指针 P2 成为脱离的，则 P1 仍和 P2 是关联的，只是和 P2 原来指向的目标没有关系。

如果指针 P1 的目标是另一个指针 P2，并且被指向的指针 P2 没有相关联的目标，则 P1 也是没有定义的。可以使用内部函数 ASSOCIATED 来判断指针是否和一个目标相关联，或两个指针和同一个目标相关联。例如：

```
REAL C (:), D(:), E(5)
POINTER C, D
TARGET E
LOGICAL STATUS
! 指针赋值
C => E
! 指针赋值
D => E
! 返回 TRUE; C 已联合
STATUS = ASSOCIATED (C)
! 返回 TRUE; C 和 E 联合
STATUS = ASSOCIATED (C, E)
! 返回 TRUE; C 和 D 与同一个目标联合
STATUS = ASSOCIATED (C, D)
```

指针可以是派生数据类型或派生类型的成员。派生类型的指针成员可以指向这种派生类型，这样就可以创建链表或树结构，例如：

```
TYPE ORDER
   INTEGER INDEX
```

```
    TYPE(ORDER), POINTER :: NEXT
END TYPE ORDER
```

这种结构可以用来作分支索引：

```
TYPE (ORDER), POINTER :: LIST
ALLOCATE(LIST)
LIST%INDEX = 1
ALLOCATE(LIST%NEXT)
LIST%NEXT%INDEX = 2
ALLOCATE(LIST%NEXT%NEXT)
LIST%NEXT%NEXT%INDEX = 3
```

下面在树的顶部加上新的实体：

```
TYPE (ORDER), POINTER :: NEW_TOP
ALLOCATE(NEW_TOP)
NEW_TOP = ORDER(0, TREE)
```

原来 LIST%INDEX 指向的第一个索引现在被 NEW_TOP%NEXT%INDEX 指向了。

3.5.6 数组与指针的动态联合

当用户分配动态存储空间时，变量或数组的大小是在运行时而不是在编译时确定的。动态分配是通过可分配数组和指针来确定的。动态分配还可以用于标量和任何类型的数组。当用户给数组指定了可分配属性时并没有立即分配内存，而是直到使用 ALLOCATE 语句后才分配。随后还可以用 DEALLOCATE 语句释放内存空间，这时数组可以以比其它形状或目的来使用。

当指定标量或数组具有 POINTER 属性时，直到和已有内存空间的目标相联系时才有内存空间。指针赋值语句和 **ALLOCATE, NULLIFY** 以及 **DEALLOCATE** 语句提供了对指针的创建、联合和释放的动态控制。

应该注意的是，动态内存分配受一些因素的限制，包括交换文件的大小和其它同时运行的应用程序所需的内存大小。如果动态分配的内存太大或试图使用其它应用程序的保护内存会产生一般内存保护错误。碰到这类问题可以通过控制面板来改变虚拟内存的大小或交换文件的大小。还有一些编程技术可以降低内存需要，比如使用一个较大的数组而不是多个独立的数组。

1. ALLOCATE 语句

ALLOCATE 语句动态创建指针目标和可分配数组，使内存和对象相联系。分配的对象可以被命名为任何有 POINTER 或 ALLOCATABLE 属性的变量。例如：

```
REAL A, B(:,:), C(:), D(:, :, :)
POINTER A, B
ALLOCATABLE C, D

READ (*, *) N, M
ALLOCATE (A, B(N, M), C(-2:40), D(3, 3, 3))
```

指针被分配后将和新创建的内存空间联系，这个空间是指针目标。其它指针也可以通过指针赋值来指向同一个目标。还可以为已经和某个目标联系的指针分配内存，这种情况下，指针和原来的目标脱离联系并和新目标相联系。

如果脱离了原来分配过的指针（通过重新赋值或 **NULLIFY** 语句），则该指针原来指向的空间将不能再被使用。显然这是一种浪费，应该在重新赋值之前释放内存空间。

当数组被分配时，内存分配给用户指定大小的数组。**ALLOCATE** 语句中的秩必须和分配的指针或可分配数组的秩相同。在分配的同时，**ALLOCATE** 语句中的上下界决定了数组的大小和形状。边界的值可以是正数、负数或零，缺省的下界为 1。如果某维上界比下界小则该维的长度为零，并且数组或指针的大小为零。大小为零的数组不能被赋值。

当前被分配的数组不能被再分配，否则会引起运行错误。错误状态可以由 **ALLOCATE** 语句中的 **STAT=**分类符获得。如果是存在的，**ALLOCATE** 语句的成功执行将返回 0，否则返回正值。例如：

```
REAL, ALLOCATABLE :: A(:)
INTEGER ERR_ALLOC
ALLOCATE (A (5), STAT = ERR_ALLOC)
IF (ERR_ALLOC .NE. 0) PRINT *, "ALLOCATION ERROR"
```

如果 **STAT=**分类符没有出现则会出现错误，程序将中止执行。可以用内部函数 **ALLOCATED** 来判断一个数组是否已被分配。例如：

```
REAL, ALLOCATABLE :: A(:)
...
IF (.NOT. ALLOCATED(A)) ALLOCATE (A (5))
```

2. NULLIFY 语句

NULLIFY 语句将一个指针和目标脱离，还可以脱离一组指针对象，其中每个指针对象都必须有 POINTER 属性。例如：

```
INTEGER, POINTER :: A(:), B, C(:, :, :)
...
NULLIFY (A, B, C)
```

释放没有分配的的指针不是错误。指针开始没有未定义的联系状态。为了使初始化指针时使它不指向任何目标，可以执行 **NULLIFY** 语句。如果一个函数返回指针，结果指针在开始执行时的联系状态是未定义的。在函数返回之前结果必须和一个目标相联系或用 **NULLIFY** 语句特别定义该指针被释放。

3. DEALLOCATE 语句

DEALLOCATE 语句用来释放已分配数组的内存和指针对象的指针。每个被释放的对象必须是已分配数组，或是指向已经用 **ALLOCATE** 语句创建的内存空间对象的指针。例如：

```
REAL, POINTER :: A(:), B, C
REAL, ALLOCATABLE, TARGET :: D(:)
REAL, TARGET :: E
REAL, ALLOCATABLE :: F(:, :)
...
ALLOCATE(B, D(5), F(4, 2))
A => D
C => E
...
DEALLOCATE (B, D, F)
```

只有被 **ALLOCATE** 语句分配的内存空间才可以被 **DEALLOCATE** 语句释放。在前面的例子中，指针 B 有 **ALLOCATE** 创建的内存空间对象，所以它可以被释放；而指针 C 被指向未分配的对象 E，所以不能被分配。释放一个和可分配目标的指针也会产生错误，例如，如果用户试图释放上面例子中的指针 A 则会产生错误。相反，如果释放指针指向的目标是合法的。例如上例中的数组 D 就可以被释放。

释放没有用 **ALLOCATE** 语句分配的内存空间会产生运行错误。可以使用 ALLOCATED 语句判断数组或目标是否被分配，或用 ALLOCATED 语句判断指针的联合状态。

错误状态可以由 **DEALLOCATE** 语句中的 **STAT=**分类符获得。如果是存在的，**DEALLOCATE** 语句成功返回 0，否则会返回正数。例如：

```
REAL, ALLOCATABLE :: A(:)
INTEGER ERR_DEALL
...
DEALLOCATE (A , STAT = ERR_DEALL)
```

```
IF (ERR_DEALL .NE. 0) PRINT *, "DEALLOCATION ERROR"
```

4. 联合状态和定义

当过程的执行被 **RETURN** 或 **END** 语句中止时，可分配数组的分配状态和指针的联合状态都变成未定义，但除了下列情况：

1. 该指针或数组有 SAVE 属性。
2. 指针是声明有 POINTER 属性的函数的返回值。

RETURN 和 **END** 语句并不释放数组或指针分配的内存，所以应该在退出子程序之前主动释放数组分配的或和指针相联系的内存。

如果可分配数组的联系状态变成未定义的，那么它就不能被引用、定义、分配或释放。而指针始终能被取消、分配和赋值，但当它的联系状态变成未定义时就不能被引用或释放。当指针对象成为未定义时，指针和指针联系也成为未定义的。

可分配数组的联合状态可以是：

1. 已分配。该数组被 **ALLOCATE** 语句分配，这种数组可以被引用、定义或释放。
2. 目前未联合。该数组从未分配或上一个操作是释放。这种数组不能被引用或定义。

当可分配数组赋值时就被定义，例如：

```
REAL, ALLOCATABLE :: A(:)
ALLOCATE (A(100))     ! A被分配但未定义
                      ! A的分配状态是已分配
A(1:100) = 1          ! A 被定义
DEALLOCATE (A)        ! A 被释放
                      ! A 的状态是未分配
```

3.5.7 DIGITAL FORTRAN 指针

DIGITAL FORTRAN（也称为整数指针）和标准 FORTRAN 90 指针不同。DIGITAL FORTRAN 指针由三部分组成：指针，指针基变量和目标对象。

指针基变量指向目标指针的过程分两个步骤：首先，把基指针变量和一个整数指针联合。这一步可以由 DIGITAL FORTRAN 的 **POINTER** 语句，格式如下：

POINTER (*p*, *var*) [, (*p*, *var*)]

DIGITAL FORTRAN 指针（上面的 *p*）不能是显式的形式，其值为 *var* 的地址。基指针（上面的 *var*）可以是任意类型的变量，包括数组或字符串。**POINTER** 语句必须在程序或过程的定义部分。DIGITAL FORTRAN 指针所赋的值就是目标对象所联系的存储空间，这可以由内部函数 LOC 或 MALLOC 来实现：

```
REAL VAR, A
POINTER (P, VAR)
```

```
P = LOC(A)
```

这时 DIGITAL FORTRAN 指针的值为目标对象的地址（上例的 A）。赋给基指针变量（上例的 VAR）的值存放在指针存放的地址中，因此该值将传给目标对象。例如：

```
REAL VAR(5), A(5)
POINTER (P, VAR)
P = LOC(A)
VAR(2) = 0.0 ! Sets A(2) to 0.0.
```

注意：不能和 DIGITAL FORTRAN 指针使用 **ALLOCATE** 或 **DEALLOCATE** 语句，为了分配内存只能使用 **MALLOC** 语句。使用以后可以使用内部函数 FREE 释放内存。

DIGITAL FORTRAN 指针应遵守以下规则：

（1） 两个指针的值可以相同，即允许指针笔名。

（2） 直接使用时指针被当成整型变量。在 Windows NT 和 Windows 9x 系统中，指针占用一个数值存储单元，即 32 位(INTEGER(4))。

（3） 指针不能是基指针变量。

（4） 指针不能出现在 ASSIGN 语句中，并且不能有如下属性：ALLOCATABLE，PARAMETER，EXTERNAL，POINTER，INTRINSIC，TARGET。

（5） 在 DATA 语句中出现的指针只能是整型直接量。

（6） 整数可以转换成指针，所以可以指向绝对内存地址。

（7） 指针不能是函数返回值。

由于指针是整型变量，所以一般的整数的数学运算都适用。这样就可以改变地址，例如：

```
REAL A(10), VAR
POINTER (P, VAR)

P = LOC(A)
DO I = 1, 10
   VAR = 0.0
   P = P + 4
END DO
```

上例中，数组 A 的 10 个实型变量通过重新定义 DIGITAL FORTRAN 指针中的地址值来置为零。A 的元素在内存中连续存储。每个实数的地址占 4 字节，所以 P 每增加 4 就指向 A 的下一个元素。

基指针变量应遵守以下规则：

（1） 基指针变量不能分配任何内存。引用基指针变量可以找到与它联系的指针的内

容，进而找到基指针变量的基地址。

（2） 基指针变量不能赋初始化数据，或有一个包含数据初始化区的记录结构。

（3） 基指针变量只能在一个 POINTER 语句中出现。

（4） 基指针变量可以有固定、可调或假设的维。

（5） 基指针变量不能出现在 COMMON, DATA, EQUIVALENCE, 或 NAMELIST 语句里，并且不能有如下属性：ALLOCATABLE，POINTER，AUTOMATIC，SAVE，INTENT，STATIC，OPTIONAL，TARGET，PARAMETER。

（6） 基指针变量不能是：哑元、函数返回值、记录区或数组元素、大小为 0、自动对象和一般接口块名。

（7） 如果基指针变量是派生类型则它必须是顺序类型。

第四章　程序单元和块结构

为了使程序有良好的可读性和可维护性，FORTRAN 90 中明确指出了块的结构，以及把块作为单元来看的可执行结构序列的概念。由于每一个结构块只有一个入口、一个出口，入口与上一结构块的出口衔接，出口与下一结构块的入口衔接，使整个程序形成一链条状。因此，程序一旦出错，其影响波及范围又被约束在结构块中，不会象结构不良的程序乱用 GOTO 语句那样使错误交叉蔓延，以致程序修改困难、不易维护。

FORTRAN 一般是顺序地执行语句，用户可以通过使用可执行过程块来改变原来的顺序，把程序的控制转向其它语句。本章主要讨论程序单元、块语句和块结构。

4.1　概述

FORTRAN 90 中的程序单元有以下几种：

- 主程序
- 模块
- 过程
- 块数据程序单元

一个程序单元不需要包含可执行语句。例如，它可以是子程序中包含接口块的模块。用户可以分别编译源文件，最后再把它们连接起来。包含内部子程序或函数的程序单元称为宿主（host）程序。表 4.1 是对这四种程序单元类型的定义：

表 4.1　　程序单元的定义

程序单元	定义
主程序	主程序是程序开始执行的标志，其第一条语句不能是 SUBROUTINE, FUNCTION, MODULE 或 BLOCK DATA 。主程序可以用 PROGRAM 语句作为第一条语句，但不是必需的
过程	子程序或函数
块数据程序单元	在命名的公共块中提供变量初始值的程序单元
模块	包含数据对象定义、类型定义、函数或子程序接口和其它程序可访问的函数或子程序的程序单元

其中过程（procedure）可以是子程序或函数。过程可以是内部过程、外部过程或模块。过程的引入使开发大型和结构良好的程序变得更容易，如表 4.2 所示。

表 4.2 使用过程的优势

用户意图	使用过程的优势
开发大型程序	可以把一个大型程序分解成多个部分，使程序的开发、调试、维护和编译更容易
希望在多个程序中都包含某一过程	创建包含这些过程的对象文件，然后连接到要使用这些过程的程序
希望改变一个过程的执行	把过程单独存为一个文件并单独编译，然后可以对这个过程进行改动而程序剩余部分不用改动

把程序分解为过程可以实现封装，使一个例程的改动和其它例程的改动相互独立。

函数是以表达式调用的过程，它返回的是表达式中使用的一个值。函数结果是函数返回给表达式的一个或一组值。还可以用 **CALL** 语句调用一个子程序，它不返回任何值（尽管它可能改变一些参数的值）。

程序单元之间的关系有：

- 联合

  这是一种机制，这种机制允许不同的程序单元共享变量，从而不用重新定义变量就可以以不同的名字选址。

- 范围

  它描述的是一个名称（或者是全局的或者是局部的）作用的范围。

4.2 主程序

程序的执行始终是开始于主程序的第一条可执行语句，所以每一个可执行程序中必须有且只有一个主程序单元。

4.2.1 主程序格式

主程序按顺序由下面几部分组成：

[PROGRAM [程序名]]
[说明部分]
[执行部分]
[内部子程序部分]
END [PROGRAM [程序名] **]**

PROGRAM 语句的格式为：

```
PROGRAM test
```

其中 *test* 是可选的程序名。

END 语句是程序唯一必要的部分。如果 **END** 语句包含了程序名，则要和 **PROGRAM** 语句中提到的程序名一致：

```
END PROGRAM test
```

主程序名和外部过程和公共块名都被认为是全局名称。全局名称在一个程序中必须是唯一的。

图 4.1 显示的是一个 Visual FORTRAN 程序的标准结构。每个框都代表一个独立的源文件：

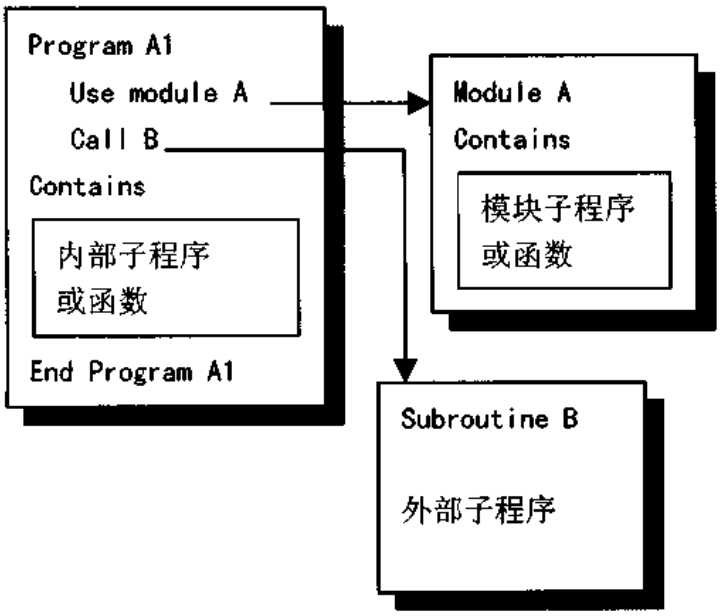


图 4.1　Visual FORTRAN 程序的标准结构

其中主程序 A1 可以包含内部子程序，调用外部子程序 B 并可以调用模块 A 中包含的模块过程。包含在模块 B 中的函数和子程序称为模块过程。

程序中的说明部分可以包含 **USE** 语句，隐式说明，参数语句，格式语句和变量与常数的声明。主程序的说明部分不能包括自动数组和指针。

外部过程、模块或接口块中可以在使用 **OPTIONAL**, **INTENT**, **PUBLIC** 或 **PRIVATE** 语句和它们相应的属性，但主程序的说明部分不能包含这些语句：**OPTIONAL** 和 **INTENT** 只能用于哑元，而哑元并不在主程序中出现；**PUBLIC** 和 **PRIVATE** 语句只对模块适用。另外，主程序中包含的 **SAVE** 语句也是无效的。

4.2.2　程序的执行

主程序中可执行部分语句的执行顺序是从第一个可执行结构顺序执行，直到遇到一条 **STOP** 或 **END PROGRAM** 语句。主程序的执行部分不能包含 **RETURN** 或 **ENTRY** 语句。主程序中的函数和子程序可以调用自身，但主程序不能递归使用。

正常程序执行顺序可以被如 **IF, CASE** 或 **DO** 块控制语句改变。

4.3 模块

模块也是 FORTRAN 90 中的一种程序单元。本节主要介绍模块的定义和引用。

4.3.1 概述

在 FORTRAN 90 中新增加了一种称之为模块的程序单元，它的功能是提供一种方便有效的常量、变量、类型定义及过程的共享途径。它可代替 FORTRAN77 中 COMMON 和 EQUIVALANCE 语句的功能（这两个语句在 FORTRAN 90 中不提倡使用）。模块的用途主要有：

- 包含通常使用的过程
- 声明全局变量和派生类型
- 声明外部过程的接口块
- 初始化全局数据和全局可分配数组
- 封装数据和处理数据的过程

模块程序单元有它独特的形式，即单元内没有可执行语句，除了说明语句外，最多包含内部过程。模块这种程序单元不能被直接执行，必须用 USE 语句引用。

以前存储在 **COMMON** 块中的变量可以被摘录和保存到命名的模块。这样，如果编写使用这些变量的程序就可以使用这样的模块而不必包括 **COMMON** 块，从而不必在多个程序单元中声明变量，还可以保证：

- 在所有使用这些变量的程序单元中的声明是一致的
- 用同一个值初始化

如果一个程序单元只使用模块中的部分变量或需要重新命名部分变量，该单元可以指定这些变量具有 ONLY 属性。

在程序中使用模块和使用 **INCLUDE** 语句很类似。每种情况下，都由某个独立文件中内容提供程序运行所需的信息。使用模块比使用 **INCLUDE** 语句的优势在于：

- 封装数据和操作这些数据的子程序

  模块提供了封装相关定义和操作的简便方法。用户可以把多个相关但不同的文件中的信息结合到一个模块中，而不需使用多个 **INCLUDE** 语句。这样保证了程序单元之间的内聚性强而耦合性弱。

- 变量、常数和模块过程的命名控制

  使用模块允许用户临时重命名常数、变量甚至过程。

- 隐式接口

  当用户使用模块时，模块过程的参数和返回值对于主程序是可知的，并且和调用过程匹配。

- 数据类型和操作符定义

  模块允许用户为派生类型定义特殊操作符，还可以把内在操作符扩展到其它数据类型。

- 只在程序中一个位置指定信息

这样能保证使用该信息的不同的程序单元对这个信息的解释是一致的。例如，某个程序单元中的一条 **IMPLICIT** 或 **EXPLICIT** 语句可能会引起对一个包括文件的解释和其它包括这个文件的程序单元的解释不同。

4.3.2 模块的定义

模块程序单元的一般形式为 ：

MODULE 模块名
类型说明部分
[CONTAINS]
[内部辅程序]
…
[内部辅程序 n]
END MODULE[模块名]

模块名和主程序名类似，是全局名称。它不能和其它程序单元、外部过程、公共块或任何模块中的局部名称重名。如果结束模块语句包括模块名则这个模块名必须和初始的命名相符。例如：

```
MODULE DATA_MODULE
   SAVE
   REAL A(10), B, C(20,20)
   COMPLEX D(5,6)
END MODULE DATA_MODULE
```

模块以 CONTAINS 语句开始，到若干个内部辅程序结束，这一段称模块辅程序部分，它是可选的，一般情况下不使用。通常在为某一个导出类型规定新的操作符时，就把实现这些新操作的过程置于 CONTAINS 后面，以便把这种操作定义，供各外部过程共享。

例如，模块：

```
MODULE DATA_MODULE
  REAL, DIMENSION (3:15)::A,B
  INTEGER::I=0
  INTEGER,PARAMETER::J=10
END MODULE DATA_MODULE
```

在该模块中说明 A、B 是实型数组；I 是整型变量 ，初始值是 0；J 是整型常数 10。如果有一个外部过程引用了此模块，则相当于这三 句类型说明移到外部过程的说明部分中。I 和 J 的值传递给外部过程中同名的实体。

在编写模块时应注意以下几点：

（1） 如果模块结束语句中写有模块名，则它一定要与模块语句中的模块名相同。

（2） 模块名不能与可执行程序内的任何程序单元名、外部过程名或分用块名相同，也不得与模块内的任何局部或相同。

（3） 在模块的说明部分中不允许含语句函数语句、ENTRY 语句或 FORMAT 语句，但它们可以出现在模块所包含的模块辅程序的说明部分。说明部分可以包含的语句如表 4.3 所示。

表 4.3　　模块说明部分可以包含的语句

ALLOCATABLE	PRIVATE	COMMON	PUBLIC
DATA	SAVE	DIMENSION	STATIC
EQUIVALENCE	TARGET	EXTERNAL	USE
IMPLICIT	VOLATILE	INTRINSIC	PARAMETER
NAMELIST	POINTER	派生类型定义	接口块

4.3.3 模块的引用（USE 语句）

模块程序单元如何被外部过程引用，使模块中的内容为外部程序单元所共享，这就是要用 USE 语句来实现。

USE 语句的一般形式为：

USE 模块名 1，模块名 2，…，模块名 n

或

USE 模块名，ONLY：访问实体 1，访问实体 2，…，访问实体 m

USE 语句在没有 ONLY 选择时，将为外部过程提供对其指明的模块中的全部公共实体进行访问；若有 ONLY 选择时，将仅提供对其指明的模块中的公共实体进行访问。

例如对模块 STATS_LIB，语句

USE STATS_LIB

提供了访问模块 STATS_LIB 的所有共用实体的可能。语句

USE STATS_LIB,ONLY:PROD

提供了仅对模块 STATS_LIB 中的共同实体 PROD 进行访问。

下面是利用模块实现数据共享的例子。本例读入实型数 A、B、C、D，调用函数 ADD 求四数之和，调用函数 PROD 求四数之积。

```
MODULE DATA_MODULE
  IMPLICIT NONE
  REAB::A,B,C,D
END MODULE
PROGRAM ABCD_1
```

```
  USE DATA_MODULE
  REAL::SUM,PROD
  READ * ,A,B,C,D
  PRINT * ,ADD(), PROD( )
END PROGRAM ABCD_1

FUNCTION ADD( ) RESULT(ADD_RESULT)
  USE DATA_ MODULE
  REAL:: ADD_RESULT
  ADD_RESULT=A+B+C+D
END FUNCTION ADD

FUNCTION PROD( )RESULT(PROD_ RESULT)
  USE DATA_ MODULE
  REAL:: PROD_RESULT
  PROD_ RESULT=A* B*C*D
END FUNCTION PROD
```

这里所有函数只有一张空哑元表（ ），主程序调用时也只有空的实际参数表（ ）。这是因为引用了模块 DATA _MODULE 后，使得函数中的 A、B、C、D 变量共享，即主程序输入 A、B、C、D 值后，函数 ADD 和 PROD 中的 A、B、C、D 也具有了同样值，不再需要哑元与实际参数的结合。

模块内除了类型说明外，在 CONTAINS 语句和 END MODULE 语句间还允许有内部过程，供引用模块的各程序单元使用，以实现过程共享。

下面是说明过程共享的例子。本例把上例中的求四变量之和与求四变量之积的函数作为模块内部辅程序写入模块内。

```
MODULE ABCD_MODULE
  IMPLICIT NONE
  READ::A,B,C,D
  CONTAINS
  FUNCTION ADD( )RESULT(ADD_RESULT)
    REAL:: ADD_RESULT
    ADD_RESULT=A+B+C+D
  END FUNCTION ADD
  FUNCTION PROD( ) RESULT(PROD_RESULT)
    REAL::PROD_RESULT
    PROD_RESULT=A*B*C*D
  END FUNCTION PROD
```

```
END MODULE

PROGRAM ABCD_2
  USE MODULE ABCD_MODULE
  READ*, A,B,C,D, ADD_RESULT,PROD_RESULT
  PRINT * ,ADD( ) ,PROD( )
END PROGRAM
```

本例中由于引入过程共享模块从而使得整个程序大大简化。

下面的例子在模块中定义一个导出类型，供主程序或其它程序单元访问：

```
MODULE SPARSE
  TYPE NONZERO
    REAL A
    INTEGER I, J
  END TYPE
END MODULE
```

该模块定义了由一个实型成分和两个整型成分组成的类型，以便存放非零矩阵元素的数值及它的行和列的下标。

4.4 过程

过程可以是子函数或函数。FORTRAN 90 包括下面几种过程类型：

- 外部（external procedure）过程
  外部过程是在某个外部程序单元中定义的。
- 内部（internal procedure）过程
  内部过程在程序单元内部定义而且只能被该程序单元调用。包含内部过程的程序称为宿主。内部过程在 **CONTAINS** 语句之后。
- 内在（intrinsic procedure）过程
  内在过程在编译器内部定义，它可以不用任何附加声明或说明。如果用户要用内在过程作为其它过程的参数需要用 **INTRINSIC** 语句声明它。
- 模块（module procedure）过程
  模块过程在模块中定义，它可以被所有使用该模块的程序调用。包含过程的模块称为宿主。

过程引用是调用一个过程的方法。子程序引用使用 **CALL** 语句或有时用定义的赋值语句。函数引用或者是使用函数名的表达式或定义的操作符。函数结果是函数表达式返回的一个或一组值。

4.4.1　外部过程

外部过程是用户编写的函数或子程序，它位于主程序外部。外部过程可以单独以源文件存储和编译，或者可以在主程序源代码 **END** 语句后包括进来。外部过程本身可以包含内部函数或子程序。

外部过程或模块过程可以包含一个或多个 **ENTRY** 语句，这样就允许用户进入过程内部的某点。效果是允许用户把多个希望有条件进入的相关函数或子程序组织起来。内部过程不能包含 **ENTRY** 语句。

- **ENTRY** 语句

  ENTRY 语句允许在特定的可执行语句中进入子程序。它的形式和用法与函数和子程序类似。**ENTRY** 语句包括一个入口点的名称，一个可选参数列表和可选的 RESULT 参数及结果名（在函数的情况下）。一个过程可以有一个或多个 **ENTRY** 语句。**ENTRY** 语句只能在外部过程或模块过程中使用。FORTRAN 90 块结构，例如 CASE，提供了比通过过程实现的控制流更好的方式。

4.4.2　块数据程序单元

一个块数据子程序是为命名的公共块中的变量定义初始值的程序单元。它只包含数据声明和初始值，不包含可执行语句。块数据程序单元的格式是：

BLOCK DATA [名称]
　　[说明]
END [BLOCK DATA [名称]]

变量一般由 **DATA** 语句来初始化。公共块中命名的变量只能在块数据子程序或某个子程序中初始化一次，或只能由所有的子程序完全一致地初始化。空（未命名）公共块中的变量不能在块数据子程序中初始化。更好的编程经验是用模块而不是块数据程序单元来声明和初始化变量。

块数据模块可以包含的语句由表 4.4 所示。

表 4.4　　**块数据模块可以包含的语句**

COMMON	PARAMETER	USE
DATA	POINTER	派生类型定义
DIMENSION	RECORD	记录结构定义
EQUIVALENCE	SAVE	类型声明语句
IMPLICIT	STATIC	
INTRINSIC	TARGET	

块数据程序单元中的类型声明不能包含下面的属性说明：

- ALLOCATABLE
- *AUTOMATIC*

- EXTERNAL
- INTENT
- OPTIONAL
- PRIVATE
- PUBLIC
- *VOLATILE*

即使用户开始不定义在命名公共块中的所有对象，如果在公共块中定义了至少一个对象也必须说明它们。可以在一个块数据程序单元的多个公共块中初始化这些对象。块数据程序单元开始可以定义非指针对象。任何于一个公共块中的对象相联系的对象都被认为是在该公共块中。

下面是使用块数据程序单元的例子：

```
BLOCK DATA WORK
   COMMON /WRKCOM/ A, B, C (10,10)
   DATA A /1.0/, B /2.0/, C /100*0.0/
END BLOCK DATA WORK
```

4.5 过程接口块

过程接口块是 FORTRAN 90 中引进的新程序块，它用来确定过程被调用的形式。

在 FORTRAN 语言中，主调程序与被调程序是分别编译的。由于 FORTRAN 90 对过程的许多功能做了较大的扩充，有些功能单靠简单的调用语句已无法反映，因而系统也就无法进行正确的编译。为了全面准确地通知编译系统，有时就需在主调程序中写入接口块，通过它可为主程序与被调程序指明一个显式的接口。

接口块一般应写在主调程序中的最前面，其形式如下：

PROGRAM 程序名
接口块
主调程序内变量说明
执行语句
…
接口块的一般形式：
INTERFACE
函数语句（或子程序语句）
被调用过程变量及函数结果值说明
函数 END 语句（或子程序 END 语句）　→！程序接口体
END INTERFACE

使用接口块时应该注意的是：

（1）　接口块以 INTEREACE 语句开始，END INTEREACE 语句结束，块内只能取被调用过程中的说明部分，不允许出现在任何可执行语句。接口块内的语句构成接口体。

（2）　接口体中一定不能含有 ENTRY 语句、DATA 语句、FORMAT 语句、成语句函数语句。

（3）　接口块不允许出现在 BLOCK DATA 程序单元中。

（4）　一个辅程序中的接口块不允许含有一个被该辅程序定义的过程的接口体。

（5）　接口块中可以有多个接口体，即一个接口块中可以说明多个被调用过程，每个过程用自己的开始语句与结束语句定界，排列次序任意。

例如，

```
INTERFACE
  SUBROUTINE EXT1 (X, Y, Z)
    REAL, DIMENSION (100, 100) : : X, Y, Z
  END SUBROUTINE EXI1

  SUBROUTINE EXT2 (X, Z)
    REAL X
    COMPLEX (KIND=4) Z (2000)
  END SUBROUTINE EXT2

  FUNCTION EXT3 (P, Q)
    LOGICAL EXT3
    INTEGER P (1000)
    LOGICAL Q (1000)
  END FUNCTION EXT3
END INTERFACE
```

这个接口块对于三个外部过程 EXT1、EXT2 和 EXT3 说明了显式接口。这些过程中的任何一个可以使用关键词调用。如

```
EXT3 (Q=P_MASK(N+1,N+1000),P=ACTUAL_P)
```

接口块并不是每个主调程序都必须写的。若仅使用 FORTRAN77 语言编写的子程序，则无需在主调程序中写接口块；但若使用 FORTRAN 90 语言编写程序，通常需要在主调程序中写入接口块。否则，编译容易出错。遇到下列情形之一时，必须在主调程序中写有接口块：

（1）　调用的外部过程是一个函数，且函数结果是一个数组；或函数结果值是一个

字符型，且长度不是常数，也不是假定长度（*）；或被调用过程中的哑元是一个数组片段。

（2） 实际参数是关键字变元，或实际参数是缺省的可选变元。

（3） 一个外部函数使系统中的内在操作符扩展了原有的功能。

（4） 外部过程扩展了赋值号的使用范围。

（5） 用一个类属名调用过程。

4.6 作用范围

FORTRAN 90 标准以范围单元的形式来定义名称。范围单元是一个程序或程序的一部分，在范围单元中定义一个名称，这个名称在范围单元中有效。范围单元可以是整个程序，程序单元，一个单独的语句或语句的一部分。范围定义的是一个名称被承认的范围，名称可以是常量、变量、过程、操作符或任何其它名称。

4.6.1 名称的范围

名称可以用于程序、子程序、变量、数组、哑元、命名常量、派生类型或块结构。名称有三种：全局（global）、局部（local）和语句（statement）。

本节除了说明这三种名称以外还要讨论内在过程的双重名称、公共块名称、函数结果名、派生类型成员名称、参数关键字的范围和其它实体（如语句标号、I/O 单元、操作符和赋值符号）的范围。

1. 全局名称

全局名称用来识别程序单元、公共块和外部过程。全局名称在程序的任何地方都是被承认的，所以只能在程序中定义一次。例如，如果用户在一个程序中使用了名为 Sort 的子程序，就不能再在该程序中使用名为 Sort 的公共块或函数。

2. 局部名称

局部名称用来识别变量（标量和数组）、常量、命名常量、语句函数、内部过程、模块过程、哑元过程、内在过程、一般标识符、派生类型和名称列表组的名称。派生数据类型的成员和参数关键字（哑元参数）也是局部名称。局部名称可以掩盖全局名称和同一程序单元中的其它局部名称（参数关键字、一般名称和公共块名称除外）。下面是模块中的局部名称的例子：

```
MODULE test1
CHARACTER (len=1), private :: S
CHARACTER (len=9) SSN

CONTAINS
```

```
SUBROUTINE print_part
DO j = 1,9
   S = ssn(j:j)
   print *, S
END DO
END SUBROUTINE

END MODULE
```

下面的语句声明 S 和 SSN 是

```
print_part:
```

中的哑元：

```
SUBROUTINE print_part (S, SSN)
```

这时，S 和 SSN 成为子函数 print_part 的局部变量，并掩盖了在模块剩下部分声明的 S 和 SSN 变量。为了识别模块声明部分的变量，它们不能规定为子函数的参数。

如果一个名称对某个程序单元是局部的，同样的名称即可以作为全局名称又可以作为其它程序单元中的局部名称。

3. 内在过程的双重名称

因为 FORTRAN 语言的关键字不予保留，所以用户可以创建名称和 FORTRAN 内在过程名称一样的变量、常量或过程。一旦创建了这样的名称，原来的内在函数就不能再被访问。例如，下面的代码定义了新的函数 sin：

```
SUBROUTINE sub
. . .
CONTAINS
   FUNCTION sin(x)
   . . .
   END FUNCTION sin
END SUBROUTINE sub
```

任何在子程序 sub 中对 sin 的引用都会调用上面代码中定义的内部函数，而不是原来的内在函数。

类似地，任何与同名内在过程的标准类型不同的类型声明都会产生一个局部名称。下面的例子声明了一个名为 sin 的变量：

```
CHARACTER (len = 10) sin
```

任何使用这个字符变量的程序或内部过程都不能再使用原来的内在函数。如果该变量在模块中以 PRIVATE 属性声明，模块外的程序单元仍可以使用内在函数 SIN。

4. 语句名称

语句名称是作用范围是一条语句的名称。语句名称可以出现在语句函数的语句中，或一个 DATA 语句的隐 DO 循环中，或一个数组构造器中。在语句函数语句中作为哑元出现的变量名在它出现的地方的范围就是语句范围。DO 变量的范围（必须是整数）是隐 DO 列表。例如：

```
DIMENSION x(10)
Add (a,b) = a + b
DO n = 1,10
x(n) = add(y,z)
END DO
```

在上面的例子中，a 和 b 的范围被限制在语句函数内部。N 的范围是整个 DO 循环。

5. 公共块名称

公共块名称是全局名称。因为局部名称可以和全局名称重名，在局部实体中对这个名称的引用指的是局部名称（除了在 **COMMON** 或 **SAVE** 语句中出现的引用）。当公共块在 **SAVE** 语句中命名时，它应该用斜杠围起来以和其它同名的局部变量区分。例如：

```
COMMON /ralph/ fred, ethel, lucy
character(20) ralph
SAVE /ralph/      !SAVE 的是公共块而不是变量
```

6. 函数结果名

函数结果是另一个允许出现重名的例子。对每一个在函数子程序中的 **FUNCTION** 语句或 **ENTRY** 语句都有一个结果变量。如果没有 **RESULT** 语句指定另外的变量名，结果变量和函数定义时同名。

7. 派生类型成员名称

如果一个变量是其它名称的成员，它的范围和包含该变量的名称的范围一样。例如，一个模块中定义的派生类型和模块的范围和任何使用该模块的范围一致。承认派生类型的的程序同样承认派生类型的成员。对于数组的情况也是类似的：在原来数组适用的范围，数组的一部分也有有效和承认的值。

派生类型的程序可以和其它局部变量重名，因为类型成员在没有限定词时是不能出现的，例如：

```
TYPE FOO
```

```
    INTEGER IJK
    CHARACTER L
END TYPE FOO
REAL IJK
TYPE (FOO) :: SAMPLE
```

在本例中，SAMPLE 的整数成员的引用形式为 SAMPLE%IJK。任何在 TYPE...END TYPE 声明之外单独引用 IJK 指的都是实型变量。

8. 参数关键字的范围

参数关键字是局部实体。哑元参数名称只对于它的宿主是可知的。宿主可以是定义该参数的内部过程、模块过程或过程接口块。如果通过使用 USE 或 CONTAINS 语句使这些过程和过程接口块可以被其它出现单元访问，对于这些程序单元该参数关键字也是可访问的。

内在过程的哑元参数是程序单元中的局部关键字，用它来对过程进行引用。内部过程的哑元、模块或其它过程接口块与宿主程序单元或其它可访问它们的程序单元的范围相同。作为关键字，哑元参数名只能在过程接口引用中出现一次。

9. 其它特殊情况

其它实体，例如语句标号、I/O 单元、操作符和赋值符号等都有范围的概念。对于这些实体应遵守下面的规则：

（1） 语句标号始终被认为是局部的。两个范围相同的单元不能使用相同的标号。

（2） 内在操作符（例如+, -, *, **, 或 /）是全局的，但用户定义的操作符是局部实体。特殊操作符的范围由定义该操作符的过程的范围决定。用户可以通过使用过程接口块使自定义的操作符成为全局或一般的。

（3） 赋值符号（=）是全局实体。用户可以在一个接口块中确定附加的一般赋值操作。

4.6.2 解决过程引用问题

因为 FORTRAN 允许重复使用名称，所以用户必须保证引用名称时不至于混淆。编译系统使用下面的步骤确定对一个名称的引用是否正确。

（1） 编译系统先寻找接口块。如果存在，系统在接口块中寻找一般名称。如果一般名称匹配，则过程是一般过程。

（2） 如果一般名称不匹配，或如果接口块不包含一般名称，编译系统开始寻找接口块中定义的过程。如果找到匹配的过程则它是特殊的而不是一般的。

（3） 过程如果有 INTRINSIC 属性，内在过程定义告知引用是一般还是特殊。

（4） 如果名称被声明为 EXTERNAL 过程，过程是特殊的。

如果这时还没有找到匹配，程序使用的任何模块和任何包含宿主的程序都重复上面的步骤 1 至步骤 4。如果这时还找不到匹配，编译系统继续下面的步骤：

（5） 系统寻找符合过程名称的哑元。

（6） 寻找具有该名称的内在过程。

如果过程仍未定义，编译系统假定其为外部过程。

1. 解决一般引用问题

一旦过程被认为是一般的，编译系统通过检查下面几项来检查接口块来查找匹配的项：

（1） 是否是子程序或函数。

（2） 参数的数目和特性。

（3） 如果是函数则考虑结果值的特性。

为了保证一般引用不致混淆，用户要保证每个使用该一般名称的过程至少满足下面的一个条件：

（1） 一个过程是子程序，另一个是函数，这样调用方法是不同的。

（2） 每个过程都有不同数目的非可选参数。

（3） 每个过程相应的哑元类型各不相同（种别参数不同或秩不同）。

下面的例子说明了一个模块如何定义三个独立的过程，在接口块中主程序给它们一个一般名称 DUP。尽管主程序用同一个一般名称调用这三个过程，但因为参数的数据类型不同，而且 DUP 是函数而不是子程序，所以不会产生混淆。模块 UN_MOD 给每个过程一个不同的名称。

```
MODULE UN_MOD
 !

 CONTAINS
    subroutine dup1(x,y)
    real x,y
    print *, ' Real arguments', x, y
    end subroutine dup1

    subroutine dup2(m,n)
    integer m,n
    print *, ' Integer arguments', m, n
    end subroutine dup2

    character function dup3 (z)
    character(len=2) z
    dup3 = 'String argument '// z
    end function dup3

 END MODULE
```

```
program unclear
!
! 说明如何使用一般过程引用

USE UN_MOD
INTERFACE DUP
   MODULE PROCEDURE dup1, dup2, dup3
END INTERFACE

real a,b
integer c,d
character (len=2) state

a = 1.5
b = 2.32
c = 5
d = 47
state = 'WA'

call dup(a,b)
call dup(c,d)
print *, dup(state)        !实际输出只有'S'
END
```

注意函数 DUP3 只打印一个字符，因为模块 NU_MOD 没有指定函数结果的参数长度。如果 DUP 的哑元 x 和 y 被声明为整型而不是实型，则对 DUP 的任何调用都将会引起混淆，这种情况一旦产生就会引起编译错误。

子程序定义中 DUP1，DUP2 和 DUP3 的名称必须有所区别。一般名称由接口块中的第一行指定，在本例中就是 DUP。

2. 解决特殊引用问题

只有一个一般名称可以引用多个过程。如果两个独立的非一般过程的名称相同，它们的范围也必须不同，具体见上一小节名称的范围。

4.7 联合

联合允许不同的程序单元以不同的名称访问同一个量。联合有如下三种：

1. 名称联合
2. 指针联合

3. 存储联合

指针联合已在上一章中讨论，本节主要讨论名称联合。

4.7.1 参数联合

参数是通过调用参数列表传递到函数或子程序，或者从函数或子程序中传递出来的量。实参是特定的变量、表达式、数组函数名或其它调用子程序或函数时传递给它们的数据项。在被调用的过程当中，参数是哑元参数。

对过程的引用在哑元和实际参数之间以不同的名称建立了联合。如果子程序有三个哑元 X，Y 和 Z，它可以有以下几种调用方式：

```
! 一般的方式.
CALL EXT1 (A, B, C)
! 只有一个哑元被指定
CALL EXT1 (A, B, Z=C)

! 参数可以不按顺序传递，按必须和正确的哑元联合
CALL EXT1 (Z=C, X=A, Y=B)
. . .
END

SUBROUTINE EXT1(X,Y,Z)
   REAL X, Y
   REAL, OPTIONAL :: Z
   . . .
END SUBROUTINE
```

参数 A 与哑元参数 X 或者通过括号中的位置或者通过显示的赋值来联合，如上例所示。一旦 EXT1 执行完毕并返回后，A 和 X、B 和 Y 以及 C 和 Z 就再没有联系了。

如果实参是常数、函数引用或多于一个变量的表达式，用户不能给相应的哑元赋值。如果这种情况发生，结果将是不可预料的。

如果一个实参是表达式，在和哑元联合之前会先被计算出来。一个实参是数组元素，其下标表达式在联合之前也将先被计算出来。在子程序或函数执行的过程中下标的表达式保持为常数。表 4.5 说明了实参和哑元怎样联合。

当实参是数组或数组元素时，其维数的数目和大小可以和哑元不同，但哑元必须在实际数组允许的内存序列的限制之内，否则虽然编译系统不认为这是错误，但运行的结果是不可预料的。

表 4.5　　实参和哑元的联合

实参的情况	可以联合的哑元情况
变量，数组元素，派生联系成员，表达式	变量名
数组或数组元素	数组名
数组	变量
在 CALL 语句中的交替返回指定符(*label)	星号(*)
完毕或内在过程名	在过程中任何子程序调用或函数引用单独的名称

当参数是数组时，每个元素都被传递到过程中，每次一个元素，过程为每个元素执行一次。如果数组将被传递给标量哑元时，过程必须用 **INTERFACE** 语句中声明，或必须是内在函数。

当参数是外部过程或内在过程时，实参必须用 **EXTERNAL** 语句声明，或必须是用 **INTRINSIC** 语句声明的内在函数。

4.7.2　使用联合

在模块和使用它的程序之间共享命名实体称为使用联合。通过使用联合，模块中的公共过程和实际对象在使用它们的程序中是可知和可定义的。

USE 语句允许程序访问模块中定义的实体，例如：

- 命名数据对象
- 派生类型
- 接口块
- 过程
- 一般标识符
- 名称列表组

因为使用联合扩展了常量、变量和过程到使用它们的程序，可能会遇到重名的问题。如果程序使用了多个模块，且一些模块中的实体和另一模块中实体的名称重名，则使用的程序必须作到以下两点：

（1）　不引用同名的任何实体。

（2）　重新命名重名的实体，使其不再有重名问题。

在程序中使用的模块实体的局部名称（例如变量或过程名称）必须不能和程序可访问的其它局部名称重名。如果模块中的函数或子程序和内在 FORTRAN 过程重名，则该内在过程将不能再被使用。

4.7.3　宿主联合

任何包含内部过程的程序单元称为该内部过程的宿主。宿主联合允许宿主程序、它的内部过程、模块子程序和派生类型定义访问同一实体。宿主联合在程序执行的过程中都是

有效的。

下面的例子说明了宿主和内部过程如何使用被宿主联合的实体：

```
program INTERNAL
! 说明内部子程序和 CONTAINS 语句的使用
   real a,b,c
   call find
   print *, c
contains
   subroutine find
     read *, a,b
     c = sqrt(a**2 + b**2)
   end subroutine find
end
```

在上例中，变量 a，b 和 c 通过宿主联合使内部子程序 find 可以访问它们。它们不需以参数的形式传递给内部过程。实际上，一个它们是参数的形式，对于子程序它们成为局部变量，并掩盖了宿主程序中说明的变量。

相反，当从已经定义 c 的内部子程序中返回时，宿主程序会知道 c 的值。

4.8 可执行结构和可执行块

可执行结构和可执行块是程序的重要组成部分，实际有意义的操作都是在可执行结构和可执行块中进行的。

4.8.1 概述

结构，也称为块结构，是一系列由 **CASE**, **DO**, **IF**, **WHERE**, 或 **FORALL** 语句开始，再由适当的语句中止的语句：

- CASE 结构。从 **SELECT CASE** 开始到 **END SELECT** 结束。
- DO 结构。从 **DO** 或 **DO WHILE** 开始到 **END DO** 结束。
- IF 结构。从 **IF** 开始到 **END IF** 结束。
- WHERE 结构。从 **WHERE** 开始到 **END WHERE** 结束。
- FORALL 结构。从 **FORALL** 开始到 **END FORALL** 结束。

结构之间可以组合和嵌套，例如，**DO** 循环可以包含 **IF** 结构。如果将一个结构嵌入另一个结构，则外层结构应该包含整个内层结构。

块是一系列语句或结构组成的，可以视为一个单元。一个块可以是一系列嵌套的结构，或结构中的一组语句。块甚至可以不包含任何可执行语句。下面的例子说明了结构和块的

形式：

```
IF (A .gt. 0) THEN      ! 外层 IF 结构开始标志
   SELECT CASE (b)      ! 内层 CASE 结构开始标志
      CASE (:0)
      . . .             ! 能组成一个块的任何语句.
      CASE (0:)
      . . .             ! 能组成另一个块的任何语句.
   END SELECT           ! 结束 CASE 结构
   . . .                ! 这里的任何代码，包括
                        ! CASE 结构，都组成 IF 结构内的块
   . . .
END IF                  ! 结束 IF 结构
```

在 **IF** 和 **END IF** 之间的任何语句都组成一个块，包括 **CASE** 结构。在每个 **CASE** 语句后面的语句也都组成一个块。

块结构应该遵守以下若干规则：

- 块内部可以传递控制，或在块内任何位置跳出块，但不能从块外面的语句把控制传入块内。
- 一般情况下语句将顺序执行，除非有其它语句改变执行顺序。
- 可以调用块里的函数或子程序。

4.8.2　结构命名

命名一个结构是 FORTRAN 90 中的新概念。如果用户要命名一个结构，则结构名必须在结构开始和结构结束时出现。结构名也可以在结构内的控制语句中出现，例如在 **IF** 结构中的 **ELSE** 和 **ELSE IF** 语句。下面的例子给出了一个命名为 DUTY 的 **IF** 结构：

```
DUTY: IF (DAY .EQ. 'SATURDAY') THEN
  CALL MOTHER(TIME, PHONE, NEWS)
END IF DUTY
```

如果使用固定的源程序格式，结构名称应该在第六列之后。

4.8.3　IF 结构

IF 结构最多选择结构中的一个块来执行。如果出现 **ELSE** 语句则说明结构中至少有一个块将被执行。FORTRAN 中有三种 **IF** 句子类型：

- **IF** 结构由多个代码块组成，如果满足某种条件它们将被执行。

- 如果条件满足只有一条语句被执行的逻辑 **IF** 语句。
- 根据表达式的值跳转到其它语句标号的算术 **IF** 语句。

其中算术 **IF** 将在后面讨论。

IF 结构的形式为：

IF [条件] **THEN**
　　块
[**ELSE IF** [条件]
　　块]
[**ELSE** 块]
END IF

下面的例子是一个命名的 **IF** 结构包含另外一个 **IF** 块：

```
if_construct: IF (A > 0) then
   B = C/A
     if (B > 0) then
        D = 1.0
     end if
ELSE IF (C > 0) then
   B = A/C
   D = -1.0
ELSE
   B = ABS (MAX (A, C))
   D = 0
END IF if_construct
```

IF 结构可以包含多个 **ELSE IF** 语句，但只能有一个 **ELSE** 语句。**IF** 语句之间可以相互嵌套。每一个单独的结构必须完整地存在于相邻的外部块之内。

控制一条语句执行的逻辑 **IF** 语句不是结构。下面是两个逻辑 **IF** 语句的例子：

```
IF (a .LT. b) temp = a      !逻辑 IF 语句
IF (c .EQ. b) goto 100      !逻辑 IF 语句
```

4.8.4 CASE 结构

在多种条件选择情况下，使用 CASE 结构可使程序显得直观、简短。CASE 结构把某一个量区分 成若干个值，按不同值作不同处理。但在处理复杂的、多种交叉的条件时，CASE 结构有时就不如 IF 结构方便。对具体问题要具体分析。

CASE 结构提供的程序控制和 **IF** 语句类似。**CASE** 语句列出一个表达式，后面有一个或多个代码块，满足某种条件后其中的一个代码块将被执行。各个取值范围之间不能交

迭。**CASE** 语句的形式是：

[case 结构名:] **SELECT CASE** (case 表达式)
CASE (case 值 e [, case 值]...) [case 结构名]
　　块
[CASE DEFAULT [case 结构名]
　　块]
END SELECT [case 结构名]

CASE 语句不能是分支语句的目标。用户只有从 **CASE** 结构内部才能把控制转移到 **END SELECT** 语句。

说明：

- 句中情况表达式是整型、字符型或逻辑型的标量表达式，而不能是实型和复型表达式。
- CASE 语句可以有任意语句，括号中 CASE 所有可取的值称为 CASE 索引，它们的类型应与 CASE 表达式的类型一致。对于字符来说，允许字符长度不同，但种别类型参数必须相同。
- 当表达式是数值型时，情况选择符值有下列四种形式：
  （1）　单值表示。例如 CASE（0），CASE（1，3，5，6，8）等。
  （2）　用起始值表示。形式（始值：），表示值域取包括始值在内的始值后的所有值。例如，CASE（：5），即取小于等于 5 的所有整数值。
  （3）　用终止值表示。形式（：终值），表示值域取包括始值终值在内的始值之间的所有值。例如，CASE（：5），即取小于等于 5 的所有整数值。
  （4）　用始终值表示。形式（始值：终值），表示值域取包括始值终值在内的始值到终值之间的所有值。例如，CASE（2：6），即为
  CASE（2，3，4，5，6）。

  上面四种表示方法还可以混合使用。例如：
  CASE（2：5，9）即为 CASE（2，3，4，5，9）
  注意：情况选择符的值也可以写成表达式，但不能有重复值。
- 当表达式是逻辑型时，情况选择符值也要取逻辑值。逻辑值只有两个，即.TRUE.和.FALSE.。
- 每个 CASE 语句后跟随语句块，称 CASE 块。当选择符中给出的值与情况表达式的值一致时，语句块被执行。
- 对于一个给定的 CASE 结构，CASE DEFAULT 语句最多只允许有一个，且写在所有 CASE 语句之后。若全部情况选择符的值都不符合表达式的值，且又有 CASE DEFAULT 语句，则执行该语句后的语句块，即 DEFAULT 块；否则直接转到出口。
- SELECT CASE（情况表达式）是 CASE 结构的入口语句；END SELECT 是 CASE 结构的出口语句。
- CASE 结构分为无名与有名两种形式。对有名 CASE 结构，其入口语句和出口语句的结构名必须要一致，且出口语句的结构名不可缺省。CASE 语句的结构名可

以省略。

- CASE 结构中包含的块可以是空的，也可以包含其它结构或另一个 CASE 结构，嵌套的形式与规则 与 IF 结构相同。

下面的例子计算了 NET_INCOME 表达式并且在此基础上定义 TAX_RATE 变量：

```
SELECT CASE (NET_INCOME)
  CASE (50000:)
    TAX_RATE=.28
  CASE (25000:49999)
    TAX_RATE=.14
  CASE DEFAULT
    TAX_RATE=.05
END SELECT
```

4.8.5 DO 循环控制

循环控制可以有三种形式：

（1） **DO** 结构（用循环次数和 DO 变量）。

（2） **DO WHILE** 循环（在每次执行循环之前测试逻辑表达式）。

（3） 重复 **DO** 结构。

像其它结构一样，**DO** 循环可以被命名：如果被命名，则 **END DO** 后必须有相同的名字。下面的例子是标准 FORTRAN 77 **DO** 循环和它的 FORTRAN 90 的等价形式：

```
      DO 100 n = 0, stop, step
      WRITE (*,*) 'N=',n
 100      CONTINUE

DO n = 0, stop, step
    WRITE (*,)) 'N='",n
END DO
```

DO 语句块中可以包含另外的 **DO** 结构、**IF** 或 **CASE** 结构，但整个结构必须完整地包含在 **DO** 块中。当执行循环发生下列情况之一时，退出 **DO** 循环：

（1） 测试到重复次数为零或循环控制 WHILE（e）中的 e 值为.FALSE。

（2） 属于 DO 结构的 EXIT 语句的执行。

（3） 在 DO 结构范围内，但属于外部 DO 结构的 EXIT 语句或 CYCLE 语句的执行。

（4） 从 DO 结构内部到 DO 结构外部的语句控制转移。

（5） DO 结构范围内 RETURN 语句的执行。

（6） 程序中任何地方 STOP 语句的执行，或其它原因引起的程序终止。

- **DO** 结构

DO 结构形式如下：

[name:] **DO** [标号[,]] do-variable = 下界，上界 [，增幅]

[*do* 块]

[标号] **END DO** or **CONTINUE**

DO 结构从 **DO** 语句开始并以 **END DO** 或 **CONTINUE** 语句结束。如果没有指定标号，结构必须以 **END DO** 结束。在入口与结束之间的语句组成语句块，称为 DO 块。写在 DO 块里的语句将被反复执行。在标号出现时称为标号 DO 语句；否则称无标号 DO 语句。如果指定了标号，该标号必须和 **END DO** 或 **CONTINUE** 语句的标号相同。

DO 结构分为三个阶段：循环初始化、循环执行的范围和循环的停止。在循环的初始化时应该采取如下步骤：

（1）　建立下界、上界和增量，它们都必须是整数，并且增量不能是 0。如果需要，也可以进行数据类型的转换。缺省的增量是 1。

（2）　由初始参数来初始化 **DO** 变量。

（3）　确定循环次数。循环次数可以由下界、上界和增量确定，具体公式为：

MAX(INT((上界－下界＋增幅) /增幅),0)

如果下界大于上界且增量为正或下界小于上界且增量为负则循环次数为 0。

循环执行时进行以下几步：

（1）　测试循环次数，如果为零循环终止。

（2）　如果循环次数不为零，则执行一次循环。

（3）　循环次数减 1，DO 变量增加一个步长。

下面的例子计算两个数组的张量积：

```
DO I=1, M
  DO J=1, N
    C(I, J)=SUM(A(I, J, :)*(B(:, I, J))
  END DO
END DO
```

这是一个带循环控制的块 DO 结构，其中 I，J 是 DO 变量。

- **DO WHILE** 循环

DO WHILE 循环由 **DO WHILE** 语句实现，如下例：

```
DO WHILE (input .NE. 'n')
   WRITE (*, '(A)') 'Enter y or n: '
   READ (*, '(A)') input
END DO
```

在程序执行到块内的代码之前先要计算逻辑表达式（上例中的 input .NE. 'n'）。如果逻

辑表达式的值为真，则块内程序继续执行；若表达式的值为假，则将跳过 DO WHILE 循环。例如，这类循环可以用于从文件中读取记录直到文件结束。

- 重复 DO 结构

重复 DO 结构是不带循环控制的 DO 结构，执行进入 DO 结构后顺次执行到 END DO 前的最后一个语句，然后返回到紧接 DO 语句后面的语句，再重复执行整个 DO 块。

循环控制语句CYCLE和EXIT只用于重复DO结构中，即一个CYCLE语句或一个EXIT语句属于特定的 DO 结构。如果 CYCLE 语句或 EXIT 语句引用 DO 结构名，它属于该 DO 结构；否则它属于它所出现的最内层 DO 结构。

- CYCLE 语句

CYCLE 语句的一般形式为：

CYCLE [DO 结构名]

执行 CYCLE 语句，其功能是在循环中它下面那部分的 DO 块被截断，由 CYCLE 语句重新返回到块的第一个语句开始执行。

注意：在块 DO 结构中，控制转移到 DO 终结的结果与执行属于该结构的 CYCLE 语句是一样的。

- EXIT 语句

EXIT 语句的一般形式为：

EXIT [DO 结构名]

执行 EXIT 语句，其功能是导致循环终止，退出重复结构。

单独使用 EXIT 语句将无条件地终止循环，通常将 EXIT 语句与 IF 语句结合使用，即在 DO 结构中使用语句：

IF（e）EXIT

当 e 为真时，条件满足，EXIT 语句被执行，终止循环；当条件不满足时，EXIT 语句不被执行，循环将继续进行。

下面的例子说明了 CYCLE 和 EXIT 语句的用法，文件名为 CYCLE.F90，出自 Visual FORTRAN 的例子，在/DF/SAMPLES/TUTORIAL 子目录下。

```
!   CYCLE.F90 Demonstrate the CYCLE and EXIT Statements

    INTEGER i, j, k, n
    PARAMETER (n = 10)                          ! Upper limit for loops.

    write (*,'(/A, I2)') &
   &' Controlling loops using CYCLE and EXIT, N = ', n
    write (*,900)

    Loop1: DO i = 1, n
      if (i.gt.3) EXIT Loop1
       write (*,910) i
```

```
      Loop2: DO j = 1, n
        if (j.gt.2) CYCLE Loop2
        if (i.eq.2.and.j.gt.1) EXIT Loop2
        write (*,920) j

        Loop3: DO k = 1, n
          if (k.gt.2) CYCLE Loop3
          if (i.eq.1.and.j.gt.1) EXIT Loop2     ! Leave both inner loops.
          write (*,930) k

        END DO Loop3
      END DO Loop2
    END DO Loop1
    WRITE (*,'(/A)') ' Loops completed.'

900 FORMAT(/' Loop: 1st 2nd 3rd')
910 FORMAT(11x, i2)
920 FORMAT(21x, i2)
930 FORMAT(31x, i2)
    END
```

4.9　分支选择

分支选择改变了程序从顶到底的执行顺序。分支把控制从某一个语句转到同一个程序单元的另一个有标号的目标语句。语句标号只能指向目标语句，**FORMAT** 语句，和 **DO** 结束处。

分支还可以把控制从循环内部传递到循环外面，或在同一个结构块中传递控制，但不能从结构外部传递到结构内部。只有 **IF** 结构中的语句才能传递到 **END IF** 语句。从 **IF** 结构外转移到 **END IF** 的分支是 FORTRAN 的过时特性。

4.9.1　GOTO 语句

GOTO 语句把控制传递到标号标识的语句，**GOTO** 语句的格式是：

GOTO *label*

GOTO 语句使程序的可读性和可维护性差，不符合结构化程序设计的要求。而绝大多数 **GOTO** 语句都可以用其它类型的块结构、子函数或模块编程来替换。所以应该尽量避

免使用 **GOTO** 语句。

4.9.2 CONTINUE 和 STOP 语句

CONTINUE 语句对程序的执行没有任何影响。**CONTINUE** 语句曾用来作为 **DO** 循坊 **IF** 块的结束语句。结束语句 **END DO** 和 **END IF** 使 **CONTINUE** 语句显得没有什么用处了。虽然 FORTRAN 90 的向下兼容性使 **CONTINUE** 语句仍然可用，但新编写的程序应该尽量使用以 **END DO** 结束的块 **DO** 结构。

STOP 语句会中止程序的执行，并且允许用户使用字符串或整型变量来提供程序停止的原因说明。缺省的信息是"Program terminated"（程序停止运行），缺省的错误代码是零。用户程序可以设定这些值。

下面的例子设定变量 ierror 来指示错误。

```
IF (ierror .NE. 0) THEN
   STOP 'ERROR DETECTED!'
END IF
```

如果出现错误，程序停止运行并显示：

ERROR DETECTED!

4.10 递归过程

在 FORTRAN 90 中，允许过程的自身调用，即在过程内直接或间接地调用过程自身或该过程中的 ENTRY 语句定义的函数。这种调用过程称为递归过程。递归过程有两种：递归函数和递归子程序。

4.10.1 递归函数

递归函数的一般形式为：

RECURSIVE FUNCTION 函数名（[哑元名表]）[RESULT（函数结果名）]
[说明部分]
[执行部分]
[内部辅程序部分]
END FUNCTION 函数名

递归函数的定义与外部函数的定义类似，只是在 FUNCTION 之前必须写上 RECURSIV（递归说明语句）。

在主程序中调用递归函数时，应写明递归函数的接口块。

递归函数调用的一般形式为：

PROGRAM 程序名
INTERFACE
程序接口体
END INTERFACE
接口块
…
CALL 函数名（[实元说明表]）
…
END PROGRAM

4.10.2　递归子程序

递归子程序的一般形式为：

RECURSIVE SUBROUTINE 子程序名[（[哑元名表]）]
[说明部分]
[执行部分]
[内部辅程序部分]
END SUBROUTINE 子程序名

这与子程序的定义类似，只是在 SUBROUTINE 前必须写明是 RECURSIVE。

在主程序中对递归子程序的调用，也应写明递归子程序的接口块，然后 CALL 主程序名[([实元说明表])]

下面是一个经典的递归例子：求 N 的阶乘 N!。

若用递归过程编程，可使用程序简单明了。因为：

N! =N（N-1）! =N（N-1）（N-2）! =…=N（N-1）（N-2）…2*1

如果设 N!的函数辅程序名为 FACTORIAL（N），则求（N-1）!调用 FACTORIAL（N-1），求（N-2）! 调用 FACTORIAL（N-2），…，如此一层层地递推下去，直到求得 FACTORIAL（1）=1 为止。然后，系统又自动地一层层向上回归，最后求得 FACTORIAL（N）的值，即为 N!。上述递推与回归的过程称之为递归过程。

求 N! 的程序如下：

```
PROGRAM CALL_FACTORIAL  !调用 FACTORIAL 函数的主程序
INTERFACE
RECURSIVE FUNCTION FACTORIAL(N) RESULT(F_RES)
INTEGER::N,F_RES
END INTERFACE
INTEGER::N
READ*,N
PRINT*,FACTORIAL(N)
END PROGRAM
```

```
RECURSIVE FUNCTION FACTORIAL(N) RESULT(F_RES) !求 N! 的递归函数
INTEGER::N,F_RES
IF(N.EQ.1) THEN
F_RES=1
ELSE
F_RES=N*FACTORIAL(N-1)
END IF
END FUNCTION FACTORIAL
```

从递归函数辅程序中可以看出，在其执行过程中多次调用了 FACTORIAL 函数，主程序有 PRINT 语句处调用了 FACTORIAL 函数。当主程序读入的实元值 N 与递归函数中的哑元结合后，就可打印求出 N! 值。

第五章 输入输出

本节讨论输入输出的格式和控制。

5.1 文件、设备和输入输出硬件

在 FORTRAN 的输入输出系统里，数据是以文件的形式进行存储和交换的。所有的数据来源和数据发送目标都被认为是文件，例如显示器、键盘和打印机等是外部文件，在 FORTRAN 看来和磁盘上存储的数据文件没有什么本质区别。内存中的变量也可以成为磁盘上的文件，它们通常要转化成二进制 ASCII 码表示的数字，这时称它们为内部文件。

在 FORTRAN 90 中没有增加新的输入/输出语句，但多数语句的功能都不同程度的有所增强。另外，由于 FORTRAN 90 中数据类型的扩充，也导致了输入/输出功能的扩充。

本节讨论 Visual FORTRAN 的文件、设备和输入输出（I/O）硬件的使用。本节与第二节输入输出编辑和第三节输入输出语句一起说明了 FORTRAN 中数据的输入和输出：文件和设备是存储和获取数据的地方；输入输出编辑决定了数据读写时的组织；输入输出语句决定了在输入输出时在数据上的操作。

5.1.1 逻辑设备

每一个文件，不论内部文件还是外部文件，都和一个逻辑设备相联系。可以通过一个单元符(**UNIT=**)把文件和逻辑设备联系起来。内部文件的单元符是和内部文件相联系的字符变量名。外部文件的单元符或者是由 **OPEN** 语句指定的数字，或者是预先连接作为单元符的数字，或者是星号（*）。

预先连接到某个设备的单元符不一定要打开。用户连接的外部单元在程序中止执行或被 **CLOSE** 语句关闭时断开连接。一个设备不能同时和多个文件连接，一个文件也不能同时和多个设备连接。可以打开一个已经打开过的文件，但只能更改部分连接的 I/O 选项而不能把已打开的文件或设备和另外的设备或文件相连接。

除了以下几种情况，在所有的输入输出语句中都必须使用单元符：

（1） **PRINT**，始终向标准输出（UNIT=6）写入信息；

（2） 只包含一个 I/O 列表和一个格式指示符的 **READ** 语句，它将从标准输入（UNIT=5）读入信息；

（3） 指定文件名而不是相联系的单元的文件查询 **INQUIRE** 语句。

1. 外部文件

和外部文件联系的单元符必须是整型表达式或星号（*）。整型表达式的值域是 0 到

2,147,483,640。下面的例子中把外部文件 UNDAMP.DAT 和 10 号单元符连接起来并向 10 号单元符写入数据：

```
OPEN (UNIT = 10, FILE = 'undamp.dat')
WRITE (10, '(A18,\)') ' Undamped Motion:'
```

星号（*）单元符在读数据时指的是键盘，在写数据时指的是屏幕。下面的例子是用星号向屏幕输出字符串：

```
WRITE (*, '(1X, A30,\)') ' Write this to the screen.'
```

Visual FORTRAN 有四个预先连接到外部文件（设备）的单元，如表 5.1 所示。

表 5.1　　预先连接的单元

外部单元符	描述 Description
星号（*）	始终代表键盘和屏幕
0	最初代表键盘和屏幕
5	最初代表键盘和屏幕
6	最初代表键盘和屏幕

星号（*）是唯一不能被其它文件重新连接的单元符，如果试图关闭它将会引起编译错误。单元 0，5 和 6 则可以用 **OPEN** 语句和任意文件连接。如果关闭了 0，5，6 号单元，下一次使用它们时会自动重新连接到它们最初代表的单元。

下面的例子先输出到预先连接的单元 6（屏幕），然后把单元 6 重新连接到一个外部文件并且向其写入数据，最后再连接到屏幕：

```
      REAL a, b
!     写到屏幕 (预先连接的单元 6).
      WRITE(6, '(' ' This is unit 6'')')
!     用 OPEN 语句把单元 6 和名为'COSINES'的外部文件连接起来
      OPEN (UNIT = 6, FILE = 'COSINES', STATUS = 'NEW')
      DO a = 0.1, 6.3, 0.1
        b = COS (a)
!     向文件'COSINES'写入数据.
        WRITE (6, 100) a, b
100     FORMAT (F3.1, F5.2)
      END DO
!     关闭.
      CLOSE (6)
```

```
!   重新连接到屏幕并输出
      WRITE(6,' (''  Cosines completed'')')
      END
```

2. 内部文件

和内部文件联系的单元符是字符串或字符数组。内部文件有两种：

（1）　字符变量、字符数组元素或只有一个记录的非字符数组元素，记录的长度和变量、数组元素或非字符数组元素相同。

（2）　字符数组、字符派生类型或非字符数组是一组元素，其中每个元素都是一个记录。记录的顺序和数组元素或类型成员的顺序相同，并且记录的长度就是一个数组元素或派生类型成员的长度。

使用内部文件时应该注意以下几点：

（1）　使用格式化的 I/O，包括用由格式定义指定的格式化 I/O 和直接列表 I/O（系统对直接列表 I/O 和顺序格式化 I/O 同等对待）。不能使用名称列表。

（2）　如果字符变量是可分配数组或可分配数组的一部分，该数组必须在当作内部文件以前进行分配。如果字符变量是指针，它必须和目标相联合。

（3）　只能使用 **READ** 和 **WRITE** 语句，不能对内部文件使用文件连接（**OPEN**, **CLOSE**）语句、文件定位（**REWIND**, **BACKSPACE**）语句或文件查询（**INQUIRE**）语句。

用户可以像对外部文件一样对内部文件用 **FORMAT** I/O 语句和直接列表 I/O 语句读写内部文件。在一条 I/O 语句执行之前内部文件定位于文件的第一个记录之前。对于内部文件，用户可以用输入输出系统的格式化能力来在外部字符表示值和 FORTRAN 内部内存表示值之间转化。也就是说，从内部文件读出内容是把 ASCII 码转化成数值、逻辑或字符量；写入一个内部文件是把这些量转化为 ASCII 码来表示。这个特性使用户可以不必知道文件的具体格式就可以读出一串字符，检查这串字符并解释它的内容。同时还允许用户在对话框中输入字符串而程序可以解释为数字。

如果写入内部文件时一个记录没有写全，则该记录剩余部分用空格填补。在下面的例子中，x 和 fname 指定了内部文件：

```
      CHARACTER(10) str
      INTEGER n1, n2, n3
      CHARACTER(14) fname
      INTEGER   i

      str = " 1   2   3"
!  List-directed READ sets n1 = 1, n2 = 2, n3 = 3.
      READ(str, *) n1, n2, n3
      i = 4
!  Formatted WRITE sets fname = 'FM004.DAT'.
```

```
      WRITE (fname, 200) i
200   FORMAT ('FW', 13.3, '.DAT')
```

5.1.2 文件

FORTRAN 支持两种文件访问方式（连续访问方式和直接访问方式）和三种文件结构（格式化结构，非格式化结构和*二进制结构*）。连续访问和直接访问文件可以是上面的任意一个格式。每种文件类型对于用户开发的应用程序都各有优点。

1. 格式化文件

如果要创建一个格式化文件，可以通过使用 **FORM='FORMATTED'**选项打开这个文件，或在创建连续文件时省略 FORM 参数。格式化文件的记录以 ASCII 码字符来保存；以二进制文件格式保存的数据也将转化为 ASCII 格式。每个记录以 ASCII 码的 CR（0D）和 LF（0A）字符结束。

如果用户要查看数据文件内容，可以使用格式化文件。还可以直接在文本编辑器中载入一个格式化的文件，也就是说，实际数据看起来像类似字符串的数字，而非格式化文件或二进制文件看起来像一组十六进制的字符。

2. 非格式化文件

如果要创建一个非格式化文件，可以通过使用 **FORM='UNFORMATTED'**选项打开这个文件，或在创建直接访问文件时省略 FORM 参数。非格式化文件由一系列物理块组成的记录组成。每个记录都含有很类似于程序内存中使用的表示方法存储的量。所以在输入输出时几乎不需作转化。

存储同样信息的非格式化文件和格式化文件相比，非格式化文件访问速度更快，文件格式更紧凑。但是，如果文件中含有数字，那么用户不能直接在文本编辑器中读出它们。

3. 二进制文件

可以通过指定 **FORM='二进制'**来创建二进制文件。二进制文件是最紧凑的存储格式，并且最适合存储大量数据。

4. 连续访问文件

连续访问文件中的数据必须按顺序一个记录一个记录的访问（除非用 **REWIND** 或 **BACKSPACE** 语句改变文件位置）。有些输入输出的方法只适用于连续访问文件。内部文件也必须是连续访问文件。用户必须为连续设备（连续设备是不允许显示动作的物理存储设备，例如键盘、屏幕和打印机）联系连续访问文件。

5. 直接访问文件

直接访问文件可以以任意顺序读写。文件中的记录被从 1 开始连续标号。所有的记录都有 **OPEN** 语句 **RECL=**选项指定的长度。对直接访问文件的访问通过指定要访问的记录来实现。如果用户需要随机访问 I/O 可以使用直接访问文件。常见的随机访问应用程序的例子是数据库。

所有文件都由记录组成。每个记录都是文件中的一个实体，文件中所有的记录类型都是相同的。

在 FORTRAN 中，向记录中写入的字节数应该小于或等于记录的长度。对于二进制文件，单独的 READ 或 WRITE 语句可以读或写传递字节数需要的任意多个记录。不完整的非格式化或二进制记录将以未定义字节填充。

6. 格式连续文件

格式连续文件是一系列连续写并按文件中出现的顺序读的格式化记录。记录的长度可以不同，也可为空记录。记录由 ASCII 码的 CR（0D）和 LF（0A）分开，如图 5.1 所示。

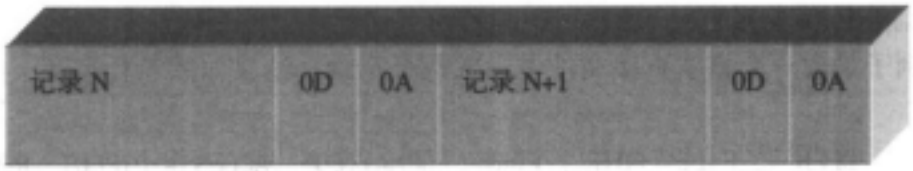


图 5.1　格式连续文件中的格式记录

下面的例子是向连续格式连续文件中写入三个记录，输出结果如图 5.2 所示。

```
      OPEN (3, FILE='FSEQ')
!     FSEQ is a formatted sequential file by default.
      WRITE (3, '(A, I3)') 'RECORD', 1
      WRITE (3, '()')
      WRITE (3, '(A11)') 'The 3rd One'
      CLOSE (3)
      END
```

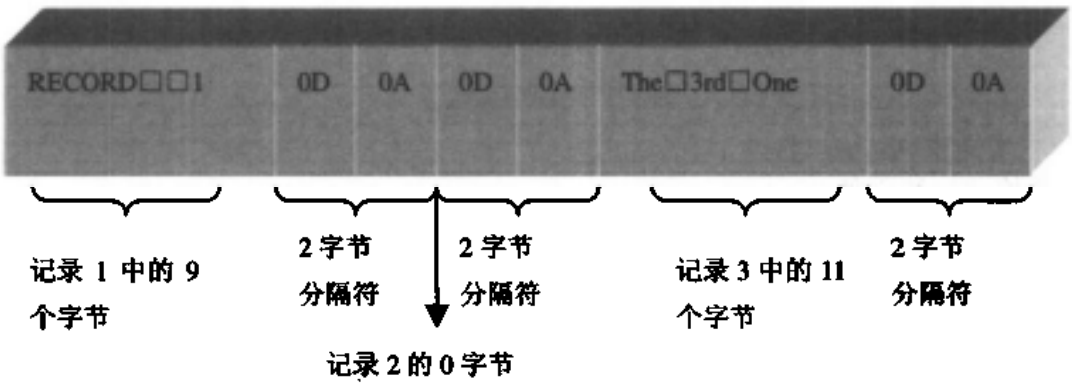


图 5.2　格式连续文件

7. 格式直接文件

在格式直接文件中，所有记录的长度都相同并且可以以任意顺序读写。记录的长度由 OPEN 语句中的 RECL=选项指定，该长度应该大于或等于最长的记录中的字节数。

CR 和 LF 是分隔符，不包括在 RECL=中。一旦某个直接访问记录被写入就不能再删

除它，但可以覆盖这个记录。

在输出到一个格式直接文件的过程中，如果数据没有占满一个记录，则编译系统会将剩下的位置补上空格。这些空格保证文件只包含长度相同的完整的记录。在输入时，如果输入列表和格式比记录中包含的数据多，则编译系统在缺省状态下也会将输入剩下的位置补上空格。

可以通过在打开文件的 **OPEN** 语句中设置 **PAD='NO'**来避免填补空格。如果 **PAD='NO'**，输入记录必须有输入列表和格式所要求的一样多的数据。否则会产生错误。**PAD='NO'**对输出没有影响。

下面的例子是向一个格式直接文件中写入两个记录，结果如图 5.3 所示。

```
OPEN (3,FILE='FDIR', FORM='FORMATTED', ACCESS='DIRECT',RECL=10)
WRITE (3, '(A10)', REC=1) 'RECORD ONE'
WRITE (3, '(I5)', REC=3) 30303
CLOSE (3)
END
```

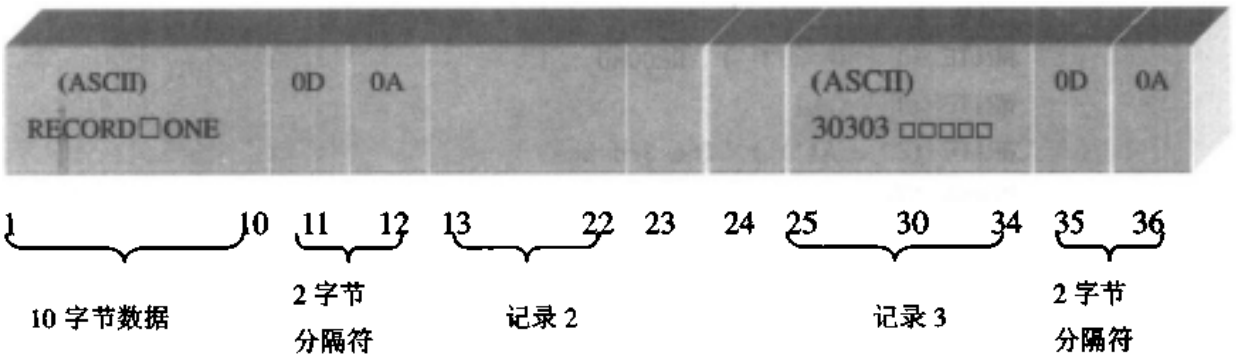


图 5.3 格式直接文件

8. 非格式连续文件

非格式连续文件在各种不同的平台上的组织只有很小的区别。这里讨论 Visual FORTRAN 创建的非格式连续文件。

非格式连续文件中的记录的长度可以不同。非格式连续文件以 130 或少于 130 字节为一个物理块进行组织。每个物理块由用户传递的数据（最多 128 字节）加上编译系统加入的两个 1 字节长的“长度字节”。长度字节说明了每个记录的起始位置。

一个逻辑记录指的是由一个或多个物理块，如图 5.4 所示。逻辑记录的大小可由用户任意指定，编译系统会相应地使用需要数量的物理块。

当用户创建一个包含多个物理块的逻辑记录时，编译系统把长度字节置为 129 以表示在当前物理块的数据和下一个物理块相连接。例如，如果写入了 140 字节的数据，则逻辑记录的结构如图 5.5 所示。

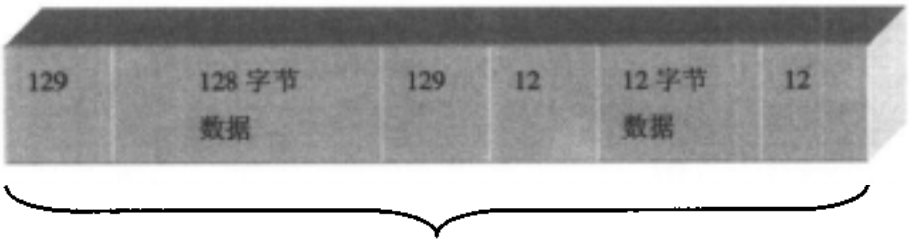
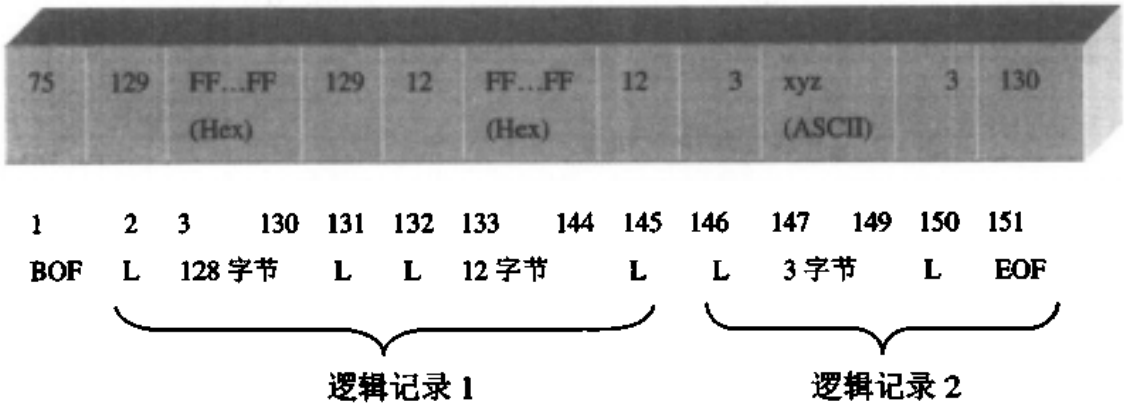


图 5.4　非格式连续文件中的逻辑记录

非格式连续文件中的第一个和最后一个字节是保留字节：第一个字节的值为 75，最后一个字节的值为 130。FORTRAN 使用这些字节作为检错和文件结束的参考。



BOF　文件起始位（75）
L　物理块长度（0<=L<=129）
EOF　文件结束位（1）

图 5.5　非格式连续文件

下面的程序创建了如图 5.5 所示的非格式连续文件：

```
!     文件缺省时是连续的
!     -1 在十六进制中是 FF FF FF FF
!
      CHARACTER xyz(3)
      INTEGER(4) idata(35)
      DATA       idata /35 * -1/, xyz /'x', 'y', 'z'/

!
```

```
!   打开这个文件并写出一个 140 字节的记录：
!   128 字节 (块) + 12 字节 = 140 (IDATA), 3 字节 (XYZ).
      OPEN (3, FILE='UFSEQ',FORM='UNFORMATTED')
      WRITE (3) idata
      WRITE (3) xyz
      CLOSE (3)
      END
```

9. 非格式直接文件

非格式直接文件是一系列非格式的记录。用户可以任意顺序读写记录。记录的长度都相同，长度由 **OPEN** 语句中的 **RECL=**选项指定。没有字节分隔符或其它表示记录结构的字节。

可以把部分文件写成非格式直接文件的形式。Visual FORTRAN 用 ASCII 的 NULL 字符把这些记录填补成固定格式长度。文件中没有写的记录中的数据是未定义的。

下面的程序创建了和图 5.6 所示结构一样的非格式直接文件：

```
OPEN (3, FILE='UFDIR', RECL=10,&
   & FORM = 'UNFORMATTED', ACCESS = 'DIRECT')
WRITE (3, REC=3) .TRUE., 'abcdef'
WRITE (3, REC=1) 2049
CLOSE (3)
END
```

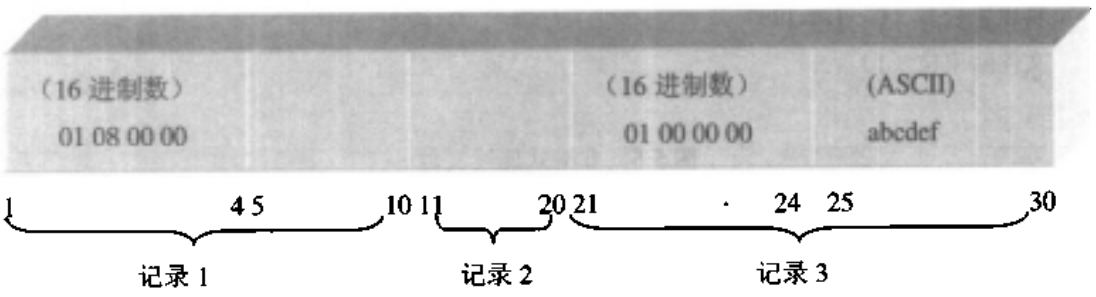


图 5.6 非格式直接文件

上例中第 1 至第 4 字节存放十进制整数 2049；第 5 至第 10 这 6 个字节是未定义的（在 Microsor FORTRAN 中是 0）；第 10 至第 20 字节是 10 个字节未定义数据；第 21 至第 24 字节存放逻辑值.TRUE.；最后 6 个字节存放字符“abcde”。

10. 二进制连续文件

二进制连续文件是一系列按同一顺序和同样二进制数个数来读写的值。其中没有记录边界，没有说明文件结构的特殊字节。数据读写时长度和形式都不改变。对于任何输入输

出数据，内存中的字节序列就是文件中的字节序列。

下面的程序创建了如图 5.7 所示的二进制连续文件。

```
!     注释：  07 是响铃字符
!     缺省时是连续的
!
      INTEGER(1) bells(4)
      CHARACTER(4) wys(3)
      CHARACTER(4) cvar
      DATA bells /4*7/
      DATA cvar /' is ',wys /'What',' you',' see'/

      OPEN (3, FILE='BSEQ',FORM='二进制')
      WRITE (3) wys, cvar
      WRITE (3) 'what ', 'you get!'
      WRITE (3) bells
CLOSE (3)
END
```

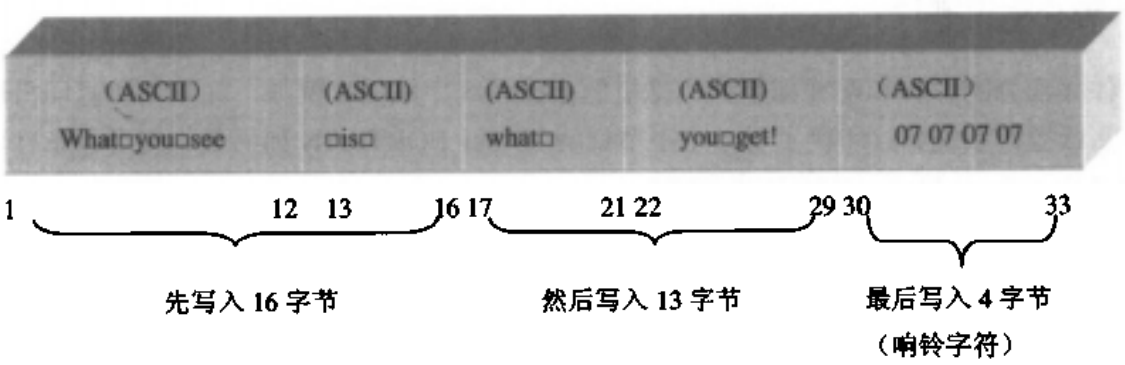


图 5.7　二进制连续文件

11. 二进制直接文件

二进制直接文件存储一系列二进制数记录，它们可以按任何顺序访问。文件中每个记录的长度相同，长度由 **OPEN** 语句中的 **RECL=**选项指定。在二进制直接文件中可以写入部分记录，记录中未使用的部分将以未定义数据填充。

一个单独的读或写操作可以通过把操作延续到下一个记录来传输多个记录。而对一个非格式直接文件进行这样的操作会产生错误。对非格式直接文件有效的输入输出操作如果用到二进制直接文件上会产生同样的结果，所提供的操作不依赖于不完整记录中填充的 0。

下面的程序创建了如图 5.8 所示的二进制直接文件：

```
      OPEN (3, FILE='BDIR',RECL=10,FORM='二进制',ACCESS='DIRECT')
      WRITE (3, REC=1) 'abcdefghijklmno'
```

```
WRITE (3) 4,5
WRITE (3, REC=4) 'pq'
CLOSE (3)
END
```

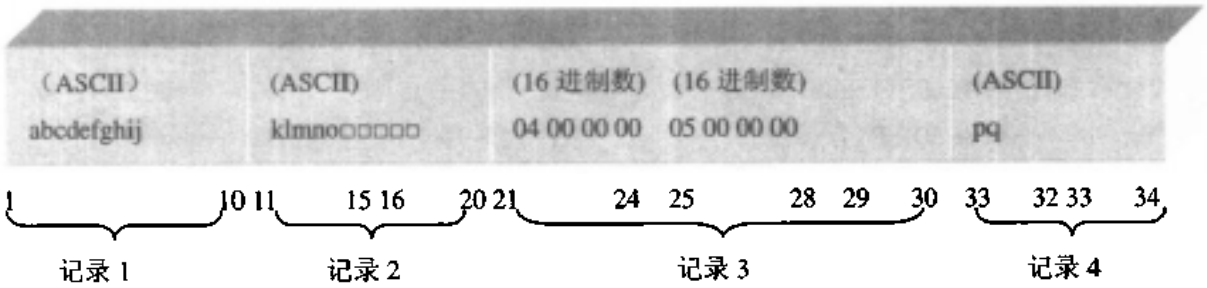


图 5.8 二进制直接文件

上例中，前 15 个字节存放字符串“abcdefghijklmno”，由于没有占满记录，第 16 至 20 字节以 5 字节的未定义数据填充；21 至 28 字节存放整数 4 和 5，同样填入 2 字节未定义数据；31 和 32 字节是字符“pq”，随后是 8 字节的未定义数据。

5.1.3 输入输出硬件

硬件大部分的设置和配置都是由计算机的操作系统完成的。例如，如果要和打印机联系应该阅读操作系统和打印机手册。本小节说明 Visual FORTRAN 如何使用物理设备打印 Microsoft Developer Studio 中的文本和图形。

1. 打印

在 Microsoft Developer Studio 中打印的最简单的方法是从菜单中选择文件（File）菜单中的打印（Print）命令，然后系统会提示用户要打印的文件和文件名。另外如果带有.F90, .FOR, .FD, .FI, 或 .RC 文档没有打开，则将文档从“我的电脑”或“Windows 资源管理器”拖到“打印机”文件夹中的打印机。 打印文档时，打印机图标出现在任务栏上的时钟旁边。如果图标消失，则表明文档已经打印完毕。可以在桌面上创建打印机的快捷方式，这样便于访问打印机。

如果用户在屏幕上绘制了图形并想把屏幕上的内容打印出来，最简单的方法是按组合键【ALT+PRINT SCREEN】，则当前活动窗口被复制到剪贴板。如果按的只是【PRINT SCREEN】则复制的是整个屏幕。

复制屏幕后打开画笔程序后选择粘贴（Paste），剪贴板中的图形就粘贴到画笔程序上了，这时可以选择保存为位图（.BMP）文件也可以选择打印输出。

2. 物理设备

没有说明具体文件或 I/O 设备的输入、输出语句是从标准输入（键盘）输出（屏幕）设备进行读写。如果要完成对键盘和屏幕之外的输入输出就应该指定设备名称作为文件读写的来源。

一些物理设备的名称由用户机器的操作系统决定，其它的由 Visual FORTRAN 识别，如表 5.2 所示。大部分设备的扩展名将被忽略。

表 5.2　输入输出设备名称

设　备	描　述
CON	控制台（即屏幕，标准输出设备）
PRN	打印机
COM1	1#串行通信口
COM2	2#串行通信口
COM3	3#串行通信口
COM4	4#串行通信口
LPT1	1#并行通信口
LPT2	2#并行通信口
LPT3	3#并行通信口
LPT4	4#并行通信口
NUL	空（NULL）设备。放弃输出，不包含任何输入
AUX	1#串行通信口
LINE '	1#串行通信口
USER '	标准输出
ERR '	标准错误
CONOUT$	标准输出
CONIN$	标准输入

如果用户使用了这些名字的扩展名，例如 LINE.TXT，FORTRAN 将会写入一个文件而不是相应的设备。

下面是打开物理设备作为单元的例子：

```
OPEN (UNIT = 4, FILE = 'PRN')
OPEN (UNIT = 7, FILE = 'COM2', ERR = 100)
```

5.2　输入输出编辑

本节讨论以下输入输出编辑主题：

- I/O 列表

I/O 列表提供将要传输的数据的信息。

- I/O 编辑的方法

说明如何在内存中的变量和外部设备和外部文件之间传递数据。

- 格式化 I/O

提供格式化输入输出信息。

- 直接列表 I/O

使用户可以不使用 **FORMAT** 语句就能在 I/O 列表中读写数据。I/O 由列表中项目中的数据类型和数目控制。

- 名称列表 I/O

使用户可以在名称列表组中确定一个或多个数据项，于是这些数值的读写可以由一个单独的 I/O 语句来完成。

5.2.1 I/O 列表

数据传输语句（**READ**、**WRITE** 和 **PRINT**）需要如何传递数据和传递什么数据的信息。其中传递什么数据由 I/O 列表（iolist）中的列出的将要读写的项确定。指定 I/O 列表可以有以下方法：

1. 无实体

I/O 列表可以是空列表。结果记录要么是零长度，要么是只包含填充字符。这可以用于写一个作为占位符的记录。如果使用只有字符串而没有 I/O 列表的格式，结果记录将包含这个字符串。例如：

```
WRITE (UNIT=7, FMT='(2I8)')
WRITE (4, "('string')")
```

2. 变量名、数组元素名、派生类型名、派生类型元素名或子字符串名

```
!  I/O 列表中的一个变量和数组元素：
       REAL b(99)
       READ (*, 300) n, b(n)
!  n 和 b(n) 是 I/O 列表
 300   FORMAT (I2, F10.5)
!  FORMAT 语句，说明输入数据的形式
!  I/O 列表中的一个派生类型及元素
       TYPE YOUR_DATA
         REAL a
         CHARACTER(30) info
         COMPLEX cx
       END TYPE YOUR_DATA
       TYPE (YOUR_DATA) yd1, yd2
       yd1.a = 2.3
       yd1.info = "This is a type demo."
       yd1.cx = (3.0, 4.0)
```

```
      yd2.cx = (4.5, 6.7)
!  I/O 列表在 (*,500) 语句之后.
      WRITE (*, 500) yd1, yd2.cx
!  FORMAT 语句，说明输出数据的形式
500   FORMAT (F5.3, A21, F5.2, ',', F5.2, ' yd2.cx = (', F5.2,
       ',',F5.2, ' )')
!  输出结果如下：
!  2.300This is a type demo 3.00, 4.00 yd2.cx = ( 4.50, 6.70 )
```

3. 指定数组名或数组片段

没有下标的数组指的是按列存储的所有数组元素。没有引用特定数组元素的假定大小哑元数组不能在 I/O 语句的列表中出现，但可分配数组和假定大小哑元数组可以出现在 I/O 列表中。

```
!  I/O 列表中的数组:
      INTEGER handle(5)
      DATA handle / 5*0 /
      WRITE (*, 99) handle
99    FORMAT (5I5)
!  I/O 列表中的数组片段.
      WRITE (*, 100) handle(2:3)
100   FORMAT (2I5)
```

4. 表达式

WRITE 和 **PRINT** 语句中的输出列表可以包含表达式。表达式的类型可以是数值、逻辑、字符或派生类型（操作符可以是为派生类型定义的）。例如：

```
PRINT *,'(I5)', 2*3    ! I/O 列表是表达式 2*3.
```

5. 名称列表

指定名称列表后可以对其中的所有变量用一个 I/O 语句进行读写：

```
! I/O 名称列表:
     INTEGER int1
     LOGICAL log1
     REAL r1
     CHARACTER (20) char20
     NAMELIST /mylist/ int1, log1, r1, char20
     int1 = 1
```

```
      log1 = .TRUE.
      r1 = 1.0
      char20 = 'NAMELIST demo'
      OPEN (UNIT = 4, FILE = 'MYFILE.DAT', DELIM = 'APOSTROPHE')
      WRITE (UNIT = 4, NML = mylist)
! 写下面的内容:
! &MYLIST
! INT1 = 1,
! LOG1 = T,
! R1 = 1.000000
! CHAR20 = 'NAMELIST demo '
! /
      REWIND(4)
      READ (4, mylist)
```

6. 隐 DO 列表

隐 DO 列表和一般的 DO 循环类似。起始、停止和增量参数决定了循环次数，*dovar*（在适当的地方）可以用来作数组元素指定符。下面的例子中 I/O 列表告知 Visual FORTRAN 向名为 mydata 的数组的第 6 至第 10 个元素输入数据。

```
      INTEGER mydata(25)
      READ (10, 9000) (mydata(I), I=6,10,1)
9000  FORMAT (5I3)
```

5.2.2 I/O 编辑的方法

I/O 编辑告知 Visual FORTRAN 的 I/O 系统如何在内存中的变量和外部设备和外部文件之间传递数据。有如下三种方法：

1. 格式化 I/O

在显式格式化 I/O 中，用户应严密地指定数据的组织。I/O 语句使用的格式指定符是希望的格式字符表达式或包含格式的 **FORMAT** 语句的标号。例如：

```
      WRITE (*, '(I3)') int1
      WRITE (*, 9000) int2
9000  FORMAT ( I5)
```

2. 直接列表 I/O

直接列表 I/O 中，用户可以不需要显式的格式或引用 **FORMAT** 语句而直接对在 I/O

列表中的数据项进行读写。这里 I/O 语句中的格式指定符是星号（*），例如：

```
WRITE (6,*) int1
```

3. I/O 名称列表

在 I/O 名称列表中，可以通过在用户使用 **NAMELIST** 语句创建的名称列表组中指定的一个或多个变量进行数据读写。格式指定符是名称列表名。例如：

```
NAMELIST /example/ int1, int2
WRITE (*, example)
```

5.2.3 格式化 I/O

格式信息包含在格式列表中，并由 **PRINT**, **READ** 和 **WRITE** 语句使用。这些语句可以包含格式列表本身，或包含有格式列表的 **FORMAT** 语句标号，或包含一个变量，该变量用来设置编辑列表或语句标号。

格式列表是一系列格式描述符，之间用逗号隔开。这些格式描述符描述了将要传输的数据，例如要被读写的数字，数据类型和长度。下面是 **FORMAT** 和 **WRITE** 语句中的格式列表的例子：

```
!  FORMAT 语句中的格式列表在连字符下面
!  连字符:     ------------------------
100   FORMAT (' A = ', I5, 'B = ', F7.2)
!
!  WRITE 语句中的格式列表在连字符下面
!  连字符:     -------------------
      WRITE(*, '(F8.5, 2I3, A20)') REAL1, INT1, INT2, "format list example"
```

格式列表（包括外部的括号）是字符常量，在 **READ** 或 **WRITE** 语句中出现时应被单引号或双引号包括。当格式列表在 **FORMAT** 语句中出现时整个格式列表并不用引号。编辑列表也可以包含另一个格式列表。在格式列表最外层括号中最多允许 8 层嵌套的括号。

格式列表由可重复和不可重复的编辑描述符组成。可重复编辑描述符描述了数据项，例如：2I3 指定编辑描述符 I3 重复两次，这样可以写两个 3 位的整数。不可重复编辑描述符可以改变数据格式，例如：SP 使正数输出时带有加号。关于可重复和不可重复的编辑描述符的详细用法见后。

指定包含格式列表的格式可以用下面一些方法：

1. **FORMAT** 语句标号

如果指定 I/O 中的 **FORMAT** 语句标号，在 **FORMAT** 语句中的格式列表规定了数据

的格式，例如：

```
      WRITE (*, 9000) int1, real1(3), char1
9000  FORMAT (I5, 3F4.5, A16)
!  I5, 3F5.2, A16 是格式列表
```

2．整型变量名

引户可以使用 **ASSIGN** 语句把 **FORMAT** 语句的标号和一个整型变量相联系，随后使用这个变量来引用 **FORMAT** 语句。在下面的例子中，整型变量名 MYFMT 指的是标号为 9000 的 **FORMAT** 语句：

```
      ASSIGN 9000 TO MYFMT
9000  FORMAT (I5, 3F4.5, A16)
!  I5, 3F5.2, A16 是格式列表.
      WRITE (*, MYFMT) iolist
```

3．字符表达式或变量

用户可以把格式表达式写成字符表达式并在 **READ**, **WRITE** 或 **PRINT** 语句中使用，例如：

```
      WRITE (*, '(I5, 3F5.2, A16)')iolist
! I5, 3F4.5, A16 是格式列表.
```

下面的例子中，格式列表被赋给名为 MYLIST 的 80 个字节长的变量：

```
CHARACTER(80) MYLIST
MYLIST = '(I5, 3F5.2, A16)'
WRITE (*, MYLIST) iolist
```

4．数组或数组元素

如果把格式列表写成字符表达式并将其赋值给一个数组，可以使用数组作为格式指定符。

```
CHARACTER(6) array(3)
DATA array / '(I5', ',3F5.2', ',A16)' /
WRITE (*, array) iolist
```

如果把格式列表写成字符表达式并将其赋值给一个数组元素，可以使用数组元素作为格式指定符。在下面的例子中，**WRITE** 语句中使用数组元素 array(2)作为传输数据的

格式指定符：

```
CHARACTER(80) array(5)
array(2) = '(I5, 3F5.2, A16)'
WRITE (*, array(2)) iolist
```

当非字符数组元素被认为和同样长度的字符变量等价时也可以作为格式指定符。

5.2.4　可重复编辑描述符

可重复编辑符告知 Visual FORTRAN 输入输出系统如何解释 **FORMAT** 语句中的数据项。它可以根据描述 I/O 列表中数据项的需要重复任意次。

```
!  这个 WRITE 语句输出 3 个整数和 4 个实数，三个整数各占 5 个字符的位置
!  两个实数的格式为 F7.2，另两个实数的格式为 F7.5
      INTEGER(2) int1, int2, int3
      REAL(4) r1, r2, r3, r4
      DATA int1, int2, int3 /143, 62, 999/
      DATA r1, r2, r3, r4 /2458.32, 43.78, 664.55, 73.8/
      WRITE (*,9000) int1, int2, int3, r1, r2, r3, r4
9000  FORMAT (3I5, 2(1X, F7.2, 1X, F5.2))
```

输出结果如下：

```
  143   62   999 2458.32 43.78   664.55 73.80
```

重复指定是一个非零的无符号整数常量*或一个被尖括号括起来的整型表达式*，它告知 Visual FORTRAN 输入输出系统数据项要重复的次数。例如<J+K>I5 就说明 I5 格式的数据项应该重复 J+K 次。

可重复编辑描述符有：

（1）　整数编辑（I）
（2）　二进制（B）、十进制（O）和十六进制（Z）编辑
（3）　没有指数的实型编辑（F）
（4）　有指数的实型编辑（E）
（5）　双精度实型编辑（D）
（6）　工程计数法编辑（EN）
（7）　科学计数法编辑（ES）
（8）　逻辑编辑（L）
（9）　字符编辑（A）
（10）　普通编辑（G）

其中 I（整数）、B（二进制）、O（十进制）、F（单精度实型）、E（有指数的实型）、EN（工

程计数法实型）、ES（科学计数法实型）、G（普通）和 D（双精度实型）编辑描述符用于数字数据的输入输出。对于数字数据编辑描述符应遵守以下规则：

（1） 输入时，全部是空格的区域始终解释为 0。拖后空格和分散空格的解释由 BN 和 BZ 两个编辑描述符和 OPEN 语句中 BLANK＝选项来控制。正号（＋）是可选的，但对指数例外。文件系统中对不位置的记录填补的空格不视为有效。

（2） F，E，EN，ES，G 和 D 编辑中的输入中，输入区域的显式小数点将覆盖编辑描述符中对小数点位置的指定。输出时小数点位置遵守编辑描述符的规定。例如：

```
      READ (*, 100) x
100   FORMAT(F4.2)
```

如果指定 x=123.4，输入中的小数点优先，所以读入的数据为 123.4 而不是 23.40。再如：

```
      WRITE(*, 200) x
200   FORMAT (F6.2)
```

这时，如果指定 x=123.4，输出格式优先，所以输出为 123.40。

输出时，如果字符的数目超过了指定的区域宽度或指数宽度，则整个区域将被写为星号（*）。如果实数小数点后包含的数字位数比区域允许的多，结果会四舍五入。例如：

```
      WRITE(*, 200) x
200   FORMAT (F4.1)
```

如果指定 x=123.4 则输出将是****，因为在 123.4 中有 5 个字符，超过了规定的 4 个字符的宽度。如果 x＝1.23，输出将是 1.2。

（3） 输出时，数字是右对齐的，如果输出数字少于规定宽度前面会补上空格。

（4） 当用 I，B，O，Z，F，E，EN，ES，G，D 或 L 编辑描述符时，输入区域可以包含逗号，表示结束该区域。下一个区域开始于逗号后的第一个字符。丢掉的字符没有意义。但在使用如 T，TL，TR 或 nX 等显式编辑描述符时不能使用这个性质，因为这样将会改变数据中的字符位置。

（5） 规定复数的格式时需要两个连续的 F，E，G 和 D 编辑描述符，而且两个编辑符可以不同，第一个指定实部，第二个指定虚部。

（6） 在可重复编辑描述符之间可以出现不可重复编辑描述符。

B、O、Z 和 EN、ES 是 FORTRAN 90 新增的编辑描述符，下面将分别它们，其它编辑描述符的使用可以参见 FORTRAN 77 的书籍或 Visual FORTRAN 的在线帮助。

1. 二进制（B）、十进制（O）和十六进制（Z）编辑描述符

语法：

B*w*[.*m*], **O***w*[.*m*], **Z***w*[.*m*]

二进制（B）、十进制（O）和十六进制（Z）编辑描述符不能包含小数点或正负号。相应进制的数据只能由允许的数字组成：B 描述符允许 0 和 1，O 描述符允许 0～7，Z 描述符允许 0～9 和 A～F。二进制（B）、十进制（O）和十六进制（Z）编辑描述符可以是用于整型、*字符型*、*实型或逻辑型*。

因为没有负号，所以 B，O 和 Z 的负值应根据所使用编码转换（如 2 的补码）来表示。对二进制、十进制和十六进制的译码，尤其是负数的译码是和 CPU 有关的。使用 B，O 和 Z 编辑描述符和以 B，O 和 Z 存储数据的程序可能不能直接在计算机之间移植。

w 是字符的宽度，*m* 指定了输出时至少输出的位数，在输入时不起作用。*M* 的缺省值为 1。如果输出少于 *w* 指定的宽度，开始将填补空格。对于二进制数，如果开始以 0 填补可读性会更好一些，例如 00010101 显示了 10101 所有的 8 位。可以使用类似 B8.8， B16.16， B32.32 等的格式强迫开始以 0 填补。

使用 B，O 和 Z 编辑可以在二进制、十进制和十六进制形式的外部数据和内部数值表示之间互相转换。内部数据的每个字节对应 8 位二进制字符，3 个十进制字符和 2 个十六进制字符。例如，255 就是二进制字符 11111111、十进制字符 377 和十六进制字符 FF 的输出。类似地，一个 INTEGER(4)是数据存储中的 32 个二进制字符、2 个十进制字符或 8 个十六进制字符的输出。

区域宽度 *w* 指定要读写数据字符长度。如果 *w* 被忽略，缺省区域宽度为 8*n 个二进制字符，3*n 个十进制字符或 2*n 个十六进制字符。n 是 I/O 列表中的数据项的字节长度。例如，一个为 INTEGER(2)的值开始是由 4 个十六进制字符表示的。

对于数值型和逻辑型，输出的字节是按有效位的顺序：有效的在最左边，无效的在最右边。例如 INTEGER(2)型值为 10 的数据，以十六进制输出时会先输出 A 再跟 3 个空格。

表 5.3 是截断或填充应用的规则，表中 n 的值为 I/O 列表中的数据项的字节长度。

表 5.3　　输入输出中截断或填充应用规则

操　作	规　则
输出	如果 w > 8*n (二进制), 3*n (十进制), 或 2*n (十六进制),字符是右对齐的，开始可以补上空格 如果 w <= 8*n (二进制), 3*n (十进制), 或 2*n (十六进制), 区域将填以星号 如果 m > 8*n (二进制), 3*n (十进制), 或 2*n (十六进制), 字符是右对齐的，开始可以补上 0 如果 m < 8*n (二进制), 3*n (十进制), 或 2*n (十六进制), m 没有作用
输入	如果 w >= 8*n (二进制), 3*n (十进制), 或 2*n (十六进制), 取最右边的 the 8*n (二进制), 3*n (十进制), or 2*n (十六进制)个字符 如果 w < 8*n (二进制), 3*n (十进制), or 2*n (十六进制), 读最开始的 w 个字符，参数 m 对输出没有影响

下面的例子说明了十六进制的输出编辑：

```
CHARACTER(2)  alpha
```

```
INTEGER(2) num

alpha = 'YZ'
num   = 3035

WRITE (*, '(Z4.4, 1X, Z2, 1X, Z6)')   alpha, alpha, alpha
WRITE (*, '(Z4.4, 1X, Z2, 1X, Z6)')   num, num, num
WRITE (*, '(B16.16, 1X, B2, 1X, B6)') num, num, num
WRITE (*, '(O5.5, 1X, O2, 1X, O6)')   num, num, num
```

输出结果为：

```
5A59 **   5A59
0BDB **    BDB
0000101111011011 ** ******
05733 **   5733
```

作为输入的例子，记录应该按下面来读：

编辑描述符	读的值
Z	YZ
Z2	Z
Z6	YZ

2. 工程计数法（EN）编辑描述符

语法：

EN*w.d* [**E***e*]

EN 和 E 编辑描述符基本类似，区别在于 EN 输出数据的非指数部分的绝对值强制在 1 到 1000 的范围内，且指数可以被 3 整除。

包括指数的区域的宽度是 w 个字符，小数点后 d 个字符，指数宽度 e 是可选的。指数的形式如表 5.4 所示。

表 5.4　　EN 编辑描述符的指数形式

编辑描述符	指数绝对值	指数的形式
ENw.d	|exp| <= 99	EN 后是加号或减号，然后是两位指数位，例如：EN8.3，EN-22
ENw.d	99 < |exp| <= 999	加号或减号，然后是三位指数位，例如：E8.3+102
ENw.d Ee	|exp| <= (10e)-1	EN 后是加号或减号，然后是 e 位指数位（可能前面以零填补），例如：EN8.3

下面是输出的具体例子：

```
      REAL x, y, z
      DATA x /-12345.678/, y /0.456789/, z /7.89123E+23/
      WRITE (*, 100) x, z
100   FORMAT (EN13.5,1X,EN13.5)
      WRTE (*, 200) y, z
200   FORMAT (EN13.2E4,1X,EN13.2E4)
```

输出结果如下：

```
-12.34568E+03 789.12300E+21
 456.79E-0003  789.12E+0021
```

3．科学计数法（ES）编辑描述符

语法：

ES*w.d* [**E***e*]

ES 和 E 编辑描述符基本类似，区别在于 EN 输出数据的非指数部分的绝对值强制在 1 到 1 的范围内。

包括指数的区域的宽度是 w 个字符，小数点后 d 个字符，指数宽度 e 是可选的。指数的形式和 EN 相同，参见表 5.4。

5.2.5　不可重复编辑描述符

不可重复编辑描述符可以改变解释重复编辑符的方式，还可以改变完成输入输出的方式。表 5.5 总结了不可重复编辑符。

表 5.5　　**不可重复编辑符**

形　式	名　称	用　途	是否可用于输入	是否可用于输出
'string' 或"string"	字符串编辑	传递 string 到输出单元	否	是
nH	Hollerith 编辑	传递下 n 个字符到输出单元	否	是
Q	*字符计数编辑*	*返回记录中剩余字符的数目*	*是*	*否*
Tc, TLc, TRc	位置编辑 (Tabs)	指定记录的位置	是	是
nX	位置编辑 editing (X)	指定记录的位置	是	是
SP, SS, S	可选加号编辑	控制加号的输出	否	是
/	斜杠编辑	指向下一个记录或写记录结束记号	是	是
\	*反斜杠编辑*	*延续相同的记录*	*否*	*是*
$	美元符号编辑	延续相同的记录	否	是
:	格式控制结束	如果 I/O 列表中没有其它记录则结束语句	否	是
kP	指数比例编辑	设置后面的 F 和 E（可重复）编辑符的指数比例	是	是
BN, BZ	空格解释	指定对数值区域空格的解释	是	否

用来分隔列表项的逗号在下列不可重复编辑符前面或后面时可以忽略：

（1） 在 P 编辑符和紧接着的 F、E、EN、ES、D 或 G 编辑符之间，例如：I3, 2PF8.6, 4F4.3

（2） 在撇号、双引号、反斜杠、美元符号或冒号编辑符前面或后面，例如：A14, A35, I5$

（3） 当可选的重复系数没有出现时，在斜杠编辑符之前；所有情况时在斜杠之后。

（4） 在 nH 或 X 编辑符之后，例如：2I3, 8HF5.3, A12

1．可变格式表达式

在任何需要整型常量作为编辑符的地方都可以在 **FORMAT** 语句中指定数值表达式。如果表达式不是整型的，则它将在使用之前被转化为整型。数值表达式必须用尖括号括起来。下面是合法的格式指定：

```
      WRITE(6,20) INT1
20    FORMAT(I<MAX(20,5)>)

      WRITE(6,FMT=30) REAL2(10), REAL3
30    FORMAT(<J+K>X, <2*M>F8.3)
```

数值表达式可以是任何 Visual FORTRAN 有效的表达式，包括函数调用和哑元引用。但应该注意：

（1） H 编辑符不能用于变量格式表达式

（2） 表达式不能包含形象操作符（例如不能包含大于号>或小于号<，但可以包含 .LT. 或 .GT.）。

变量格式表达式不能出现在像下面的赋值表达式语句中：

```
CHARACTER(80) S
S = '(I<J+K>)'
WRITE(6, S) N
```

但可以用在下面这样的语句中：

```
WRITE(6, '(I<J+K>)') N
```

在执行 **READ**, **WRITE** 或 **PRINT** 语句时，表达式的值都会被重新计算，例如：

```
      INTEGER width, value
      width=2
      READ (*,10) width, value
10    FORMAT(I1, I <width>)
      PRINT *, value
      END
```

当输入是 3123 时打印的将是 123 而不是 12。

2．格式指定和 I/O 列表之间的相互作用

如果 I/O 列表包含一个或多个数据项，则在格式指定时至少有一个可重复编辑符。空的编辑指定（）只能用在 I/O 列表没有数据项的情况。一条编辑指定为空的格式 **WRITE** 语句输出的是回车换行。一条编辑指定为空的格式 **READ** 语句将跳过相邻的下一个记录，除非输入输出设置成 **ADVANCE='NO'**，这时文件位置将保持不变。

记录中的字符如果少于编辑符指定的长度，在右侧会填以空格，除非在 **OPEN** 语句中指定 **PAD='NO'**（缺省时 **PAD='YES'**）。用户输入的空格的解释取决于空格编辑描述符（**BN** 或 **BZ**）的作用或 **OPEN** 语句中的 **BLANK=**选项。**BN** 和 **BZ** 的优先级比 **BLANK=**选项要高。

下面的例子是使用 **BZ** 编辑的 **READ** 语句（缺省 **PAD='YES'**）：

```
READ (*, '(BZ, I5)') n
```

如果输入：

```
5
```

输入记录的总共字符数是 2（5 和后面的空格）。这个记录右边会填补 3 个空格，但这些附加的空格会被忽略。于是输入记录被解释为 5 而不是 5000。反之，如果用户输先入一个空格，再输入 5 和一个空格，则用户输入的空格将被解释成 0，整个输入也会被解释成 5000。

在输入输出语句执行时，I/O 列表中的每一项都和一个可除非编辑符联系。I/O 列表中的复型数据需要两个编辑符。非重复编辑符不和 I/O 列表中的数据项联系，*而 Q 编辑符例外*。

在格式输入输出过程中，格式控制器从左向右扫描格式数据项。下面列出了格式控制器可能碰到的具体情况及相应的解释：

（1）　如果 I/O 列表中出现了可重复编辑符和相应的数据项，该数据项和编辑符是互相联系的，该数据项的输入输出会在编辑符的格式控制下执行。

（2）　如果 I/O 列表中出现了可重复编辑符而没有相应的数据项，格式控制器将中止输入输出。例如下面的语句：

```
      i = 5
      WRITE (*, 100) i
100   FORMAT ('I=', I5, ' J=', I5, ' K=', I5)
```

这时输出的形式为：

```
I=    5 J=
```

输出在 J=之后被中止，因为 I/O 列表中第二个 I5 之后没有相应的数据项。

（3） 如果出现冒号编辑符（中止格式控制）且 I/O 列表中没有其它项，则格式控制器将中止输入输出。

（4） 如果出现格式指定中最后一个配对的右括号且 I/O 列表中没有其它项，则格式控制器将中止输入输出。

（5） 如果出现格式指定中最后一个配对的右括号且 I/O 列表中还有其它项，文件将会定位到下一个记录的开始处。

（6） 如果没有前面那样的右括号，格式控制器重新从格式的开始处扫描。在格式重新扫描时必须有至少一个可重复编辑符。

（7） 如果重新扫描格式指定开始于重复嵌套的格式指定，重复系数说明了重复嵌套格式指定的重复次数。重新扫描不会改变原来设置的指数比例或起作用的 **BN** 或 **BZ** 空格编辑符。

5.2.6 直接列表 I/O

当用户使用格式 I/O 时，必须指定数据在外部设备上如何出现。而使用直接列表 I/O 可以从一个 I/O 列表中读写数据而不需要 **FORMAT** 语句。输入输出由 I/O 列表中的数据项的数目和类型决定。例如：

```
INTEGER(2)    INT1, INT2
REAL(8)       REAL1, REAL2
CHARACTER     CHAR(7)
READ (10, *) INT1, INT2, REAL1, REAL2, CHAR(7)
```

上例中 **READ** 语句需要两个 2 字节整数，两个 8 字节实数和七个字符。

直接列表 I/O 可以用于读写外部或内部文件。对于以规定格式提供且无误差的数据的情况，直接列表 I/O 输入就显得很有用。如果对数据的格式更感兴趣，直接列表 I/O 输出是最适合的。

注意：非格式 I/O（直接列表 I/O 和名称列表 I/O）不使用显示的格式或 **FORMAT** 语句，当用户执行非格式 I/O 时，Visual FORTRAN 清除对输出设备的内存而不是格式化 I/O 列表中的数据。如果用户持续使用非格式 I/O（并避免和格式 I/O 混合），程序会产生更少的运行错误，并且可执行文件也会更小一些。

如果用户使用直接格式列表对读写进行格式化，数据项必须符合它们映射的内部表示。执行直接列表输入时，必须提供符合数据流的 I/O 列表。执行直接列表输出时，用户提供 I/O 列表，格式由 Visual FORTRAN 提供。

1. 直接列表输入

执行直接列表输入时，必须提供包含要读变量名称的 I/O 列表。直接列表输入数据记

录是一系列由逗号或空格分开的值。在直接列表数据记录中的每个数据项必须要么是某个值要么是占位符。

占位符对其映射的变量没有影响：有值的变量保留它们，没有值的变量仍然保持为空。斜杠（/）会结束输入流。输入列表中的其它项就像它们值为空一样不会产生任何影响。例如：

```
INTEGER I1, I2, I3
I1 = 1
I2 = 2
READ (*, *) I1, I2, I3
```

如果用户输入：

```
8, , / 9
```

I1 被赋成 8；I2 的输入为空，所以它的值仍为 2；然后因为有斜杠所以输入记录被中止，这样 I3 的值仍为空（0）。

只有字符串常量可以包含嵌入的空格。两个数字之间的空格被解释为分隔符。与分隔符（逗号，斜杠或其它空格）相邻的空格被忽略。例如：

```
5,  6  /  7
```

等价于：5,6/7

重复输入的值可以通过给该值乘上要重复的次数来实现。例如，3*5 告知 Visual FORTRAN 重复读取值 5 三次。举例如下：

```
REAL R(10)
READ (5,*) R
```

如果输入记录中包含 10*3.1416，则数组 R 的十个元素都被置为 3.1416。

格式 I/O 中的大部分输入形式在直接列表格式中也是可用的。下面的规则适用于为直接列表输入的所有值：

（1）　输入值的形式必须是输入列表项类型可接受的。

（2）　空格始终被视为分隔符而不是 0。

（3）　嵌入空格只能在字符常量中出现。

记录结束标志的作用和空格一样（显然当空格在字符常量中出现时例外）。

此外，对于特定的值应遵守表 5.6 的规则。

表 5.6 直接列表输入

值的类型	约 束
单精度或双精度实型常量	一个实型或双精度常量必须是数值输入区（F 编辑符适用的区）。除非在区域内有一个小数点，一般假设没有分段的位
复型常量	一个复型常量是一对有顺序、中间由逗号隔开并由括号括起来的实型常量或整型常量。第一个常量为实部，第二个为虚部
逻辑型常量	逻辑常量不能在 L 编辑符允许的可选字符之间包含斜杠或逗号
字符常量	字符常量是由单引号包括的非空字符串。某个字符串中如果要出现的单引号则要用双引号来代替这个单引号 字符常量可以从一个记录的末尾继续到下一个记录的开头。上一记录的结束并不产生空格或其它会成为该常量一部分的字符。这个常量可以延续所需要的任意多个记录，并且还可以包含空格、逗号或斜杠等字符 如果列表项中的长度 n 小于或等于字符常量的长度 m，字符常量最左边的 n 个字符被传递到列表项，如果 n 大于 m，字符常量最左边的 n 个字符被传递到列表项，剩下的 n-m 个字符位置被空格填充。这种效果和在字符赋值语句中把常量赋给列表项的结果是一样的
派生类型	在输入列表中的一个派生类型数据等价于在输入列表中列出这个派生类型数据所有成员，其顺序和声明该派生类型时的顺序一致
Null（空）值	Null 说明一个数据项缺给定的变量。指定一个 Null 有三种方法： 1．在连续的分隔符之间没有字符。例如在包含 54.23,,141 的记录中 54.23 之后的两个值为空 2．每次执行直接列表输入语句读入的第一个记录中的第一个分隔符之前没有字符 3．还可以用星号格式说明一组 Null。例如 10*等价于 10 个空值或一个包含类似 16,,,,,,,,,,23.8 的数据流 空值对当前相应的输入列表项的定义没有影响。如果输入列表项已经被定义，它保持原来的值，如果输入列表项未定义则它仍保持未定义状态 直接列表输入语句中作为分隔符的斜杠在赋了以前的值以后会中止那条直接列表输入语句的执行。任何输入列表中的其它项都将认为是空值
空格	所有直接列表输入记录中的空格都认为是分隔符，以下情况除外： 1．空格嵌在字符常量 2．次执行直接列表输入语句读入的第一个记录中开始的个空格，除非紧接着斜杠或逗号的第一个分隔符之前没有字符

下面是使用直接列表输入和输出的例子。

```
real    a
INTEGER i
COMPLEX c
LOGICAL up, down
DATA a /2358.2E-8/, i /91585/, c /(705.60,819.60)/
DATA up /.TRUE./, down /.FALSE./
OPEN (UNIT = 9, FILE = 'listout', STATUS = 'NEW')
WRITE  (9, *) a, i
```

```
WRITE  (9, *) c, up, down
REWIND (9)
READ   (9, *) a, i
READ   (9, *) c, up, down
WRITE  (*, *) a, i
WRITE  (*, *) c, up, down
END
```

上面的程序输出为：

```
 2.3582001E-05      91585
(705.6000,819.6000) T F
```

2．直接列表输出

在直接列表输出中，用户提供 I/O 列表，Visual FORTRAN 提供格式。

如果需要，系统会创建新的记录，但是常量中既不能出现记录末尾也不能出现空格（字符常量除外）。为了在打印记录时提供走纸控制，每个输出记录自动以空格作为开头。表 5.7 说明了每种内部数据类型的缺省输出格式。

表 5.7　内部数据缺省输出格式

数据类型	输出格式
BYTE	I5
LOGICAL(1)	L2
LOGICAL(2)	L2
LOGICAL(4)	L2
LOGICAL(8)[1]	L2
INTEGER(1)	I5
INTEGER(2)	I7
INTEGER(4)	I12
INTEGER(8)[1]	I22
REAL(4)	1PG15.7E2
REAL(8) T_floating	1PG24.15E3
REAL(8) D_floating	1PG24.16E2
REAL(8) G_floating	1PG24.15E3
REAL(16)[2]	1PG43.33E4
COMPLEX(4)	'(',1PG14.7E2,',',1PG14.7E2,')'
COMPLEX(8) T_floating	'(',1PG23.15E3,',',1PG23.15E3,')'
COMPLEX(8) D_floating	'(',1PG23.16E2,',',1PG23.16E2,')'
COMPLEX(8) G_floating	'(',1PG23.15E3,',',1PG23.15E3,')'
CHARACTER	Aw[3]

注：

（1） 只适用于 Alpha 系统。

（2） VMS 和 U*X 系统。

（3） *w* 是字符表达式的长度。

在输出列表中的一个派生类型数据等价于在输出列表中列出这个派生类型数据所有成员，其顺序和声明该派生类型时的顺序一致。

下面是使用直接列表输出的例子。

```
INTEGER        i, j
REAL           a, b
LOGICAL        on, off
CHARACTER(20)  c
DATA i /123456/, j /500/, a /28.22/, b /.0015555/
DATA on /.TRUE./, off/.FALSE./
DATA c /'Here''s a string'/
WRITE (*, *)   i, j
WRITE (*, *)   a, b, on, off
WRITE (*, *)   c
END
```

上面程序的输出结果为：

```
 123456           500
 28.22000        1.555500E-03 T F
Here's a string
```

5.2.7 名称列表 I/O

名称列表 I/O 是功能强大的读入数据或向文件（终端）写入数据的方法。通过在名称列表组中确定一个或多个变量，用户可以用一个单独的输入输出语句读写它们所有的值。

名称列表组由 **NAMELIST** 语句创建，形式为：

NAMELIST /名称列表/变量列表

名称列表用来识别一组名称，变量列表是变量、派生类型或数组名称的列表。

1. 名称列表输入

名称列表扫描输入文件来获得组的名称。当它找到组名称后，语句会继续扫描寻找给组中的变量赋值的赋值语句。名称列表输入以连字符（&）或美元符号（$）作为开始符号，以斜杠、连字符或美元符号为结束符号。大写、小写或大小写混合的单词 END 可以在结束符号&或$后出现，但不能出现在斜杠之后。例如：

```
      INTEGER a, b
      NAMELIST /mynml/ a, b
...
!  The following are all valid namelist variable assignments:
      &mynml a = 1 /
      $mynml a = 1 $
      $mynml a = 1 $end
      &mynml a = 1 &
      &mynml a = 1 $END
      &mynml
      a = 1
      b = 2
      /
```

注释（以!开始）可以出现在名称列表输入的任何位置，注释延伸到源文件行的末尾。

2. 名称列表输出

名称列表输出语句先写名称列表组中的名称，接着是名称列表中每个变量的名称，然后是一个等号和变量当前的取值。名称列表输出由斜杠结束。名称列表中变量的值通过 **WRITE** 语句写入一个文件或屏幕，语句中出现的是名称列表组的名称而不是格式指定符。注意，这里不需要 I/O 列表也不允许出现 I/O 列表。名称列表输出格式如下：

WRITE (*, **[NML=]** *namelist*)

NML=是可选参数，只有使用其它关键字（例如 **END=**）时才需要。

第一个输出记录是连字符，紧接着是大写的名称列表组名称。随后的记录列出了该组中所有变量名和它们的取值。每个输出记录是都由一个如果打印该记录就会提供的走纸控制而产生的空格开始的。最后一个输出记录是斜杠。

变量的值采用它们在直接列表 I/O 中的输出格式。除非当输出文件打开时 **DELIM=** 'QUOTE' 或 'APOSTROPHE'，否则字符常量就不会被定界分隔，而且这个创建的文件也不能由名称列表的 **READ** 语句来读，因为这条语句需要字符串分隔符。

下面的例子先声明一些变量，然后放入名称列表中并初始化，最后用名称列表 I/O 写入屏幕：

```
INTEGER(1) int1
INTEGER    int2, int3, array(3)
LOGICAL(1) log1
LOGICAL log2, log3
REAL    real1
REAL(8) real2
COMPLEX z1, z2
CHARACTER(1)  char1
```

```
CHARACTER(10) char2

NAMELIST /example/ int1, int2, int3, log1, log2, log3,        &
&          real1, real2, z1, z2, char1, char2, array

int1     = 11
int2     = 12
int3     = 14
log1     = .TRUE.
log2     = .TRUE.
log3     = .TRUE.
real1    = 24.0
real2    = 28.0d0
z1       = (38.0,0.0)
z2       = (316.0d0,0.0d0)
char1    = 'A'
char2    = '0123456789'
array(1) = 41
array(2) = 42
array(3) = 43
WRITE (*, example)
```

上面例子的输出结果是：

```
&EXAMPLE
INT1  =   11,
INT2  =          12,
INT3  =          14,
LOG1  = T,
LOG2  = T,
LOG3  = T,
REAL1 =   24.00000     ,
REAL2 =   28.0000000000000     ,
Z1    = (38.00000,0.0000000E+00),
Z2    = (316.0000,0.0000000E+00),
CHAR1 = A,
CHAR2 = 0123456789,
ARRAY =          41,          42,          43
/
```

3．名称列表 READ

一条名称列表 **READ** 语句的操作几乎是一个 **WRITE** 操作的逆操作。READ 语句先从当前位置开始扫描文件（或者在终端上或者在磁盘上），直到找到一个紧接着名称列表组名称的连字符，或直到文件末尾（后面跟其它名称的连字符被忽略）。在组名称后面必须至少有一个空格或回车来把名称和后面的赋值对区分开。

一个赋值对由一个变量名、数组元素或子字符串和后面的等号以及一个或多个值和值分隔符组成。等号前后可以有任意个或没有空格。值分隔符可以是单独的逗号、一个或多个逗号或 tab。如果逗号前面没有值则认为是空值，相应的变量或数组元素原来的值不变。变量出现的顺序是任意的。同一个变量可以在多个赋值对中出现，变量最终值由最后一个赋值决定。名称列表中的所有变量不一定都要赋值，其中没有出现的或与空值联系的变量保持它们原来的值。在输入文件中的变量名如果不在名称列表组中会产生运行错误。

如果派生类型名称在输入列表中出现，则记录中的第一个值被送给派生类型所定义的第一个成员，第二个值送给第二个成员等等。输入数据类型必须和成员类型相同。单独的派生类型成员也可以像其它变量一样出现在输入列表中。

如果数组名没有限定的下标，输入记录的第一个值赋给数组的第一个元素，第二个值赋给第二个元素等等。数组赋值是按行顺序进行的。

用户所赋的值不能多于数组中元素的个数。例如，不能给一个大小为 100 的数组指定 101 个值。但没有必要给数组每个元素赋值。没有赋值的元素认为空，相应的元素值也不变。孤立的值也可以赋给带下标的数组元素。

重复赋值时可以在要赋的值前加上重复系数和星号。例如：7*'Hello'把'Hello'赋给数组或变量列表的下七个元素。如果星号后没有值，则认为是空值，相应的元素原来的值不变。看下面的例子：

```
matrix = 10, 50*25, 50*, -101
matrix(42) = 63
```

第一条语句给元素 0 赋成 10，元素 1 至元素 50 赋成 25，元素 51 至元素 100 为空，元素 101 赋成-101。第二条语句把元素 42 的值该为 63。

字符串必须用撇号或引号隔开。名称列表 **READ** 语句由斜杠中止，或到达文件末尾为止。如果 **READ** 语句到达了文件末尾会产生错误。除了希望提前结束读文件，否则不要用斜杠作为值之间的分隔符。

假设用户希望在上面所讲的名称列表输出中的程序中从名称列表组里读一些变量的新值，注意下面连接到单元四（其中包含列表指定和赋值语句）的文件：

```
&example
Z1 = (99.0,0.0)
INT1=99
array(1)=99
REAL1 = 99.0
```

```
CHAR1='Z'
CHAR2(4:9) = 'Inside'
LOG1=.FALSE.
/
```

这种情况下，下面的名称列表 **READ** 语句将给指定的变量赋成新值。

```
READ (UNIT = 4, example)
```

第二个 **WRITE** (*, example)语句会显示它们改变的值如下：

```
&example
INT1   =   99,
INT2   =          12,
INT3   =          14,
LOG1   = F,
LOG2   = T,
LOG3   = T,
REAL1  =   99.00000     ,
REAL2  =   28.0000000000000     ,
Z1     = (99.00000,0.0000000E+00),
Z2     = (316.0000,0.0000000E+00),
CHAR1  = Z,
CHAR10 = 012Inside9,
ARRAY  =          99,          42,          43
/
```

5.3 输入输出语句

输入输出语句决定了作用在数据上的 I/O 操作。

5.3.1 输入输出语句概览

数据传输语句有：**READ**, ***ACCEPT***, **WRITE**, **PRINT** (或 ***TYPE***)和 ***REWRITE***。

文件连接、查询和定位语句有：**BACKSPACE**, **CLOSE**, ***DELETE***, **ENDFILE**, **INQUIRE**, **OPEN**, **REWIND** 和 ***UNLOCK***。

表 5.8 给出了 Visual FORTRAN 输入输出语句的简要描述。另外，内在函数 EOF 可以用来判断在文件当前位置之后是否还有剩余数据。

表 5.8　输入输出语句

语句	功能
ACCEPT	输入数据，和格式连续 READ 语句类似
BACKSPACE	文件定位于上一个记录开始处
CLOSE	断开和一个单元的连接
DELETE	从相关文件中输出一个记录
ENDFILE	写文件结束记录
INQUIRE	返回单元或外部文件的属性
OPEN	使一个单元号和一个文件或设备相连接
PRINT (或 TYPE)	向星号单元输出数据
READ	输入数据
REWIND	重新定位于文件的开头
REWRITE	覆盖当前记录
UNLOCK	释放先前被锁定的相关或连续文件中的一个记录
WRITE	输出数据

5.3.2　I/O 语句说明符

I/O 语句可以被可选参数修改。例如下面的 **OPEN** 语句：

```
OPEN (UNIT= 4, FILE= 'BESSEL.DAT', POSITION= 'APPEND')
```

这个 **OPEN** 语句把文件 BESSEL.DAT 和单元号 4 联系起来，它还用了一个可选参数 **POSITION='APPEND'**来定位于文件的末尾。任何接下来的 **WRITE** 语句就都接着文件末尾向后写而不会覆盖已经存在的数据。

可选 I/O 参数可以用来指定文件结构、位置和访问、数据分隔符和空格解释、错误处理和其它众多 I/O 特性。使用 I/O 说明符能改变 I/O 操作以适应用户需要。表 5.9 列出了可用的 I/O 说明符以及它们可以取的值、功能和可以使用它们的 I/O 语句：

表 5.9　I/O 说明符

说明符	取值	描述	可用语句
ACCESS=*access*	'DIRECT','*APPEND*' 或 SEQUENTIAL'	指定文件访问方法	INQUIRE, OPEN
ACTION=*permission*	'READ'、'WRITE' 或 'READWRITE' (缺省值是 'READWRITE')	指定文件 I/O 模式	INQUIRE, OPEN
ADVANCE=*ad_switch*	'NO' 或 'YES'(缺省值是 'YES')	指定格式联系数据输入为前进式或非前进式	READ

表 5.9 续表

说明符	取值	描述	可用语句
ASSOCIATEVARIABLE =*var*	整型变量	指定一个将被更新的变量来反映文件中下一个连续记录的记录号	INQUIRE, OPEN
BINARY=*bin*	'NO' 或 'YES'	返回文件格式是否为二进制	INQUIRE
BLANK=*blank_control*	'NULL' 或 'ZERO'(缺省值是 'NULL')	指定数值区域中的空格是被忽略还是解释为 0	INQUIRE, OPEN
BLOCKSIZE=*blocksize*	正整型变量或表达式	指定或返回 I/O 使用的内部缓冲大小	INQUIRE, OPEN
BUFFERCOUNT=*bc*	数值表达式	指定和多缓冲 I/O 单元相联系的缓冲个数	OPEN
CARRIAGECONTROL =*control*	'FORTRAN', 'LIST',或 'NONE'	指定走纸控制处理	INQUIRE, OPEN
CONVERT=*form*	'LITTLE_ENDIAN', 'BIG_ENDIAN', 'FGX','CRAY', 'FDX', 'IBM','VAXD', 'VAXG', 或 'NATIVE' (缺省值是 'NATIVE')	指定非格式数据的数据格式	INQUIRE, OPEN
DEFAULTFILE=*var*	字符表达式	指定缺省文件路径名字符串	INQUIRE, OPEN
DELIM=*delimiter*	'APOSTROPHE', 'QUOTE' 或 'NONE' (缺省值是 'NONE')	指定直接列表或名称列表数据的分隔符	INQUIRE, OPEN
DIRECT=*dir*	'NO' 或 'YES'	返回文件是否为直接访问而连接	INQUIRE
DISPOSE =*dis* (或 DISP=*dis*)	'KEEP', 'SAVE', 'DELETE', 'PRINT', 'PRINT/DELETE', 'SUBMIT', 或 'SUBMIT/DELETE' (对于过期文件缺省值是 'DELETE', 对于其它文件缺省值为'KEEP'	指定文件在单元关闭之后的状态	OPEN, CLOSE
formatlist	字符变量或表达式	列出编辑符, 用在 FORMAT 语句和格式说明中来描述数据格式	FORMAT, PRINT, READ, WRITE
END=*endlabel*	1 至 99999 之间的整数	遇到文件结束时把控制传递给指定标号的语句	READ
EOR=*eorlabel*	1 至 99999 之间的整数	遇到文件结束时转到指定标号的语句	READ
ERR=*errlabel*	1 至 99999 之间的整数	出现 I/O 错误之后指定要执行的可执行语句的标号	所有语句, 但 PRINT
EXIST=*ex*	.TRUE. 或 .FALSE.	返回文件是否存在和可打开	INQUIRE
FILE=*file* (或 *NAME=name*)	字符变量或表达式, 名称的长度和格式由操作系统决定	指定文件名	INQUIRE, OPEN

表 5.9 续表

说明符	取值	描述	可用语句
[FMT=]*formatspec*	字符变量或表达式	指定使用格式数据的编辑列表	PRINT, READ, WRITE
FORM=*form*	'FORMATTED', 'UNFORMATTED', *或 'BINARY'*	指定文件格式	INQUIRE, OPEN
FORMATTED=*fmt*	'NO' 或 'YES'	返回一个文件是否为格式数据传输而被连接	INQUIRE
IOFOCUS=*iof*	.TRUE. 或 .FALSE.（缺省值是 .TRUE. 除非指定了单元'*'）	指定一个单元是否是 QuickWin 应用程序的活动窗口	INQUIRE, OPEN
iolist	任何类型的变量、字符表达式或 NAMELIST	指定要被输入输出的数据项	PRINT, READ, WRITE
IOSTAT=*iostat*	整型变量	指定一个变量，它的值显示是否出现 I/O 错误	除了 PRINT 的所有语句
MAXREC=*var*	数值表达式	指定直接访问文件可以传入或传出的最大记录数目	OPEN
MODE=*permission*	*'READ', 'WRITE' 或 'READWRITE' (缺省值是 'READWRITE')*	同 ACTION	INQUIRE, OPEN
NAMED=*var*	.TRUE. 或 .FALSE.	返回文件是否命名	INQUIRE
NEXTREC=nr	整型变量	返回文件下一个记录是否可以读写	INQUIRE
[NML=]*nmlspec*	名称列表名	指定要输入输出的名称列表组	PRINT, READ, WRITE
NUMBER=*num*	整型变量	返回和文件连接的单元号	INQUIRE
OPENED=*od*	.TRUE. 或 .FALSE.	返回文件是否连接	INQUIRE
ORGANIZATION=*org*	'SEQUENTIAL' 或 'RELATIVE' (缺省值是 'SEQUENTIAL')	指定文件内部组织	INQUIRE, OPEN
PAD=*pad_switch*	'YES' 或 'NO' (缺省值是 'YES')	当输入列表或格式要求的数据多于记录中的数据时是否填充空格，或是否输入记录要求包含说明数据	INQUIRE, OPEN
POSITION=*file_pos*	'ASIS', 'REWIND' 或 'APPEND' (缺省值是 'ASIS')	指定文件位置	INQUIRE, OPEN
READ=*rd*	'NO' 或 'YES'	返回文件是否可读	INQUIRE
READONLY		指定文件是否只能由 REAS 语句访问该连接	OPEN
READWRITE=*rdwr*	'NO' 或 'YES'	返回文件是否既可读又可写	INQUIRE
REC=*rec*	正整型变量或表达式	指定要读写的文件的第一个（或唯一的）记录	READ, WRITE
RECL=*length* (或 RECORDSIZE=*length*)	正整型变量或表达式	指定直接访问文件的记录长度，或连续文件的最大记录长度	INQUIRE, OPEN

表 5.9 续表

说明符	取值	描述	可用语句
RECORDTYPE=*typ*	*'FIXED'*, *'VARIABLE'*, *'SEGMENTED'*, *'STREAM'*、*'STREAM_LF'*, *或 'STREAM_CR'*	指定文件中的记录类型	INQUIRE, OPEN
SEQUENTIAL=*seq*	'NO' 或 'YES'	返回文件是否被连接到连续访问	INQUIRE
SHARE=*share*	'COMPAT'、'DENYNONE'、'DENYWR'、'DENYRD'、或 'DENYRW' (缺省值是 'DENYNONE')	控制其它过程如何能够同时访问在网络系统中的文件	INQUIRE, OPEN
SHARED		指定一个文件是为多个同时执行的程序共同访问的连接	OPEN
SIZE=*size*	整型变量	返回在非前进式 READ 语句中文件结束之前读出的字符个数	READ
STATUS=*status*	'OLD'、'NEW'、'UNKNOWN' 或 'SCRATCH' (缺省值是 'UNKNOWN')	指定文件在打开和/或关闭时的状态	CLOSE, OPEN
TITLE=*name*	字符表达式	指定 QuickWin 程序中子窗口的名称	OPEN
UNFORMATTED=*unf*	'NO' 或 'YES'	返回文件是否为非格式数据传输而连接	INQUIRE
[UNIT=]*unitspec*	整型变量或表达式	指定文件连接的单元号	All except PRINT
USEROPEN=*fname*	用户编写函数的名称	指定控制打开文件的外部函数	OPEN
WRITE=*rd*	'NO' 或 'YES'	返回文件是否可写	INQUIRE

第六章　使用项目进行工作

Visual FORTRAN 是以项目（Project，也可称为工程）为单位来组织开发工作的。本章将说明如何在 Visual FORTRAN 中使用项目进行工作。

6.1　运行第一个程序

熟悉和掌握 Microsoft Developer Studio 和 Visual FORTRAN 的最好的方法是亲自动手实践。本节将以 Visual FORTRAN 自带的一个例子来初步说明项目的使用。例子的项目名称为 CELSIUS。CELSIUS 项目的内容是输出一个简单的华氏（Fahrenheit）温度到摄氏（Celsius）温度的转换表。

6.1.1　打开一个存在的工程

打开一个存在的工程步骤如下：

（1）　在文件菜单（File）中选择打开工作空间（Open Workspace）。

（2）　出现打开工作空间对话框显示缺省项目文件夹，初次安装时项目文件夹在 MyProjects 下。

（3）　在 MyProjects 下的文件夹和文件列表中双击 Celsius 文件夹，这时列表中显示的是 Celsius 工作空间文件。

（4）　选择 Celsius 工作空间文件 Celsius.dsw。单击打开（Open）按钮。

（5）　这时 Developer Studio 在 FileView 选项卡中显示出 Celsius 项目。单击 Celsius 文件夹的加号（＋）展开它，其中只有一个文件，即 CELSIUS.FOR。双击 CELSIUS.FOR 后在右侧的文本编辑器中可以看到源程序：

```
!
! Program Celsius Table: Prints simple Fahrenheit-Celsius table
!
  program celsius_table
  implicit none
  real Fahrenheit, Celsius

  print *,'   Fahrenheit        Celsius'
  print *,'-------------------------'
  do Fahrenheit = 30, 220, 10
```

```
        Celsius = (5.0/9.0) * (Fahrenheit-32.0)
        print '(F13.0,F12.3)',Fahrenheit,Celsius
    end do
    end
```

6.1.2 建立和执行项目

建立（Build）和执行（Execute）上面项目的步骤如下：

（1） 如果需要，应该更新相关文件，方法是在建立（build）菜单中选择“更新所有相关文件”（Update All Dependencies），选择需要更新的相关文件的适当的配置并单击 OK。对于 CELSIUS.FOR 这样的项目没有相关文件，所以这一步并不需要。

（2） 在 Build 菜单中选择 Build Celsius.exe。

（3） 建立的过程和状态在屏幕底部输出（Output）窗口的 Build 选项卡中显示。

（4） 在 Build 菜单中选择 Execute Celsius.exe 来运行这个程序。

运行结果如下：

```
 Fahrenheit       Celsius
 ----------------------------------
         30.        -1.111
         40.         4.444
         50.        10.000
         60.        15.556
         70.        21.111
         80.        26.667
         90.        32.222
        100.        37.778
        110.        43.333
        120.        48.889
        130.        54.444
        140.        60.000
        150.        65.556
        160.        71.111
        170.        76.667
        180.        82.222
        190.        87.778
        200.        93.333
        210.        98.889
```

220.　　104.444

（1）　观察运行结果后在 File 菜单中选择 Close Workspace 关闭工作空间。

6.2　Visual FORTRAN 的项目

在上面的例子的基础上，本节讨论 Visual FORTRAN 项目中的一些具体内容。

6.2.1　项目中的文件

从第一节的例子可以看出 Visual FORTRAN 以及 Microsoft Developer Studio 是 以项目（Project）来组织开发工作的。项目包括应用程序需要的源文件和建立（build）项目的详细说明。项目包含在工作空间（workspace）里。一个工作空间可以包含多个项目。

用户需要为每一个二进制的可执行程序创建一个项目。例如，一个 FORTRAN 主程序和一个 FORTRAN 动态链接库（后缀为.DLL，即 dynamic-link library）将在同一个工作空间占据不同的项目。

当用户创建一个项目时，Microsoft Developer Studio 将创建下列文件：

- 项目工作空间文件
  文件扩展名为.DSW。它保存项目工作空间信息。
- 项目文件
  文件扩展名为.DSP。它用来建立一个独立的项目或子项目。
- 工作空间选项文件
  文件扩展名为.OPT。它包含 Visual FORTRAN 的环境设置，例如窗口大小和位置、插入点位置、项目断点状态、观察窗口的内容等等。

Visual FORTRAN 并不支持用文本编辑器对.DSW 和.DSP 文件直接改动。

创建项目的同时也确定了项目的子目录。Developer Studio 所创建的上述文件都保存在这个子目录中。

创建项目时 Developer Studio 指定了存放用户指定的各种配置的中间文件和最终输出文件的子目录。这些子目录允许用户在建立配置而不会覆盖文件名相同的中间和最终输出文件。项目设置（Project Settings）对话框的普通（General）选项卡允许用户按自己的需要修改这些子目录。

如果用户已经有现成的源代码，应该在建立项目之前把源代码组织到目录中去。

如果用户在程序中使用了模块，则不需要显式地把模块加入到项目，模块以相关文件的形式出现。Developer Studio 会在程序单元使用模块之前扫描并编译文件列表中的模块。Developer Studio 自动在加入的项目中递归扫描 USE 和 INCLUDE 语句指定的模块。Developer Studio 不仅扫描源文件（扩展名为.FOR, .F, .F90）而且扫描资源文件（扩展名为.RC），并把这些文件加入 Dependencies 文件夹。用户不能直接添加或删除这个文件夹中的文件。

6.2.2 Visual FORTRAN 项目的类型

定义 Visual FORTRAN 的项目时必须选择项目的类型。在 File 菜单中选择新建（New）文件出现如图 6.1 所示的项目对话框。

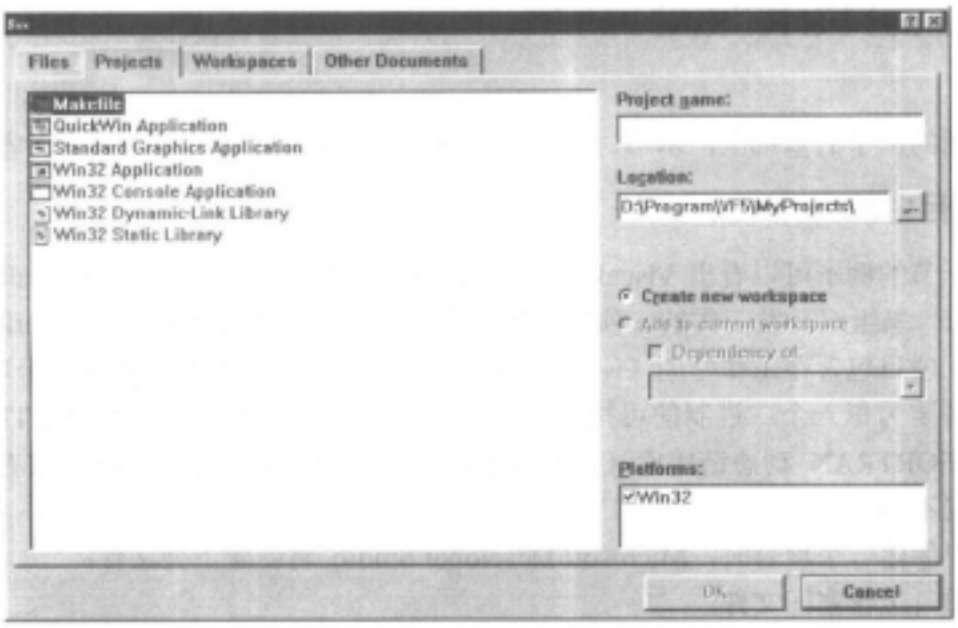


图 6.1 项目对话框

Visual FORTRAN 有以下几种项目类型：

- 控制台应用程序（Console application）项目

这种类型适用于基于字符而不需屏幕图形输出的应用程序。这些项目只在一个单独的窗口中操作，并允许在窗口中进行一般的读和写。

用户程序调用的图形子程序可能不会产生输出，但是可能返回错误代码。如果错误产生程序不会自动退出，所以用户编写的代码应该考虑到对这种情况的处理。

在控制台应用程序中可以使用静态库（.LIB），动态库（.DLL）和对话框，但不能使用 QuickWin 函数。用户可以选择多线程库和这种类型或其它所有类型的项目一起工作。

- 标准图形应用程序（Standard graphics application）项目

标准图形应用程序(扩展名为.EXE) 允许在一个单独的窗口中输出图形（如画直线和基本图形）或进行其它操作（例如清屏）。标准图形是 QuickWin 的一个子集，有时称为“Quickwin 单独窗口”。用户可以在这些项目中使用所有的 QuickWin 图形函数。 还可以和使用对话框。

选择标准图形类型时，Developer Studio 会自动包含 QuickWin 库，使用户可以使用图形函数。当以命令行进行建立（building）时必须指定/libs:qwins 选项。在建立标准图形项目时不能使用运行时间函数，也不能把标准图形类型制成.DLL。

标准图形类型的运行窗口即可以是全屏幕也可以是带有边界的，两种模式用ALT+ENTER 加以切换。

- QuickWin 应用程序（QuickWin application）项目

QuickWin 图形应用程序（扩展名为.EXE）比标准图形程序功能更强：项目在执行时可以打开多个窗口。例如，用户可能希望产生多个窗口并且能在一个窗口中控制这些窗口的切换。窗口的大小和位置都是任意的。

标准图形程序项目类似，选择 QuickWin 图形类型时，Developer Studio 也会自动包含 QuickWin 库，使用户可以使用图形函数。当以命令行进行建立（building）时也必须指定/libs:qwins 选项。不能把标准图形类型制成.DLL。

- Win32 应用程序（Win32 application）项目

Win32（Windows）应用程序（扩展名为.EXE）是选择 Win32 程序项目的主程序。这种项目可以完全访问 Win32 API，从而提供比 QuickWin 更完整（也有区别）的函数集。注意，Windows 项目种类要比 Visual FORTRAN 复杂得多。在试图使用全部 Windows 编程功能之前用户应该熟悉编写 C 语言应用程序和掌握 Windows 软件开发包（Software Development Kit ，SDK）。

- 静态库（文件扩展名为.LIB）项目

静态库是经过编译但独立存在于程序主体之外的一些代码，通常应该保持在它们的子目录下。静态库为组织大型程序和多个程序之间共享子程序提供了很多方便。这些库中只包含子程序而不包括主程序。如果静态库和程序相联系，当建立可执行文件时，需要用到的子函数将被从库中链接。

如果用户有大小很可观的库应该为它建立专门的目录。使用库的项目在编译时会访问到这个库。某个项目使用该库时，库中所选对象代码被链接成为项目的可执行代码来满足对外部函数的调用。不需要的对象文件不会被包括。

在命令行中编译静态库时要包含/c 选项来禁止链接。如果没有这个选项编译器将会因为库没有包含主程序而报错。如果要调试一个静态库必须用一个调用库函数的主程序，主程序和静态库都要被编译。

- Win32 动态链接库（文件扩展名为.DLL）

动态链接库是编译和链接到一个单元的源文件库，它独立于调用它的应用程序。DLL 和调用它的应用程序共用库的代码和数据地址空间。DLL 只包含子程序而不包含主程序。DLL 提供相当于静态库更有组织结构上的优势，但是更小的可执行文件的代价是稍许复杂的连接。DLL 中的对象代码并不包含在程序的可执行文件中，但当程序执行时将会以动态方式来联系。同一时刻可以有多个程序访问到 DLL。

表 6.1 对控制台应用程序、标准图形应用程序、QuickWin 应用程序和 Win32 应用程序工程类型进行了总结

控制台应用程序或标准图形应用程序可以转换成 QuickWin 应用程序。做法是新建一个 QuickWin 项目，进行同样的项目设置，加入文件到 QuickWin 项目即可。

表 6.1 几种项目类型的比较

工程类型	典型外观	编程复杂度	在线例子
控制应用程序	一个字符窗口	简单,类似字符应用程序	见 \MYPROJECTS\CELSIUS.
标准图形或 QuickWin 应用程序	应用例如菜单、图表、图标等图形的一个窗口（标准图形）或多个窗口	难度从简单到中等，依所使用的图形和用户交互而定	见 \DF\SAMPLES\GENERAL 下的子目录，例如 QWPIANO 和 QWPAINT.
Win32 (Windows) 应用程序	多个全面应用图形界面和 Win32 API 函数的窗口	复杂,需要专门的编程技术和 Win32 API	见\DF\SAMPLES\ADVANCED \WIN32,下的子目录，例如 PLATFORM 或 POLYDRAW.

6.2.3 Visual FORTRAN 项目的配置

每个项目的建立可以指定多个配置。配置确定诸如应用程序类型和建立时应用的工具设置等信息。每个项目在缺省状态下有调试（debug）和发行（release）配置。

- 调试配置

  缺省状态下调试配置设定项目选项来包括在调试配置下的调试信息。这时优化（optimizations）选项将被关闭。在调试应用程序之前必须为项目建立一个调试配置。

- 发行配置

  发行配置不包括调试信息，但会使用用户选择的优化选项。

指定配置可以在建立（Build）菜单中选择设置活动配置（Set Active Configuration）。例如对于第一节的例子，系统缺省时给出的是调试配置，现在可以重新打开 Celsius 项目，该为发行配置。如图 6.2 所示。

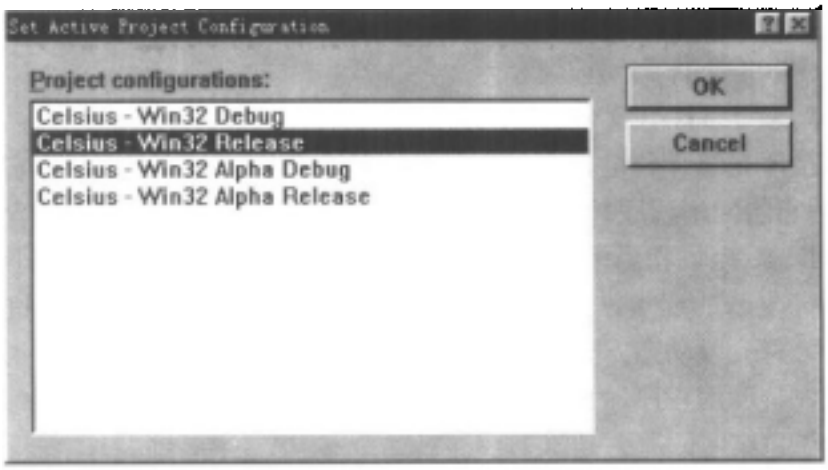


图 6.2 设置项目配置

现在 Celsius 文件夹下有 Debug 和 Release 两个文件夹，分别存放相应的可执行文件和其它相关文件。可以看出 Debug 下的 Celsius.exe 大小为 314KB，Release 下的 Celsius.exe

大小为 202KB。

6.2.4　创建 Visual FORTRAN 工作空间和项目

在 Developer Studio 中是以工作空间(workspace)和工程(Project)来组织文件和进行工作的。工作空间位于这个结构的最顶层，因此，通常在创建项目之前首先需要创建一个工作空间。创建工作空间通常有两种方法。

第一种方法是显式的创建一个空白的工作空间，然后再向工作空间中添加工程。这时，从 Developer Studio 的 File 菜单下选择 New...命令，这时出现如图 6.1 所示的对话框。选择 Workspace 选项卡，如图 6.3 所示。在 Workspace name 处键入工作空间的名字，这里假设为 MyWorkSpace，则 Developer Studio 将在 Location 所指定的目录下创建名为 WorkSpc 的子目录(当在 Workspace name 处键入完工作空间名后，可以在 Location 处修改这个默认设置)，然后以 WorkSpc.dsw 的文件名将该工作空间保存到这个目录下。

第二种创建工作空间的办法时直接创建一个工程。创建一个新的工程同样是选择 File 菜单下的 New...命令，出现如图 6.1 所示的的 Project 选项卡。然后在该对话框中选择 Create new workspace 单选钮(这是 Developer Studio 的默认选项)。注意在图 6.1 所示的对话框中，Project name 和 Location 的意义和图 6.3 中的 Worksapce name 和 Location 的意义类似。这样，在创建工程时，Developer Studio 将创建一个同名的工作空间。然后将所创建的工程添加到该工作空间中。

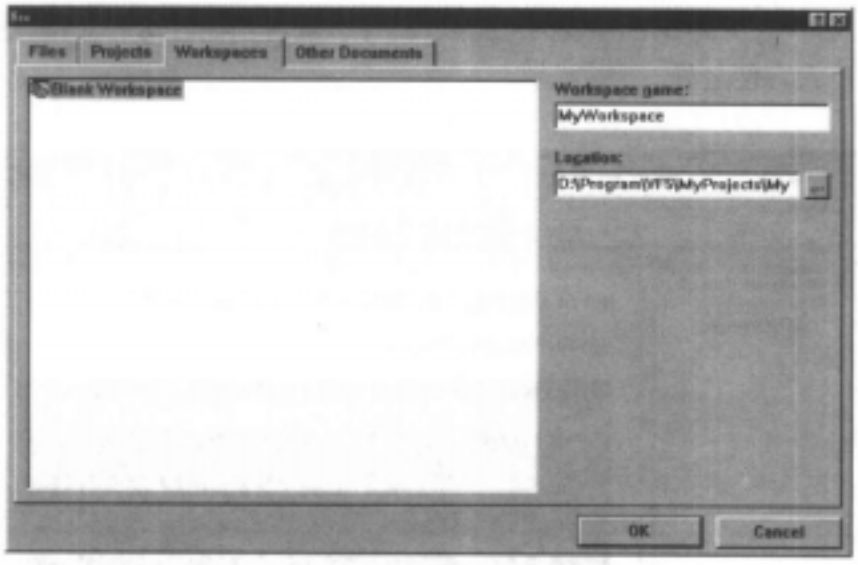


图 6.3　创建一个新的工程

如果仅在工作空间中包括一个工程的话，使用第二种方式显然是很合理的，并且，也要比使用第一种方式创建空白工作空间，然后再在空白工作空间中添加工程的方法要简洁和方便。在今后的很多情况下，用户都将使用第二种方式来创建工程和包括工程的工作空间。但并不是说第一种方式就没有用处了。事实上，在第二种方法中，Developer Studio 将工作空间和工程保存到 Location 所指定的同一个目录下，这对于单个工程的工作空间

是合理的。但如果用户想在工作空间中包括多于一个的工程的话，也许希望在保存工作空间的目录下新建子目录来保存这些工程，因为这样更有条理，更利于文件的管理。这时就需要使用第一种方式来创建空白工作空间，然后再在这个工作空间中新建和添加工程。

在工作空间中新建工程的方法和上面的第二种方式几乎一样。只不过这时应该在图 6.3 所示的对话框中选择 Add to current workspace (在图 6.3 中，这个单选钮是灰的，这是因为当前并没有打开的工作空间的缘故)。要注意这时 Location 处的目录名是基于当前工作空间所在的目录的。单击 OK 后，Developer Studio 根据在 Project name 处所键入的工程名以.dsp 的扩展名来保存该工程文件。

除了向工程中添加由 Developer Studio 和相应的开发包(如 Visual FORTRAN、Visual C++、Visual J++和 Visual InterDev)管理的文件外，还可以添加其它类型的文档，这些文档包括分成两类，一类由 ActiveX 部件创建和维护，另一类由其它的软件创建和维护。Developer Studio 在编辑这些文档时的行为是不同的。

对于由 ActiveX 部件(最常见的 ActiveX 部件有 Microsoft Word 和 Excel 等，但是，这里所指的 ActiveX 部件并不限于 Microsoft 的产品，其它任何符合 ActiveX 部件标准的第三方开发的应用程序都是 ActiveX 部件)创建的文档，用户可以在 Developer Studio 窗口内部打开并编辑它们，这时，由该部件提供的菜单项融合进了 Developer Studio 原有的菜单项，由该部件所提供的工具条取代了 Developer Studio 原有的工具条。并且，所打开的文档显示于原有的 InfoViewer Topic 窗口所在位置(如图 6.4 所示，在这幅图中，向工程 Hello 中添加了一个新建的 Microsoft Word 文档 Readme，并在 Developer Studio 内部打开并编辑该文档)。这样，用户无需离开 Developer Studio 就可以查看和修改这些文档，这就是 ActiveX 技术所带来的巨大方便之处。

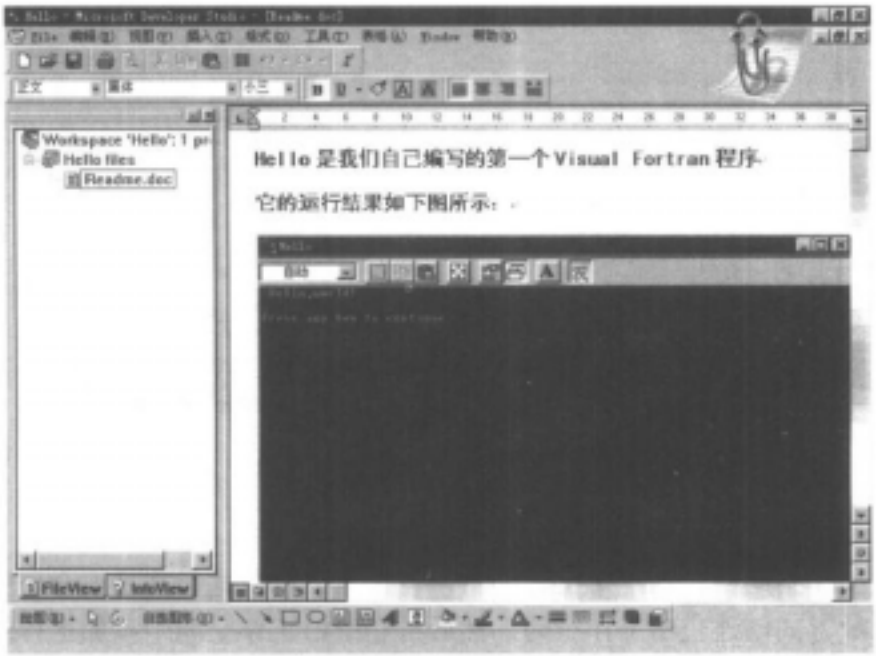


图 6.4 向工程中添加并编辑 Word 文档

向工程中新建这类文档只需在图 6.1 和图 6.3 所示的对话框中选择 Other Document 选项卡，然后指定新建的文档的类型，并给出文档文件名即可(对于向工程中添加的文档，必段指定文件名，如果只是在 Developer Studio 中编辑该文档，则不受此限)。如果是工程中添加已有的文档，则必须保证这些文件的扩展名与文档的类型相符合，因为 Developer Studio 是根据相应的文件扩展名来判断文档的类型和寻找创建和维护该文档的 ActiveX 部件的。如果添加由其它非 ActiveX 部件的软件创建和维护的文档，也必须遵从这个约定。对于非 ActiveX 部件的软件创建和维护的文档，在 Developer Studio 选择打开时，Developer Studio 将在另一个单独的窗口中打开该文档以供用户进行编辑。

6.3　编写程序的一般步骤

本节将以一个 Hello world 程序来说明在 Visual FORTRAN 中编写程序的一般步骤。

6.3.1　新建一个工程

在正式编写代码之前必须新建一个工程，按上一节第二种方法步骤如下：

（1）　在 File 菜单中选择 New。

（2）　出现新建对话框，其中有文件（Files），项目（Projects），工作空间（Workspace），其它文档（Other Documents）四个选项卡。

（3）　项目（Projects）选项卡中列出了第一节说明的各种项目类型。选定项目名称和路径。项目的名称为 Hello。单击 OK。

6.3.2　向项目中添加文件

在项目（Project）菜单中选择添加到项目（Add To Project）可以向项目中添加文件。添加有两种情况：

- 添加一个已存在的文件

（1）　在 Project 菜单中选择文件（Files）子菜单，如图 6.5 所示。

（2）　出现插入文件到项目（Insert Files into Project）对话框，使用这个对话框选择要加入工程的文件。如果要加入多个文件可在单击所选文件时按住 CTRL 键。

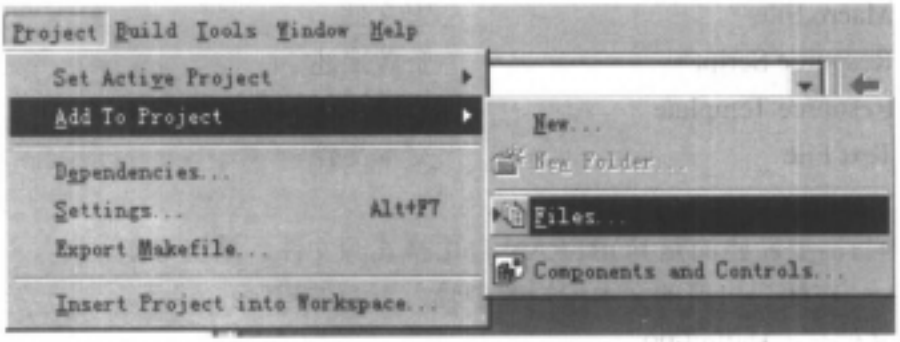


图 6.5　向项目添加文件

- 添加一个新文件

如果要添加的文件还不存在则需要向项目中添加新文件。

（1） 在 Project 菜单中选择新建（New）子菜单。

（2） 出现新建文件对话框，如图 6.6 所示。确定文件名和路径。对于本项目，文件名为 Hello。

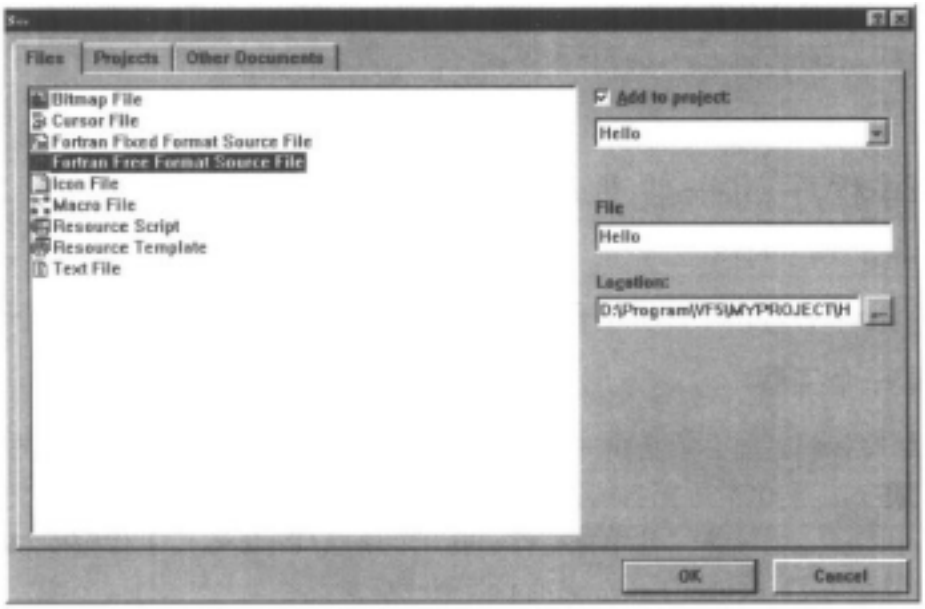


图 6.6 新建文件对话框

（3） 在图 6.4 所列出的文件类型中选择新建的类型。文件类型如下：

Bitmap File	位图文件
Cursor File	（鼠标）指针文件
FORTRAN Fixed Format Source File	FORTRAN 固定格式源文件
FORTRAN Free Format Source File	FORTRAN 自由格式源文件
Icon File	图标文件
Macro File	宏文件
Resource Script	资源描述
Resource Template	资源模板
Text File	文本文件

本项目需要新建的是 FORTRAN 自由格式源文件。单击 OK。

（4） 在随后出现的文本编辑器中输入下面的程序：

文件名：Hello.F90

```
! File: HELLO.F90
```

```
PROGRAM HELLO_WORLD

    PRINT *, 'Hello,world!'
    PRINT *, ' '

END PROGRAM HELLO_WORLD
```

（5）　按第一节的步骤编译项目。

程序的运行结果是在屏幕输出：

```
Hello,world!
```

第七章 使用编辑器提高效率

编辑器是开发过程中必须要用到的工具，选择合适的编辑器并熟练掌握相应的技巧可以使开发工作事半功倍。

7.1 前言

程序员在计算机前的绝大部分时间是用来对程序进行修改。就在二三十年前人们还不得不在纸带上穿孔来向计算机输入程序（这也造成了 FORTRAN 90 以前近乎刻板的书写格式），那时根本没有“编辑”可言。编辑器（哪怕是行编辑器）的出现使程序员摆脱了这一毫无意义的繁重劳动。但是还要在编辑和编译、链接、调试工作之间不停切换。Borland 公司的 Turbo Pascal 是一个划时代的产品，它开创了集程序编辑、编译、调试于一体的集成开发环境（IDE）的先河。

计算机软件发展到今天，在 Visual FORTRAN 之外有很多优秀的编辑器，有些读者可能熟悉并习惯使用这些编辑器。但是其中没有一种会比为编译和调试系统量身定做的编辑器更加适合于开发工作了。所以我们强烈推荐读者使用 Visual FORTRAN 自己集成的编辑器。

另一方面，Visual FORTRAN 集成的编辑器有拥有许多编辑器的共性，上手并不困难。但是要熟练使用它们仍需要系统深入的研究和经常的练习。在使用过程会发现，熟练掌握 Visual FORTRAN 的编辑器会大大提高开发效率。

正是基于以上考虑本章将详细介绍 Visual FORTRAN 中常用两种编辑器：文本编辑器和图形编辑器，重点放在文本编辑器。

7.2 文本编辑器

Visual FORTRAN 提供了一个专门为“代码剪切”而设计的文本编辑器。这个编辑器与各种环境工具（如调试器）组成一个完整的统一体，并提供了大量复杂的特征，包括撤销/重做（Undo/Redo）、可定制的快捷键命令和对 Win32 的快速访问。本节将介绍 Visual FORTRAN 文本编辑器最重要的几个方面，讲述一些有用的和隐含的特性，并说明怎样有效地使用编辑器。如果只想修改编译器错误，那么在编辑文本时，将文本保持在 Developer Studio 环境中是很方便的。由于 Visual FORTRAN 是一个基于 Windows 的软件，所以其文本文件以 ANSI 格式存储。

7.2.1　启动文本编辑器

在 Visual FORTRAN 环境中只需给出文件类型， Visual FORTRAN 将会根据文件类型自动启动相应的编辑器。进入编辑器后，如果想创建一个新文件，可以单击文件（File）菜单，再选择新建（New）。

在列表中选定其类型，如 FORTRAN Fixed Format Source File、FORTRAN Free Format Source File、Macro File 或 Text File。再单击 OK（确定）就会发现，文本编辑器以一个空文档窗口的形式出现在屏幕上。此时菜单和工具栏都基本不变。为了保持工作的连续性，Visual FORTRAN 采用通用的外观和动作，即不同的编辑器窗口和菜单命令是类似的。文档的类型由在菜单栏的左边缘的小页标（见图 7.2）或在主窗口顶部标题栏中文档名称来指出。如果在 New（新建）对话框中没有给出文档名称，编辑器将自动给此新文档赋一个临时名称，例如 Text1.选择保存（Save）可以另给文档新名称。Visual FORTRAN 的菜单在各种编辑器中是通用的。虽然表面的菜单和工具栏基本不变，但在内部会有一些变化。例如，在使用文本编辑器时，许多在主窗口中呈灰色的命令被激活，而变为正常体。对不同的编辑器，Visual FORTRAN 会自动激活相应的命令。例如，由于文本只对文本编辑器有效，因此，Find（查找）命令在使用文本编辑器时被激活而呈正常体，而在使用图形编辑器时，则因未被激活而呈灰色。使用文本编辑器时可用的菜单命令如图 7.1 所示。

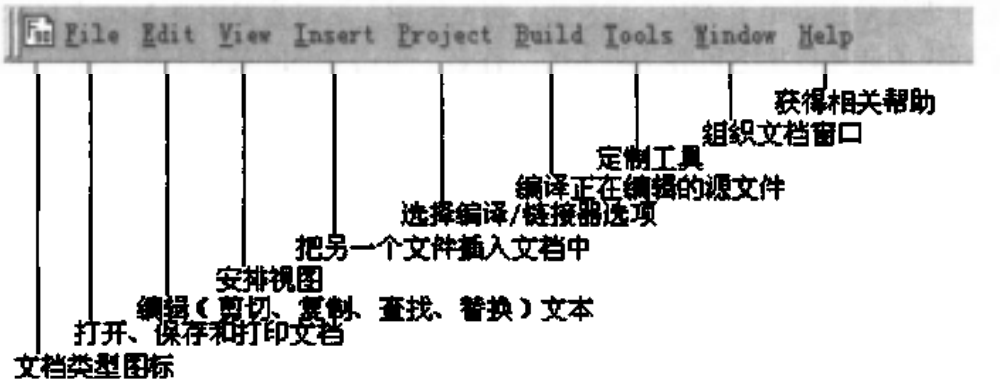


图 7.1　文本编辑菜单

除了上图中所列的命令外，还有一些可用的命令。例如，如果在编辑器中删除了文本，却又想恢复它， 可使用 Edit（编辑）菜单下的 Undo（撤销）和 Redo（重做）命令，Undo（撤销）能记忆最近一次删除操作。如果要恢复更早删除的文本，连续单击 Undo（撤销），或重复按 Ctrl+Z 直到恢复到需要的文本为止。Redo（重做）命令来取消最近一次 Undo（撤销）操作。

7.2.2　文档

本节说明怎样创建、打开、浏览、存储和打印文本文档。

1. 打开文档

Visual FORTRAN 文本编辑器支持多文档操作，所以可以同时打开多个文档。再使用 New（新建）命令，就可以创建另一个新文档，系统自动默认文档名为 Text2。如果是满屏显示，则正在使用的文档名称显示在屏幕顶端标题栏中。可以通过按 Ctrl+F6 在已打开的文档间进行切换，也可通过从 Window（窗口）菜单中挑选所需的文件的名称来实现切换。同一个文档只能创建一次。当将一个新文档存到盘上时成为一个文件，想再次对它进行处理时，就必须打开它，而不能再创建它。用下述办法可打开一个已存在的文件：

（1）单击标准工具栏中的 Open 按钮。

（2）按 Ctrl+O。

（3）选择 File 菜单中的 Open 命令。

（4）选择 File 菜单中的最新文件（Recent File）命令,再选择所需文件的名称。

前三种办法都会出现 Open 对话框。在对话框中可以通过浏览各个文件夹来寻找想要打开的文件。最后一种方法不需要对话框，系统自动列出最近编辑过的文件列表（称做 MRU 列表）。从这个列表中可以直接打开所需要的文件。MRU 列表在默认情况包含最近编辑过的四个文件，这四个文件并不只是文本编辑器编辑过的，也包含其他任意一种编辑器编辑过的文件。拉下 File 菜单，将鼠标光标放在 Recent File 命令行上就可显示 MRU 列表。在表中挑选文件名，就可在相应的编辑器中打开文件。当同时对几个文件进行操作时，使用 MRU 列表是很方便的。在 Windows 时代，即使对于一个很小的软件项目也需同时对很多文件进行操作以至于 MRU 列表都显得不够用。这时需要用到 Visual FORTRAN 扩展 MRU 列表的选项以使它可容纳更多的文件名：单击 Tools 菜单下的选项（Options）命令，选择工作空间（Workspace）选项卡。在标有 Recent File List Contains（最新文件列表包含）的文本框中输入需要的值即可，如图 7.2 所示。

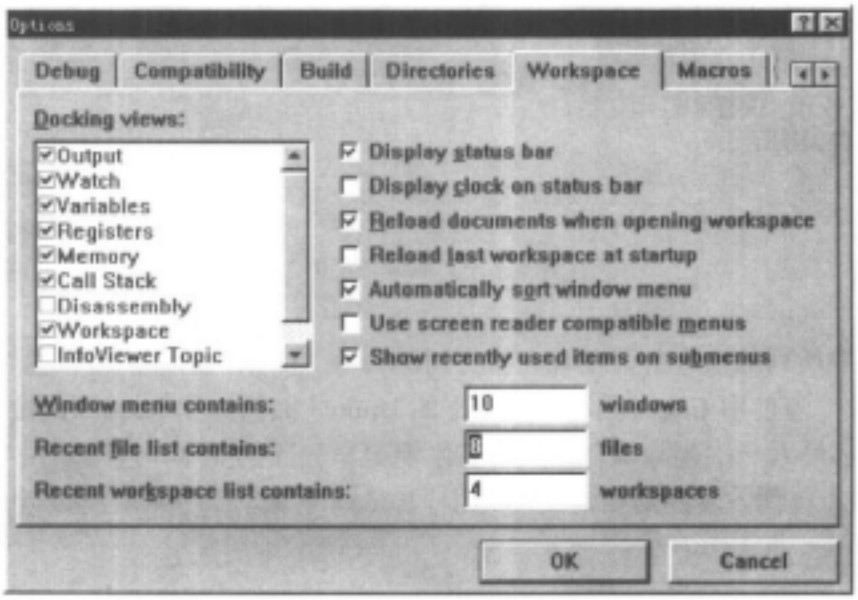


图 7.2 改变 MRU 列表文件名数目

Workspace 选项还提供了另一种影响 MRU 列表外观的选项：如果希望 MRU 列表直接出现在 File（文件）菜单中，而不是隐藏在下一级命令里，只需取消 ShowRecently Used Items On Submenus（在子菜单上显示最近使用的项）选项即可。不过，由于文件名的增加，File（文件）菜单会显得很拥挤。

如果所需的文件不在 MRU 列表中，那么，必须使用 Open（打开）命令，再在 Open（打开）对话框确定文件。这个对话框中的默认路径是当前软件所在的文件夹。在选择要打开的文件时，按下 Ctrl 键再单击文件名，就可以同时打开一组文件，每单击一次就多打开一个文件。如果想取消某个已打开的文件，只需再次按下 Ctrl，同时单击该文件名即可。选定文件后，单击 Open（打开），就可将所有选中的文件分别打开成不同的文档。如果想打开的文件在其目录下是连续排列的，有一种更快的方法来选中它们：可先单击选中第一个文件，再按住 Shift 单击最后一个文件，这样，在这两个文件间的所有文件都被选中了。如果想不要其中某个文件，只需按住 Ctrl 的同时单击该文件即可。

值得注意的是：在 Open（打开）对话框中列出的文件并不是该文件夹下的所有文件，这是由于文件类型（Files Of Type）组合框中有过滤设置。“过滤”是指只选择一组相关的文件。例如，系统默认的 FORTRAN 文件过滤器将使 Open（打开）对话框中只包含以 FOR，F90，RC 等为扩展名的文件。如要列出其他类型的文件，如以 DOC 或 TXT 为扩展名的文件，则须在文件类型框中选择相应的过滤器。

在 Open 对话框中有一个作为只读文件打开（Open As Read-Only）选项。这是一个很重要的选项，一旦它被选中，将不能对打开的文件作任何改变，而只能对其浏览、打印，或将其复制到剪贴板上。通常，只需使用 File（文件）菜单下的 Save as（另存为）命令，将此只读文档以一个新名称保存，就可以解除只读限制。注意，此时原文件只读属性并未改变，只是新文件中没有只读限制。Tools 菜单下 Options 对话框中 Compatibility（兼容性）选项卡下，有一个阻止编辑只读文件（Protect Read-Only Files From Editing）选项，当它被选中后，只读文件将不能以新名称另存。

2. 查看文档

尽管文本编辑器尽可能有效地使用屏幕空间，但当有好几个窗口同时显示时，空间常常显得很紧张。为了使可用空间尽可能大，选择 View（查看）菜单下的全屏（FullScreen）命令，或者按 Ctrl+V，就可使标题栏、菜单、工具栏都消失，以便最大限度地腾出空间。如果要返回正常显示，只需按 Esc，或单击浮动的 Full Screen 工具栏即可。在满屏状态下，由于菜单栏消失，不能再用鼠标来单击下拉菜单。如果要使用下拉菜单，可先按 Alt，再按所需菜单的首字母即可。例如，先按 Alt，再按 F，就可激活 File（文件）菜单，再按左、右箭头键，可激活相邻的菜单（在这里，按 Alt 起激活菜单栏的作用，因此两个键不必同时按下）。

单击 Full Screen 工具栏上的关闭按钮可以移走它。之后将只能通过按 Esc 键来返回正常显示。如要恢复该工具栏，按 Alt+T 显示 Tools（工具）菜单，再单击 Custmize（自定义）命令，在 Custmize（自定义）对话框内的 Toolbar（工具栏）选项卡下选中工具栏列表中的 Full Screen。以同样的方法，也可以在满屏状态下显示其他工具栏或菜单栏。

文本编辑器的文档窗口允许多窗格显示，可以将同一文档的一个、两个或四个不同部分同时显示在窗口上。图 7.3 就是同一文档的四窗格显示的例子。

图 7.3 典型的分隔窗口

当创建或打开一个文档时，多窗格功能自动生效。可以通过选择 Window 菜单下的分隔（Split）命令来一次设定两个窗格分隔条的位置。Split 命令设定了两个窗格分隔条所决定的分割线在编辑窗口中交点的位置，移动鼠标，将此交点移到所需位置上，再单击，即可将两个窗格分隔条同时设定。

由于各窗格没有自己的滚动条，因此，最有效的方法是将显示窗口划分为一上一下两个窗格。要想做此划分，只需要将垂直分隔条（即位于水平滚动条上的分隔条）移到最左端或最右端，直到其消失为止。通过单击窗格或按 F6，可从一个窗格切换到另一窗格。上面的两窗格显示对于上下划分的文档非常方便，但由于各窗格不能独立滚动，因此对于左右划分的文档就不大有效了。但是，Window 菜单提供了另一个命令，它可以垂直视图的方式显示将一个文档分成两个或多个窗格。在文本编辑器只打开一个文档的情况下，单击新窗口（New Window）命令，就会打开另一个包含同一文档的显示窗口。这并不是重新打开文件，而不过是在编辑器的工作空间中提供了原文档的第二个视图。这两个视图各有其滚动条及闪烁光标，所以可以同时浏览文档的不同部分。此时，再单击 Window 菜单下的 Tile Vertically（垂直平铺）命令，就可使两个浏览器实现左右显示。通过单击某个窗口，或按 F6 键，可实现视图之间的切换。

也可以通过 New Window 命令再增加视图。不过，尽管各视图之间相互独立，但它们都只能显示同一文档的内容，因此，在一个视图中对文档做的任何修改，都会立即显示在其他视图中。

3. 保存文档

当开始在一个文档窗口中输入文本时，一个“*”符号会显示在标题栏和 Window（窗口）菜单下已打开文档表中的文档名后。这表明此时该文档已被改变，它和原文件已不相同。与文字处理器不同的是，Visual FORTRAN 文本编辑器不能自动存盘.因此在输入源代码时，一定要养成定时存盘的习惯。存盘可用以下方法：

（1）单击标准工具栏上的 Save 按钮。

（2）按 Ctrl+S

（3）选择 File 菜单下的 Save 命令。

保存文件时，文件名后的“*”符号将消失。当再次改变文本时，它又将出现。当“*”符号存在时，编辑器会提示先存盘。定时存盘主要是针对长时间编辑一个文档的场合。如果只是对源文件作一些小改动，再重新编译它，那么存盘就不那么重要了，因为在重新编译之前，系统会自动保存文档。第一次保存一个未命名的文档时，会出现另存为（Save as）对话框，在这个对话框里，可给出文件名及其扩展名。

当保存一个新文档，并给定其文件名时，这个文件名就代替系统默认的文件名，并显示在标题栏和 Window 菜单下。从此，当再保存此文件时，系统将自动覆盖前一个版本，并不再显示 Save as 对话框。而且，系统也不会以任何形式保存前一个版本。因此，一旦保存了一个文档，则它从前的版本将不复存在。如果源文件需要多个版本，则必须通过 File（文件）菜单下的 Save as（另存为）命令来给不同版本赋予不同名称。

4. 打印文档

要打印一个正在编辑的文档，只需单击 File 菜单下的打印（Print）命令，或按 Ctrl+P，来打开 Print 对话框。如果只想打印该文档中的一部分，先选择要打印的部分，再启用 Print 对话框中的选择（Selection）单选按钮。

5. 浏览文档

一般来说，通过按上下左、右箭头键或拉动滚动条，可以浏览文档。但是通过一些其他办法可更方便、更有效地浏览文档。

编辑文本时闪烁光标称为提示符。鼠标位置的箭头（或其他标志）称为光标。在文本编辑器中移动提示符的快捷键在表 7.1 中。

表 7.1　　文本编辑器提示符移动键

按　键	提示符移动
● 左箭头，右箭头	向前或向后移动一个字符。如果提示符在最左端，则左移将移到前一行的末尾；如果提示符在右端，则右移将移到下一行的开头
上箭头，下箭头	向上或向下一行。如果目标行比当前行短，则移动后提示符的位置取决于提示符现在的具体位置
Ctrl+左箭头，Ctrl+右箭头	向前或向后移动一个单词（Visual FORTRAN 编辑器将很多标点符号按独立单词对待）
Home，End	移到一行的开头或末尾
Ctrl+Home，Ctrl+End	移到文档的开头或末尾

在浏览较大的文档时利用书签（Bookmark）可以显著节省时间。文本编辑器的书签可以记忆一个位置，这样，无论在文档的任何位置，都可以迅速返回书签所在位置。书签会自动保留在所在行，并随文档的增减而改变位置。Visual FORTRAN 的文本编辑器提供了两种类型的书签：有名书签和无名书签。

（1）有名书签

一个命名的书签将会长久留在文档中，直到删掉它为止。它在文本中标志了一个精确的位置。实际上，当一个有书签的文档没有打开时，也可以从另一个已打开的文档中直接

跳到该文档中。这里，编辑器自动打开该文档，并将提示符置于书签所在的位置。

要创建一个有名书签，先将提示符置于想标志的位置，再单击编辑（Edit）菜单下的 Bookmarks 命令，此时就会看到 Bookmark 对话框。输入给此书签所起的名字，再单击添加（Add）按钮，就可将此书签加入到列表中。当关闭对话框后，此新书签就开始起作用了。

有两种方式可返回书签处：直接方式和间接方式。如果文档中的书签不很多时，用间接方式最方便。只需按 F2 就可跳到下一个书签处，按 Shift+F2 可跳到前一个书签处，或单击 Edit 工具栏中的下一个书签（Next Bookmark）按钮，如图 7.4 所示。

图 7.4 Edit 工具栏的书签按钮

通过直接方式移动提示符到书签处，需要再次访问 Boodmark 对话框。双击书签名，或选择目标书签，再单击转到（GoTo）按钮即可。也可以通过 Edit 菜单下的 GoTo 命令来跳到某个有名书签处。具体来说，有名书签是一个 32 位量，用以记录文本中一个特定位置。当在书签前的文本中添加或删除一个字节时，这个量会相应增加或减小。这样，书签就会继续指向其目标，文本怎样变更都不会影响它。与字处理器不同的是，关闭文档后，Visual FORTRAN 编辑器不将有名书签保留在文档中，因为额外字符会使编辑器产生混乱。

（2）无名书签

通常用不到有名书签的全部功能。当想定义这样一个书签时，它标志源代码的一处地方，在编辑文档的其他部分时，只想返回这个地方一两次，然后它就再也用不到了。在这种情况下，有名书签就显得不是很方便。这时可使用一个无名书签。无名书签是暂时的，当取消它或关闭文档后，无名书签就随之消失。而且无名书签只标志一行，而不是一个精确位置。当跳到一个无名书签处时，提示符自动置于该行开头。如果删除被标志的行，则书签也将被删除。无名书签的好处是，它易于设置，删除更简便。要使用一个无名书签，只需将提示符置于该行，按 Ctrl+F2，或单击如图 7.5 所示的标有旗帜的按钮即可。

图 7.5 设置无名书签按钮

如果页边选择被激活，那么，一个小盒子图标就会显示在被标志行左边的页边处。否则，编辑器将以不同颜色来显示整个被标志行。可通过单击工具栏按钮，或按 F2/Shift+F2，来移动提示符到无名书签处。每按一次，插入符将向前或向后移到下一个书签处，这时并

不区分该书签是有名书签还是无名书签。

有以下几种方法可删除无名书签：

① 将插入符移到此行，再按 Ctrl+F2 即可删除此书签。

② 按 Shift+Ctrl+F2。这将删除所有无名书签。

③ 文档关闭，无名书签将自动消失。

7.2.3　文本搜索

Visual FORTRAN 编辑器提供了三种文本搜索类型：

（1）在已打开的文档中搜索文本。

（2）在已打开的文档中替换文本。

（3）在磁盘文件中搜索文本。

前两种类型在文本编辑器中是很普通的，而在磁盘文件中搜索文本，则是一个不常见但却极有用的类型，它可显示出包含一个特定词或短语的所有文件。以下将详细说明这三个类型。

1．在打开的文档中搜索文本

像大多数文本编辑器一样，Visual FORTRAN 编辑器能够扫描文档，并找出给定单词或短语（称之为目标字符串）的位置。有两种方法可设定目标字符串的内容。最方便的方法是使用标准工具栏中的下拉框，如图 7.6 所示。

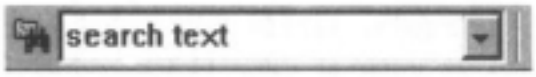


图 7.6　查找文本下拉框

在下拉框中输入字符串，或者单击框旁的箭头按钮，再从列表中选择以前输入的字符串。按 Enter 键开始搜索。当编辑器找到该字符串后，它将其高亮显示，并将提示符置于此高亮字符串的首字符旁。只要组合框允许，按 Enter 键就可继续按下一个已给出的目标字符串搜索。按 Esc 键，或单击文档窗口的任何部分，就可返回编辑模式。再按 F3 或 Shift+F3，就可继续向后或向前继续搜索。Visual FORTRAN 提供了这些命令的工具按钮，不过需要自己将它们加到工具栏中。它们被加到工具栏中后如图 7.7 所示。

图 7.7　添加按钮后的 Edit 工具栏

第二种设定目标字符串的方法需要 Find 对话框。虽然这种方法不如第一种直接，但

它提供了更多的选择。例如，如果想搜索的字符串碰巧在屏幕中，可以直接借用它，而不必再输入它。如果目标字符串只是一个单词，只需通过单击这个单词或将插入符置于这个单词上；否则，可通过拖动鼠标光标来选中所需文本。接下来，通过按 Ctrl+F，或选择 Edit（编辑）菜单下的 Find 命令，来打开 Find 对话框。当对话框显示出来时，所选中的文本已经填入对话框中了。

可通过设定是否区别大小写，或是否整词匹配，来使搜索更为精确。单击区别大小写（MatchCase）复选框，将设定一个区别大小写的搜索。单击整词匹配（Match Whole Word Only）复选框，可忽略包含于另一单词之中的目标串。例如，对于目标串“any”，在整词匹配搜索中，编辑器只找出以完整单词形式出现的“any”，而忽略掉类似“many”、“anywhere”之类的单词。

在 Find 对话框中，单击全部标记（Mark All）按钮，就可在每一个搜索到的目标处设置一个无名书签。当继续使用 Find 命令搜索其他目标时，这种设置能够回到前面的目标处。

Visual FORTRAN 编辑器中还有一个叫增量搜索（Incremental Search）的命令，它使搜索工作在输入目标串的同时就开始进行。在一个打开的文档中按 Ctrl+I，则“Incremental Search:”就会立即出现在屏幕左下角的状态栏中。在输入目标串的同时，编辑器就开始搜索，通常不等输完，搜索就已完成。当编辑器找到正在寻找的字符串时，按 Enter 键，或一个方向键就可回到编辑模式。如要再次搜索同样的目标串，按 F3 或单击适当的工具按钮即可。按 Shift+Ctrl+I 键可改变搜索方向，以使编辑器从插入符所在位置向前搜索。

2. 在已打开的文档中替换文本

如果想替换文档中的某些字符串，可选择 Edit 命令下的替换（Replace）命令。这个命令将给出一个类似于 Find 对话框的对话框，只不过这个对话框有两个文本框，需要两个字符串。第一个文本框中放置普通的目标字符串，第二个文本框中则要放置要用来替换的新字符串。如果将第二个文本框置空，则编辑器将把找到的所有目标串删除。为了有选择地搜索或替换，当编辑器找到目标串时，单击 Replace 按钮，系统将自动跳到下一个目标串处。单击查找下一个（FindNext）按钮，系统将跳过这个字符串，而不改变它。单击全部替换（Replace All）按钮，将会一次性完成所有替换。可以只向前，或只向后搜索或替换，但不能在多个文件中搜索或替换。

如果在激活 Replace 对话框之前，已选中了超过一行的文本，则选择（Selection）单选钮将自动打开，这表明编辑器将把搜索或替换操作限制在所选章节之中。单击整个文件（Whole File）单选钮会取消这一设置。虽然可通过在按下 Alt 键的同时向右下方拖动鼠标光标来选中一列字符串，但通常情况下，不能进行列的替换。如果选中一列字符串，Selection 单选钮将不会打开。

3. 在磁盘文件中搜索文本

给出一个目标字符串，Visual FORTRAN 文本编辑器能找出某个文件夹下所有包含目标串的文件，包括位于子文件夹内的文件。单击 Edit 菜单下的在文件中查找（Find In Files）命令或按编辑工具栏中的按钮，可打开如图 7.8 所示的对话框。

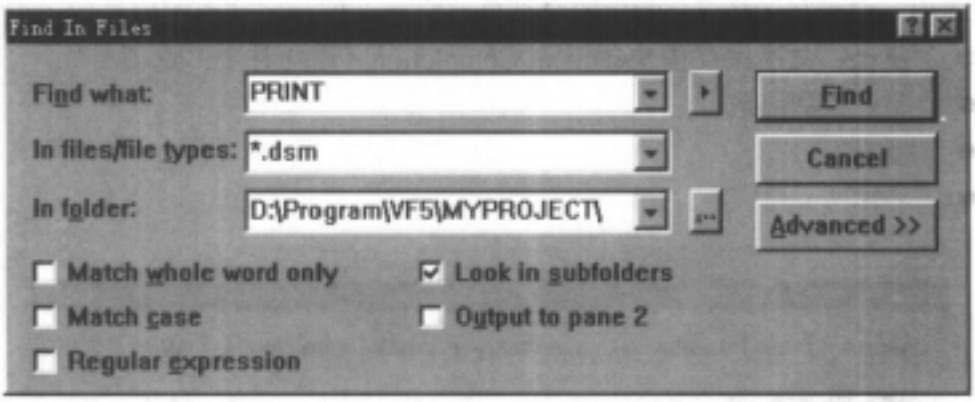


图 7.8　Find In Files 对话框

这个对话框需要指定目标串、文件类型以及搜索过程所在的文件夹。通常，默认文件夹是当前软件所在的文件夹；如果想搜索其他文件夹，在文件夹中（In Folder）框中，输入其路径，或单击框旁的有三个点的按钮，来寻找所需文件夹。在子文件夹中查找（Look In Subfolders）复选框决定编辑器是否在子文件夹中进行搜索。其默认值为打开。单击高级（Advanced）按钮，可设定除子文件夹之外的其他文件夹，搜索也将在这些文件夹中进行。

在第 1 章中已经看到，Find In Files 命令通常将其找到的文件在 Output 窗口中的 Find In Files 1 选项卡下列出（参看第一章中的图 1.9）。如要将其在 Find In Files 2 选项卡下列出，只需激活输出到窗格 2（Output To Pane 2）复选框，如图 7.8 所示。这个选项可同时显示两个独立的文件列表，这样，当前搜索的结果不会覆盖前一次搜索的结果。

如果已经设置好各项，单击 Find 按钮。当 Visual FORTRAN 找到一个包含所给出的目标串的文件时，它把其文件名和路径列在 Output 窗口中，同时还列出此文件中第一个目标串所在行，这便可以看到目标串在文件中是怎么用的。双击某个文件，可在文本编辑器中打开它。

在进行文件搜索之前，Visual FORTRAN 首先保存任何在文本编辑器中打开而未保存的文档，这将确保所搜索的文件都是最新版本的。可通过调整 Options 对话框中编辑器（Editor）选项卡下的两个标有“Save Before Running Tools（在运行工具之前保存）”和“Prompt Before Saving Files（在保存文件之前提示）”的复选框来改变这一属性。关闭“Save Before Running Tools”复选框，将使 Visual FORTRAN 在搜索前不保存文档，这样就使搜索在上次保存的文档中进行。如果希望在激活 Find In Files 命令之前，由自己决定是否保存文档，则关闭这两个选项。这将使编辑器在保存每个文档前先征求用户的意见。

7.2.4　定制编辑器

Visual FORTRAN 文本编辑器可以改变自己的许多特征，以更好地适应用户的工作风格。前面已经讨论过 Tools 菜单中的 Customize 命令，它让用户定制工具栏，并给命令约定快捷键。如果要改变编辑器界面的其他特征，从 Tools 菜单中选择选项（Options）命令。

Options（选项）命令显示图 7.9 所示的对话框，它允许你指定文本编辑器的下列特征：

- 外观、文档保存和 Statement Completion（语句补全）选项
- 制表符和缩进
- 模拟
- 字体

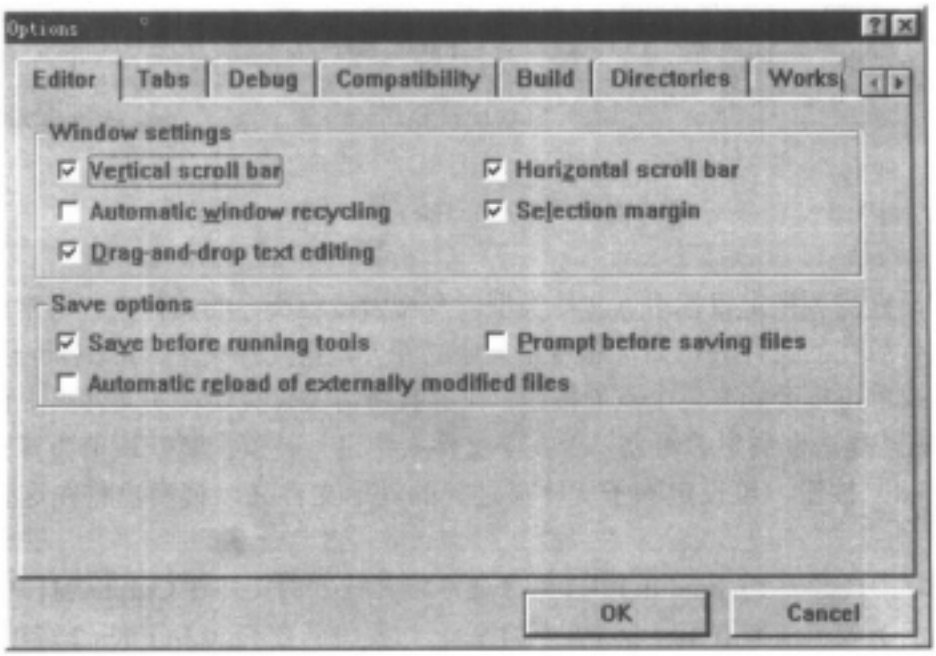


图 7.9 Options 对话框的 Editor 选项卡

在对话框的 Edit 选项卡中，可以根据用户的爱好来选择编辑器如何及何时需要保存文档复选项。也可以指定编辑器在编译之前是否自动保存已改变的文件，以及在保存文档之前是否提示用户。

值得一提的是同一页中的选择边界（Selection Margin）复选项。选择边界是文档窗口左边的大约半英寸宽的一列阴影区，通过单击该边界就可以选中附近的一整行。该空格也可以保留书签的图标，以及后面将在后面提到的调试断点。如果用户希望恢复这些空白来显示文档，可以清除掉此复选项来禁用选择空格。

Visual FORTRAN 文本编辑器可以在一定程度上模拟 Visual C++2.0、BRIEF 或 Epsilon 程序员的编辑器。如果用户习惯于这些产品，可以打开相应的模拟选项。单击兼容性（Compatibility）选项卡，然后从列表中选择编辑器中的一个，或通过打开单个的复选项来设置想要的选项。

格式 Format）选项卡允许用户给编辑器窗口指定字体类型和颜色。可以从 Category 列表中单击源窗口（Source Windows）来查看当前的字体。默认情况下，字体为 10 号 Courier，但用户可以把它改成任何所希望的类型和大小。Color 区允许希望为编辑器中不同的标记和文本（如源文件注释和 HTML 标记）调整背景和前景颜色。要改变颜色，从列表中选择一个条目，然后从组合框中选择想要的颜色。

7.3　图形编辑器

本节简要介绍图形编辑器和相关的一些基本概念。

7.3.1　位图，工具栏和光标

Visual FORTRAN 图形编辑器是创建和修改程序的图形资源的地方。图形资源包括位图、工具栏、图标和光标。

1．位图

在为一个新的位图打开图形编辑器时，它提供一个边长为 48 像素的正方形的干净区域。位图不要求一定是正方形，它们可以是边长不大于 2048 个像素的任何尺寸的矩形。要调整工作区域的大小，拖动工作区域边界上一个调整大小的手柄，在拖动手柄的同时，注意一下编辑器状态栏里的新尺寸。也可以通过单击查看（View）菜单上的 Properties 打开位图属性（Bitmap Properties）对话框，从中输入想得到的大小。

2．工具栏

工具栏位图是一系列覆盖工具栏按钮的图形，每一个按钮都有一个图形。默认状态下，每一个图形都是 16 个像素宽，15 个像素高，它适合标准的 24 个像素、宽 22 个像素高的工具栏按钮。通过以典型的 Windows 方式拖动编辑器工作空间的边界，可以设置一个图形的尺寸是大一点，还是小一点，是宽一点，还是窄一点。由于不能使一个工具栏里按钮具有不同的尺寸，所以，新尺寸将应用于所有的工具栏图形。当保存工作时，Visual FORTRAN 自动指定 RC 文件里工具栏按钮的新尺寸。

3．图标

图标是特殊的位图，设计用来用图形方式表示程序或文档。一般情况下，图标指定给一个框架窗口，以便在窗口的标题栏里显示图形，当图标指定给程序的主窗口时，图标资源叫做程序图标或应用程序图标。图标是最普遍使用的一种图标资源。但不论图标是表示主窗口，还是屏幕上的另一个对象，图标都是在 Visual FORTRAN 图形编辑器里创建的，而且创建的方式相同。

一个图标资源包含多个图形，它们经常是同样设计的不同尺寸的图形。例如，Microsoft 建议，Windows 程序提供给它图标资源的三个图形，每一个图形具有不同的尺寸：

（1）　一个边长为 16 个像素的正方形，16 色图形，Windows 将在程序的标题栏、任务栏和用小图标显示的目录列表里显示它。

（2）　边长为 32 个像素的正方形，16 色图形，在关于（About）对话框这样的对话框窗口里使用，或在桌面上表示程序，或在用大图标显示的目录列表里显示。

（3）　边长为 48 个像素的 256 色图形，在系统 Display Properties 对话框里，若选中效果（Effects）选项卡里的使用大图标（Use Large Icon）选项

光标也是位图，是为鼠标光标而设计的，当鼠标放置在程序的用户区域里时，它将提

供服务。

这些图形资源的编辑是统一的：Visual FORTRAN 环境提供一个适用于所有场合的图形编辑器。用户可能会看到，在线帮助里存在“toolbar editor（工具栏编辑器）”或者“icon editor（图标编辑器）”参考，这些只是意味着 Visual FORTRAN 图形编辑器可以应用于任何一种特定类型的资源。

图形编辑器可以处理不同资源类型的多个文档。对于每一种资源类型，编辑器的外观差别不大，文档窗口的左上角的图标指示你正在使用的资源类型。表 7.2 显示每一种资源文档对应的图标，并列出这个图形编辑器可以读写的文件类型。当使用 Open 命令打开已有的资源文档，而且这个文档还具有表 7.2 第三列里的某个扩展名时，Visual FORTRAN 自动打开图形编辑器，并调整具体形式以适合这种资源类型。

表 7.2 图形编辑器图标和文件类型

资 源	图 标	输入文件类型（扩展名）	输出文件类型
位图（Bitmap）		BMP、DIB、EPS、GIF、JPG	BMP
光标（Cursor）		BMP	BMP
图标（Icon）		CUR	CUR
工具栏（Toolbar）		ICO	ICO

7.3.2 启动图形编辑器

可以用两种略有不同的方法来启动图形编辑器并创建一种新的图形资源。

第一种方法是打开一个项目，并添加一个新的图形资源到项目中，从 Insert 菜单中选择 Resource，出现如图 7.10 所示的对话框。在列出的资源文件中双击位图或光标，或者工具栏。Visual FORTRAN 打开图形编辑器，在标题栏里显示为这种资源文档指定的标识符。根据资源类型的不同，标识符是像 IDB_BITMAP1，IDC_CURSOR1，IDI_ICON1 或 IDR_TOOLBAR1 这样的名字。以后在编辑器里打开的资源将得到相似的标识符，只是在数字上加 1，例如 IDI_ICON2，IDI_ICON3 等等。

启动图形编辑器的第二种方法是，创建一个新的图形资源，但不添加它到项目文件列表里。例如，创建一个资源库，或为工具栏按钮设计位图。这种方法不要求打开项目，只单击 File 菜单上的 New，并在 File 选项卡里，从列表里选择想要的资源类型：位图文件（Bitmap File），图标文件（IconFile）或光标文件（Cursor File）。

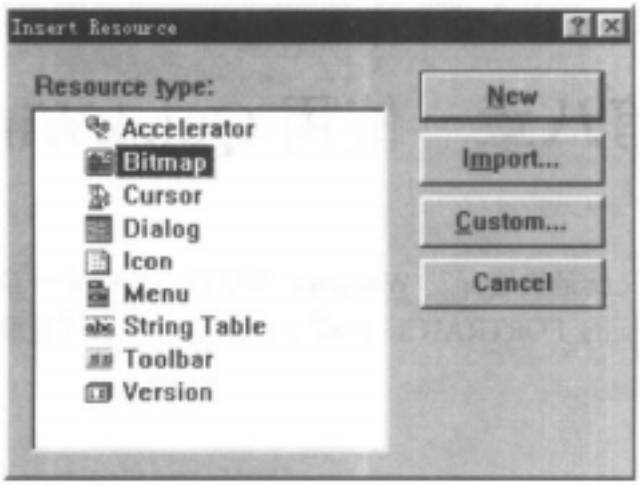


图 7.10　插入资源对话框

7.3.3　图形编辑器的工具栏

图形编辑器的工作区是分成两个窗口。默认状态下，左边窗口以图形的实际尺寸显示图形，右边窗口显示的图形放大到大约 36（6×6）倍。放大的图形有一个覆盖形式的网格，网格中，每一个方块表示实际图形上的一个像素。如果用户以前使用过画图程序，例如 Microsoft Windows 的画笔程序，那么，就会对 Visual FORTRAN 图形编辑器很熟悉。通过单击左边或右边窗口中的任何地方，来选择要绘制的图形。如果要处理细节，可以在大的工作区域里进行，并在实际尺寸图形窗口里观察效果。将两个窗口分割开的分隔条是可以移动，只要用鼠标向左或向右拖动它即可。要开始画图形，通过单击显示在图 7.11 里图形（Graphics）工具栏上的一个按钮，来选择一个合适的工具。与在 Visual FORTRAN 环境里使用其他工具栏一样，用户可以通过向内或向外拖动工具栏来缩小或放大它。

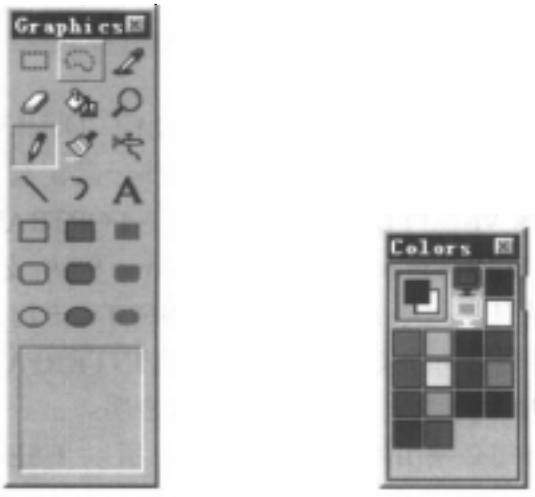


图 7.11　Graphics 和 Colors 工具栏

图形编辑器具体的工具按钮都很直观，使用也比较灵活，这里就不再详述。

第八章 使用 QuickWin

QuickWin 顾名思义就是快速开发 Windows 应用程序。在这一章里，用户可以很快就写出具有 Windows 特征的 FORTRAN 应用程序，这对于使用其他编译器的用户来说是不可思议的。

8.1 概述

Visual FORTRAN 的 QuickWin 运行库帮助用户把图形程序转化为简单的 Windows 应用程序。尽管通过 QuickWin 并不能使用 Windows 的全部能力，但 QuickWin 无疑是简单易学易用的。QuickWin 应用程序支持基于像素的图形、实坐标图形、文本窗口、字符字体、用户定义的菜单、鼠标事件和文本或图形的编辑（选择/复制/粘贴）。

使用 QuickWin 特性的程序必须显示地用语句 **USE DFLIB** 访问 QuickWin 图形库，同时必须把用户项目的类型选为“QuickWin Graphics”或“Standard Graphics”。

在 Visual FORTRAN 中，图形程序必须是 QuickWin, Standard Graphics、 Windows 或 OpenGL 应用程序。Standard Graphics（标准图形）应用程序是 QuickWin 的子集，它只支持一个窗口。用户可以在 Developer Studio 中创建新项目的下拉菜单中的项目类型中选择 QuickWin 或 Standard Graphics 应用程序类型。或在编译选项中选择/libs:qwin（选择 QuickWin）或/libs:qwins（选择 Standard Graphics）。

注意：QuickWin 和 Standard Graphics 应用程序不能是 DLL，并且 QuickWin 和 Standard Graphics 不能被连接到 DLL 中的运行例函数（routine）。这意味着/libs=qwin 选项和/libs=dll 选项以及/threads 选项不能一起使用。

本章介绍了 QuickWin 库例程序的主要类别，给出了 QuickWin 图形特性和创建、显示特性中的应用的概览，并能和定制菜单和鼠标例程序一起定制 QuickWin 应用程序。

用户还可以使用其它像 Visual FORTRAN 一样支持 FORTRAN 调用约定的语言来访问 QuickWin 库。这个特性包支持 Windows NT 和 Windows 95 所支持的一切视频格式。

使用 QuickWin 特性的任何程序必须包含 **USE DFLIB** 语句来包含 **USE DFLIB** 图形库。DFLIB.MOD 模块文件包含每个 QuickWin 例程的 **INTERFACE** 语句、派生类型声明、符号常量声明和 **EXTERNAL** 声明中的子程序和函数声明。

由于 **INTERFACE** 语句必须出现在程序体外部，所以 **USE DFLIB** 语句必须程序在程序单元体的外部。通常这意味着在其它语句之前就要写上 **USE DFLIB** 语句。

如果图形例程没有 **PROGRAM** 语句，则 **USE DFLIB** 必须在任何声明语句（例如 **IMPLICIT NONE** 或 **INTEGER**）或任何包含声明语句的其它模块之前在每个作图形调用的子程序中出现。

8.1.1　QuickWin 的能力

用户可以使用 QuickWin 库来完成下面一些任务：

- 把控制台程序（console program）编译成简单的 Windows 应用程序
- 像基于 Windows 的应用程序一样可以最小化和最大化 QuickWin 应用程序
- 调用图形例程
- 载入和保存位图
- 选择、复制和粘贴文本、图形或文本和图形
- 检测和响应鼠标按键
- 显示图形输出
- 改变缺省应用程序菜单或增加编程菜单
- 创建定制图标
- 打开多个子窗口

第六章的最后我们编写了一个简单的程序 Hello，编译的结果实际上是 Win32 控制台应用程序，它的运行的形式和一般在 Windows 中的 DOS 窗口运行 MSDOS 程序类似，如图 8.1 所示。

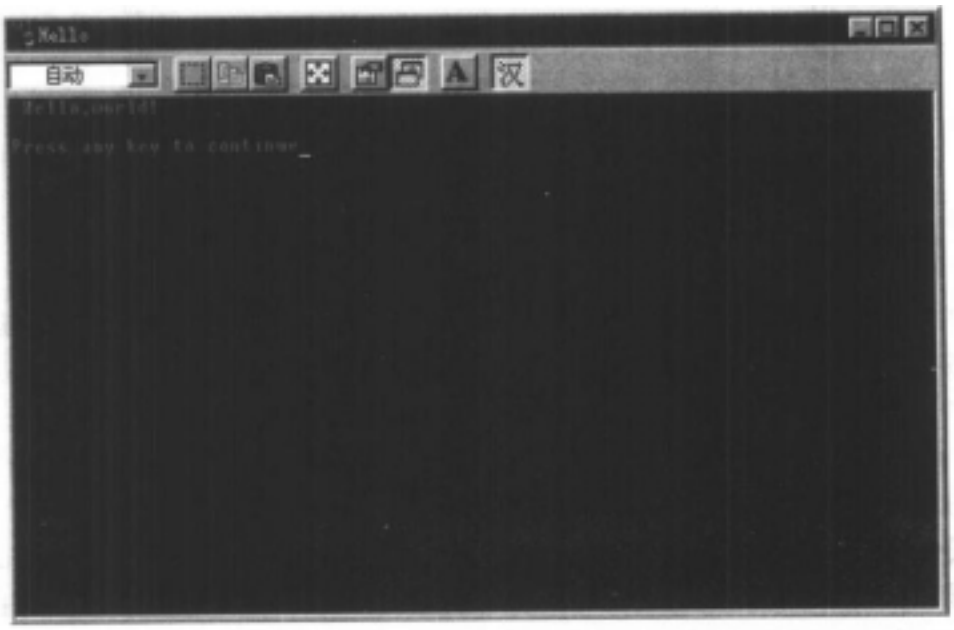


图 8.1　Win32 控制台程序运行结果

下面我们来创建一个 QuickWin 程序：

- 首先，在文件（File）菜单中选择新建（New）选项，出现如第六章中图 6.1 所示的新建文件对话框，在对话框中选择 QuickWin Application，然后在项目名称（Project name）行输入项目名，例如 QuickWin_Hello。文件位置（Location）也

可以由用户改变。然后单击 OK 按钮。

- 这时工作空间（Workspace）窗口中出现 FileView 选项卡。从 FileView 可以看到 QuickWin_Hello 的工作空间的结构：现在只有两层，一层是 QuickWin_Hello 工作空间，另一层是 QuickWin_Hello 文件，如图 8.2 所示。
- 在 QuickWin_Hello 文件这一层次上单击右键会弹出一个菜单，选择向项目添加文件（Add Files to Project），这时会出现添加文件的对话框，在对话框中我们可以通过浏览选择第六章编写的 Hello.f90 文件。
- 这时，QuickWin_Hello 文件左边的节点出现了一个加号，表明它包含了文件，那就是 Hello.f90 文件。

图 8.2 工作区间结构

- 然后在建立（Build）菜单中选择全部建立（Build All）或在 Build 工具栏中按■按钮，编译系统开始对程序进行编译、连接，最后生成 QuickWin_Hello.exe 文件。
- 如果编译通过，在建立（Build）菜单中选择运行 QuickWin_Hello.exe（Execute QuickWin_Hello.exe）或在 Build 工具栏中按■按钮就可以运行本程序了。程序的运行结果如图 8.3 所示。

对比 QuickWin_Hello.exe 和 Hello.exe 这两个程序不难发现 QuickWin_Hello.exe 是一个具有明显 Windows 特征的应用程序：可以用鼠标按住标题条 QUICKWIN_HELLO 或 Graphic1 在屏幕上移动窗口；可以按住边框改变窗口的大小；还可以按最大化和最小化按钮；它甚至还有几个标准的菜单。如果单纯用 C 语言来写一个即使没有菜单的类似的程序大约需要 80 行代码，而在我们的程序中没有任何关于 Windows 本身的代码，一切都在创建 QuickWin 应用程序时就完成了。

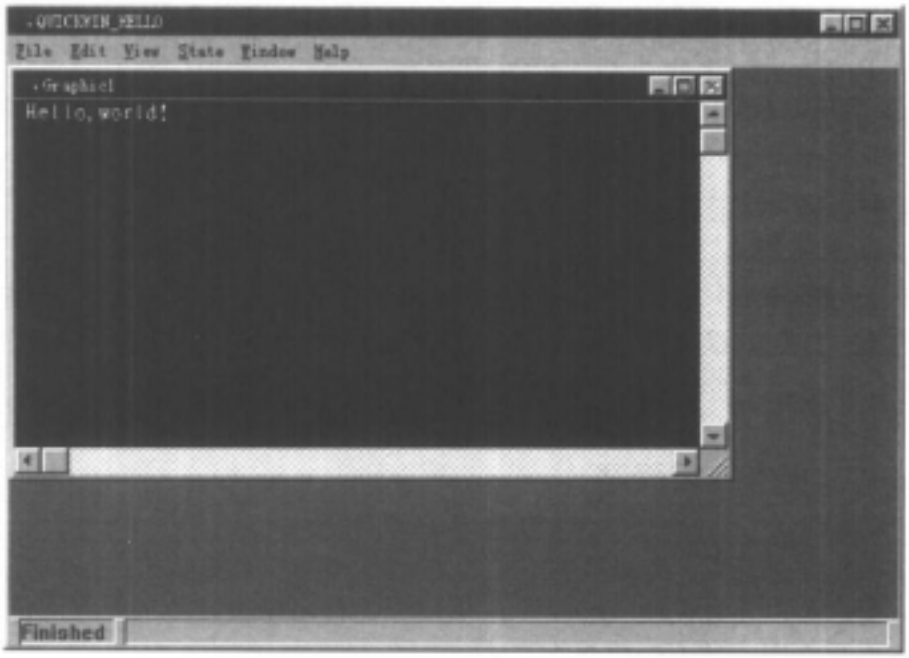


图 8.3　QuickWin_Hello.exe 的运行结果

8.1.2　QuickWin 和基于 Windows 的应用程序的比较

QuickWin 并不能提供 Windows 的全部功能。尽管在 QuickWin 和控制台程序中可以调用很多 Win32 API（Application Programming Interface，应用程序编程接口），但还有相当一部分 Win32 API（例如 GDI 函数）不能使用，这些 Win32 API 只能由完整的 Windows 应用程序才能调用。如果需要下列某种应用时，用户应该选择基于 Windows 的应用程序（Windows-based applications）而不是 QuickWin 。

- 用户程序有 OLE（Object Linking and Embedding，对象嵌入和连接）控件
- 用户希望直接访问 GDI（Graphical Data Interface，图形数据接口）函数
- 用户希望自定义标准信息
- 用户希望创建内容比标准 SDI（Single Document Interface，单文档界面）或 MDI（Multiple Document Interface，多文档界面）多一些的应用程序（例如，用户希望自己的应用程序有像 Windows 中计算器的用户区域的对话框）

8.2　QuickWin 程序的类型

QuickWin 库创建标准图形应用程序或 QuickWin 图形应用程序，这由用户选择的项目类型来决定。标准图形应用程序只支持一个窗口并且不支持可编程菜单。QuickWin 图形应用程序支持多个窗口和用户定义的菜单。任何 FORTRAN 程序，不论是否包含图形，都可以以 QuickWin 应用程序来编译。Microsoft Developer Studio 可以用来创建、调试和

执行标准图形应用程序和 QuickWin 应用程序。

在 Developer Studio 中建立 QuickWin 应用程序或标准图形应用程序可以从新建项目的项目类型中选择。

8.2.1 标准图形应用程序

标准图形应用程序是只有一个最大化后覆盖整个区域的窗口。应用程序窗口可以包含文本和图形的输入输出，缺省是还是一个可滚动的文本窗口。窗口体只有边界，标题栏和滚动条。标准图形应用程序不能创建可编程菜单和多个子窗口。

8.2.2 QuickWin 图形应用程序

图 8.3 就是一个典型的 QuickWin 图形应用程序，窗口体有边界、标题栏、滚动条和缺省的菜单。用户可以通过使用 QuickWin 的增强特性来修改、添加或删除缺省菜单项，鼠标事件的响应和在窗口体内创建多个子窗口。

8.2.3 QuickWin 用户界面

所有的 QuickWin 应用程序都创建应用程序窗口，子窗口是可选的。标准图形应用程序和 QuickWin 图形应用程序的一般特征如下：

- 窗口内容可以以位图或文本复制到剪贴板以供其它程序的粘贴或打印之用。在标准图形应用程序中，复制的是整个窗口，因为没有编辑（Edit）菜单。在 QuickWin 图形应用程序中，窗口中的任何部分都是可以选择和复制的。
- 如果需要，垂直和水平滚动条会自动出现。
- 基本应用程序的.exe 的名称出现在窗口标题栏里。
- 关闭窗口的同时中止程序的运行。

另外，QuickWin 图形应用程序还有状态条和菜单条。在窗口的按钮上的状态条显示了窗口程序的当前状态。

8.2.4 缺省的 QuickWin 菜单

缺省的多文档界面菜单条由六个菜单：文件（File）、编辑（Edit）、参看（View）、状态（State）、窗口（Window）和帮助（Help）。这些菜单分别如图 8.4 至图 8.9 所示。

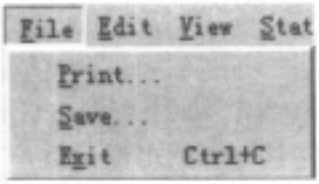


图 8.4 文件（File）菜单

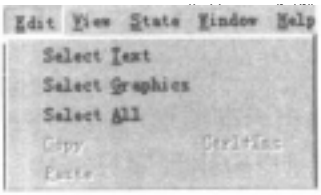


图 8.5 编辑（Edit）菜单

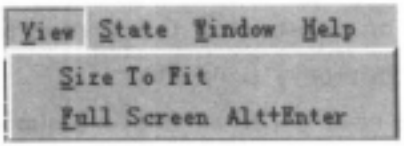


图 8.6　参看（View）菜单

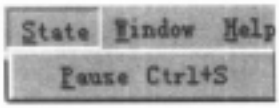


图 8.7　状态（State）菜单

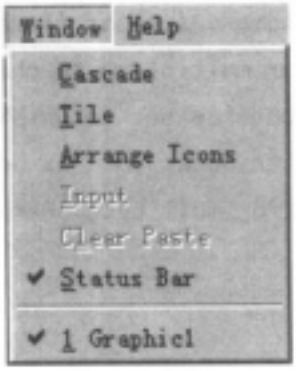


图 8.8　窗口（Window）菜单

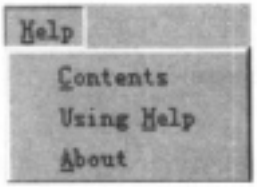


图 8.9　帮助（Help）菜单

8.3　创建 QuickWin 窗口

QuickWin 库包含多种创建和控制 QuickWin 窗口的例程。

8.3.1　访问窗口属性

SETWINDOWCONFIG 和 GETWINDOWCONFIG 分别用来设置和获得当前窗口属性。这些属性存储在由 DFLIB.MOD 定义的 windowconfig 派生类型中。它包含下列参数：

```
TYPE windowconfig
    INTEGER(2) numxpixels          ! Number of pixels on x-axis.
    INTEGER(2) numypixels          ! Number of pixels on y-axis.
    INTEGER(2) numtextcols         ! Number of text columns available.
    INTEGER(2) numtextrows         ! Number of scrollable text lines available.
    INTEGER(2) numcolors           ! Number of color indexes.
    INTEGER(4) fontsize            ! Size of default font. Set to
                                   ! QWIN$EXTENDFONT when using multibyte
                                   ! characters, in which case
                                   ! extendfontsize sets the font size.
    CHARACTER(80) title            ! Window title, where title is a C string.
    INTEGER(2) bitsperpixel        ! Number of bits per pixel. This value
```

```
                                    ! is calculated by the system and is an
                                    ! output-only parameter.
                                    ! The next three parameters support multibyte
                                    ! character sets (such as Japanese)
    CHARACTER(32) extendfontname    ! Any non-proportionally spaced font
                                    ! available on the system.
    INTEGER(4) extendfontsize       ! Takes same values as fontsize, but
                                    ! used for multiple-byte character sets
                                    ! when fontsize set to QWIN$EXTENDFONT.
    INTEGER(4) extendfontattributes ! Font attributes such as bold and
                                    ! italic for multibyte character sets.
END TYPE windowconfig
```

1. SETWINDOWCONFIG 语句

QuickWin 功能：

设置子窗口属性

模块：

USE DFLIB

语法：

result = **SETWINDOWCONFIG** (*wc*)

wc 的类型是派生类型 windowconfig ，它包含了窗口的属性。

返回值：

类型为 LOGICAL(4)，如果成功为 **.TRUE.** 否则为 **.FALSE.**。

2. GETWINDOWCONFIG 语句

QuickWin 功能：

获得子窗口属性

模块：

USE DFLIB

语法：

result = **GETWINDOWCONFIG** (*wc*)

wc 的类型是派生类型 windowconfig ，它包含了窗口的属性。

返回值：

类型为 LOGICAL(4)，如果成功为 **.TRUE.** 否则为 **.FALSE.**（例如当前没有活动的子窗口）。

如果用户使用 SETWINDOWCONFIG 将变量 windowconfig 中的变量置为-1，系统将被设为最高的分辨率。用户还可以通过指定影响窗口大小的参数（例如 x 和 y 方向的像素，行与列的数目，以及字体大小）来设置窗口的实际尺寸。如果不调用 SETWINDOWCONFIG，窗口默认为最适合的分辨率和 8×16 的字体。颜色的数目由使用

的系数驱动决定。下面的例子中指定了 x 和 y 方向的像素数目，系统还计算了窗口的行数和列数：

```
USE DFLIB
TYPE (windowconfig) wc
LOGICAL status
! 设置 x & y 像素为 800X600 ，字体为 8x12.
        wc.numxpixels  = 800
        wc.numypixels  = 600
        wc.numtextcols = -1
        wc.numtextrows = 302
        wc.numcolors   = -1
        wc.title       = " "C
        wc.fontsize    = #0008000C
        status         = SETWINDOWCONFIG(wc)
```

在上例中，变量 wc.numtextrows 被置为 302，为的是允许 300 行可滚动文本（可使用的是 n-2 行）。

如果无法设置用户要求的配置，**SETWINDOWCONFIG** 返回.**FALSE.** 并计算最适合要求配置并能够设置的参数值。对 **SETWINDOWCONFIG** 的另一个调用可以建立这些值：

```
IF(.NOT.status) status = SETWINDOWCONFIG(wc)
```

8.3.2　创建子窗口

OPEN 语句中的 **FILE='USER'**选项会打开一个 Visual FORTRAN 和其它单元同等对待的单元。而 Windows NT 和 Windows 95 把单元视为子窗口。子窗口在缺省时是一个 30 行 80 列的可滚动文本窗口。如果 OPEN 语句包含 **FILE=' '**，Visual FORTRAN 会显示一个 Windows 打开文件对话框来提示输入要打开的文件的文件名。用户最多可以打开 40 个子窗口。

打开一个窗口显示的是窗口框而不是子窗口。用户必须调用 SETWINDOWCONFIG 或执行一条 I/O 语句或图形语句来显示子窗口。窗口由它的单元号来接受输出，例如：

```
OPEN (UNIT= 12, FILE= 'USER', TITLE= 'Product Matrix')
WRITE (12, *) 'Enter matrix type: '
```

用 **FILE='USER'**打开的子窗口必须是作为顺序访问格式文件打开的（也是缺省的格式），其它文件指定（直接访问、二进制或非格式）会产生运行错误。

8.3.3 赋给窗口焦点和设置活动窗口

当一个窗口是活动窗口时，它接受图形输出（例如由 **ARC, LINETO** 和 **OUTGTEXT** 等发出的输出），但是整个窗口并不在最前面显示，于是也没有焦点（focus）。当一个窗口需要焦点时，可以用鼠标单击它、对其进行 I/O 操作或调用 **FOCUSQQ** 。如果一个窗口需要放到最前台，它必须有焦点。

有焦点的窗口始终在最上层，所有其它窗口的标题栏都会变灰。一个窗口可以有焦点但不是活动并且不直接接受图形输出，也就是说图形输出是独立于焦点的。

在大多数情况下，焦点和活动应该应用于同一个窗口。这是 QuickWin 的默认行为，程序员也必须有意识地覆盖默认值。

某些像 GETCHARQQ, PASSDIRKEYSQQ 和 SETWINDOWCONFIG 之类的 QuickWin 例程常常影响活动窗口而不管窗口是否是焦点，这是因为它们不把单元号作为输入参数。

如果另一个窗口成为活动的但不是焦点，在调用例程时这些例程会影响窗口的活动。不过有时对用户会显得不正常，因为在这些情况下 **GETCHARQQ** 会在一个灰色的后台程序要求输入，用户将不得不在输入之前先单击这个窗口。

为了使用这些影响活动窗口的例程，可以对希望设为焦点的窗口（同时也希望设为活动）的调用号作 I/O 操作，或调用 FOCUSQQ （指定单元号）。如果只有一个打开的窗口，则这个窗口是被影响的窗口；如果有多个打开的窗口，则最后一个打开的是被影响的窗口。

OPEN (IOFOCUS=)参数也可以决定当 I/O 语句执行到窗口单元号时窗口是否收到焦点，例如：

```
OPEN (UNIT = 10, FILE = 'USER', IOFOCUS = .TRUE.)
```

IOFOCUS=缺省为**.TRUE.**，除了打开的子窗口是 0,5 或 6 号单元和直接到终端设备，这时 **IOFOCUS=** 缺省为 **.FALSE.**。如果 **IOFOCUS= .TRUE.**，子窗口在每个 **READ, WRITE** 或 **PRINT** 之前都获得焦点。对 **OUTTEXT** 或图形函数（例如 **OUTGTEXT, LINETO** 和 **ELLIPSE**）的调用不会引起焦点的转移。如果对任何非 QuickWin 子窗口使用 **IOFOCUS=**都会产生运行错误。

如果对一个窗口用 **FOCUSQQ**，或由鼠标单击，或非图形的 I/O 操作时，焦点就转移到这个窗口上，除非该窗口在打开时说明了 **IOFOCUS=.FALSE.**。INQFOCUSQQ 可以确定哪个单元有焦点。例如：

```
    USE DFLIB
    INTEGER(4) status, focusunit
    OPEN(UNIT = 10, FILE = 'USER', TITLE = 'Child Window 1')
    OPEN(UNIT = 11, FILE = 'USER', TITLE = 'Child Window 2')
!通过向 Child Window 2 写入来给它焦点:
    WRITE (11, *) 'Giving focus to Child 2.'
```

```
!通过调用 FOCUSQQ 来给 Child Window 1 焦点:
    status = FOCUSQQ(10)
    ...
!寻找当前具有焦点的子窗口的单元号:
    status = INQFOCUSQQ(focusunit)
```

还有一些对焦点进行处理和控制的函数：SETACTIVEQQ 使一个窗口不必在最前面显示就变成活动的； GETACTIVEQQ 返回当前活动子窗口的单元号；GETHWNDQQ 把单元号转换成一个 Windows 句柄，供需要它的函数使用。这些函数的具体用法参见在线文档。

8.3.4 保持窗口打开

只要子窗口对应的单元是打开的，该子窗口就保持打开状态。**CLOSE** 语句中的 **STATUS=**参数决定了在关闭单元后子窗口是否保持打开。如果设置 **STATUS='KEEP'**，和单元联系的窗口保持打开，但不再允许输入或输出。同时，子窗口的菜单上增加了关闭（Close）命令，并且单词“Closed”也添加到窗口的标题中。缺省情况是 **STATUS='**DELETE，它将关闭窗口。

STATUS='KEEP'所保持的关闭单元号后的窗口也属于QuickWin应用程序可打开的40个窗口之一。

8.3.5 控制窗口的大小和位置

SETWSIZEQQ 和 **GETWSIZEQQ** 用来设置和获得窗口的大小和位置。子窗口的位置和尺寸用字符高度和宽度单元来表示，窗口框的位置和尺寸用像素表示。这些位置和大小由 DFLIB.MOD 中定义的 qwinfo 派生类型返回。Qwinfo 派生类型如下：

```
TYPE QWINFO
    INTEGER(2) TYPE      ! Type of action performed by SETWSIZEQQ.
    INTEGER(2) X         ! x-coordinate for upper left corner.
    INTEGER(2) Y         ! y-coordinate for upper left corner.
    INTEGER(2) H         ! Window height.
    INTEGER(2) W         ! Window width.
END TYPE QWINFO
```

GETWSIZEQQ 返回当前窗口框或子窗口的当前或最大化窗口的大小。为了获得子窗口的信息，应该指定和窗口联系的单元号。如果用户没有用 OPEN 语句显式地打开单元号 0, 5 和 6，则它们指的是缺省的启动窗口。为了访问窗口框的信息，应该指定单元号为符号常量 QWIN$FRAMEWINDOW，例如：

```
    USE DFLIB
    INTEGER(4) status
    TYPE (QWINFO) winfo
    OPEN (4, FILE='USER')
    ...
! 获得与4号单元联系的子窗口的当前尺寸
    status = GETWSIZEQQ(4, QWIN$SIZECURR, winfo)
    WRITE (*,*) "Child window size is ", winfo.H, " by ", winfo.W
! 获得窗口框的最大尺寸
    status = GETWSIZEQQ(QWIN$FRAMEWINDOW, QWIN$SIZEMAX, winfo)
    WRITE (*,*) "Max frame window size is ", winfo.H, " by ", winfo.W
```

SETWSIZEQQ 用来指定窗口的位置和大小，例如：

```
    USE DFLIB
    INTEGER(4) status
    TYPE (QWINFO) winfo
    OPEN (4, FILE='USER')
    winfo.H    = 30
    winfo.W    = 80
    winfo.TYPE = QWIN$SET
    status     = SETWSIZEQQ(4, winfo)
```

8.4 定义图形特性

正确使用图形之前要先定义图形特性。

8.4.1 选择显示选项

QuickWin 运行库为用户定义文本和图形显示提供了丰富的例程。这些例程确定了图形环境并控制光标。

SETWINDOWCONFIG 命令可以用来配置窗口属性；DISPLAYCURSOR 用来控制光标是否出现，调用 SETWINDOWCONFIG 之后光标就变成不可见了。要显示光标必须用

SETGTEXTROTATION 设置文本输出的当前方向；GETGTEXTROTATION 返回当前设置。当前方向用于对 OUTGTEXT 的调用中。

8.4.2 设置图形坐标

设置坐标例程控制图形在屏幕上出现的位置。Visual FORTRAN 可以识别下列坐标：

- 固定物理坐标
  由硬件和使用的显示模式决定。
- 观察口坐标
  用户可以在应用程序中定义的坐标。
- 窗口坐标
  用户可以定义简化浮点数据缩放。

如果用户不干预，固定物理坐标和观察口坐标是同样的。物理坐标的原点(0, 0)始终在显示区域的左上角。对于 QuickWin，显示区域指的是子窗口的用户区域，而不是显示器的整个屏幕（除非子程序是全屏幕模式）。X 轴的正方向是从左到右，y 轴的正方向是从上到下。缺省的观察口具有所选模式的尺寸。在 QuickWin 应用程序中，用户可在子窗口的当前用户区域之外绘图。如果把子窗口变大些，可看到原来在窗口框外面的部分。

Visual FORTRAN 还提供在物理坐标、观察口坐标和窗口坐标之间的相互转换的例程：GETPHYSCOORD 把观察口坐标转换为物理坐标；GETVIEWCOORD 把物理坐标转换为观察口坐标；GETVIEWCOORD_W 把窗口坐标转换为观察口坐标；GETWINDOWCOORD 把观察口坐标转换为窗口坐标。具体见下一章“绘制图形基础”。

用户可以用 SETWINDOWCONFIG 函数来设置 x 轴和 y 轴的像素数目，还可以通过 GETWINDOWCONFIG 函数返回的 *wc.numxpixels* 和 *wc.numypixels* 值访问 x 轴和 y 轴的像素数目。类似地，GETWINDOWCONFIG 函数还通过 *wc.numcolors* 的值返回当前显示模式下可用的颜色范围。

SETCLIPRGN 和 SETVIEWPORT 函数可以用来定义图形区域。这两个函数定义的都是可以用来作图形输出的窗口区域的一部分或子集。SETCLIPRGN 函数不该变观察口坐标，只是遮盖部分屏幕；SETVIEWPORT 函数则重新设置用户限定的边界，并设原点在该区域的左上角。

观察口坐标系统的原点可以用 SETVIEWORG 函数移到一个和物理原点相关的新位置。如果不考虑观察口坐标，用户可以始终通过使用 GETCURRENTPOSITION 函数 和 GETCURRENTPOSITION_W 定位图形输出位置。

使用窗口坐标系统时，用户可对任何数据的集合进行缩放以适应屏幕。可以定义数据方便的任何范围的坐标（例如 0 至 5000）作为窗口坐标。通过告诉程序用户希望窗口坐标系要适合一个屏幕上的特定区域（到特定的观察口坐标集合的一个映射），可以缩放一个图表或图形到任意希望的尺寸。SETWINDOW 定义由指定值限定的用户坐标系统。

8.4.3　使用颜色

如果用户使用的是 VGA 显示器，一次最多显示 256 色，这 256 色保存在调色板中。用户可以在调色板中选择 262,144 色 (256K)，但一次只能显示 256 色。调色板例程 REMAPPALETTERGB 和 REMAPALLPALETTERGB 把红绿蓝（RGB）赋给调色板索引。

使用颜色索引的函数和子例程靠调色板和 RGB 之间的映射来创建图形输出。REMAPPALETTERGB 把一种颜色重新映射到一种 RGB；而 REMAPALLPALETTERGB 则重新映射整个调色板，最多至 236 种（其中系统保留 20 种颜色）。在只能显示 20 或更

少颜色的机器上不能重新映射调色板。

SVGA 和真彩色显示适配器可以分别显示 262,144 (256K)种和 16.7M 种颜色。如果使用调色板，所能使用的颜色由调色板能够获得的颜色决定。

为了访问整个可用的颜色集合而不是调色板的 256 或更少的颜色，用户应该使用直接指定颜色值的函数。这些函数的名称最后为 RGB，并使用红绿蓝颜色值，而不是调色板的索引。例如 SETCOLORRGB, SETTEXTCOLORRGB 和 SETPIXELRGB 直接指定颜色值，而 SETCOLOR, SETTEXTCOLOR 和 SETPIXEL 每个函数都指定一个调色板颜色索引。如果希望同时显示多于 256 颜色，必须专门使用 RGB 直接颜色值函数。

8.4.4 设置图像属性

画弧、椭圆以及其它基本图形的输出例程并不指定颜色或线型等信息。不仅如此，这些例程还依赖于其它例程对属性的独立设定。

GETCOLORRGB （或 GETCOLOR）和 SETCOLORRGB （或 SETCOLOR）获得或设置当前颜色值（或索引），使用这些颜色的有 FLOODFILLRGB （或 FLOODFILL）、OUTGTEXT 和画外形的例程。类似地，GETBKCOLORRGB （或 GETBKCOLOR）和 SETBKCOLORRGB（或 SETBKCOLOR）则用来获得或设置当前背景颜色。

GETFILLMASK 和 SETFILLMASK 可以用来返回和设置填充掩模（mask）。掩模是一个 8×8 位的数组，每一位都表示一个像素。如果某一位的值为 0，在内存中的这个像素保留不变；如果某一位的值为 1，则该像素将赋为当前颜色值。掩模数组像一个模板，并可以在整个填充区域重复。在掩模中，当前颜色画的像素形成一个图案，图案的多次重复就掩盖了背景并创建了相当数量的填充图案。在画阴影时这两个例程是很有用的。

GETWRITEMODE 和 SETWRITEMODE 返回或设置画线时使用的当前逻辑写模式。逻辑写模式可以设置为$GAND, $GOR, $GPRESET, $GPSET 或$GXOR，这些值决定在新画的和已经存在的屏幕和图形之间的相互作用。逻辑写模式影响的是 LINETO, RECTANGLE 和 POLYGON 例程。

GETLINESTYLE 和 SETLINESTYLE 获得或设置当前线型。线型由 16 个字长的掩模决定，掩模可以从 5 种可选择的线型中确定一种。用户可以用这两个例程创建多种虚线，它们可以影响 LINETO, RECTANGLE 和 POLYGON 例程。

8.5 显示图形输出

运行图形库例程可以画几何图形、显示文本、显示基于字体的字符和在内存与图像之间互相传递。

8.5.1 绘制图形

如果用户希望改变缺省线型（实线）、掩模（无掩模）、背景颜色（黑色）或前境颜色

（白色），必须在调用绘图例程之前调用适当的例程，之后的输出例程的属性不变，直到改变属性或打开一个新的子窗口为止。

表 8.1 是查询当前图形设置、设置新的图形设置和绘制图形的例程列表。

表 8.1　　绘制图形例程

例　程	用　途
ARC, ARC_W	画弧
CLEARSCREEN	清除屏幕，观察口或文本窗口
ELLIPSE, ELLIPSE_W	画圆或椭圆
FLOODFILL, FLOODFILL_W	使用当前填充掩模和颜色索引填充屏幕上的封闭区域
FLOODFILLRGB, FLOODFILLRGB_W	使用当前填充掩模和 RGB 颜色填充屏幕上的封闭区域
GETARCINFO	决定最近画的弧或饼图的终点
GETCURRENTPOSITION, GETCURRENTPOSITION_W	返回当前图形输出位置的坐标
GETPIXEL, GETPIXEL_W	返回像素的颜色索引
GETPIXELRGB, GETPIXELRGB_W	返回像素的 RGB 颜色值
GETPIXELS	获得多个像素的颜色索引
GETPIXELSRGB	获得多个像素的 RGB 颜色值
GRSTATUS	返回最近调用的图形例程的状态（成功或失败）
INTEGERTORGB	把真彩色值转换成红绿蓝成份
LINETO, LINETO_W	从当前图形输出点向指定点连线
MOVETO, MOVETO_W	把当前图形输出位置转移到指定点
PIE, PIE_W	画薄饼形图
POLYGON, POLYGON_W	画多边形
RECTANGLE, RECTANGLE_W	画矩形
RGBTOINTEGER	使用 RGB 函数和例程把红绿蓝三色值转换成真彩色值
SETPIXEL, SETPIXEL_W	把指定位置的像素置为颜色索引
SETPIXELRGB, SETPIXELRGB_W	把指定位置的像素置为 RGB 颜色
SETPIXELS	设置多个像素的颜色索引
SETPIXELSRGB	设置多个像素的 RGB 颜色值

大多数例程都有多种形式。名称以_W 结尾的例程使用窗口坐标系统和 REAL(8)参数类型。没有后缀的例程使用的是观察口坐标和 INTEGER(2)参数类型。

曲线图形，例如弧或椭圆，在一个边界矩形里居中，由矩形左上角和右下角的点确定。矩形的中点就是图形的中点，矩形的边界决定了图形的大小。图 8.10 说明了点(x1, y1) 和(x2, y2)定义的边界矩形：

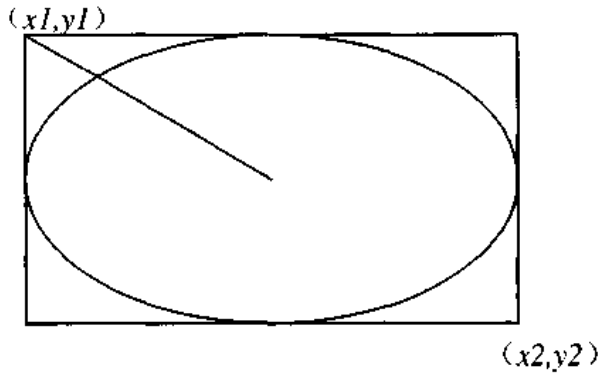


图 8.10 边界矩形

8.5.2 显示基于字符的文本

表 8.2 列出的是查询影响文本显示的屏幕属性的例程，这些例程准备文本屏幕并把文本输到屏幕。

除了这些一般的文本例程外，用户可以用 MODIFYMENUSTRINGQQ 定制菜单中的文本。还可以用 SETMESSAGEQQ 定制其它 QuickWin 过程的字符串，包括状态条信息，状态信息（例如暂停或运行）和对话框信息。

表 8.2 显示文本的例程

例 程	用 途
CLEARSCREEN	清除屏幕，观察口或文本窗口
DISPLAYCURSOR	设置光标的可见或不可见
GETBKCOLOR	返回当前背景颜色索引
GETBKCOLORRGB	返回当前背景 RGB 颜色值
GETTEXTCOLOR	返回当前文本颜色索引
GETTEXTCOLORRGB	返回当前文本 RGB 颜色值
GETTEXTPOSITION	返回当前文本位置
GETTEXTWINDOW	返回当前窗口的边界
OUTTEXT	发送文本到当前位置
SCROLLTEXTWINDOW	滚动文本窗口的内容
SETBKCOLOR	设置当前背景颜色索引
SETBKCOLORRGB	设置当前背景 RGB 颜色值
SETTEXTCOLOR	设置当前文本颜色给新的颜色索引
SETTEXTCOLORRGB	设置当前文本颜色给新的 RGB 颜色值
SETTEXTPOSITION	改变当前文本位置
SETTEXTWINDOW	设置当前文本显示窗口
WRAPON	打开或关闭直线包边

这些例程不提供格式化文本的能力。如果用户想打印整数或浮点数的值，在调用这些例程之前必须先把它们转换成字符串（使用内部 WRITE 语句）。文本例程指定按字符行和列坐标形式的所有屏幕位置。

SETTEXTWINDOW 是 SETVIEWPORT 图形例程的文本等价形式（除了它只限制由 OUTTEXT, PRINT 和 WRITE 输出的文本显示区域外）。GETTEXTWINDOW 返回由 SETTEXTWINDOW 设置的当前文本窗口边界。SCROLLTEXTWINDOW 滚动文本窗口的内容。OUTTEXT, PRINT 和 WRITE 显示写到当前窗口的文本字符串。

注意：WRITE 语句在其后面第一条 I/O 语句的开始处输出回车和换行符（CR 和 LF），如果用户把图形例程 SETTEXTPOSITION 和 OUTTEXT 于 WRITE 语句混合使用，会引起不可预料的文本位置。为了把这种影响降至最低，应该在相关的 FORMAT 语句中使用反斜杠(\)或美元符号编辑描述符来覆盖 CR 和 LF。

8.5.3 显示基于字体的字符

因为 Visual FORTRAN 图形库提供多种字体，用户在显示基于字体的字符时必须说明使用的是哪种字体。选择字体之后，用户可以对这种字体中的字符串打印宽度或字体特征进行查询。表 8.3 列出的是控制显示基于字体的字符的函数：

表 8.3 控制显示基于字体的字符的函数

例 程	用 途
GETFONTINFO	返回当前字体的特征
GETGTEXTEXTENT	确定当前字体中指定文本的宽度
GETGTEXTROTATION	获得用 OUTGTEXT 输出的字体的当前方向
INITIALIZEFONTS	初始化字体库
OUTGTEXT	以当前字体向当前图形输出位置发送文本
SETFONT	寻找一种符合一组指定特性的字体并用 OUTGTEXT 将其设为当前字体
SETGTEXTROTATION	以度为单位设置字体文本输出的方向角

字符可以用两种方法“画”出来：以位图形式（字母的图像）或 TrueType 字符。

8.6 屏幕图像

本节讨论的是以下几种保存和恢复图像的方法：

- 内存缓冲和屏幕之间的图像传输
  在内存缓冲和屏幕之间传输图像是一种快速灵活的屏幕内容传输方式。内存图像可以和当前屏幕图像相互作用：例如，可以对内存图像和屏幕图像实施逻辑与操作，或添加内存图像的负片到屏幕。
- 屏幕和 Windows 位图文件之间的图像传输

从文件中传输图像提供了一种访问其它程序创建的图像的途径，并能保存图像和图形以备它用。从位图文件载入的图像会覆盖它要粘贴位置的内容，并且保持创建图像文件时的属性而不是接受当前属性。

- 屏幕和 QuickWin 编辑菜单剪贴板之间的图像传输

  用 QuickWin 编辑菜单编辑屏幕图像是在屏幕上交互地移动和改变图像的快速而简便的方法，屏幕的属性保持不变，并且为应用程序之间的图像交换提供在剪贴板上的暂时保存。

8.6.1 在内存中传输图像

GETIMAGE 和 **PUTIMAGE** 例程可以在内存和屏幕之间传输图像，并能提供对图像和屏幕的相互作用的控制选项。当用户在内存中保存一幅图像时，应用程序会为这幅图像分配一个内存缓冲区。**IMAGESIZE** 例程就是用来计算存储给定图像所需的内存大小的。名称以_W 结尾的例程使用的是窗口坐标，其它函数使用的是观察口坐标。

相关例程如表 8.4 所示。

表 8.4 内存图像传输例程

例 程	用 途
GETIMAGE, GETIMAGE_W	把屏幕上的图像保存到内存
IMAGESIZE, IMAGESIZE_W	以字节为单位返回图像的大小
PUTIMAGE, PUTIMAGE_W	从内存中恢复图像并显示

8.6.2 载入图像和保存图像到文件

LOADIMAGE 和 **SAVEIMAGE** 例程在屏幕和 Windows 位图文件之间进行图像传输，如表 8.5 所示。

表 8.5 文件图像传输例程

例 程	用 途
LOADIMAGE, LOADIMAGE_W	从磁盘读入一个 Windows 位图文件（.BMP）并按指定坐标显示
SAVEIMAGE, SAVEIMAGE_W	捕获屏幕指定位置的图像并保存为 Windows 位图文件

用户还可以用图形程序创建的 Windows 位图文件作为用 Visual FORTRAN 图形函数和子例程所画的图形的背景图像。

8.6.3 从 QuickWin 编辑菜单编辑文本和图形

在 QuickWin 编辑菜单中可以选 Select Text（选择文本），Select Graphics（选择图形）或 Select All（全选）这几个选项。然后用户可以用鼠标或键盘的方向键选择所需部分。

如果选的是 Select Text 选项，所选部分会变成高亮；如果选择的是 Select Graphics 或 Select All，所选的区域会用方框表示。

选择屏幕上的部分内容后，可以按【DEL】键删除，也可以按【CTRL+INS】组合键或使用编辑菜单中的复制（Copy）选项把所选部分复制到剪贴板。如果选择的部分只是文本，它就会以文本的形式复制到剪贴板；如果选择的部分包含图形或文本和图形都有，它将会以位图的形式复制到剪贴板。

编辑菜单的粘贴（Paste）选项只能粘贴文本。位图可以通过粘贴到其它 Windows 应用程序（例如画笔），组合键为【CTRL+V】或【SHIFT+INS】

选择屏幕的部分内容时应该注意以下几点：

（1）　如果已经从编辑菜单中选择了 Select All 选项，这个屏幕区域都被选择，这时不能再选择屏幕的一部分。

（2）　文本所选择不受用 **SETTEXTWINDOW** 设置的当前文本窗口的限制。

（3）　文本被复制到剪贴板时，每行的拖后空格被去掉。

（4）　已经写入窗口的文本可以被后来输出的图形覆盖，这时文本虽然不可见但仍保存在屏幕文本缓冲内。如果用户选择屏幕的一部分用来复制，那么选择的是不可见但实际存在的文本，并且文本会被复制到剪贴板。

（5）　当用户从编辑菜单选择文本或图形时，应用程序暂停运行，符号(^)出现在当前活动窗口的左上角，所有用户定义的反馈被禁止，并且窗口标题变成"Mark Text – *windownam* "或"Mark Graphics – *windowname* "，其中 *windowname* 是当前活动窗口的名称。

8.7　定制 QuickWin 程序

QuickWin 库是用户可以用来创建 Windows 图形程序或简单应用程序的程序库。

8.7.1　菜单程序控制

用户可以不使用缺省的 QuickWin 菜单，也就是说，用户可以删除和改变菜单、菜单选项列表、菜单标题或菜单项标题。控制菜单的 QuickWin 函数如下：

1．控制开始菜单和窗口框

用 **INITIALSETTINGS** 函数可以改变应用程序开始菜单和窗口框的外观。如果不提供用户定义的 **INITIALSETTINGS** 函数，QuickWin 将调用预先定义的 **INITIALSETTINGS** 例程来控制缺省菜单和窗口框外观。**INITIALSETTINGS** 不需要主动调用，QuickWin 会根据情况自动调用。

如果用户提供，**INITIALSETTINGS** 能够调用 QuickWin 函数来设置初始菜单和窗口框的大小及位置。除此之外，**INITIALSETTINGS** 还可以调用 **SETWSIZEQQ** 在第一次画窗口之前调整窗口框的大小和位置。

下面是关于 **INITIALSETTINGS** 的例子。

```
    LOGICAL(4) FUNCTION INITIALSETTINGS( )
      USE DFLIB
      LOGICAL(4) result
      TYPE (qwinfo) qwi
! 设置窗口框尺寸.
      Qwi.x = 0
      qwi.y = 0
      qwi.w = 400
      qwi.h = 400
      qwi.type = QWIN$SET
      I = SetWSizeQQ( QWIN$FRAMEWINDOW, qwi )
! 创建第一个称为 Games 的菜单.
      Result = APPENDMENUQQ(1, $MENUENABLED, '&Games'C, NUL )
! 添加称为 TicTacToe 的项.
      Result = APPENDMENUQQ(1, $MENUENABLED, '&TicTacToe'C, WINPRINT)
! 画一个分离条
      result = APPENDMENUQQ(1, $MENUSEPARATOR, ''C, NUL )
! 添加称为 Exit 的项
      result = APPENDMENUQQ(1, $MENUENABLED, 'E&xit'C, WINEXIT )
! 添加第二个称为 Help 的菜单
      result = APPENDMENUQQ(2, $MENUENABLED, '&Help'C, NUL )
      result = APPENDMENUQQ(2, $MENUENABLED, '&QuickWin Help'C, WININDEX)
      INITIALSETTINGS= .true.
    END FUNCTION INITIALSETTINGS
```

下面是 **INITIALSETTINGS** 接口的例子。

```
    PROGRAM MENUS
      USE DFLIB
      LOGICAL(4) res
      INTERFACE
        LOGICAL(4) FUNCTION INITIALSETTINGS
        END FUNCTION
      END INTERFACE
    OPEN (10, FILE="User")
    WRITE(10, *) "Hello, child window"
    END
```

创建窗口框之前，在初始化过程中 QuickWin 执行用户的 **INITIALSETTINGS** 函数。当用户函数执行完毕以后，控制返回 QuickWin，由 QuickWin 继续完成剩下的初始化工作。然后控制才交给 Visual FORTRAN 应用程序。

如果成功，用户的 **INITIALSETTINGS** 函数应该返回**.TRUE.**，否则返回**.FALSE.**。缺省的 **INITIALSETTINGS** 函数只返回**.TRUE.**。

值得注意的是，在调用 **INITIALSETTINGS** 函数之后，如果以后不创建自己的菜单则 QuickWin 会创建缺省的菜单。所以，在 **INITIALSETTINGS** 中使用 **DELETEMENUQQ**，**INSERTMENUQQ**，**APPENDMENUQQ** 或其它 QuickWin 菜单配置函数之会影响用户定制的菜单而不是缺省的 QuickWin 菜单。

2．删除、插入和添加菜单项

菜单从左向右定义编号，最左边的编号为 1。每个菜单的菜单项从上到下编号，最上面的编号为 0（菜单标题本身）。如果用户提供，在 **INITIALSETTINGS** 中可以删除、插入和添加菜单项。在 **INITIALSETTINGS** 之外，用户也可以像定制菜单一样在应用程序的任何位置修改缺省的 QuickWin 菜单。

为了删除一个菜单项，需要在 **DELETEMENUQQ** 函数中指定要删除的菜单和菜单项的编号。如果要删除整个菜单，则应该删除这个菜单编号为 0 的菜单项。例如：

```
USE DFLIB
LOGICAL status
! 删除菜单 1（缺省为 FILE 菜单）的第二个菜单项
status = DELETEMENUQQ(1, 2).
! 删除菜单 5（缺省为 Windows 菜单）的全部
status = DELETEMENUQQ(5, 0)
```

INSERTMENUQQ 函数用来插入一个菜单或菜单项并登记相应的反馈例程。QuickWin 提供很多类似 WINEXIT（中止程序运行）、WININDEX（列出 QuickWin 帮助）和 WINCOPY（复制当前窗口的内容到剪贴板）的标准反馈例程。表 8.6 列出的是 **INSERTMENUQQ** 和 **APPENDMENUQQ** 可用的标准例程。

一般来说，用户不能为多个菜单项指定同一个反馈例程，这是因为如果这样的话当改变菜单项的状态时（例如在菜单项旁边作选中标志、使它变灰、使它失效或生效），菜单项的状态可能不能适当地改变。用户不能在已有的编号范围之外插入菜单或菜单项，例如，如果编号 5 和 6 没有定义时，编号为 7 的菜单或菜单项是不能被插入的。如果要插入整个菜单可以指定菜单项 0。新插入的菜单可以在已有菜单之间或紧邻之后的任何位置。

如果用户指定的菜单占据了一个已经存在的菜单，则这个存在的菜单及其右侧的所有菜单都向右移动一个单位，同时这些菜单的编号也相应增加。例如，下面的代码在菜单 5（缺省时为 Windows 菜单）处插入第五个称为 Position 的菜单项：

```
USE DFLIB
LOGICAL(4) status
status = INSERTMENUQQ (5, 5, $MENUCHECKED, 'Position'C, WINPRINT)
```

表 8.6 标准反馈例程

标准例程	功 能
WINPRINT	打印程序
WINSAVE	保存程序
WINEXIT	中止程序运行
WINSELTEXT	从当前窗口选择文本
WINSELGRAPH	从当前窗口选择图形
WINSELALL	从当前窗口选择全部内容
WINCOPY	复制从当前窗口选择的文本或/和图形到剪贴板
WINPASTE	允许用户在 READ 语句中把剪贴板中的文本粘贴到当前活动窗口
WINCLEARPASTE	清除粘贴缓冲
WINSIZETOFIT	调整输出的大小以适合窗口尺寸
WINFULLSCREEN	全屏显示输出
WINSTATE	在暂停和继续文本输出的状态之间切换
WINCASCADE	层叠活动窗口
WINTILE	平铺活动窗口
WINARRANGE	排列图标
WINSTATUS	使状态条失效
WININDEX	显示 QuickWin 标准索引
WINUSING	显示如何使用帮助的信息
WINABOUT	显示当前 QuickWin 程序的“关于（About）”信息
NUL	无反馈例程

下面的代码则在菜单 3 处插入名为 My List 的新菜单。这时右侧的菜单(缺省时为 View, State, Windows 和 Help）都向右移动一个位置：

```
USE DFLIB
LOGICAL(4) status
status = INSERTMENUQQ(3, 0, $MENUENABLED, 'My List'C, WINSTATE)
```

可以用 **APPENDMENUQQ** 添加菜单项。添加的菜单项会加到菜单列表的顶部。如果菜单中还没有任何菜单项，用户添加的菜单项会被认为是顶级菜单项，用户指定的字符串出现在菜单条上。下面的代码在第一个菜单处（缺省的 File 菜单）添加名为 Cascade Windows 的菜单项：

```
USE DFLIB
LOGICAL(4) status
status = APPENDMENUQQ(1, $MENUCHECKED, 'Cascade Windows'C,    &
```

```
& WINCASCADE)
```

上例中的$MENUCHECKED 标志会在菜单旁边作上选中标志，如果要去掉这个标志可以在 **MODIFYMENUFLAGSQQ** 函数中设置标志$MENUUNCHECKED。有些预先定义的例程（例如 WINSTATUS）可以根据情况自动设置选中标志。但是，如果这些例程如果登记为多个菜单项，这些标志可能不能适当更新。

3. 修改菜单项

MODIFYMENUSTRINGQQ 函数可以用来修改菜单项中作为标识符的字符串，**MODIFYMENUROUTINEQQ** 函数可以改变当某个菜单项被选择时调用的反馈例程，**MODIFYMENUFLAGSQQ** 函数可以修改菜单项的状态（例如生效、变灰、选中等）。

下面的代码使用 **MODIFYMENUSTRINGQQ** 函数修改第一个菜单（缺省为 File 菜单）的菜单项 4 的字符串，又使用 **MODIFYMENUROUTINEQQ** 改变选择 WINTILE 时的反馈例程，还使用 **MODIFYMENUFLAGSQQ** 给这个菜单项作选中记号。

```
status = MODIFYMENUSTRINGQQ( 1, 4, 'Tile Windows'C)
status = MODIFYMENUROUTINEQQ( 1, 4, WINTILE)
status = MODIFYMENUFLAGSQQ( 1, 4, $MENUCHECKED)
```

4. 创建可用子窗口的菜单列表

缺省状态下，Windows 菜单包含用户的 QuickWin 应用程序中所有打开的子窗口的列表。**SETWINDOWMENUQQ** 可以改变列出当前打开的子窗口的指定菜单。子窗口名称列表添加在用户选择的菜单下面并删除先前包含它的任何菜单。例如：

```
USE DFLIB
LOGICAL(4) status
...
! 在菜单 1 中添加子窗口列表
status = SETWINDOWMENUQQ(1)
```

5. 模拟菜单选择

CLICKMENUQQ 可以模拟从 Windows 菜单中点击或选择一个菜单命令时的效果。下面的代码段模拟从 Window 菜单中选择 Tile 选项的效果：

```
USE DFLIB
INTEGER(4) status
status = CLICKMENUQQ(QWIN$TILE)
```

8.7.2　改变状态条和状态信息

通过与合适的信息 ID 一起调用 **SETMESSAGEQQ** 可以改变任何字符串 QuickWin 过

程。和其它 QuickWin 信息函数不同的是，**SETMESSAGEQQ** 使用普通字符串，而不是以 null 作为结束标志的 C 字符串。例如，改变 PAUSED 状态为 I am waiting：

```
USE DFLIB
CALL SETMESSAGEQQ('I am waiting', QWIN$MSG_PAUSED)
```

SETMESSAGEQQ 函数对于使用不同的语言进行 QuickWin 程序本地化是很有用的。信息 ID 如表 8.7 所示。

表 8.7 信息 ID 列表

ID	信息
QWIN$MSG_TERM	"Program terminated with exit code"
QWIN$MSG_EXITQ	"\nExit Window?"
QWIN$MSG_FINISHED	"Finished"
QWIN$MSG_PAUSED	"Paused"
QWIN$MSG_RUNNING	"Running"
QWIN$MSG_FILEOPENDLG	"Text Files(*.txt), *.txt; Data Files(*.dat), *.dat; All Files(*.*), *.*;"
QWIN$MSG_BMPSAVEDLG	"Bitmap Files(*.bmp), *.bmp; All Files(*.*), *.*;"
QWIN$MSG_INPUTPEND	"Input pending in"
QWIN$MSG_PASTEINPUTPEND	"Paste input pending"
QWIN$MSG_MOUSEINPUTPEND	"Mouse input pending in"
QWIN$MSG_SELECTTEXT	"Select Text in"
QWIN$MSG_SELECTGRAPHICS	"Select Graphics in"
QWIN$MSG_PRINTABORT	"Error! Printing Aborted."
QWIN$MSG_PRINTLOAD	"Error loading printer driver"
QWIN$MSG_PRINTNODEFAULT	"No Default Printer."
QWIN$MSG_PRINTDRIVER	"No Printer Driver."
QWIN$MSG_PRINTINGERROR	"Print: Printing Error."
QWIN$MSG_PRINTING	"Printing"
QWIN$MSG_PRINTCANCEL	"Cancel"
QWIN$MSG_PRINTINPROGRESS	"Printing in progress..."
QWIN$MSG_HELPNOTAVAIL	"Help Not Available for Menu Item"
QWIN$MSG_TITLETEXT	"Graphic"

8.7.3 显示信息框

MESSAGEBOXQQ 使用户程序显示一个信息框。用户可以指定显示的信息和出现在标题栏上的标题。这些字符串必须都是以 null 作为结束标志的 C 字符串。还可以指定信息框的类型。信息框的类型是在 DFLIB.MOD 中定义的字符常量，还可以用 IOR 内在函数或.OR.操作符来结合使用。使用信息框的例子如下：

```
USE DFLIB
```

```
INTEGER(4) response
response = MESSAGEBOXQQ('Retry or Cancel?'C, 'Smith Chart   &
& Simulator'C, MB$RETRYCANCELQWIN .OR. MB$DEFBUTTON2)
```

可用的信息框类型如表 8.8 所示。

表 8.8　信息框类型

信息代号	相应信息
MB$ABORTRETRYIGNORE	Abort, Retry 和 Ignore 按钮
MB$DEFBUTTON1	第一个按钮为缺省按钮
MB$DEFBUTTON2	第二个按钮为缺省按钮
MB$DEFBUTTON3	第三个按钮为缺省按钮
MB$ICONASTERISK	在蓝色圆圈中的星号图标
MB$ICONEXCLAMATION	感叹号图标
MB$ICONHAND	停止符号图标
MB$ICONINFORMATION	在蓝色圆圈中的小写 i 图标
MB$ICONQUESTION	问号图标
MB$ICONSTOP	停止符号图标
MB$OK	OK 按钮
MB$OKCANCEL	OK 和 Cancel 按钮
MB$RETRYCANCEL	Retry 和 Cancel 按钮
MB$SYSTEMMODAL	挂起所有用户程序直到用户应答
MB$YESNO	Yes 和 No 按钮
MB$YESNOCANCEL	Yes, No 和 Cancel 按钮

8.7.4　定义关于（About）框

ABOUTBOXQQ 函数指定的是在用户从 QuickWin 程序的帮助菜单中选择关于（About）命令时显示的信息。如果用户程序不调用 **ABOUTBOXQQ**，QuickWin 运行库提供缺省字符串，信息字符串必须是以 null 作为结束标志的 C 字符串，例如：

```
USE DFLIB
INTEGER(4) status
status = ABOUTBOXQQ ('Sound Speed Profile Tables Version 1.0'C)
```

8.7.5　使用定制图标

当用户最小化应用程序窗口框或子窗口时 QuickWin 运行库提供缺省缺省图标。用户可以向可执行文件添加定制图标，这时 Windows 显示的是定制的图标而不是缺省图标。为 QuickWin 程序添加定制子窗口的步骤如下：

（1）　在 Developer Studio 中的插入（Insert）菜单中选择资源（Resource），出现如

图 3.10 所示的资源对话框，从中选择图标（Icon），按新建（New）按钮后屏幕上会出现图标绘制工具。中间的图标规格如图 8.11 所示。

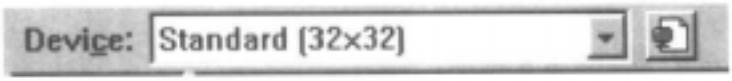


图 8.11 位图规格

按下拉菜单右侧的按钮可以更改图标规格，如图 8.12 所示。

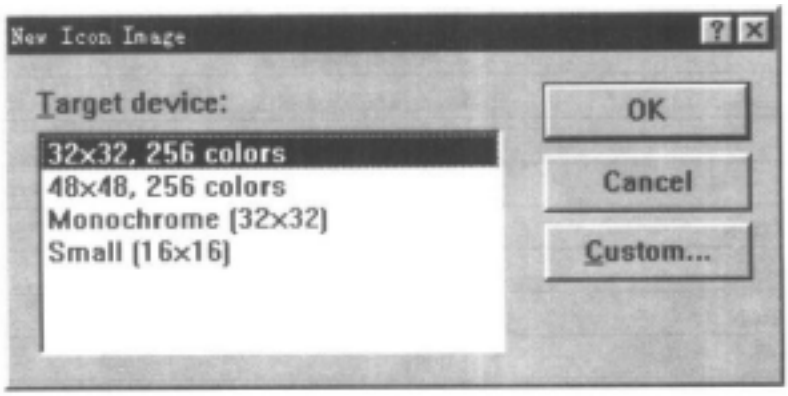


图 8.12 图标规格

（2） 绘制图标。如果图标已经存在，用户希望导入而不是绘制图标，则可以在资源对话框中选择导入（Import）按钮，然后 Visual FORTRAN 会提示包含图标的文件。

（3） 命名图标。窗口框的图标的名称必须是"frameicon"，子窗口图标的名称必须是"childicon"，这些名称必须作为字符串输入到图标属性对话框。

为了显示图标属性对话框，可以在图标编辑器区域内、图标网格之外的区域双击或按【ALT+ENTER】，这时出现如图 8.13 所示的图标属性对话框：

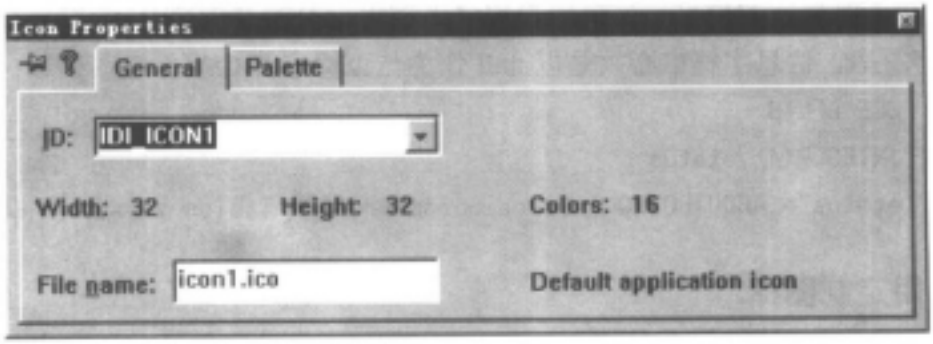


图 8.13 图标属性对话框

在 General 选项卡的 ID 区域中，输入缺省图标名称"frameicon"或"childicon"。输入必须带有引号，这样才能被解释为字符串。用户创建的图标会保存成后缀为.ICO 的文件。

（4）　创建包含图标的脚本（script）文件。选择 File/Save As，用户会被提示包含图标的脚本文件名称。指定脚本文件名，其文件扩展名必须是.RC，例如 myicons.rc。使用这种方法后，图标和字符串值会自动保存为脚本文件。另一种方法是用任意一种编辑器创建脚本文件，然后手工添加图标名称和它们的字符串值。

（5）　向包含用户 QuickWin 程序的项目添加脚本文件。选择建立（Build）后脚本文件将被建立可供执行。编译后的脚本文件的扩展名为.RES。

运行用户自己的应用程序时，用户创建的图标将取代缺省的子图标或框图标，它出现在窗口框的左上角，最小化窗口时图标会出现在最小化窗口条上。

8.8　使用鼠标

用户程序可以检测和应答鼠标事件，例如鼠标左键单击，右键单击或双击。鼠标事件可以代替部分键盘功能或操作屏幕显示的内容。

QuickWin 提供两类鼠标函数：

- 基于事件的函数

  当鼠标点击事件发生后调用程序定义的反馈例程。

- 顺序函数

  它提供中止程序的一系列函数，直到有鼠标输入。

鼠标是异步设备，所以用户可以在程序运行的任何时候点击鼠标（鼠标输入不需任何同步要求）。当鼠标点击发生时，Windows 发送一个信息给应用程序，这个信息使程序产生相应动作。应用程序支持的鼠标多数是基于事件的，也就是说，鼠标点击后程序会进行某些事情。

一个程序还可以使用顺序函数来对待鼠标点击的发生。这样可以允许一个程序按特定的顺序执行并提供鼠标支持。

QuickWin 执行的默认处理是基于鼠标事件的。

8.8.1　基于事件的函数

QuickWin 函数 REGISTERMOUSEEVENT 登记特定的鼠标事件发生后被调用的列出。用户可以定义要处理的事件和相应的处理例程。UNREGISTERMOUSEEVENT 取消登记的例程，特定的鼠标事件发生后 QuickWin 调用的是缺省处理。

缺省时，QuickWin 忽略除了发生在菜单或对话框控制中的点击鼠标事件。注意一个最小化的窗口是不能接受任何事件的。所以，窗口必须处在还原或最大化状态才能接受窗口里发生的鼠标事件。

下面是使用 REGISTERMOUSEEVENT 的例子：

```
USE DFLIB
```

```
INTEGER(4) result
OPEN (4, FILE='USER')
...
result = REGISTERMOUSEEVENT (4, MOUSE$LBUTTONDBLCLK, CALCULATE)
```

上例登记了例程 CALCULATE。当用户双击鼠标左键并且鼠标光标处在以单元 4 打开的子窗口内时就调用 CALCULATE。标识鼠标事件的符号常量如表 8.9 所示。

表 8.9 鼠标事件的符号常量

鼠标事件	描 述
MOUSE$LBUTTONDOWN	按下鼠标左键
MOUSE$LBUTTONUP	鼠标左键恢复
MOUSE$LBUTTONDBLCLK	双击鼠标左键
MOUSE$RBUTTONDOWN	按下鼠标右键
MOUSE$RBUTTONUP	鼠标右键恢复
MOUSE$RBUTTONDBLCLK	鼠标右键双击
MOUSE$MOVE	鼠标移动

对于每个“按键”和“双击”事件都有“键恢复”事件和它联系。当用户双击时会产生 4 个事件：第一次点击的“按键”和“键恢复”事件，以及第二次点击的“双击”和“键恢复”事件。“双击”和“按键”之间的区别在于第二次点击是否在双击间隔时间之内。双击间隔时间由系统的“控制面板”中的“鼠标”来设置。

取消上例中的登记例程可以用下面的代码：

```
result = UNREGISTERMOUSEEVENT (4, MOUSE$LBUTTONDBLCLK)
```

如果没有取消前一个登记例程而再次调用 **REGISTERMOUSEEVENT**，将会覆盖第一次调用。相应时间发生时会调用新的反馈例程。

鼠标事件发生时用户创建的反馈例程应该有下面的原型：

```
INTERFACE
  SUBROUTINE MouseCallBackRoutine (unit, mouseevent, keystate,   &
   & MouseXpos,MouseYpos)
     INTEGER unit
     INTEGER mouseevent
     INTEGER keystate
     INTEGER MouseXpos
     INTEGER MouseYpos
  END SUBROUTINE
END INTERFACE
```

参数 unit 是和事件被检测到的子窗口相联系的单元号，参数 mouseevent 是表 8.9 所列出的事件中的一个。参数 MouseXpos 和 MouseYpos 指定鼠标在事件中的位置。参数 keystate 说明发生鼠标事件时【Shift】和【Ctrl】键的状态，并且可以和表 8.10 中的常量作“或”结合：

表 8.10　按键状态参数

键状态参数	描　述
MOUSE$KS_LBUTTON	在事件中按下鼠标左键
MOUSE$KS_RBUTTON	在事件中按下鼠标右键
MOUSE$KS_SHIFT	在事件中保持按下【Shift】键
MOUSE$KS_CONTROL	在事件中保持按下【Ctrl】键

QuickWin 的鼠标事件反馈例程应该作尽量少的处理并返回。处理反馈时，程序显得像没有反应，因为这时不接受任何信息。所以快速返回是非常重要的。如果反馈需要的处理事件比较长，应该开始另外一条线程来处理反馈。调用 Win32 API **CreateThread** 可以创建线程。详细的内容见“高级主题”中的“创建多线程”一章。

注意：基于事件的函数中，事件没有缓冲。所以，会碰到诸如多线程和同时访问共享资源等问题。为了避免多线程问题可以使用顺序函数。顺序函数在进行顺序处理的应用程序中表现很好。而只有少量顺序流而大部分由用户决定在程序中跳转的应用程序或许应该最好由基于事件的函数来实现。

8.8.2　顺序函数

QuickWin 顺序函数 WAITONMOUSEEVENT 会锁定程序执行直到接受到特定的鼠标事件发生。这个函数和 INCHARQQ 函数类似，区别在于一个是等待鼠标事件一个是等待键盘按键。

例如：

```
      USE DFLIB
      INTEGER(4) mouseevent, keystate, x, y, result
      ...
      mouseevent = MOUSE$RBUTTONDOWN .OR. MOUSE$LBUTTONDOWN
      result = WAITONMOUSEEVENT (mouseevent, keystate, x , y) ! Wait
  ! until right or left mouse button clicked, then check the keystate
  ! with the following:
      if ((MOUSE$KS_SHIFT .AND. keystate) == MOUSE$KS_SHIFT) then      &
     & write (*,*) 'Shift key was down'
      if ((MOUSE$KS_CONTROL .AND. keystate) == MOUSE$KS_CONTROL) then  &
     & write (*,*) 'Ctrl key was down'
```

用户程序把一个鼠标事件参数传递给 **WAITONMOUSEEVENT**，该参数可以和鼠标事件作“或”结合。函数就等待并锁定程序执行，直到指定事件之一发生。事件发生后以参数 keystate 返回【Shift】键和【Ctrl】键的状态，还以参数 x 和 y 返回事件发生时鼠标的位置。

最初 **WAITONMOUSEEVENT** 被调用时鼠标事件必须发生在有焦点的窗口中，其它窗口发生的事件不会结束等待。

8.8.3 缺省的 QuickWin 处理

QuickWin 会基于鼠标事件进行某些动作。它使用鼠标事件从全屏模式返回到选择要复制到剪贴板的文本或/和图像。鼠标事件服务函数优先于从全屏模式返回（【ALT+ENTER】组合键可以用来从全屏模式返回）。而剪切/粘贴选择模式又优先于鼠标事件服务程序。一旦选择模式结束，鼠标事件函数会继续进行。

8.9 增强 QuickWin 应用程序

除了基本的 QuickWin 特性之外，用户可以使用表 8.11 描述的特性定制和增强自己的 QuickWin 应用程序。使用这些特性可以创建定制菜单、鼠标响应事件和添加定制图标。

表 8.11　　增强 QuickWin 应用程序

种类	QuickWin 函数	描述
初始化设置	INITIALSETTINGS	控制初始化菜单设置和窗口框
显示/添加信息框	MESSAGEBOXQQ	显示信息框
	ABOUTBOXQQ	以定制的文本添加 About 信息框
菜单项	CLICKMENUQQ	模拟点击或选择一个菜单项的效果
	APPENDMENUQQ	添加一个菜单项
	DELETEMENUQQ	删除一个菜单项
	INSERTMENUQQ	插一个菜单项
	MODIFYMENUFLAGSQQ	修改菜单项状态
	MODIFYMENUROUTINEQQ	修改菜单项反馈例程
	MODIFYMENUSTRINGQQ	改变菜单项的文本串
	SETWINDOWMENUQQ	设置附加当前子窗口列表的菜单
方向键	PASSDIRKEYSQQ	使作为输入的方向键和翻页键生效或失效
QuickWin 信息	SETMESSAGEQQ	改变 QuickWin 信息，包括状态条信息、状态信息和对话框信息
鼠标动作	REGISTERMOUSEEVENT	向应用程序登记定义为鼠标事件的例程
	UNREGISTERMOUSEEVENT	删除由 REGISTERMOUSEEVENT 登记的例程
	WAITONMOUSEEVENT	直到一个鼠标事件发生才返回块

第九章　图形和字体

Visual FORTRAN 提供的图形例程可以设置点，画直线和文本，改变颜色以及画圆形、矩形、弧等图形。因为 QuickWin 是绘制图形的基础，所以用户在在阅读本章之前应先阅读上一章“使用 QuickWin”。

Visual FORTRAN 图形库还包含以各种字体风格和大小打印文本的例程。这些例程提供对用户文本外观的控制，还能美化屏幕的显示。使用字体要用到图形和 QuickWin 的一些知识。

本章使用下列约定和术语：

- 原点（0,0）在屏幕或子窗口写入的用户区域的左上角。x 轴和 y 轴开始于原点。某些坐标系下用户可以更改原点位置。
- 水平方向用 x 轴代表，向右为正方向。
- 数值方向用 y 轴代表，向下为正方向。
- 有些图形适配器提供可更改的调色板。
- 有些图形适配器（VGA 和 SVGA）允许用户提供描述一种新颜色的值来改变颜色索引指向的颜色。颜色值指的是红绿蓝（RGB）三色的混合。颜色值始终用类型为 INTEGER(4)的数表示。

9.1　使用图形模式

显示图形之前先要用 SETWINDOWCONFIG 设置希望的图形模式，然后才能调用相应的例程创建图形。

9.1.1　检测当前图形模式

获得子窗口的设置可以调用 GETWINDOWCONFIG 函数。在\DF\INCLUDE 子目录下的 DFLIB.F90 模块定义了一种派生类型 windowconfig，GETWINDOWCONFIG 使用它作为参数：

```
TYPE windowconfig
  INTEGER(2) numxpixels      ! Number of pixels on x-axis
  INTEGER(2) numypixels      ! Number of pixels on y-axis
  INTEGER(2) numtextcols     ! Number of text columns available
  INTEGER(2) numtextrows     ! Number of text rows available
```

```
    INTEGER(2) numcolors        ! Number of color indexes
    INTEGER(4) fontsize         ! Size of default font
    CHARACTER(80) title         ! window title
    INTEGER(2) bitsperpixel     ! Number of bits per pixel
END TYPE windowconfig
```

缺省时，QuickWin 子窗口是 640×480 像素的可滚动文本窗口，30 行 89 列，字体大小为 8×16。标准图形窗口缺省时是全屏幕。

9.1.2 设置图形模式

可以使用 SETWINDOWCONFIG 来设置需要的窗口属性。如果给 windowconfig 派生类型中的 *numxpixels*, *numypixels*, *numtextcols* 和 *numtextrows* 的值赋成-1，就设置了用户显示卡允许的最大分辨率。这会使标准图形程序以全屏模式开始。

如果用户指定的比最大图形区域小，应用程序以一个窗口开始。用户可以按【ALT+ENTER】键在全屏和窗口之间切换。如果用户程序为 QuickWin 程序且没有调用 SETWINDOWCONFIG，子窗口缺省是 640×480 像素的可滚动文本窗口，30 行 89 列，字体大小为 8×16。颜色的数目取决于用户使用的显示卡。

如果 SETWINDOWCONFIG 返回的是.**FALSE.**，说明显卡不支持指定的选项。函数会调整 windowconfig 派生类型中的值，使这些值尽可能接近配置要求又可以工作。然后可以再次调用 SETWINDOWCONFIG 调整后的值，例如：

```
LOGICAL statusmode
TYPE (windowconfig) wc
wc.numxpixels  = 1000
wc.numypixels  = 300
wc.numtextcols = -1
wc.numtextrows = -1
wc.numcolors   = -1
wc.title = "Opening Title"C
wc.fontsize = #000A000C ! 10 X 12
statusmode = SETWINDOWCONFIG(wc)
IF (.NOT. statusmode) THEN statusmode = SETWINDOWCONFIG(wc)
```

如果使用 SETWINDOWCONFIG，就必须为每个区指定值（-1 或用户自己指定的数据，题目用 C 字符串）。如果只用 SETWINDOWCONFIG 指定部分值将导致其它值无效。

9.1.3 编写图形程序

像很多其它程序一样，由一些较小的单元构成的图形程序会工作得好一些。使用多个

离散的例程可以通过隔离程序的函数成员来帮助调试程序。下面的例程序及相应的子例程说明了包括初始化、绘制和关闭一个图形程序的各个步骤。

SINE 程序是 Visual FORTRAN 自带的例子，在\DF\SAMPLES\TUTORIAL 子目录下。SINE 程序运行结果是画一条正弦取消，它调用了很多图形程序例程。下面的主程序调用了实际上实施作图命令的 5 个子例程：

```
!  SINE.F90 - Illustrates basic graphics commands.
!
     USE DFLIB
     CALL graphicsmode( )
     CALL drawlines( )
     CALL sinewave( )
     CALL drawshapes( )
     END
```

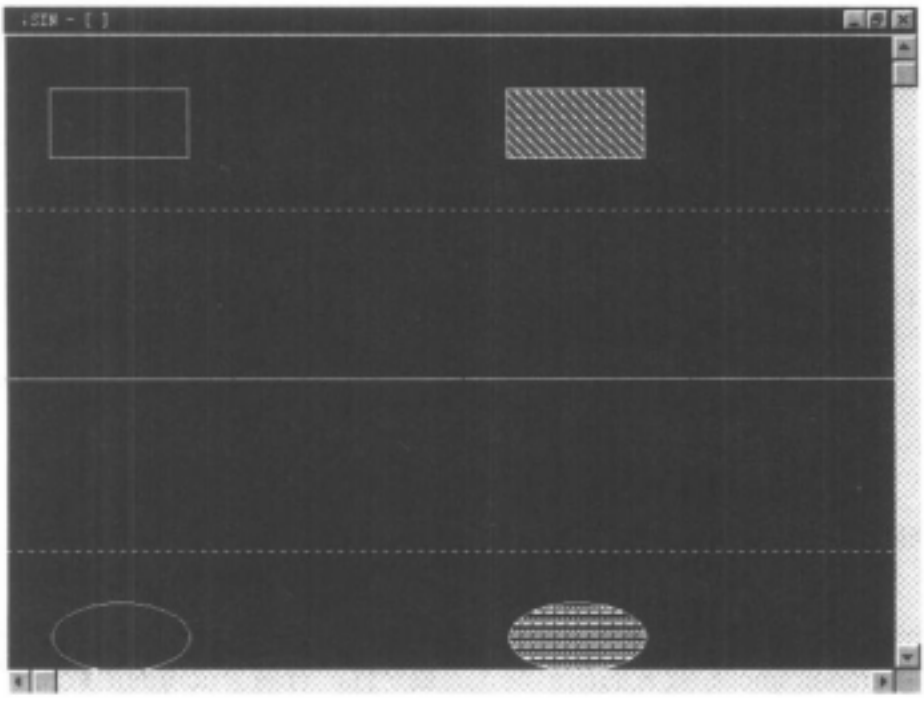


图 9.1　SIN 程序输出结果

下面将仔细分析这个程序。

1. 激活图形模式

如果用户调用图形例程时没有 SETWINDOWCONFIG 用设置图形模式，QuickWin 会自动以缺省值设置图形模式。

SINE 在子例程 graphicsmode 中选择并设置了图形模式，所选的模式是当前显卡提供的最高分辨率：

```
SUBROUTINE graphicsmode( )
   USE DFLIB
   LOGICAL              modestatus
   INTEGER(2)           maxx, maxy
   TYPE (windowconfig) myscreen
   COMMON               maxx, maxy

!  Set highest resolution graphics mode.

   myscreen.numxpixels=-1
   myscreen.numypixels=-1
   myscreen.numtextcols=-1
   myscreen.numtextrows=-1
   myscreen.numcolors=-1
   myscreen.fontsize=-1
   myscreen.title = " "C ! blank

   modestatus=SETWINDOWCONFIG(myscreen)

!  Determine the maximum dimensions.

   modestatus=GETWINDOWCONFIG(myscreen)
   maxx=myscreen.numxpixels - 1
   maxy=myscreen.numypixels - 1
END
```

像素坐标开始于 0，如果分辨率为 640×480 则水平方向像素的最大坐标为 639。

为了保持 graphicsmode 设置的图形模式的独立性，有两个很短的函数用来把 1000×1000 像素的任意屏幕尺寸转换为实际的显示模式。此后，程序就认为每个方向都是 1000 像素了。为了在屏幕上画点，newx 和 newy 映射每一点到它们的实际物理坐标：

```
! NEWX - This function finds new x-coordinates.

   INTEGER(2) FUNCTION newx( xcoord )

   INTEGER(2) xcoord, maxx, maxy
```

```
    REAL(4) tempx
    COMMON maxx, maxy

    tempx = maxx / 1000.0
    tempx = xcoord * tempx + 0.5
    newx = tempx
    END

! NEWY - This function finds new y-coordinates.
!
    INTEGER(2) FUNCTION newy( ycoord )

    INTEGER(2) ycoord, maxx, maxy
    REAL(4) tempy
    COMMON maxx, maxy

    tempy = maxy / 1000.0
    tempy = ycoord * tempy + 0.5
    newy = tempy
    END
```

用户还可以用窗口坐标建立类似的独立坐标系统。

2. 在屏幕上画直线

SINE 程序然后调用子例程 drawlines，这个子例程画了屏幕的边框和三条直线，如图 9.1 所示。程序如下：

```
!   DRAWLINES - This subroutine draws a box and
!   several lines.

    SUBROUTINE drawlines( )

    USE DFLIB

    EXTERNAL        newx, newy
    INTEGER(2)      status, newx, newy, maxx, maxy
    TYPE (xycoord)  xy
    COMMON          maxx, maxy
!
!   Draw the box.
```

```
    status = RECTANGLE( $GBORDER, INT2(0), INT2(0), maxx, maxy )
    CALL SETVIEWORG( INT2(0), newy( INT2( 500 ) ), xy )     ! This sets
!           the new origin to 0 for x and 500 for y. See comment after subroutine.

!   Draw the lines.

    CALL MOVETO( INT2(0), INT2(0), xy )
    status = LINETO( newx( INT2( 1000 )), INT2(0))
    CALL SETLINESTYLE( INT2( #AA3C ))
    CALL MOVETO( INT2(0), newy( INT2( -250 )), xy )
    status = LINETO(newx( INT2( 1000 )),newy( INT2( -250 )))
    CALL SETLINESTYLE( INT2( #8888 ))
    CALL MOVETO(INT2(0), newy( INT2( 250 )), xy )
    status = LINETO( newx( INT2( 1000 )),newy( INT2( 250 ) ) )
    END
```

RECTANGLE 的第一个参数是填充标志，可以是$GBORDER 或$GFILLINTERIOR。如果只希望用当前线型画矩形的边界可以选$GBORDER；如果希望得到一个用当前颜色和样式填充的实心矩形可以选择$GFILLINTERIOR。颜色和填充样式会在后面的“添加颜色”和“添加形状”中详细讨论。

RECTANGLE 的第二个和第三个参数是矩形左上角的 x 和 y 坐标；第四个和第五个参数是矩形右下角的 x 和 y 坐标。因为上面的 RECTANGLE 的两个点的参数分别为(0,0)和 (maxx, maxy)，所以是给整个屏幕画了边框。

程序调用 SETVIEWORG 来关闭观察口原点的位置。通过在 1000×1000 的观察口中设置原点为(0, 500)，用户实际上是把观察口设为从左上点(0, -500)到右下点(1000, 500)：

```
CALL SETVIEWORG( INT2(0), newy( INT2( 500 ) ), xy )
```

调用 SETLINESTYLE 把线型从实线该为虚线。一串 16 位长的参数告知例程线的样式：某位为“1”代表一个实像素，某位为“0”代表空像素。所以 1111 1111 1111 1111 代表的是实线，而 1111 1111 0000 0000 代表的是较长的虚线，1111 0000 1111 0000 代表的是较短的虚线。任何进制的 INTEGER(2)数都可以作为可接受的输入，但二进制和十六进制对线型来说更为直观。例如，十六进制常量#AA3C 等价于 1010 1010 0011 1100。

画线时应该先设置线型，然后移到线的起点，再调用画线函数 LINETO，使直线由起点连到终点。drawlines 子例程使用了下面的代码：

```
CALL SETLINESTYLE(INT2( #AA3C ) )
```

```
CALL MOVETO( INT2(0), newy( INT2( -250 ) ), xy )
dummy = LINETO( newx( INT2( 1000 )), newy( INT2( -250 )))
```

MOVETO 是把一个形象的像素光标移到屏幕上的一点，实际上没有任何显示；LINETO 则是实际画一条直线。当程序调用 SETVIEWORG 时，它改变了观察口原点，初始的 y 轴范围从 0 到 1000 该为-500 到+500。于是，负值-250 用来画通过上半屏幕中点的 y 坐标，正值 250 用来画通过下半屏幕中点的 y 坐标。

3. 画正弦曲线

边框和坐标轴画完之后，SINE 程序准备开始画正弦曲线。Sinewave 例程计算两个周期的 x 和 y 坐标并把它们绘成曲线：

```
!  SINEWAVE - This subroutine calculates and plots a sine
!             wave.
!
     SUBROUTINE sinewave( )
     USE DFLIB

     INTEGER(2)     dummy, newx, newy, locx, locy, i
     INTEGER(4)     color
     REAL           rad
     EXTERNAL       newx, newy

     PARAMETER     ( PI = 3.14159 )
!
!    Calculate each position and display it on the screen.
     color = #0000FF ! red
!
     DO i = 0, 999, 3
       rad   = -SIN( PI * i / 250.0 )
       locx  = newx( i )
       locy  = newy( INT2( rad * 250.0 ) )
       dummy = SETPIXELRGB( locx, locy, color )
     END DO
     END
```

SETPIXELRGB 有 locx 和 locy 两个位置参数，它把像素设置到该位置并指定颜色值（红）。

4. 添加形状

曲线画完之后，SINE 程序调用 drawshapes 向屏幕添加两个矩形和两个椭圆。填充标

志在$GBORDER 和$GFILLINTERIOR 之间改变：

```
!  DRAWSHAPES - Draws two boxes and two ellipses.
!
       SUBROUTINE drawshapes( )

       USE DFLIB

       EXTERNAL     newx, newy
       INTEGER(2)   dummy, newx, newy
!
!      Create a masking (fill) pattern.
!
       INTEGER(1) diagmask(8), horzmask(8)
       DATA diagmask / #93, #C9, #64, #B2, #59, #2C, #96, #4B /
       DATA horzmask / #FF, #00, #7F, #FE, #00, #00, #00, #CC /
!
!      Draw the rectangles.
!
       CALL SETLINESTYLE( INT2(#FFFF ))
       CALL SETFILLMASK( diagmask )
       dummy = RECTANGLE( $GBORDER, newx(INT2(50)), newy(INT2(-325)), &
      & newx(INT2(200)), newy(INT2(-425)))
       dummy = RECTANGLE( $GFILLINTERIOR, newx(INT2(550)), &
      & newy(INT2(-325)), newx(INT2(700)), newy(INT2(-425)))
!
!      Draw the ellipses.
!
       CALL SETFILLMASK( horzmask )
       dummy = ELLIPSE( $GBORDER, newx(INT2(50)), newy(INT2(325)), &
      & newx(INT2(200)), newy(INT2(425)))
       dummy = ELLIPSE( $GFILLINTERIOR, newx(INT2(550)), &
      & znewy(INT2(325)), newx(INT2(700)), newy(INT2(425)))
       END
```

SETLINESTYLE 的调用重新设置线型为实线，忽略调用 SETLINESTYLE 使第一个矩形以虚线边界出现，因为事先调用的 drawlines 例程把线型设成了虚线。

ELLIPSE 使用类似 RECTANGLE 的参数来画椭圆。并且也需要填充标志和边界矩形的两个角点。边界矩形参见图 8.10。

$GFILLINTERIOR 常量按当前样式填充图形。与创建线型类似，要创建填充样式时，某位为“1”代表一个实像素，某位为“0”代表空像素。表示填充样式的矩阵存在 8 字节数组中，其地址要传递给 **SETFILLMASK**。在 drawshapes 例程中，diagmask 数组由表 9.1 所示的样式初始化：

表 9.1　　填充样式

填充样式	diagmask 中的值
位编号.76543210	
xooxooxx	diagmask(1) = #93
xxooxoox	diagmask(2) = #C9
oxxooxoo	diagmask(3) = #64
xoxxooxo	diagmask(4) = #B2
oxoxxoox	diagmask(5) = #59
ooxoxxoo	diagmask(6) = #2C
xooxoxxo	diagmask(7) = #96
oxooxoxx	diagmask(8) = #4B

至此，SINE 程序基本结束。

9.2　添加颜色

Visual FORTRAN 的 QuickWin 库支持彩色图形。可用颜色的总数由用户的显示卡和当前显示驱动决定；用户使用的颜色数由调用的函数决定。

9.2.1　颜色混合

如果用户使用的是 VGA 显示器，一次最多显示 256 色，这 256 色保存在调色板中。用户可以在调色板中选择 262,144 色 (256K)，但一次只能显示 256 色。SVGA 和真彩色显示适配器可以分别显示 262,144 (256K)种和 256×256×256=16.7M 种颜色。

如果使用调色板，所能使用的颜色由调色板能够获得的颜色决定。如果希望同时显示多于 256 颜色，必须显式指定 RGB 颜色值而不是调色板索引。

当用户选择了一个颜色索引时，就指定了系统预先定义的调色板中的一种颜色。SETCOLOR, SETBKCOLOR 和 SETTEXTCOLOR 分别设置当前颜色，背景颜色和文本颜色。SETCOLORRGB, SETBKCOLORRGB 和 SETTEXTCOLORRGB 设置的颜色则从整个可用颜色范围内选择。当用户选择一个颜色值时，则为红绿蓝三种颜色各指定了一个范围为 0～255 的颜色深度。定义一个颜色值的长整型数由 3 字节组成：

MSB　　LSB

```
BBBBBBBB GGGGGGGG RRRRRRRR
```

其中 R，G 和 B 分别代表红、绿、蓝的颜色深度的位。例如要混合粉红色：最深的红色与适度的绿色和蓝色混合，即：

```
10000000 10000000 11111111
```

在十六进制中，上面的颜色值等于#8080FF。可以使用函数：

```
i = SETCOLORRGB (#8080FF)
```

来设置当前颜色为上面定义的粉红。

上面函数也可以用十进制数值为参数。注意 1 代表很浅的颜色深度，而 255 是饱和的颜色深度。“创建”黄色（饱和的红色与饱和的绿色混合）：

```
i = SETCOLORRGB( #00FFFF )
```

白色则是所有颜色的饱和混合：

```
i = SETCOLORRGB( #FFFFFF)
```

所有的颜色位为 0 是黑色：

```
i = SETCOLORRGB( #000000)
```

常见颜色的 RGB 值如表 9.2 所示。

表 9.2　　常见颜色的 RGB 值

颜　色	RGB 值	颜　色	RGB 值
黑	#000000	纯白	#FFFFFF
暗红	#000080	红	#0000FF
暗绿	#008000	绿	#00FF00
土黄	#008080	黄	#00FFFF
深蓝	#800000	栏	#FF0000
暗洋红	#800080	洋	#FF00FF
墨绿	#808000	青绿	#FFFF00
深灰	#808080	浅灰	#C0C0C0

如果用户的机器是 64K 颜色且用户设置了不等于预先设置的 64K 种 RGB 值的 RGB 颜色值，系统则以最接近所要求 RGB 值的可用 RGB 值来近似。如果用户使用的是 VGA

机器，而设置的颜色不在调色板内，系统也会以可用的颜色来近似。

但是，在上述情况下，虽然颜色是近似的，如果用 GETCOLORRGB, GETBKCOLORRGB 或 GETTEXTCOLORRGB 获得颜色时，返回的是用户指定的颜色而不是近似后的颜色。这是因为 **SETCOLORRGB** 函数并不执行绘图，它只设置颜色，颜色的近似是由实际绘制过程中完成的。GETPIXELRGB 和 GETPIXELSRGB 可以返回实际使用的近似颜色，因为是 SETPIXELRGB 和 SETPIXELSRGB 来完成实际在屏幕上设置像素颜色和颜色近似的。

9.2.2　VGA 颜色面板

VGA 机器同时最多可以显示 256 种颜色。在 QuickWin 中，VGA 可以在 320×200 的分辨率下显示 256 种颜色，但在更高的分辨率下只能显示 2 或 16 色。VGA 调色板中可选择的颜色数取决于用户程序，颜色数在 windowconfig 派生类型中的 *wc.numcolors* 变量设置（由 SETWINDOWCONFIG 可以设置成 2，16 或 256）。

RGB 颜色值必须在 VGA 可显示的调色板中。可以使用 REMAPPALETTERGB 改变缺省颜色和定制用户自己的调色板。下面的例子重新映射了颜色索引 1（缺省为蓝色），使它指向红色（RGB 值为#0000FF）：

```
USE DFLIB
INTEGER(4) status
status = REMAPPALETTERGB( 1, #0000FF )     ! 重新指定索引 1 为红色
```

REMAPPALETTERGB 可以重新映射一种或同时映射多种颜色索引。其参数是映射到调色板的 RGB 值数组。数组中的第一个元素成为和索引为 0 的颜色联系，第二个元素和索引 1 联系，以次类推。但是最多只能映射 236 个索引，因为有 20 个索引是系统保留的。

重新映射调色板对 64K 色、SVGA 或真彩色机器无效，除非通过使用例如 SETCOLOR 的索引函数限制到调色板。在 VGA 机器中，如果把调色板中的所有颜色都重新映射并在图形中显示了该调色板，就不能在重新映射并同时显示第二个调色板。

例如，在 VGA256 色模式下，如果重新映射了 256 中调色板颜色并在子窗口中显示图形，并另外打开一个子窗口，而且在第二个子窗口重新映射调色板并显示图形，这时相当于同时显示多于 256 种颜色，这是 VGA 机器力所不能及的。所以只有具有焦点的窗口才会正确显示颜色，而没有焦点的窗口颜色会改变。

Windows 9x 以及 Windows NT 可以创建映射到显示硬件调色板的逻辑调色板。在支持 256 色以下的显示硬件上，重新映射调色板会覆盖当前调色板并以新的颜色重新绘制屏幕。通过重新映射改变屏幕的技术称为调色板动画。但是在支持多于 256 色的机器上不能以重新映射调色板来实现动画，这是因为在支持多于 256 色的大型硬件调色板上，重新映射会作用在调色板未使用部分。重新映射并不覆盖当前颜色，也不重新绘制屏幕。

缺省颜色编号的符号常量（名称）由图形模块提供。名称是表义的，例如代表黑色、黄色和红色的符号常量分别为$BLACK, $YELLOW 和$RED。

所有的 VGA 显示模式都支持 VGA（模拟）显示器，在兼容的模拟单色显示器中颜色则以灰度表示。

9.2.3 使用文本颜色

SETTEXTCOLORRGB（SETTEXTCOLOR）和 SETBKCOLORRGB（SETBKCOLOR）设置文本输出的前景和背景颜色。它们都只使用一个单独的参数来指定由 OUTTEXT 和 WRITE 显示的文本的颜色值（或颜色索引）。对于颜色索引函数，颜色由范围 0～31 决定，其中 16～31 范围的索引值和 0～15 所对应的颜色相同。

9.3 坐标系

Visual FORTRAN 的 QuickWin 库支持几种不同的坐标系：文本坐标系是行和列的形式；物理坐标系则为创建定制窗口和观察口坐标提供绝对位置的参考和开始点。坐标变换函数使两种不同的坐标系之间的变换也不困难。

本节将讨论下列问题：

5. 文本坐标系
6. 图形坐标系
7. 一个实坐标系的例程序

9.3.1 文本坐标系

文本模式使用的坐标系把屏幕分为行和列，如图 9.2 所示。

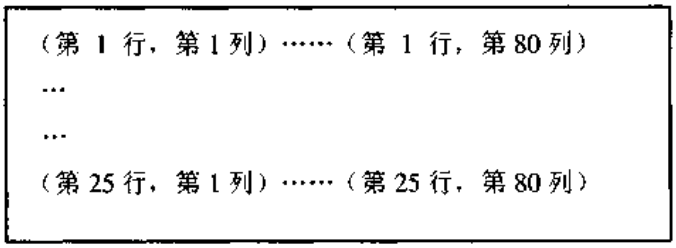


图 9.2 文本坐标系

文本坐标系使用下面的约定：

- 从 1 开始编号。一个 80 列的屏幕包含第 1 列到第 80 列。
- 行始终列在列前。

如果屏幕显示 25 行 80 列（如图 9.2 所示），行编号为 1～25，列编号为 1～80。文本定位例程，例如 SETTEXTPOSITION 和 SCROLLTEXTWINDOW 就是使用行、列坐标。

9.3.2　图形坐标

描述屏幕上的像素有三种坐标：物理坐标，观察口坐标和窗口坐标。在这三种坐标中，x 坐标都列在 y 坐标前面。

1．物理坐标（Physical Coordinates）

物理坐标是表示一个窗口用户区域中像素的整数。缺省时编号以 0 而不是 1 开始。如有 640 个像素，那么它们的编号就是 0～639。

假设用户的程序调用了 SETWINDOWCONFIG 设置一个包含 640 个水平像素和 480 个垂直像素的用户区域，每个像素的位置都用对应的 x 坐标和 y 坐标来表示，如图 9.3 所示。图中，左上角是原点，原点的 x 坐标和 y 坐标始终是(0, 0)。

物理坐标中直接以整数代表像素，如果用变量表示像素位置应该声明为整数或在向图形函数传递它们时使用类型转换例程，例如：

```
ISTATUS = LINETO( INT2(REAL_x), INT2(REAL_y))
```

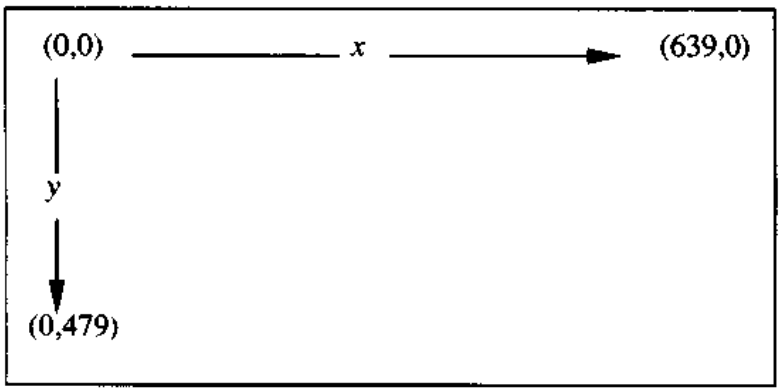


图 9.3　物理坐标系

如果程序使用缺省的窗口尺寸，观察口（绘图区）等于 640×480。SETVIEWORG 可以改变观察口原点：传入两个分别代表原点 x 坐标和 y 坐标的两个整数即可。还可以传入 xycoord 类型，则 SETVIEWORG 例程充满原来原点的物理坐标。例如，下面的例子将观察口原点移到物理位置(50, 100)：

```
TYPE (xycoord) origin
CALL SETVIEWORG(INT2(50), INT2(100), origin)
```

上面代码执行后的原点如图 9.4 所示。

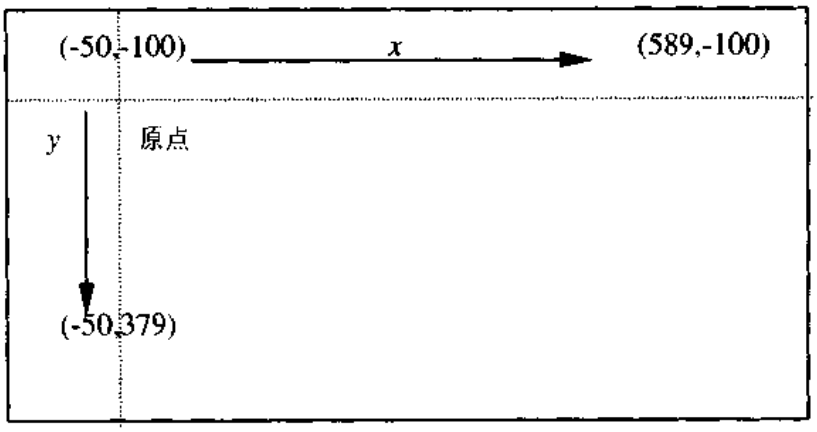


图 9.4 改变原点坐标

像素的数目并没有改变，但是代表像素的坐标改变了：x 坐标的范围从 0～639 变成 -50～589，y 坐标的范围从 0～480 变成-100～379。

所有使用观察口坐标的图形例程都会受到新原点的影响，包括：**MOVETO**, **LINETO**, **RECTANGLE**, **ELLIPSE**, **POLYGON**, **ARC** 和 **PIE**。例如，如果在改变观察口原点后调用以参数(0, 0)和(40, 40)调用 **RECTANGLE**，则矩形的左上角会与左边界距离 50 个像素，距离上边界 100 个像素。

SETCLIPRGN 在屏幕上创建一个称为“剪辑区域”的不可见区域。用户可以在剪辑区域内作图，但在剪辑区域之外的作图会失败（在剪辑区域之外没有任何显示）。缺省的剪辑区域是整个屏幕。QuickWin 库忽略在屏幕外作图的尝试。

通过调用 SETCLIPRGN 可以改变剪辑区域。例如，假设当前分辨率为 320×200，如果画一个从(0, 0)到(319, 199)的对角线，得到的图形如图 9.5 所示：

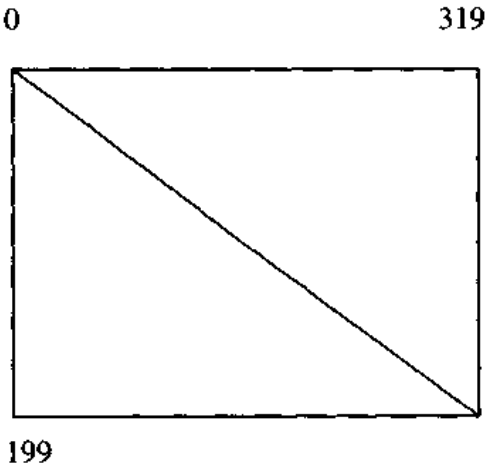


图 9.5 全屏下的直线

创建剪辑区域：

```
CALL SETCLIPRGN(INT2(10), INT2(10), INT2(309), INT2(189))
```

剪辑区域生效后，同样的 LINETO 命令将产生如图 9.6 的结果，在剪辑区域外边界的虚线是不可见的。

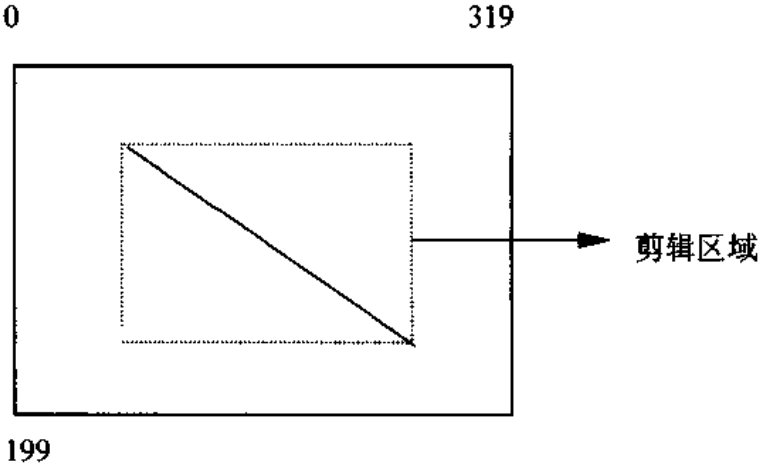


图 9.6　剪辑区域中的直线

2. 观察口坐标（Viewport Coordinates）

观察口是屏幕显示的区域，它可以是窗口的用户区域的一部分。观察口坐标代表的是当前观察口中的像素。SETVIEWPORT 在物理用户区域的边界内距离新的观察口。一个标准的观察口有两个与众不同的特性：

（1）观察口的原点是左上角。

（2）缺省的剪辑区域和观察口的外边界吻合。

SETVIEWORG 和 **SETCLIPRGN** 结合使用的效果和 **SETVIEWPORT** 相同，它指定了和 **SETCLIPRGN** 风格相同的有限区域，然后设置此区域的左上角为观察口原点。

3. 窗口坐标（Window Coordinates）

引用用户屏幕区域上的坐标和观察口内的坐标的函数需要的是整型参数。但是，很多应用程序需要的是浮点数，例如频率、粘度和质量等。SETWINDOW 允许用户缩放屏幕到几乎任何大小。另外，窗口相关函数可以接受双精度参数值。

窗口坐标使用当前观察口作为它的边界，窗口会覆盖当前观察口。在窗口外所画的图形和在观察口之外的图形一样都会被剪裁掉。

例如，要画出金星 12 个月份的平均温度（从-50 到 450），在程序中加入下面一行：

```
status = SETWINDOW(.TRUE., 1.0D0, -50.0D0, 12.0D0, 450.0D0)
```

第一个参数为转换标志，它把最低的 y 值放到最下方。随后的参数分别是 x 坐标的最小值和最大值以及 y 坐标的最小值和最大值，其中小数点表示它们为浮点数。新的屏幕组织如图 9.7 所示。

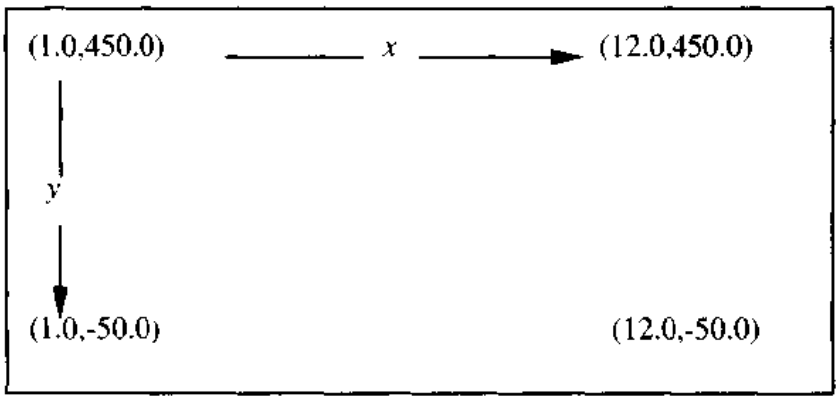


图 9.7 窗口坐标

一月和十二月分别位于屏幕的左右边界。在类似的应用程序中，如果把 x 轴编号为 0.0 到 13.0 可以留有部分边框，可以程序外观得到改善。

然后，如果用 **SETPIXEL_W** 画点或用 **LINETO_W** 画线，传输值会自动缩放以适应已建立的窗口。

用浮点参数的窗口坐标的设置和使用步骤如下：

（1） 用 SETWINDOWCONFIG 设置图形模式。

（2） 用 SETVIEWPORT 创建观察口区域。如果要使用全屏就不需要这一步。

（3） 用 SETWINDOW 创建实坐标窗口，需要传递的参数是逻辑型转换标志和四个双精度 x 和 y 坐标的最小、最大值。

（4） 用 RECTANGLE_W 或类似的例程绘制图形。注意不要把 **RECTANGLE**（使用观察口坐标）和 **RECTANGLE_W**（绘制矩形的窗口例程）。所有的窗口函数名都以下划线和字母 W。

实坐标图形的使用非常灵活并且是和设备无关的。例如，用户可以使坐标窄至 151.25 到 151.45，宽至-50000.0 到+80000.0。此外，通过改变窗口坐标，用户可以实现图像放大和移动。窗口坐标使用户的图像和计算机硬件无关，输出到观察口和实际屏幕分辨率也是无关的。

9.3.3 实坐标例程序

REALG.F90 程序（在\DF\SAMPLES\TUTORIAL 子目录下）是 Visual FORTRAN 自带示例程序，用来说明如何创建多个窗口坐标，其中每个都在单独的屏幕上有各自的观察口。程序开头如下：

```
! REALG.F90 (main program) - Illustrates coordinate graphics.
!
    USE DFLIB
    LOGICAL              statusmode
    TYPE (windowconfig)  myscreen
```

```
    COMMON                  myscreen
!
!   Set the screen to the best resolution and maximum number of
!   available colors.
    myscreen.numxpixels  = -1
    myscreen.numypixels  = -1
    myscreen.numtextcols = -1
    myscreen.numtextrows = -1
    myscreen.numcolors   = -1
    myscreen.fontsize    = -1
    myscreen.title       = " "C
    statusmode = SETWINDOWCONFIG(myscreen)
    IF(.NOT. statusmode) statusmode = SETWINDOWCONFIG(myscreen)
    statusmode = GETWINDOWCONFIG( myscreen )
    CALL threegraphs( )
    END
```

程序的主体很短，它设置窗口为显示驱动的最大分辨率（设置前四个值为-1）和最多的颜色数（设置 numcolors 为-1）。然后窗口调用画出三个图形的 threegraphs 子例程。程序输出结果如图 9.8 所示。

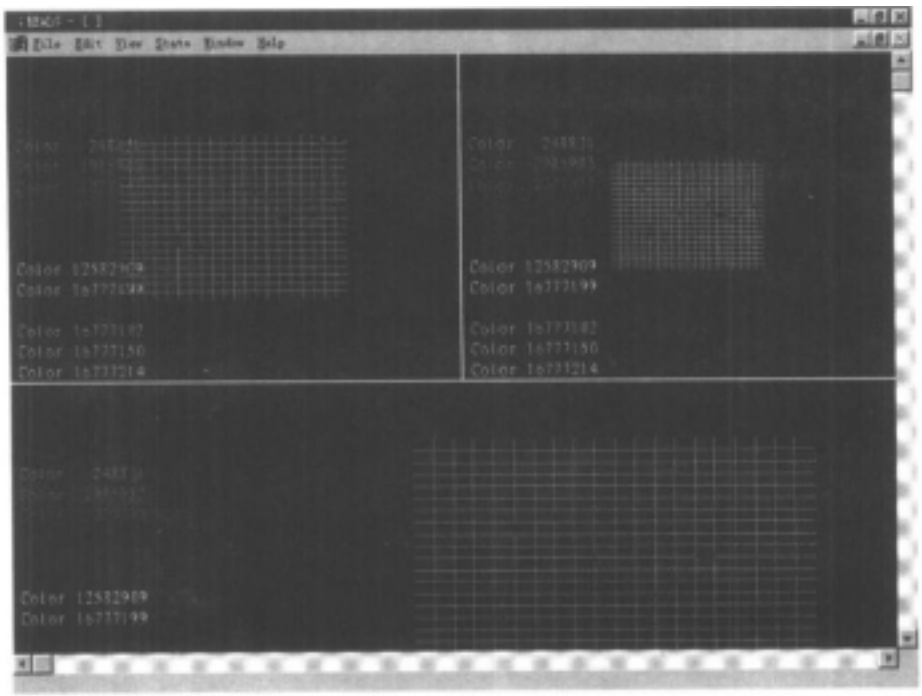


图 9.8　REALG.F90 程序输出结果

绘制图形的 gridshape 子例程在每种情况下都使用同样的数据，但是使用的坐标窗口各不相同。上半部分的两个观察口在物理坐标中大小是相同的，但窗口的大小不同。在所有的三种情况下，图形区域的宽度都为两个单元。左上的窗口在 x 轴方向宽度为四个单元；右上的窗口在 x 轴方向宽度为六个单元，这使右侧的图形显得小一些。

其中的两个图形各有一条直线超过了边界，即超过了剪辑区域，而这两条直线并没有出现在其它观察口中，因为定义了观察口而创建了剪辑区域。

最后，下面的图形转置了绘图数据，可以看出下面的图形和上面两个图形的曲线是颠倒的。

下面将说明由 REALG.F90 调用的例程。

主程序调用绘制三个图形的 threegraphs 例程：

```
    SUBROUTINE threegraphs()
    USE DFLIB
    INTEGER(2)            status, halfx, halfy
    INTEGER(2)            xwidth, yheight, cols, rows
    TYPE (windowconfig)   myscreen
    COMMON                myscreen

    CALL CLEARSCREEN( $GCLEARSCREEN )
    xwidth  = myscreen.numxpixels
    yheight = myscreen.numypixels
    cols    = myscreen.numtextcols
    rows    = myscreen.numtextrows
    halfx   = xwidth / 2
    halfy   = (yheight / rows) * ( rows / 2 )
!
!   First window
!
    CALL SETVIEWPORT( INT2(0), INT2(0), halfx - 1, halfy - 1 )
    CALL SETTEXTWINDOW( INT2(1), INT2(1), rows / 2, cols / 2 )
    status = SETWINDOW( .FALSE., -2.0_8, -2.0_8, 2.0_8, 2.0_8)
!   The 2.0_8 notation makes these constants REAL(8)

    CALL gridshape( rows / 2 )
    status = RECTANGLE( $GBORDER, INT2(0), INT2(0), halfx-1, halfy-1)
!
!   Second window
!
    CALL SETVIEWPORT( halfx, INT2(0), xwidth - 1, halfy - 1 )
```

```
      CALL SETTEXTWINDOW( INT2(1), (cols/2) + 1, rows/2, cols)
      status = SETWINDOW( .FALSE., -3.0D0, -3.0D0, 3.0D0, 3.0D0)
!     The 3.0D0 notation makes these constants REAL(8)

      CALL gridshape( rows / 2 )
      status = RECTANGLE_W( $GBORDER, -3.0_8,-3.0_8,3.0_8, 3.0_8)

!
!     Third window
!
      CALL SETVIEWPORT( 0, halfy, xwidth - 1, yheight - 1 )
      CALL SETTEXTWINDOW( (rows / 2 ) + 1, 1_2, rows, cols )
      status = SETWINDOW( .TRUE., -3.0_8, -1.5_8, 1.5_8, 1.5_8)
      CALL gridshape( INT2( (rows / 2) + MOD( rows, INT2(2))))
      status = RECTANGLE_W( $GBORDER, -3.0_8, -1.5_8, 1.5_8, 1.5_8)
      END
```

尽管屏幕开始是空的，threegraphs 还是调用 CLEARSCREEN 例程清屏：

```
CALL CLEARSCREEN( $GCLEARSCREEN )
```

$GCLEARSCREEN 常量代表清除整个屏幕，其它选项例，例如$GVIEWPORT 和 $GWINDOW 可以分别清除当前观察口和当前文本窗口。

赋值给变量后，threegraphs 创建第一个窗口：

```
CALL SETVIEWPORT( INT2(0), INT2(0), halfx - 1, halfy - 1)
CALL SETTEXTWINDOW( INT2(1), INT2(1), rows / 2, cols / 2)
status = SETWINDOW( .FALSE., -2.0_8, -2.0_8, 2.0_8, 2.0_8)
```

第一句定义覆盖左上四分之一屏幕的观察口，然后定义在边界内的文本窗口，第三句创建一个 x 和 y 值都是-2.0 到 2.0 的窗口。**.FALSE.**常量使 y 轴增加方向为从上到下（缺省值）。符号_8 说明常量的类型为 REAL(8)。

然后函数 threegraphs 插入网格并绘制数据的图像，还给窗口加边界：

```
CALL gridshape( rows / 2 )
status = RECTANGLE( $GBORDER, INT2(0), INT2(0), halfx-1, halfy-1)
```

这是标准的 **RECTANGLE** 例程，所用的坐标是和观察口而不是窗口相关的。

Gridshape 子例程绘制数据的图像：

```
! GRIDSHAPE - This subroutine plots data for REALG.F90
!
      SUBROUTINE gridshape( numc )
!
      USE DFLIB
      INTEGER(2)            numc, i, status
      INTEGER(4)            rgbcolor, oldcolor
      CHARACTER(8)          str
      REAL(8)               bananas(21), x
      TYPE (windowconfig)   myscreen
      TYPE (wxycoord)       wxy
      TYPE (rccoord)        curpos
      COMMON                myscreen
!
!     Data for the graph:
!
      DATA bananas / -0.3, -0.2, -0.224, -0.1, -0.5, 0.21, 2.9, &
     & 0.3, 0.2, 0.0, -0.885, -1.1, -0.3, -0.2, &
     & 0.001, 0.005, 0.14, 0.0, -0.9, -0.13, 0.31 /
!
!     Print colored words on the screen.
!

      IF(myscreen.numcolors .LT. numc) numc = myscreen.numcolors-1
      DO i = 1, numc
        CALL SETTEXTPOSITION( i, INT2(2), curpos )
        rgbcolor = 12**i -1
        rgbcolor = MODULO(rgbcolor, #FFFFFF)
        oldcolor = SETTEXTCOLORRGB( rgbcolor )
        WRITE ( str, '(I8)' ) rgbcolor
        CALL OUTTEXT( 'Color ' // str )
      END DO
!
!     Draw a double rectangle around the graph.
!
      oldcolor = SETCOLORRGB( #0000FF ) ! full red
      status = RECTANGLE_W( $GBORDER, -1.00_8, -1.00_8, 1.00_8,1.00_8)
! constants made REAL(8) by appending _8
```

```
      status = RECTANGLE_W( $GBORDER, -1.02_8, -1.02_8, 1.02_8, 1.02_8)
!
!     Plot the points.
!
      x = -0.90
      DO i = 1, 19
        oldcolor = SETCOLORRGB( #00FF00 ) ! full green
        CALL MOVETO_W( x, -1.0_8, wxy )
        status = LINETO_W( x, 1.0_8 )
        CALL MOVETO_W( -1.0_8, x, wxy )
        status = LINETO_W( 1.0_8, x )
        oldcolor = SETCOLORRGB( #FF0000 ) ! full blue
        CALL MOVETO_W( x - 0.1_8, bananas( i ), wxy )
        status = LINETO_W( x, bananas( i + 1 ) )
        x = x + 0.1
      END DO

      CALL MOVETO_W( 0.9_8, bananas( i ), wxy )
      status = LINETO_W( 1.0_8, bananas( i + 1 ) )
      oldcolor = SETCOLORRGB( #00FFFF ) ! yellow
      END
```

以_W 为名称后缀的例程和它们的观察口形式的例程作用相同，只是参数类型是实型而不是整型。例如，可以传递 INTEGER(2)型参数给 **LINETO**，但 REAL(8)型参数只能传递给 **LINETO_W**。

其它两个窗口和第一个类似。这三个窗口都调用 gridshape 函数，虽然网格都是从(-1.0, -1.0) 到 (1.0, 1.0)，但由于窗口中的坐标不同所以网格也不同。第二个窗口范围为从(-3.0, -3.0) 到 (3.0, 3.0)，第三个窗口范围为从(-3.0, -1.5) 到 (1.5, 1.5)。

第三个窗口包含值为**.TRUE.**的转置参数，这使 y 轴的增加方向为从下到上，结果所画的图像和上面两个图像是颠倒的。

调用 gridshape 后，程序使用下面的语句为每个窗口画边框：

```
status = RECTANGLE_W( $GBORDER, -3.0_8, -1.5_8, 1.5_8, 1.5_8)
```

第一个参数是选择填充图形还是选择只画图形的边框。剩下的参数分别是左上角和右下角的 x 和 y 坐标。

创建了不同的图形元素后用户可以用字体导向例程修饰题目、标题、注释或标签的外观。下面将介绍如何使用字体。

9.4 可用字型

字体是一组具有特定大小和形式的字符。字型是指显示文本的形式，例如宋体、黑体楷体等。字型尺寸是单个字符所占的屏幕大小，并且用像素作为字型尺寸的单位。例如"12 9"表示每个字体垂直方向占 12 个像素，水平方向占 9 个像素。

QuickWin 库的字体例程可以使用所有操作系统安装的字体。第一种字体为位图（或光栅图）字体。位图字体中每一个字符都有一个二进制数据图像，图中的每一位都对应屏幕上的一个像素。如果某位为 1，则该像素设置成当前颜色；如果某位为 0 则该像素设置成当前背景颜色。

第二种字体称为 TrueType 字体。有些字体看起来和打印出来的不同，但 TrueType 字体打印出的结果和屏幕上显示的字体是相同的（只要打印机支持）。TrueType 字体可以缩放到任意高度。

每种字体都有优点和缺点：位图字符由于预先定义的像素而在屏幕上显得更光滑，但位图字体不能缩放；TrueType 可以任意缩放字体，但不如位图字符显得光滑。因为通常屏幕的效果不易觉察，而且 TrueType 打印出来后结果和位图字符一样光滑，所以推荐在图形程序中使用 TrueType 字体。

任何字体的实际大小依赖于屏幕分辨率和显示类型。

9.5 使用字体

字体的使用包括字体的初始化和设置。

9.5.1 初始化字体

使用字体的程序必须先在内存中组织字体列表，这个过程称为初始化。列表向计算机提供可用字体的信息。

初始化字体由调用 INITIALIZEFONTS 例程完成：

```
USE DFLIB
INTEGER(2) numfonts
numfonts = INITIALIZEFONTS( )
```

如果计算机成功初始化了一种或多种字体，INITIALIZEFONTS 返回初始化字体的编号。如果失败则返回负的错误代码。

9.5.2　设置字体和显示文本

在程序以某种字体显示文本之前，程序必须知道使用何种字体。SETFONT 使初始化的一种字体为当前字体（或活动字体）。SETFONT 的语法是：

SETFONT(可选项)

函数参数由描述希望使用的字体的字母代码组成：字型，以像素为单位的字符高度，固定或成比例，以及其它诸如粗体还是斜体。例如：

```
USE DFLIB
INTEGER(2) index, numfonts
numfonts = INITIALIZEFONTS ( )
index = SETFONT('t''Cottage''h18w10')
```

上面的例子设置字型为 Cottage，字符高度为 18 像素，宽度为 10 个像素。
而下面的例子设置字型为 Arial，字符高度为 14 像素，并有成比例的间距且是斜体：

```
index = SETFONT('t''Arial''h14pi')
```

如果 SETFONT 成功设置了字体，则它返回字体的编号；如果失败则返回负整数，这时可以调用 GRSTATUS 寻找问题的原因。GRSTATUS 的返回值说明了函数失败的原因。如果在初始化字体之前调用 SETFONT 会产生运行错误。

当用 SETFONT 选择一种字体时会更新字体信息。GETFONTINFO 可以用来获得当前选择字体的信息。SETFONT 设置 fontinfo 类型（在 DFLIB.MOD 中定义）中的用户字段，GETFONTINFO 返回用户选择值。下面是 fontinfo 中包含的用户字段：

```
TYPE fontinfo
   INTEGER(2) type ! 1 = truetype, 0 = bit map
   INTEGER(2) ascent ! Pixel distance from top to baseline
   INTEGER(2) pixwidth ! Character width in pixels, 0=prop
   INTEGER(2) pixheight ! Character height in pixels
   INTEGER(2) avgwidth ! Average character width in pixels
   CHARACTER(32) facename ! Font name
END TYPE fontinfo
```

为了获得当前字体的参数可以调用 GETFONTINFO，例如：

```
USE DFLIB
TYPE (fontinfo) font
INTEGER(2) i, numfonts
numfonts = INITIALIZEFONTS()
i = SETFONT ( ' t ' 'Arial ' )
i = GETFONTINFO(font)
WRITE (*,*) font.avgwidth, font.pixheight, font.pixwidth
```

初始化字体并设置其中一个为活动字体后就可以在屏幕上显示文本了，步骤如下：

（1） 用 MOVETO 选择文本的开始位置。

（2） 可以用 SETGTEXTROTATION 设置文本显示角度。这一步不是必须的。

（3） 用 OUTGTEXT 以当前字体在屏幕上显示文本。

调用 MOVETO 时把当前图形点移到传递给 MOVETO 的坐标对应的像素。这一点就是文本中第一个字符的左上角的开始位置。

9.6 字体示例程序

字体示例程序 SHOWFONT.F90（在\DF\SAMPLES\TUTORIAL 子目录下）显示了用户系统可用的字体。运行时，如果文本充满了屏幕可用按【ENTER】显示下一屏。SHOWFONT 程序调用 SETFONT 指定字型，然后用 MOVETO 确定每个文本字符串的起点。每种字体初始化之后程序都会向屏幕发送示例文本的信息：

```
! Abbreviated version of SHOWFONT.F90.
    USE DFLIB

    INTEGER(2) grstat, numfonts,indx, curr_height
    TYPE (xycoord) xyt
    TYPE (fontinfo) f
    CHARACTER(6) str      ! 5 chars for font num
                          ! (max. is 32767), 1 for 'n'

! Initialization.
    numfonts=INITIALIZEFONTS( )
    IF (numfonts.LE.0) PRINT *,"INITIALIZEFONTS error"
    IF (GRSTATUS().NE.$GROK) PRINT *,' INITIALIZEFONTS GRSTATUS error.'
    CALL MOVETO (0.0,xyt)
    grstat=SETCOLORRGB(#FF0000)
```

```
      grstat=FLOODFILLRGB(0, 0, #00FF00)
      grstat=SETCOLORRGB(0)
! Get default font height for comparison later.
      grstat = SETFONT('n1')
      grstat = GETFONTINFO(f)
      curr_height = f.pixheight
! Done initializing, start displaying fonts.
      DO indx=1,numfonts
         WRITE(str,10)indx
         grstat=SETFONT(str)
         IF (grstat.LT.1) THEN
            CALL OUTGTEXT('SetFont error.')
         ELSE
            grstat=GETFONTINFO(f)
            grstat=SETFONT('n1')
            CALL OUTGTEXT(f.facename(:len_trim(f.facename)))
  CALL OUTGTEXT(' ')
! Display font.
            grstat=SETFONT(str)
            CALL OUTGTEXT('ABCDEFGabcdefg12345!@#$%')
         END IF
! Go to next line.
         IF (f.pixheight .GT. curr_height) curr_height=f.pixheight
         CALL GETCURRENTPOSITION(xyt)
         CALL MOVETO(0,INT2(xyt.ycoord+curr_height),xyt)
      END DO
10    FORMAT ('n',I5.5)
      END
```

第十章 使用对话框

对话框是一种请求输入程序控制命令的友好方式：程序执行时，可以在屏幕上显示一个对话框，用户可以点击按钮或滚动条或输入数据或选择下一步将要如何做。用 Visual FORTRAN 和 Developer Studio 提供的对话框函数可以向用户的 Windows (Win32), QuickWin (多文档)，标准图形(QuickWin 单文档)，控制台，DLL 和库对象添加对话框。这些函数定义了对话框的边框和控件（滚动条和按钮等）以及响应用户选择的子例程。

创建对话框有两个步骤：

（1）指定对话框外观和它包含的控件的名称和属性。

（2）编写通过识别和响应用户选择来激活这些控件的应用程序。

后面将要讨论以下几个方面：

- 使用资源编辑器设计对话框
- 编写对话框应用程序
- 对话框函数
- 对话框控件
- 使用对话框控件

10.1 使用资源编辑器设计对话框

对话框的设计要用资源编辑器来完成。

10.1.1 设计对话框

使用资源编辑器设计对话框的内容包括设计对话框的外观、选择并命名对话框中的控件和设置控件的其它属性。本节将设计一个对话框用来在摄氏和华氏温标之间相互转换。

1．打开对话框编辑器

打开对话框编辑器的步骤如下：

（1）从插入（Insert）菜单中选择资源（Resource）。

（2）在资源列表中选择对话框（Dialog）。

（3）单击新建（New）按钮，则会出现控件工具箱（如图 10.1 所示）和对话框编辑器（如图 10.2 所示）。

图 10.1　控件工具箱

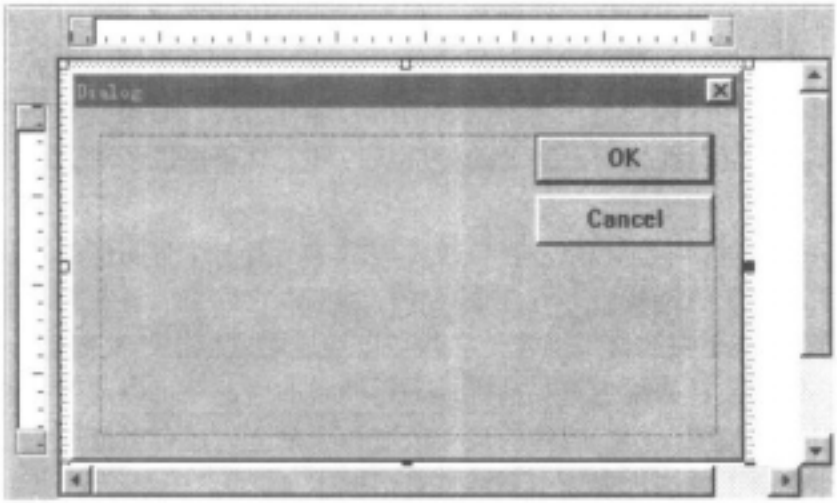


图 10.2　对话框编辑器

2．控件（Control）工具箱

图 10.3 标识了控件工具箱上可以提供的控件类型：

选择　图片　静态文本

编辑框　分组框　按钮

复选框　单选按钮　组合框

列表框　水平滚动条　垂直滚动条

微调按钮　进度指示器　滑块

热键　列表控件　树式控件

图 10.3 控件标识

3．向对话框添加控件

利用控件工具箱，用户可以单击访问对话窗口中放置的控件。单击代表希望使用的工具箱按钮，并且，将控件从工具箱拖入对话框中的位置。在拖动的时候，一个点划线的矩形显示了控件窗口的轮廓，这样，在释放鼠标按钮放置控件之前，就可以对控件的尺寸和位置一目了然。另一种方法是，只需在从控件工具箱选中一个按钮，然后，在对话窗口的任何地方单击即可。控件窗口将在单击位置的中心地方出现。如果要删除控件则单击它按【DEL】键即可。一旦已经把自己想要的控件从控件工具箱拖进了对话区，那么，下面就是在需要的位置安排控件了。

将控件窗口拖入对话框的时候，在该控件周围包围着一个有阴影的矩形，这就表示该控件已被选中了。阴影的选择矩形有八个尺寸控制柄，它们是放置在矩形四角和四边的小正方形。像在 Windows 中的典型操作一样，可以通过用鼠标指针来拖动一个尺寸控制柄，来调整选中的控件，或者如果想更加精确地进行调整的话，还可以通过按箭头键的同时按下 Shift 键来完成。每按一次，控件的尺寸改变一个单位。还可以通过在工作内的任何地方而不是在控件窗口上单击，来选中和调整对话框本身。

当用户将控件窗口拖入对话框的时候，没有必要对其进行仔细定位，因为移动已经就位的控件是很容易的。单击对话工作区中的控件来选中控件窗口，然后使用鼠标拖动或使用箭头键移动它。为了更好地对齐，可单击对话（Dialog）工具栏上的开关网格（Toggle Grid）按钮打开网格。如图 10.4 所示。

图 10.4 对话工具栏

当可以在对话窗口中看到网格的时候，控件仅从一个网格行移向另一个网格行，这个特征就是“与网格对齐”。在默认状态下，水平和垂直网格间距是 5 个对话单位，但可以在布局（Layout）菜单中的导向设置（Guide Settings）的网格设置（Grid Settings）对话框中改变这个对话框，如图 10.5 所示。

如果与网格对齐特征阻碍用户按照自己的方式来安排控件，那么只需将其关闭即可。如果后来又打开它，编辑器也不影响对话框中已经存在的控件窗口的放置。如果想临时抑制一下此特征，可以在拖动控件的时候按 Alt 键。每次将几个控件作为一组而不是单独一个进行移动或调整往往更加方便。对话编辑器为同时选中几个控件提供了两种方法。第一

种方法是在按住 Shift 键的同时依次单击想选中的控件。第二种方法最适于成组安排的控件。单击控件工具栏上的选择（Selection）工具，并将虚线矩形拖到控件上面来选中它们。如果想取消对所选组中某个控件的选中，可以在按住 Shift 键的同时单击该控件。还可以用同样的方法向该组中添加控件。选中了多个控件之后，在选中的组中，除了一个控件之外，除有控件的尺寸控制柄全都显示为空的，以表明它们处于非激活状态。剩下的那个具有实尺寸控制柄的控件被认为是选中组中的支配控件，编辑器可以从中确定作为整体的选中组的调整或对齐方式。在按下【CTRL】键的同时，单击选中组中的另一个控件，可以使其成为新的支配控件。

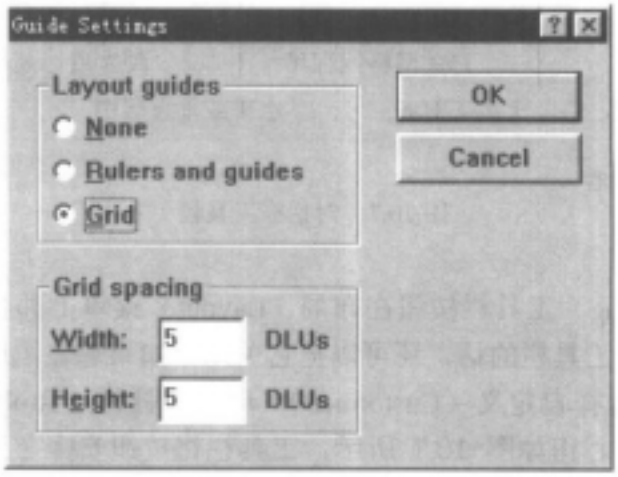


图 10.5　网格设置

图 10.6 显示了向对话框中加入的一个水平滚动条，两个文本编辑框，两个静态文本线和一个分组框。

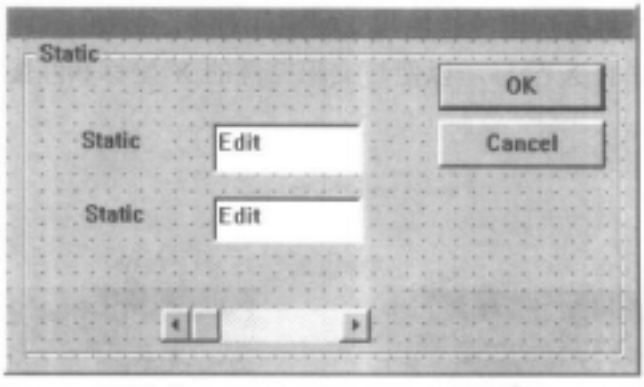


图 10.6　添加了一些控件的对话框

OK 和 CANCEL 按钮是由资源编辑器加上的，但它们和其它控件没有任何不同，可以被删除、移动、缩放或重命名。

4．对话（Dialog）工具栏

如果要精确对齐控件窗口，应该使用对话工具栏上的工具，如图 10.7 所示。利用它们可以在对话框内部对齐了的行和列中放置控件窗口，为对话提供整齐的和专业的外观。这个工具栏上还有一个测试模式开关，允许用户将新的对话用于测试驱动程序，来观察它在实际中的外观和行为。

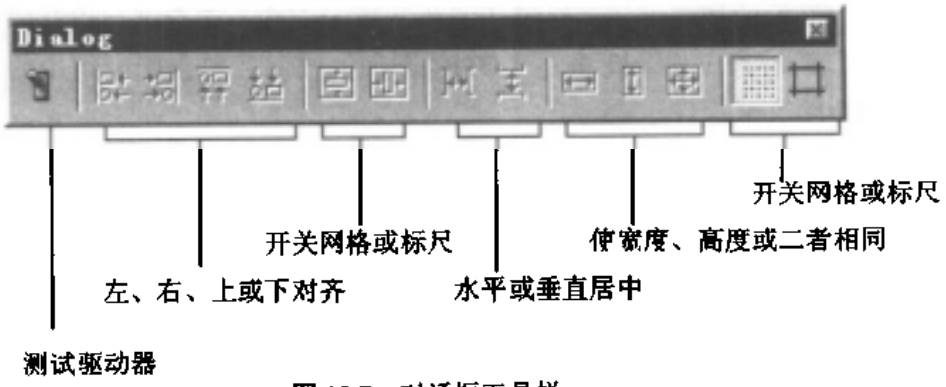


图 10.7 对话框工具栏

所有的对话（Dialog）工具栏按钮在布局（Layout）菜单上都具有功能相同的命令，因此，如果不希望看见工具栏的话，还可以把它关闭。如果想显示或隐藏工具栏，可以从工具（Tools）菜单中选择自定义（Customize），单击工具栏（Toolbars）选项卡，然后单击列表中的对话复选框。正如图 10.7 所示，工具栏将按钮安排在五个逻辑组中，它们分别用于对齐、居中、间隔、调整控件的尺寸，以及打开和关闭网格和标尺指南。在图 10.7 中呈现为灰色的对齐、间隔和尺寸调整工具，仅仅当对话中选中了两个或多个控件时才被启用。下面将阐述 Dialog 工具的几个效果。通常，为了将一组控件放到所需要的位置，需要采用几个工具。因此，必须考虑一下应用工具的顺序。如果一个工具的效果不合适，只需单击编辑（Edit）菜单上的撤消（Undo）即可。

（1）对齐工具

对齐工具的作用是将所选组的控件与支配控件对齐。例如，单击 Align Left（左对齐）按钮，将改变选中控件的 X 坐标，以便与支配控件的 X 坐标和匹配，而不影响 Y 坐标。如图 10.8 所示。


左对齐工具

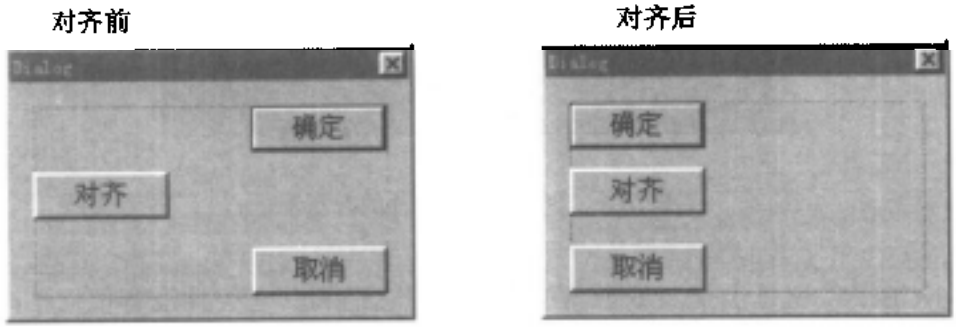


图 10.8 对齐工具

（2）居中工具

居中工具可以作用于单个选中控件或一组控件，将选中部分放在对话客户区的水平或垂直中心，如图 10.9 所示。

 居中工具

居中前　　居中后

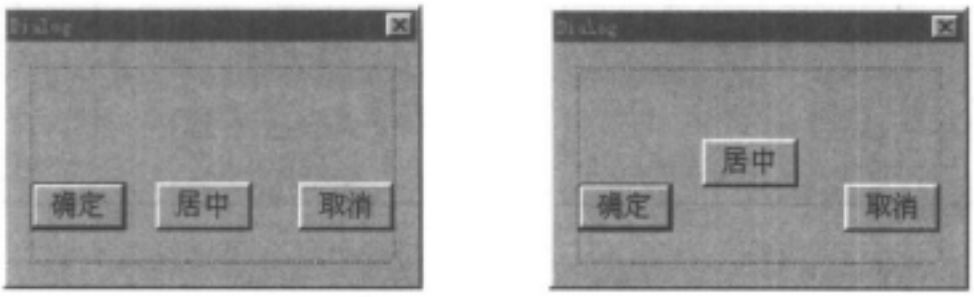


图 10.9　居中工具

（3）间隔工具

间隔工具仅仅在包含有三个或更多控件的选中组上才能工作。在对话（Dialog）工具中，它们是独一无二的，因为它不能区分哪一个选中控件是支配控件。水平间隔将改变选中组中除了最左边和最右边控件之外的 X 坐标，使水平方向的控件均匀排列。垂直间隔工具完成 Y 坐标方向上相同的工作，使控件在垂直方向上均匀分布。在下例中，垂直间隔工具(编辑器为它取了个容易引起混淆的名字，向下间隔（Space Down）)调整的仅仅是间隔按钮，确定和取消按钮依旧在原来的位置不变，如图 10.10 所示。

 间隔工具

调整间隔前　　调整间隔后

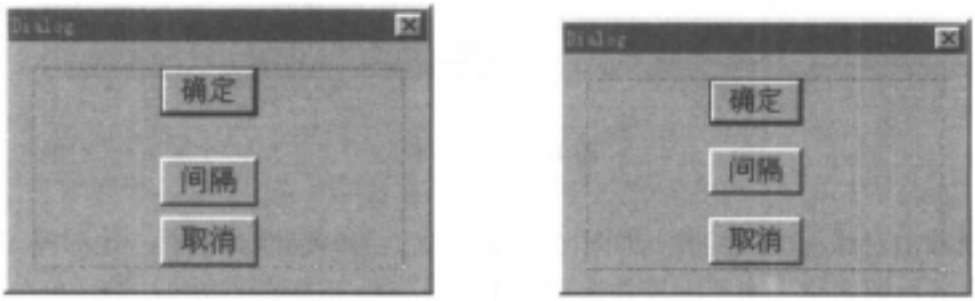


图 10.10　间隔工具

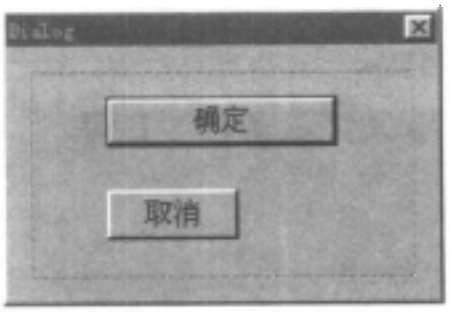
（4）尺寸调整工具

尺寸调整工具作用于包含有两个或更多控件的选中组。尺寸调整并不移动控件窗口，而仅仅改变选中的控件的高度和宽度，以便与选中部分中支配控件的尺寸相配，如图 10.11

所示。

 尺寸调整工具

调整尺寸前



调整尺寸后

图 10.11 尺寸调整工具

5. 指定控件的名称和属性

对话工作区中的每个控件都有一个属性（Properties）对话框，可以在里面为控件指定标识符和值，键入标签，设置适合于控件窗口的样式标志。如果想展现控件的属性框，可以右击工作区中的控件窗口，并从弹出的上下文菜单上选择 Properties。还可以选中一个控件，并单击查看（View）菜单上的 Properties 命令。每个控件类型的属性对话在外观和内容上略有不同。图 10.12 显示了一个滚动条控件的属性框。

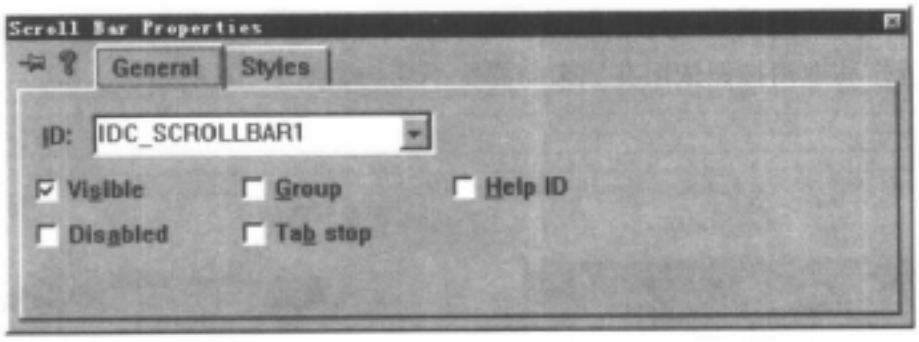


图 10.12 属性框

还可以双击对话框中的控件，同样会出现类似上图的属性对话框。在标题（Caption）栏中可以设置标题名称，在其它控件选项中也可以作相应的设置改动。最后单击右上角的×按钮来保存属性并关闭对话框。

在程序中使用控件时，每个控件都需要一个符号名。在上面的水平滚动条中，符号名改变为 IDC_SCROLLBAR_TEMPERATURE，这是用户程序引用控件的途径。例如，当获得滚动条的滑动位置时：

```
INTEGER slide_position
```

```
retlog = DLGGET (dlg, IDC_SCROLLBAR_TEMPERATURE,          &
                 slide_position, DLG_POSITION)
```

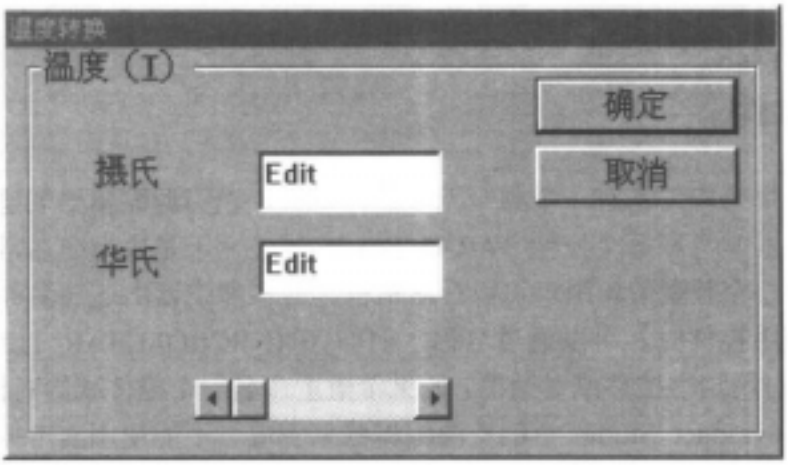


图 10.13　修改设置后的对话框

图 10.13 的结果是由图 10.6 中的对话框修改而成：图 10.6 中对话框上面的编辑框命名（ID）为 IDC_EDIT_CELSIUS，相邻的静态文本命名为 IDC_TEXT_CELSIUS，标题改为"摄氏"；下面的编辑框命名为 IDC_TEXT_FAHRENHEIT，相邻的静态文本命名为 IDC_TEXT_FAHRENHEIT，标题改为"华氏"。

分组框命名为 IDC_BOX_TEMPERATURE，标题设置为温度（&T）（连字符&给字母"T"加一个下划线并且使它成为 Windows 热键，可以用【ALT+T】键激活）。对话框本身命名为 IDD_TEMP，标题设置为 Temperature Conversion。其它的控件属性都设为缺省值不变。

6．保存对话框为资源文件

保存对话框为资源文件的步骤如下：

（1）从文件（File）菜单中选择另存为（Save As）。

（2）输入想要的资源文件名。

在这个例子中，源文件的名称为 TEMP.RC，Developer Studio 把它存为资源文件并创建一个包含文件，其名称为 RESOURCE.H。

这时对话框的外观已经完成，控件也已经命名，但对话框不能自己工作，必须创建一个应用程序使它运行。

10.1.2　设置控件属性

在资源编辑器中有解释每个对话框控件的帮助。有些控件有两组属性：一般（General）和风格（Styles）。在属性组的名称设置需要查看或修改的属性，用户可以在对话框中任意空白区域双击可以更改对话框本身的属性。

修改对话框的属性框中的 x 值和 y 值可以改变对话框出现在屏幕上的位置：它们指定

了对话框在左上角。

如果以后要再编辑对话框的外观可以从文件菜单中打开资源文件（扩展名为.RC），然后单击对话框图标，或者选择资源查看（Resource View）选项卡。对话框打开时会自动调用编辑器。

10.1.3 包含文件

对话框中的每个控件都有一个独一无二的标识符，当资源编辑器创建包含文件（.H）时，它赋给对话框本身和每个控件 PARAMETER 属性，于是成为命名常量。它还给话框本身和每个控件一个整型值。用户可以在对话框包含文件中读出这些名称和值的列表。

用户程序使用控件时，可以通过名称（例如 IDC_SCROLLBAR_TEMPERATURE 或 IDD_TEMP）来引用这个控件或对话框。如果希望重命名一个控件或给对话框做其它改动，用户可以通过 Developer Studio 中的资源编辑器来完成。不能使用文本编辑器来修改.H，因为对话框资源不会接受这些改动。

10.2 编写对话框程序

用资源编辑器创建一个对话框后，用户必须指定框中包含的控件的类型和显示。然后必须提供使对话框活动的过程，这些过程使用对话框函数和子例程来控制程序对对话框输入的响应。

通过向项目中加入.H 文件可以使用户的应用程序访问对话框资源，并用这些文件中的 dialog 类型来联系对话框的属性。

用户的应用程序必须包含 USE DFLOGM 语句来访问对话框函数，但它必须包含对话框创建的.H 文件。例如：

```
PROGRAM TEMPERATURE
USE DFLOGM
IMPLICIT NONE
INCLUDE 'TEMP.FD'
CALL DoDialog( )
END PROGRAM
```

10.2.1 初始化并激活对话框

每个对话框都有和派生类型 dialog 联系的变量。Dialog 派生类型是在 DFLOGM.F90 模块中定义，可以用 USE DFLOGM 语句访问。当用户编写对话框应用程序时，引用对话框作为 dialog 类型的变量，例如：

```
USE DFLOGM
INCLUDE 'TEMP.FD'
TYPE (dialog) dlg
LOGICAL return
return = DLGINIT( IDD_TEMP, dlg )
```

和对话框联系的 dialog 代码（本例中的 IDD_TEMP）是在资源和包含文件中定义。向项目中添加.RC 文件可以使应用程序访问对话框资源文件。在每个子程序中通过包含.H 文件使应用程序可以访问对话框包含文件。通过和对话框名称一起调用 DLGINIT 把这些文件中的对话框属性和 dialog 类型联系起来。

一个控制对话框的应用程序应该实施下列操作：

（1）调用 DLGINIT 初始化 dialog 联系并和对话框的属性相联系。

（2）用对话框设置函数（例如 DLGSET）初始化控件。

（3）用 DLGSETSUB 设置用户在对话框中操作控件时要执行的反馈例程。

（4）用 DLGMODAL 运行对话框。

（5）用对话框获得函数（例如 DLGGET）来获得控件信息。

（6）用 DLGUNINIT 释放对话框中的资源。

下面的代码初始化上面例子中的温度对话框。它激活了对话框和控件，还设置了反馈例程 UpdateTemp，并显示对话框，在这些完成之后释放对话框资源：

```
SUBROUTINE DoDialog( )
USE DFLOGM
IMPLICIT NONE
INCLUDE 'TEMP.FD'

INTEGER retint
LOGICAL retlog
TYPE (dialog) dlg
EXTERNAL UpdateTemp
! Initialize.
IF ( .not. DlgInit( idd_temp, dlg ) ) THEN
  WRITE (*,*) "Error: dialog not found"
ELSE
! Set up temperature controls.
  retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, 200, DLG_RANGE)
  retlog = DlgSet( dlg, IDC_EDIT_CELSIUS, "100" )
  CALL UpdateTemp( dlg, IDC_EDIT_CELSIUS, DLG_CHANGE)
  retlog = DlgSetSub( dlg, IDC_EDIT_CELSIUS, UpdateTemp )
```

```
    retlog = DlgSetSub( dlg, IDC_EDIT_FAHRENHEIT, UpdateTemp )
    retlog = DlgSetSub( dlg, IDC_SCROLLBAR_TEMPERATURE, UpdateTemp )
! Activate the dialog.
    retint = DlgModal( dlg )
! Release dialog resources.
    CALL DlgUninit( dlg )
  END IF
  END SUBROUTINE DoDialog
```

对话框函数（例如 DLGSET 和 DLGSETSUB）通过在资源编辑器中创建对话框使在属性框中指定的对话框控件名称来引用控件。例如：

```
retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, 200, DLG_RANGE)
```

在上面的语句中，对话框函数 DLGSET 指定名为 IDC_SCROLLBAR_TEMPERATURE 的控件值为 200，索引 DLG_RANGE 指定了滚动条的范围，考虑下面的语句：

```
retlog = DlgSet( dlg, IDC_EDIT_CELSIUS, "100" )
CALL UpdateTemp( dlg, IDC_EDIT_CELSIUS, DLG_CHANGE)
```

上面的语句设置了图 10.13 中上面的编辑框，它在资源编辑器里被命名为 IDC_EDIT_CELSIUS，初始值为 100，并调用 UpdateTemp 例程来把这个初始值写到屏幕上。考虑下面的语句：

```
retlog = DlgSetSub( dlg, IDC_EDIT_CELSIUS, UpdateTemp )
retlog = DlgSetSub( dlg, IDC_EDIT_FAHRENHEIT, UpdateTemp )
retlog = DlgSetSub( dlg, IDC_SCROLLBAR_TEMPERATURE,UpdateTemp )
```

上面的语句把 UpdateTemp 反馈例程和三个控件联系起来。例程是由 DLGSETSUB 设置成与控件相联系的，第一个参数是对话框变量，第二个是控件名称，第三个是为该控件编写的例程，第四个是可选参数，是可以在多个例程中选择的索引。用户可以在自己应用程序中的任何位置设置对话框控件的反馈例程：在用 DLGMODAL 打开对话框之前设置或在其它反馈例程中设置。

10.2.2 对话框反馈例程

所有反馈例程应该有下面的接口：

SUBROUTINE callback (*dlg*,控件名称, 反馈类型)

dlg：

引用对话框并允许反馈改变对话框的值

控件名称：

引起反馈的控件的名称

反馈类型：

说明发生什么反馈（例如 DLG_CLICKED, DLG_CHANGE, DLG_DBLCLICK）

最后两个参数使用户可以编写可供多个控件使用的独立子例程。例如，上面的例子中的对话框的所有控件都和同应该反馈例程 UpdateTemp 联系。也可以把同一个控件和多个反馈例程，但必须提供调用哪个反馈例程的索引参数。

下面是一个反馈例程的例子。

```
SUBROUTINE UpdateTemp( dlg, control_name, callbacktype )
USE DFLOGM
IMPLICIT NONE
TYPE (dialog) dlg
INTEGER control_name
INTEGER callbacktype
INCLUDE 'TEMP.FD'
CHARACTER(256) text
INTEGER cel, far, retint
LOGICAL retlog
! Suppress compiler warnings for unreferenced arguments.
INTEGER local_callbacktype
local_callbacktype = callbacktype

SELECT CASE (control_name)
  CASE (IDC_EDIT_CELSIUS)
  ! Celsius value was modified by the user so
  ! update both Fahrenheit and Scroll bar values.
    retlog = DlgGet( dlg, IDC_EDIT_CELSIUS, text )
    READ (text, *, iostat=retint) cel
    IF ( retint .eq. 0 ) THEN
      far = (cel-0.0)*((212.0-32.0)/100.0)+32.0
      WRITE (text,*) far
      retlog = DlgSet( dlg, IDC_EDIT_FAHRENHEIT,                    &
&                      TRIM(ADJUSTL(text)) )
      retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, cel,         &
&                      DLG_POSITION )
    END IF
```

```
    CASE (IDC_EDIT_FAHRENHEIT)
    ! Fahrenheit value was modified by the user so
    ! update both celsius and Scroll bar values.
       retlog = DlgGet( dlg, IDC_EDIT_FAHRENHEIT, text )
       READ (text, *, iostat=retint) far
       IF ( retint .eq. 0 ) THEN
         cel = (far-32.0)*(100.0/(212.0-32.0))+0.0
         WRITE (text,*) cel
         retlog = DlgSet( dlg, IDC_EDIT_CELSIUS, TRIM(ADJUSTL(text)) )
         retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, cel,          &
    &                     DLG_POSITION )
       END IF
    CASE (IDC_SCROLLBAR_TEMPERATURE)
    ! Scroll bar value was modified by the user so
    ! update both Celsius and Fahrenheit values.
       retlog = DlgGet( dlg, IDC_SCROLLBAR_TEMPERATURE, cel,          &
    &                    DLG_POSITION  )
       far = (cel-0.0)*((212.0-32.0)/100.0)+32.0
       WRITE (text,*) far
       retlog = DlgSet( dlg, IDC_EDIT_FAHRENHEIT, TRIM(ADJUSTL(text)) )
       WRITE (text,*) cel
       retlog = DlgSet( dlg, IDC_EDIT_CELSIUS, TRIM(ADJUSTL(text)) )
    END SELECT
  END SUBROUTINE UpdateTemp
```

除了按钮之外，对话框中的每一个控件都有一个不进行任何操作的缺省反馈。而按下按钮的缺省反馈是符号对话框的值并退出对话框。这使所有的按钮在缺省情况下都退出对话框，并给“确定”和“取消”按钮较好的缺省动作。调用 DLGMODAL 可以测试是哪个按钮引起对话框的退出。

特定的控件的反馈例程是在控件的值被用户操作改变后调用的。调用 DLGSET 并不调用改变控件值的反馈例程。特别地，在反馈内部，对一个控件执行 DLGSET 包含引起和该控件联系的反馈例程的被调用。

在 DLGMODAL 之前或之后调用 DLGSET 不会调用例程。如果反馈例程需要被调用，可以在执行 DLGSET 之后手动使用 CALL callbackroutine 来调用。

10.3 对话框函数

用户可以像使用任何内在函数或运行函数一样使用对话框函数，它们和标准图形项

目、QuickWin 项目和 Windows (Win32)项目都是兼容的。对话框函数可以：

（1）初始化和关闭对话框

（2）从对话框中获得用户输入

（3）在对话框中显示数据

（4）修改对话框控件

对话框包含文件（.FD）包含了创建对话框时在资源编辑器中指定的对话框控件属性。模块 DFLOGM.MOD 包含了预先定义的变量名和类型定义。在管理用户的对话框时，这些控件名、变量和类型定义用来作对话框函数的参数列表。

对话框函数如表 10.1 所示。

表 10.1　　对话框函数

对话框函数	功　能
DLGEXIT	关闭一个打开的对话框
DLGGET	获得控件变量的值
DLGGETCHAR	获得字符控件变量的值
DLGGETINT	获得整型控件变量的值
DLGGETLOG	获得逻辑型控件变量的值
DLGINIT	初始化对话框
DLGMODAL	显示对话框
DLGSET	给控件变量赋值
DLGSETCHAR	给字符控件变量赋值
DLGSETINT	给整型控件变量赋值
DLGSETLOG	给逻辑控件变量赋值
DLGSETRETURN	设置 DLGMODAL 的返回值
DLGSETSUB	给控件指定一个定义的反馈例程
DLGUNINIT	释放初始化对话框的内存

10.4　对话框控件

对话框中的每个控件都有一个单独的整型标识符和名称，用户可以提供它的名称，例如 IDC_SCROLLBAR_TEMPERATURE，来引用控件，或通过它的整型标识符来引用控件。整型标识符可以在包含文件中读出。

每个控件都有一个或多个变量和控件联系，称为控件索引。这些控件索引可以是整型、逻辑型、字符型或外部类型。例如，一个简单的按钮有三个联系的变量：一个是和按钮当前状态联系的逻辑型变量，一个是决定其标题（如“OK”或“CANCEL”）的字符变量，另一个是外部变量，它说明了如果鼠标单击发生将要调用的子例程。

控件可以有同一类型的多个变量。例如，滚动条可以有四个相联系的整型变量：滚动条位置，滚动条范围，用户点击滚动条箭头的位置改变（小步改变）和用户点击滑块旁边的空白区域产生的位置改变（大步改变）。

本节从以下几个方面来讨论控件及其索引：

- 控件索引
- 每个对话框控件的可用索引
- 说明控件索引

10.4.1 控件索引

对话框控件索引的值是在函数 DLGSET 中设置的：DLGSET, DLGSETINT, DLGSETLOG, DLGSETCHAR 和 DLGSETSUB。控件名称和控件索引名称都是 DLGSET 函数的参数并指定了特定控件索引的设置。例如：

```
retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, 45, DLG_POSITION )
```

索引 DLG_POSITION 指定了滚动条的位置被设置为 45。考虑下面的语句：

```
retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, 200, DLG_RANGE )
```

索引 DLG_RANGE 指定了滚动条的范围为 200。DLGSET 函数的语法如下：

return = **DLGSET** (dlg,控件名称, 值, 控件索引名称)

控件索引名称决定了 DLGSET 函数表示的值是什么。

控件索引名称在模块 DFLOGM.MOD 中声明，不能在用户的例程中声明。可用控件索引和它们如何指定参数值的解释如表 10.2 所示：

表 10.2 控件索引

控件索引	值的解释
DLG_BIGSTEP	当用户点击滑块旁边的区域时滚动条位置改变的大小（缺省为 10）
DLG_CHANGE	用户修改控件和控件在屏幕上更新后调用的子例程
DLG_CLICKED	控件获得鼠标点击后调用的子例程
DLG_DBLCLICK	控件被双击时调用的子例程
DLG_DEFAULT	和不指定控件索引相同
DLG_ENABLE	控件的可用状态（值为.TRUE. 代表可用，值为.FALSE.代表禁止）
DLG_NUMITEMS	列表框或结合框中项的总数目
DLG_POSITION	滚动条当前位置
DLG_RANGE	滚动条的最大位置（缺省为 100）；最小值始终为 1
DLG_SELCHANGE	列表框或结合框中的选择改变时调用的子例程
DLG_SMALLSTEP	单击滚动条箭头时滚动条位置的改变量
DLG_STATE	控件的用户可变状态
DLG_TITLE	和控件联系的标题文本
DLG_UPDATE	在用户改变控件状态之后、控件在屏幕更新之前调用的子例程

除了控件中都多个同一类型的变量而且用户想改变缺省值之外，不需要使用和对话框联系的索引名称。例如：

```
retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, 45, DLG_POSITION )
retlog = DlgSet( dlg, IDC_SCROLLBAR_TEMPERATURE, 45)
```

这两条语句都设置滚动条的位置为 45，因为 DLG_POSITION 是滚动条的缺省索引。

10.4.2　对话框控件的可用索引

每个控件的可用索引和缺省值在表 10.3 中列出。

表 10.3　　对话框控件的可用索引

控件类型	整型索引名	逻辑索引名	字符索引名	子例程索引名
静态文本		DLG_ENABLE	DLG_TITLE	
分组框		DLG_ENABLE	DLG_TITLE	
按钮		DLG_ENABLE	DLG_TITLE	DLG_CLICKED
复选框		DLG_STATE (缺省值) DLG_ENABLE	DLG_TITLE	DLG_CLICKED
单选按钮		DLG_STATE (缺省值) DLG_ENABLE	DLG_TITLE	DLG_CLICKED
编辑框		DLG_ENABLE	DLG_STATE	DLG_CHANGE (缺省值) DLG_UPDATE
滚动条	DLG_POSITION (缺省值) DLG_RANGE DLG_BIGSTEP DLG_SMALLSTEP	DLG_ENABLE		DLG_CHANGE
列表框	DLG_NUMITEMS 设置或返回列表中项的数目，或包含一个索引 1 到 n，来决定选择哪一项和它们的顺序	DLG_ENABLE	DLG_STATE 缺省时设置或返回第一个被选项的文本，或可以包含索引 1 到 n 来设置或返回特定的文本	DLG_SELCHANGE (缺省值) DLG_DBLCLICK
组合框	DLG_NUMITEMS 设置或选择列表中的项数目，或包含一个索引 1 到 n，来决定选择哪一项和它们的顺序	DLG_ENABLE	DLG_STATE 缺省时设置或返回第一个被选项的文本，或可以包含索引 1 到 n 来设置或返回特定的文本	DLG_SELCHANGE (缺省值) DLG_DBLCLICK DLG_CHANGE DLG_UPDATE

续表

控件类型	整型索引名	逻辑索引名	字符索引名	子例程索引名
下拉列表框	DLG_NUMITEMS (缺省值) 设置或返回列表中项的数目，或包含一个索引 1 到 n，来决定选择哪一项和它们的顺序 DLG_STATE 设置或返回被选择项的索引	DLG_ENABLE	DLG_STATE 缺省时设置或返回第一个被选项的文本，或可以包含索引 1 到 n 来设置或返回特定的文本	DLG_SELCHANGE (缺省值) DLG_DBLCLICK

10.4.3 指定控件索引

当某个对话框的控件索引类型（整型、逻辑型、字符型或子例程）只有一种可能性，可以不需在参数列表中指定控件索引名称。例如，可以通过下面两条语句都可以设置静态文本控件 IDC_TEXT_CELSIUS 为一个新值：

```
retlog = DLGSETCHAR (dlg, IDC_TEXT_CELSIUS, "New Celsius Title", &
&                    DLG_TITLE)
retlog = DLGSET (dlg, IDC_TEXT_CELSIUS, "New Celsius Title")
```

因为静态文本控件只有一个字符索引，所以不需要控件索引 DLG_TITLE。一般函数 DLGSET 基于参数类型选择控件索引来改变，本例中是字符型 CHARACTER。

每种索引类型都可以使用一般函数 DLGSET 或指定 DLGSET 函数的类型：DLGSETINT, DLGSETLOG 或 DLGSETCHAR。例如，可以通 DLGSET 或 DLGSETLOG 设置逻辑值为.FALSE.来禁止静态文本控件 IDC_TEXT_CELSIUS

```
retlog = DLGSETLOG (dlg, IDC_TEXT_CELSIUS, .FALSE., DLG_ENABLE)
retlog = DLGSET (dlg, IDC_TEXT_CELSIUS, .FALSE., DLG_ENABLE)
```

上面两种情况下，控件索引 DLG_ENABLE 可以被忽略，因为静态文本控件只有一个逻辑控件索引。

可以用 DLGGET 函数和 DLGGET, DLGGETINT, DLGGETLOG, DLGGETCHAR 来查询某个控件索引的值，例如：

```
INTEGER current_val
LOGICAL are_you_enabled
    retlog = DLGGET (dlg, IDC_SCROLLBAR_TEMPERATURE, current_val,      &
   &                 DLG_RANGE)
    retlog = DLGGET (dlg, IDC_SCROLLBAR_TEMPERATURE, are_you_enabled,  &
   &                 DLG_ENABLE)
```

上面的代码返回的是滚动条的范围和状态。声明的存有查询值的参数(上例中的current_val 和 are_you_enabled)必须和获得的值类型相同。如果使用特殊 DLGGET 函数（例如 DLGGETINT 或 DLGGETCHAR），获得的控件索引值必须是相应的类型。例如，不能用 DLGGETCHAR 获得整型或逻辑型值。DLGGET 函数对于非法的类型结合会返回.FALSE.。用户不能查询外部反馈例程的名称。

一般来说，使用一般函数 DLGSET 和 DLGGET 比使用特定类型函数要好，因为不用考虑函数类型与设置或获得值的匹配。DLGSET 和 DLGGET 可以根据传递给它们的参数自动作出正确的操作。

10.5　使用对话框控件

在上面内容的基础上可以开始使用对话框控件。

10.5.1　概述

资源编辑器中提供的对话框控件功能多样使用灵活，如果结合使用会给用户的应用程序提供非常专业、友好的界面。

用户的应用程序可以在任何时候禁止任何控件，此后它就不再改变或响应用户。这是通过用 DLGSET 和 DLGSETLOG 把控件索引 DLG_ENABLE 设置为.FALSE.实现的。例如：

```
LOGICAL retlog
retlog = DLGSET (dlg, IDC_CHECKBOX1, .FALSE., DLG_ENABLE)
```

上面的例子禁止了名为 IDC_CHECKBOX1 的控件。

在资源编辑器中创建对话框后，对话框控件就有一个按【Tab】键移动的顺序。用户可以通过反复按【Tab】键跳到某一控件。每当系统检测到【Tab】键的时候，它就把输入焦点移动到【Tab】键移动顺序的下一个控件。检查顺序通常是在单击对话（Dialog）工具栏上的测试开关之前所做的最后一件事情。从布局（Layout）菜单选择 Tab 次序（TabOrder），或按组合键【CTRL＋D】，可以显示当前对话的 Tab 键移动顺序，如图 10.14 所示。该顺序是作为序号码在对话控件附近出现的。如果想修改 Tab 键的移动顺序，可以单击以第一个控件开头的序列中的每个控件，也就是当对话首次出现时，用户希望让其具有输入焦点的那个控件。如果现有的顺序仅仅对于某一点来说是正确的，那么只改变错误的是很容易的：在单击具有最高的正确【Tab】键的移动顺序号的控件窗口的同时按下 Ctrl 键，然后释放 Ctrl 键，并继续单击所要求顺序中的控件，直到顺序正确为止。例如，如果想改变控件 4 到 6 的顺序，请在按下【CTRL】键的时候单击控件 3，然后依次单击希望让其拥有制表号 4，5 和 6 的控件。按【Enter】键设置顺序并返回到编辑模式。

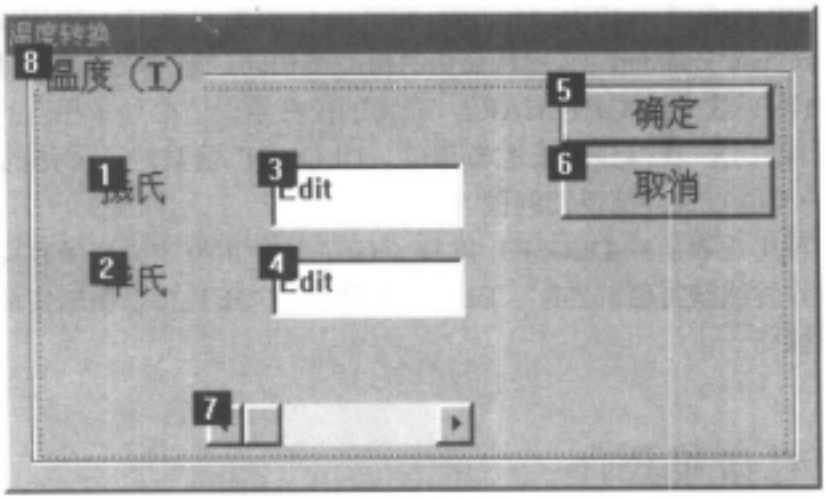


图 10.14 Tab 顺序

下面将说明这几种对话框控件的函数和使用：

- 静态文本
- 编辑框
- 分组框
- 复选框和单选框
- 按钮
- 滚动条
- 设置返回值和退出

10.5.2 静态文本

静态文本是在对话框中写入文本的区域，程序运行后使用者不能改变这些文本。应用程序可以在任何时候修改静态文本，例如显示当前用户的选择。静态文本通常是用来标志其它文本或为用户显示信息。图 10.14 中编号为 1 和 2 的控件都是静态文本。

10.5.3 编辑框

编辑框是程序可以在任何时候写入文本的区域。图 10.14 中编号为 3 和 4 的控件都是编辑框。与静态文本不同，程序使用者可以单击编辑框后输入文本。下面的语句向编辑框写入内容：

```
CHARACTER(20) text /"Send text"/
retlog = DLGSET (dlg, IDC_EDITBOX1, text)
```

下面的语句读出编辑框中的文本：

```
retlog = DLGGET (dlg, IDC_EDITBOX1, text)
```

输入到编辑框中的内容始终以字符串的形式获得，应用程序需要把这些字符串解释成它们所代表的数据。例如，在一个计算器程序中，程序要把编辑框（相当于实际计算器的液晶屏幕）输入的字符串解释为数值。数值和字符串之间的转换可以使用内部读写语句来完成。

为了从编辑框中读入数值，可以先用 DLGGET 或 DLGGETCHAR 获得一个字符串，然后用需要的内部数值类型（例如整型或实型）变量执行读入，例如：

```
REAL    x
LOGICAL retlog
CHARACTER(256) text
retlog = DLGGET (dlg, IDC_EDITBOX1, text)
READ (text, *) x
```

上面的例子中，实型变量 x 被赋为在编辑框中输入的值，包括小数点。

复型值和双精度值也可以用同样的方法来读入，只是应用程序必须把编辑框中的字符串分为实部和虚部。可以使用两个独立的编辑框，一个用来输入实部另一个用来输入虚部。还可以在同一个编辑框中输入，但要求在实部和虚部之间输入分隔符，程序在转换为数值之前先根据分隔符把字符串分为两部分。如果分隔符为逗号，可以用两个实型编辑描述符读入而不需把字符串分成两部分。

向编辑框写入数值时应该进行写入字符串的内部操作，然后再用 DLGSET 把这个字符串送入编辑框。例如：

```
INTEGER  j
LOGICAL  retlog
CHARACTER(256) text
WRITE (text,'(I4)') j
retlog = DLGSET (dlg, IDC_EDITBOX1, text)
```

Visual FORTRAN 不支持多行编辑框。

10.5.4　分组框

分组框可视为组织一组控件集合。图 10.14 中编号为 8 的控件就是分组框。当用户在资源编辑器中选择分组框的同时就创建了一个在希望分组的控件旁边的展开（或收缩）框，并给这组控件一个标题。可以通过给控件组的标题一个连字符来定义热键，例如：

```
&Temperature
```

这使字母“T”有一条下划线，并且成为热键。

禁止分组框也会禁止热键，但并不禁止组中的任何控件。一般来说，在禁止了分组框后也应主动禁止该组中的控件。

10.5.5 复选框和单选框

复选框和单选框提供给用户选择的机会。顾名思义，复选框允许多选，如图 10.15 所示。而单选框只允许选择其一，如图 10.16 所示。

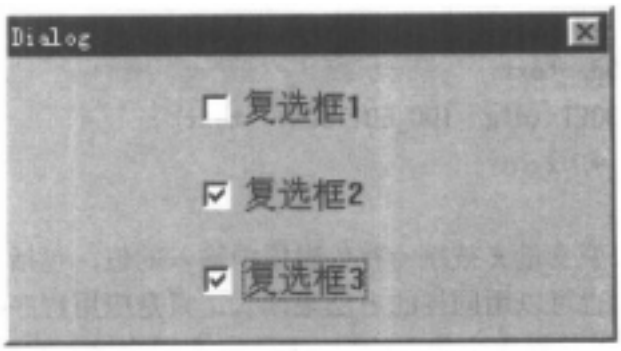


图 10.15 复选框

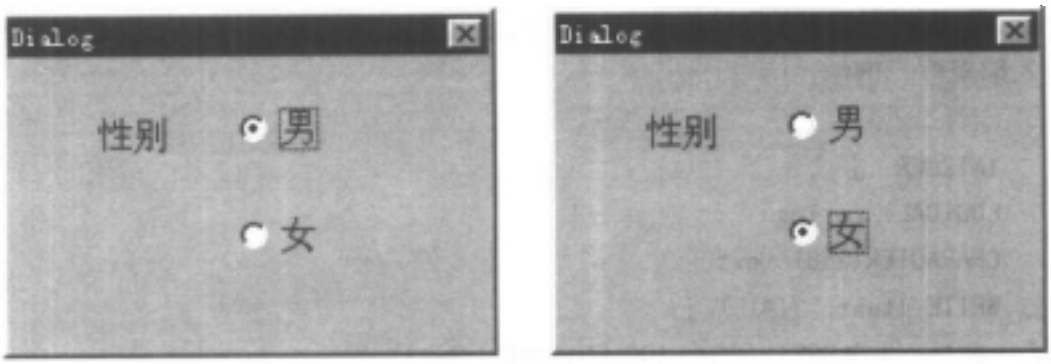


图 10.16 单选框

单选框的状态是按下或没有按下，复选框的状态是选了或没有选。可以使用 DLGGET 或 DLGGETLOG 来检查这两个控件的状态。表示状态的是一个逻辑量，如果为.TRUE.代表按下或选择，否则为.FALSE.：

```
LOGICAL pushed_state, checked_state, retlog
retlog = DLGGET (dlg, IDC_RADIOBUTTON1, pushed_state)
```

```
retlog = DLGGET (dlg, IDC_CHECKBOX1, checked_state)
```

如果为了初始化或响应用户的输入而需要改变按钮的状态可以使用 DLGSET 或 DLGSETLOG，例如：

```
LOGICAL retlog
retlog = DLGSET (dlg, IDC_RADIOBUTTON1, .FALSE.)
retlog = DLGSET (dlg, IDC_CHECKBOX1, .TRUE.)
```

10.5.6 按钮

与复选框和单选框不同，按钮没有状态：按钮不保存按下或释放的值。当用户用鼠标点击按钮时，按钮的反馈例程就被调用，按钮的作用是触发动作。指定为反馈的外部过程决定了触发的动作。例如：

```
LOGICAL retlog
EXTERNAL DisplayTime
retlog = DlgSetSub( dlg, IDC_BUTTON_TIME, DisplayTime)
```

Visual FORTRAN 不支持用户画的按钮。

10.5.7 列表框和组合框

当需要从一组值中选择一个值时要用到列表框和组合框。它们和单选框类似，只是列表框和组合框是可滚动的并不受屏幕显示区域的限制而可以包含更多的选项。并且，与单选按钮不同的是，列表框和组合框的实体数目在运行时是可以改变的。

列表框和组合框的区别是：列表框只是一个选项的列表，而组合框是一个列表框和一个编辑框的组合；列表框允许用户在列表中一次选择多个选项，而组合框只能选择作单一选择；组合框允许用户对选择的值作编辑，而列表框只能选择不能编辑。

下拉列表框看起来像组合框，因为它有一个用来显示列表的下拉箭头。和组合框类似，在下拉列表框中一次只能选择一项；但和列表框相似的是，所选的值不能被编辑。下拉列表框的功能基本和列表框相同，除了用户一次只能作单一的选择，另外它所占的空间较少。

Visual FORTRAN 不支持用户画的列表框或组合框。列表框或组合框的创建只能由资源编辑器来进行。

1. 列表框

对于列表框和组合框，控件索引 DLG_NUMITEMS 决定了框中选项的个数。一旦个数确定以后，就可以通过为每个选项索引指定字符串来设置列表框选项的文本。例如：

```
LOGICAL retlog
retlog = DlgSet ( dlg, IDC_LISTBOX1, 3, DLG_NUMITEMS )
retlog = DlgSet ( dlg, IDC_LISTBOX1, "Moe", 1 )
retlog = DlgSet ( dlg, IDC_LISTBOX1, "Larry", 2 )
retlog = DlgSet ( dlg, IDC_LISTBOX1, "Curly", 3 )
```

上面的语句向列表框中放置了三个选项，每个列表框实体的初始值为空，在被设置后值才为非空。

可以在任何时候改变列表长度和选项，包括在反馈例程中改变。如果列表缩短，则实体的集合会被截断。如果列表拉长，则会添加空实体。在上面的例子中，可以拉长列表并定义新的选项：

```
retlog = DLGSET ( dlg, IDC_LISTBOX1, 4)
retlog = DLGSET ( dlg, IDC_LISTBOX1, "Shemp", 4)
```

因为列表框允许作多重选择，所以需要一种方法来决定哪些实体被选择。但用户选择了一个列表框选项时，会赋给选项一个等于被选项顺序的整数索引。通过读取选择索引可以测试哪些列表项被选择，直到读到一个为 0 的值。例如，在上面的列表框汇总，用户如果选择了 Moe 和 Curly，列表框的选择索引值如表 10.4 所示。

表 10.4　　一种选择索引的取值情况

选择索引	取值
1	1 (Moe)
2	3 (Curly)
3	0 (没有其它选项)

如果只有 Larry 被选择，则现在列表框的选择索引如表 10.5 所示。

表 10.5　　另一种选择索引的取值情况

选择索引	取值
1	2 (Larry)
2	0 (没有其它选项)

为了决定被选项，可以用 DLGGET 读取列表框的值，直到碰到 0 为止。例如：

```
INTEGER j, num, test
INTEGER, ALLOCATABLE :: values(:)
LOGICAL retlog
```

```
retlog = DLGGET (dlg, IDC_LISTBOX1, num, DLG_NUMITEMS)
ALLOCATE (values(num))
j = 1
test = -1
DO WHILE (test .NE. 0)
   retlog = DLGGET (dlg, IDC_LISTBOX1, values(j), j)
   test = values(j)
   j = j + 1
END DO
```

本例中，j是选择索引，values(j)按顺序保存着被选项（如果有的话）列表标号。

为了只读一个选项，或读一组中的第一个被选项，可以使用 DLG_STATE，因为对于一个列表框，DLG_STATE 保存了第一个被选项（如果有的话）的字符串。例如：

```
! Get the string for the first selected item.
retlog = DLGGET (dlg, IDC_LISTBOX1, str, DLG_STATE)
```

此外，可以先获得被选项的列表编号，然后再获得和该项相关的字符串：

```
INTEGER value
CHARACTER(256) str
  ! Get the list number of the first selected item.
  retlog = DLGGET (dlg, IDC_LISTBOX1, value, 1)
  ! Get the string for that item.
  retlog = DLGGET (dlg, IDC_LISTBOX1, str, value)
```

在这些例子中，如果用户没有作出选择，str 就是一个空字符串。

资源编辑器里的 Properties/Styles 中，列表框可以指定为排序或不排序。缺省时为排序，这时列表框中的项按以 A 开始的字母顺序排列，大写比小写优先。如果列表框是排序的，在每个反馈被调用前或在 DLGMODAL 返回时，列表框中的项会按字母顺序排列。

2. 组合框

组合框是列表框和编辑框的结合。用户可以从列表中选择并在编辑框中显示，或直接在编辑框中输入文本。所有的对话框都认为用户输入为字符串，用户的应用程序必须解释这些字符串代表的数据。

因为输入可以有两种方式（从列表框中选择或直接输入到编辑框），所以需要用 DLGSETSUB 为组合框登记两个反馈类型。这些反馈类型是要处理被用户选择的新列表的 dlg_selchange，以及处理用户直接输入的文本的 dlg_update。例如：

```
retlog = DlgSetSub( dlg, IDC_COMBO1, UpdateCombo, dlg_selchange )
```

```
retlog = DlgSetSub( dlg, IDC_COMBO1, UpdateCombo, dlg_update )
```

组合框列表的创建和列表框的列表是相同的，但用户一次只能从组合框中选择一个项目。当用户从列表中选择了一个选项后，Windows 自动把该项放到组合框的编辑框里。所以，没有必要也没有这种机制来获得被选项的编号。

如果用户直接向组合框的编辑框中输入内容，Windows 也会自动显示。可以用下面的语句获得选择的或直接输入的字符串：

```
! Returns the character string of the selected item or Edit box entry as str.
retlog = DLGGET (dlg, IDC_COMBO1, str)
```

在组合框属性风格（Styles）选项卡中有三种组合框类型选择： 简单（Simple）组合框，下拉列表（Drop list）组合框，和下拉（Drop-down）组合框，如图 10.17 所示。简单组合框和下拉列表组合框基本相同，除了简单组合框始终在列表中显示选项而下拉列表有一个下拉按钮并在下拉列表中显示选项。下拉组合框介于简单组合框和下拉列表组合框之间。

图 10.17 组合框风格

3．下拉列表框

为创建下拉（Drop-down）列表框，可以在控件工具条中选择组合框并放到对话框中，双击组合框打开属性对话框，在风格（Styles）选项卡中选择 Drop-down 作为控件类型。

下拉列表框有一个用来显示列表的下拉箭头。和组合框类似，在下拉列表框中一次只能选择一项；但和列表框相似的是，所选的值不能被编辑。下拉列表框的功能基本和列表框相同，除了用户一次只能作单一的选择，另外它所占的空间较少。

下拉列表框和组合框有相同的控件索引，并增加了另外一个设置或返回项目列表编号的 INTEGER 索引。例如：

```
INTEGER num
  ! Returns index of the selected item.
```

```
retlog = DLGGET (dlg, IDC_DROPDOWN1, num, DLG_STATE)
```

10.5.7　滚动条

使用滚动条可以通过把滑块上下或左右滑动来确定输入。应用程序设置滚动条的范围，然后把滑块的位置解释为一个数值。如果希望显示这个数值，可以把该数值作为文本送入静态文本或编辑框控件。

滚动条的范围始终从 1 开始，缺省的上限值为 100，可以用 DLGSET 或 DLGSETINT 设置控件索引 DLG_RANGE 来设置范围的上限。例如：

```
LOGICAL retlog
retlog = DLGSET (dlg, IDC_SCROLLBAR1, 212, DLG_RANGE)
```

可以用 DLGGET 或 DLGGETINT 获得控件索引 DLG_POSITION 进而获得滑块的位置。例如：

```
INTEGER slide_position
retlog = DLGGET (dlg, IDC_SCROLLBAR1, slide_position, DLG_POSITION)
```

点击滚动条箭头按钮引起的滑块的增量或减量由控件索引 DLG_SMALLSTEP 设置，而点击滑块旁边的空白区域引起的滑块的增量或减量由控件索引 DLG_BIGSTEP 设置。例如：

```
retlog = DLGSET (dlg, IDC_SCROLLBAR1, 4, DLG_SMALLSTEP)
retlog = DLGSET (dlg, IDC_SCROLLBAR1, 20, DLG_BIGSTEP)
```

10.5.8　设置返回值和退出

用户选择确定（或 OK）或取消（CANCEL）按钮时，对话框过程退出并且对话框被关闭。DLGMODAL 返回引起对话框退出的控件名称（在包括文件中和一个整型标识符联合），例如 IDOK 或 IDCANCEL。如果希望在选择确定（或 OK）或取消（CANCEL）按钮之外的某种情况退出对话框，需要在反馈例程中包括对话框子例程 DLGEXIT 的调用。例如：

```
SUBROUTINE EXITSUB (dlg, exit_button_id, callbacktype)
USE DFLOGM
TYPE (DIALOG) dlg
INTEGER exit_button_id, callbacktype
```

```
...
CALL DLGEXIT (dlg)
```

DLGEXIT 的唯一参数是 dialog 派生类型。对话框在 DLGEXIT 返回控件到对话框管理之后退出。也就是说如果在反馈例程中的 DLGEXIT 后有其它语句，这些语句将被执行，同时在对话框退出之前反馈例程返回。

如果希望 DLGMODAL 返回除了引起对话框退出的控件名之外的值（或如果 DLGMODAL 打开对话框失败返回-1），可以用子例程 DLGSETRETURN 指定自己的返回值。例如：

```
TYPE (DIALOG) dlg
INTEGER altreturn
...
altreturn = 485
CALL DLGSETRETURN (dlg, altreturn)
CALL DLGEXIT(dlg)
```

为了避免和缺省的失败情况混淆，最好不要使值为-1。

如果希望用户能从系统菜单中或通过按【ESC】键关闭对话框，需要一个 ID 为 IDCANCEL 的控件。如果对话框中没有 ID 为 IDCANCEL 的控件，关闭命令将被忽略，对话框也不能用这样的方式关闭。

如果希望使系统能够或通过按【ESC】键关闭对话框，但不想使用取消按钮，可以在对话框中添加 ID 为 IDCANCEL 的按钮并在按钮属性中去掉可见属性。按下【ESC】键会激活缺省点，即取消按钮的反馈例程并关闭对话框。

第十一章　混合语言编程

使用混合语言编程具有很多优势，但是代价是要考虑更多的语言兼容方面的问题。

11.1　概述

由两种以上语言写成的源代码建立程序的过程称为混合语言编程，混合语言编程允许用户的以下操作：

- 调用已经存在的用其它语言编写的代码
- 使用以某种语言很难实现的过程
- 获得处理速度上的优势

混合语言编程可以在 32 位的 Visual FORTRAN, Visual C/C++, Visual Basic 和 MASM 之间实现。在 Win32 中使用混合语言编程与在 16 位操作环境中有所不同，在很多方面使用 Win32 进行混合语言编程显得更为简单。

为了正确创建混合语言程序，必须为命名变量和过程、堆栈使用和在不同语言编写的例程之间传递的参数等等建立一套规则。这些规则统称为调用约定。

调用约定包括：

- 堆栈考虑

（1）　一个例程是否接受可变或固定数目的参数？

（2）　调用之后哪个例程清除堆栈？

- 命名约定

（3）　大小写是否区分？

（4）　名称是否被修饰（像在 Visual C++中）？

- 参数传递协议

（5）　数传递是通过值还是通过引用？

（6）　语言之间的等价数据类型和数据结构是什么？

本章主要提供用 FORTRAN, C, Visual C++, Visual Basic 和 *x*86 汇编语言进行混合语言编程时关于调用约定的一些知识。这一章里，“例程”是一个一般的说法，可以用来指不同语言中的函数、子例程和过程。

11.2　混合语言问题

混合语言编程包括从一种语言编写的程序中调用以另一种语言编写的函数、过程或子

例程。例如，一个 FORTRAN 主程序可能需要执行一个由汇编语言单独编写的任务，或者需要调用已经存在的动态链接库，或者系统过程。

由于 Visual FORTRAN, Visual C/C++, Visual Basic 和 MASM 中的函数、子例程和过程的实现基本类似，所以它们之间可以进行混合语言编程。例如，C 语言中主程序可以调用外部 void 函数，对应的在 FORTRAN 语言中是以子例程来实现的。每种语言不同例程的相互对应关系如表 11.1 所示。

表 11.1　　不同语言调用例程的等价形式

语　言	有返回值调用	无返回值调用
FORTRAN	函数（FUNCTION ）	子例程（SUBROUTINE ）
C 和 Visual C++	函数（function）	空函数（(void) function）
Visual Basic	函数（Function）	子函数（Sub）
汇编语言	过程（Procedure）	过程（Procedure）

但是这些语言实现例程的方式之间还有一些重要的区别，任何两种语言之间的参数传递、命名约定和其它接口问题必须仔细加以考虑以避免程序调用失败和其它不可知结果。

- FORTRAN 和汇编语言

  汇编语言程序由于不需要像高级语言那样作初始化，所以它的文件小且执行速度快。并且汇编语言允许对通常高级语言无法访问的硬件进行访问。在 FORTRAN/汇编语言混合程序中，在 FORTRAN 的主程序里提供汇编代码访问的 FORTRAN 高级过程和库函数，于是可以获得最大的效率和最快的速度。主程序也可以是汇编程序。

- FORTRAN 和 Visual Basic

  FORTRAN 和 32 位的 Visual Basic 的混合既可以使用 Visual Basic 中十分简单的搭积木式的界面设计又可以用 FORTRAN 程序进行计算，尤其是浮点数学运算。在 FORTRAN/Visual Basic 程序中，主程序必须是 Visual Basic 程序，不能从 FORTRAN 中调用 Basic 程序。

- FORTRAN 和 C（或 C++）

  一般来说，FORTRAN 和 C（或 C++）允许互相调用，所以主程序可以既可以是 FORTRAN 程序也可以是 C（或 C++）程序。

  为了在同样的 Developer Studio 环境中使用多种语言，必须使用同样版本的 Developer Studio。

11.2.1 调整混合语言中的调用约定

调用约定决定了程序如何调用一个例程，参数如何传递和例程如何命名。在单语言程序中，调用约定几乎永远是正确的，因为对于所有例程都有同一个缺省约定，并且有接口块的头文件或 FORTRAN 模块文件保证了在主调和被调例程之间的连贯性。

在混合语言程序中，不同的语言不能共用同样的头文件。如果用不同的调用约定连接 FORTRAN 和 C，直到运行时到无效调用时错误才会出现。在执行过程中，无效调用会引

起不确定的结果或致命错误，例如内存/堆栈的崩溃。所以，每个混合语言的调用都要仔细检查调用约定。

这里说明的调用约定只适用于外部过程，用户不能从包含内部过程的程序单元中调用这些内部过程。

调用约定在以下几个方面影响编程：

（1）主调例程使用调用约定来决定向其它例程传递参数的顺序；被调例程使用调用约定来决定接受传递给它的参数的顺序。在 FORTRAN 里，可以在混合语言界面中用 INTERFACE 语句或在函数、数据声明中指定这些约定。32 位的 Visual C/C++和 FORTRAN 传递参数的顺序都是从左到右。

（2）主调例程和被调例程使用调用约定来决定它们之中哪一个是在被调例程结束后负责调节堆栈（为了清除参数）。

（3）主调例程和被调例程使用命名约定来选择传递可变参数数目的选项。

（4）主调例程和被调例程使用命名约定来以值传递参数（值传递）或以引用传递参数（地址传递）。各自的 FORTRAN 参数也可以指定为 ATTRIBUTES 的 VALUE 或 REFERENCE。

（5）主调例程和被调例程使用命名约定来建立过程名称的命名约定。用户可以用 ALIAS 指令（或 ATTRIBUTES 的 ALIAS 选项）建立希望的任何名称而不管其 FORTRAN 名称如何。C 对大小写是敏感的，而 FORTRAN 不敏感，所以这一点很有用。

不同的 FORTRAN 调用约定可以通过声明 FORTRAN 过程拥有特定的属性来确定，例如，在 x86 系统中：

```
INTERFACE
  SUBROUTINE MY_SUB (I)
    !DEC$ ATTRIBUTES C, ALIAS:'_My_Sub' :: MY_SUB   ! x86 系统
    INTEGER I
  END SUBROUTINE MY_SUB
END INTERFACE
```

上面的代码声明了名为 MY_SUB、具有 C 属性且外部名称为_My_Sub（在 x86 系统上设置的 ALIAS 属性）的子例程。

在 Alpha 系统上，外部名称_My_Sub 开头的下划线被去掉，所以!DEC$ ATTRIBUTES 行应为：

```
!DEC$ ATTRIBUTES C, ALIAS:'My_Sub' :: MY_SUB  ! Alpha 系统
```

ATTRIBUTES 选项 C, STDCALL, REFERENCE, VALUE 和 VARYING 都会影响例程的调用约定。缺省情况下，FORTRAN 通过引用传递所有参数（除了字符串的隐含长度参

数)。如果使用了 C 或 STDCALL 选项，缺省设置被改为通过值来传递除了数组之外的所有数据。数组只能通过引用来传递参数。

表 11.2 汇总了大部分公共 FORTRAN 调用约定指令的影响。

表 11.2 调用约定的 ATTRIBUTES 选项

	缺省情况	C	STDCALL	C, REFERENCE	STDCALL, REFERENCE
			参数		
标量	引用	值	值	引用	引用
标量 [值]	值	值	值	值	值
标量 [引用]	引用	引用	引用	引用	引用
字符串	引用或 Len: 混合 或 Len:End	字符串(1:1)	字符串(1:1)	引用或 Len: 混合 或 Len:End	引用, 无 Len
字符串 [值]	错误	字符串(1:1)	字符串(1:1)	字符串(1:1)	字符串(1:1)
字符串 [引用]	引用或 Len: 混合 或无 Len	引用, 无 Len	引用, 无 Len	引用, 无 Len	引用, 无 Len
数组	引用	引用	引用	引用	引用
数组[值]	错误	错误	错误	错误	错误
数组[引用]	引用	引用	引用	引用	引用
派生类型	引用	值, 依赖大小	值, 依赖大小	引用	引用
派生类型 [值]	值, 依赖大小	值, 依赖大小	值, 依赖大小	值, 依赖大小	值, 依赖大小
派生类型 [引用]	引用	引用	引用	引用	引用
F90 指针	描述符	描述符	描述符	描述符	描述符
F90 指针 [值]	错误	错误	错误	错误	错误
F90 指针 [引用]	描述符	描述符	描述符	描述符	描述符
			过程名称		
后缀	@n(x86 系统)	无	@n(x86 系统)	无	@n(x86 系统)
大小写	大写	小写	小写	小写	小写
堆栈清除	被调过程	主调过程	被调过程	主调过程	被调过程

上表的术语含义如表 11.3 所示。

表 11.4 说明的是 FORTRAN 的 ATTRIBUTES 和其它语言调用约定相匹配的选项。

表 11.3　术语含义

[值]	指定 VALUE 属性
[引用]	指定引用属性
值	参数值被推入堆栈，所有的值都填补到相邻的 4 字节
引用	4 字节参数地址被推入堆栈
Len:混合 或 Len: End	对于特定字符串参数： Len: 混合…当设置/iface:mixed_str_len_arg 时应用。字符串的长度在字符串首地址之后被推入堆栈（通过值） Len: End…当设置/iface:nomixed_str_len_arg 时应用。字符串的长度在所有其它参数之后被推入堆栈（通过值）
Len: 混合或无 Len	对于某个字符串参数： Len: 混合…在设置/iface:mixed_str_len_arg 时应用。字符串的长度在字符串首地址之后被推入堆栈（通过值） 无 Len…在设置//iface:nomixed_str_len_arg 时应用。对于被调过程，字符串的长度不可知
无 Len	对于字符串参数，被调过程不能知道字符串的长度
String(1:1)	对于字符串参数，第一个字符转换为 INTEGER(4)并把值推入堆栈
错误	产生编译器错误
描述符	4 字节数组地址描述符
@n	在 x86 系统上，at 符号(@)后是参数列表要求的字节数（十进制）
依赖大小	派生类型参数，由下列参数传递的值指定： 1．由值传递的 1 至 4 字节参数 2．在两个寄存器中的 5 至 8 字节参数（两个参数） 3．通过引用传递临时存储地址提供值的多于 8 字节的参数
大写	过程的名称都是大写
小写	过程的名称都是小写
被调过程	被调过程负责在返回到主调过程之前从堆栈中去除参数
主调过程	主调过程负责在调用结束后从堆栈中去除参数

表 11.4　匹配的调用约定

其它语言调用约定	与 ATTRIBUTES 匹配的选项
Visual C/C++ cdecl (缺省)	C 选项
Visual C/C++ __stdcall	STDCALL 选项
Visual Basic	无
Visual Basic CDECL 关键字	C 选项
MASM C (在 PROTO 和 PROC 声明中)	C 选项
MASM STDCALL (在 PROTO 和 PROC 声明中)	STDCALL 选项

注意，ALIAS 选项可以被任何其它 FORTRAN 调用约定选项使用来保留混合情况的名称。

1. 调用约定中的堆栈考虑

在 C 调用约定中，调用例程始终在被调例程返回控制后立即调整堆栈。这样会引起代码有少许增加，因为恢复堆栈的代码必须在被调用的每一点退出。在 STDCALL 调用约定中，是被调过程控制堆栈。在被调过程中要存储的代码在堆栈中驻留，所以代码只需要出现一次。

但是，C 调用约定与可变数目参数一起调用。因为在 C 调用约定中由主调程序来清除堆栈，所以有可能编写使用可变数目参数的例程。因此，它有和帧相关的指针有相同的地址而不用考虑实际传递了多少参数。所以当主调例程控制堆栈时，它知道传递了多少参数，知道它们的大小和在堆栈中驻留的位置。它还可以跳过传递一个参数并仍然保留堆栈。

可以通过在用户对例程的接口中包含 ATTRIBUTES 的 C 和 VARYING 选项来以可变参数数目来调用例程。VARYING 选项使 FORTRAN 不必强迫在例程中参数数目的匹配。VARYING 选项对于有可选参数的内在 FORTRAN 90 例程不是必要的，其中参数顺序和/或关键字决定了出现哪个参数还是缺少哪个参数。

在 MASM 中，堆栈控制也是由过程声明的 C 或 STDCALL 约定设置的，但用户也可以编写 MASM 代码控制在任何希望的过程中的堆栈。另外，可以在 PROC 指令中的 USES 选项来自动存储和恢复某个寄存器。

2. FORTRAN/C 调用约定

在 C 和 Visual C++模块中，可以通过在函数原型或定义中使用__stdcall 关键字指定 STDCALL 调用约定。__stdcall 约定也被窗口过程和 API 使用。下面的 C 语言原型用 STDCALL 调用约定建立一个函数对子例程的调用：

```
extern void __stdcall FORTRAN_ROUTINE (int n);
```

除了改变 C 代码的调用约定之外，还可以使用 C 选项来调整 FORTRAN 源代码。这个选项是由 ATTRIBUTES 指令设置的。例如，下面的声明假设子例程以 C 调用约定被调用：

```
SUBROUTINE CALLED_FROM_C (A)
    !DEC$ ATTRIBUTES C :: CALLED_FROM_C
    INTEGER A
```

3. FORTRAN/Visual Basic 调用约定

以 Visual Basic 形式建立 FORTRAN 子例程和函数之后 FORTRAN 例程可以从一个 Basic 模块中被调用。FORTRAN 例程必须是从 Basic 调用的动态链接库（DLL）。

如果保留混合情况的名称（缺省时 FORTRAN 把所有名称转化为大写），需要调用约定的 ALIAS 指令（或 ATTRIBUTES 的 ALIAS 选项）。但以下两种情况需要区别对待：

（1） 如果传递的是可变数目的参数，在 FORTRAN 过程定义中需要 C 和 VARYING 选项，并且为了建立 C 调用和命名约定，在 Basic 中的 DECLARE 语句中需要 CDECL 关键字。

（2） 在传递字符参数时，FORTRAN 例程必须使用 ATTRIBUTES 的 STECALL 选项，所以它并不需要字符参数的隐含长度。因为 STDCALL 也是小写的 FORTRAN 名称，从 Visual Basic 程序中应该以小写来引用 FORTRAN 子程序的名称。

下面的 FORTRAN 和 Visual Basic 语句建立了可以被 Basic 调用的 FORTRAN 示例函

数：

```
!  建立 FORTRAN 函数的 FORTRAN 子程序
      INTERFACE
        DOUBLE PRECISION FUNCTION GetFVal (r1)
          !DEC$ ATTRIBUTES ALIAS:'GetFVal' :: GetFVal
          !DEC$ ATTRIBUTES VALUE :: r1
          REAL r1
        END FUNCTION
      END INTERFACE
'FORM.FRM Basic Form to establish FORTRAN function
Declare Function GetFVal Lib "C:\f90\FVAL.DLL" (ByVal r1 As Single) As Double
```

4. FORTRAN/MASM 调用约定

可以在 PROTO 和 PROC 指令中指定为 MASM 的调用约定。PROTO 和 PROC 指令中的 STDCALL 选项告诉过程使用 STDCALL 调用约定。PROTO 和 PROC 指令中的 C 选项告诉过程使用 C 调用约定。PROC 指令中的 USES 选项指定了在被调 MASM 例程中用来存储和恢复的寄存器。PROTO 和 PROC 指令的 VARAG 选项指定过程允许使用可变数目参数。

下面的 FORTRAN 和 MASM 语句用 STDCALL 调用约定建立了可以从 Visual FORTRAN 中调用的 MASM 函数：

```
!FORTRAN STDCALL 原型.
     INTERFACE
       INTEGER FUNCTION forfunc(I1, I2)
       !DEC$ ATTRIBUTES STDCALL :: forfunc
       INTEGER I1
       INTEGER(2) I2
     END INTERFACE
     WRITE (*,*) forfunc(I1,I2)

;MASM STDCALL Prototype
        .MODEL FLAT, STDCALL
forfunc PROTO STDCALL, forint: SDWORD, shorti: SWORD
        .CODE
forfunc PROC STDCALL, forint: SDWORD, shorti: SWORD
        ...
forfunc ENDP END
```

下面的 FORTRAN 和 MASM 语句用 C 调用约定建立了可以从 FORTRAN 中调用的 MASM 函数：

```
!FORTRAN C 原型
      INTERFACE
         INTEGER FUNCTION Forfunc (I1, I2)
         !DEC$ATTRIBUTES C, ALIAS:'Forfunc' :: Forfunc
         INTEGER I1
         INTEGER(2) I2
      END INTERFACE
      WRITE(*,*) Forfunc (I1, I2)
      END

; MASM C PROTOTYPE
         .MODEL FLAT, C
Forfunc PROTO C, forint:SDWORD, shorti:SWORD
         .CODE
Forfunc PROC C, forint:SDWORD, shorti:SWORD
         ...
Forfunc ENDP END
```

11.2.2 调整混合语言编程中的命名约定

C 和 STDCALL 既可以决定调用约定也可以决定命名约定。调用约定说明的是参数如何移动和存储，而命名移动说明符号名称在.OBJ 文件中如何替换。名称不仅仅是在同一个程序中不同的部分，也是外部例程之间共享外部数据的问题。符号名称（例如子例程的名称）确定了一个内存地址，这个地址必须在所有的调用例程中保持一致。参数名称（过程定义中给被传递的变量的名称）永远不会受到影响。替换名称是因为大小写敏感问题（在 C, Visual Basic 和 MASM 中），缺少大小写敏感问题（在 FORTRAN 中），名称修饰问题（在 Visual C++中）或其它问题。如果命名约定不统一，程序就不能成功地连接，并且出现“unresolved external”的错误。

1. Visual C/C++和 Visual Basic 命名约定

和 FORTRAN 不同，缺省时 Visual C/C++和 Visual Basic 保留符号表中对大小写的敏感，这个差别需要注意。不过，可以使用 FORTRAN 中 ATTRIBUTES 的 ALIAS 选项来解决这项差异，或保留混合大小写名称，或由 FORTRAN 的缺省命名把所有的自动名称约定覆盖为大写，或由 FORTRAN 的 STDCALL 和 C 命名约定把所有的自动名称约定为小写。

2. MASM 命名约定

对于 MASM（Microsoft Assembler, *x86* 系统），如果 CASEMAP 选项不存在，在 PROC 和 PROTO 语句中指定 C 或 STDCALL 选项会保留大小写敏感。MASM 的 OPTION CASEMAP 指令（相应的命令行中为/C）也设置了大小写敏感和在 PROC 和 PROTO 语句中指定的覆盖命名约定。CASEMAP: NONE（等价于/Cx）保留了 PUBLIC, COMM, EXTERNDEF, EXTERN, PROTO 和 PROC 声明中的标识符的大小写。CASEMAP: NOTPUBLIC（等价于/Cp）保留了所有隐含标识符的大小写。CASEMAP: ALL（等价于/Cu）把所有标识符转换成大写。

3．命名约定概要

表 11.5 汇总了 FORTRAN, Visual C/C++, Visual Basic 和 MASM 如何处理过程名称的方式。注意对于 MASM，如果使用了 CASEMAP: ALL 选项表中的内容就不再适合。

表 11.5 中：

（1）　开头的下划线（例如*_name*）只能用于 x86 系统（不能在 Alpha 系统上）。

（2）　*@n* 代表的是堆栈地址，以十进制表示，也是只能用于 x86 系统（不能在 Alpha 系统上）。

表 11.5　　命名约定

语　言	属　性	名称被转化为	.OBJ 文件中名称的大小写
FORTRAN	cDEC$ ATTRIBUTES C	*_name*	全部小写
	cDEC$ ATTRIBUTES STDCALL	*_name@n*	全部小写
	缺省	*_name@n*	全部大写
C	cdecl (缺省)	*_name*	保留混合大小写
	__stdcall	*_name@n*	保留混合大小写
Visual C++	缺省	*_name@@decoration*	保留混合大小写
Visual Basic	缺省	*_name@n*	保留混合大小写
MASM	C (在 PROTO 和 PROC 声明中)	*_name*	保留混合大小写
	STDCALL (在 PROTO 和 PROC 声明中)	*_name@n*	保留混合大小写

例如，假设一个在 C 中声明的函数为：

```
extern int __stdcall Sum_Up( int a, int b, int c );
```

每个整数占 4 字节，于是 x86 系统上.OBJ 文件中的符号名称为_Sum_Up@12。

在 Alpha 系统上，.OBJ 文件中的符号名称为 Sum_Up。

4．全部大写名称

如果调用了使用 FORTRAN 缺省情况且不能重新编译的 FORTRAN 代码的 FORTRAN 例程，在 C 和 Visual Basic 中，要完成调用必须使用全部为大写的名称；在 MASM 中，要么使用全部为大写的名称，要么设置 OPTION CASEMAP 指令为 ALL，它将把所有描述符转化为大写。在 C 代码中使用__stdcall 约定或在 MASM 的 PROTO 和 PROC 声明中使用 STDCALL 是不够的，因为__stdcall 和 STDCALL 始终保持大小写状态。FORTRAN

缺省情况下产生的是全部为大写的名称，C 和 MASM 代码必须与之匹配。

例如，下面这些原型建立了 FORTRAN 函数 FFARCTAN(angle)，其中参数 angle 具有 **ATTRIBUTES** VALUE 属性：

在 C 语言中，

```
extern float __stdcall FFARCTAN( float angle );
```

在 Visual Basic 中，

```
Declare Function FFARCTAN Lib "C:\f90ps\FBAS.DLL" (ByVal angle As Single) As Single
```

在 MASM 中，

```
.MODEL FLAT, STDCALL
FFARCTAN PROTO STDCALL, angle: PTR REAL4
...
FFARCTAN PROC STDCALL, angle: PTR REAL4
```

5．全部小写名称

如果在 C 或 MASM 中例程的名称都是小写，那么当在 FORTRAN 声明中使用 C 或 STDCALL 选项时命名约定会自动正确。在 FORTRAN 源代码中大小写都可以使用，包括大小写混合，因为 C 或 STDCALL 选项把所有名称都改为小写。STDCALL 和 FORTRAN 缺省行为不同。从 FORTRAN 里不能直接调用 Visual Basic 例程，所以 Basic 例程名称不会被转化。

6．混合大小写名称

如果在 C 或 MASM 中出现例程的混合大小写名称且用户不能改变名称，可以使用 FORTRAN 的 ATTRIBUTES ALIAS 属性来解决命名冲突。在这种情况下需要 ALIAS，因为 FORTRAN 不会保留混合大小写名称。

为了使用 ALIAS 选项，在名称出现在.OBJ 文件中要把名称放在引号里。下面是在 x86 系统中引用 C 函数 My_Proc 的例子：

```
!DEC$ ATTRIBUTES ALIAS:'_My_Proc' :: My_Proc
```

在 Alpha 系统上，应该写为：

```
!DEC$ ATTRIBUTES ALIAS:'My_Proc' :: My_Proc
```

7．FORTRAN 模块名称

FORTRAN 模块实体（数据和过程）有和其它外部实体不同的外部名称。模块名称使

用的约定为：

_MODULENAME_mp_ENTITY [@stacksize]

MODULENAME 是模块的名称并且缺省时都是大写。ENTITY 是 MODULENAME 中包含的模块过程或模块数据的名称。_mp_是在模块和实体之间的分隔符，它始终是小写。

例如：

```
MODULE mymod
   INTEGER a
CONTAINS
   SUBROUTINE b (j)
      INTEGER j
   END SUBROUTINE
END MODULE
```

在 x86 系统上被编译过的.OBJ 文件中被定义的符号里的结果：

```
_MYMOD_mp_A
_MYMOD_mp_B@4
```

或在 Alpha 系统上：

```
MYMOD_mp_A
MYMOD_mp_B
```

编译器选项可以影响模块数据和过程的命名。

注意：除了 ALIAS 之外，ATTRIBUTES 选项并不影响模块名称，模块的名称仍然保持大写。

表 11.6 说明的是每个 ATTRIBUTES 选项如何影响上面示例模块中的子例程。

表 11.6　　FORTRAN 模块中 ATTRIBUTES 选项的影响

给例程'b'的 ATTRIBUTES 属性	x86 系统上的.OBJ 文件中的过程名称	系统上的.OBJ 文件中的过程名称
无	_MYMOD_mp_B@4	MYMOD_mp_B
C	_MYMOD_mp_b	MYMOD_mp_b
STDCALL	_MYMOD_mp_b@4	MYMOD_mp_b
ALIAS	覆盖所有其它名称，名称以别名给出	覆盖所有其它名称，名称以别名给出
VARYING	对名称无影响	对名称无影响

用户可以编写代码来从其它语言中调用 FORTRAN 模块或访问模块数据。考虑到其它命名和调用约定，两种语言之间的模块名称必须匹配。一般来说，这意味着在 FORTRAN 中使用 C 或 STDCALL 约定，并且如果在其它语言中定义模块时中使用 ALIAS 选项来和 FORTRAN 中的名称匹配。

8．Visual C++名称

Visual C++使用和 C 相同的调用约定和参数传递技术，但它们的命名约定不同，因为 Visual C++有对外部符号的修饰。语法 extern "C"让 Visual C++ 丢掉名称修饰而使 Visual C++模块和其它语言共享数据和例程成为可能。

下面的例子用 C 命名约定声明 prn 为该外部函数，在 Visual C++源代码中出现：

```
extern "C" { void prn(); }
```

为了调用 FORTRAN 写的函数，在 C 语言中声明该函数并使用"C"连接指定。例如，在 Visual C++中调用 FORTRAN 函数 FACT，把它声明为：

```
extern "C" { int __stdcall FACT( int n ); }
```

语法 extern "C"可以用来调整 Visual C++中对其它语言的调用，或用来改变被调 Visual C++例程的调用约定。但是语法 extern "C"只能用在 Visual C++内部，如果 Visual C++代码没有使用 extern "C"并且无法改动，只能通过确定名称修饰和从其它语言产生来调用 Visual C++例程了。

Extern "C"的使用有以下限制：

（1） 不能用 extern "C"声明成员函数。

（2） 只能为重载函数的一个实例指定 extern "C"，所有其它重载函数的实例都有 Visual C++连接。

11.2.3 定义 FORTRAN 中过程的原型

要告诉 FORTRAN 编译器想对外部引用使用什么语言约定时，应该在 FORTRAN 源代码中定义一个原型（接口块）。接口块通过 INTERFACE 语句引入，INTERFACE 语句的一般形式为：

INTERFACE
　例程语句
　[例程 **ATTRIBUTE** 选项]
　[参数 **ATTRIBUTE** 选项]
　正式参数声明
　END 例程名称
END INTERFACE

例程语句定义了一个 FUNCTION 或 SUBROUTINE。是定义 FUNCTION 还是定义 SUBROUTINE 依赖于是否要返回一个值。可选的“例程 **ATTRIBUTE** 选项”（例如 C 和 STDCALL）决定了在原型语句中例程的调用、命名和参数传递约定。可选的“参数 **ATTRIBUTE** 选项”（例如 VALUE 和 REFERENCE）是隶属于各个参数的属性。“正式参数声明”是 FORTRAN 数据类型声明。注意，同一个 INTERFACE 块可以指定多个过程。

例如，假设要调用具有下面原型的 C 函数：

```
extern void My_Proc (int I);
```

在 x86 系统上，该函数的 FORTRAN 调用应该用下面的 INTERFACE 块声明：

```
INTERFACE
  SUBROUTINE my_Proc (I)
   !DEC$ ATTRIBUTES C, ALIAS:'_My_Proc' :: my_Proc
   INTEGER I
  END SUBROUTINE my_Proc
END INTERFACE
```

应该注意：

（1）　在 Alpha 系统上，开头的下划线被省略，即 Alpha 系统上!DEC$ ATTRIBUTES 这行包括：

```
!DEC$ ATTRIBUTES C, ALIAS:'My_Proc' :: my_Proc
```

（2）　除了在 ALIAS 字符串中之外，FORTRAN 程序里 My_Proc 的大小写是无关紧要的。

11.3　混合语言编程中的数据交换和访问

在混合语言例程中可以使用多种方法实现数据共享，这些方法是：

- 在混合语言编程中传递参数
- 在混合语言编程中使用模块
- 在混合语言编程中使用公共外部数据

一般来说，如果有很多参数或有很多参数类型，可以考虑使用模块或外部数据声明。

11.3.1 在混合语言编程中传递参数

用户可以像在每种单独的语言中的参数列表（例如 CALL MYSUB(a,b,c)中的参数列表 a,b 和 c）一样通过调用参数列表在 FORTRAN 与 C, Visual C++, Visual Basic 与 MASM 之间传递数据。传递独立的参数有两种方法：

（1） 通过值，这种方法传递参数的值。

（2） 通过引用，这种方法传递参数的地址（在 FORTRAN, Visual Basic, C 和 Visual C++中，所有 5.0 版本在 x86 和 Alpha 系统上地址都是 4 字节）。

应该保证的是，对于每个调用，主调程序和被调例程在每个参数的传递方面应该一致。否则，被调例程会接受无效数据。

传递参数的 FORTRAN 技术随着指定的调用约定的不同而改变。缺省时 FORTRAN 通过引用来传递所有参数（除了字符串的隐含长度之外）。如果使用了 ATTRIBUTES 的 C 或 STDCALL 选项，则除了数组之外所有参数都是通过值来传递的。如果过程既有 REFERENCE 选项又有 C 或 STDCALL 选项，所有参数在缺省时都是通过引用传递的。

在 FORTRAN 中，除了用 C 和 STDCALL 调用约定选项建立的参数传递之外，还可以指定参数选项 VALUE 和 REFERENCE 来由值或引用传递参数。在混合语言编程中，显式地指定传递技术要比依赖缺省设置好。

值得注意的是，除了 ATTRIBUTES，编译选项/iface 也建立了一些缺省参数传递约定（例如字符串的隐含长度）。

后面是一些 C, Visual Basic 和 MASM 通过引用或值进行参数传递的例子。它们都是下面的 FORTRAN 子例程 TESTPROC 的接口。TESTPROC 的定义声明了参数如何传递。REFERENCE 选项在本例中并不是必要的，但使用它使参数的调用约定更加明显：

```
SUBROUTINE TESTPROC( VALPARM, REFPARM )
   !DEC$ ATTRIBUTES VALUE :: VALPARM
   !DEC$ ATTRIBUTES REFERENCE :: REFPARM
   INTEGER VALPARM
   INTEGER REFPARM
END
```

1. FORTRAN/C 通过值和引用传递参数的例子

在 C 和 Visual C++中，除了数组，所有的参数都是通过值来传递的。数组是通过对其首地址的引用传递的。和 FORTRAN 不同，C 和 Visual C++没有影响被传递的各个参数调用约定的指令。为了通过引用传递非数组的 C 数据，必须给它传递一个指针。下面的 C 声明建立了一个对上面 FORTRAN 子例程的调用：

```
extern void __stdcall TESTPROC( int ValParm, int *RefParm );
```

2. FORTRAN/Visual Basic 通过值和引用传递参数的例子

在 Visual Basic 中，缺省时参数是通过引用传递的。为了通过值传递参数，可以在 DECLARE 语句中的参数之前使用关键字 BYVAL，例如：

```
Declare Sub TESTPROC Lib "C:\f90\TESTPROC.DLL"

    (ByVal Valparm As Long, Refparm As Long)
```

3. FORTRAN/MASM 通过值和引用传递参数的例子

在 MASM 中，缺省时参数是通过值传递的。通过引用传递的参数由 PROTO 和 PROC 指定为 PTR 选项。例如：

```
TESTPROC PROTO STDCALL, valparm: SDWORD, refparm: PTR SDWORD
```

为了通过值传递参数，应该使用变量的值，例如：

```
mov eax, valparm ; 装载参数值
```

这条语句把变量 valparm 的值放入 EAX 寄存器。

为了通过引用传递参数，应该使用变量的地址，例如：

```
mov ecx, refparm ; 装载参数地址
mov eax, [ecx]   ; 装载参数值
```

这两条语句把变量 valparm 的值放入 EAX 寄存器。

表 11.7 汇总了如何通过值和引用传递参数的方法。C 语言中数组名称和其首地址是等价的，因为数组正常就是通过引用传递的。可以指定过程的 REFERENCE 属性，而不仅仅是各个参数。

表 11.7　　通过引用和值传递参数

语　言	ATTRIBUTE	参数类型	引用传递	值传递
FORTRAN	缺省	标量和派生类型	缺省	VALUE 选项
	C 或 STDCALL 选项	标量和派生类型	REFERENCE 选项	缺省
	缺省	数组	缺省	不能值传递
	C 或 STDCALL 选项	数组	缺省	不能值传递
Visual C/C++		非数组	Pointer argument_name	缺省
		数组	缺省	Struct{type} array_name
Visual Basic		所有类型	缺省	ByVal
汇编器 (x86) MASM		所有类型	PTR	缺省

FORTRAN 90 指针和字符串的构造和其它参数不同，所以上表没有给出在 Visual FORTRAN 中这两种参数的传递。缺省时，FORTRAN 和字符串长度一起通过引用来传递字符串，字符串长度的放置由设置的编译器选项：是/iface:mixed_str_len_arg（紧接着在字符串首地址之后）还是/iface:nomixed_str_len_arg（在所有参数之后）。

FORTRAN 90 数组指针和迟形数组通过长度数组描述符的地址来传递。

11.3.2 在混合语言编程中使用模块

模块是和 C 交换大批变量的最简单的方法，因为 Visual FORTRAN 模块可以被 Visual C/C++直接访问。下面是在 FORTRAN 中声明一个模块，然后从 C 中访问它的数据的例子：

```
! F90 模块定义
      MODULE EXAMP
         REAL A(3)
         INTEGER I1, I2
         CHARACTER(80) LINE
         TYPE MYDATA
            INTEGER N
            CHARACTER(30) INFO
         END TYPE MYDATA
      END MODULE EXAMP

\* 访问模块数据的 C 代码*\
extern float EXAMP_mp_A[3];
extern int EXAMP_mp_I1, EXAMP_mp_I2;
extern char EXAMP_mp_LINE[80];
extern struct {
           int N;
           char INFO[30];
} EXAMP_mp_MYDATA;
```

也可以在 C 中定义一个模块过程，并通过使用 ALIAS 定义这个过程为一个 FORTRAN 模块的一部分：

```
// C 过程
void pythagoras (float a, float b, float *c)
{
    *c = (float) sqrt(a*a + b*b);
}
```

```
! 包含过程的 FORTRAN 90 模块
      MODULE CPROC
         INTERFACE
            SUBROUTINE PYTHAGORAS (a, b, res)
             !DEC$ ATTRIBUTES C :: PYTHAGORAS
             !DEC$ ATTRIBUTES REFERENCE :: res
! 由于它自己的属性，res 通过引用传递
! 覆盖子例程的 C 属性
                REAL a, b, res
! a 和 b 缺省时有 VALUE 属性，因为子例程有 C 属性
             END SUBROUTINE
         END INTERFACE
      END MODULE
```

11.3.3　在混合语言编程中使用公共外部数据

公共外部数据结构包括 FORTRAN 公共块和定义为全局或外部的 C 结构和变量。指定这些数据创建都创建外部变量，对于定义它们的例程之外的例程也是可访问的。

这一小节只对 FORTRAN/C 和 FORTRAN/MASM 混合语言程序适用，因为与 Visual Basic 是无法共享公共数据的，必须在 Visual Basic 和 FORTRAN 之间把所有数据都作为参数传递。

外部变量对大小写是敏感的，所以不同语言之间的大小写必须匹配。

1．使用全局变量

在 FORTRAN 和 C 或 MASM 之间，通过在一种语言声明一个变量为全局（或 COMMON）可以实现变量共享。Visual Basic 既不能访问其它语言的全局数据也不能共享自己的。如前所述，在 FORTRAN/Basic 程序中变量必须通过参数传递。

在 FORTRAN 中，可以通过使用 ATTRIBUTES 的 EXTERN 选项使一个变量能以全局参数被访问，例如：

```
!DEC$ ATTRIBUTES C, EXTERN :: idata
INTEGER idata (20)
```

EXTERN 告诉编译器这个变量在另一个源文件中被定义并被声明为全局的。如果 FORTRAN 用 EXTERN 声明变量为外部的，和它共享该变量的语言必须也声明该变量为全局的。

在 C 中，全局变量声明语句为：

```
int idata[20]; // 声明为全局变量（不在任何函数之内）
```

MASM 中，声明参数为全局（PUBLIC）的语法为：

```
PUBLIC [langtype] name
```

其中，name 是要被引用的全局变量的名称，可选的 langtype 是 STDCALL 或 C。选项 langtype 如果存在会覆盖在.MODEL 指令中指定的调用约定。

相反，FORTRAN 可以声明变量为全局的（COMMON），其它语言可以把它作为外部变量进行引用：

```
!FORTRAN 声明 PI 为全局变量
 REAL PI
 COMMON /PI/ PI ! 公共块和变量的名称相同
```

在 C 中，该变量被作为外部变量引用：

```
//对 PI 进行外部引用的 C 代码
extern float PI;
```

注意，C 引用的全局名称是 FORTRAN 公共块的名称而不是公共块中变量的名称。因此，不能使用空的公共块使数据可以被 C 和 FORTRAN 访问。在上例中，公共块和变量有相同的名称，这样有助于保留两种语言中变量的轨迹。显然，如果一个公共块包含多个变量就不能都具有公共块的名称。

MASM 也可以访问带有 EXTERN 指令的 FORTRAN 全局（COMMON）参数，语法为：

```
EXTERN [langtype] name
```

其中 name 是要被引用的全局变量的名称，可选的 langtype 是 STDCALL 或 C。

2. 使用 FORTRAN 公共块和 C 结构

为了能从 FORTRAN 公共块和 C 结构中相互引用，必须考虑公共块和结构在内存中存储成员变量时方式的差别。

在内存中，FORTRAN 把公共块变量尽量放到一起，规则是：

（1） 公共块中一个单独的 BYTE, INTEGER(1), LOGICAL(1)或 CHARACTER 变量紧接着它前面的变量或数组存储。

（2） 所有其它单个变量在紧接它前面变量或数组的偶地址存储。

（3） 所有数组变量在紧接它前面变量或数组的偶地址存储，CHARACTER 数组除外，它紧接着它前面的变量或数组存储。

（4） 全部的公共块开始于一个 4 字节排列地址。

因为存在这些存储规则，用户必须考虑 C 结构元素和 FORTRAN 公共块元素的结合以保证既能使所有的变量类型完全等价又能使种别（如果只使用 4 字节和 8 字节可以使这一点简化）等价。或者通过在 C 结构周围使用 C 包装编译指示使 C 数据按 FORTRAN 的格式进行包装，例如：

```
#pragma pack(2)
struct {
        int N;
        char INFO[30];
} examp;
#pragma pack()
```

为了恢复初始包装，必须在结构结尾加上#pragma pack( )。注意，通过适当的命名，FORTRAN 模块数据可以和 C 结构直接共享数据。

一旦解决了格式的问题，可以使 C 访问整个公共块或公共块集合。同样，也可以把 FORTRAN 公共块的各个成员以参数列表传递，就像等待其它数据项一样。

3. 直接访问公共块和 C 结构

通过以合适区域定义外部 C 结构可以从 C 中直接访问 FORTRAN 公共块，并要保证 FORTRAN 和 C 之间的联合与填充是兼容的。ATTRIBUTES 的 C 和 ALIAS 选项可以和公共块使用来允许混合大小写名称。

假设 FORTRAN 有名为 Really 的公共块：

```
!DEC$ ATTRIBUTES ALIAS:'Really' :: Really
REAL(4) x, y, z(6)
REAL(8) ydbl
COMMON / Really / x, y, z(6), ydbl
```

使用下面的外部数据结构可以从 C 代码中直接访问上面的数据结构：

```
#pragma pack(2)
extern struct {
   float x, y, z[6];
   double ydbl;
} Really;
#pragma pack()
```

也可以通过创建对应这些结构的 FORTRAN 公共块来访问 C 结构。

4. 传递公共块的地址

为了传递公共块的地址，可以只简单地传递块中第一个变量的地址，也就是说通过引

用传递第一个变量。接受地址的 C 或 Visual C++模块应该准备通过引用接受结构。

在下面的例子中，C 函数 initcb 接受用户定义的类型 n 的地址：

```
! FORTRAN 源代码
!
   INTERFACE
      SUBROUTINE initcb (BLOCK)
        !DEC$ ATTRIBUTES C :: initcb
        !DEC$ ATTRIBUTES REFERENCE :: BLOCK
        INTEGER BLOCK
      END SUBROUTINE
   END INTERFACE
!
   INTEGER n
   REAL(8) x, y
   COMMON /CBLOCK/n, x, y
     . . .
   CALL initcb( n )

/* C 源代码 */
//
#pragma pack(2)
struct block_type
{
   int n;
   double x;
   double y;
};
#pragma pack()
//
void initcb( struct block_type *block_hed )
 {
    block_hed->n = 1;
    block_hed->x = 10.0;
    block_hed->y = 20.0;
}
```

11.4 处理混合语言编程的数据类型

处理好调用约定、命名约定和数据交换的方式后，还需要考虑数据类型，因为每种语言对数据类型的处理都不同。表 11.8 列出了 FORTRAN, C, Visual Basic 和 MASM 的等价数据类型：

表 11.8 **等价数据类型**

FORTRAN 数据类型	C 数据类型	Visual Basic 数据类型	MASM 数据类型
INTEGER(1)	char	---	SBYTE
INTEGER(2)	short	Integer	SWORD
INTEGER(4)	int, long	Long	SDWORD
REAL(4)	float	Single	REAL4
REAL(8)	double	Double	REAL8
CHARACTER(1)	unsigned char	---	BYTE
CHARACTER*(*)	见 11.4.5 “处理字符串”		
COMPLEX(4)	struct complex4 float real, imag; };	---	COMPLEX4 STRUCT 4 real REAL4 0 imag REAL4 0 COMPLEX4 ENDS
COMPLEX(8)	struct complex8 double real, imag; };	---	COMPLEX8 STRUCT 8 real REAL8 0 imag REAL8 0 COMPLEX8 ENDS
All LOGICAL types	使用 C, MASM 和 Visual Basic 的整数类型		

本节将讨论以下内容：

- 处理数字、复型和逻辑型数据类型
- 处理 FORTRAN 90 数组指针和可分配数组
- 处理 DIGITAL 指针
- 处理数组和 Visual FORTRAN 数组描述符
- 处理字符串
- 处理用户定义类型

11.4.1 处理数字、复型和逻辑型数据类型

通常，传递数字类型不会产生问题。如果一个 C 程序传递一个无符号数据类型给 FORTRAN 例程，这个例程可以用等价的数据类型来接受参数，但必须注意有符号类型的范围不要产生矛盾。

C, Visual C++和 MASM 没有直接实现 FORTRAN 的 COMPLEX(4)和 COMPLEX(8)类

型，但是可以编写出等价的结构。COMPLEX(4)有两个区域，每个都是 4 字节浮点数，第一个是实部，第二个是虚部。COMPLEX 类型和 COMPLEX(4)类型等价，COMPLEX(8)和它们类似，只是每个区域包含的是 8 字节浮点数。

注意：COMPLEX 型的 FORTRAN 函数在其参数列表的开头放置了一个隐含的 COMPLEX 参数。执行从 FORTRAN 中调用的 C 函数必须显式地声明该参数，并用它作为返回值。C 返回类型应该是 void。

下面是 Visual C/C++中定义与 FORTRAN 的 COMPLEX 相应的结构：

```
struct complex4 {
   float real, imag;
};
struct complex8 {
   double real, imag;
};
```

下面是 MASM 中定义的与 FORTRAN 的 COMPLEX 相应的结构：

```
COMPLEX4 STRUCT 4
  real REAL4 0
  imag REAL4 0
COMPLEX4 ENDS
COMPLEX8 STRUCT 8
  real REAL8 0
  imag REAL8 0
COMPLEX8 ENDS
```

FORTRAN 的 LOGICAL(2)是以 2 字节指示符值存储的；FORTRAN 的 LOGICAL(4)是以 4 字节指示符值存储的；LOGICAL(1)则是以单字节存储的。LOGICAL 等价于 LOGICAL(4)，也等价于 C 语言中的 int。

FORTRAN 的 LOGICAL 类型的变量可以用在参数列表、模块或全局变量中。在公共块中推荐使用 LOGICAL(4)。

11.4.2 处理 FORTRAN 90 数组指针和可分配数组

实际 ATTRIBUTES 和 ATTRIBUTES（如果有的话）的选项影响了 FORTRAN 90 数组指针和数组的传递。如果 ATTRIBUTES 声明数组指针或数组为迟形（例如 ARRAY(:)），则传递的是描述符。对于数组和数组指针是如此，但可分配数组不同。如果 INTERFACE 声明数组指针或数组为固定格式，或没有接口，则数组或数组指针会以基地址被传递，这和传递数组的第一个元素类似。当 FORTRAN 90 数组或数组指针传递给其它语言时，传

递的可以是描述符或基地址。

ATTRIBUTES 的选项对被作为参数传递的 FORTRAN 90 数组指针和可分配数组的影响如下：

- 如果数组指针或数组的属性为 none，则通过描述符传递，而不管进行传递的过程的属性如何（None; C; STDCALL; C, REFERENCE; 或 STDCALL, REFERENCE）。
- 如果数组指针或数组的属性为 VALUE，会返回错误，不管进行传递的过程的属性如何。
- 如果数组指针或数组的属性为 REFERENCE，则通过描述符传递，管不管进行传递的过程的属性如何。

当用户通过描述符给非 FORTRAN 例程传递一个 FORTRAN 数组指针或数组，该例程需要知道如何解释描述符。描述符的一部分是指向地址空间的指针，与 C 指针类似；另一部分是对数组或指针属性的描述，例如秩、增幅和上下界。指向一个包含数据地址的标量数据的 FORTRAN 90 指针并不能通过描述符传递。

11.4.3　处理 DIGITAL FORTRAN 指针

DIGITAL FORTRAN（整型）指针和 FORTRAN 90 指针有所不同，但和 C 指针却是类似的。DIGITAL FORTRAN 指针是 4 字节整型量。

在把 DIGITAL FORTRAN 指针传递给其它语言编写的例程中时：

- 参数应该在非 FORTRAN 例程中被声明为适当数据类型的指针。
- 从 FORTRAN 例程中传递的参数应该是 DIGITAL FORTRAN 指针名，而不是基于指针的变量名。

例如，在 x86 系统上：

```
! FORTRAN 主程序
      INTERFACE
        SUBROUTINE Ptr_Sub (p)
        !DEC$ ATTRIBUTES C, ALIAS:'_Ptr_Sub' :: Ptr_Sub
          INTEGER p
        END SUBROUTINE Ptr_Sub
      END INTERFACE
      REAL A(10), VAR(10)
      pointer (p, VAR) ! VAR 是基于指针的变量
      p = LOC(A)

      CALL Ptr_Sub (p)
      WRITE(*,*) 'A(4) = ', A(4)
      END
```

```
!
//C 子程序
     void Ptr_Sub (float *p)
     {
        p[3] = 23.5;
     }
```

在 Alpha 系统上，!DEC$ ATTRIBUTES 这行仍要省略_Ptr_Sub 前的下划线，即

```
!DEC$ ATTRIBUTES C, ALIAS:'Ptr_Sub' :: Ptr_Sub
```

当 FORTRAN 主程序和 C 函数被建立和执行后，输出如下：

```
A(4) = 23.50000
```

当在其它语言编写的例程中接受指针时：

- 参数应该在非 FORTRAN 例程中作为合适数据类型指针的参数并可以被传递。
- 被 FORTRAN 例程接受的参数应该被声明为一个 DIGITAL FORTRAN 指针名称，于是 pointer 语句应该把它和合适数据类型的基于指针的变量联系起来。在 FORTRAN 例程内部时，使用基于指针的变量来设置和访问指针所指的数据。

例如，在 *x*86 系统上：

```
! FORTRAN 子例程
     SUBROUTINE Iptr_Sub (p)
     !DEC$ ATTRIBUTES C, ALIAS:'_Iptr_Sub' :: Iptr_Sub
        integer VAR(10)
        pointer (p, VAR)
        OPEN (8, FILE='STAT.DAT')
        READ (8, *) VAR(4) ! 从文件中读取并保存 VAR 的第四个元素
     END SUBROUTINE Iptr_Sub
!
//C 主程序
extern void Iptr_Sub(int *p);

main ( void )
{
  int a[10];
  Iptr_Sub (&a[0]);
  printf("a[3] = %i\n", a[3]);
```

```
}
```

在 Alpha 系统上，!DEC$ ATTRIBUTES 这行仍要省略前面的下划线，即

```
!DEC$ ATTRIBUTES C, ALIAS:'Iptr_Sub' :: Iptr_Sub
```

当 C 主程序和 FORTRAN 子例程被建立和执行后输出结果为：

```
a[3] = 4
```

11.4.4 处理数组和 Visual FORTRAN 数组描述符

FORTRAN 90 允许数组以数组元素、数组片段或整个数组来引用。在 FORTRAN 90 里，数组元素是按列存储的。

当在 FORTRAN 和其它语言中使用数组时，元素索引和次序的问题就出现了。必须分别引用数组并保留其轨迹。FORTRAN, Visual Basic, MASM 和 C 数组元素索引都不尽相同。数组索引是应该在编写源程序时考虑的。

Array indexing is a source-level consideration and involves no difference in the underlying data.

Visual Basic 把数组和字符串保存为描述符：包含数组大小和地址的数据结构。但是存储对于用户的解释不同。

为了从 Visual Basic 传递数组给 FORTRAN，可以传递数组的第一个元素。缺省时，Visual Basic 通过引用传递变量，所以传递数组的第一个元素会给 FORTRAN 数组的首地址。Visual Basic 缺省时给第一个数组元素的索引编号为 0，而 FORTRAN 数组的第一个的缺省索引为 1。Visual Basic 使用下面的语句可以把索引设置成从 1 开始：

```
Option Base 1
```

此外，在任意语言中的数组声明中可以指定数组的下界为-32,768 到 32,767 的整数，例如：

```
' 在 Basic 中
Declare Sub FORTARRAY Lib "fortarr.dll" (Barray as Single)
DIM barray (1 to 3, 1 to 7) As Single
Call FORTARRAY(barray (1,1))

! 在 FORTRAN 中
Subroutine FORTARRAY(arr)
   REAL arr(3,7)
```

在 MASM 中，数组是一维的，数组元素必须按字节引用。汇编器在内存中连续存储，数组名称指的是其第一个地址。然后可以通过和第一个元素的关系来引用，例如：

```
xarray    REAL4    1.1, 2.2, 3.3, 4.4 ; initializes
                        ; a four element array with
                        ; each element 4 bytes
```

在 MASM 中引用 xarray 指的是第一个元素，该元素值为 1.1；如果要引用第二个元素，必须用 xarray[4] 或 xarray+4 指定第一个元素后的 4 个字节，例如：

```
yarray    BYTE     256 DUP     ; establishes a
                ; 256 byte buffer, no initialization
zarray SWORD 100 DUP(0) ; establishes 100
                ; two-byte elements, initialized to 0
```

FORTRAN 和 C 的区别有以下两方面：

- 数组的下界不同。缺省时，FORTRAN 的第一个元素索引为 1，而 C 和 Visual C++ 为 0，所以 FORTRAN 的下标应该比 C 的下标多 1。FORTRAN 也提供指定其它整型下标的选项。
- 如果数组是多维的，FORTRAN 最左侧的下标变化最快，而 C 是最右侧的下标变化最快，即一个是按列存储，一个是按行存储。

在 C 中声明为 X[3][3]的数组的前 4 个元素为：

```
X[0][0] X[0][1] X[0][2] X[1][0]
```

在 FORTRAN 中，前 4 个元素为：

```
X(1,1) X(2,1) X(3,1) X(1,2)
```

多维数组的索引随着维数的增加而扩展。例如，C 的声明：

```
int arr1[2][10][15][20];
```

等价的 FORTRAN 声明为：

```
INTEGER arr1( 20, 15, 10, 2 )
```

C 数组声明中使用常量扩展时，上界和其它语言不同。例如如果所声明的数组为 arr[5][5] 的最后一个元素是 arr[4][4]而非 arr[5][5]。

表 11.9 说明的是数组声明的等价形式。

表 11.9　各种语言的等价数组声明

语　言	数组声明	从 FORTRAN 中引用数组
FORTRAN	DIMENSION x(i, k) 或 type x(i, k)	x(i, k)
Visual Basic	DIM x(i, k) As type	x(i -1, k -1)
Visual C/C++	type x[k][i]	x(i -1, k -1)
MASM	在连续存储空间中声明和引用数组元素	

11.4.5　处理字符数组

缺省时，Visual FORTRAN 为数组传递隐含长度。隐含长度参数由一个 4 字节的无符号数组成，并且始终通过值传递。隐含长度紧跟在字符串地址的后面。可以通过使用属性来改变字符传递的缺省方式。表 11.10 说明了不同属性对被传递的字符串的影响。

表 11.10　ATTRIBUTES 选项对字符串传递的影响

参　数	缺省情况	C	STDCALL	C, REFERENCE	STDCALL, REFERENCE
字符串	和长度一起通过值传递	第一个字符被转换为 INTEGER(4) 并通过值传递	第一个字符被转换为 INTEGER(4) 并通过值传递	和长度一起通过引用传递	通过引用传递，无长度
有 VALUE 选项的字符串	错误	第一个字符被转换为 INTEGER(4) 并通过值传递	第一个字符被转换为 INTEGER(4) 并通过值传递	第一个字符被转换为 INTEGER(4) 并通过值传递	第一个字符被转换为 INTEGER(4) 并通过值传递
有 REFERENCE 选项的字符串	引用传递，可能和长度一起	通过引用传递，无长度	通过引用传递，无长度	通过引用传递，无长度	通过引用传递，无长度

关于表 11.10 还有以下几个值得注意的方面：

（1）　被传递给 C 或 STDCALL 例程并且没有 VALUE 或 REFERENCE 属性的字符串不能被引用。只有第一个字符通过值被传递。

（2）　被传递给 C 或 STDCALL 例程并且有 VALUE 或 REFERENCE 属性的字符串不是通过把整个字符串推入堆栈来传递的。只有第一个字符被传递。

（3）　对于有 **ATTRIBUTES** DEFAULT 或 C, REFERENCE 的字符串参数：

- 如果设置了/iface:mixed_str_len_arg，字符串的长度会紧接着字符串的首地址被推入堆栈（通过值）。
- 如果设置了/iface:nomixed_str_len_arg，字符串的长度会在所有其它参数之后被推入堆栈（通过值）。

（4）　对于通过有缺省 ATTRIBUTES 引用的字符串参数：

- 如果设置了/iface:mixed_str_len_arg，字符串的长度会紧接着字符串的首地址被推入堆栈（通过值）。

- 如果设置了/iface:nomixed_str_len_arg，对于被调过程字符串的长度不可知。

因为所有 C 中的字符串都相当于指针，C 要求字符串以引用来传递，而不需要字符串的长度。另外，和 FORTRAN 字符串不同，C 字符串是以 null 为结尾的。有两种办法可以在 FORTRAN 和 C 之间传递字符串：把 FORTRAN 字符串转换为 C 字符串，或编写 C 例程来接受 FORTRAN 字符串。

要把 FORTRAN 字符串转换为 C 字符串，可以选择一个无长度、通过引用来传递的属性结合，而且以 null 来结束字符串。例如，在 x86 系统上：

```
INTERFACE
  SUBROUTINE Pass_Str (string)
    !DEC$ ATTRIBUTES C, ALIAS:'_Pass_Str' :: Pass_Str
    CHARACTER*(*) string
    !DEC$ ATTRIBUTES REFERENCE, string
  END SUBROUTINE
END INTERFACE
CHARACTER(40) forstring
DATA forstring /'This is a null-terminated string.'C/
```

Alpha 系统上第一行!DEC$ ATTRIBUTES 要把开头的下划线省略：

```
!DEC$ ATTRIBUTES C, ALIAS:'_Pass_Str' :: Pass_Str
```

下面的例子说明了在 FORTRANDATA 语句中使用字符串的 null 结束符扩展：

```
DATA forstring /'This is a null-terminated string.'C/
```

C 接口为：

```
void Pass_Str (char *string)
```

为了使 C 字符串接受 FORTRAN 字符串，C 必须说明和字符串地址一起传递的长度参数，例如：

```
FORTRAN code
INTERFACE
SUBROUTINE Pass_Str (string)
CHARACTER*(*) string
```

```
END INTERFACE
```

C 例程必须要两个参数：

```
void __stdcall PASS_STR (char *string, unsigned int length_arg )
```

这个接口处理隐含长度参数，但还必须协调结尾为 null 的 C 字符串和没有 null 的 FORTRAN 字符串。另外，如果赋给 FORTRAN 字符串的字符少于声明的长度，FORTRAN 字符串将补以空格。

试图在 C 例程中处理这些字符串的差别，最好的方法是在 FORTRAN/C 混合语言编程中尽量采用 C 字符串形式。使用 C 字符串的另一个原因是 Win32 API 和绝大多数 C 库函数需要以 null 结尾的字符串。

使用 CHARACTER*(*)返回一个字符串的 FORTRAN 函数把一个隐含字符串参数和字符串的地址放在参数列表的开始处。

执行这样的 FORTRAN 函数调用的 C 函数必须显式地声明这个隐含字符串参数，并把它返回一个值。C 的返回类型应该是 void。但是，要注意不能使用字符串返回函数，否则可能出错。尽可能使用子例程或把字符串放置在模块或全局变量中。

Visual Basic 字符串必须以值传递的方式传递给 FORTRAN。Visual Basic 字符串实际是以包含长度和位置信息的结构存储的。值传递不使用这个结构而只传递 FORTRAN 需要的字符串的位置。例如：

```
! 在 Basic 中
Declare Sub forstr  Lib "forstr.dll" (ByVal Bstring as String)
DIM bstring As String * 40 Fixed-length string
CALL forstr(bstring)

! 在 FORTRAN 中
SUBROUTINE forstr(s)
!DEC$ ATTRIBUTES STDCALL:: forstr
CHARACTER(40) s
s = 'Hello, Visual Basic!'
END
```

FORTRAN 指令!DEC$ **ATTRIBUTES** STDCALL 告知 FORTRAN 不需要从主调 Visual Basic 程序传递的隐含长度参数。Visual Basic 程序中的名称被指定为小写，因为 STDCALL 使 FORTRAN 名称为小写。

MASM 缺省时并不增加一个字符串长度或 null 字符。为了添加字符串长度，可以使用下面的语法：

```
lenstring BYTE "String with length", LENGTHOF lenstring
```

为了添加 null 字符，可以：

```
nullstring BYTE "Null-terminated string", 0
```

11.4.6 处理用户定义类型

FORTRAN 90 支持用户定义的类型（类似于 C 语言中结构的数据结构）。用户定义类型可以像其它数据类型一样通过模块或公共块来传递，但其它语言必须知道这种类型的结构。例如：

```
! FORTRAN 代码
      TYPE LOTTA_DATA
         SEQUENCE
         REAL A
         INTEGER B
         CHARACTER(30) INFO
         COMPLEX CX
         CHARACTER(80) MOREINFO
      END TYPE LOTTA_DATA
      TYPE (LOTTA_DATA) D1, D2
      COMMON /T_BLOCK/ D1, D2

/* 访问 D1 和 D2 的 C 代码 */
extern struct {
   struct {
      float a;
      int b;
      char info[30];
      struct {
         float real, imag;
         } cx;
      char moreinfo[80];
   } d1, d2;
} T_BLOCK;
```

11.5　Visual FORTRAN/Visual C++混合语言编程

理解并解决了 FORTRAN 和 C 之间的调用、命名和参数传递约定的矛盾之后，可以准备建立应用程序了。

如果使用的是可编辑的 Visual C/C++，可以在 Microsoft Developer Studio 内部直接编译和调试代码。如果使用的是其它 C 编译器写的代码，可以在 Microsoft Developer Studio 内部通过从文件（File）菜单中选择新建（New）并在 File 选项卡中选择 Visual C/C++源代码来编辑。

但是，如果不是使用 Visual C/C++，必须在 Microsoft Developer Studio 之外编译代码或在命令行中建立 FORTRAN/C 程序或添加编译过的.OBJ 文件到 Microsoft Developer Studio 的 FORTRAN 项目中。

例如，如果已经有一个 C 主程序 CMAIN.C，它调用包含在 FORSUBS.F90 中的子例程，可以用下面的命令创建 CMAIN 程序：

```
cl /c cmain.c
DF cmain.obj forsubs.f90
```

FORTRAN（DF）编译器接受用 C 编写、编译的主程序对象文件。DF 编译器编译.F90 文件并用连接器创建名为 CMAIN.EXE 的可执行文件。

如果编译器都可以作上面的连接，而不管主程序是由什么语言编写的；但如果先用的是 DF 编译器，用 C 编译器是必须包括 DFOR.LIB，而且有可能碰到一些与 C 编译器使用的 LIBC.LIB 版本有关的问题。所以，最好先用 C 编译器，或使 FORTRAN 和 C 的项目设置都和缺省的 C 链接库相容。

当使用 Developer Studio 建立应用程序时，根据在项目（Project）菜单的 FORTRAN 选项卡的设置选项，FORTRAN 使用相应的缺省库。也可以在项目设置对话框中指定连接器设置。

在 FORTRAN 选项卡中的库（Libraries）项里，下列选项决定了所选的默认库：

（5）　使用的 FORTRAN 运行库（Static 或 DLL）和 DF 命令选项/dll

（6）　使用多线程库（/[no]threads）

（7）　使用 C 调试库（/[no]threads）

这些选项的结合关系如表 11.11 所示。

Visual C++选择库的方式还决定于项目（Project）菜单中的设置（Settings）选项，而不是在 C/C++选项卡中。在代码生成（Code Generation）项中，使用运行库选项如表 11.12 所示。

如果使用的是 Microsoft Visual C/C++，那么 Microsoft Developer Studio 可以很简单地创建 FORTRAN/C 混合程序而不需用户的特殊指令或步骤。用户可以按相应语言的语法颜色来编辑和浏览 C 和 FORTRAN 程序，可以把 C 源文件添加到 FORTRAN 项目或 FORTRAN 源文件添加到 C 项目中，它们会被自动编译和连接。

表 11.11 选项的结合关系

是 Static 还是 DLL 项目	是否使用多线程库	是否使用 C 调试库	使用的 FORTRAN 链接库	使用的 C 链接库
Static	否	否	dfor.lib	libc.lib
Static	否	是	dfor.lib	libcd.lib
Static	是	否	dformt.lib	libcmt.lib
Static	是	是	dformt.lib	libcmtd.lib
DLL	否	否	dfordll.lib (dfort.dll)	msvcrt.lib (msvcrt.dll)
DLL	否	是	dfordll.lib (dfort.dll)	msvcrtd.lib (msvcrtd.dll)
DLL	是	否	dformd.lib (dformd.dll)	msvcrt.lib (msvcrt.dll)
DLL	是	是	dformd.lib (dformd.dll)	msvcrtd.lib (msvcrtd.dll)

表 11.12 运行库选项

选择的菜单项	激活的 CL 选项或项目类型	项目文件中指定的缺省库
Single-threaded	/ML	libc.lib
Multithreaded	/MT	libcmt.lib
Multithreaded DLL	/MD	msvcrt.lib (msvcrt.dll)
Debug Single-threaded	/MLd	libcd.lib
Debug Multithreaded	/MTd	libcmtd.lib
Debug Multithreaded DLL	/MDd	msvcrtd.lib (msvcrt.dll)

调试 Visual C/FORTRAN 混合程序时，调试器会随着调试的代码类型作出相应的动作：根据被调试的代码会自动选择 C 或 FORTRAN 表达式求值程序，堆栈窗口会显示 FORTRAN 过程的 FORTRAN 数据类型和 C 过程的 C 数据类型。

多线程程序应该具有完整的多线程支持，所以如果使用的是 DFORMT.LIB，应该保证指定 LIBCMT.LIB 为缺省库。

第十二章 高级主题

本章对一些高级主题进行了简要介绍。如果需要更深入的信息请参考在线文档。

12.1 高效使用数组

本节简要说明数组的高效访问技术，这些技术在 DIGITAL FORTRAN 的循环转换优化中会自动应用。

12.1.1 数组整体操作

最快的数组操作发生在连续访问整个数组或大部分数组的时候。对整个数组或大部分数组实施一个或几个数组操作要比对分散的数组元素进行数目庞大的操作效率高得多。

与其使用显式的数组循环访问不如用数组的基本操作，例如需要把数组变量 A 的每个元素都增加 1：

```
A = A + 1.
```

读写一个数组时，应该使用数组名而不是指定每个元素编号的 DO 循环或隐 DO 循环。FORTRAN 90 的数组语法允许在表达式中用数组的名称引用整个数组。例如：

```
REAL ::  A(100,100)
A = 0.0
A = A + 1.              ! 数组 A 全部元素增加 1
.
.
.

WRITE (8) A             ! 快速的数组整体引用
```

类似地，可以使用派生数组结构成员，例如：

```
TYPE X
  INTEGER A(5)
```

```
END TYPE X
.
.
.
TYPE (X) Z
WRITE (8) Z%A      ! 快速数组结构成员引用
```

12.1.2 使用列为主的数组访问和存储

引用多维 FORTRAN 数组时，要保证使用正确地引用语法并注意列为主的“正常”增加顺序。对于列为主的顺序，数组最左边的下标变化最快。

行为主的顺序（例如 C 语言）或任意顺序的多维数组访问常常会使 CPU 高速缓存的使用效率低下，所以应该避免行为主的顺序。例如，考虑下面访问二维数组的嵌套 DO 循环，其中 J 循环为内层循环：

```
INTEGER  X(3,5), Y(3,5), I, J
Y = 0
DO I=1,3                    ! I 为外层循环，变化最慢
  DO J=1,5                  ! J 为内层循环，变化最快
    X (I,J) = Y(I,J) + 1    ! 低效率的按行存储顺序
  END DO                    ! (最右侧下标变化最快)
END DO
.
.
.
END PROGRAM
```

因为 J 变化最快，而且它是数组表达式 X (I,J)的第二个下标，所以这个数组是以行为主的顺序进行访问的。为了确保数组是按列为主的正常顺序访问，应该仔细检查数组的运算法则和被更改的数据。如果把 I 改为内层循环就可以获得按列为主的正常访问顺序：

```
INTEGER  X(3,5), Y(3,5), I, J
Y = 0

DO J=1,5                    ! J为外层循环，变化最慢
  DO I=1,3                  ! I为内层循环，变化最快
    X (I,J) = Y(I,J) + 1    ! 高效率的按行存储顺序
  END DO                    ! (最左侧下标变化最快)
END DO
```

```
.
.
.
END PROGRAM
```

FORTRAN 的整体访问(*X= Y + 1*)使用的是高效率的以列为主的顺序。但是，如果应用程序要求 J 变化最快或有时无法在不改变结果的前提下修改循环顺序，这时就应该考虑重新安排数组维的顺序。程序的改动包括重新安排以下顺序：

- 数组 X(5,3)和 Y(5,3)声明的维的顺序
- 在 DO 循环中的 X(J,I)和 Y(J,I)赋值的顺序
- 所有其它引用数组 X 和 Y 的顺序

这时，使用的是最初的 J 为内层循环的嵌套循环：

```
INTEGER  X(5,3), Y(5,3), I, J
Y = 0
DO I=1,3                    ! I 为外层循环，变化最慢
  DO J=1,5                  ! J 为内层循环，变化最快
    X (J,I) = Y(J,I) + 1    ! 高效率的按行存储顺序
  END DO                    ! (最左侧下标变化最快)
END DO
.
.
.
END PROGRAM
```

12.1.3 尽量使用 FORTRAN 90 内在数组过程

要完成同一个任务，应该尽可能使用 FORTRAN 90 内在数组过程而不是创建自己的例程。一方面 FORTRAN 90 的内在数组过程和 Visual FORTRAN 运行部分合作的效率很高，另一方面尽可能使用 FORTRAN 90 内在数组过程也使程序的可移植性更好。

12.1.4 多维数组维的宽度

在分散访问数组元素的多维数组中，应该避免最左侧的数组维的宽度成为 2 的幂次方（例如 512 或 1024）。

因为高速缓存的大小是 2 的幂次方，如果数组的大小也是 2 的幂次方，那么，在分散访问数组元素时就会降低高速缓存的使用效率。在 Alpha 系统上，如果数组最左侧的维宽度正好是高速缓存的倍数的话，则这个程序可能几乎无法利用高速缓存。

如果是顺序访问或整体访问数组则不存在这个问题。

一种改善的方法是增加维的宽度，允许存在一些没用的元素，从而使最左侧维的宽度变成非 2 的幂次方。例如，把最左侧的维宽度从 512 改为 520 能够更有效的利用高速缓存：

```
REAL A (512, 100)
DO I = 2, 511
  DO J = 2, 99
    A(I, J)=(A(I+1, J-1) + A(I-1, J+1)) * 0.5
  END DO
END DO
```

在上面的代码中，数组 A 的最左侧的宽度为 512，是 2 的 9 次方。内层循环是按行访问的，这样会使访问效率下降。如果把 512 增加到 520(即 REAL A (520,100))会使循环效率提高，代价是一些没用的元素。

因为循环索引变量 I 和 J 也参与计算，所以改变 DO 循环的嵌套顺序会改变运行结果。

在较高级的优化中（三级以上），编译系统会填补由 2 的幂次方可能引起的高速缓存的使用效率低下。

12.2 使用 IMSL 数学和统计库

Visual FORTRAN 的专业版包含了 IMSL 库。IMSL 库是 Microsoft Developer Studio 可以方便访问的近千个数学和统计函数的集合。

安装 IMSL 时也应该安装 IMSL 在线文档。在线文档可以使用户迅速找到 IMSL 库例程的用途和用法。使用 IMSL 之前应该先查看 IMSL 自述文档和 Visual FORTRAN 程序文件夹中的在线帮助。通过 IMSL 帮助文件可以访问以下主题：

- IMSL MATH/LIBRARY 子例程
- IMSL MATH/LIBRARY 特殊函数
- IMSL STAT/LIBRARY 子例程
- IMSL FORTRAN 90 MP 子例程

12.2.1 从 Visual FORTRAN 中使用 IMSL 库

为了使用 IMSL 库需要：

（1） 执行 DFVARS.BAT 文件（缺省的安装目录在\DF\BIN）来为开发环境设置必要的 IMSL 环境变量。DFVARS.BAT 文件执行之后设置了 INCLUDE 路径和库搜索路径。

在 Visual FORTRAN 里的 F90 命令行窗口中，DFVARS.BAT 文件已经被执行，Developer Studio 中 DFVARS.BAT 文件的等价形式也被执行。用户可以按下列步骤查看 Developer Studio 中的这些路径：

- 在工具（Tools）菜单中单击选项（Options）。
- 选择路径（Directory）选项卡。
- 在显示路径的下拉列表中选择库文件（Library files）并查看库文件的路径。
- 在显示路径的下拉列表中选择包含文件（Include files）并查看包含文件的路径。
- 如果改动了信息，单击 OK。

（2）　显式地把 IMSL 库传递给连接器。

在大多数情况下，IMSL 库可以通过使用 cDEC$ OBJCOMMENT LIB 源指令自动传递。用户可以按下列步骤查看 Developer Studio 中传递给连接器的库名称列表：

- 打开项目工作空间。
- 在项目（Project）菜单中选择设置（Settings）。
- 单击连接（Link）选项卡来查看 Object/Library 模块列表。IMSL 库在 Library Naming Conventions 中列出，并包含下列库名称：

```
sstatd.lib sstats.lib smathd.lib smaths.lib sf90mp.lib
```

- 如果改动了信息，单击 OK。

（3）　使 IMSL 例程及其接口能被程序利用

让程序能够利用 IMSL 例程及其接口的步骤如下：

- FORTRAN 90 程序中调用 MATH 和 STAT 库例程时，用户应该使用数值库模块来提供例程的接口块和参数定义。在用户的调用程序中包含下面的 USE 语可以在编译时校验 IMSL 例程是否正确使用：

```
USE numerical_libraries
```

在 FORTRAN 77 风格的程序中调用 MATH 和 STAT 库例程时，可以使用相应的 INCLUDE 语句来实施 USE 语句的等价形式：

```
INCLUDE IMSLF90.FI
```

调用 MATH 和 STAT 库例程时不需要分别声明函数和子例程。

- 如果同时也调用 FORTRAN 90 MP 库例程，应该使用 imslf90 模块来为提供所有 FORTRAN 90 MP 例程和 MATH 和 STAT 库例程的接口块和参数定义。在用户的调用程序中包含下面的 USE 语可以在编译时校验 IMSL 例程是否正确使用：

```
USE IMSLF90
```

下面是一个自由格式的 FORTRAN 90 例子，它调用了 IMSL 库中的 AMACH 函数和

UMACH 子例程。AMACH 函数获得了定义计算机实数算法的实型机器常量。返回的是正的机器无穷大。子程序 UMACH 获得的是删除单元号。

```
!  This free-form example demonstrates how to call
!  IMSL routines from Visual FORTRAN.
!
! The module numerical_libraries includes the Math and
! Stat libraries; these contain the type declarations
! and interface statements for the library routines.

PROGRAM SHOWIMSL

USE NUMERICAL_LIBRARIES
INTEGER NOUT
REAL RINFP

! The AMACH function and UMACH subroutine are
! declared in the numerical_libraries module

CALL UMACH(2, NOUT)
RINFP = AMACH(7)
WRITE(NOUT,*) 'REAL POSITIVE MACHINE INFINITY = ',RINFP
END PROGRAM
```

注意：一般来说，IMSL 例程对于多线程不是安全的。在多线程环境中，用户应该注意不能同时有两个活动的 IMSL 例程。为了保证这一点可以使用多线程控制技术，见本章最后一节“创建多线程程序”。

12.2.2 库命名约定

IMSL FORTRAN 77 MATH 和 STAT 数值库有单精度和双精度两个版本。IMSL 库使用的库名称如表 12.1 所示。

表 12.1　　IMSL 库名称

文件名	库 描 述
SMATHS	单精度 MATH 库，IMSL FORTRAN 77 数值库之一
SMATHD	双精度 MATH 库，IMSL FORTRAN 77 数值库之一
SSTATS	单精度 STAT 库，IMSL FORTRAN 77 数值库之一
SSTATD	双精度 STAT 库，IMSL FORTRAN 77 数值库之一
SF90MP	FORTRAN 90 MP 库，基于 FORTRAN 90 运算法则的新版本，为多处理器和其它高性能系统作了优化

IMSL FORTRAN 77 数值库一般用于应用数学和科学与商业应用中的分析和提出统计数据。

12.2.3　在混合语言环境中使用 IMSL 库

本节介绍如何在 Visual FORTRAN 和 Microsoft Visual C++的混合开发环境中使用 IMSL 库。

在混合语言编写的或为 Windows 编写的应用程序中，如果 IMSL 例程的标准输出或错误输出信息写到屏幕上会显得很笨拙。可以通过从 FORTRAN 例程中调用 UMACH 来避免这个问题，这样可以重新映射输出和错误单元至文件而不是屏幕。例如，下面的自由格式把 VHSTP 中的输出写入文件 STD.TXT，而把 AMACH 中的错误信息写到文件 ERR.TXT：

```
  PROGRAM fileout
!       This program demonstrates how to use the UMACH routine to
!       redirect the standard output and error output from IMSL
!       routines to files instead of to the screen. The routines
!       AMACH and UMACH are declared in the numerical_libraries module
!
    USE numerical_libraries
    INTEGER STDU, ERRU
    REAL x, frq(10)/3.0,1.0,4.0,1.0,5.0,9.0,2.0,6.0,5.0,3.0/
!
!       Redirect IMSL standard output to STD.TXT at unit 8
!
    CALL umach(-2, STDU)
    OPEN (unit=STDU, file='std.txt')
    CALL vhstp(10,frq,1,'Histogram Plot')
    CLOSE(8)
!
!       Redirect IMSL error output to ERR.TXT at unit 9
!
    CALL umach(-3, ERRU)
    OPEN (unit=ERRU, file='err.txt')
    x = amach(0)    ! Illegal parameter error
    CLOSE(9)
    END
```

IMSL 中的 VHSTP 例程输出到 STD.TXT 的标准输出为：

```
1
                Histogram Plot
Frequency-------------------------
   9  *           I           *
   8  *           I           *
   7  *           I           *
   6  *           I   I       *
   5  *         I I   I I     *
   4  *     I   I I   I I     *
   3  *  I  I   I I   I I I   *
   2  *  I  I   I I I I I I   *
   1  *  I I I I I I I I I I   *
  ---------------------------------
Class           5        10
```

IMSL 中的 AMACH 例程输出到 ERR.TXT 的错误输出为：

```
*** TERMINAL ERROR 5 from AMACH.  The argument must be between 1 and 8
***          inclusive.  N = 0
```

考虑下面使用 IMSL 库的简单 FORTRAN 例子：

```
USE numerical_libraries
real rinfp
rinfp = AMACH(7)
write(*,*) 'Real positive machine infinity = ',rinfp
end
```

输出为：

```
Real positive machine infinity = Infinity
```

相应的 C 语言例子为：

```
/* FILE CSAMPO.C */
#include <stdio.h>
#include <stdlib.h>
```

```
extern float _stdcall AMACH(long *);

main()
{
    long n;
    float rinfp;

    n = 7;
    rinfp = AMACH(&n);
    printf("Real positive machine infinity = %16E\n", rinfp);

    fflush(stdout);
    _exit(0);
}
```

这个 C 语言例子说明了当调用 IMSL 库时在需要的函数原型中的_stdcall 修饰符的用法和传递给子程序的变量地址的&地址操作符的用法。

上面的例子可以用 cl 命令来创建该目标文件，该目标文件可以被 DF 命令连接。

12.3 使用本国语言支持例程

对本国语言的支持是 FORTRAN 90 标准的新增特性，这当中我国的计算机工作者做出了突出的贡献。

12.3.1 概述

Visual FORTRANt 提供完整的本国语言支持（National Language Support，缩写为 NLS）库，包括地方化语言例程和多字节字符例程。用户可以使用这些例程来编写不同语言的应用程序。在很多国家的语言中，标准 ASCII 字符集显得不够用，因为对于有些语言缺少公共符号或标点（例如英镑符号），或因为有些语言使用的是非 ASCII 原本（例如俄语的字母），或有些语言的字符太多，不能使每个字符都用一个单独的字节表示（例如汉语）。

在很多非 ASCII 语言的情况，例如阿拉伯语和俄语，扩展单字节字符就足够了。用户只需要改变语言现场和代码页，这些可以在系统这一级或在程序中实现。但是，东方的语言例如汉语和日语使用的是成千上万个单独的字符，所以不能用单字节编码，要表示它们只能使用多字节字符。

字符集保存在称为编码集的表中。编码集有三部分：现场，是语言和国家（因为像西

班牙语在一些国家也不相同）；代码页，是用来补充计算机字母表的字符表；代表在屏幕上显示的字体。这三个部分可以分别设置。每台运行 Windows NT 或 Windows 9x 的计算机都内建了多种字符集，例如英语、阿拉伯语和西班牙语。而像汉语这样的多字符集不是标准的，只有相应的专门版本，例如 Windows NT-J 就是配置了 Japanese 编码集的日文版。

多字节字符例程（MB 例程）是由用户选择的代码页来控制。只有安装了特定多字符编码集的计算机才需要使用 MB 例程。标准编码集使用的都是单字节字符编码集。

值得注意的是，多字节字符只能用在字符串和源文件的注释中。它们不能用于变量名或语句。和程序对现场的改变类似，程序对代码页的改变只影响程序而不影响系统缺省值。

NLS 和 MB 例程包含在 DFNLS.LIB 库中。DFNLS.LIB 库由 DFNLS.MOD 和 DFNLS.F90 组成。为了使用这些例程，程序中应该出现 USE DFNLS 语句。

12.3.2 单字符集和多字符集

ASCII 字符集定义了从 0 到 127 的字符和从 128 到 255 的控制字符集。一些另外的单字节字符集，主要是欧洲的，0 到 127 的定义和标准 ASCII 码相同，但是 128 到 255 的定义不同。通过这些扩展，对于大多数欧洲派生的语言来说，定义需要的字符只要 8 位来代表就可以了。但像汉语等字符数远多于单字节所能表示的字符数的语言来说就需要多字节编码了。

多字节字符集包含单字节和双字节字符。多字节字符串可以包含单字节和双字节。双字节字符有一个引导字节和拖后字节。在特定的多字符集中，引导字节符和拖后字节可以重叠，并且有必要用字节的上下文判断一个字节是引导字节还是拖后字节。

12.3.3 本国语言支持库例程

在下面的内容中，函数和参数名是由大写和小写字母混合给出，这样使名称更容易理解。用户编写自己的应用程序时也可以采用这种形式。

1．现场设置和查询例程

程序启动时，当前语言和国别设置是从操作系统获得的。这些设置可以在 Windows NT 的控制面板（Control Panel）中的国际（International）菜单或 Windows 9x 的控制面板中的区域设置（Regional Settings）中修改。当前代码页也是从操作系统获得的，系统有一个缺省的控制台代码页和缺省 Windows 代码页。控制台程序获得的是系统控制台代码页，而 Windows 程序（包括 QuickWin 程序获得的是）Windows 代码页。

NLS 库提供了确定当前现场（本地编码集）的例程，可以返回当前现场的参数，提供系统支持的现场列表和设置现场为其它语言、国别或代码页。这些例程汇总在表 12.2 中，注意由这些例程设置的现场和代码页只影响调用例程的程序而不会改变系统的缺省设置，更不会影响其它程序。

表 12.2　本国语言支持库例程

名　称	程序类型	描　述
NLSSetLocale	函数	设置语言、国别和代码页
NLSGetLocale	子例程	获得当前语言、国别和代码页
NLSGetLocaleInfo	函数	获得关于当前本地编码集要求的信息
NLSEnumLocales	函数	获得系统支持的所有语言和国别的结合
NLSEnumCodepages	函数	返回系统支持的所有代码页
NLSSetEnvrionmentCodepage	函数	为当前控制台更改代码页
NLSGetEnvrionmentCodepage	函数	返回 Window 代码页或控制台代码页的编号

下面是使用现场设置和查询例程的一个例子：

```
USE DFNLS
INTEGER(4) strlen, status
CHARACTER(40) str

strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME1, str)
print *, str     ! 打印星期一
strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME2, str)
print *, str     ! 打印星期二
strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME3, str)
print *, str     ! 打印星期三
! 把现场改为墨西哥的西班牙语
status = NLSSetLocale("Spanish", "Mexico")
strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME1, str)
print *, str     ! 打印星期一（lunes）
strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME2, str)
print *, str     ! 打印星期二（martes）
strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME3, str)
print *, str     ! 打印星期三（miorcoles）
END
```

编译该程序可以按下列步骤：

（1）　输入上面的代码，存为.f90 格式，例如 Test1.f90。

（2）　在文件（File）菜单中选择新建（New）。

（3）　在项目（Projects）选项卡中选择 Win32 Console Application，填入项目名称，例如 NLS1。

（4）　这时项目工作空间中出现 File View 选项卡，在 NLS1 files 标题上单击右键，选择 Add Files to Project，出现添加文件对话框，选择上面的 Test1.f90 格式。

（5） 在建立（Build）菜单中选择 Build NLS1.exe 即可。

另一种方法是在保存为.f90 格式后直接在建立菜单中选择建立（Build），这时系统会提示建立命令需要一个活动的工作空间，是否创建一个缺省的项目工作空间。单击 OK 后系统会创建和文件同名的项目工作空间并完成编译和连接得到可执行文件。

2. NLS 格式化例程

时间、日期、货币和数字格式可以在 Windows NT 的控制面板中的国际菜单或 Windows 9x 的控制面板中的区域设置中设置。NLS 库也能为当前现场提供格式化例程。这些例程汇总在表 12.3 中，它们返回当前代码页中的字符串。这些字符串是程序开始时的缺省设置或由 NLSSetLocale 设置。

所有格式化例程返回的都是格式化字符串的字节数而不是字符数，因为如果包含多字节字符则字符数和字节数会有所不同。如果输出字符串不比格式化字符串长，将会填补空格。如果输出字符串比格式化字符串段则会产生错误，同时返回 NLS$ErrorInsufficient Buffer，输出字符串中不写入任何内容。

表 12.3 NLS 格式化例程

名 称	程序类型	描 述
NLSFormatCurrency	函数	格式化数字字符串并返回当前现场的正确现金字符串
NLSFormatDate	函数	返回包含当前现场的日期的正确格式化字符串
NLSFormatNumber	函数	格式化数字字符串并返回当前现场的正确数字字符串
NLSFormatTime	函数	返回包含当前现场的时间的正确格式化字符串

下面是一个使用 NLS 格式化例程的例子：

```
USE DFNLS
INTEGER(4) strlen, status
CHARACTER(40) str
strlen = NLSFormatTime(str)
print *, str                ! 打印时间                  11:30:36
strlen = NLSFormatDate(str, flags= NLS$LongDate)
print *, str                ! 打印日期          1999 年 5 月 10 日
! 把现场改为墨西哥的西班牙语
status = NLSSetLocale ("Spanish", "Mexico")
strlen = NLSFormatTime(str)
print *, str                ! 打印时间                  11:30:36 AM
strlen = NLSFormatDate(str, flags= NLS$LongDate)
print *, str                ! 打印日期 Lunes, 10 de Mayo de 1999
END
```

3. 多字节字符例程

这部分例程都主动使用多字节字符集（Multibyte Character Sets，缩写 MBCS）。汉语、日语和韩文都是多字节字符集。多字节字符例程从由 NLSSetLocale 设置、由 NLSGetLocale 读取的当前代码页开始工作。字符串比较例程，例如 MBLLT，是建立在当前语言和国别字符串的基础上的。

（1）MBCS 查询例程

MBCS 查询例程可以提供的信息有：多字节字符的最大长度，字符串中字符的长度、数目和位置以及一个多字节字符是引导字节还是拖后字节。MBCS 查询例程汇总在表 12.4 中。NLS 库提供在 NLS 模块中定义的参数 MBLenMax 作为在任何代码页中任何字符的最大长度（以字节为单位），这个参数在比较和测试时非常有用。确定当前代码页的最大字符长度可以使用 MBCurMax 函数。

表 12.4　　MBCS 查询例程

名　称	程序类型	描　述
MBCharLen	函数	返回字符串的第一个多字节字符的长度
MBCurMax	函数	返回当前代码页可能的最长的多字节字符
MBLead	函数	确定一个给出的字符是否是一个多字节字符串的首字节
MBLen	函数	返回字符串中的多字节字符数目，包括拖后空格
MBLen_Trim	函数	返回字符串中的多字节字符数目，不包括拖后空格
MBNext	函数	返回紧接着给出字符串位置后面多字节字符首字节的位置
MBPrev	函数	返回紧接着给出字符串位置前面多字节字符首字节的位置
MBStrLead	函数	进行上下文敏感测试来确定给定字节是引导字节还是拖后字节

下面是一个使用 MBCS 查询例程的例子：

```
USE DFNLS
CHARACTER(4) str
INTEGER status
status = NLSSetLocale ("China")
str = "中国"
PRINT '(1X,''String by char = '',\)'
DO i = 1, len(str)
  PRINT '(A2,\)',str(i:i)
END DO
PRINT '(/,1X,''MBLead = '',\)'
DO i = 1, len(str)
  PRINT '(L2,\)',mblead(str(i:i))
END DO
PRINT '(/,1X,''String as whole = '',A,\)',str
PRINT '(/,1X,''MBStrLead = '',\)'
DO i = 1, len(str)
```

```
    PRINT '(L1,\)',MBStrLead(str,i)
  END DO
  END
```

输出结果为：

```
MBStrLead = TFTF String by char =  ????
MBLead =  T T T T
String as whole = 中国
MBStrLead = TFTF
```

（2）MBCS 转换例程

MBCS 转换例程有四个，如表 12.5 所示。两个用来在日本工业标准（JIS）字符和 Microsoft 日本汉字（JMS）字符之间相互转换，另外两个用来在代码页多字节字符串和统一字符编码标准字符串之间相互转换。

表 12.5　　MBCS 转换例程

名　称	程序类型	描　述
MBConvertMBToUnicode	函数	把代码页多字节字符串转换为统一字符编码标准字符串
MBConvertUnicodeToMB	函数	把统一字符编码标准字符串转换为代码页多字节字符串
MBJISToJMS	函数	把 JIS 转换为 JMS
MBJMSToJIS	函数	把 JMS 转换为 JIS

（3）MBCS 的 FORTRAN 等价例程

NLS 库提供的 MBCS 的 FORTRAN 等价例程如表 12.6 所示。

表 12.6　　MBCS 的 FORTRAN 等价例程

名　称	程序类型	描　述
MBINCHARQQ	函数	和 INCHARQQ 基本相同，除了可以一次读取一个单独的多字节字符并返回读取的字节数
MBINDEX	函数	和 INDEX 基本相同，除了参数中可以包含多字节字符
MBLGE、MBLGT、MBLLE、MBLLT、MBLEQ、MBLNE	函数	和 LGE、LGT、LLE、LLT 和.EQ.、.NE.操作符基本相同，除了参数中可以包含多字节字符
MBSCAN	函数	和 SCAN 基本相同，除了参数中可以包含多字节字符
MBVERIFY	函数	和 VERIFY 基本相同，除了参数中可以包含多字节字符

下面是 Visual FORTRAN 说明 MBCS 例程的示例程序，在\DF\SAMPLES\TUTORIAL 子目录下，文件名为 MBCOMP.FOR：

```
USE DFNLS
```

```
      INTEGER(4) i, len(7), infotype(7)
      CHARACTER(10) str(7)
      LOGICAL(4) log4

      data infotype / NLS$LI_SDAYNAME1, NLS$LI_SDAYNAME2, &
     & NLS$LI_SDAYNAME3, NLS$LI_SDAYNAME4, &
     & NLS$LI_SDAYNAME5, NLS$LI_SDAYNAME6, &
     & NLS$LI_SDAYNAME7 /
      WRITE(*,*) 'NLSGetLocaleInfo'
      WRITE(*,*) '----------------'
      WRITE(*,*) ' '
      WRITE(*,*) 'Getting the names of the days of the week...'

      DO i = 1, 7
        len(i) = NLSGetLocaleInfo(infotype(i), str(i))
        WRITE(*, 11) 'len/str/hex = ', len(i), str(i), str(i)
      END DO
   11 FORMAT (1X, A, I2, 2X, A10, 2X, '[', Z20, ']')

      WRITE(*,*) ' '
      WRITE(*,*) 'Lexically comparing the names of the days...'

      DO i = 1, 6
        log4 = MBLGE(str(i), str(i+1), NLS$IgnoreCase)
        WRITE(*, 12) 'Is day ', i, ' GT day ', i+1, '? Answer = ', log4
      END DO
   12 FORMAT (1X, A, I1, A, I1, A, L1)

      WRITE(*,*) ' '
      WRITE(*,*) 'Done.'
      END
```

程序的输出结果为：

```
NLSGetLocaleInfo
 ----------------

 Getting the names of the days of the week...
 len/str/hex =  6  星期一       [20202020BBD2DAC6C7D0]
```

```
len/str/hex =  6  星期二      [20202020FEB6DAC6C7D0]
len/str/hex =  6  星期三      [20202020FDC8DAC6C7D0]
len/str/hex =  6  星期四      [20202020C4CBDAC6C7D0]
len/str/hex =  6  星期五      [20202020E5CEDAC6C7D0]
len/str/hex =  6  星期六      [20202020F9C1DAC6C7D0]
len/str/hex =  6  星期日      [20202020D5C8DAC6C7D0]

Lexically comparing the names of the days...
Is day 1 GT day 2?  Answer = T
Is day 2 GT day 3?  Answer = F
Is day 3 GT day 4?  Answer = F
Is day 4 GT day 5?  Answer = F
Is day 5 GT day 6?  Answer = T
Is day 6 GT day 7?  Answer = F

Done.
```

（4） 处理 MBCS 字符的标准 FORTRAN 例程

空格永远也不能是引导字节或拖后字节，所以许多处理空格的例程是专为包含 MBCS 字符的字符串而工作的。这些函数包括：

ADJUSTL(字符串), **ADJUSTR**(字符串), **TRIM** (字符串)

有些例程按计算机比较顺序返回一个序列中特定位置的字符或一个特定字符的位置。这些函数不依赖于一个特定的比较顺序，例如：

ACHAR(位置), **CHAR**(位置 [,种别]), **IACHAR**(*c*), **ICHAR**(*c*)

因为 FORTRAN 使用长度而不是 NULL 来确定字符串的长度，有些函数只根据字符串的长度而不是内容工作。这些函数和包含 MBCS 字符的字符串函数的工作形式类似，包括：

REPEAT (字符串, *ncopies*)

12.4 创建多线程应用程序

当使用 Windows 95 或者其它现在比较流行的操作系统时，可以同时运行几个程序，操作系统的这种能力称之为多任务处理。现在的许多操作系统也支持线程。一个应用程序能够创建几个线程。线程能够使用户在多任务中进行多任务。

Visual FORTRAN 提供对多线程应用程序的支持。用户在需要管理多个活动时可以考虑使用多线程。例如同时进行的键盘输入和计算时，一个线程用来处理键盘输入，而另一个线程用来完成数据交换和计算，第三个线程可以根据键盘输入线程提供的数据来更新屏

幕的显示，同时还可以有其它线程访问磁盘文件或从通讯端口获得数据等。

12.4.1　多线程的基本概念

线程是程序中一条执行路径，是操作系统分配处理器时间的最基本单元。线程是属于且只属于一个程序的可执行实体。每个程序在该程序创建时都至少有一个线程，即程序的基本线程或主线程。用户的主程序运行它的第一个线程。每一个 Win32 线程都有它自己的堆栈、CPU 寄存器的状态、安全机制和系统安排执行列表中的一个实体，并且可以独立的操作在同一程序中运行的其它线程。每个线程都可以使用所有程序的资源。

一个程序由一个或多个线程和相应的代码、数据以及在内存中的程序的其它资源组成。典型的程序资源有打开的文件、信号量（一种线程间通讯的手段）和动态分配的内存。当系统调度程序给出一个线程执行控制时这个线程才开始运行。优先级较低的线程需要等优先级较高的线程完成后才能进行。在多处理器的机器上，系统调度程序可以在多个处理器之间转移独立的线程来平衡 CPU 的负担。

因为线程所需的系统负担比创建一个完整的程序少，并且难度也要小，所以它们对于和其它任务同时进行的时间或资源紧张的操作是十分有用的。线程可以用于后台打印、管理输入设备或在编辑时备份数据等操作。

当打开了线程、程序、文件和通讯设备时，创建它们的函数返回一个句柄。每个句柄都有一个用来检查程序安全的相关访问控制列表（Access Control List，缩写为 ACL）。使用本节介绍的函数，程序和线程可以继承或送出句柄。对象和句柄调节了对系统资源的访问。

一般情况下，程序中的所有线程相互之间的执行是独立的。除非用户采取特殊手段使线程之间相互通讯，否则每个线程都会像全然意识不到程序中的其它线程一样独立操作。而共享公共资源的多个线程必须使用信号量或其它手段协调它们的工作。

12.4.2　编写多线程程序

多线程最适合用在下列几种情况：

（1）后台任务，例如数据计算、数据库查询和输出整理等，这些任务不直接包含窗口管理或用户接口

（2）相互之间独立的操作

（3）例如轮流检测串行端口的异步任务

如果用户的应用程序包含需要私有地址空间和私有资源的任务，应该通过创建多个程序而不是多线程来保护这些私有地址空间和私有资源不受其它线程获得的干扰。

下面将分别介绍创建多线程程序应该考虑的问题。

1．多线程程序的模块

Visual FORTRAN 提供了名为 DFMT.MOD 的模块。这个模块包含了基本 Win32 API 例程使用的接口语句和结构定义。为了使用多线程 API 需要在每个 FORTRAN 程序单元

（程序、子例程、函数或模块）包括一条 USE DFMT 语句。

DFMT 模块（文件名 DFMT.F90）的源代码包含类型定义和外部函数声明，用户可以使用它作为多线程程序的调用语法、数字和参数类型的引用。

其它支持多线程任务的 Windows API（例如窗口管理函数）包含在 DFWIN.F90 模块中，用 USE DFWIN 语句使该模块可以被程序使用。

2．线程的开始和停止

向程序中添加线程时应该考虑程序的开销。创建合适数目的线程可以使程序更好地响应和执行。通过多任务可以节省时间，但要注意为了保持多线程的信息会引起附近的时间开销。在决定线程创建个数之前，应该考虑哪些数据是只能针对程序，哪些数据是只能针对线程的。

对 CreateThread 函数的单独调用结果是创建一个线程、指定安全属性和内存堆栈大小，还命名了线程运行的例程。Windows 在包含线程的应用程序的虚拟地址空间中为该线程堆栈分配内存。线程结束执行后 CreateHandle 例程释放线程使用的资源。

（1） 开始线程

CreateThread 函数创建一个新线程，返回值为 INTEGER(4)型的线程句柄。返回值用于线程之间的通讯和关闭线程。CreateThread 函数的语法为：

CreateThread (security, stack, thread_func, argument, flags, thread_id)

除了 thread_func 外，所有的参数都是 INTEGER(4)型变量。thread_func 是运行 CreateThread 的例程的名称。

第一个参数 security 的类型是 SECURITY_ATTRIBUTES，在 DFMT.F90 中定义。如果 security 为零，线程具有和母程序相同的缺省安全属性。

第二个参数 stack 定义了新线程的堆栈大小。一个应用程序的所有缺省堆栈空间都在第一个线程执行时分配。用户要为程序需要的每个线程的各个堆栈所需分配的内存。调用 CreateThread 允许指定用户创建的每一个线程的堆栈大小，如果该值为零则该堆栈和程序的主线程大小相同。如果需要，堆栈的大小会动态增加，上限为 1 兆。

第三个参数 thread_func 是线程函数的起始地址。

第四个参数 argument 为 thread_func 的可选参数，用户程序定义该参数及其用法。

在创建线程之后并不会开始执行处理，直到程序给线程发出信号。第五个参数 flags 可以取两个值：0 或 CREATE_SUSPENDED。如果指定的是 CREATE_SUSPENDED，线程虽然被创建，但直到用户调用 ResumeThread 函数后才会运行。

最后一个参数 thread_id 是由 CreateThread 返回的。它是线程唯一的标识符，可以在调用其它多线程例程时使用。线程运行时不能有其它相同的标识符，但是该线程结束后操作系统还可以再次使用这个标识符。

引用线程可以使用它的句柄，也可以使用它的线程标识符。像 WaitForSingleObject 和 WaitForMultipleObjects 这样的同步函数用线程句柄作为参数。

（2） 停止线程

ExitThread 例程允许一个线程结束它自己的执行，语法为：

CALL EXITTHREAD ([中止状态])

中止状态可以被其它线程查询，中止状态为 0 说明的是正常中止。可以在程序中指定其它中止状态值和这些值的意义。当调用的线程不再需要后，调用它的线程应该关闭该线程的句柄。可以使用 CloseHandle 例程来释放例程使用的内存。直到最后一个线程句柄被关闭后线程对象才被删除。

多个句柄打开一个线程是可能的：例如，如果一个程序创建了两个线程，一个等待另一个的信息。在这种情况下，两个句柄先为第一个线程打开：一个是来自要求信息的线程，另一个是来自创建它的线程。当包含它们的程序结束时所有的句柄都被隐式地关闭。

TerminateThread 例程允许一个线程中止另一个线程，条件是两个线程的安全属性的设置合适。如果线程中止，与线程联系的 DLL 并不知道，其初始的堆栈也不会被释放，所以除非在紧接情况下最好不要使用 TerminateThread。

（3）其它线程支持函数

可以通过 GetThreadPriority 和 SetThreadPriority 安排线程的优先级。使用线程的优先级是用来区分急需时间和正常或低于正常安排需要的应用程序。要处理优先级时，要特别注意给线程太高的优先级，否则可能消耗所有可用的 CPU 时间。一个具有高于 11 的基本优先级的线程就会对操作系统的正常操作产生影响。使用 REALTIME_PRIORITY_CLASS 可能会引起磁盘缓存停止交换，挂起鼠标等等。

和其它线程通讯时，线程用 pseudohandle 指线程本身。Pseudohandle 是一个特殊的常量，被解释为当前线程的句柄。Pseudohandle 只有在调用线程时才有效，也不能被其它线程继承。GetCurrentThread 函数返回当前线程的 pseudohandle。

为了获得线程的标识符可以使用 GetCurrentThreadId 函数。GetExitCodeThread 可以检查一个线程是否是获得的，如果不是则返回退出状态。关于退出状态的更多信息可以调用 GetLastError。

3．线程例程格式

运行在主程序下的单独线程中的一个函数或子例程可以有一个参数。下面的代码简要说明了一个函数或子例程的框架：

```
INTEGER(4) FUNCTION thrdfnc(arg)
USE DFMT
integer(4) arg
arg = arg + 1        ! Sample only; real work goes here.
thrdfnc = 0          ! Sets exit code to 0.
END FUNCTION

SUBROUTINE thrdfnc2 (arg2)
USE DFMT
integer(4) arg2
```

```
                    ! Subroutine work goes here.
Call exitthread(0) ! Exit code is 0.
END
```

当主程序调用 CreateThread 时，参数 arg 或 arg2 被传递给函数或子例程。

当函数或子例程中止后，线程也会自动结束。

4．共享资源

每个线程都有其自己的堆栈和 CPU 寄存器中的拷贝。其它资源，例如文件、单元、静态数据和堆积内存都可以在程序中被所有线程共享。使用这些公共资源的线程必须相互协调它们之间的工作。有以下几种方法来共享资源，每种方法提到的对象的状态都是被通知或者是未被通知。被通知状态说明一个资源对要使用它的程序或线程是可用的；未被通知状态说明资源正在被使用。下面介绍的例程管理的是创建、初始化和结束资源共享机制。其中有些把未被通知状态转变为被通知状态。例程 WaitForSingleObject 和 WaitForMultipleObjects 也可用来改变对象的通知状态。

（1）临界片段

临界片段是访问不能共享资源的一个代码块。临界片段的典型应用是限制对代码或数据的访问，这些代码或数据在程序中一次只能被一个线程使用（例如，对公共块中共享数据的改动）。

在使用临界片段使线程同步之前必须通过调用 InitializeCriticalSection 来初始化临界片段。开始处理全局变量时调用 EnterCriticalSection，结束时调用 LeaveCriticalSection。在一个程序中 EnterCriticalSection 和 LeaveCriticalSection 都可以被多次调用。

关于使用临界片段的多线程示例可以参考\DF\SAMPLES\ADVANCED\PEEKAPP 子目录下的 PEEKAPP.F90。

（2）互斥对象（MUTual Exclusion，缩写为 Mutex）

互斥对象是一次只允许一个线程访问某个资源的机制。互斥对象的典型应用是限制对系统的访问，这些系统资源在程序中一次只能被一个线程使用（例如打印机），或者如果被多线程共享可能会引起不可预料的结果。

CreateMutex 创建一个互斥对象，如果互斥对象已经存在（被其它程序或线程创建的同名对象）则返回错误信息。调用 CreateMutex 之后可以调用 GetLastError 来查看错误状态参数 ERROR_ALREADY_EXISTS。还可以使用 OpenMutex 函数来确定是否存在一个指定名称的互斥对象，如果存在，OpenMutex 将返回对象的句柄；如果指定名称的互斥对象不存在则返回值为空（null）。

ReleaseMutex 把一个互斥对象从未被通知状态改为通知状态，但只有调用它的线程也拥有这个互斥对象时 ReleaseMutex 函数才有效。当互斥对象是被通知状态时，任何等待这个互斥对象的线程都可以获得它并开始执行。

（3）信号量

信号量是用来调整可以使用某个资源的线程的编号的计数器。信号量的典型应用是控制对同一资源以指定编号的访问。

处理信号量的函数和管理互斥对象的函数很类似。CreateSemaphore 创建一个信号量，

并指定可以访问资源线程的最大编号为初始值。OpenSemaphore 和 OpenMutex 类似，返回的是指定名称的信号量（如果存在的话）对象的句柄。而后，该句柄可以被任何需要它的函数使用。调用 OpenSemaphore 并不减少资源可用的数目，它是由等待资源的函数完成的。

可以用指定数量的 ReleaseSemaphore 来增加资源的可用数目。当线程结束使用资源时可以调用这个函数。另一个可能的用法是调用 CreateSemaphore 指定初始数目为 0 来保护在初始化过程中使资源不被访问，当程序初始化完成后再调用 ReleaseSemaphore 来增加资源的可用数目至最大值。

（4） 事件

事件对象声明一个事件已经对一个或多个线程发生。

事件对象可以引发其它线程的执行。如果一个线程向多个其它线程提供数据可以使用事件。事件对象可以用 CreateEvent 函数创建，创建时指定了对象的初始状态和是手动重置还是自动重置。手动重置事件是在显式调用 ResetEvent 重置之前一直保持被通知状态的事件。自动事件在等待线程的信号释放时就被系统重置。

可以用 SetEvent 或 PulseEvent 来设置事件对象的状态。OpenEvent 返回事件的句柄，这个句柄可以被其它函数使用。ReleaseEvent 则释放对事件的所有权。

5. 内存使用和线程堆栈

因为每个线程都有自己的堆栈，用户可以通过使用尽可能少的静态数据来避免潜在的数据项冲突。在多线程例程中声明的所有变量缺省状态下都是静态数据并可以被线程共享。如果不希望一个线程覆盖另一个线程使用的变量，可以采取下列措施：

（1） 声明变量为 **AUTOMATIC**

（2） 创建变量值的向量，每个线程一个分量，于是不同的线程变量值的存储地址就不会相同。（可以用由 CREATETHREAD 传递的单个整型参数作为识别线程的索引）

（3） 使用线程当地存储（TLS）

声明为自动的变量存在于堆栈中并成为和线程一起保存的线程索引的一部分。过程中的自动变量在过程结束执行时被丢弃。

6. 输入输出操作

虽然文件和单元可以在线程之间共享，但用户也可能不需要协调线程对这些共享资源的使用。FORTRAN 把每个输入输出语句都当作自动操作。如果两个独立的线程如果都试图写入同一个单元，并且一个线程的输出操作已经开始，则要等到这个输出操作结束后其它线程的输出才可以开始。

操作系统并不给线程对单元或文件的访问规定一个顺序。例如，多线程程序的非确定特性会引起一个顺序文件中的记录在每一次执行中都以不同的顺序写入。这种情况下直接文件访问可能是比顺序文件访问更好的选择。如果不能使用直接文件访问，可以使用互斥对象规定顺序文件的输入输出顺序。

7. 线程本地存储

线程本地存储（TLS）调用允许用户存储每个线程的数据。TLS 方法的机制是：多线程程序中的每一个线程都可以分配用于存储指定线程数据的地址。

动态（运行）特殊线程数据可以被下列例程支持：TlsAlloc（分配存储数据的索引），TlsGetValue（从一个索引获得值），TlsSetValue（存储值到一个索引）和 TlsFree（释放动态存储）。线程分配动态存储并使用 TlsSetValue 把索引和指向存储区域的指针相联合。当线程需要访问存储区域时会调用 TlsSetValue。

当所有线程完成使用索引后 TlsFree 释放动态存储空间。

8．线程同步

例程 WAITFORSINGLEOBJECT 和 WAITFORMULTIPLEOBJECTS 使线程等待不同的事件发生。它们允许线程和程序有效地等待，不消耗 CPU 资源，直到指定的超时间隔结束。WAITFORSINGLEOBJECT 把对象句柄作为第一个参数，而且直到由句柄引用的对象获得被通知状态或直到指定的超时间隔结束才会返回。WAITFORSINGLEOBJECT 的语法为：

WaitResult = **WAITFORSINGLEOBJECT** (*ObjectHandle*, [*Timeout*])

如果使用超时，第二个参数就是以毫秒为单位的超时值。值 WAIT_INFINITE 代表一个每有限制的超时，这时函数等待直到 *ObjectHandle* 结束。

WAITFORMULTIPLEOBJECTS 是类似的，除了第二个参数是 Windows 对象句柄组成的数组。第一个参数指定要等待的句柄编号，这个编号可以比创建线程的总的数目要少，并且最大值为 64。函数可以等到所有事件都完成，也可以在一个对象结束时立即继续。

当线程等待一个不会变为可用的对象时会出现死锁，所以，在有等待可能不会结束的线程时应该使用超时参数。

可以使用 SuspendThread 来停止一个线程的执行。SuspendThread 在线程同步时并不是特别有用，因为它不控制代码中线程被挂起的位置。不过，如果需要确定用户将要结束工作的输入，可以挂起一个线程。如果确认了，线程被结束，否则线程仍会继续。

如果线程以挂起状态创建，直到与挂起线程一起的 Resume Thread 被调用时才开始运行。这在线程运行之前初始化线程的状态是很有用的。

9．处理多线程程序错误

如果任何多线程例程返回错误代码，可以使用 GetLastError 函数获得错误信息。注意它返回的是上次错误的错误代码，而不是上次调用的错误状态。

错误代码是 32 位的值，第 29 位为程序定义代码保留。用户可以设置该位，并且如果创建自己的动态链接库可以使用 SetLastError 来仿效 Win32 API 行为。Win 32 函数只有在失败时才调用 SetLastError，成功时不能调用。上一个错误代码保存在线程当地存储中，所以多线程不会覆盖其它的值。

12.4.3 编译和连接多线程程序

DFORMT.LIB 支持库是为创建静态连接的多线程程序的一个重入库，调用 DFORMD.DLL 中代码的 DFORMD.LIB 库也是重入的。与 DFORMT.LIB 一起建立的程

序并不共享它们调用的任何动态链接库（DLL）中的代码或 FORTRAN 运行库。如果希望调用 DLL 必须和 DFORMD.LIB 连接。

为了建立使用 FORTRAN 运行库的多线程应用程序，必须告诉编译器使用特殊的库版本。可以在命令行中指定/threads 编译选项，或在 Microsoft Developer Studio 中的项目设置窗口（Project Settings）中指定。

在\DF\SAMPLES\ADVANCED\WIN32\THREADS 和\DF\SAMPLES\TUTORIAL 子目录下分别有多线程项目和源文件的示例，文件名为 THREADS.MAK 和 THREADS.F90。为了建立这个示例可以打开 THREADS.MAK 项目并在建立（Build）菜单中选择 Build All。使用 Microsoft Developer Studio 编译和连接用户自己的多线程程序的步骤如下：

（1）创建一个新项目。（示例 THREADS.F90 是 QuickWin 项目）

（2）向项目中添加含有源代码的文件。

（3）在项目（Project）菜单中，选择设置（Settings）.

（4）选择 FORTRAN 选项卡，在类别（Category）下拉菜单中选择 FORTRAN Libraries，设置运行库为多线程库（DFORMT.LIB）或 DLL 中的多线程库（DFORMD.LIB）。

（5）在建立（Build）菜单中选择 Build All 来创建可执行文件。

附录 Visual FORTRAN 语言简表

为了读者查询方便，这里给出 Visual FORTRAN 的一个语言简表。

以下表格的括号中附上了过程（procedure）的可选参数：[optinal arg]。表中以关键字作为参数名。关键字允许用户指定可选参数但不必考虑顺序。例如，当调用

PRODUCT(ARRAY [, DIM][, MASK])

时，可以省略参数 DIM：

array3 = PRODUCT(array1, MASK = array2)

程序单元的调用和定义

名 称	过程类型	描 述
BLOCK DATA	语句	定义块数据子程序
CALL	语句	执行一个子程序
COMMON	语句	定义多个程序单元共用的变量
CONTAINS	语句	在主模块中确定一个模块的开始
ENTRY	语句	定义子程序或外部函数的第二个入口点
EXTERNAL	语句	声明用户定义的子程序或函数可作为参数传递
FUNCTION	语句	定义一个程序单元为函数
INCLUDE	语句	在资源文件中插入指定文件内容
INTERFACE	语句	为外部函数或子函数定义显式接口
INTRINSIC	语句	声明预先定义的函数
MODULE	语句	定义一个模块程序单元
PROGRAM	语句	定义一个程序单元为主程序
RETURN	语句	调用子程序或函数的程序单元返回控制
RUNQQ	运行时函数	RUNQQ(filename, commandline). 调用另一个程序并等待它执行
SUBROUTINE	语句	定义一个程序单元为子程序
USE	语句	使用模块

程序控制

名 称	过程类型	描 述
CASE	语句	在 SELECT CASE 结构中标记一个语句块，如果相关量符合 SELECT CASE 表达式就执行这个语句块
CONTINUE	语句	不进行任何操作，通常作为 GOTO 语句的目标，或作为 DO 循环的结束语句
CYCLE	语句	DO 循环末尾语句的高级控制；之间的循环语句不执行。
DO	语句	执行确定次数的 DO 循环内的语句
DO WHILE	语句	执行 DO WHILE 循环内的语句，直到逻辑条件变为.FALSE.
ELSE	语句	把控制流程引入 ELSE 块
ELSE IF	语句	把控制流程引入 ELSE IF 块
ELSEWHERE	语句	把控制流程引入 ELSEWHERE 块
END	语句	标记程序单元结束
END DO	语句	在 DO 或 DO WHILE 语句后标记一系列语句的结束
END FORALL	语句	在 FORALL 语句后标记一系列语句的结束
END IF	语句	在 IF 语句后标记一系列语句的结束
END SELECT	语句	在 SELECT CASE 语句后标记一系列语句的结束
END WHERE	语句	在 WHERE 语句后标记一系列语句的结束
EXIT	语句	跳出 DO 循环；执行紧接着的第一个语句
EXIT	运行时间子程序	CALL EXIT(exitvalue)结束程序，刷新和关闭所有打开的文件并返回到操作系统
FORALL	语句	控制其它语句有条件地执行
GOTO	语句	跳转到程序指定位置
IF	语句	控制其它语句有条件地执行
PAUSE	语句	挂起程序，可以选择执行操作系统命令
RAISEQQ	运行时函数	RAISEQQ(sig)模拟操作系统中断，向执行中的程序发出中断
SELECT CASE	语句	由控制语句把程序转到一个语句块
SIGNALQQ	运行时函数	SIGNALQQ(sig, func)控制信号处理
SLEEPQQ	运行时子程序	CALL SLEEPQQ(duration)延迟程序的执行指定时间
STOP	语句	中止程序执行
WHERE	语句	控制其它语句的条件执行

变量定义

名 称	过程类型	描 述
AUTOMATIC	语句	在堆栈中而不是静态内存中定义变量
BYTE	语句	定义变量为 BYTE 数据类型，BYTE 等价于 INTEGER(1).
CHARACTER	语句	定义变量为 CHARACTER 数据类型
COMPLEX	语句	定义变量为 COMPLEX 数据类型
DATA	语句	给变量赋初值
DIMENSION	语句	定义变量为数组并指定元素个数
DOUBLE COMPLEX	语句	定义变量为 DOUBLE COMPLEX 数据类型，等价于 COMPLEX(8)
DOUBLE PRECISION	语句	定义变量为 DOUBLE-PRECISION 数据类型,等价于 REAL(8).

续表

名 称	过程类型	描 述
EQUIVALENCE	语句	指定多个变量或数组共享同一内存地址
IMPLICIT	语句	指定实型和整型变量和函数的缺省类型
INTEGER	语句	定义变量为 INTEGER 数据类型
LOGICAL	语句	定义变量为 LOGICAL 数据类型
MAP...END MAP	语句	在 UNION 语句中给将在内存中顺序排列的一组变量类型定界
NAMELIST	语句	定义在一个单独语句中要读或写的一组变量的名称
PARAMETER	语句	给一个名称赋常数表达式
REAL	语句	定义变量为 REAL 数据类型.
RECORD	语句	定义一个或多个用户定义结构类型的变量
SAVE	语句	使变量保留其在被定义的过程的引用之间的值
STATIC	语句	在静态内存中而不是堆栈中定义一个变量
STRUCTURE..END STRUCTURE	语句	定义一种由其它变量类型组成的新的变量类型
TYPE...END TYPE	语句	定义一种由其它变量类型组成的新的变量类型
UNION...END UNION	语句	在一个结构中使多个映射占据同一内存空间
VOLATILE	语句	指定一个对象的值在当前程序单元提供的信息下是完全不能确定的

系统、磁盘和目录

名 称	过程类型	描 述
CHANGEDIRQQ	运行时函数	CHANGEDIRQQ(dir)使指定目录为当前（默认）目录
CHANGEDRIVEQQ	运行时函数	CHANGEDRIVEQQ(drive)使指定磁盘为当前磁盘
DELDIRQQ	运行时函数	DELDIRQQ(dir)删除指定目录
GETDRIVEDIRQQ	运行时函数	GETDRIVEDIRQQ(drivedir)获得并返回当前磁盘和路径
GETDRIVESIZEQQ	运行时函数	GETDRIVESIZEQQ(drive, total, avail)获得指定磁盘容量
GETDRIVESQQ	运行时函数	GETDRIVESQQ( )向系统报告可用的磁盘
GETENVQQ	运行时函数	GETENVQQ(varname, value)从当前环境获得一个值
MAKEDIRQQ	运行时函数	MAKEDIRQQ(dirname)建立指定目录名称的目录
SETENVQQ	运行时函数	SETENVQQ(varvalue)增加一个新的环境变量，或将其设为存在值
SYSTEMQQ	运行时函数	SYSTEMQQ(commandline)通过向操作系统的命令解释器传递命令字符串来执行一个命令

文件管理

名 称	过程类型	描 述
DELFILESQQ	运行时函数	DELFILESQQ(files)在指定目录删除指定文件
FINDFILEQQ	运行时函数	FINDFILEQQ(filename, varname, pathbuf)在由 PATH 环境变量指定的目录下查找一个文件
FULLPATHQQ	运行时函数	FULLPATHQQ(name, pathbuf)返回指定文件或目录的完整路径
GETDRIVEDIRQQ	运行时函数	GETDRIVEDIRQQ(drivedir)返回当前磁盘和路径
GETFILEINFOQQ	运行时函数	GETFILEINFOQQ(files, buffer, handle)返回文件名符合指定字符串要求的文件信息
PACKTIMEQQ	运行时子程序	CALL PACKTIMEQQ(timedate, iyr, imon, iday, ihr, imin, isec)把时间值打包供 SETFILETIMEQQ 使用
RENAMEFILEQQ	运行时函数	RENAMEFILEQQ(oldname, newname)重命名一个文件
SETFILEACCESSQQ	运行时函数	SETFILEACCESSQQ(filename, access)为指定文件设置文件通道模式
SETFILETIMEQQ	运行时函数	SETFILETIMEQQ(filename, timedate)由提供的文件修改时间
SPLITPATHQQ	运行时函数	SPLITPATHQQ(path, drive, dir, name, ext)把一个完整的路径分解成四部分
UNPACKTIMEQQ	运行时子程序	CALL UNPACKTIMEQQ(timedate, iyr, imon, iday, ihr, imin, isec)给一个时间日期打包文件解包成年月日小时分秒

输入/输出

名 称	过程类型	描 述
ACCEPT	语句	类似于一条格式化的连续的 READ 语句
BACKSPACE	语句	定位到文件上一个记录的开始处
CLOSE	语句	断开指定文件或设备
DELETE	*语句*	*从相关文件中删除一个记录*
ENDFILE	语句	写一个文件结束记录
EOF	内在函数	EOF(unit)检查是否为文件结束记录，如果位于或超过文件末尾值为.TRUE.
INQUIRE	语句	返回一个文件或单元的属性
OPEN	语句	把一个外部设备或文件与一个单元数字相联系
PRINT*(或 TYPE)*	语句	在屏幕上显示数据
READ	语句	从一个文件向 I/O 列表中的项目传输数据
REWIND	语句	重新定位到文件的第一个记录
REWRITE	语句	覆盖当前记录
UNLOCK	语句	释放一条在被前一 READ 语句锁定的相关或顺序文件中的记录
WRITE	语句	从 I/O 列表中的项目向一个文件传输数据

随机数

注：方括号[...]代表可选参数。

名 称	过程类型	描 述
RAN	内在函数	result = RAN(i)获得下一个 0 到 1 之间均匀分布的伪随机数
RANDOM	运行时间子程序	CALL RANDOM(ranval)返回一个大于等于 0 小于 1 的实伪随机数
RANDOM_NUMBER	内在子函数	CALL RANDOM_NUMBER(harvest)返回一个大于等于 0 小于 1 的实伪随机数
RANDOM_SEED	内在子函数	CALL RANDOM_SEED([size] [, put] [, get])改变 RANDOM_NUMBER 的开始点，只需一个或不需参数
RANDU	内在子函数	CALL RANDU(i1, i2, x)以单精度计算一个伪随机数
SEED	运行时间子程序	CALL SEED(seedval)改变 RANDOM 的开始点

时间和日期

注：方括号[...]代表可选参数。

名 称	过程类型	描 述
CPU_TIME	内在子函数	CALL CPU_TIME (time)以秒返回 CPU 时间
DATE	内在子函数	CALL DATE (buf)返回以 ASCII 表示的当前日期（日-月-年）
DATE_AND_TIME	内在子函数	CALL DATE_AND_TIME([date] [, time] [, zone] [, values])返回日期和时间
GETDAT	运行时间子程序	CALL GETDAT (iyr, imon, iday)返回日期
GETTIM	运行时间子程序	CALL GETTIM (ihr, imin, isec, i100th)返回时间
IDATE	内在子函数	CALL IDATE (i, j, k)返回代表当前月日年的三个整数值
SETDAT	运行时间函数	SETDAT (iyr, imon, iday)设置日期
SETTIM	运行时间函数	SETTIM (ihr, imin, isec, i100th)设置时间
SYSTEM_CLOCK	内在子函数	CALL SYSTEM_CLOCK (count, count_rate, count_max)，返回系统时钟数据
TIME	内在子函数	CALL TIME (buf)返回以 ASCII 表示的当前时间（小时:分:秒）

键盘和扬声器

名 称	过程类型	描 述
BEEPQQ	运行时间子程序	CALL BEEPQQ(freq, duration)使扬声器以指定频率（Hz）发声并持续指定时间（ms）
GETCHARQQ	运行时间函数	GETCHARQQ( )返回下一个敲键
GETSTRQQ	运行时间函数	GETSTRQQ(buffer)从键盘缓冲区输入中读取一个字符串
PEEKCHARQQ	运行时间函数	PEEKCHARQQ( )查看缓冲看是否有敲键准备

错误处理

名 称	过程类型	描 述
GETLASTERRORQQ	运行时间函数	GETLASTERRORQQ( )返回上一个由运行时间函数或子程序设置的错误
MATHERRQQ'	运行时间子程序	CALL MATHERRQQ(name, len, info, retcode)替换内部数学函数的默认错误处理
SETERRORMODEQQ	运行时间子程序	CALL SETERRORMODEQQ(prompt)设置处理临界错误的模式

只适用于 x86 系统

参数查询

注：方括号[...]代表可选参数。

名 称	过程类型	描 述	参数/函数类型
ALLOCATED	内在函数	ALLOCATED(array) 测定一个可分配数组是否被分配	array:可分配数组 结果:逻辑量
ASSOCIATED	内在函数	ASSOCIATED(pointer[, target]) 测定一个指针和（可选）对象是否相关联	pointer: 任何类型 target: 任何类型 结果: 逻辑型
DIGITS	内在函数	DIGITS(x) 返回和 x 类型相同的数据的有效位数	x: 整型或实型 结果:整型
EPSILON	内在函数	EPSILON(x) 当加上一个数结果大于本身时返回和 x 类型相同的最小正数	x: 实型 结果: 同 x
GETARG	运行时间子程序	CALL GETARG(n, buffer[, status]) 返回指定的命令行参数	n: INTEGER(2) 或 INTEGER(4) buffer: Character*(*) status: INTEGER(2)
HUGE	内在函数	HUGE(x) 返回 x 类型数据所能表示的最大数据	x: 整型或实型 结果:同 x
ILEN	内在函数	ILEN(*i*) 返回一个整数 2 的补码的长度（以位为单位）	*i*: 整型 结果:类型和 i 一致
KIND	内在函数	KIND(x) 返回参数 x 的种别值	x: 任何内部类型 结果: 整型
LOC	内在函数	LOC(a) 返回 a 的地址，a 可以是变量，函数调用，表达式或常数	a: 如何类型 结果：INTEGER(4)
%LOC	内在函数	和 LOC 一致	
MAXEXPONENT	内在函数	MAXEXPONENT(x) 返回同 x 的十进制最大正数的指数	x: 实型 结果: INTEGER(4)

续表

名 称	过程类型	描 述	参数/函数类型
MINEXPONENT	内在函数	MINEXPONENT(x) 返回同 x 的十进制最大负数的指数	x: 实型 结果: INTEGER(4)
NARGS	运行时间函数	NARGS( ) 返回命令行所有参数个数，包括命令本身	结果: INTEGER(4)
PRECISION	内在函数	PRECISION(x) 返回同 x 的数据的有效位	x: 实型或复型 结果: INTEGER(4)
PRESENT	内在函数	PRESENT(a) 检查是否有可选参数出现	a: 任何类型 结果: 逻辑型
RADIX	内在函数	RADIX(x) 返回同 x 的数据的基数	x: 整型 或 实型 结果: INTEGER(4)
RANGE	内在函数	RANGE(x) 返回同 x 的数据十进制指数范围	x: 整型,实型或复型 结果: INTEGER(4)
SELECTED_ INT_KIND	内在函数	SELECTED_INT_KIND(r) 返回在 r 的范围内参数的种别值	r: 整型 结果: 整型
SELECTED_ REAL_KIND	内在函数	SELECTED_REAL_KIND([p], [r]) 返回（可选）p 位或（可选）r 指数范围的实数的种别。至少需要一个可选参数	p: 整型 r: 整型 结果: 整型
SIZEOF	内在函数	SIZEOF(x) 返回储存参数所需要的字节数	x: 任何类型 结果: INTEGER(4)
TINY	内在函数	TINY(x) 返回和 x 类型相同的数据所能表示的最小正数	x: 实型 结果: 同 x

内存分配和释放

名 称	过程类型	描 述	参数/函数类型
ALLOCATE	语句	动态分配数组维数	
ALLOCATED	内在函数	ALLOCATED(array)判断一个可分配的数组是否已被分配	array：可分配数组 结果：逻辑量
DEALLOCATE	语句	释放在 ALLOCATE 语句中保留的存储空间	
FREE	内在子函数	FREE(addr)释放整数指针 addr,所指定的内存块	addr: INTEGER(4)
MALLOC	内在函数	MALLOC(size)分配大小为 size 字节的内存块并返回这个块的整型指针	size: INTEGER(4) 结果: INTEGER(4)

数组过程

注：方括号[...]代表可选参数。

名 称	过程类型	描 述	参数/函数类型
ALL	内在函数	ALL(mask[, dim]) 确定是否所有数组值沿着（可选）dim 维都满足 mask 条件	mask: 逻辑型 im: 整型 结果: 逻辑型，如果无 dim 或 mask 是一维的为标量，否则为比 mask 少一维的数组
ANY	内在函数	ANY(mask[, dim]). 确定是否有数组值沿着（可选）dim 维满足 mask 条件	mask: 逻辑型 dim: 整型 结果: 逻辑型，如果无 dim 或 mask 是一维的为标量，否则为比 mask 少一维的数组
BSEARCHQQ	运行时间函数	BSEARCHQQ(adr1, adr2, length, size) 对非结构数据类型的排列一维数组中的指定元素进行二进制查询	adr1: INTEGER(4) adr2: INTEGER(4) length: INTEGER(4) size: INTEGER(4) 结果: INTEGER(4)
COUNT	内在函数	COUNT(mask[, dim]). 清点沿着（可选）dim 维满足 mask 条件的数组元素个数	mask: 逻辑型 dim: 整型 结果: 整型，如果无 dim 或 mask 是一维的为标量，否则为比 mask 少一维的数组
CSHIFT	内在函数	CSHIFT(array, shift [, dim]). 对沿着（可选）dim 维进行循环替换	array: 任何类型 shift: 整型 dim: 整型 结果:和数组的类型和形状相同
DIMENSION	语句	确定数组变量并指定元素个数	
DOT_PRODUCT	内在函数	DOT_PRODUCT(vector_a, vector_b). 对向量（一维数组）实施点乘	vector_a:除字符型的任何类型的向量 vector_b: 类型和大小和 vector_a 相同 结果:和 vector_a 类型相同的标量
EOSHIFT	内在函数	EOSHIFT(array, shift [, boundary] [, dim]). 沿着（可选）dim 维替换掉末尾元素并复制（可选）边界值到另一末尾	array: 任何类型 shift: 整型 boundary:同 array dim: 整型 结果:类型和形状和 array 相同
LBOUND	内在函数	LBOUND(array [, dim]). 返回沿着（可选）dim 维的下界	array: 任何类型 dim: 整型 结果: 整型，如果无 dim 或 array 为一维时为标量，否则为向量
MATMUL	内在函数	MATMUL(matrix_a, matrix_b). 对矩阵（二维数组）实施矩阵乘	matrix_a: 除字符型的任何类型 matrix_b:同 matrix_a 结果:同 matrix_a
MAXLOC	内在函数	MAXLOC(array[, dim] [, mask]). 返回沿着 dim 维（可选）满足 mask 条件（可选）的数组最大值的位置	array: 整型 或 实型 dim: 整型 mask: 逻辑型 结果: 整型向量，大小和数组的维数相同

续表

名 称	过程类型	描 述	参数/函数类型
MAXVAL	内在函数	MAXVAL(array [, dim] [, mask]). 返回沿着 dim 维（可选）满足 mask 条件（可选）的数组最大值	array: 整型 或 实型 dim: 整型 mask: 逻辑型 结果: 同 array，如果无 dim 或 array 为一维时为一维数组，否则比 array 少一维
MERGE	内在函数	MERGE(tsource, fsource, mask). 根据 mask 条件合并两个数组	tsource: 任何类型 fsource: 类型和形状同 tsource mask: 逻辑型 结果: 类型和形状同 tsource
MINLOC	内在函数	MINLOC(array [, dim] [, mask]). *返回沿着 dim 维（可选）*满足 mask 条件（可选）的数组最小值的位置	array: 整型 或 实型 dim: 整型 mask: 逻辑型 结果: 整型向量，大小和数组的维数相同
MINVAL	内在函数	MINVAL(array [, dim] [, mask]). 返回*沿着 dim 维（可选）*满足 mask 条件（可选）的数组最小值	array: 整型 或 实型 dim: 整型 mask: 逻辑型 结果: 同 array，如果无 dim 或 array 为一维时为一维数组，否则比 array 少一维
PACK	内在函数	PACK(array, mask [,vector]). 使用 mask 条件把数组展成和 vector 大小相同的向量	array: 任何类型 mask: 逻辑型 vector: 同 array 结果:类型和 array 相同的向量
PRODUCT	内在函数	PRODUCT(array [, dim] [, mask]). 返回沿着（可选）dim 维满足 mask 条件的数组元素的乘积	array: 整型、实型或复型 dim:整型 mask: 逻辑型 结果:类型同 array，如果无 dim 或 array 为一维时为一维数组，否则比 array 少一维
RESHAPE	内在函数	RESHAPE(source, shape [, pad] [, order]). 用下标 order（可选）改变数组形状，填充数组 pad 的元素（可选）	source: 任何类型 shape: 整型 pad: 同 source order: 整型 结果: 类型同 source 形状同 shape
SHAPE	内在函数	SHAPE(source). 返回数组的形状	source: 任何类型 结果:和 source 类型相同的向量
SIZE	内在函数	SIZE(array [, dim]). 沿着 dim 维（可选）返回数组的长度	array: 任何类型 dim: 整型 结果: 整型标量

续表

名 称	过程类型	描 述	参数/函数类型
SORTQQ	*运行时间子程序*	*CALL SORTQQ(addr, count, size).* *排列无结构数据类型的一维数组（不允许派生类型）*	addr: 整型(4) count: INTEGER(4) size: INTEGER(4)
SPREAD	内在函数	SPREAD(source, dim, ncopies). 通过增加一维来复制数组	source: 任何类型 dim: 整型 ncopies: 整型 结果:类型同 source，但多一维
SUM	内在函数	SUM(array [, dim] [, mask]). 对沿着 dim 维（可选）满足 mask 条件（可选）的数组元素求和	array: 整型、实型或复型 dim: 整型 mask: 逻辑型 结果:类型同 array，如果无 dim 或 array 为一维时为一维数组，否则比 array 少一维
TRANSPOSE	内在函数	TRANSPOSE(matrix). 转置矩阵（二维数组）	matrix: 任何类型 结果: matrix 的转置
UBOUND	内在函数	UBOUND(array [, dim]). 返回沿着（可选）dim 维的上界	array: 任何类型 dim: 整型 结果: 整型，如果无 dim 或 array 为一维时为标量，否则为向量
UNPACK	内在函数	UNPACK(vector, mask, field). 在 mask 条件下把向量元素与 field 中的值填充到数组中	vector: 任何类型 mask: 逻辑型 field: 同 vector 结果:类型同 vector 形状同 mask

数值和类型转换过程

注：方括号[...]代表可选参数。

名 称	过程类型	描 述	参数/函数类型
ABS	内在函数	ABS(a). 返回 a 的绝对值。 当 ABS 作为参数传递时 a 必须为 REAL(4).	A: 整型，实型或复型 结果: 类型同 a，除了复型的结果为实型
AIMAG	内在函数	AIMAG (z). 返回复数的虚部	z: COMPLEX(4) 结果: REAL(4)
AINT	内在函数	AINT (a[, kind]). 截取 a 为指定种别（可选）的整数。 当 AINT 作为参数传递时 a 必须为 REAL(4)	a: 实型 kind: 整型 结果: 种别为 kind（如果存在）实型，否则同 a

续表

名 称	过程类型	描 述	参数/函数类型
AMAX0	内在函数	AMAX0(a1, a2 [, a3...]). 返回整型参数中最大的值的实型值	所有的 a: INTEGER(4) 结果: REAL(4)
AMIN0	内在函数	AMIN0(a1, a2 [, a3...]). 返回整型参数中最大的值的实型值	所有的 a: INTEGER(4) 结果: REAL(4)
ANINT	内在函数	ANINT(a [, kind]). 舍入到指定种别（可选）的整数 当 ANINT 作为参数传递时 a 必须为 REAL(4)	a: 实型 kind: 整型 结果: 种别为 kind（如果存在）实型，否则同 a
CEILING	内在函数	CEILING(a). 返回比 a 大的最小整数	a: 实型 结果: INTEGER(4)
CMPLX	内在函数	CMPLX(a). 返回比 a 大的最小整数	a: 实型 结果: INTEGER(4)
CONJG	内在函数	CONJG(z). 返回复数的共轭	z: COMPLEX(4) 结果: COMPLEX(4)
DBLE	内在函数	DBLE(a). Converts a to double precision type.	A: 整型，实型或复型 结果: REAL(8)
DCMPLX	*内在函数*	DCMPLX(x [,y]). 把参数转换为双精度类型	x: 整型，实型或复型. Y: 整型或实型（如果 x 是复型就不能出现） 结果:双精度复型
DFLOAT	*内在函数*	DFLOAT(a). 把整数转换为双精度类型	a: 整型 结果: REAL(8)
DIM	内在函数	DIM(x, y). 如果 x-y 为正则返回 x-y；否则为 0。如果 DIM 作为参数传递，a 必须为 REAL(4)	x: 整型 或 实型 y: 同 x 结果: 同 x
DPROD	内在函数	DPROD(x, y). 返回单精度的 x 和 y 的双精度乘积	x: REAL(4) y: REAL (4) 结果: REAL(8)
FLOAT	内在函数	FLOAT(I). 把 I 转换为 REAL(4).	I: 整型 结果: REAL(4)
FLOOR	内在函数	FLOOR(a). 返回比 a 小或相等的最大整数	a: 实型 结果: 整型
IFIX	内在函数	IFIX(a). 通过截断把单精度实参数转换为整型参数	x: REAL(4) 结果: 缺省整型 (通常为 INTEGER(4))
IMAG	*内在函数*	*同 AIMAG.*	
INT	内在函数	INT(a [, kind]). 转换一个值为整型	a: 整型、实型，或 复型 kind: 整型 结果: 种别为 kind（如果存在）的整型；否则同 a

续表

名 称	过程类型	描 述	参数/函数类型
LOGICAL	内在函数	LOGICAL(l [, kind]). 在两个种别为 kind（可选）逻辑参数之间转换	l: 逻辑型 kind: 整型
MAX	内在函数	MAX(a1, a2 [, a3...]). 返回参数当中的最大值	所有的 a: 任何类型 结果: 同 a
MAX1	内在函数	MAX1(a1, a2 [, a3...]). 返回参数当中的最大值的整型形式	所有的 a: Real(4) 结果: 整型
MIN	内在函数	MIN(a1, a2 [, a3...]). 返回参数间最小的参数	所有的 a: 任何类型 结果: same type as a
MIN1	内在函数	MIN1(a1, a2 [, a3...]). 返回参数当中的最小值的整型形式	所有的 a: REAL(4) 结果: 整型
MOD	内在函数	MOD(a, p). 返回 a/p 的余数。当 MOD 被作为参数传递时必须为整数	a: 整型或实型 p: 同 a 结果: 同 a
MODULO	内在函数	MODULO(a, p). a 对 b 的模	a: 整型或实型 p: 同 a 结果: 同 a
NINT	内在函数	NINT(a [, kind]). 返回最接近 a 的整数	a: 实型 kind: 整型 结果: 种别为 kind（如果存在）的整型
REAL	内在函数	REAL(a [, kind]). 把一个值转换为实型	a: 整型,实型或复型 结果: REAL(4)
SIGN	内在函数	SIGN(a, b). 返回 a 倍的 b 的 SIGN 值	a: 整型或实型 b: 同 a 结果: 同 a
SNGL	内在函数	SNGL(a). 把双精度参数转换为单精度实型	a: REAL(8) 结果: REAL(4)
TRANSFER	内在函数	TRANSFER(source, mold[, size]). 用数组大小（可选）传递第一个参数给第二个参数类型	source: 任何类型 mold: 任何类型 size: 整型 结果: 同 mold
ZEXT	*内在函数*	ZEXT(x). 以 0 扩展 x	a: 逻辑型或整型 结果: 缺省整型（通常为 INTEGER(4)）

三角、指数、根和对数过程

名称	过程类型	描述	参数/函数类型
ACOS	内在函数	ACOS(x). 返回 x（从 0 到 π）的反余弦，如果 ACOS 被以参数传递则 x 必须为 REAL(4).	x: 实型 结果: 同 x
ACOSD	内在函数	ACOSD(x). 返回 x（从 0 到 180 度）的反余弦，如果 ACOS 被以参数传递则 x 必须为 REAL(4).	x: 实型 结果: 同 x
ALOG	内在函数	ALOG(x). 返回 x 的自然对数	x: REAL(4) 结果: REAL(4)
ALOG10	内在函数	ALOG10(x). 返回 x 的一般对数（10 为底）	x: REAL(4) 结果: REAL(4)
ASIN	内在函数	ASIN(x). 返回 x（从-π/2 到π/2）的反正弦，如果 ASIN 被以参数传递则 x 必须为 REAL(4).	x: 实型 结果: 同 x
ASIND	内在函数	ASIND(x). 返回 x（从-90 度到 90 度）的反正弦，如果 ASIN 被以参数传递则 x 必须为 REAL(4).	x: 实型 结果: 同 x
ATAN	内在函数	ATAN(x). 返回 x（从-π/2 到π/2）的反正切，如果 ATAN 被以参数传递则 x 必须为 REAL(4).	x: 实型 结果: 同 x
ATAND	内在函数	ATAND(x). 返回 x（从-90 度到 90 度）的反正切，如果 ATAND 被以参数传递则 x 必须为 REAL(4).	x: 实型 结果: 同 x
ATAN2	内在函数	ATAN2(y, x). 返回 y/x（从-π到π）的反正切，如果 ATAN2 被以参数传递则 x 必须为 REAL(4).	y: 实型 x: 同 y 结果: 同 y
ATAN2D	内在函数	ATAN2D(y, x). 返回 y/x（从-180 度到 180 度）的反正切，如果 ATAN2D 被以参数传递则 x 必须为 REAL(4).	y: 实型 x: 同 y 结果: 同 y
CCOS	内在函数	CCOS(x). 返回 x 的复型余弦	x: OMPLEX(4) 结果: COMPLEX(4)
CDCOS	内在函数	CDCOS(x). 返回 x 的双精度余弦	x: OMPLEX(8) 结果: COMPLEX(8)
CDEXP	内在函数	CDEXP(x). 返回 e**x.的双精度复型值	x: OMPLEX(8) 结果: COMPLEX(8)
CDLOG	内在函数	CDLOG(x). 返回 x 的双精度型的自然对数	x:COMPLEX(8) 结果: COMPLEX(8)

续表

名 称	过程类型	描 述	参数/函数类型
CDSIN	内在函数	CDSIN(x). 返回 x 的双精度正弦	x: COMPLEX(8) 结果: COMPLEX(8)
CDSQRT	内在函数	CDSQRT(x). 返回 x 的复型双精度平方根	x COMPLEX(8) 结果: COMPLEX(8)
CEXP	内在函数	CEXP(x). 返回 e**x.的复型值	x: COMPLEX(4) 结果: COMPLEX(4)
CLOG	内在函数	CLOG(x). 返回 x 的复型自然对数	x: COMPLEX(4) 结果: COMPLEX(4)
COS	内在函数	COS(x). 返回 x 余弦。如果 COS 被以参数传递则 x 必须为 REAL(4).	x: 实型 或 复型 结果: 同 x
COSD	内在函数	COSD(x). 返回 x（以度为单位）余弦。如果 COSD 被以参数传递则 x 必须为 REAL(4).	x: 实型 结果: 同 x
COSH	内在函数	COSH(x). 返回 x 的双曲余弦。如果 COSH 被以参数传递则 x 必须为 REAL(4).	x: 实型 结果: 同 x
COTAN	内在函数	COTAN (x). 返回 x 的余切	x: Real 结果: 同 x
CSIN	内在函数	CSIN(x). 返回 x 的复型正弦	x: COMPLEX(4) 结果: COMPLEX(4)
CSQRT	内在函数	CSQRT(x). 返回 x 的复型平方根	x: COMPLEX(4) 结果: COMPLEX(4)
DACOS	内在函数	DACOS(x). 返回 x（从 0 到 π）的双精度反余弦	x: REAL(8) 结果: REAL(8)
DACOSD	内在函数	DACOSD(x). 返回 x（从 0 度到 180 度）的反正切，如果 DACOSD 被以参数传递则 x 必须为 REAL(4).	x: REAL(8) 结果: REAL(8)
DASIN	内在函数	DASIN(x). 返回 x（从- π /2 到 π /2）的双精度反正弦	x: REAL(8) 结果: REAL(8)
DASIND	内在函数	DASIND(x). 返回 x（从 0 到 180 度）的双精度反正弦	x: REAL(8) 结果: REAL(8)
DATAN	内在函数	DATAN(x). 返回 x（从- π /2 到 π /2）的双精度反正切	x: REAL(8) 结果: REAL(8)

续表

名 称	过程类型	描 述	参数/函数类型
DATAND	内在函数	DATAND(x). 返回 x（从-90 到 90 度）的双精度反正切	x: REAL(8) 结果: REAL(8)
DATAN2	内在函数	DATAN2(y, x). 返回 y/x（从-π 到 π）的双精度反正切	y: REAL(8) x: REAL(8) 结果: REAL(8)
DATAN2D	内在函数	DATAN2D(y, x). 返回 y/x（从-180 到 180）的双精度反正切	y: REAL(8) x: REAL(8) 结果: REAL(8)
DCOS	内在函数	DCOS(x). 返回 x（弧度）的双精度余弦	x: REAL(8) 结果: REAL(8)
DCOSD	内在函数	DCOSD(x). 返回 x（度）的双精度余弦	x: REAL(8) 结果: REAL(8)
DCOSH	内在函数	DCOSH(x). 返回 x 的双精度双曲余弦	x: REAL(8) 结果: REAL(8)
DCOTAN	内在函数	DCOTAN(x). 返回 x 的双精度双曲余切	x: REAL(8) 结果: REAL(8)
DEXP	内在函数	DEXP(x). 返回 e**x 的双精度值	x: REAL(8) 结果: REAL(8)
DLOG	内在函数	DLOG(x). 返回 x 的双精度自然对数值	x: REAL(8) 结果: REAL(8)
DLOG10	内在函数	DLOG10(x). 返回 x（以 10 为底）的双精度一般对数	x: REAL(8) 结果: REAL(8)
DSIN	内在函数	DSIN(x). 返回 x（弧度）的双精度正弦	x: REAL(8) 结果: REAL(8)
DSIND	*内在函数*	*DSIND(x).* *返回 x（度）的双精度正弦*	x: REAL(8) 结果: REAL(8)
DSINH	内在函数	DSINH(x). 返回 x 的双精度双曲正弦	x: REAL(8) 结果: REAL(8)
DSQRT	内在函数	DSQRT(x). 返回 x 的双精度平方根	x: REAL(8) 结果: REAL(8)
DTAN	*内在函数*	*DTAN(x).* *返回 x（弧度）的双精度正切*	x: REAL(8) 结果: REAL(8)
DTAND	内在函数	DTAND(x). 返回 x（度）的双精度正切	x: REAL(8) 结果: REAL(8)
DTANH	内在函数	DTANH(x). 返回 x 的双精度双曲正切	x: REAL(8) 结果: REAL(8)
EXP	内在函数	EXP(x). 返回 e**x 的值。当 EXP 被以参数传递时 x 必须是 REAL(4)	x: 实型或复型 结果: 同 x

续表

名 称	过程类型	描 述	参数/函数类型
LOG	内在函数	LOG(x) 返回 x 的自然对数	x: 实型或复型 结果: 同 x
LOG10	内在函数	LOG10(x). 返回 x（以 10 为底）的一般对数	x: 实型 结果: 同 x
SIN	内在函数	SIN(x). 返回 x（弧度）的正弦。当 SIN 被以参数传递时 x 必须是 REAL(4)	x: 实型或复型 结果: 同 x
SIND	内在函数	SIND(x). 返回 x（度）的正弦。当 SIND 被以参数传递时 x 必须是 REAL(4)	x: 实型 结果: 同 x
SINH	内在函数	SINH(x). 返回 x 的双曲正弦。当 SINH 被以参数传递时 x 必须是 REAL(4)	x: 实型 结果: 同 x
SQRT	内在函数	SQRT(x). 返回 x 的平方根。当 SQRT 被以参数传递时 x 必须是 REAL(4)	x: 实型或复型 结果: 同 x
TAN	内在函数	TAN(x). 返回 x（弧度）的正切。当 TAN 被以参数传递时 x 必须是 REAL(4)	x: 实型 结果: 同 x
TAND	内在函数	TAND(x). 返回 x（度）的正切。当 TAND 被以参数传递时 x 必须是 REAL(4)	x: 实型 结果: 同 x
TANH	内在函数	TANH(x). 返回 x 的双曲正切。当 TANH 被以参数传递时 x 必须是 REAL(4)	x: 实型 结果: 同 x

浮点查询和控制过程

名 称	过程类型	描 述	参数/函数类型
DIGITS	内在函数	DIGITS(x). 返回类型同 x 的数据的有效位数	x:整型或实型 结果: 整型
EPSILON	内在函数	EPSILON(x). 返回加上类型同 x 的数据后大于 x 的最小正数	x: 实型 结果: 同 x
EXPONENT	内在函数	EXPONENT(x). 返回 x 表示的指数部分	x: 实型 结果: 整型
FRACTION	内在函数	FRACTION(x). 返回 x 表示的小数部分	x: 实型 结果: 同 x
GETCONTROLFPQQ[1]	运行时子程序	CALL GETCONTROLFPQQ(control). 返回浮点过程控制字的值	control: INTEGER(2)
GETSTATUSFPQQ[1]	运行时子程序	CALL GETSTATUSFPQQ(status). 返回浮点过程状态字的值	status: INTEGER(2)
HUGE	内在函数	HUGE(x). 返回可由 x 表示的最大数	x: 整型或实型 结果: 同 x

续表

名 称	过程类型	描 述	参数/函数类型
LCWRQQ[1]	运行时子程序	同 SETCONTROLFPQQ.	
MAXEXPONENT	内在函数	MAXEXPONENT(x). 返回类型同 x 的数据的最大十进制指数	x: 实型 结果: INTEGER(4)
MINEXPONENT	内在函数	MINEXPONENT(x). 返回类型同 x 的数据的最大负十进制指数	x: 实型 结果: INTEGER(4)
NEAREST	内在函数	NEAREST(x, s). 返回 x 在 s 符号方向最小的不同的机器可表示的数	x: 实型 s: 实型且非 0 结果: 同 x
PRECISION	内在函数	PRECISION(x). 返回类型同 x 的数据的有效位数	x:实型或复型 结果: INTEGER(4)
RADIX	内在函数	RADIX(x). 返回类型同 x 的数据的底	x: 整型或实型 结果: INTEGER(4)
RANGE	内在函数	RANGE(x). 返回类型同 x 的十进制指数范围	x: 整型实型或 复型 结果: INTEGER(4)
RRSPACING	内在函数	RRSPACING(x). 返回接近 x 的数据间隔的倒数	x: 实型 结果: 同 x
SCALE	内在函数	SCALE(x, i). x 和 2 的 I 次方的乘积	x: 实型 i: 整型 结果: 同 x
SCWRQQ[1]	运行时子程序	同 GETCONTROLFPQQ.	
SETCONTROLFPQQ[1]	运行时间子程序	CALL SETCONTROLFPQQ (controlword). 设置浮点处理器控制字的值	controlword: INTEGER(2)
SET_EXPONENT	内在函数	SET_EXPONENT(x, i). 返回小数部分为 x，指数部分为 i 的数	x: 实型 i: 整型 结果: 同 x
SPACING	内在函数	SPACING(x). 返回接近 x 的数据间隔	x: 实型 结果: 同 x
SSWRQQ[1]	运行时间子程序	同 GETSTATUSFPQQ.	
TINY	内在函数	TINY(x). 返回以类型 x 可以表示的最小正数	x: 实型 结果: 同 x

字符过程

注：方括号[...]代表可选参数。带星号的过程可以作为其它调用过程的参数

名 称	过程类型	描 述	参数/函数类型
ACHAR	内在函数	ACHAR(i). 返回 ASCII 串位置 i 的字符	i: 整型 0 到 255 结果: CHARACTER(1)
ADJUSTL	内在函数	ADJUSTL(string). 调整左侧，移动引导空格并插入拖后空格	string: Character*(*) 结果: 同 string
ADJUSTR	内在函数	ADJUSTR(string). 调整右侧，移动引导空格并插入拖后空格	string: Character*(*) 结果: 同 string
CHAR	内在函数	CHAR(i [, kind]). 返 kind（可选）字符串中位置 i 的字符	i: 整型 kind: 整型 结果: 种别为 kind（如果存在）的 CHARACTER(1)，否则为缺省种别
IACHAR	内在函数	IACHAR(c). 返回 c 在 ASCII 序列的位置	c: CHARACTER(1) 结果: 整型
ICHAR	内在函数	ICHAR(c). 返回 c 在当前字符串中的位置	c: CHARACTER(1) 结果: 整型
INDEX*	内在函数	INDEX(string, substring , back). 返回 string 中子字符串 substring 的起始位置	string: Character*(*) substring: Character*(*) back: 逻辑型 结果: 整型
LEN*	内在函数	LEN(string). 返回变量 string 的大小	string: Character*(*) 结果: 整型
LEN_TRIM	内在函数	LEN_TRIM(string). 返回字符串中字符数，不包括拖后空格	string: Character*(*) 结果: 整型
LGE	内在函数	LGE(string_a, string_b). 测试哪个字符在 ASCII 序列中后出现，如果相等或 string_a 在后则为 TRUE	string_a: Character*(*) string_b: Character*(*) 结果: 逻辑型
LGT	内在函数	LGT(string_a, string_b). 测试哪个字符在 ASCII 序列中后出现，如果 string_a 在后则为 TRUE	string_a: Character*(*) string_b: Character*(*) 结果: 逻辑型
LLE	内在函数	LLE(string_a, string_b). 测试哪个字符在 ASCII 序列中后出现，如果相等或 string_a 在前则为 TRUE	string_a: Character*(*) string_b: Character*(*) 结果: 逻辑型
LLT	内在函数	LLT(string_a, string_b). 测试哪个字符在 ASCII 序列中后出现，如果 string_a 在前则为 TRUE	string_a: Character*(*) string_b: Character*(*) 结果: 逻辑型
REPEAT	内在函数	REPEAT(string, ncopies). 连接 string 的多个拷贝	string: Character*(*) ncopies: 整型 结果: Character*(*)
SCAN	内在函数	SCAN(string, set [, back]). 扫描 string 查找 set 中的任何字符并返回匹配的最左侧或最右侧（可选）的位置	string: Character*(*) set: Character*(*) back: 逻辑型 结果: 整型
TRIM	内在函数	TRIM(string). 去除 string 的拖后空格	string: Character*(*) 结果: Character*(*)
VERIFY	内在函数	VERIFY(string, set [, back]). 返回出现在 string 中而不出现在 set 中的字符的最左侧或最右侧（可选）的位置，如果所有的字符都出现则返回 0	string: Character*(*) set: Character*(*) back: 逻辑型 结果: 整型

位操作过程

注：方括号[...]代表可选参数。

名 称	过程类型	描述	参数/函数类型
BIT_SIZE	内在函数	BIT_SIZE(i). 返回类型为 i 的整数的位数	i: 整型 结果: 同 i
BTEST	内在函数	BTEST(i, pos). 测试 i 的第 poc 位，如果是 1 则返回真	i: 整型 pos:正整型 结果: 逻辑型
IAND	内在函数	IAND(i, j). 逻辑与	i: 整型 j: 整型 结果: 同 i
IBCHNG	内在函数	IBCHNG(i, pos). 对 i 的第 pos 位取反	i: 整型 pos:正整型 结果: 同 i
IBCLR	内在函数	IBCLR(i, pos). 把 i 的第 pos 位清零	i: 整型 pos:正整型 结果: 同 i
IBITS	内在函数	IBITS(i, pos, len). 对 i，从第 pos 位起取长度为 len 的位序列	i: 整型 pos:正整型 len: 正整型 结果: 同 i
IBSET	内在函数	IBSET(i, pos). 置 i 的第 pos 位为 1	i: 整型 pos:正整型 结果: 同 i
IEOR	内在函数	IEOR(i, j). 异或操作	i: 整型 j: 整型 结果: 同 i
IOR	内在函数	IOR(i, j). 或操作	i: 整型 j: 整型 结果: 同 i
ISHA	内在函数	ISHA(i, shift). 对 i 算术移位 shift 位；如果 shift 为正则左移，为负则右移。 0 从右侧移入，1 从左侧移入	i: 整型 shift: 整型 结果: 同 i
ISHC	内在函数	ISHC(i, shift). 对 i 循环移位 shift 位；如果 shift 为正则左移，为负则右移	i: 整型 shift: 整型 结果: 同 i
ISHFT	内在函数	ISHFT(i, shift). 对 i 逻辑移位 shift 位；如果 shift 为正则左移，为负则右移。 0 从相反方向移入	i: 整型 shift: 整型 结果: 同 i

续表

名 称	过程类型	描述	参数/函数类型
ISHFTC	内在函数	ISHFTC(i, shift[, size]). 对 i 最右侧 size（可选）位进行循环移位 shift 位	i: 整型 shift: 整型 size:正整型 结果: 同 i
ISHL	内在函数	ISHL(i, shift). 对 i 逻辑移位 shift 位；如果 shift 为正则左移，为负则右移。 0 从相反方向移入	i: 整型 shift: 整型 结果: 同 i
MVBITS	内在子函数	MVBITS(from, frompos, len, to, topos). 复制一个整数的位序列到另一个整数	from: 整型 frompos:正整型 to: 整型 topos:正整型
NOT	内在函数	NOT(i). 逻辑补操作	i: 整型 结果: 同 i

QuickWin 过程

注：使用下列过程的程序必须用 USE DFLIB 访问适当的库。

名 称	过程类型	描 述
ABOUTBOXQQ	QuickWin 函数	添加有定制文本的 About 对话框
APPENDMENUQQ	QuickWin 函数	添加一个菜单项
CLICKMENUQQ	QuickWin 函数	把点击菜单的消息传给应用程序窗口
DELETEMENUQQ	QuickWin 函数	删除一个菜单项
FOCUSQQ	QuickWin 函数	使一个子窗口变为活动的，并给它焦点
GETACTIVEQQ	QuickWin 函数	获得活动子窗口的单元号
GETHWNDQQ	QuickWin 函数	从有指定单元号的窗口获得真正窗口的句柄
GETWINDOWCONFIG	QuickWin 函数	返回当前窗口属性
GETWSIZEQQ	QuickWin 函数	获得子窗口或帧窗口的大小
INCHARQQ	QuickWin 函数	从键盘读输入并返回其 ASCII 码值
INITIALSETTINGS	QuickWin 函数	控制初始菜单和初始帧窗口设置
INQFOCUSQQ	QuickWin 函数	确定哪个窗口是活动并具有焦点
INSERTMENUQQ	QuickWin 函数	插入一个菜单项
INTEGERTORGB	QuickWin 子例程	把实际的颜色转换为红、绿、蓝成份
MESSAGEBOXQQ	QuickWin 函数	显示信息对话框
MODIFYMENUFLAGSQQ	QuickWin 函数	改变菜单项状态
MODIFYMENUROUTINEQQ	QuickWin 函数	改变菜单项的反馈例程
MODIFYMENUSTRINGQQ	QuickWin 函数	改变菜单项的文本字符串

续表

名 称	过程类型	描 述
PASSDIRKEYSQQ	QuickWin 函数	决定方向键和翻页键的行为
REGISTERMOUSEEVENT	QuickWin 函数	登记程序定义的例程为鼠标事件的被调程序
RGBTOINTEGER	QuickWin 函数	把红绿蓝值转换为实际颜色值，以供其它 RGB 函数和子例程使用
SETACTIVEQQ	QuickWin 函数	使指定窗口为当前活动窗口而不给它焦点
SETMESSAGEQQ	QuickWin 子例程	改变所有 QuickWin 信息，包括状态条信息、状态信息和对话框信息
SETWINDOWCONFIG	QuickWin 函数	配置当前窗口属性
SETWINDOWMENUQQ	QuickWin 函数	设置 Window 菜单，当前子窗口名称被加入
SETWSIZEQQ	QuickWin 函数	设置子窗口或帧窗口的大小
UNREGISTERMOUSEEVENT	QuickWin 函数	取消 REGISTERMOUSEEVENT 登记的反馈例程
WAITONMOUSEEVENT	QuickWin 函数	等待直到鼠标事件发生

图形过程

注：使用下列过程的程序必须用 USE DFLIB 访问适当的库。

名 称	过程类型	描 述
ARC, ARC_W	图形函数	画弧
CLEARSCREEN	图形子例程	清除屏幕、观察口或文本窗口
DISPLAYCURSOR	图形函数	开启或关闭鼠标指针
ELLIPSE, ELLIPSE_W	图形函数	画椭圆或圆
FLOODFILL, FLOODFILL_W	图形函数	用当前颜色索引和当前掩模填充屏幕上的封闭区域
FLOODFILLRGB, FLOODFILLRGB_W	图形函数	用当前 RGB 颜色和当前掩模填充屏幕上的封闭区域
GETARCINFO	图形函数	确定最近画的弧或饼图的终点
GETBKCOLOR	图形函数	返回当前背景颜色索引
GETBKCOLORRGB	图形函数	返回当前背景 RGB 颜色
GETCOLOR	图形函数	返回当前颜色索引
GETCOLORRGB	图形函数	返回当前 RGB 颜色
GETCURRENTPOSITION, GETCURRENTPOSITION_W	图形子例程	返回当前图形输出点的坐标
GETFILLMASK	图形子例程	返回当前掩模
GETFONTINFO	图形函数	返回当前字体特性
GETGTEXTEXTENT	图形函数	确定以当前字体的指定文本的宽度
GETGTEXTROTATION	图形函数	获得当前文本转角
GETIMAGE, GETIMAGE_W	图形子例程	存储一幅屏幕图像到内存中.
GETLINESTYLE	图形函数	返回当前线型
GETPHYSCOORD	图形子例程	把观察口坐标转换为物理坐标
GETPIXEL, GETPIXEL_W	图形函数	返回一个像素的颜色索引

续表

名 称	过程类型	描 述
GETPIXELRGB, GETPIXELRGB_W	图形函数	返回一个像素的 RGB 颜色
GETPIXELS	图形函数	返回多个像素的颜色索引
GETPIXELSRGB	图形函数	返回多个像素的 RGB 颜色
GETTEXTCOLOR	图形函数	返回当前文本颜色索引
GETTEXTCOLORRGB	图形函数	返回当前文本 RGB 颜色
GETTEXTPOSITION	图形子例程	返回当前文本输出位置
GETTEXTWINDOW	图形子例程	返回当前文本窗口的边界
GETVIEWCOORD, GETVIEWCOORD_W	图形子例程	把物理坐标或窗口坐标转换为观察口坐标
GETWINDOWCOORD	图形子例程	把观察口坐标转换为窗口坐标
GETWRITEMODE	图形函数	返回直线的逻辑写模式
GRSTATUS	图形函数	返回最近调用图形例程的状态（成功或失败）
IMAGESIZE, IMAGESIZE_W	图形函数	返回图像的大小（以字节为单位）
INITIALIZEFONTS	图形函数	初始化字体库
LINETO, LINETO_W	图形函数	从当前点画到指定点的直线
LOADIMAGE, LOADIMAGE_W	图形函数	读一幅 Windows 位图文件并在指定位置显示
MOVETO, MOVETO_W	图形子例程	把当前位置移到指定点
OUTGTEXT	图形子例程	把具有当前字体的文本送到指定位置
OUTTEXT	图形子例程	把文本送到屏幕的指定位置
PIE, PIE_W	图形函数	画饼图
POLYGON, POLYGON_W	图形函数	画多边形
PUTIMAGE, PUTIMAGE_W	图形子例程	从内存中恢复一幅图像并显示
RECTANGLE, RECTANGLE_W	图形函数	画矩形
REMAPALLPALETTERGB	图形函数	重新映射 RGB 颜色值到当前显示配置承认的索引
REMAPPALETTERGB	图形函数	重新映射一个 RGB 颜色值到颜色索引
SAVEIMAGE, SAVEIMAGE_W	图形函数	截取屏幕图像并存为位图文件
SCROLLTEXTWINDOW	图形子例程	滚动文本窗口的内容
SETBKCOLOR	图形函数	设置当前背景颜色
SETBKCOLORRGB	图形函数	设置当前背景颜色为一种直接颜色而不是已定义调色板中的颜色索引
SETCLIPRGN	图形子例程	限制图形输出到屏幕的一部分
SETCOLOR	图形函数	设置当前颜色为一种新的颜色索引
SETCOLORRGB	图形函数	设置当前颜色为一种直接颜色而不是已定义调色板中的颜色索引
SETFILLMASK	图形子例程	把当前掩模换成新的式样
SETFONT	图形函数	找一种符合指定特性的字体并设为 OUTGTEXT.
SETGTEXTROTATION	图形子例程	设置文本沿着指定角度写的方向
SETLINESTYLE	图形子例程	改变当前线型
SETPIXEL, SETPIXEL_W	图形函数	设置指定位置的像素颜色
SETPIXELRGB, SETPIXELRGB_W	图形函数	设置指定位置的像素 RGB 颜色

续表

名 称	过程类型	描 述
SETPIXELS	图形子例程	设置多个像素的颜色索引
SETPIXELSRGB	图形子例程	设置多个像素的 RGB 颜色
SETTEXTCOLOR	图形函数	设置当前文本颜色为一个新的颜色索引
SETTEXTCOLORRGB	图形函数	设置当前文本颜色为一种直接颜色而不是已定义调色板中的颜色索引
SETTEXTPOSITION	图形子例程	改变当前文本位置
SETTEXTWINDOW	图形子例程	设置当前文本显示窗口
SETVIEWORG	图形子例程	定观察口坐标的原点
SETVIEWPORT	图形子例程	定义观察口的大小和屏幕位置
SETWINDOWSETWRITEMODE	图形函数	改变直线的当前逻辑写模式
WRAPON	图形函数	打开或关闭直线包装

对话框过程

注：使用下列过程的程序必须用 USE DFLOGM 访问对话框库。方括号[...]代表可选参数。

名 称	过程类型	描 述	参数/函数类型
DLGEXIT	对话框子例程	CALL DLGEXIT(dlg). 关闭一个打开的对话框	dlg: DIALOG 派生类型 结果: 逻辑型
DLGGET	对话框函数	DLGGET(dlg, controlid, value [, index]). 获得对话框控制变量的值	dlg: DIALOG 派生类型 controlid: 整型 value: 整型, 逻辑型 或 字符型 index: 整型 结果: 逻辑型
DLGGETCHAR	对话框函数	DLGGETCHAR(dlg, controlid, value [, index]). 获得对话框显示字符的控制变量	dlg: DIALOG 派生类型 controlid: 整型 value: 字符型 index: 整型 结果: 逻辑型
DLGGETINT	对话框函数	DLGGETINT(dlg, controlid, value [, index]). 获得对话框显示整数的控制变量	dlg: DIALOG 派生类型 controlid: 整型 value: 整型 index: 整型 结果: 逻辑型

续表

名 称	过程类型	描 述	参数/函数类型
DLGGETLOG	对话框函数	DLGGETLOG(dlg, controlid, value [, index]). 获得对话框显示逻辑型数据的控制变量	dlg: DIALOG 派生类型 controlid: 整型 value: 逻辑型 index: 整型 结果: 逻辑型
DLGINIT	对话框函数	DLGINIT(dlgid, dlg). 初始化对话框	dlgid: 整型 dlg: DIALOG 派生类型结果: 逻辑型
DLGMODAL	对话框函数	DLGMODAL(dlg). 显示对话框并处理用户的选择	dlg: DIALOG 派生类型结果: 整型
DLGSET	对话框函数	DLGSET(dlg, controlid, value [, index]). 指定对话框控制变量的值	dlg: DIALOG 派生类型 controlid: 整型 value: 整型, 逻辑型或字符型 index: 整型 结果: 逻辑型
DLGSETCHAR	对话框函数	DLGSETCHAR(dlg, controlid, value [, index]). 指定对话框显示字符的控制变量	dlg: DIALOG 派生类型 controlid: 整型 value: 字符型 index: 整型 结果: 逻辑型
DLGSETINT	对话框函数	DLGSETINT(dlg, controlid, value [, index]). 指定对话框显示整数的控制变量	dlg: DIALOG 派生类型 controlid: 整型 value: 整型 index: 整型 结果: 逻辑型
DLGSETLOG	对话框函数	DLGSETLOG(dlg, controlid, value [, index]). 指定对话框显示逻辑数据的控制变量	dlg: DIALOG 派生类型 controlid: 整型 value: 逻辑型 index: 整型 结果: 逻辑型
DLGSETRETURN	对话框子例程	CALL DLGSETRETURN(dlg, retval). 设置 DLGMODAL 的返回值	dlg: DIALOG 派生类型 retval: 整型
DLGSETSUB	对话框函数	DLGSETSUB(dlg, controlid, value [, index]). 指定对话框控制的过程（反馈例程）	dlg: DIALOG 派生类型 controlid: 整型 value: 外部过程名 index: 整型 结果: 逻辑型
DLGUNINIT	对话框子例程	CALL DLGUNINIT(dlg). 释放初始化对话框所占的内存	dlg: DIALOG 派生类型

编译指令

注：下面每个指令名称的前缀为 cDEC$，例如 cDEC$ALIAS，c 可以是 c, C, !, 或 *。

名 称	描 述
ALIAS	指定引用外部子程序时的别名
ATTRIBUTES	为变量和过程申请属性
DECLARE	给未定义的变量产生警告信息
DEFINE	创建变量，它的存在可以在条件编译时被检测
ELSE	标记一个可选择的条件编译块的开始为 IF 指令结构
ELSEIF	标记一个可选择的条件编译块的开始为 IF 指令结构
ENDIF	标记条件编译块的末尾
FIXEDFORMLINESIZE	设置混合格式的每行的长度。对自由形式的代码无效
FREEFORM	源代码使用自由格式
IDENT	给指定对象模块确定一个标识符
IF	标记条件编译块的开始
IF DEFINED	标记条件编译块的开始
INTEGER	选择缺省整数种别
MESSAGE	把一个字符串送入标准输出设备
NODECLARE	（缺省）关闭未定义变量的警告信息
NOFREEFORM	（缺省）使用标准 FORTRAN 77 代码格式
OPTIONS	控制公共块中记录和数据项中的域是自然结合还是按任意字节结合
PACK	指定派生类型项的内存开始地址
PSECT	改变公共块的某种属性
REAL	选择缺省实型种别
STRICT	禁止不在标准 FORTRAN 90 中的 Visual FORTRAN 特性
SUBTITLE	打印指定副标题
TITLE	打印指定标题
UNDEFINE	清除由 DEFINE 创建的符号变量名

本国语言标准过程

注：使用下列过程的程序必须用 USE DFNLS 访问 NLS 库。

名 称	过程类型	描 述
MBCharLen	NLS 函数	返回字符串中第一个多字节字符的长度
MBConvertMBToUnicode	NLS 函数	把代码页多字节字符串转换为统一字符编码标准字符串
MBConvertUnicodeToMB	NLS 函数	把统一字符编码标准字符串转换为代码页多字节字符串
MBCurMax	NLS 函数	返回当前代码页可能的最长的多字节字符
MBINCHARQQ	NLS 函数	和 INCHARQQ 基本相同，除了可以一次读取一个单独的多字节字符并返回读取的字节数
MBINDEX	NLS 函数	和 INDEX 基本相同，除了参数中可以包含多字节字符
MBJISToJMS	NLS 函数	把 JIS 转换为 JMS
MBJMSToJIS	NLS 函数	把 JMS 转换为 JIS
MBLead	NLS 函数	确定一个给出的字符是否是一个多字节字符串的首字节
MBLen	NLS 函数	返回字符串中的多字节字符数目，包括拖后空格
MBLen_Trim	NLS 函数	返回字符串中的多字节字符数目，不包括拖后空格
MBLGE、MBLGT、MBLLE、MBLLT,MBLEQ, MBLNE	NLS 函数	和 LGE. LGT, LLE、LLT 和.EQ., .NE.操作符基本相同，除了参数中可以包含多字节字符
MBNext	NLS 函数	返回紧接着给出字符串位置后面多字节字符首字节的位置
MBPrev	NLS 函数	返回紧接着给出字符串位置前面多字节字符首字节的位置
MBSCAN	NLS 函数	和 SCAN 基本相同，除了参数中可以包含多字节字符
MBStrLead	NLS 函数	进行上下文敏感测试来确定给定字节是引导字节还是拖后字节
MBVERIFY	NLS 函数	和 VERIFY 基本相同，除了参数中可以包含多字节字符
NLSEnumCodepages	NLS 函数	返回系统支持的所有代码页
NLSEnumLocales	NLS 函数	获得系统支持的所有语言和国别的结合
NLSFormatCurrency	NLS 函数	格式化数字字符串并返回当前现场的正确现金字符串
NLSFormatDate	NLS 函数	返回包含当前现场的日期的正确格式化字符串
NLSFormatNumber	NLS 函数	格式化数字字符串并返回当前现场的正确数字字符串
NLSFormatTime	NLS 函数	返回包含当前现场的时间的正确格式化字符串
NLSGetEnvironmentCodepage	NLS 函数	返回 Window 代码页或控制台代码页的编号
NLSGetLocale	NLS 子例程	获得当前语言、国别和代码页
NLSGetLocaleInfo	NLS 函数	获得关于当前本地编码集要求的信息
NLSSetEnvironmentCodepage	NLS 函数	为当前控制台更改代码页
NLSSetLocale	NLS 函数	设置语言、国别和代码页

可移植性过程

注：使用下列过程的程序必须用 USE DFPORT 访问可移植库。方括号[...]代表可选参数。

名 称	过程类型	描 述	参数/函数类型
ABORT	可移植性子例程	CALL ABORT([string]). 清洗并关闭所有 I/O 缓冲，用 string 中的中止信息（可选）结束执行	string: CHARACTER*(*)
ACCESS	可移植性函数	ACCESS(name, mode). 检查由 name 指定的文件对于以 mode 模式主调者的可访问性	name: CHARACTER*(*) mode: CHARACTER*(*) 结果: INTEGER(4)
ALARM	可移植性函数	ALARM(time, proc). 在 time 秒后执行子例程 proc	time: INTEGER(4) proc:外部过程 结果: INTEGER(4)
BESJ0, BESJ1, BESJN, BESY0, BESY1, BESYN	可移植性函数	第一类和第二类 Bessel 函数 BESJ0(x), BESJ1(x), BESY0(x), BESY1(x) 的参数为 x BESJN(n, x), BESYN(n, x) 参数为 n 和 x	n: INTEGER(4) x: REAL(4) 结果: REAL(4)
BIC, BIS, BIT	可移植性子例程和函数	位清除和设置子例程和位测试函数 CALL BIC(bitnum, target)清除位 CALL BIS(bitnum, target)设置位 BIT(bitnum, source)测试位	bitnum: INTEGER(4) target: INTEGER(4) source: INTEGER(4) BIT 返回: 逻辑型
CHDIR	可移植性函数	CHDIR(dir_name). 改变缺省目录为 dir_name.	dir_name: CHARACTER*(*) 结果: INTEGER(4)
CHMOD	可移植性函数	CHMOD(name, mode). 改变由 name 指定的文件的 mode 属性	name: CHARACTER*(*) mode: CHARACTER*(*) 结果: INTEGER(4)
CLOCK	可移植性函数	CLOCK( ). 以 HH:MM:SS 格式返回时间	结果: CHARACTER(8)
CTIME	可移植性函数	CTIME(stime). 转换系统时间为 24 字节 ASCII 字符串	stime: INTEGER(4) 结果: CHARACTER(24)
DATE	可移植性子例程或函数	CALL DATE(string)或 DATE( ) 以 ASCII 字符串返回日期	string: CHARACTER*(*) DATE 函数返回: CHARACTER(8)
DBESJ0, DBESJ1, DBESJN, DBESY0, DBESY1, DBESYN	可移植性函数	REAL(8) 第一类和第二类 Bessel 函数 DBESJ0(x), DBESJ1(x), DBESY0(x), DBESY1(x) 参数为 x. DBESJN(n, x), DBESYN(n, x) 参数为 n 和 x.	n: INTEGER(4) x: REAL(8) 结果: REAL(8)
DRAND, DRANDM	可移植性函数	DRAND(iflag)和 DRANDM(iflag). 返回 0.0 至 1.0 之间的随机数	iflag: INTEGER(4) 结果: REAL(8)
DTIME[1]	可移植性函数	DTIME(tarray). 返回从上一次调用 DTIME 或程序开始经过的时间	tarray(2): REAL(4) 结果: REAL(4)

续表

名 称	过程类型	描 述	参数/函数类型
ETIME'	可移植性函数	ETIME(tarray). 返回从上一次调用 ETIME 或程序开始经过的 CPU 时间	tarray(2): REAL(4) 结果: REAL(4)
FDATE	可移植性子例程或函数	CALL FDATE(string)或 FDATE( ). 返回以 ASCII 字符串表示的日期和时间	string: CHARACTER*(*) FDATE 函数返回: CHARACTER(24)
FGETC	可移植性函数	FGETC(lunit, char). 从 lunit 中读取下一个字符并放到 char 中	lunit: INTEGER(4) char: CHARACTER(1) 结果: INTEGER(4)
FLUSH	可移植性子例程	CALL FLUSH(lunit). 使 lunit 缓冲中的内容被相联系的文件清洗	lunit: INTEGER(4)
FPUTC	可移植性函数	FPUTC(lunit, char). 把字符 char 写入和 lunit 相联系的文件	lunit: INTEGER(4) char: CHARACTER(1) 结果: INTEGER(4)
FSEEK	可移植性子例程	FSEEK(lunit, offset, from). 重新定位逻辑单元上文件上相对位置 form 的 offset 字节	lunit: INTEGER(4) offset: INTEGER(4) from: INTEGER(4) 结果: INTEGER(4)
FSTAT	可移植性函数	FSTAT(lunit, statb). 返回数组 statb 中关于 lunit 的信息	lunit: INTEGER(4) statb(12): INTEGER(4) 结果: INTEGER(4)
FTELL	可移植性函数	FTELL(lunit). 返回和 lunit 相联系的文件当前位置	lunit: INTEGER(4) 结果: INTEGER(4)
GERROR	可移植性子例程	CALL GERROR(string). 由上次检测到的 IERRNO 错误填充 string	string: CHARACTER*(*)
GETC	可移植性函数	GETC(char). 从逻辑单元 5 中获得下一个可获得的字符（通常连接到控制台），并放在 char 里	char: CHARACTER(1) 结果: INTEGER(4)
GETCWD	可移植性函数	GETCWD(dirname). 把当前工作路径放入 dirname 中	dirname: CHARACTER*(*) 结果: INTEGER(4)
GETENV	可移植性子例程	CALL GETENV(ename, evalue). 查找环境列表中 ename=evalue 的字符串，并返回在 evalue 中找到的值或空格	ename: CHARACTER*(*) evalue: CHARACTER*(*)
GETGID	可移植性函数	GETGID( ). 为可移植性包含，始终返回 1	结果: INTEGER(4) 等于 1
GETLOG	可移植性子例程	CALL GETLOG(name). 返回用户的登录名或空格	name: CHARACTER*(*)
GETPID	可移植性函数	GETPID( ). 返回当前过程的 ID 号	结果: INTEGER(4)

续表

名 称	过程类型	描 述	参数/函数类型
GETUID	可移植性函数	GETUID( ). 为可移植性包含，始终返回 1	结果: INTEGER(4) 等于 1
GMTIME	可移植性子例程	CALL GMTIME(stime, tarray). 分隔由 TIME( )返回的时间 stime 成 GMT 日期和时间并保存在 tarray 中	stime: INTEGER(4) tarray(9): INTEGER(4)
HOSTNAM	可移植性函数	HOSTNAM(name). 把当前宿主名放入 name	name: CHARACTER*(*) 结果: INTEGER(4)
IARGC	可移植性函数	IARGC( ). 返回上一命令行参数的索引	结果: INTEGER(4)
IDATE	可移植性子例程	CALL IDATE(iarray)或 CALL IDATE(month, day, year). 返回 iarray 中的日、月和年或参数 month, day, and year.	iarray(3): INTEGER(4) month: INTEGER(4) day: INTEGER(4) year: INTEGER(4)
IERRNO	可移植性函数	IERRNO( ). 返回上一个 IERRNO 错误码	结果: INTEGER(4)
IRAND, IRANDM	可移植性函数	IRAND(iflag)和 IRANDM(iflag). 返回 0 至(2**31)−1 之间的整型随机数	iflag: INTEGER(4) 结果: INTEGER(4)
ITIME	可移植性子例程	CALL ITIME(iarray). 返回 iarray 中的当前时间	iarray(3): INTEGER(4)
JDATE	可移植性函数	JDATE( ). 返回 YYDDD 形式中的 Julian 日期	结果: CHARACTER(8)
KILL	可移植性函数	KILL(pid, signum). 发送由 signum（SIGNAL 中定义）指定的信号给 pid（GETPID 返回的）指定的主调过程	pid: INTEGER(4) signum: INTEGER(4) 结果: INTEGER(4)
LNBLNK	可移植性函数	LNBLNK(string). 返回 string 中上一个非空格字符索引	string: CHARACTER*(*) 结果: INTEGER(4)
LONG	可移植性函数	LONG(int2). 返回 INTEGER(2)参数为 INTEGER(4)	int2: INTEGER(2) 结果: INTEGER(4)
LSTAT	可移植性函数	LSTAT(name, statb). 返回数组 statb 中名为 name 的文件信息	name: CHARACTER*(*) statb(12): INTEGER(4) 结果: INTEGER(4)
LTIME	可移植性子例程	CALL LTIME(stime, tarray). 分隔由 TIME( )返回的时间 stime 成当地时区日期和时间并保存在 tarray 中	stime: INTEGER(4) tarray(9): INTEGER(4)
PERROR	可移植性子例程	CALL PERROR(string). 把一条错误信息发送给标准错误流	string: CHARACTER*(*)
PUTC	可移植性函数	PUTC(char). 把字符 char 写入 6 号逻辑单元	char: CHARACTER(1) 结果: INTEGER(4)
QSORT	可移植性子例程	CALL QSORT(array, len, isize, compar). 依照用户提供的函数 compar 的分类顺序排列 array 的 len 个元素,每个长度 isize	array: 任何类型 len: INTEGER(4) isize: INTEGER(4) compar: 外部 INTEGER(2) 函数

续表

名 称	过程类型	描 述	参数/函数类型
RAN	可移植性函数	RAN(iseed). 返回 0.0 至 1.0 之间均匀分布的随机数	seed: INTEGER(4) 结果: REAL(4)
RAND, RANDOM	可移植性函数	RAND(iflag) and RANDOM(iflag). 返回 0.0 至 1.0 之间的随机数.	iflag: INTEGER(4) 结果: REAL(4)
RENAME	可移植性函数	RENAME(from, to). 把文件由 from 命名为 to，如果 to 存在则先被清除	from: CHARACTER*(*) to: CHARACTER*(*) 结果: INTEGER(4)
RINDEX	可移植性函数	RINDEX(string, substr). 返回 string 中最后出现 substr 的索引，或 0	string: CHARACTER*(*) substr: CHARACTER*(*) 结果: INTEGER(4)
RTC	可移植性函数	RTC( ). 返回从 1970 年 1 月 1 日 00:00:00 GMT 到当前的秒数	结果: REAL(8)
SECNDS	可移植性函数	SECNDS(offset). 返回从午夜到当前减去 offset 的秒数	offset: REAL(4) 结果: REAL(4)
SHORT	可移植性函数	SHORT(int4). 返回 INTEGER(4)参数为 INTEGER(2)	int4: INTEGER(4) 结果: INTEGER(2)
SIGNAL	可移植性函数	SIGNAL(signum, proc, flag). 改变为由 signum 指定的信号对外部信号处理过程 proc 采取的行动。Proc 的执行由 flag 控制	signum: INTEGER(4) proc:外部函数 flag: INTEGER(4) 结果: INTEGER(4)
SLEEP	可移植性子例程	CALL SLEEP(itime). 挂起主调过程 itime 秒	itime: INTEGER(4)
STAT	可移植性函数	STAT(name, statb). 返回在数组 statb 中名为 name 的文件的相关信息	name: CHARACTER*(*) statb(12): INTEGER(4) 结果: INTEGER(4)
SYSTEM	可移植性函数	SYSTEM(string). 使 string 为作为系统外壳的输入	string: CHARACTER*(*) 结果: INTEGER(4)
TIME	可移植性子例程或函数	CALL TIME(string) 或 TIME( ). 作为子例程，用当前的 HH:MM:SS 格式的时间填充 string；作为函数返回从 1970 年 1 月 1 日 00:00:00 GMT 到当前的秒数	string: CHARACTER(8) 结果: INTEGER(4)
TIMEF	可移植性函数	TIMEF( ). 返回距它第一次调用时的秒数。第一次调用时返回 0.0D0	结果: REAL(8)
UNLINK	可移植性函数	UNLINK(name). 清除由路径 name 指定的文件	name: CHARACTER*(*) 结果: INTEGER(4)

[1] 只适用于 WNT

注意：由 DATE, IDATE, 和 JDATE 返回的两位的年可能会引起 2000 年问题。

其它运行时过程

名 称	过程类型	描 述
FOR_CHECK_FLAWED_PENTIUM	运行时间函数	FOR_CHECK_FLAWED_PENTIUM ( ). 检查处理器来决定是否有 Pentium 浮点缺陷
FOR_GET_FPE	运行时间函数	FOR_GET_FPE ( ). 返回当前浮点异常标志设置
FOR_RTL_FINISH_	运行时间函数	FOR_RTL_FINISH_ ( ). 清除 FORTRAN 运行环境
FOR_RTL_INIT_	运行时间子程序	FOR_RTL_INIT_ (argcount, actarg). 初始化 FORTRAN 运行环境
FOR_SET_FPE	运行时间函数	FOR_SET_FPE (a). 设置浮点异常标志设置
FOR_SET_REENTRANCY	运行时间函数	FOR_SET_REENTRANCY (mode). 控制 FORTRAN 运行库（RTL）的重入保护类型

不能作为实参的函数

下面的特殊函数不能作为实参传递

AIMAX0	DREAL	INT2	KMIN1
AIMIN0	EOF	INT4	LGE
AJMAX0	FLOAT	*JFIX*	LGT
AJMIN0	FLOATI	JIDINT	LLE
AKMAX0	FLOATJ	JIFIX	LLT
AKMIN0	FLOATK	JINT	*LOC*
AMAX0	ICHAR	JMAX0	MALLOC
AMAX1	IDINT	*JMAX1*	MAX0
AMIN0	IFIX	*JMIN0*	MAX1
AMIN1	*IIDINT*	JMIN1	MIN0
CHAR	IIFIX	KIDINT	MIN1
DBLE	IINT	KIFIX	*QEXT*
DBLEQ	IMAX0	KINT	QEXTD
DFLOTI	IMAX1	KIQINT	QMAX1
DFLOTJ	IMIN0	KIQNNT	QMIN1
DFLOTK	IMIN1	KMAX0	REAL
DMAX1	INT	KMAX1	SNGL
DMIN1	INT1	KMIN0	SNGLQ