# Intel Visual Fortran
# 应用程序开发

周振红 徐进军 毕苏萍
苗丽 王宗敏 编著




http://www.fcode.cn
Fortran Coder 程序员聚集地

黄 河 水 利 出 版 社

## 内 容 提 要

本书针对 Compaq Visual Fortran (CVF)6.6 的后继编译器 Intel Visual Fortran (IVF)9.0，系统介绍 Visual Studio.NET 环境下各种 Fortran 应用程序的开发，重点讲解 Fortran QuickWin、Fortran Windows、 动态链接库及多线程应用程序的开发， 以及对话框和控件、 自动化服务器和 ActiveX 控件的使用，并全面、深入地探讨 Intel Fortran 与 Visual C++/Basic.NET 的混合编程。另外，本书对 Visual Studio.NET 开发环境、Intel Fortran 9.0 编译器以及 CVF 向 IVF 的转换等也进行了简要介绍。

本书实例丰富、注重使用，面向具有 Fortran90/95 基础的中、高级读者，适合作为理工科高年级本科生、研究生的教学及教学参考用书，也可作为科学与工程计算领域研究、开发参考用书。

**图书在版编目(CIP)数据**

Intel Visual Fortran 应用程序开发 / 周振红等编著。
郑州：黄河水利出版社，2006.9
ISBN： 7-80734-129-7

Ⅰ.1... Ⅱ.周...Ⅲ.FORTRAN 语言-程序设计-高等学校-教材 Ⅳ. TP312

中国版本图书馆 CIP 数据核字(2006)第 105517 号

组稿编辑：岳德军 0371-66022217

出版社：黄河水利出版社
地址：河北省郑州市金水路 11 号 邮政编码：450003
发行单位：黄河水利出版社
发行部电话：0371-66026940 传真：0371-66022620
E-mail：hhslcbs@126.com
承印单位：黄河水利委员会印刷厂
开本：787 mm × 10921 mm 1/16
印张：22.75
字数：480 千字 印数：1--2000
版次：2006 年 9 月第一版 印次：2006 年 9 月第一次印刷

书号：ISBN 7-80734-129-7 / TP・26 定价：38.00 元

# 前　言

Microsoft.NET Framework 是微软 2002 年开始推出的新一代计算平台，它简化了在高度分布式 Internet 环境中的应用程序开发。.NET Framework 以公共语言运行库和.NET Framework 类库为基础平台，它为 Visual Basic.NET、Visual C++.NET 和 Visual C#.NET (统称 Visual Studio.NET)提供了统一的开发环境，遗憾的是该环境不包含针对科学与工程计算的 Visual Fortran.NET 开发工具。

尽管 Fortran 语言标准发展很快，但支持它的编译器通常要滞后一到两年的时间。2004 年 5 月，在 ISO、IEC 的联合工作组 JTC1/SC22/WG5 以及美国 Fortran 委员会 NCITS/J3 的共同努力下发布了 Fortran2003 标准；直到 2005 年 6 月，Intel 公司才正式发布 Intel Visual Fortran 2003 的部分功能给予支持，且其中的 IA-32 编译器能以插件方式集成到 Visual Studio.NET 开发环境、共享 Visual C++.NET 中的工具和运行库。

此前，Fortran 90/95 的 Win32 开发环境多采用 Compaq Visual Fortran 6.x,但在 2005 年 CVF 开发团队加盟到 Intel 公司，HP 宣布其 CVF6.6 截至 2005 年 12 月 31 日、IVF9.0 将作为其新一代后继编译器。此编译器将 CVF 前端与英特尔处理器后端相结合，拥有 CVF 丰富的语言功能和英特尔处理器的代码生成及优化功能，使运行在英特尔平台上的程序能得到大幅度提高。所以，从 CVF6.x（Visual C++ 6.0）转移到 IVF9.0（Visual C++.NET 2002/2003）已是势在必行。

从针对 Fortran90/95 语言标准支持方面看，IVF9.0 时 CVF6.6 的升级版本，但在许多方面做出了改进，且所有的系统库都重新加以实现；若从开发环境方面看，两者却差异很大，比如，Visual Studio.NET 解决方案资源管理器中的一个工程只能由一种语言的源程序组成，而 Visual Fortran 6.6 （Visual C++ 6.0 ）项目空间中的一个工程可由 Fortran 和 C++/C 两种语言的源程序组成；况且，Intel Fortran9.0 的联机文档大部分借用了 Visual Fortran 6.6 的，当中有些地方并不合适，缺乏完整的开发实例，致使其推广应用受到一定程度的限制。有鉴于此，我们在对 Intel Fortran 9.0 全面认识、深入探究的基础上编撰了此书，书中内容涵盖了 Visual Studio.NET 环境下各种 Fortran 应用程序的开发，甚至包含 Intel Fortran 与 Visual C++/Basic.NET 的混合编程。

本书第 1 章、第 9 章、第 10 章由周振红(郑州大学)、王宗敏(郑州大学)共同编写，第 2 章至第 4 章由徐进军(武汉大学)编写，第 5 章、第 6 章由毕苏萍(郑州大学)编写，第 7 章、第 8 章由苗丽(郑州大学)编写，全书由周振红统编。

本书的出版得到了郑州大学水利工程一级学科(博士点)建设资金赞助。

感谢读者选择使用本书，欢迎您对本书内容提出批评和建议，我们不胜感激。

作者的联系方式为，电子邮件：zzh6374@yahoo.com.cn；通信地址：(450002)郑州大学环境与水利学院，周振红。

**作 者**

2006 年 6 月

# 导　读

第 1 章　IVF 编译器及 CVF 向 IVF 的转换。介绍 IVF9.0 的主要功能及特点、IVF9.0 软件包的安装，以及 CVF 向 IVF 的转换。

第 2 章　.NET 开发 Fortran 应用程序。介绍.NET 开发环境的设置、各种类型的 Intel Fortran 应用程序及其在.NET 环境下的创建。

第 3 章　QuickWin 绘图和输出文本。介绍常用的 Fortran QuickWin 应用程序的窗口操作、绘图选项的设置以及图形和文本(包括字符文本和字体文本)的输出。

本章示例程序 15 个。

第 4 章　QuickWin 界面定制。内容包括对 QuickWin 框架窗口的菜单、图标和标题进行定制，字符串及提示文字的本地化，以及子窗口客户区的鼠标输入。

本章示例程序 14 个。

第 5 章　使用对话框和控件。内容包括对话框程序的界面设计、代码编写，以及 Windows 常用控件的使用。

本章示例程序 3 个。

第 6 章　设计 Windows 应用程序。简要介绍 Windows 应用程序的结构，在此基础上讲述无模式对话框、单文档(SDI)和多文档(MDI)应用程序的设计，并进一步探讨 SDI 程序中使用无模式对话框、设计菜单和绘制图形的问题。

本章示例程序 3 个。

第 7 章　使用 COM 组件。先引入相关的 COM 组件技术，包括组件对象模型(COM)、自动化和 ActiveX 控件技术，然后讲解如何利用 Intel Fortran 模块向导来操作自动化服务器和 ActiveX 控件。

本章示例程序 3 个。

第 8 章　创建多线程应用程序。先给出进程、线程及其优先级的基本概念，然后讲述线程的创建、终止等基本操作，使用临界段、互斥体、信号计数器和事件对象的线程同步以及线程的局部存储，多进程应用程序的创建，并给出多线程操作的相关例程。

本章示例程序 2 个。

第 9 章　创建和使用动态链接库。先介绍动态链接库(DLL)的基本概念及其组成等，然后重点讲解如何在调用程序和 DLL 被调用(两者位于同一进程)间共享数据、例程及对话框资源，如何跨进程共享 DLL 中的 Fortran 共用区数据。在 DLL 的使用上，则给出了隐式和显式两种链接方式。

本章示例程序 7 个。

第 10 章　Intel Fortran 与 Visual C++/Basic.NET 的混合编程。先简要介绍混合编程的优越性、可行性、调用约定的基本概念及.NET 环境下的混合编程实现方式，然后重点讲解如何在 Intel Fortran 与 Visual C++.NET 间协调它们所使用的调用约定、命名约定，如何传递数据，如何匹配各种数据类型；针对 Intel Fortran 与 Visual Basic.NET 间的混合变成，则重点讲解如何传递 Intel Fortran 中的字符串、数组及派生类型三种特殊数据。最后，指出了基于静态库运行 Intel Fortran/Visual C++.NET 编译和链接需要注意的特殊方面。

本章示例程序 21 个。

# 第 1 章　IVF 编译器及 CVF 向 IVF 的转换

2005 年 6 月 14 日，英特尔公司正式发布了 Intel Visual Fortran(IVF)Complier 9.0 for Windows，以作为 Compaq Visual Fortran (CVF)6.6 的后继编译器。此编译器将 CVF 前端与英特尔处理器后端相结合，拥有 CVF 丰富的语言功能和英特尔处理器的代码生成及优化功能。此编译器包含标准版与专业版，专业版包含 Visual Numerics 公司的 IMSL Fortran 函数库 5.0。

**本章主要内容：**

- IVF 编译器的优化功能
- IVF 9.0 的安装
- CVF 向 IVF 的转换

## 1.1 IVF 编译器 9.0 Windows 版

使用 Intel Visual Fortran (IVF) 9.0 编译器(Windows 版)，可以使开发的应用程序在现有的英特尔处理器与体系结构、多核心英特尔处理器及最新英特尔处理器上获得前所未有的执行性能。

### 1.1.1　标准版与专业版

IVF 编译器 Windows 版有“标准版”与“专业版”两个版本。“标准版”包含英特尔编译器、英特尔数组可视化器、英特尔调试器以及其他一些功能。“专业版”拥有“标准版”的所有功能，此外，还包含 Visual Numerics 的“IMSL Fortran 函数库 5.0”。

“IMSL Fortran 函数库 5.0”将世界知名的 IMSL F90 函数库、并行处理功能及 IMSL Fortran 77 函数库集成到一个紧密相关的软件包中。它包含强大而灵活的新接口模块，可以使用所有高级 Fortran 语法和可选参数，同时仍提供真正的向后兼容性。这些接口模块可降低发生错误的可能性，简化并加速编码工作，同时还可提高代码质量。同时，还可以促进开发更简洁的 Fortran 应用程序，为熟练的程序员提供深入到地层并取得完全控制的能力。“IMSL Fortran 函数库 5.0”包含许多新的、有价值的数值算法，并扩展了对 SMP 并行处理的支持，在此基础上，可以开发出复杂的数值分析应用程序。

### 1.1.2 优化及其他功能

IVF 编译器开发的应用程序，能以较高的速度运行在英特尔 IA-32 处理器、英特尔 EM64T 处理器及安腾 2 处理器上。主要优化功能有：支持 Pentium4 和 PentiumM 的第三代数据流单指令多数据扩展指令集、安腾 2 的软件管道、过程间优化(IPO)、档案导引优化(PGO)等，通过自动并行和 OpenMP 技术支持多线程开发。具体包括以下几方面。

#### 1.1.2.1 经过优化的浮点指令吞吐能力

在 IA-32 系统上，IVF 编译器使用堆栈来高效执行浮点(FP)指令。由于重叠指令的计算结果可放入任何堆栈寄存器，应用程序性能因此得以提高。安腾 2 处理器提供可直接寻址的浮点寄存器(支持经管道化处理的浮点循环)，与采用传统体系结构的处理器相比，它需要的加载与存储操作次数大大减少。这样，运行应用程序时的执行速度就会快很多。

#### 1.1.2.2 过程间优化(IPO)

对于包含许多常用中、小函数的程序，特别是循环内包含调用的程序，IPO 可以极大地提高应用程序性能。

#### 1.1.2.3 档案导引优化(PGO)

PGO 编译过程可以使 IVF 编译器更好地使用指令调度与高速缓存，并可以更好地执行分支预测。它通过重新组织代码布局、缩短代码长度并减少分支预测失误来减少指令缓存反覆，从而提高应用程序性能。

#### 1.1.2.4 仅限 IA-32 体系结构

（1）支持第三代数据流单指令多数据扩展指令集。32 位功能支持“第三代数据流单指令多数据扩展指令集”，使得奔腾 4 处理器引入的“英特尔 NetBurst”微体系结构与众不同。此编译器也支持继续支持的性能改善功能，如 MMX 技术。“第三代数据流单指令多数据扩展指令集”超越了改善应用程序多媒体或图形组件性能这一初衷，它包含了更强劲的性能，可以满足进行浮点与双精度计算的要求。这些新指令可以通过许多途径来支持，其中包含内嵌(ASM)、编译器内建函数、类库、矢量器以及英特尔性能库。

(2)自动矢量器。提供可以自动对代码进行并行化处理的矢量器，以最大限度利用处理器的潜在能力。此矢量器包含齐全的文档，其中的示例显示了如何提高应用程序的执行速度。矢量器还提供多项功能，包括支持先进的动态数据调整策略，其中包含可以生成平衡负载的循环剥离技术，以及可以匹配整个缓存线预取情况的循环展开技术。

(3)对各代英特尔处理器的 Run-Time 支持——处理器调度。32 位功能提供了一个称为处理器调度的选项，可用于构建对特定的某代英特尔处理器的应用程序。调度选项现在可以针对多个特定目标，可以开发针对最新英特尔处理器——奔腾 4 处理器的应用程序，同时还可以确保可执行文件能在当前的 IA-32 体系结构上正常运行。在同一个可执行文件中，就可以使开发的应用程序既能利用最新处理器的性能优势，同时也可以在基于以前英特尔体系结构的系统上保持很高的性能。

#### 1.1.2.5 仅限英特尔安腾 2 微体系结构

(1)断定。先前的体系结构通过分支指令来实现条件执行，针对安腾 2 处理器的 IVF 编译器通过断定指令来实现条件执行。通过采用断定这项重要的优化措施，可以从程序序列中完全排除分支，从而形成更大的基本指令块，并消除相关的预测失误所造成的损失，这两项都有助于改善应用程序性能。由于使用断定之后存在的分支更少，控制六发生改变的概率也更小，因此动态指令获取的效率会更高。

(2)改进的分支预测。利用分支预测功能，处理器可以确定分支指令后面最可能的代码路径。处理器根据断定获取并开始执行最可能的代码路径上的指令。沿错误路径执行的指令的结果必须丢弃，并且必须获取并执行正确路径中的指令，所以分支预测失误会导致执行延迟。在安腾 2 微体系结构上，IVF 编译器与处理器可以互通分支信息，借此减少分支预测失误。它还可以使编译的代码能够利用 Run-Time 信息管理处理器硬件。这两项功能对断定起着补充作用，提供了多项性能优势：分支预测失误更少的应用程序运行速度更快，由仍可能

出现的分支预测失误引起的性能开销得到降低，，应用程序的缓存失误次数也更少。为确保应用程序在所有情况下均能正确运行，需要时会执行代码恢复过程。

(3)推测。开发人员可以利用推测技术在实际需要之前执行一些操作(如消耗很高的加载指令)来提高性能。 为确保代码正确，编译器会根据需要执行代码恢复过程，以确保在先前的推测发生失误时仍能正确执行所有受影响的操作。

(4)软件通道。通过支持软件通道，减少处理循环所需的时钟周期数。它会试图将每个循环分解成几个迭代阶段，每个阶段包含几条指令，并将这些迭代阶段搭接起来。软件管道技术允许同时存在多个循环迭代，并且不展开整个循环，所以在基于安腾 2 的系统上运行的应用程序中，它可以执行单周期、整循环计算。尽管并非所有循环都能得益于软件管道技术，但对于能够获益的那些循环而言，通过结合软件管道与断定、推测技术，可以显著减少代码扩展、路径长度及分支预测失误，从而实现应用程序性能上的飞跃。

#### 1.1.2.6 支持英特尔扩展内存 64 位技术(EM64T)

包括使用英特尔 EM64T 开发高性能应用程序时所需的各项功能。

#### 1.1.2.7 多线程应用程序支持

(1)OpenMP 支持。OpenMP 是可移植多线程应用程序开发领域的行业标准，在细粒度(循环级别)与粗粒度(函数级别)线程技术上具有很高的效率。对于江串行应用程序转换成并行应用程序，OpenMP 指令是一种容易使用且作用强大的手段，它具有使用用程序因为在多核心与对称多处理器上并行执行而获得大幅性能提升的潜力。IVF 编译器支持 OpenMP Fortran 2.0 版 API 规则（Workshare OpenMP 指令除外），并且能够为共享内存并行编程执行代码转换。利用 OpenMP Fortran 2.0 版，IVF 编译器可以支持多线程应用程序的开发与调试。

(2)自动并行。包含自动将循环线程化的自动并行功能，提供一个简单的办法以充分利用并行机制的优势，来提高应用程序在多处理系统上的性能。他会检测能够安全地并行执行的循环，并自动为这些循环生成多线程代码。开发人员不需要处理迭代划分、数据共享、线程调度及同步等低级别的细节。优化功能的改善包括将并行循环矢量化的能力，以及新增的“基于软件的推测性提前计算”功能，此功能可以改善应用程序在含“超线程技术”的英特尔处理器上的性能。

#### 1.1.2.8 自动接口检查

调用例程时，对于难以诊断的应用程序错误，同一来源导致的错误的参数列表并不一致。Fortran 90 引用了“显式接口”，使得编译器能够检查正确的调用，但是过去已经写好的和以后将编写的许多程序都没有提供这些额外的信息。IVF 编译器 9.0 可以自动检查参数列表的一致性，甚至可以扩越多个源文件来进行折项检查。选择此选项时，编译器会创建一个模块文件，包含每个子例程与函数的接口，然后在看到对子例程或函数的调用时，就会在生成的模块中查找。如果参数的数量、数据类型或数组形状不匹配，则显示一则错误信息。

#### 1.1.2.9 未初始化变量检查

现在，IVF 编译器可以对未初始化的变量执行运行时检查，帮助用户消除编程错误。应用程序检测到变量未经设置就进行使用时，会在调试器中显示一则指明该变量的详细信息，并停止执行。

#### 1.1.2.10 英特尔调试器

对优化代码的调试(即队在特定硬件体系结构上为取得最优执行效果而大幅改动的代码

进行调试)，IVF9.0 编译器可产生多项标准的调试信息，供所有支持 IVF9.0 编译器的调试器使用。“英特尔调试器 9.0”能够进行多线程应用程序的调试，为多核心体系结构提供了有力支持。英特尔调试器提供以下相关功能：

(1)全部停止/全部执行模型，即一个线程停止时所有的线程都停止，一个线程恢复时所有的线程都恢复。

(2)列出所有创建的线程。

(3)在线程之间切换焦点。

(4)检查详细线程状态。

(5)设置断点(包括所有的停止、跟踪及观察方式)，并显示所有线程或一部分线程的堆栈回溯。

(6)内置的 GUI 提供“线程”面板(在“当前源代码”窗格中)，创建线程时会激活次面板，供操作员选择线程焦点，并显示相关的详细信息。

Microsoft.NET 调试器也可用于基于 IA-32 处理器的并行应用程序。

#### 1.1.2.11 符合多项标准

IVF9.0 编译器符合 ANSI C/C++ 与 ISO C/C++ 标准，支持所有 Fortran 标准：包括美国国家标准 Fortran66(ANSI X3.0-1996)、Fortran77(ANSI X3.9-1978)、Fortran90(ANSI X3.198-1992)及 Fortran95(ANSI X3J3/96-007),国际标准 ISO 1539-1980(E)、ISO/IEC 1539:1997(E),以及 Fortran2003 的部分语言特征(如提供命令行与环境变量查询内部函数)。并提供了许多 Fortran95 语言扩展。

#### 1.1.2.12 其他功能

包括数据预取、编译器代码覆盖工具、编译器测试优先级调整工具等。

### 1.1.3 兼容性

#### 1.1.3.1 与 CVF 保持兼容

(1)Compaq Visual Fortran(CVF)的前端与 IVF 编译器的后端：结合了 CVF 丰富的语言功能于 IVF 编译器后端的代码生成及优化功能。

(2)与 CVF5.0 到 6.6 版在源代码保持兼容。用户只需重新编译应用程序,即可提高性能。为便于一致，此编译器提供了 250 多条 CVF 与 IVF 通用的命令与同义词。

(3)扩展的 Fortran95 编译器。

(4)Cray 指针。

(5)VAX 结构。

(6)“Fortran 90 模块” 的可靠性。

(7)在 Microsoft Visual Studio.NET 开发环境中的文本编辑器中,用彩色显示 Fortran 源代码(仅限 IA-32 处理器)。

(8)与“Microsoft.NET 调试器”集成(仅限 IA-32 处理器)。

(9)可实现与 C 例程轻松实现相互调用的特性。

(10)包含 QuickWin 函数库，便于轻松开发 Windows 应用程序(仅限 IA-32 处理器)。

(11)优化器报告生成功能。

(12)“英特尔数组可视化器”可用于按一维、二维或三维方式显示和以图形方式显示数组数据。用户可以旋转视图，也可以按一定的网格查看数据值，这样可以更容易找出多维数组中隐藏的模式。支持广泛的对象模型接口 ActiveX 控件及函数库例程。

(13)用于 Windows API 接口的*.mod 文件(仅限 IA-32 处理器)。

(14)集成到Microsoft Visual Studio.NET 2002 与 2003 开发环境(仅限 IA-32 处理器)。

#### 1.1.3.2 与针对 IA-32 的 Visual Studio.NET 开发环境集成

(1)IVF9.0 可继承到 Microsoft Visual C++.NET 2002 或 2003 开发环境。可以选择多种 Fortran 项目类型，包括控制台(Console)应用程序、DLL、QuickWin 应用程序及 Win32 应用程序。使用特定于 Fortran 的代码编辑与调试器，在一个环境中就可以进行混合语言应用程序的构建与调试。

(2)使用项目转换向导升级现有的 CVF 应用程序，在保护现有软件开发投资的同时，改善应用程序性能。

#### 1.1.3.3 处理 big-endian 文件

应用程序文件可以按 big-endian(高位在先，低位在后)模式进行读、写。

## 1.2 IVF 9.0 的安装

IVF9.0 标准版，包含针对不同的英特尔处理器的 IVF 编译器、英特尔调试器、英特尔数组可视化器以及 IVF 编译器集成到 Microsoft Visual C++.NET 2002 或 2003 的插件；IVF 9.0 专业版，包括标准版的所有组件，并提供了 Visual Numerics 的 IMSL Fortran 函数库 5.0。

### 1.2.1 系统要求

IVF 编译器针对不同的英特尔处理器进行应用程序的开发和部署(运行)，其组合情况如表 1-1 所示，

表 1-1 开发和部署(运行)的系统组合

| 开发和部署(运行) | IA-32 | Intel EM64T | Intel Itanium |
|---|---|---|---|
| IA-32 | Yes | Yes | Yes |
| Intel EM64T | No | Yes | No |
| Intel Itanium | No | No | Yes |

其中，IA-32 系统包含英特尔奔腾系列(诸如奔腾Ⅱ、奔腾Ⅲ、奔腾 4、奔腾 D)和 Xeon 处理器；EM64T 包含英特尔 EM64T、AMD Athlon 和 AMD Opteron 处理器；Itanium 系统则包含安腾 2 处理器。

表 1-1 显示：在 IA-32 系统上开发的应用程序，能够运行在任何英特尔处理器上；而在英特尔 EM64T 和 Itanium 系统上开发的应用程序，只能运行在同类型的英特尔处理器上。

#### 1.2.1.1 开发

1)硬件

（1）IA-32 系统要求不少于 256MB 内存，推荐使用 512MB 内存。

（2）其他系统要求不少于 512MB 内存，推荐使用 1GB 内存。

（3）要求不少于 600MB 的硬盘空间。IVF9.0 专业版若安装 IMSL Fortran 函数库 5.0，则需要额外的 250MB 硬盘空间。

（4）虚拟内存页面文件则要求 100MB 的硬盘空间，至少不低于操作系统要求的虚拟内存空间。

2)软件

**A.IA-32**

(1) Windows 2000、Windows XP 或 Windows Server 2003。

(2) 需要安装下列开发工具之一：①Visual C++.NET 2002 或 2003 ； ② Visual Studio.NET 2002 2003(含 Visual C++.NET)。

(3) 针对 x64 Edition 的 WindowsXP Professional 或 Windows Server 2003，只能安装 Visual C++.NET 2003 或 Visual Studio.NET 2003 (含 Visual C++.NET 2003)。

**B.Intel EM64T**

(1)Windows 2000、Windows XP 或 Windows Server 2003。

(2)针对 Windows Server 2003，需要安装 Microsoft Platform SDK。

**C.Intel Itanium**

(1)Windows 2000、Windows XP 或 Windows Server 2003。

(2)Microsoft Platform SDK。

**1.2.1.2 部署(运行)**

(1) IA-32: Windows 98、Windows ME、Windows NT、Windows 2000、WindowsXP 或 Windows Server 2003。

(2) Intel EMT64T：Windows Server 2003 x64 Edition 或 Windows XP Professional x64 Edition。

(3) Intel Itanium: Windows Advanced Server 或 Windows Server 2003(Enterprise 或 Datacenter Editions)。

### 1.2.2 安装支持软件

在安装 IVF9.0 之前，先要安装相关的支持软件：

(1) 针对 IA-32 目标系统，安装 Visual C++.NET 2002 或 2003，或者包含它的 Visual Studio.NET 2002 或 2003。

(2) 针对 Intel EM64T 或 Itanium 目标系统，安装 Microsoft Platform SDK。

(3) 移除 IVF8.x 与 Visual C++.NET 2002 或 2003 集成的插件。

IVF9.0 软件包并不包含 Visual C++.NET 2002、2003 或 Microsoft Platform SDF。

### 1.2.3 安装 IVF9.0

执行 IVF9.0 安装包的 Setup,启动英特尔软件安装助手，如图 1-1 所示。以下图形可能会与读者的有差异，因为录入人使用的版本是 IVF9.1

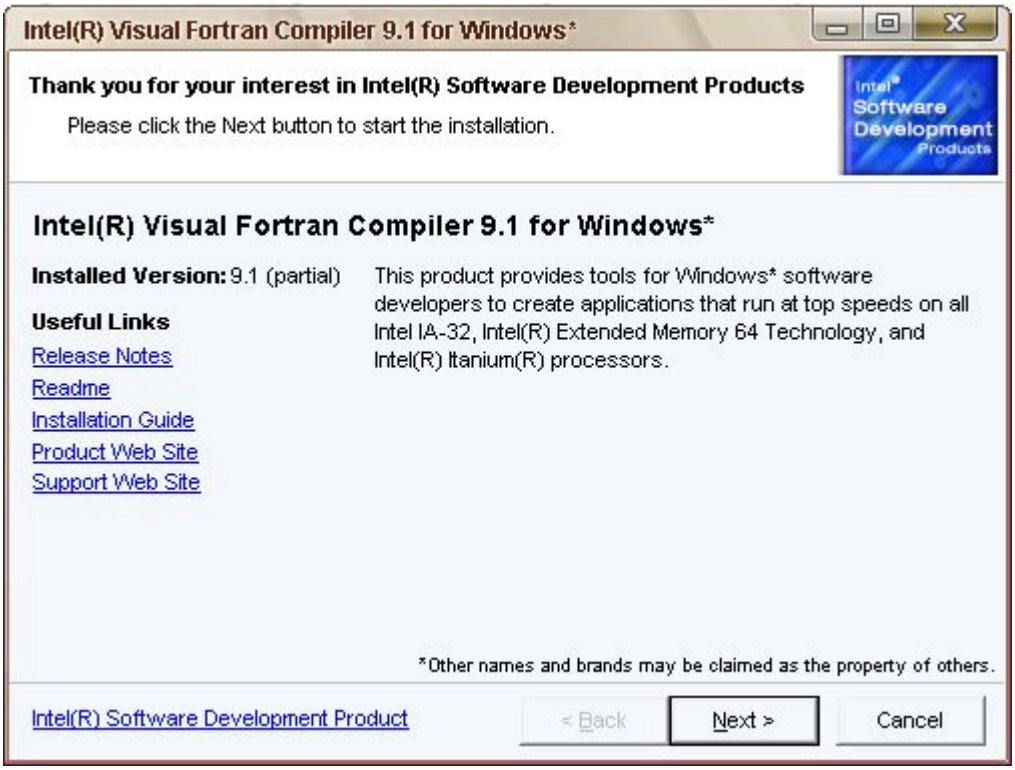



图 1-1 英特尔软件安装助手

在图 1-2 中，可以选择输入序列号或许可文件。需要使用许可文件的话，请点击“<<More Options”按钮。

图 1-3 为 IVF9.0(9.0.024)安装主界面，从主界面分别进入 IVF9.0 编译器、调试器、IVF 编译器集成到 Microsoft Visual C++.NET 2002 或 2003 的插件以及数组可视化器的安装界面(图 1-4～图 1-7)。(IVF9.1 则只需要在此页面勾选，安装助手会自动顺序安装)IVF9.0(9.0.024)编译器分以下三种，分别针对不同的目标系统。

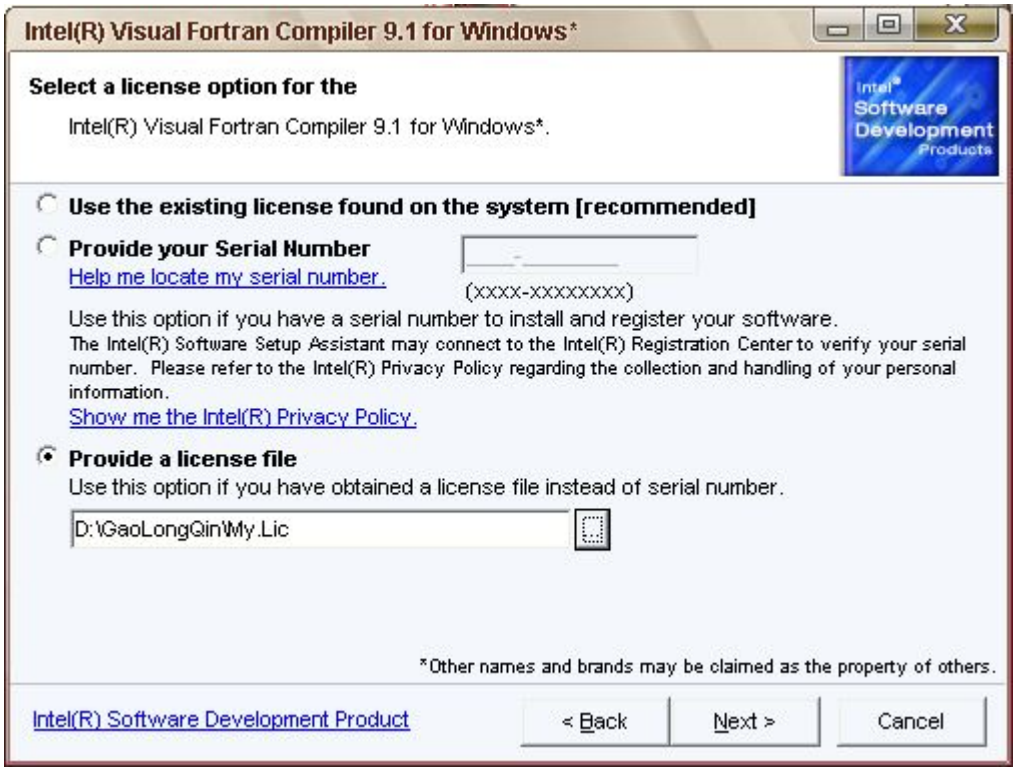


图 1-2 序列号或许可文件输入界面

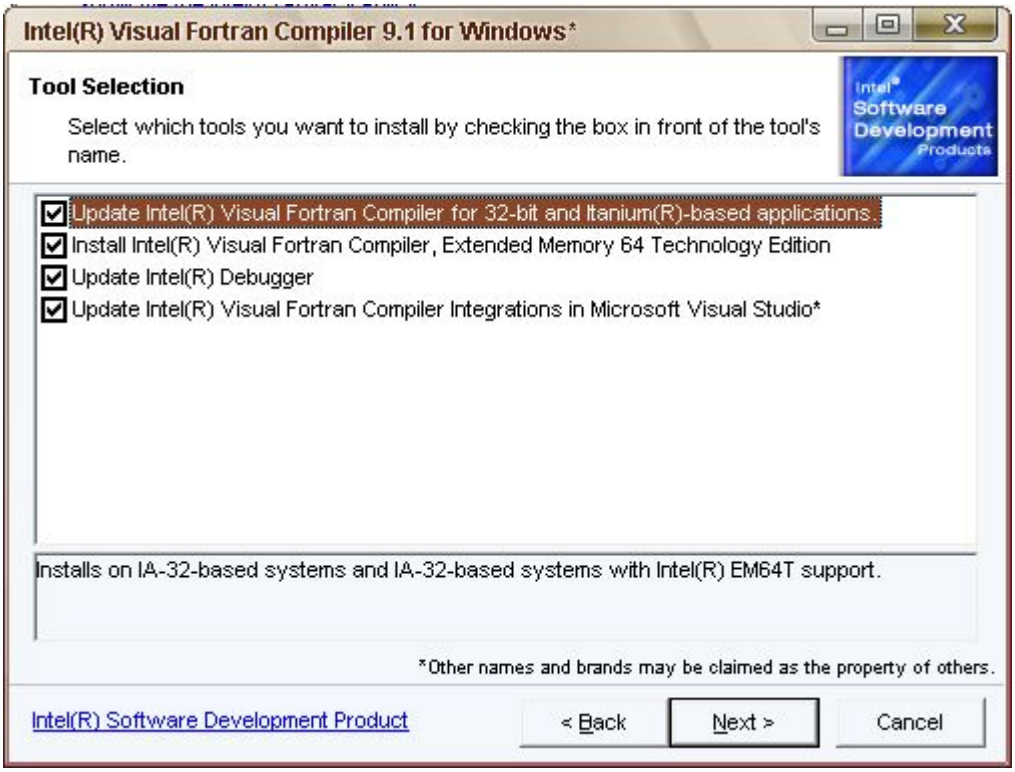



图 1-3 IVF9.0 安装主界面[IVF9.1 在此勾选后会自动顺序安装]

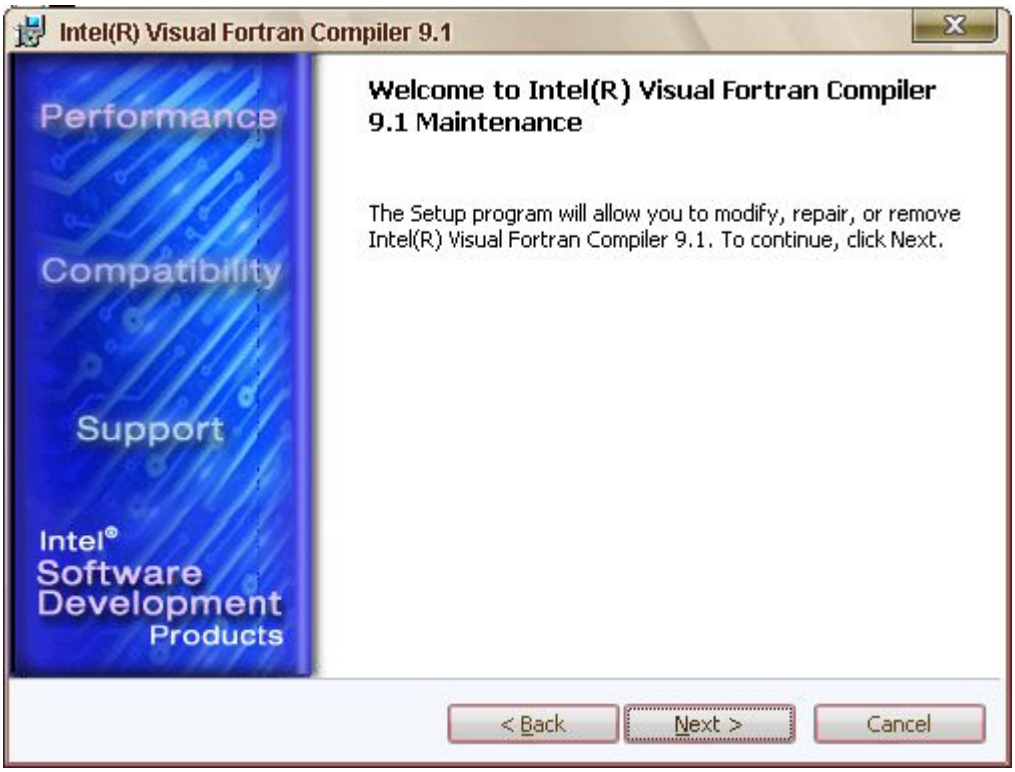



图 1-4 英特尔编译器安装界面




图 1-5 英特尔调试器安装界面

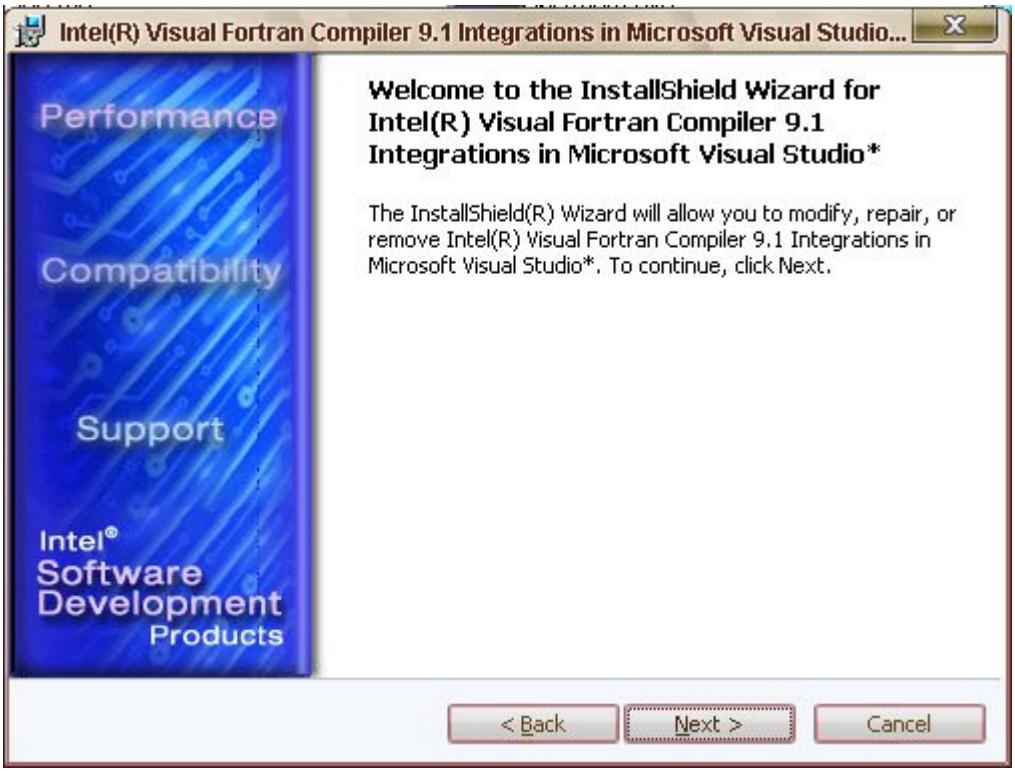



图 1-6 与.NET 集成插件安装界面

(1)IA-32 编译器。该编译器可安装在 IA-32 或支持 EM64T 的 IA-32 系统上，开发的应用程序则运行在 IA-32 系统上。

(2)EM64T 编译器。该编译器可安装在 EM64T 或支持 EM64T 的 IA-32 系统上，开发的应用程序则运行在 EM64T 系统上。

(3)Itanium 编译器。该编译器只安装在 Itanium 系统上，开发的应用程序也只则运行在 Itanium 系统上。

值得注意的是，与 Visual C++.NET 2002 或 2003 集成的插件只适用于 IA-32 编译器，且要先安装编译器再安装插件。专业版包含 IMSL Fortran 函数库 5.0，须在编译器安装后单独进行安装。

### 1.2.4 示例开发环境

本书示例采用的开发环境为：

(1) Pentium4,CPU2.80GHz,内存 512MB。

(2) Windows XP Professional 2002 (Service Pack 2)。

(3) Visual Studio.NET 2003 (含 Visual C++.NET 2003)。

(4) IVF 9.0(9.0.024)专业版，包括针对 IA-32 的编译器、调试器、数组可视化器、编译器集成到 Visual C++.NET 2003 的插件以及 IMSL Fortran 函数库 5.0。

(5) Compaq Visual Fortran 6.0(Visual C++ 6.0)。

录入人使用 IVF9.1，Visual Studio.NET 2005 部分截图可能会与原书不同

## 1.3 CVF 转换到 IVF

CVF6.x 是一个独立的开发环境,若系统同时安装了 Visual C++ 6.0,则两者的可视化开发环境自动集成为一体,且允许单一工程同时包含 Fortran 和 C/C++两种源文件;但 IVF 9.0 的 IA-32 编译器的可视化开发环境实际上是由 Visual C++.NET 提供的。该开发环境为 IA-32 编译器提供了必要的库、工具及可视化开发环境，且单一工程只能包含 Fortran 或 C/C++一种语言。

由 CVF 转换到 IVF，所有的 Fortran 源文件都需要在 IVF 下重新编译。IVF 编译器不能直接使用 CVF 编译的对象、模块和静态链接库；但只要不在两个环境下共享输入/输出单元，IVF 编译的应用程序可以使用 CVF 建造的动态链接库。

大多数情况下无须改变 Fortran 源文件，只在 IVF 下重新建造现存的 CVF 工程即可;但有些工程需要改变小量源代码，有些则需要对建造方法做出适当调整。

### 1.3.1 IVF 与 CVF 的兼容性

IVF 编译器支持所有的 CVF 语法，包括 Digital Equipment Corporation(DEC) Fortran 和 Microsoft Fortran PowerStation 4.0 的语言扩展；支持所有的 CVF 库例程，包括可移植库，QuickWin 以及所有的系统接口模块。

IVF 不支持的 CVF 特征有：

(1)Compaq Extended Math Library(CXML)。可以使用 Intel Math Kernel Library 或 IMSL、NAG 等第三方数学库来替换。

(2)跨引用的源代码浏览器。

(3)COM 服务器向导。假如要转换由 CVF 的 COM 服务器向导创建的组件工程，需要在 IVF

下重新构建；假如要调整组件接口，比如增加新的接口，则需要在 CVF 下进行。

### 1.3.2 移植 CVF 工程

IVF 编译器提供有工程转换向导，以方便 CVF 向 IVF 的转换。转换分两步进行：

(1)打开 CVF 项目空间。可以直接在 CVF 项目空间(*.DSW)上点击鼠标右键，选择“打开”；或者在 Microsoft Visual Studio.NET 2003 开发环境中，执行菜单“文件/打开/项目...”，打开 CVF 项目空间文件。在随后出现的消息框中(图 1-8)，点击“全是”按钮，将 CVF 项目空间下的每一项工程转换为 IVF 解决方案下对应的 Visual C++.NET 工程。




图 1-8 初始化的工程转换对话框

(2)提取 Fortran 工程项目。在上一步执行完毕，会展示 Microsoft Visual Studio.NET 开发环境，待转换的 CVF 工程作为 Microsoft Visual C++ 工程出现在解决方案资源管理器中，如图 1-9 所示。在每一个工程名上点击右键，从弹出的上下文菜单中选择执行“Extract Compaq Visual Fortran Project Items”，至此工程才算转换完毕，如图 1-10 所示。




图 1-9 提取 Fortran 工程项目上下文菜单




图 1-10 转换后的 Fortran 工程

由于 Microsoft Visual Studio.NET 的一个工程只能使用一种语言，所以，若 CVF 工程中包含 Fortran 和 C/C++两种源文件，必须将 CVF 混合语言工程转换成两个单一语言工程：一个为 Visual C++ 工程；另一个为 Intel Fortran 工程。而且，IVF 向导讲其中的一个工程改造成主工程。具体由哪一种语言建造主工程，由开发人员来决定，如图 1-11 所示。




图 1-11 转换 CVF 混合语言工程对话框




图 1-12 转换后的混合语言工程

## 1.3.3 IVF 引入的变化

由于 IVF 编译器和 CVF 编译器在实现上及开发环境方面的不同，两者必然存在一些差异，但 IVF 编译器在许多方面做出了改进。

### 1.3.3.1 语法检查更为严格

IVF 编译器的 Fortran 源代码编译功能直接继承自 CVF，但进行了一些改进，其语法检

查更为严格，并提供了较为详细的诊断信息。例如，在 Fortran 系统例程内调用非系统例程，IVF9.0 编译器将其诊断为错误用法，开发人员不得不纠正编码错误，才能在 IVF9.0 编译器上编译通过。

#### 1.3.3.2 缺省的调用约定发生改变

CVF 编译器的缺省调用约定为 STDCALL，编译产生的目标例程名统一被转换为大写，并增加了后缀@n,这里的“n”代表例程参数列表所占字节数；IVF9.0 编译器调用约定采用了其早期编译器使用的 C 调用约定，目标例程名仍统一被转换为大写，但去掉了后缀@n，并增加了一个前导下划线。

缺省调用约定的改变使字符串参数长度传递方式也发生改变。CVF 编译器紧接字符串参数地址传递其长度；IVF 编译器则在参数列表尾部传递所有的字符串长度，换句话说，编译选项已由/iface:mixed_str_len_arg 改为：/iface:nomixed_str_len_arg。假如应用程序只涉及 Fortran 一种语言，这一变化对编程不产生影响；但若应用程序涉及两种及以上的混合语言，就需要考虑这一变化带来的影响，可以在外部例程属性页规定 CVF 缺省调用约定，或者命令行使用/iface:CVF 选项。

#### 1.3.3.3 增加了新的系统例程

IVF9.0 编译器将命令行查询例程 GetArg，IArgc 和 NArgs 规定为系统例程。这样一来，开发人员不能用 External 声明它们，或建立其显式例程接口；否则，将产生链接错误。

#### 1.3.3.4 系统库被重新命名

CVF 采用模块对 Win32 API 例程及 Fortran 库例程进行了封装，如 DFWin、DFLib 等。为了保持与 CVF 的兼容性，IVF 中仍可以使用这些模块名；但这些模块并不是真正的 CVF 同名模块，而是 IVF 新模块的包装子(模块)。例如，下列为 IVF 中的 DFLib 包装子：

```
Module DFLib
  Use IFQWin
  Use IFCore
  Use IFPort,only:SignalQQ,RaiseQQ,LCWRQQ,SCWRQQ,SSWRQQ,SetControlFPQQ,&
     ......
End Module DFLib
```

这样，开发人员无须改变 CVF 程序中引用的模块，但在新的开发中应使用 IVF 新的系统模块，表 1-2 列出了 CVF 与 IVF 分别使用的系统模块。

表 1-2 显示：IVF 将 CVF 中的 DFLib 分解成 IFCore，IFPort 和 IFQWin 三个模块，为使产生的执行文件尽可能小,应选择使用具体的模块;在 CVF 的 DFWin 和 IVF 的 IFWin 模块中，包含 Win32 API 例程的具体模块名相同，如 Kernel32、User32、GDI32。

表 1-2 CVF 和 IVF 所使用的系统模块

| CVF | IVF | 描述 |
|---|---|---|
| DFAuto | IFAuto | 自动化 |
| DFCom | IFCom | COM，OLE |
| DFComity | IFComTY | 过时的，用 IFWinTY 替换 |
| DFLib | IFCore，IFPort，IFQWin | QuickWin、可移植库及常用库 |
| DFLogM | IFLogM | 对话框 |

| | | |
|---|---|---|
| DFLogMT | IFLogMT | 对话框数据类型和常量 |
| DFMT | IFMT | 多线程例程 |
| DFNLS | IFNLS | 国际语言支持 |
| DFOpnGL | IFOpnGL | OpenGL 例程 |
| DFOpnGLT | IFOpnGLT | OpenGL 数据类型和常量 |
| DFPort | IFPort | 可移植例程库 |
| DFWBase | IFWBase | Win16 例程(不推荐使用) |
| DFWin(包含 Kernel32、User32、GDI32 等) | IFWin(包含 Kernel32、User32、GDI32 等) | Win32 API 例程 |
| DFWinA | IFWinA | 与 QuickWin 例程冲突的 Win32 例程 |
| DFWinTY | IFWinTY | Win32 API 数据类型和常量 |

#### 1.3.3.5 编译命令发生改变

在 CVF 中，可以使用的编译命令有 DF、F90、F77、FL32 四个：DF 和 F90 是等价的，F77 与 Compaq Fortran 77 保持兼容，FL32 则与 Microsoft Fortran PowerStation 保持兼容。在 IVF 中，编译命令改为 IFort，F77 和 FL32 被淘汰，DF 和 F90 可以使用但会给出警告信息，使用/quiet 编译选项可以将警告信息压缩掉。

IVF 和 CVF 可以安装在同一操作系统上。当使用命令行时，要确保使用的是正确的命令行提示：CVF 为“Fortran Command Prompt.”，IVf 为“Fortran Build Environment fot IA-32 Applications.”,它们建立与编译器相匹配的环境。而它们的可视化继承开发环境(IDE)是独立的，相互之间没有影响。

#### 1.3.3.6 使用 IMSL Fortran 函数库

IVF 专业版包含 Visual Numerics 的 IMSL Fortran 函数库 5.0。除了提供许多新的例程和对已有例程更新外，IMSL Fortran 函数库 5.0 还提供了一些 CVF 专业版所不包含的特征：

(1) 用于改善在英特尔多核处理器、多处理器和支持超线程技术处理器上的运行性能所提供的多线程库。

(2) DLL 形式的数学库。

(3) 支持英特尔安腾处理器和 EM64T 处理器。

(4) 利用 Fortran 95 重载接口和可选参数的统一接口模块。

假如不准备利用这些新特征，现存使用 IMSL 库的源程序几乎不需要改变。

CVF 只提供了 1 套单线程的 IMSL 静态链接库；而 IVF 专业版则提供了 4 套以不同名称命名的 IMSL 库：

(1) Link_F90_Static,在单处理器上使用的 IMSL 静态库。

(2) Link_F90_DLL,在单处理器上使用的 IMSL 动态库。

(3) Link_F90_Static_SMP,在多处理器上使用的 IMSL 静态库。

(4) Link_F90_DLL_SMP,在多处理器上使用的 IMSL 动态库。

# 小　结

■ IVF 9.0 编译器将 CVF 前端与针对英特尔处理器系列的英特尔编译器后端相结合，拥有 CVF 丰富的语言功能与 IVF 编译器的代码生成及优化功能。此编译器分标准版与专业版，专业版包含 Visual Numerics 公司的 IMSL Fortran 函数库 5.0。

■ IVF 编译器开发的应用程序，能以较高的速度运行在英特尔 IA-32 处理器、英特尔 EM64T 处理器和安腾 2 处理器上。

■ IVF 编译器的主要优化功能包括支持 Pentium4 和 PentiumM 的第三代数据流单指令多数据扩展指令集、安腾 2 的软件通道、过程间优化(IPO)、档案导引优化(PGO)及通过自动并行和 OpenMP 技术支持多线程开发等。

■ IVF9.0(9.0.024)提供了英特尔 IA-32、EM64T、和安腾 3 种编译器，分别针对相应的目标系统：英特尔 IA-32、EM64T、和安腾 2。

■ 在安装 IVF9.0 编译器前，需要先安装 Microsoft 相应的开发工具：IA-32 编译器，安装 Visual C++.NET 2002 或 2003；EM64T 和安腾编译器，则安装 Platform SDK。

■ 通过安装与 Visual C++.NET 2002 或 2003 集成的插件，IA-32 编译器可以利用 Visual Studio.NET IDE 来高效开发 Fortran 应用程序。

■ 利用 IVF9.0 提供的转换向导，可以方便地将 CVF 工程转换为 IVF 工程。大部分 CVF 工程只需在 IVF 下重新建造即可，但 CVF 混合语言工程需要分解为两个由单一语言构成的主工程和静态库工程。

■ IVF9.0 没有提供 COM 服务器向导，在 CVF 下由 COM 服务器向导创建的组件工程，需要在 IVF 下重新编译；要改变组件接口，只能在 CVF 下进行。

■ IVF9.0 引入了许多变化，在很多方面对 CVF 进行了改进，并用新的名称命名系统模块。IVF9.0 专业版提供了 4 套针对英特尔单处理器和多处理器的 IMSL 静态与动态库。

# 第 2 章　.NET 开发 Fortran 应用程序

一旦系统安装了 Visual Studio.NET(含 Visual C++.NET)、针对英特尔 IA-32 处理器的 Intel Fortran 9.0 编译器及其集成到.NET IDE 中的插件，就可在 Visual Studio.NET 可视化集成开发环境下高效开发 Intel Fortran 应用程序。

**本章主要内容：**

- 设置.NET 开发环境
- Fortran 应用程序类型
- 创建 Fortran 工程

## 2.1 设置.NET 开发环境

在基于.NET IDE 创建不同类型的 Fortran 应用程序前，首先应对开发环境进行设置，包括规定编译器版本、创建工程时 IDE 搜索的系统目录以及自定义的 Fortran 文件扩展名。

### 2.1.1　编译器版本

从.NET IDE“工具”菜单中打开“选项”对话框，点击对话框左边的“Intel(R) Fortran”列表项，选取“General”子项，此时，对话框右边展示的各项如图 2-1 所示。

在“Selected Compiler Version”下拉框中，选取特定的编译器版本。目前，本书所用开发环境只安装了 Intel Fortran 9.0。(录入人使用 IVF9.1，某些插图会有细微的差别)

### 2.1.2 搜索目录

在图 2-1 所示的“选项”对话框中，从“Project Directories”栏目下规定建造工程时 IDE 要搜索的目录：

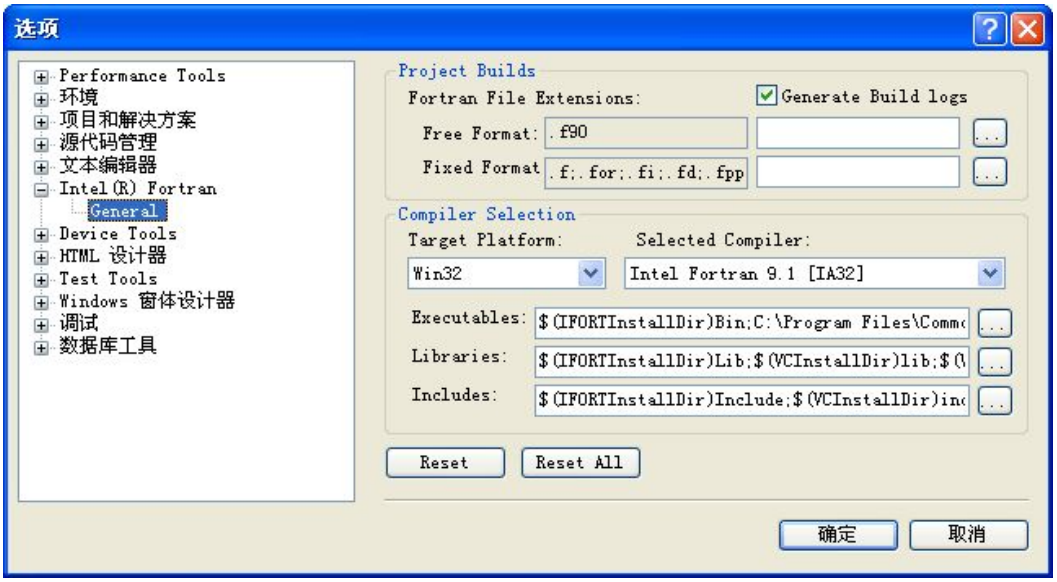



图 2-1 IDE“工具”菜单下的“选项”对话框

(1)Executables:搜索执行文件的系统目录；

(2)Libraries:搜索运行库的系统目录；

(3)Includes:搜索包含文件的系统目录。

需要注意的是：如果从命令行 devenv/useenv 命令打开.NET IDE，那么 IDE 使用系统规定的 Path、Include、Lib 环境变量，而不是通过图 2-1“选项”对话框设置的搜索目录。

点击图 2-1“选项”对话框中的“Restore Default Options”按钮，将恢复各项设置为系统默认设置。

事实上，在安装 Visual Studio.NET 和 Intel Fortran 编译器时，系统按安装的具体情况来规定开发环境的默认设置。所以，在实际开发 Fortran 应用程序时，一般不需要对开发环境重新进行设置。

### 2.1.3 自定义源文件扩展名

除了 Intel Fortran 编译器默认的 Fortran 自由格式和固定格式源文件扩展名外，还允许用户规定额外的Fortran源文件扩展名，IDE把额外的文件扩展名看成是可编译的Fortran源文件。

当规定一个新的文件扩展名时，IDE 检查注册表看是否有特定的语言、工具及文件格式与新的文件扩展名相关联。如果发现有，系统就会给出提示，并禁止规定该文件扩展名。

要自定义 Fortran 文件扩展名，可在图 2-1 对话框的“Fortran File Extensions”栏目下，以点号(“.”)开头规定额外的自由格式和固定格式文件扩展名，假如规定的扩展名不止一个，它们之间用分号(“;”)隔开。

新规定的文件扩展名，在下一次启动 Visual Studio.NET 时生效。

如果点击图 2-1 中的“Restore Default Options”按钮、Fortran 源文件扩展名也恢复为系统默认设置。

## 2.2 Fortran 应用程序类型

IDE 开发环境设置好后，就可以着手设计 Fortran 应用程序了。首先，新建一个 Fortran 工程(项目)，在 Visual Studio.NET IDE 下执行菜单命令“文件/新建/项目”，或从起始页对话框内点击“新建项目”按钮，弹出“新建项目”对话框，如图 2-2 所示。

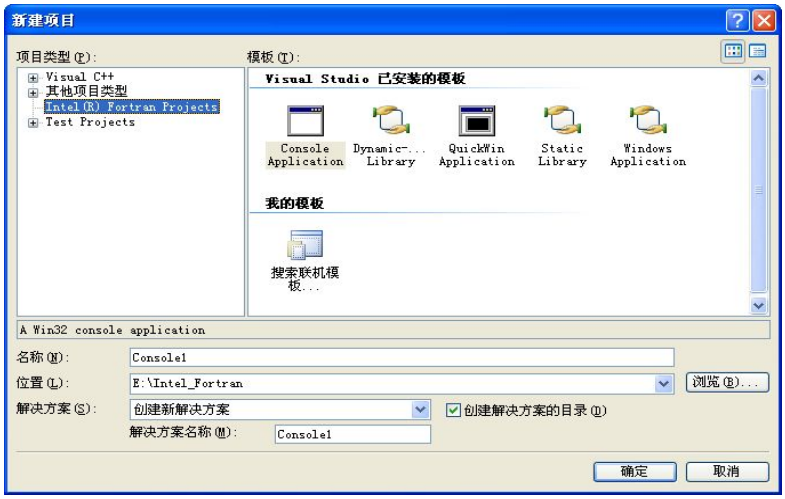



图 2-2 “新建项目对话框”

从中选取某种工程类型，启动相应的应用程序向导，完成初始的工程设置，如工程类型、应用程序特征等。应用程序向导自动为用户生成工程解决方案，针对某些工程类型，还生成应用程序骨架代码。表 2-1 列出了 Intel Visual Fortran 可以创建的工程类型，前 4 个为可以独立执行的应用程序，要求有主程序；后 2 个为库应用程序，不能包含主程序。

表 2-1 Intel Visual Fortran 可以创建的应用程序类型及其主要特征

| 应用程序类型 | 主要特征 |
|---|---|
| Console (.Exe) | 没有图形的(基于字符的)单窗口独立执行应用程序 |
| Standard Graphics (.Exe) | 有图形的单窗口独立执行应用程序 |
| Quick Win (.Exe) | 有图形的多窗口独立执行应用程序 |
| Windows (.Exe) | 有丰富的图形界面、可以调用所有 Windows API 的多窗口独立执行应用程序 |
| Static Library(.Lib) | 链接到 EXE 文件的例程库 |
| Dynamic-Link Library (.DLL) | 在程序执行期间动态链接的例程库 |

### 2.2.1 控制台应用程序

Fortran 控制台(Console)应用程序是基于字符的，它不要求从屏幕上输出图形。这种应用程序外观上类似于运行在 UNIX 工作站或主机终端上的程序，在单一的窗口中运行，用户通过标准的读、写命令与程序进行交互。

控制台应用程序更适合解决纯数学处理问题，而不适合于解决具有图形输出或图形用户界面的问题。与其他类型的应用程序相比，这种类型的应用程序更容易移植到别的平台上。因为不产生消耗资源的图形输出，所以 Fortran 控制台应用程序比标准图形或 QuickWin 图形应用程序运行得要快。

控制台应用程序如果调用了图形例程，将不会产生图形输出，而是返回错误代码。此刻，程序不会自动退出，因此要求用户编写异常处理代码对这种情况进行处理。控制台应用程序不能使用 QuickWin 函数，但可以使用单线程或多线程的静态链接库、动态链接库和对话框。

若从命令行创建 Fortran 控制台程序，由于控制台程序是缺省的工程类型，可不加入任何命令参数，直接输入 ifort 命令，系统自动添加链接选项/subsystem:console。

图 2-3 显示的即为一个控制台应用程序运行画面。

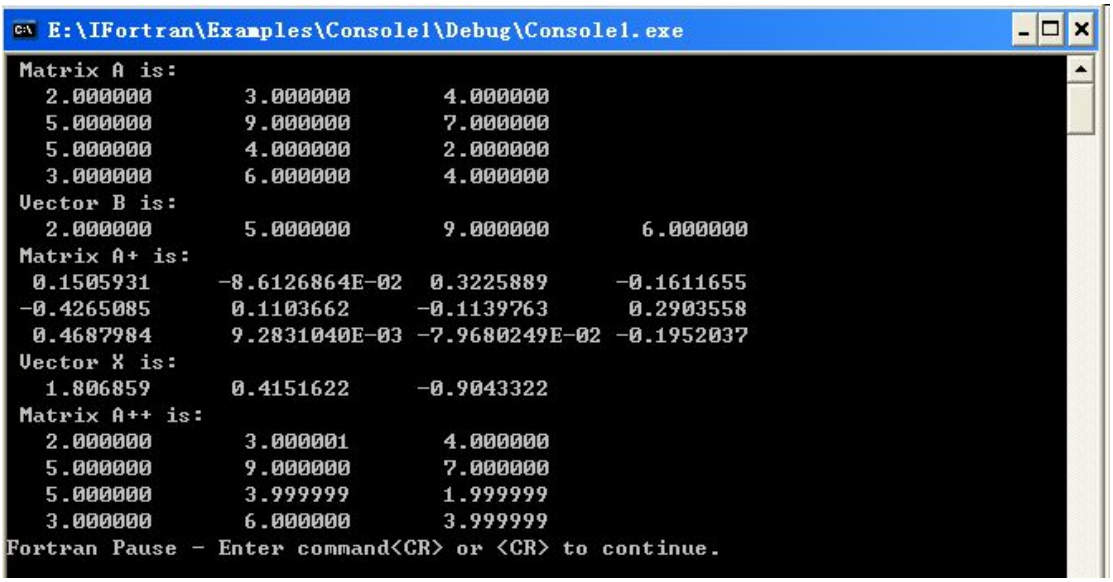



图 2-3 一个控制台应用程序运行画面

### 2.2.2 标准图形应用程序

要创建 Fortran 图标图形(Standard Graphics)应用程序，须先在“新建项目”对话框(图 2-2)中选取“QuickWin Application”工程类型，再在随后出现的应用设置对话框中选取“Empty Standard Graphics project(single window)”，如图 2-4 所示。即 Intel Visual Fortran 将标准图形应用程序归结为单窗口的 QuickWin 应用程序。

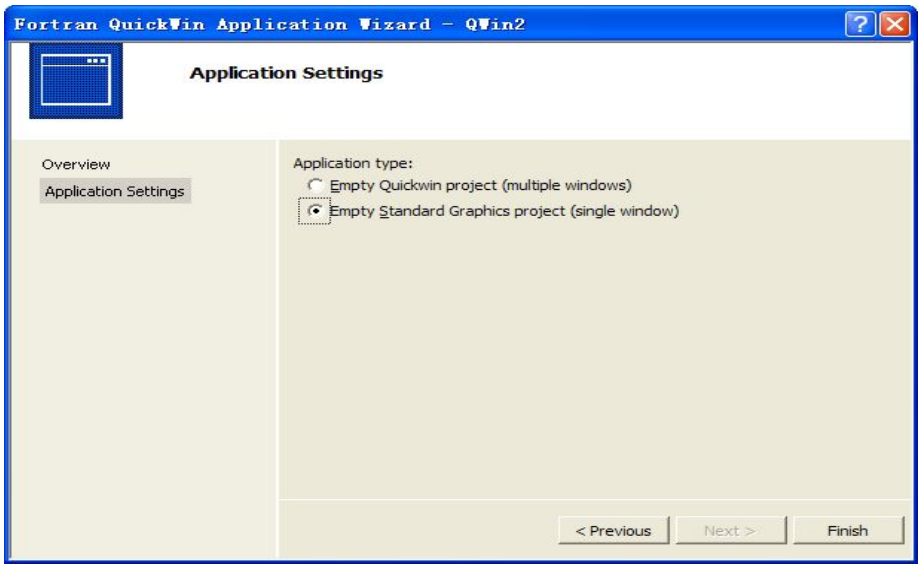



图 2-4 标准图形应用程序向导

Fortran 标准图形(QuickWin 单窗口，有时称为单文档)应用程序看起来与直接对图形硬件进行操作的 MS-DOS 程序类似，它运行产生图形，如画直线、绘制基本图形，允许书用屏幕函数，如清屏操作。标准图形是 QuickWin 的一个子集，Fortran 标准图形应用程序可以使用所有的 QuickWin 图形函数，也可以像其他工程类型那样使用对话框。

文本能以位图或文本模式显示，通过 Windows API 来加载和卸载位图文件。Fortran 标准图形程序，通常按多线程应用程序设计。

Fortran 标准图形应用程序一运行即全屏显示，可以通过 Alt+Enter 键在全屏和具有最小化、最大化及关闭按钮的窗口之间切换。如果选定的分辨率与屏幕大小匹，应用程序窗口会覆盖整个屏幕；否则，显示一个带滚动条、大小可调的窗口，该窗口既没有菜单栏也没有状态栏。在标准图形应用程序中，不能打开附加的窗口。

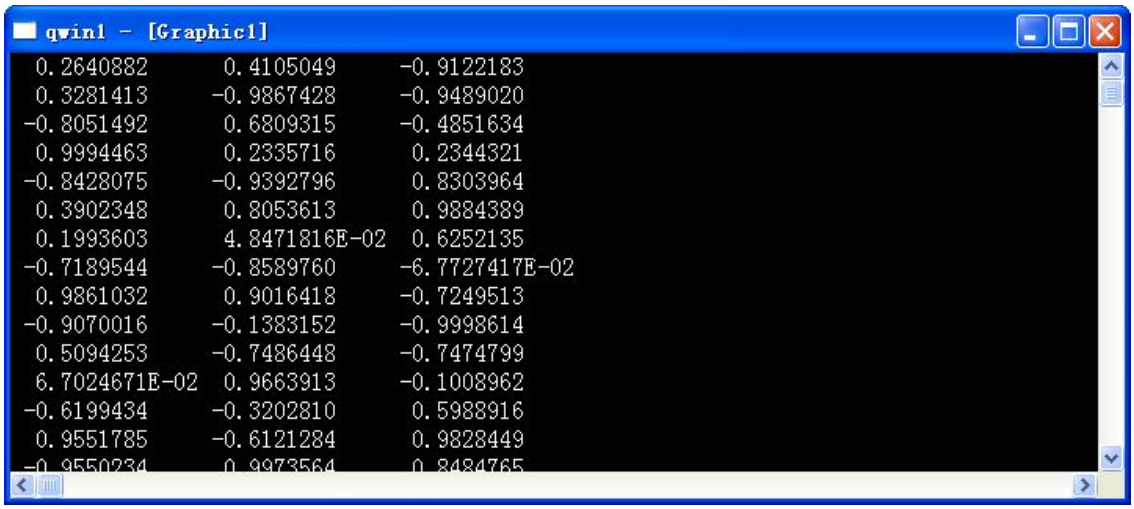



图 2-5 一个标准图形应用程序运行画面

### 2.2.3 QuickWin 应用程序

在 QuickWin 应用向导设置对话框中，默认的工程类型为多窗口的 QuickWin 工程。

Fortran QuickWin 图形应用程序是多线程的，它比标准图形应用程序更为通用，在其程序执行过程中可以打开多个窗口。故此，QuickWin 图形应用程序也被称为 MDI 多文档界面应用程序。多个窗口为程序运行提供了灵活性，如可以打开几个窗口(最多 40 个)，其中的一个窗口控制程序运行，其他的窗口输出图形，并可在不同的窗口间切换。这些窗口既能全屏显示，又能被缩小放在屏幕的不同位置上。

QuickWin 工程类型允许用户使用 Windows 接口的简化版本来设计应用程序，QuickWin 库提供了丰富的 Windows 特征集，但它不包含全部的 Windows 应用程序编程接口 API (Applications Programming Interface)。如果程序需要加入额外的 Windows 功能，必须创建 Windows 类型的应用程序，来直接调用 Windows API。

MDI 应用程序在框架窗口(也称为主窗口)的顶部有菜单栏，底部也状态栏，在菜单栏和状态栏之间的区域称为客户区，MDI 子窗口位于客户区内。QuickWin 库提供了缺省的菜单栏和菜单项，用户可以通过 QuickWin API 对其进行定制。

QuickWin 应用程序要使用对话框，须引用 IFLogM 模块，该模块封装了对话框操作函数，它们是 Windows API 的一个子集。程序通过对话框函数来展示、初始化窗口，与对话框进行通信。

当新建 Fortran QuickWin 应用程序时，集成开发环境自动包含 QuickWin 库，使程序可以调用图形函数。若用 IFort 命令从命令行建造 QuickWin 应用程序，需规定/libs:qwin 选项。程序运行时，如果选定的分辨率与屏幕大小匹配，应用程序窗口会覆盖整个屏幕;否则，显示一个带流动条、大小可调的窗口。此外，不能将 QuickWin 应用程序建造成动态链接库。

图 2-6 显示的为一个 QuickWin 应用程序运行画面。Fortran QuickWin 应用程序的开发详见第 3 章和第 4 章。

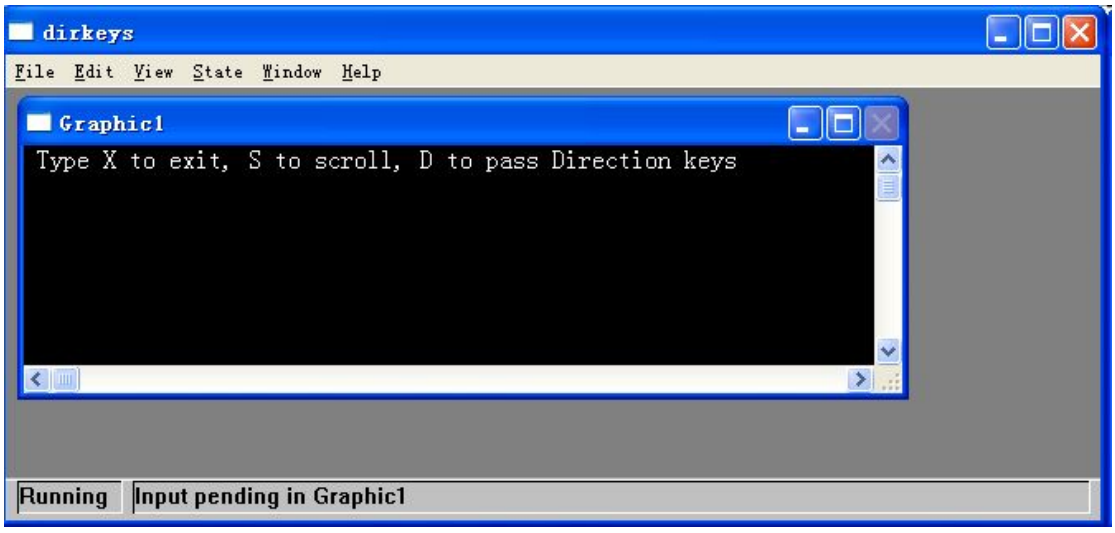



图 2-6 一个 QuickWin 应用程序运行画面

### 2.2.4 Windows 应用程序

Fortran Windows 应用程序能够以 Intel Visual Fortran 中直接调用 Windows API，与 QuickWin 相比，它可以访问更多的 Windows API。尽管其它类型的应用程序可以调用某

些 Windows API，但 Fortran Windows 应用程序可以调用全部的 API 例程。使用其它类型的应用程序无法使用的系统特征。

假如程序要使用系统提供的基础服务，需在程序中引用 IFWin 模块。该模块包含了大多数通用 Windows API 接口，包括窗口管理、图形设备接口、Windows 系统服务、多媒体及远程过程调用。

窗口管理例程提供了创建和管理用户界面的手段，可以用来创建窗口、展示程序输出、用户输入提示等。图形设备接口(GDI)函数用来将产生的图形输出到屏幕、打印机以及其他输出设备上。Windows 系统服务函数允许用户管理和监控内存资源，访问文件、目录和 I/O 设备，对错误、事件记录和异常处理等进行操作。

多媒体函数可以创建文档和演示文稿，将音频、视频和文本、图形合成为一体，它提供有诸如音频、视频、文件 I/O、媒体控制、游戏杆及计时器等服务。

远程过程调用(RPC)用来执行分布式计算，充分利用网络上的计算机资源。分布式应用程序中，调用程序作为一个进程运行在客户机的一个地址空间内，被调用的程序则运行在另一台计算机的一个地址空间内。一个分布式应用程序由客户程序和服务器程序组成，客户程序向用户展示数据；服务器程序则存储、提交、管理数据及执行计算任务。例如网络数据库、远程文件服务器等。

若用 IFort 命令从命令行创建 Fortran Windows 应用程序，需规定/winapp 选项。图 2-7 显示的为一个 Windows 应用程序运行画面。

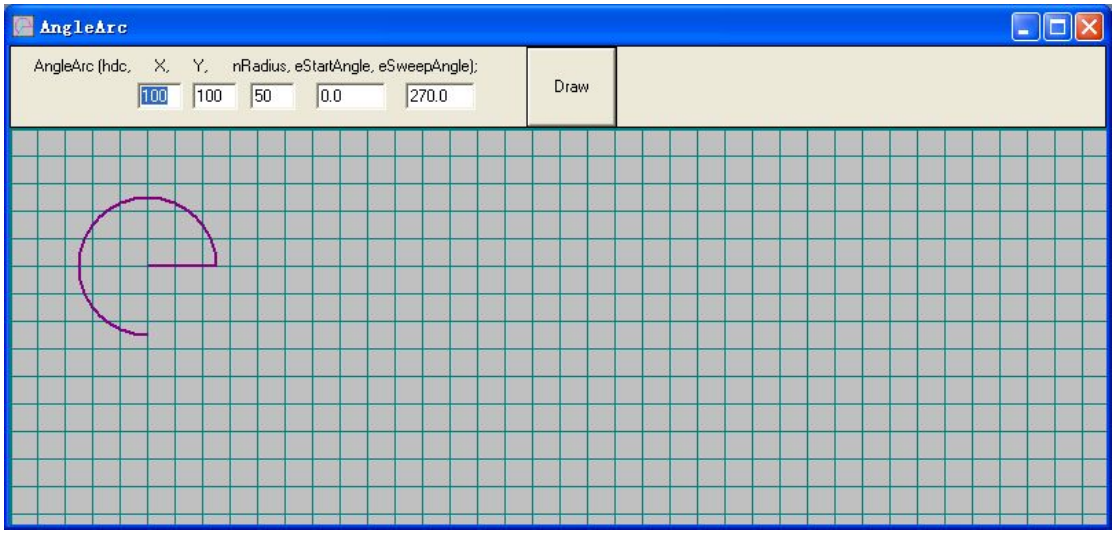



图 2-7 一个 Windows 应用程序运行画面

编写 Fortran Windows 应用程序要比其它类型的 Fortran 应用程序复杂，需要有 Windows 应用程序的相关基础。Fortran Windows 应用程序的开发详见第 6 章。

### 2.2.5 静态链接库

Fortran 静态库(*.Lib)是单独编译和保存的代码块，程序运行前直接链接到应用程序执行文件(*.Exe)中。从集成开发环境(IDE)创建 Fortran 静态库，直接选取“Static Library”工程类型；若从命令行创建，须使用/compile_only 选项来取消编译器链接，并使用 Lib 命令管理静态库。

当从 IDE 创建静态库时，先要规定是否阻止插入缺省库的链接指令。系统默认情况下，不插入缺省库的链接指令。该选项阻止静态库规定 Fortran 运行库的版本，和其他类型的

Fortran 工程链接时，后者的 Fortran 运行库同时也作为静态库的运行和使用。

假如与 C/C++进行混合编程，Fortran 静态库作为子程序，C/C++作为主程序，可以取消该选项，即插入缺省库的链接指令；否则，需要在链接器“Additional Dependencies”属性页中显式命名 Fortran 运行库。在创建静态库工程后，可以通过 Fortran 的“Disable Default Library Search Rules”属性改变该选项的设置。

静态库为组织大型程序和在几个程序间共享例程提供了方便，它只包含例程(子程序和函数的统称)，不包含主程序。静态库的文件扩展名为.Lib，由目标代码组成。当与主程序链接产生可执行文件(.Exe)时，所需的静态库例程被链接起来。静态库通常被保存在特定的目录下。

当使用静态库时，只有调用程序用到的库例程才被链接到 Exe 可执行文件中，没有用到的不会被链接，这可以产生较小的 Exe 可执行文件。用户不必考虑要包含哪些库例程，这由编译器决定。

因为使用静态库的所有程序都保持同一个库例程版本，这可以帮助识别当前的应用程序是不是最新版本。若修改了静态库中的例程，只需要重新进行链接，就能更新使用静态库的应用程序。

静态库是在解决方案浏览器窗口中定义的源代码和目标代码的集合，当建造静态库工程时，源代码被编译产生目标代码，目标代码则被收集到.Lib 文件中，而不进行链接。建造的静态库文件名与工程名相同。假如静态库的规模比较大，应将它存放在专门的目录下，使用静态库的应用程序在链接期间对它进行访问。

静态库不能单独进行调试，须和使用它的应用程序一起进行调试。在用 Debug 选项对静态库和主程序进行编译、链接后，再按常规的调试办法进行调试。

当在 IDE 中向主程序插入静态库时，可以在向导对话框中输入路径和带.Lib 扩展名的库文件名；若要使用 MakeFile 文件，须在主程序的 MakeFile 文件中添加静态库；若从命令行建造工程，需添加带.Lib 扩展名的库文件参数，必要时还要包含路径。

### 2.2.6 动态链接库

动态链接库(.DLL)同样之包含例程，而不存在主程序，当中的源代码被编译、链接成一个与主程序分离的执行单元，它与调用程序(外部程序)共享代码和数据地址空间，即两者运行在同一个进程中。

DLL 除了拥有静态库组织代码的优点外，还能使主程序产生更小的执行文件，不过接口稍复杂一些。DLL 的目标代码并不包含在 Exe 执行文件中，而是在应用程序运行需要时进行动态加载，且允许有多个应用程序同时访问一个 DLL。

组织通用例程到一个 DLL 中，在程序运行调用时再动态加载到内存。这种方式有效地节省了存储空间，且不增大调用它的应用程序执行文件的大小；此外，更新 DLL 中的例程不需要重新建造使用它的应用程序。

在 Intel Fortran 环境中，可以建造自己的 DLL，也可以使用储存在特定 DLL 中的运行库：

(1) 在 IDE 中建造 DLL，直接选取“Dynamic-Link Library”工程类型；从命令行建造 DLL，使用带/DLL 选项的 IFort 命令。

(2) 在 IDE 中使用储存在特定 DLL 中的运行库(该运行库没有添加到当前工程中)，可以在工程属性对话框中，从“Fortran”选项集中选择“Library”类别，在右边的“Runtime Library”栏中选取一个要链接的运行库，如图 2-8 所示；从命令行建造工程，需使用 /libs:dll 编译器选项。

建造和使用 DLL 详见第 9 章。

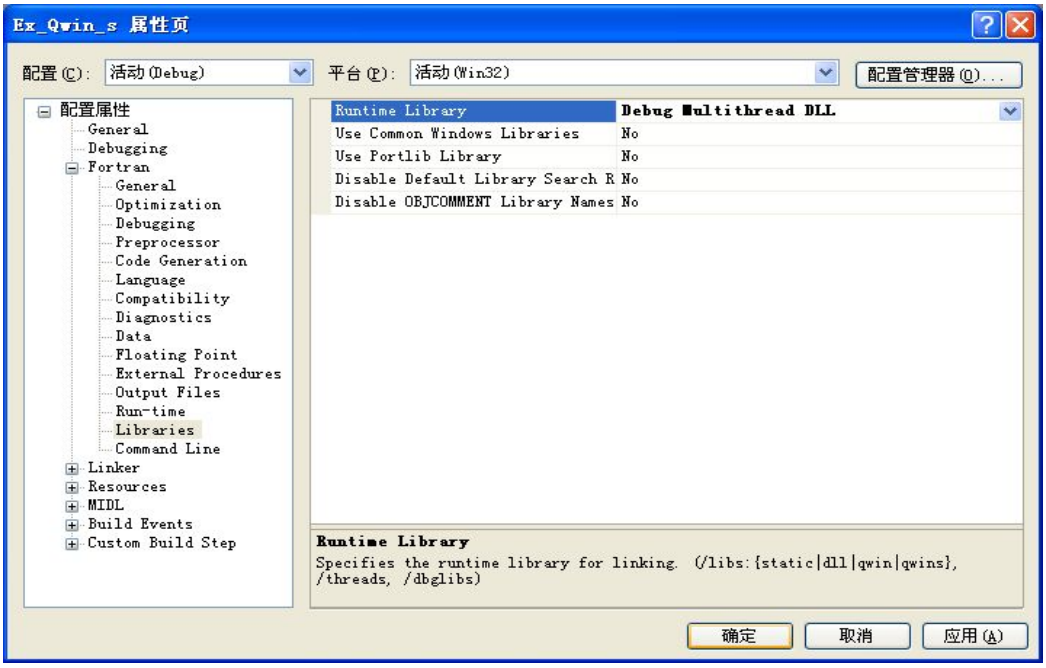



图 2-8 设置要使用的运行库

## 2.3 创建工程

应用程序向导完成初始的工程设置后，展示如图 2-9 所示的 Visual Studio.NET IDE 窗口。可见，应用程序向导自动生成了包含工程的解决方案，即在 Visual Studio.NET IDE 下，通过解决方案(Solution)对工程(Project)实施管理，一个解决方案可以包含多个工程。如果是独立执行的应用程序，多个工程中只能规定一个工程为包含主程序的主工程，库应用程序不包含主程序。和 Compaq Visual Fortran 不同的是，Visual Studio.NET 要求每个工程都只使用一种语言。

IDE 创建解决方案时产生的文件如表 2-2 所示。

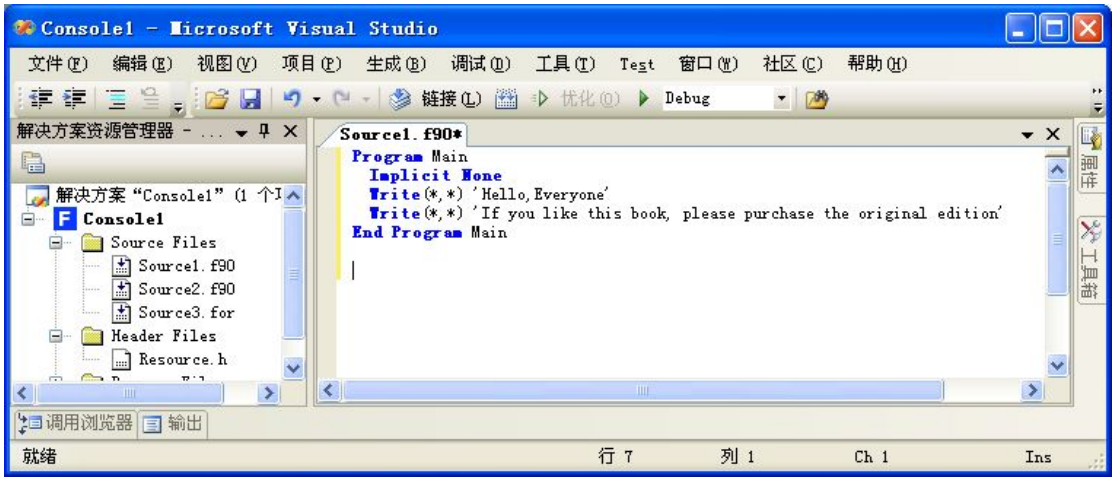



图 2-9 Visual Studio.NET IDE 窗口

表 2-2 IDE 创建解决方案时产生的文件

| 文件 | 扩展名 | 内容 |
|---|---|---|
| 解决方案文件 | .sln | 存储解决方案信息，包括解决方案中的工程、项目以及它们的位置信息 |
| 工程文件 | .vfproj,.vcproj,.icproj | 包含用来创建主程序、子程序的相关信息，.vfproj、.vcproj 及.icproj 分别为 Intel Visual Fortran、Visual C++.NET 及 Intel C++ 工程文件 |
| 解决方案选项 | .suo | 包含解决方案的 IDE 用户定制信息 |

### 2.3.1 添加文件

如果向导生成的解决方案中不包含任何工程，或需要向工程中添加其他的文件，可以从 IDE“项目”菜单或在解决方案资源管理器窗口点击鼠标右键弹出的上下文菜单中，选择“添加新项”或“添加现有项”对话框，从中选取要插入工程的文件。

从简化工程设置方面考虑，如果从现有的源文件建造新工程，最好是先利用特定应用程序向导生成空的工程，然后将现有的源文件复制到当前目录下，再往当前工程中添加现有文件。

如果程序中引用了 Fortran 系统模块，不需要显式地将其加入工程，编译时引用的系统模块自动以依赖文件(.mod)形式组织到工程中；如果模块式用户编写的工程中的一个文件，不需要进行单独编译，当编译工程时，Intel Visual Fortran 决定编译哪些文件。

### 2.3.2 选择配置

配置包含了定义生成工程二进制数出文件所规定的设置选项，如生成的应用类型、应用程序运行的平台、编译和链接时的工具设置等。

可以从“生成”菜单或“项目”菜单“属性”项打开的“属性页”对话框中，访问“配置管理器”对话框，如图 2-10 所示。

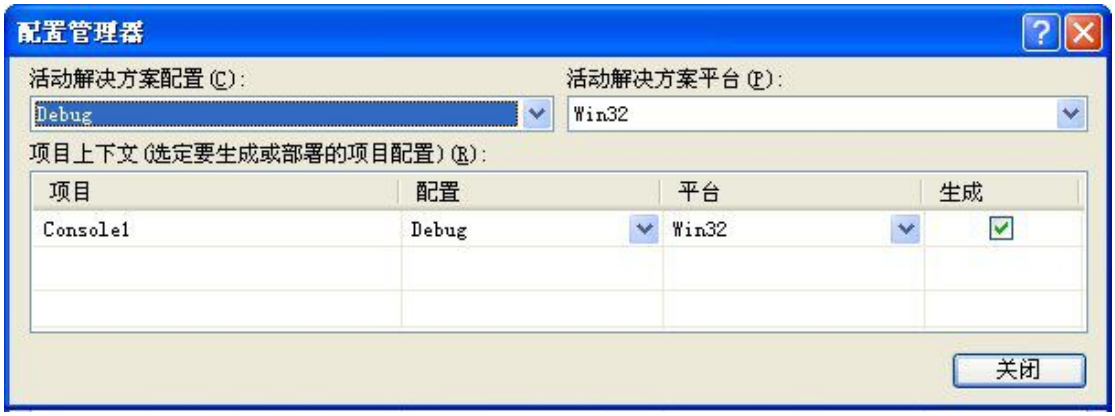



图 2-10 “配置管理器”对话框

当新建工程时，IDE 自动创建 Debug 和 Release 配置。Debug 配置(缺省)，设置工程选项以包含调试信息，并关闭优化选项；Release 配置，工程选项不包含调试信息，可以使用规定的任何优化选项。一次只能规定一个配置选项。

当编译、链接工程时，按规定的活动配置生成。若选择的是 Debug 配置，一个以 Debug 命名的子目录被创建，当中包含的是调试版的工程输出文件；若选择的是 Release 配置，一个以 Release 命名的子目录被创建，当中包含的是发布版的工程输出文件。

一个配置决定了要生成的 Fortran 应用程序类型，也规定了工程生成选项，包括编译器和链接器选项。

尽管两个配置通常使用的是同一工程文件集，但两者的工程设置一般是不同的。例如，默认的 Debug 配置提供了充分的调试信息，不包含优化选项；而默认的 Release 配置则提供了最少的调试信息，但可以包含充分的优化选项。

图 2-10 配置管理器中的平台类型，规定了工程的操作环境，并针对该平台类型设置选项，如源文件使用的编译器选项、链接器针对该平台使用的库、定义的常数等。Intel Visual Fortran 支持 Win32 平台。

对解决方案生成配置所做的任何更改，都会反映在“属性页”对话框的相应选项中。

### 2.3.3 设置工程

要设置当前工程的编译和相关选项，先从“项目”菜单或在工程名商点击鼠标右键展示的上下文菜单中，通过“属性”菜单项打开“属性页”对话框，或从“视图”菜单的“属性页”菜单项直接打开“属性页”对话框，如图 2-11 所示。

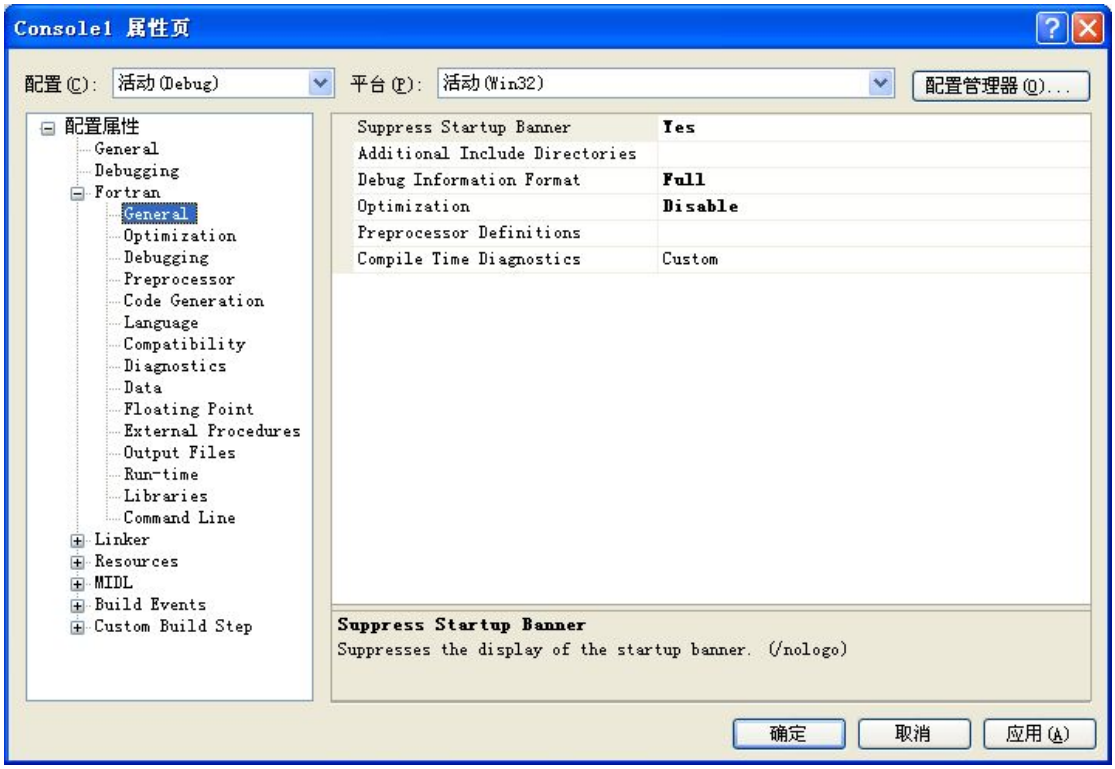



图 2-11 设置工程选项窗口

除了对当前工程进行设置外，Intel Fortran 还允许对当前工程中的单个 Fortran 源文件规定编译选项。方法是：先选取要进行设置的文件名，然后从“视图”菜单或在文件名上点击鼠标右键展示的上下文菜单中打开“属性页”对话框。

图 2-11 工程“属性页”对话框中的 Fortran 选项各子项的简要说明列于表 2-3，其它选项及其子项的详细说明请参考联机文档。

表 2-3 Fortran 选项中的各子项

| Fortran 选项 | 说明 |
|---|---|
| General | 工程常用的一般选项，当中的一些选项可在其他属性页详细规定 |
| Optimization | 程序优化选项，包括速度、特定处理器、代码大小等 |
| Preprocessor | 预编译选项，规定编译器如何预处理文件，在哪里寻找源文件目录 |
| Code Generation | 代码生成选项，规定如何生成可执行代码 |
| Language | 语言选项，规定语义、语法和源文件件格式 |
| Compatibility | 兼容性选项，规定源文件、数据文件与老版本的文件或其他操作系统的兼容性，如 endian 无格式数据文件、OpenVMS 系统 Run-Time、Microsoft Fortran PowerStation |
| Diagnostics | 诊断选项，规定警告和错误信息的种类 |
| Data | 数据选项，规定在编译、优化和生成代码时如何处理数据 |
| Floating Point | 浮点选项，规定如何处理浮点数据 |
| External Procedure | 外部例程选项，规定外部例程如何调用 |
| Output Files | 输出文件选项，规定编译产生的输出文件名称和路径 |
| Run-Time | Run-Time 选项，规定 Run-Time 进行错误检查，而不是在编译时检查 |
| Libraries | 库选项，规定应用程序运行依赖的库 |
| Command Line | 命令行选项，某些选项不能从 IDE 属性页规定，可在此处输入。 |

## 2.3.4 生成可执行文件

要生成应用程序的可执行文件，在 IDE 窗口中的“生成”菜单(图 2-12)上选择执行下列命令：

(1)只编译一个文件；

(2)生成工程或解决方案；

(3)重新生成工程或解决方案；

(4)批生成一个工程的几个配置；

(5)清理工程或解决方案(删除生成的所有文件)。




图 2-12 IDE 窗口的“生成菜单”

当执行生成工程命令时，IDE 检查工程中的所有文件，看哪些文件自上一次生成后发生

改变，然后自动更新相关的依赖文件，编译发生改变的文件，并链接所有的目标文件成一个可执行文件。

当执行重新生成工程命令时，编译器重新编译工程中的所有源文件，并链接生成可执行文件。

在一次操作中，可以选择生成单个工程、当前工程或多个工程配置(批生成)，也可以选择生成整个解决方案。

另外，可以对解决方案工程中的单个文件进行编译。方法是：先在解决方案资源管理器窗口选取工程中的文件，然后在文件名上点击鼠标右键展示的上下文菜单中，或从 IDE“生成”菜单中，执行编译命令。

要执行生成的应用程序，可以在 IDE“调试”菜单上点击“启动”菜单项(F5)，或点击“开始执行(不调试)”菜单项(Ctrl+F5)，也可从命令行执行应用程序。

### 2.3.5 宏指令

Intel Fortran 支持某些建造宏指令，在工程“属性页”对话框接收字符的地方，可以直接输入这些宏指令。宏指令名不区分字母大、小写。

表 2-4 列出了 Visual Studio.NET 和 Intel Fortran 共同支持的建造宏。

表 2-5 列出了只有 Intel Fortran 支持的建造宏。

表 2-4 Visual Studio.NET 和 Intel Fortran 共同支持的建造宏

| 指令名 | 宏指令 |
|---|---|
| 配置名 | $(ConfigurationName) |
| 平台名 | $(PlatformName) |
| 生成的中间文件目录 | $(IntDir) |
| 输出目录 | $(OutDir) |
| 输入目录 | $(InputDir) |
| 输入路径 | $(InputPath) |
| 输入名 | $(InputName) |
| 输入文件名 | $(InputFileName) |
| 输入文件扩展名 | $(InputExt) |
| 继承属性 | $(Inherit) |
| 不继承属性 | $(NoInherit) |
| 工程目录 | $(ProjectDir) |
| 工程路径 | $(ProjectPath) |
| 工程文件名 | $(ProjectFileName) |
| 工程文件扩展名 | $(ProjectExt) |
| 解决方案目录 | $(SolutionDir) |
| 解决方案路径 | $(SolutionPath) |
| 解决方案名 | $(SolutionName) |
| 解决方案文件名 | $(SolutionFileName) |
| 解决方案文件扩展名 | $(SolutionExt) |
| 目标目录 | $(TargetDir) |
| 目标路径 | $(TargetPath) |
| 目标名 | $(TargetName) |

| | |
|---|---|
| 目标文件名 | $(TargetFileName) |
| 目标文件扩展名 | $(TargetExt) |
| Visual Studio 安装目录 | $(VSInstallDir) |
| Visual C++ 安装目录 | $(VCInstallDir) |
| .NET Framework 目录 | $(FrameworkDir) |
| .NET Framework 版本 | $(FrameworkVersion) |
| .NET Framework SDK 目录 | $(FrameworkSDKDir) |

表 2-5 Intel Fortran 支持的建造宏

| 指令名 | 宏指令 |
|---|---|
| Intel Fortran IDE 安装目录 | $(IFIDEInstallDir) |
| Intel 数组可视化器安装目录 | $(IAvInstallDir) |

## 小　结

■针对英特尔 IA-32 处理器的 Intel Fortran 9.0 编译器，需要有 Visual C++.NET 的支持，并与其共享 Visual Studio.NET 集成开发环境。

■.NET 开发环境设置，包括规定编译器版本、建造工程时 IDE 搜索的系统目录以及自定义的 Fortran 文件扩展名。通常，开发环境可以采用系统默认设置。

■Intel Visual Fortran 9.0 可以创建的工程类型有 6 种：控制台、标准图形、QuickWin、Windows、静态链接库和动态链接库。前 4 种是可以独立执行的应用程序，要求有主程序；后 2 种是库应用程序，不能包含主程序。

■在.NET IDE 中，由解决方案资源管理器对工程实施统一管理。一个解决方案可以包含多个工程，多个工程中只能规定一个工程为包含主程序的主工程。.NET 要求每个工程只能使用一种语言。

■在.NET IDE 中开发 Intel Fortran 程序，通常先利用 Fortran 应用向导生成骨架工程，然后，向工程中添加文件，选择一个配置，设置工程选项，最后编译、链接并生成应用程序的可执行文件。

■Intel Fortran 支持某些建造宏指令，可以在“工程属性”对话框中直接输入这些宏指令。宏指令不区分字母大、小写。

# 第 3 章　QuickWin 绘图和输出文本

在 Intel Fortran 中，QuickWin 既可在后台运行计算程序，又可同时在前台(子窗口)输出图形和文本，因此，它是最常用的工程类型。其运行库 IFQWin 是 Windows API 的一个子集，它提供有缺省菜单栏、状态栏的框架窗口，允许打开多达 40 个子窗口。

QuickWin 程序分两章介绍，本章主要讲解在 QuickWin 程序中绘图和输出文本，第 4 章讲解 QuickWin 程序的界面定制。

**本章主要内容：**

- 子窗口相关操作
- 图形输出设置
- 输出图形和文本

## 3.1 操作子窗口

### 3.1.1　创建子窗口

当运行 QuickWin 应用程序时，若没有任何输入/输出语句，程序只展示框架窗口；若有屏幕输入/输出(*)，则额外展示一个以“Graphic1”命名的缺省子窗口，如图 3-1 所示。

在 QuickWin 应用程序中，通过使用带 File="User"参数的 Open 语句来创建子窗口，通过随后运行的 I/O 操作或图形输出来展示子窗口。缺省的子窗口是可滚动的文本窗口，大小为 30 行×80 列。QuickWin 允许打开 40 个子窗口。

创建子窗口的 Open 语句，必须以顺序访问方式(缺省打开文件方式)打开用"USER"命名的文本文件,其他的规定(如直接访问，二进制或无格式)将引发 Run-Time 错误。

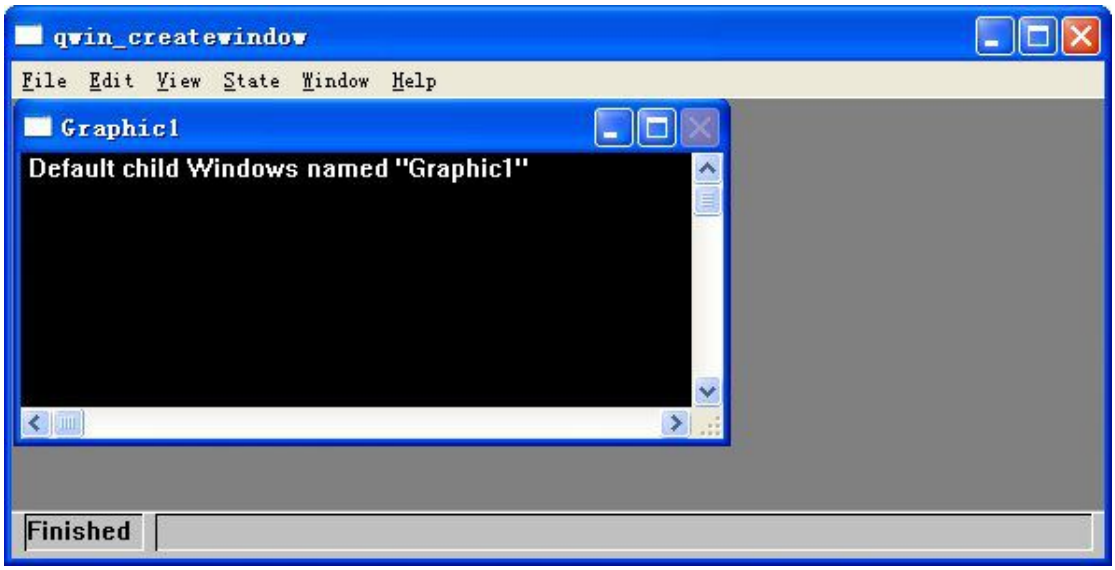



图 3-1 “Graphic1”缺省子窗口

QuickWin 程序的 I/O 操作，需要在特定的子窗口中进行。

【例 3-1】打开一个子窗口，规定其标题为“Product Matrix”，在子窗口内输出字符

串“Enter matrix type:”。

例 3-1 程序实现代码为：

```
Program Ex_1
  Implicit None
  Open(Unit=12,File='User',Title='Product Matrix')
  Write(12,*) 'Enter matrix type:'
End Program Ex_1
```

Write 语句中的 12,表示将输出定位到与单元号 12 相连的文本文件(这里是子窗口)。

单元号 0、5 和 6 有特殊含义，它们都指缺省的 Graphic1 子窗口，即进行屏幕输入/输出的子窗口。Read(*,*)和 Read(5,*)指从屏幕读取数据；Write(*,*)，Write(0,*)及 Write(6,*)指向屏幕写数据。在使用这些语句前，无须显式使用 Open 语句打开 Graphic1 子窗口。若在 Read 语句中使用单元号 0 和 6，将产生 Run-Time 错误。

值得注意的是：若只使用带 File="User"参数的 Open 语句来创建子窗口,而不进行针对该窗口的 I/O 操作或图形操作，该窗口将不可见。

### 3.1.2 焦点窗口和活动窗口

当使用 Open 语句打开子窗口时，该子窗口即被激活成为活动窗口。活动窗口可以接收图形输出，但未必被放置到前台，即未必拥有焦点。也就是说，要进行图形输出的子窗口必须是活动的，但可以没有焦点，即图形输出独立于焦点。

当电机鼠标、进行 I/O 操作或调用 FocusQQ 例程，相应的子窗口获取焦点，同时该子窗口也成为活动的。当一个窗口获得焦点，其他先前获得焦点的窗口将失去焦点，也就是说，某个时刻只能由一个窗口拥有焦点。拥有焦点的窗口位于顶层或前台，其标题栏高亮显示；没有焦点的窗口位于底层或后台，标题栏灰色显示。

大多数情况下，焦点和激活被施加于同一个窗口，这时 QuickWin 的默认行为。但有些 QuickWin 例程不接收单元号参数(如 GetCharQQ、PassDirKeysQQ 及 GetWindowConfig),其操作时针对活动窗口进行的，和该窗口是否拥有焦点无关。这可能导致出现异常情况，如当调用 GetCharQQ 例程获取用户输入字符时，若当前活动的窗口没有焦点，用户不得不先在该窗口上点击，然后才能输入字符。为了避免出现这种情况，在调用针对活动窗口操作的例程前，可以对活动窗口进行 I/O 操作或调用 FocusQQ 例程，使激活和焦点施加于同一个窗口。

当调用 OutText 或绘图函数(如 OutGText、LineTo、Ellipse)时，窗口焦点不发生转移；但当用鼠标点击一个窗口、对一个窗口进行 I/O 操作或调用 FocusQQ 函数时，该窗口获得焦点。通过调用 InqFocusQQ 函数，可以查询当前哪一个窗口拥有焦点。

【例 3-2】先后打开单元号 10、11 的两个子窗口，将当前焦点移至单元号 10 的子窗口，此时查询拥有焦点的子窗口。

例 3-2 程序实现代码为：

```
Program Ex_2
  Use IFQwin
  Implicit None
  Integer(4) status,focusUnit
  Open(Unit=10,File='User',Title='Child Window 1')
  Write(10,*) 'Giving focus to Child 1.'
  Open(Unit=11,File='User',Title='Child Window 2')
```

```
  !Give focus to Child Window 2 by writing to it:
  Write(11,*) 'Giving focus to Child 2.'
  !Give focus to Child Window 1 with the FocusQQ function:
  status=FocusQQ(10)
  !Find out the unit number of the child window that currently has focus:
  status=InqFocusQQ(focusUnit)
  Print *,focusUnit
End Program Ex_2
```

程序在缺省的 Graphic1 内输出 10。

在 IFWin 运行库中，SetAciveQQ 函数用来激活一个子窗口，GetActiveQQ 函数用来返回当前活动子窗口的单元号，GetThwndQQ 函数则用来获取特定单元号子窗口的句柄。

### 3.1.3 设置子窗口属性

上面提到 SetWindowConfig 例程是针对活动窗口进行操作的，SetWindowConfig 例程和 GetWindowConfig 例程分别用来设置和获取活动窗口属性，而 SetWSizeQQ 例程则用来设置活动可视窗口属性。若活动窗口大于可视窗口，窗口自动添加滚动条，以容许展示活动窗口中所有的文本和图形。

类似于 C++中的窗口类，活动窗口通过 WindowConfig 派生类型来定义。

```
Type windowConfig
  Integer(2) numXPixels                !X 轴方向的像素数
  Integer(2) numYPixels                !Y 轴方向的像素数
  Integer(2) numTextCols               !可利用的文本列数
  Integer(2) numTextRows               !可利用的文本行数
  Integer(2) numColors                 !颜色索引数
  Integer(4) fontSize                  !缺省字体大小
  Character(80) title                  !C 字符串格式的窗口标题
  Integer(2) bitsPerPixel              !每像素位数，该值由系统确定
  Character(32) extendFontName         !非比例多字节字体
  Integer(4) extendFontSize            !多字节字体大小
  Integer(4) extendFontAttributes      !多字节字体属性，如 Bold、Italic
End Type windoConfig
```

加入没有调用 SetWindowConfig 例程来设置 windowConfig 派生类型，缺省的活动窗口采用系统最优分辨率、8×16 字体大小以及以来与显示卡的颜色数。可以通过规定影响窗口大小的参数，如 X/Y 像素数、行列数和文字大小，来设置活动窗口的实际大小。

X/Y 像素数、行列数和字体大小是相互关联的，不能随意设置。

【例 3-3】将子窗口大小设置为 800×600，字体大小设为 8×12，行列数和颜色数交由系统决定。

例 3-3 程序实现代码为：

```
Program Ex_3
  Use IFQwin
  Implicit None
  Type(windowConfig) wc
```

```
  Logical status
  !Set the x & y pixels to 800*600 and font size to 8*12.
  wc%numXPixels = 800                    !pixels on x-axis,window width
  wc%numYPixels = 600                    !pixels on y-axis,window height
  wc%numTextCols = -1                    !-1 requests system default/calculation
  wc%numTextRows = -1
  wc%numColors   = -1
  wc%title = "Defined Window Title"C
  wc%fontSize = #0008000C                !Request 8*12 pixel fonts
  Open(11,File="User")
  status = SetWindowConfig(wc)
  Write(11,*) "Hello 11"
End Program Ex_3
```

其中，wc%numTextCols、wc%numTextRows 和 wc%numColors 成员赋值-1，表示其值由系统决定；wc%title 成员值后添加了 C 后缀，表示是 C 语言格式字符串(带有\0 结束符)；wc%fontSize 成员赋常量值#0008000C,左边的 0008(8)规定字体宽度 8 个像素，右边的 000C(12)则规定字体高度 12 个像素。

假如规定的设置不满足系统要求，SetWindowConfig 函数返回逻辑假(.False.)，系统进一步计算最匹配设置的参数。所以，若再一次调用 SetWindowConfig 函数，就会获得匹配的设置。如下列语句所示：

```
status=SetWindowConfig(wc)
If (.NOT.status) status=SetWindowConfig(wc)
```

值得注意的是：SetWindowConfig 函数是针对当前活动子窗口进行设置的，和子窗口是否拥有焦点无关；一旦调用了 SetWindowConfig 函数，当前活动子窗口就被重新设置，当中原来的内容会消失。所以，若要对子窗口进行设置，通常是在打开子窗口后就立即调用函数 SetWindowConfig。

### 3.1.4 控制窗口大小和位置

SetWSizeQQ 和 GetWSizeQQ 例程用来设置和获取可视子窗口的大小和位置。可视子窗口的位置和尺寸以字符高度和宽度为单位，框架窗口则按屏幕像素表示。窗口的大小和位置信息在 QWInfo 派生类型中定义：

```
Type QWInfo
  Integer(2) Type                    !SetWSizeQQ 执行的动作类型
  Integer(2) X                       !窗口左上角 X 坐标
  Integer(2) Y                       !窗口左上角 Y 坐标
  Integer(2) H                       !窗口高度
  Integer(2) W                       !窗口宽度
End Type QWInfo
```

在设置或获取可视窗口位置、大小时，若是子窗口则直接规定其单元号；若是框架窗口则用符号常量 QWIN$FRAMEWINDOW 代替单元号。

【例 3-4】将子窗口大小设置为 3080，并在缺省图形窗口中予以回显；或取当前框架窗口尺寸的最大值。

例 3-4 程序实现代码为：

```
Program Ex_4
  Use IFQwin
  Implicit None
  Integer(4) status
  Type(QWInfo) winfo
  Open(11,File="User")
  winfo.H = 30
  winfo.W = 80
  winfo.Type = QWIN$SET
  status = SetWSizeQQ(4,winfo)
  !Get current size of child window associated with unit 4
  status = GetWSizeQQ(4,QWIN$SIZECURR,winfo)
  Write(*,*) "Child windows size is ",winfo.H,"by",winfo.W
  !Get maximum size of frame window.
  status = GetWSizeQQ(QWIN$FRAMEWINDOW,QWIN$SIZEMAX,winfo)
  Write(*,*) "Max frame window size is ",winfo.H,"by",winfo.W
End Program Ex_4
```

程序运行结果：框架窗口尺寸的最大值为 702×1024,该值和屏幕分辨率有关。

通过 SetWSizeQQ 函数设置窗口位置、大小后，窗口在外观上立即发生改变，这一点与 SetWindowConfig 函数不同；另外，SetWSizeQQ 函数不仅可设置当前活动窗口，还可设置框架窗口。

### 3.1.5 保持子窗口显示

在 QuickWin 应用程序退出前，创建的子窗口一直保持打开状态，除非执行了 Close 命令。Close 命令有一个可选命名参数 Status，它的缺省值为“Delete”，即关闭窗口的同时删除窗口；该函数的另一个取值为“Keep”，它保持子窗口是打开状态，但该子窗口是无效的，即针对该子窗口进行的 I/O 操作不起作用，并在该窗口的标题上追加“Closed”。

【例 3-5】使用带有“Keep”参数选项的 Close 命令关闭窗口，并验证关闭后的子窗口是无效的。程序运行结果见图 3-2。

例 3-5 程序实现代码为：

```
Program Ex_5
  Implicit None
  Open(Unit=11,File="User",Title='Child 1')
  Write(11,*) "Enter matrix type:"
  Open(Unit=12,File="User",Title='Child 2')
  Write(12,*) "To be closed"
  Close(12,Status='Keep')
  Write(12,*) "Invalid"
End Program Ex_5
```




图 3-2 保持打开状态的关闭窗口

程序中，使用带“Keep”参数的 Close 命令关闭单元号 12 的子窗口，随后针对该子窗口的 Write 语句并不产生输出。

## 3.2 设置图形输出选项

### 3.2.1 控制光标显示

在 QuickWin 中，可以由 DisplayCursor 例程控制光标是否显示。该例程参数有$GCURSORON 和$GCURSOROFF 两个取值，前者是打开光标显示，后者则是关闭光标显示。在默认情况下调用 SetWindowConfig 函数设置活动窗口属性后，子窗口内的光标不显示；若要使光标显示，须调用 DisplayCursor 例程。

【例 3-6】调用 SetWindowConfig 函数来设置子窗口属性，并保持该子窗口内的光标处于显示状态。

例 3-6 程序实现代码为：

```
Program Ex_6
  Use IFQwin
  Implicit None
  Type(windowConfig) wc
  Logical status
  Integer(4) res
  wc%numXPixels = 900                !pixels on x-axis,window width
  wc%numYPixels = 600                !pixels on y-axis,window height
  wc%numTextCols = -1                !-1 requests system default/calculation
  wc%numTextRows = -1
  wc%numColors   = -1
  wc%title = "Defined Window Title"C
  wc%fontSize = #0008000C            !Request 8*12 pixel fonts
```

```
  Open(11,File="User")
  status = SetWindowConfig(wc)
  res = DisplayCursor($GCURSORON)
  Write(11,*) "Cursor is on"
End Program Ex_6
```

## 3.2.2 设置绘图坐标系

在 Intel Fortran 中，绘图坐标分为物理坐标、视口坐标和窗口坐标。

### 3.2.2.1 物理坐标系

物理坐标指窗口客户区的屏幕坐标，单位是像素。假如调用 SetWindowConfig 例程将活动子窗口的客户区设置成 640×480，即水平方向包含 640 个像素，垂直方向包含 480 个像素，那么物理坐标系如图 3-3 所示。物理坐标系原点(0,0)在客户区左上角，X 轴从左到右，Y 轴从上到下，当前客户区右上角的坐标为(639,479)，而不是(640,480)。

假如程序没有调用 SetWindowConfig 例程设置活动子窗口属性，缺省视口(绘图区)大小为 640×480，可以通过调用 SetViewOrg 例程，来重新设置视口原点。 该例程需要传入 3 个参数：前两个整型参数分别代表视口原点的 X 和 Y 物理坐标，第三个参数为 xyCoord 派生类型，代表视口先前原点的物理坐标。例如，下列代码将视口原点设为(50,100)：

```
Type (xyCoord) origin
Call SetViewOrg(Int2(50),Int2(100),origin)
```

改变后的视口原点位置如图 3-4 所示。当前视口坐标范围：X 从-50 到+589，Y 从-100 到+379。新的视口原点将影响使用视口坐标绘图的例程，这些例程包括：MoveTo、LineTo、Rectangle、Ellipse、Polygon、Arc 和 Pie。

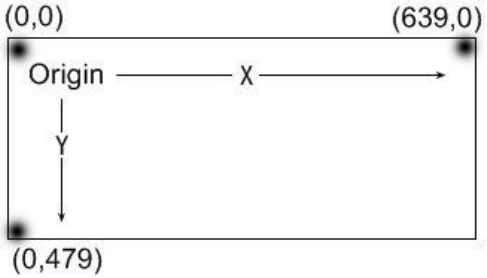



图 3-3 绘图物理坐标系

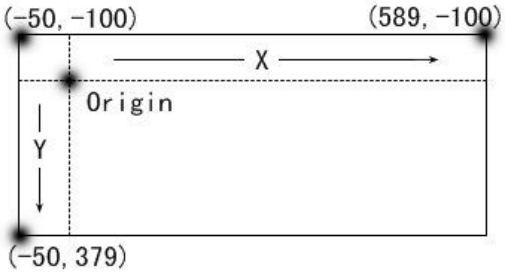
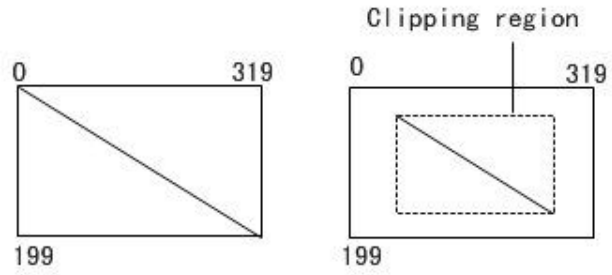



图 3-4 通过 SetViewOrg 设置视口原点

SetClipRgn 例程用来创建裁剪区，用户只能在裁剪区内绘图，在裁剪区外绘图无效。缺省的裁剪区为整个屏幕，现假定屏幕分辨率为 320×200 像素，要画一条从左上角(0,0)到右下角(319,199)的对角线，并用下列语句设置裁剪区：

```
Call SetClipRgn(Int2(10),Int2(10),Int2(309),Int2(189))
```

调用同样的 LineTo 例程画对角线，不设置裁剪区和设置裁剪区的效果如图 3-5 所示。图中的虚线只是用来标识裁剪区范围，在屏幕上并不显示。




### 3.2.2.2 视口坐标系

视口通常指部分窗口客户区，它采用的坐标仍然是物理坐标，即用屏幕像素来表达。SetViewport 例程用来在客户区内建立一个新的视口，一个视口有两个特征：

图 3-5 设置视口裁剪区的效果

(1)左上角为视口原点

(2)视口范围即为缺省的裁剪区。

调用设置视口例程 SetViewport，相当于调用设置视口原点 SetViewOrg 和裁剪区 SetClipRgn 两个例程，既规定了屏幕上一个受限的区域，又将视口原点设置在该区域的左上角。

【例 3-7】将视口原点设为(0,30)，视口范围设为 250×70 像素，从视口左上角到右下角画对角线，并画出视口边界。程序运行结果见图 3-6。

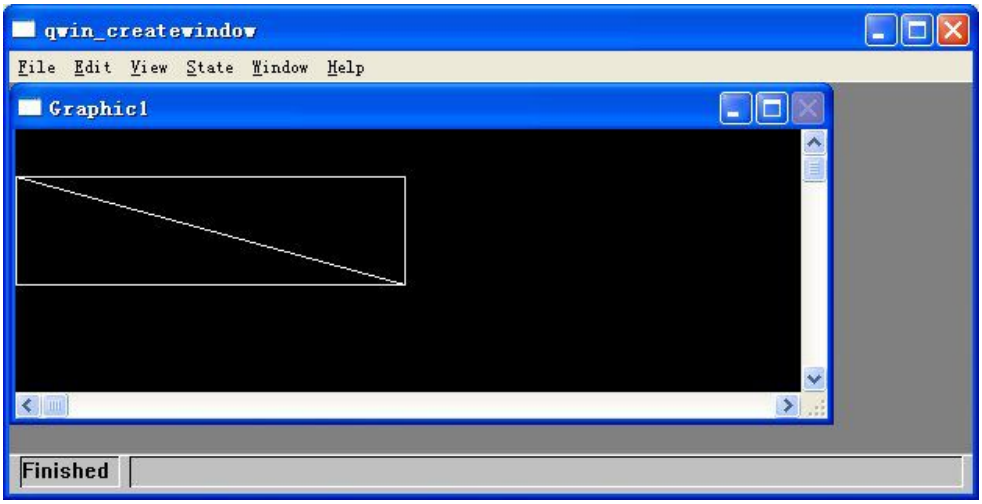



图 3-6 例 3-7 程序运行结果

例 3-7 程序实现代码为：

```
Program Ex_7
  Use IFQwin
  Implicit None
  Integer(2) upX,upY
  Integer(2) downX,downY
  Integer(2) res
  Type(xycoord) pos
  upX = 0
  upY = 30
  downX = 250
  downY = 100
  Call SetViewport(upX,upY,downX,downY)
  Call MoveTo(Int2(0),Int2(0),pos)
  res = LineTo(Int2(249),Int2(69))
  res = Rectangle($GBORDER,Int2(0),Int2(0),Int2(249),Int2(69))
End Program Ex_7
```

### 3.2.2.3 窗口坐标系

视口坐标系采用的是物理坐标，要求是整数。但许多实际问题的数据都是实数，如频率、粘性、质量等，在这种情况下，采用窗口坐标系绘图更为方便。

窗口坐标以双精度实型数表示当前视口范围，范围外的图形同样被裁剪。假定要绘制金星上一年的平均温度过程线，其温度范围设为-50 到+450，对此可调用 SetWindow 例程设置如下的窗口坐标系：

```
status = SetWindow(.TRUE.,1.0D0,-50.0D0,12.0D0,450.0D0)
```

其中,第一个参数为Y轴坐标的反转标识,逻辑真表示 Y 轴朝上,后两对参数分别表示视口左下角和右上角的窗口坐标(若 Y 轴朝下,则分别表示视口左上角和右下角的窗口坐标)。视口角点的窗口坐标如图 3-7 所示，1 月和 12 月分别对应视口的左边和右边，绘制的温度过程线自然被缩放到该视口内。

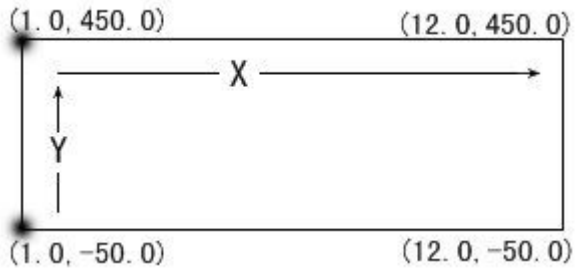



图 3-7 设置窗口坐标系

通常，使用窗口坐标绘图要经历一下步骤：

(1)设置窗口属性(SetWindowConfig)。

(2)创建视口区(SetViewport)。若计划在整个客户区绘图，这一步可以省略。

(3)设置窗口坐标系(SetWindow)。传入一个表示 Y 轴反转标识的逻辑参数，和两对双精度实型参数——分别表示 X 和 Y 的最小值和最大值。

(4)调用绘图例程(MovieTo_W、LineTo_W、Rectangle_W 等)绘图。注意区别两类不同的绘图例程：使用视口坐标绘图的例程尾部没有“_W”后缀；反之，使用窗口坐标绘图的例程尾部有“_W”后缀。

窗口坐标既能适应较小的取值范围(如从 151.25 到 151.45)，又能适应较大的取值范围(如从-50000.0 到+80000.0)，而且避免了从实际数据到像素图形数据转换的麻烦，并提高了绘图的精度；通过改变窗口坐标范围，很容易实现图形的缩放和预览功能；窗口坐标使得绘图不依赖于计算机硬件，即独立于实际屏幕分辨率。

## 3.2.3 设置颜色

IFQWin 运行库支持彩色图形。可利用颜色总数取决于计算机显示卡和颜色的系统设置，实际使用的颜色数受 QuickWin 绘图例程限制。

假如使用的是 VGA 机器，一次最多显示 256 色，尽管可从调色板 262144(256K)色范围内选取，但一次只能选取 256 色；有些适配器(大多数 SVGA)能够显示所有的 256K 色；有些适配器(真彩色适配器)甚至能显示 256×256×256=16777216 色。若使用调色板，则要受到调色板可利用颜色限制。若要访问系统所有颜色，须规定一个 RGB(Red-Green-Blue)值，而不是调色板颜色索引。

要设置前景颜色，背景颜色和文本颜色，可以分别调用 SetColor、SetBkColor、SetTextColor 例程，或 SetColorRGB、SetBkColorRGB 和 SetTextColorRGB 例程。没有 RGB 后缀的例程，须传入调色板颜色索引参数；带 RGB 后缀的例程，须传入 RGB 浓度值参数,可从系统整个颜色范围内选取。

RGB 浓度值是一个 3 字节(24 位)的整型数，其二进制结构如下：

```
BBBBBBBB GGGGGGGG RRRRRRRR
```

当中的 R、G 和 B 分别代表红色、绿色和蓝色值，每种颜色浓度值范围为 0～255。例如，

要设置淡红色(粉红色)，其红、绿、和蓝浓度值分别为 255、128 和 128：

10000000 10000000 11111111

相当于 16 进制下的#8080FF。若要设置前景色为粉红色，可进行如下的函数调用：

i = SetColorRGB(#8080FF)

现将一些常用的 RGB 颜色值列于表 3-1。

假如机器是 64K 色的，若设置的 RGB 颜色值不等于预置的 64K 中的 RGB 颜色值，则系统匹配最接近的 RGB 颜色值。

尽管系统使用近似的颜色绘制图形，假如使用 GetColorRGB、GetBkColorRGB 或 GetTextColorRGB 例程来获取颜色，仍然返回设置的颜色，而不是实际采用的颜色。若要返回实际采用的近似颜色，可使用 GetPixelRGB 和 GetPixelsRGB 例程。

表 3-1 十六进制 RGB 颜色值

| 颜色 | RGB | 颜色 | RGB |
|---|---|---|---|
| 黑色 | #000000 | 白色 | #FFFFFF |
| 暗红色 | #000080 | 红色 | #0000FF |
| 暗绿色 | #008000 | 绿色 | #00FF00 |
| 暗黄色 | #008080 | 黄色 | #00FFFF |
| 暗蓝色 | #800000 | 蓝色 | #FF0000 |
| 暗紫色 | #800080 | 紫色 | #FF00FF |
| 暗青绿色 | #808000 | 青绿色 | #FFFF00 |
| 深灰色 | #808080 | 浅灰色 | #C0C0C0 |

值得注意的是：SetColor 和 SetColorRGB 是设置的前景色，包括使用 OutGText 例程输出的文本图形;而 SetTextColor 和 SetTextColorRGB 例程则是用来设置使用 OutText、Write 或 Print 输出的字符颜色。

## 3.2.4 设置图形属性

### 3.2.4.1 设置掩码

程序中，通过 SetFillMask 和 GetFillMask 例程设置和获取填充当前图案的掩码。掩码中有 8 个字节，字节(8 个位)中的每个位代表了一个像素，构成了 8×8 的填充图案。填充时，位置 1 的像素被设置成当前图形颜色，位置为 0 的像素被设置成当前背景颜色，8 字节掩码在整个区域内被重复。设置填充图案的掩码，实际上是通过设计字节数组来获得位图图案，如图 3-8 所示。

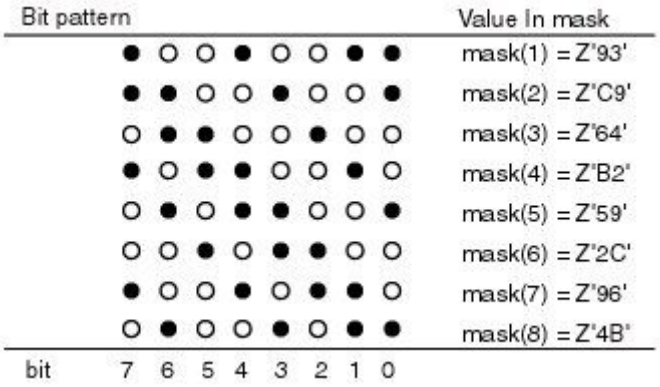



图3-8 不同掩码位置的位图图案

通过 SetFillMask 例程设置的掩码，用来控制形状绘图例程使用的填充图案。形状绘图图例有 FloodFillRGB、Pie、Ellipse、Polygon 和 Rectangle。

【例 3-8】用不同的掩码图案填充矩形。程序运行结果见图 3-9。

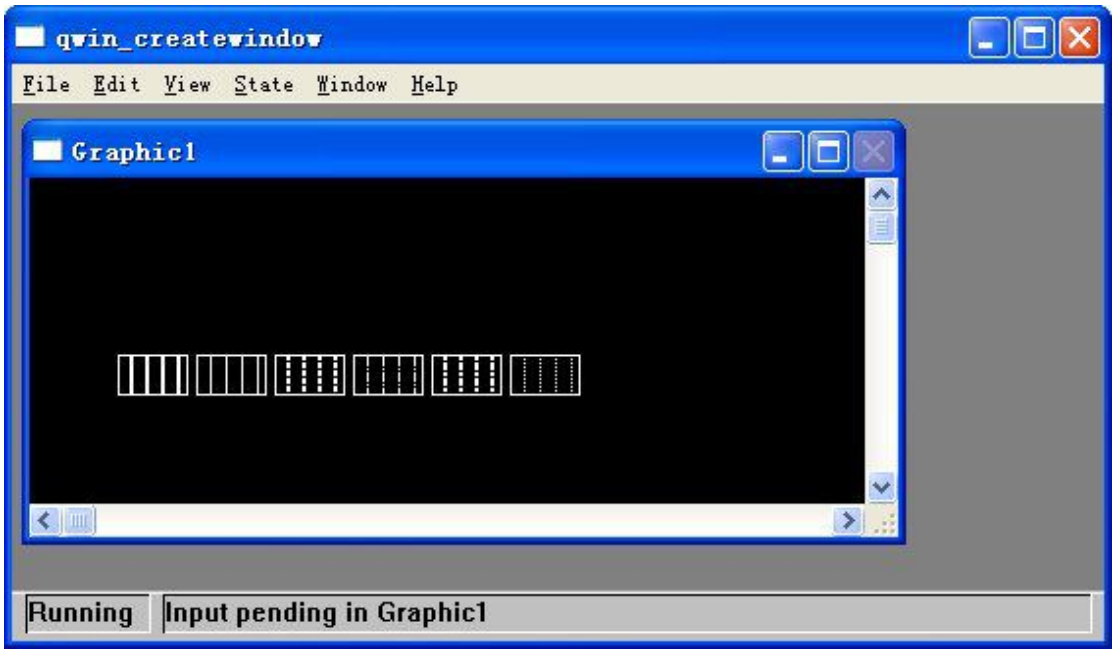



图 3-9 不同的掩码图案填充的矩形

例 3-8 程序实现代码为：

```
Program Ex_8
  Use IFQwin
  Implicit None
  Integer(1),Target::Style1(8)=&
    (/Z'18',Z'18',Z'18',Z'18',Z'18',Z'18',Z'18',Z'18'/)
  Integer(1),Target::Style2(8)=&
    (/Z'08',Z'08',Z'08',Z'08',Z'08',Z'08',Z'08',Z'08'/)
  Integer(1),Target::Style3(8)=&
    (/Z'18',Z'00',Z'18',Z'18',Z'18',Z'00',Z'18',Z'18'/)
  Integer(1),Target::Style4(8)=&
    (/Z'00',Z'08',Z'00',Z'08',Z'08',Z'08',Z'08',Z'08'/)
  Integer(1),Target::Style5(8)=&
    (/Z'18',Z'18',Z'00',Z'18',Z'18',Z'00',Z'18',Z'18'/)
  Integer(1),Target::Style6(8)=&
    (/Z'08',Z'00',Z'08',Z'00',Z'08',Z'00',Z'08',Z'00'/)
  Integer(1) oldStyle(8) !Placeholder for old style
  Integer loop
  Integer(1),Pointer::ptr(:)
  Integer(2) status

  Call GetFillMask(oldStyle)
  !Make 6 rectangles,each with a different fill
```

```
  Do loop = 1 , 6
    Select Case (loop)
      Case (1)
        ptr => style1
      Case (2)
        ptr => style2
      Case (3)
        ptr => style3
      Case (4)
        ptr => style4
      Case (5)
        ptr => style5
      Case (6)
        ptr => style6
    End Select
    Call SetFillMask(ptr)
    status = Rectangle($GFILLINTERIOR,Int2(loop*40+5),&
                  Int2(90),Int2((loop+1)*40),Int2(110))
  End Do
  Call SetFillMask(oldStyle)        !Restore old style
  Read(*,*)                         !Wait for Enter to be pressed
End Program Ex_8
```

### 3.2.4.2 设置逻辑写模式

逻辑写模式决定了要绘制的图形、已存在的图形及当前颜色图形颜色之间的相互作用。当用 LineTo、Rectangle 和 Polygon 例程绘制图形时，SetWriteMode 和 GetWriteMode 例程分别用来设置和获取当前的逻辑写模式。逻辑写模式可以设为$GPSET(缺省)、$GAND、$GOR、$GPRESENT 及$GXOR，如表 3-2 所示：

表 3-2 逻辑写模式

| 逻辑写模式 | 说明 |
|---|---|
| $GPSET(缺省) | 以当前图形颜色画图 |
| $GAND | 以当前图形颜色和背景颜色的逻辑与操作颜色画图 |
| $GOR | 以当前图形颜色和背景颜色的逻辑或操作颜色画图 |
| $GXOR | 以当前图形颜色和背景颜色的逻辑异或操作颜色画图 |

【例 3-9】在黄色背景上，分别采用$GPSET(缺省)和$GAND 逻辑写模式画两条直线。

例 3-9 程序实现代码为：

```
Program Ex_9
  Use IFQwin
  Implicit None
  Integer(2) result,oldMode
  Integer(2) oldColor
  Type(xycoord) xy
```

```
  oldColor = SetBkColorRGB(Z'00FFFF')      !yellow
  Call ClearScreen($GCLEARSCREEN)
  oldColor = SetColorRGB(Z'FF00FF')        !purple
  Call MoveTo(Int2(0),Int2(0),xy)
  result = LineTo(Int2(200),Int2(200))
  oldMode = SetWriteMode($GAND)
  Call MoveTo(Int2(50),Int2(0),xy)
  result = LineTo(Int2(250),Int2(200))    !red
End Program Ex_9
```

程序中，第一次画直线采用缺省的逻辑写模式，即以当前图形颜色(紫色)画线；第二次画直线采用$GAND 逻辑写模式，即以当前图形颜色(紫色)和背景颜色(黄色)的逻辑与操作颜色(红色)画线。

#### 3.2.4.3 设置画线式样

式样(如实线、虚线、点画线等)影响所画直线或由直线围成的形状图形的外观。在 QuickWin 中，SetLineStyle 和 GetLineStyle 例程分别用来设置和获取画线式样，共有 5 种选择的样式，如表 3-3 所示。

表 3-3 5 种直线样式

| QuickWin 式样(十六进制) | Windows 式样 | 直线外观 |
|---|---|---|
| 0xFFFF | PS_SOLID | ———————— |
| 0xEEEE | PS_DASH | - - - - - - - - |
| 0xECEC | PS_DASHDOT | -.-.-.-.-.-.-.-. |
| 0xECCC | PS_DASHDOTDOT | -..-..-..-..-..- |
| 0xAAAA | PS_DOT | . . . . . |

设置的画线式样只对直线围成的形状绘图例程 LineTo、Rectangle 和 Polygon 起作用，对画曲线的绘图例程 Arc、Ellipse、Pie 不起作用。

【例 3-10】以实线式样画直线。

例 3-10 程序实现代码为：

```
Program Ex_10
  Use IFQwin
  Implicit None
  Integer(2) status,style
  Type(xycoord) xy
  style = Z'FFFF'
  Call SetLineStyle(Style)
  Call MoveTo(Int2(50),Int2(50),xy)
  status = LineTo(Int2(300),Int2(300))
End Program Ex_10
```

## 3.3 输出图形和文本

### 3.3.1 绘制图形

在调用绘图例程画图时，若不想使用 QuickWin 缺省的绘图选项(实线式样、没有掩码、黑背景色、白前景色)，则可按上一节介绍设置特定的绘图选项。新设置的绘图选项将影响随后的绘图操作，直到这些选项被重新设置，或打开一个新的子窗口为止。

表 3-4 列出了 IFQWin 运行库中与绘图相关的例程。

表 3-4 IFQWin 运行库中与绘图相关的例程

| 例程 | 说明 |
|---|---|
| Arc，Arc_W | 画一条弧 |
| ClearScreen | 清空屏幕、视口及文本窗口 |
| Ellipse,Ellipse_W | 画一个椭圆或圆 |
| FloodFill,FloodFill_W | 用当前的颜色索引、填充图案填充屏幕上的一个封闭区域 |
| FloodFillRGB,<br>FloodFillRGB_W | 用当前的 RGB 颜色值、填充图案填充屏幕上的一个封闭区域 |
| GetArcInfo | 确定最近绘制的弧或饼图的终止点 |
| GetCurrentPosition，<br>GetCurrentPosition_W | 获取当前图形输出的位置目标 |
| GetPixel,GetPixel_W | 获取一个像素的颜色索引 |
| GetPixelRGB,<br>GetPixelRGB_W | 获取一个像素的 RGB 颜色值 |
| GetPixels | 获取多个像素的颜色索引 |
| GetPixelsRGB | 获取多个像素的 RGB 颜色索引 |
| GRStatus | 获取最近调用的绘图例程执行情况(成功或失败) |
| IntegerToRGB | 将一个实颜色值转换为 RGB 三原色 |
| LineTo,LineTo_W | 从当前图形输出位置到一个特定点画一直线 |
| LineToAr/LineToArEx | 由 X、Y 坐标数组画一组同色同风格/不同色不同风格的直线 |
| MoveTo,MoveTo_W | 将当前图形输出位置移至特定点 |
| Pie,Pie_W | 画一扇形 |
| Polygon,Polygon_W | 画一多边形 |
| Rectangle,Rectangle_W | 画一矩形 |
| RGBToInteger | 将 RGB 三原色转换为一个实颜色值 |
| SetPixel,SetPixel_W | 设置一个像素点的颜色索引 |
| SetPixelRGB,SetPixelRGB_W | 设置一个像素点的 RGB 颜色值 |
| SetPixels | 设置多个像素点的颜色索引 |
| SetPixelsRGB | 设置多个像素点的 RGB 颜色值 |

注：带“_W”后缀的例程使用窗口坐标绘图；不带“_W”后缀的例程使用视口坐标绘图

值得注意的是，曲线图(如弧、椭圆等)由其外接矩形设定。外接矩形的中心作为曲线的中心，外界矩形的边界决定了曲线的大小。外界矩形本身则由其左上角和右下角的坐标定义，如图 3-10 中，点(x1,y1)和点(x2,y2)定义了当前的外接矩形。

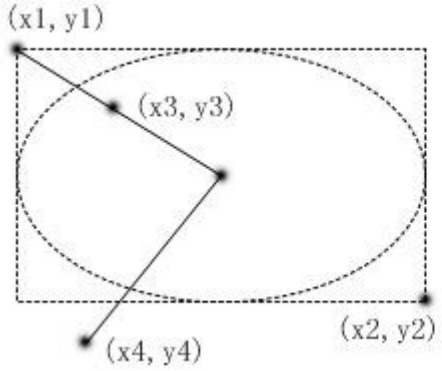



图 3-10 曲线图的外接矩形

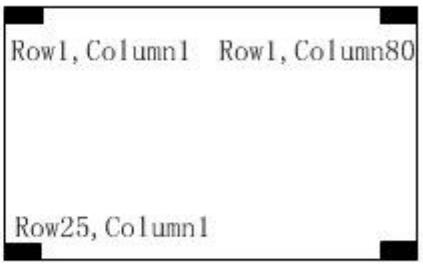



图 3-11 文本窗口坐标

### 3.3.2 输出字符文本

当利用 OutText 例程、Write 和 Print 语句输出基于字符的文本时，使用的是文本窗口坐标。在 QuickWin 中，屏幕被映射到子窗口，子窗口既可用于绘图，也可用于输出字符文本。用于绘图时，子窗口成为绘图窗口；用于输出字符文本时，子窗口成为文本窗口。文本窗口坐标以行、列来定位输出的字符文本，如图 3-11 所示。

文本窗口坐标遵循以下两条规则：

(1)行、列数从 1 开始，假如一个文本窗口为 25 行×80 列(图 3-11)，其行、列数分别为 1～25 和 1～80。

(2)行坐标排在列坐标前。

表 3-5 列出了 IFQWin 运行库中用于字符文本输出的相关例程。

表 3-5 IFQWin 运行库中用于字符文本输出的相关例程

| 例程 | 说明 |
|---|---|
| ClearScreen | 清空屏幕、视口及文本窗口 |
| DisplayCursor | 设置光标是否展示 |
| GetBkColor | 获得当前背景颜色索引 |
| GetBkColorRGB | 获得当前背景颜色 RGB 颜色值 |
| GetTextColor | 获得当前文本颜色索引 |
| GetTextColorRGB | 获得当前文本 RGB 颜色值 |
| GetTextPosition | 获得当前文本输出位置 |
| GetTextWindow | 获得当前文本窗口边界 |
| OutText | 在指定位置输出文本 |
| ScrollTextWindow | 滚动文本窗口内容 |
| SetBkColor | 设置当前背景颜色索引 |
| SetBkColorRGB | 设置当前背景 RGB 颜色值 |
| SetTextColor | 设置当前文本颜色索引 |
| SetTextColorRGB | 设置当前文本 RGB 颜色值 |
| SetTextPosition | 设置当前文本输出位置 |
| SetTextWindow | 设置当前的文本展示窗口 |
| WrapOn | 设置文本是否环绕 |

类似于在视口内绘制图形，在文本窗口内输出字符文本，也应先设置文本窗口(缺省的

文本窗口为子窗口整个客户区，即屏幕)。在调用 SetTextWindow 例程设置文本窗口时，使用的是行、列坐标，输出的文本被限定在文本窗口范围内。

【例 3-11】在缺省子窗口内定义两个文本窗口，分别输出环绕字符文本与非环绕字符文本。

例 3-11 程序实现代码为：

```
Program Ex_11
  Use IFQwin
  Implicit None
  Integer(2) row,status2
  Integer(4) status4,i
  Type(rccoord) curPos
  Type(windowConfig) wc
  Logical status
  status = GetWindowConfig(wc)
  wc%numTextCols    = 80
  wc%numXPixels     = -1
  wc%numYPixels     = -1
  wc%numTextRows    = -1
  wc%numColors      = -1
  wc%fontSize       = -1
  wc%title = "This is a test"C
  status = SetWindowConfig(wc)
  status4 = SetBkColorRGB(#FF0000)
  Call ClearScreen($GCLEARSCREEN)

  !Display wrapped and unwrapped text in text windows.
  Call SetTextWindow(Int2(1),Int2(1),Int2(5),Int2(25))
  Call SetTextPosition(Int2(1),Int2(1),curPos)
  status2 = WrapOn($GWRAPON)
  status4 = SetTextColorRGB(#00FF00)
  Do i = 1 , 5
    Call OutText('Here text does wrap.')
  End Do
  Call SetTextWindow(Int2(7),Int2(10),Int2(11),Int2(40))
  Call SetTextPosition(Int2(1),Int2(1),curPos)
  status2 = WrapOn($GWRAPOFF)
  status4 = SetTextColorRGB(#008080)
  Do row = 1 , 5
    Call SetTextPosition(Int2(row),Int2(1),curPos)
    Call OutText('Here text doesnot wrap.')
  End Do
  Read(*,*) !Wait for ENTER to be pressed
End Program Ex_11
```

程序中，定义的第一个文本窗口的行、列数分别为 1～5 和 1～25；第二个文本窗口的行、列数分别为 7～11 和 10～40。

## 3.3.3 输出字体文本

### 3.3.3.1 概述

字体是具有特定字样和大小的文本字符集，字样指文本被展示的式样，如 Arial、Times New Roman、宋体；字体大小指单一字符所占据的屏幕面积，以屏幕像素为单位，例如“Courier12 9”，表示字样为 Courier，其每个字符占 12 像素(水平)×9 像素(垂直)的屏幕面积。

IFQWin 运行库的字体例程能够使用 Windows 操作系统上安装的所有字体，Windows 字体分位图字体和真实字体(TrueType)两种类型。位图字体，其每个字符对应一个二进制数据栅格图。栅格图中的每个位对应一个屏幕像素。位值为 1，对应的像素被设置为当前屏幕颜色；位值为 0，对应的像素被设置为当前背景颜色。

真实字体，其字符在屏幕上显示和在打印机上输出是相同的。它可以是位图字体，也可以是软字体(字体在打印前被下载到打印机)，具体取决于打印机的性能。真实字体可以被缩放到任意高度，因此在图形程序中提倡使用真实字体。

两种类型的字体各有其优缺点，位图字体由于预先定义了像素映射关系，其字符在屏幕上显示得较为光滑，但不能被缩放；真实字体可以被缩放到任意大小，但其字符在屏幕上显示不像位图字体那么实，在打印机输出时却能达到位图字体的光滑程度，甚至比位图字体还要光滑。

在 QuickWin 中，基于字体的文本按图形对待，并使用 OutGText 例程输出。IFQWin 运行库中用于字体文本输出的相关例程列于表 3-6。

表 3-6 IFQWin 运行库中的字体文本输出相关例程

| 例程 | 说明 |
|---|---|
| GetFontInfo | 获得当前字体信息 |
| GetGTextExtent | 获取以当前字体输出的文本宽度 |
| GetGTextRotation | 获取当前字体文本输出的方位角(单位:0.1°) |
| InitializeFonts | 初始化字体库 |
| OutGText | 在当前图形输出位置输出当前字体文本 |
| SetFont | 选取与规定的字体特性相匹配的字体，以用于 OutGText 例程的文本输出 |
| SetGTextRotation | 设置当前字体文本输出的方向角(单位:0.1°) |

注：只有真实字体才能设置或获取文本输出的方向角。

### 3.3.3.2 输出字体文本的步骤

要输出字体文本，须经历以下三个步骤。

1)初始化字体

初始化字体，旨在将系统所有字体组织到内存列表中，以提供计算机可利用的字体信息。要实现字体的初始化，须调用 InitializeFonts 例程。InitializeFonts 例程的语法格式为：

```
result = InitializeFonts()
```

其中，result 为 Integer(2)的函数返回结果，假如初始化字体成功，函数返回初始化字体的数量；假如失败，则返回负的错误码。

2)选取字体

在输出字体文本前，必须使用 SetFont 例程从初始化的字体中选取特定的字体，作为当前字体；否则，会导致随后调用的 OutGText 函数输出字体文本失败(因为 QuickWin 并不设置缺省字体)。SetFont 例程的语法格式为：

result = SetFont(options)

result 为 Integer(2)的函数返回结果，假如选取字体成功，函数返回被选取字体的索引号；否则，返回-1。

options 为 Character*(*)参数，代表描述字体特征的字符串，它由表 3-7 所列的字符选项构成。

假如规定 nx 选项，SetFont 忽略其他的所有选项，只提供索引号为 x 的字体；假如规定相互排斥的选项(f/p 和 r/v)，SetFont 忽略它们；假如 b 选项被规定，同时至少有 1 个字体被初始化，SetFont 选取一个字体并返回 0，以标识选取字体成功。

假如规定了多个选项，SetFont 函数按下列优先次序选取字体：①像素高度；②字样；③像素宽度；④固定或比例间隔字体。

表 3-7 描述字体特性的选项

| 选项 | 描述 |
|---|---|
| t'fontname' | 字样名，可以使系统安装的任何字体 |
| hy | 字符高度，其中的 y 是以像素表示的高度值 |
| wx | 字符宽度，其中的 x 是以像素表示的宽度值 |
| f | 只选取固定间隔的字体 |
| p | 只选取比例间隔的字体 |
| v | 只选取矢量映射字体,或称为绘图仪字体(如 Roman、Modern、Script),真实字体(如 Arial、Symbol、Times New Roman)不属于矢量映射字体 |
| r | 只选取光栅映射字体，或称为位图字体(如 Courier、Helvetica、Palatino)，真实字体不属于光栅映射字体 |
| e | 选取加粗文本格式，假如选取的字体不支持加粗文本格式，该选项被忽略 |
| u | 选取下划线本格式，假如选取的字体不支持下划线文本格式，该选项被忽略 |
| i | 选取倾斜文本格式，假如选取的字体不支持倾斜文本格式，该选项被忽略 |
| b | 选取与其他选项最匹配的字体 |
| nx | 选取索引号为 x 的字体，索引好为 x 小于等于由 InitializeFonts 函数返回的字体数 |

注：选项参数对字母大小写和位置不敏感

假如规定了 b、t 选项和不存在的像素高度和宽度值，SetFont 选择最匹配的字体。在选择过程中，小字体优先于大字体，如规定 Arial 12 和 b 选项的字体，而只有 Arial 10 和 Arial 14 可用，SetFont 选择 Arial 10。

假如只规定了 b 选项和不存在的像素高度和宽度值，SetFont 函数使用一个缩放比例因子来选择适当大小的矢量映射字体。

假如规定字体高度值，而没有规定宽度值，SetFont 按正确的字体比例给出宽度值；假如规定字体宽度值，而没有规定高度值，SetFont 采用一个缺省的字体高度值，这可能导致字符变形，尤其是规定的宽度值较大时。

GetFontInfo 函数用来获取由 SetFont 函数设置的当前字体信息。GetFontInfo 函数的语法格式为：

result = GetFontInfo(font)

result 为 Integer(2)得函数返回结果。假如函数调用成功，返回值为 0；否则，返回值为-1。

font 为派生类型 FontInfo 参数，用来描述当前字体的特征集。派生类型 FontInfo 的定义为：

```
Type FontInfo
  Integer(4) type                    !真实字体为 1，位图字体为 0
  Integer(4) ascent                  !顶到基线的像素距离
  Integer(4) pixWidth                !像素字符宽度，比例间隔字体为 0
  Integer(4) pixHeight               !像素字符高度
  Integer(4) avgWidth                !平均像素字符宽度
  Character(81) fileName             !包括路径在内的文件名
  Character(32) faceName             !字体字样名
  Logical(1) italic                  !倾斜文本格式为.True.
  Logical(1) emphasized              !加粗文本格式为.True.
  Logical(1) underline               !下划线文本格式为.True.
End Type FontInfo
```

【例 3-12】获取当前字体的平均像素字符宽度、像素字符高度和像素字符宽度信息。

例 3-12 程序实现代码为：

```
Program Ex_12
  Use IFQwin
  Implicit None
  Type(fontInfo) font
  Integer(2) i,numFonts
  Integer(4) status
  numFonts = InitializeFonts()
  i = SetFont("t'Arial'")
  i = GetFontInfo(font)
  Print *, font.avgWidth,font.pixheight,font.pixWidth
  status = ClickMenuQQ(QWIN$CASCADE)
End Program Ex_12
```

当前字体的平均像素字符宽度、像素字符高度和像素字符宽度分别为 38、40 和 0。

值得注意的是，照常理 SetFont 函数设置字体只影响 OutGText 输出的字体文本，而不影响 OutText、Write 和 Print 输出的字符文本，但实际情况并非如此。此处，在输出字符文本后调用 ClickMenuQQ 函数，执行 QuickWin 缺省菜单 Windows 下的层叠(Cascade)命令，来刷新子窗口，使输出的字符文本正常显示。

### 3)输出字体文本

在初始化字体和选取字体完成后，调用 OutGText 例程输出字体文本。因为字体文本按图形对待，所以字体文本的输出从当前图形输出位置开始，以当前图形颜色作为字体文本颜色。在调用 OutGText 例程输出字体文本后，系统自动更新当前图形输出位置。

【例 3-13】设置高 18、宽 10 的 Arial 倾斜字体，并顺时针旋转 90° 输出字体文本。

例 3-13 程序实现代码为：

```
Program Ex_13
  Use IFQwin
  Implicit None
```

```
  Integer(2) fontNum,numFonts
  Integer(4) deg
  Type(xycoord) pos
  numFonts = InitializeFonts()
  !Set typeface to Arial,character height to 18.
  !character width to 10,and italic
  fontNum = SetFont('t"Arial"h18w10i')
  deg = - 900
  Call SetGTextRotation(deg)
  Call MoveTo(Int2(50),Int2(30),pos)
  Call OutGText('Demo text')
End Program Ex_13
```

其中，SetGTextRotation 例程用来设置输出字体(须是 TrueType 字体)文本的方向角。方向角以 0.1° 为单位，以逆时针方向为正。方向角-900，表示输出的字体顺时针旋转 90°。

在 Intel Fortran 中，字符(串)用单引号和双引号界定。SetFont 函数的参数是由字体选项构成的字符串，当中字样选项 t 的后面需要接字体名(字符串)，即在字符串中再嵌入字符串，这时，要求内字符串和外字符串分别使用不同的界定符：外字符串如果使用双引号，内字符串就要使用单引号；反之，外字符串如果使用单引号，内字符串就要使用双引号。

【例 3-14】将整个屏幕划分生 3 个绘图窗口和 3 个文本窗口，分别采用不同的窗口设置，输出由相同数据集绘制的图形和文本。程序运行结果如图 3-12 所示。

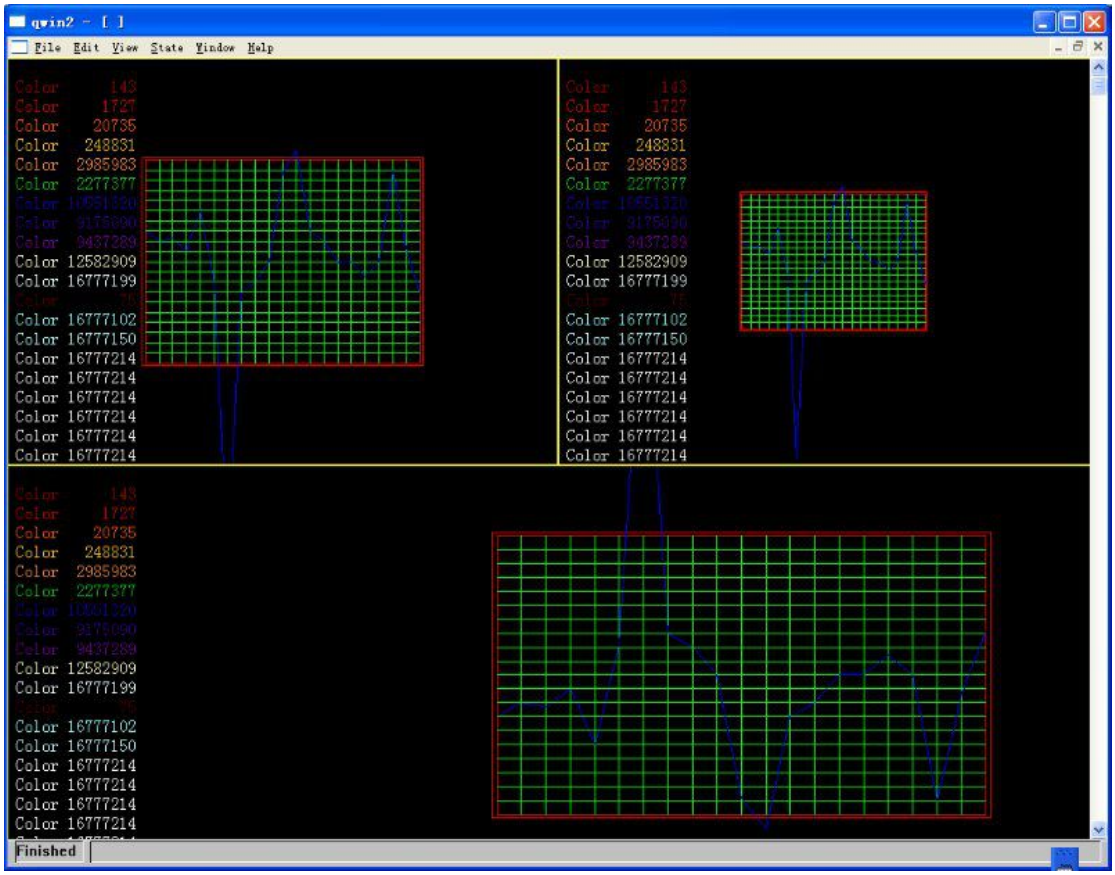



图 3-12 在屏幕上设置 3 个绘图窗口和 3 个文本窗口

(1)设置窗口属性(或绘图模式)

在输出图形和文本程序中，通常先设置活动窗口属性或绘图模式。

例 3-14 主程序实现代码为：

```
Program Ex_14    !Illustrates coordinate graphics.
  Use IFQwin
  Implicit None
  Logical statusMode
  Type(windowConfig) myScreen
  Common myScreen
  !
  !Set the screen to the best resolution and maximum number of
  !available colors.
  myScreen.numXPixels           = 1024
  myScreen.numYPixels           = 768
  myScreen.numTextCols          = -1
  myScreen.numTextRows          = -1
  myScreen.numColors            = -1
  myScreen.title                = " "C
  statusMode = SetWindowConfig(myScreen)
  If (.Not.statusMode) then
    statusMode = SetWindowConfig(myScreen)
  End If
  statusMode = GetWindowConfig(myScreen)
  Call threeGraphs()
End Program Ex_14
```

程序在窗口属性设置完毕，调用 threeGraphs 子程序进一步设置视口、绘图窗口和文本输出窗口。

(2)设置视口、绘图窗口和文本输出窗口。

该示例将活动子窗口(代表屏幕)三划分为 3 个视口，并在视口内进一步设置绘图窗口和文本输出窗口。

子程序 threeGraphs 实现代码为：

```
Subroutine threeGraphs()
  Use IFQWin
  Implicit None
  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 Subroutine threeGraphs
```

程序在视口设置前，先调用 ClearScreent 例程清空屏幕窗口：

```
Call ClearScreen($GCLEARSCREEN)
```

在获取了屏幕窗口的实际分辨率和总的文本行、列数后，threeGraphs 子程序分别创建视口、文本窗口和绘图窗口各 3 个。例如：

```
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)
```

上列第一条语句设置一个覆盖屏幕左上 1/4 的视口；第二条语句设置一个对应的文本窗口；第三条语句设置一个对应的绘图窗口，其 X、Y 坐标均从-2.0 到 2.0，.Flash.常量规定 Y 轴正向朝下，_8 后缀标识常量为 Real(8)。

### (3) 绘图和输出文本

绘图和输出文本的 gridShape 子程序为：

```
Subroutine gridShape(numc)   !GridShape-This subroutine plots data
  Use IFQWin
  Implicit None
  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 grah:
  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 ) then
    numc = myScreen.numColors - 1
  End If
  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 Subroutine gridShape
```

程序中,3 个图形的网格面积都是 20×20，画线使用同一数据集，但在不同的视口、窗口坐标系下，呈现出不同的图形效果(图 3-12)；上半部分右边的图形比左边的小，下半部分的图形是倒置的；3 个文本窗口内展示的文本受文本窗口自身的行、列范围限制，超出范围的字符文本不显示，如同超出视口范围的图形被裁剪。

在实际应用中，绘图通常采用窗口坐标。窗口坐标系实际是用特定的实数范围来定义视口边界，以避免绘图数据(通常是实数)向视口像素坐标的转换，简化绘图操作。这样，超出窗口坐标范围的数据点，因落在视口外而被裁剪掉。

【例 3-15】在两个子窗口内分别输出图形和字符文本。其中，在文本子窗口内提供字符命令菜单，按输入的不同菜单选项，在另一个子窗口内绘制相应的函数图形。程序的运行结果如图 3-13 所示。

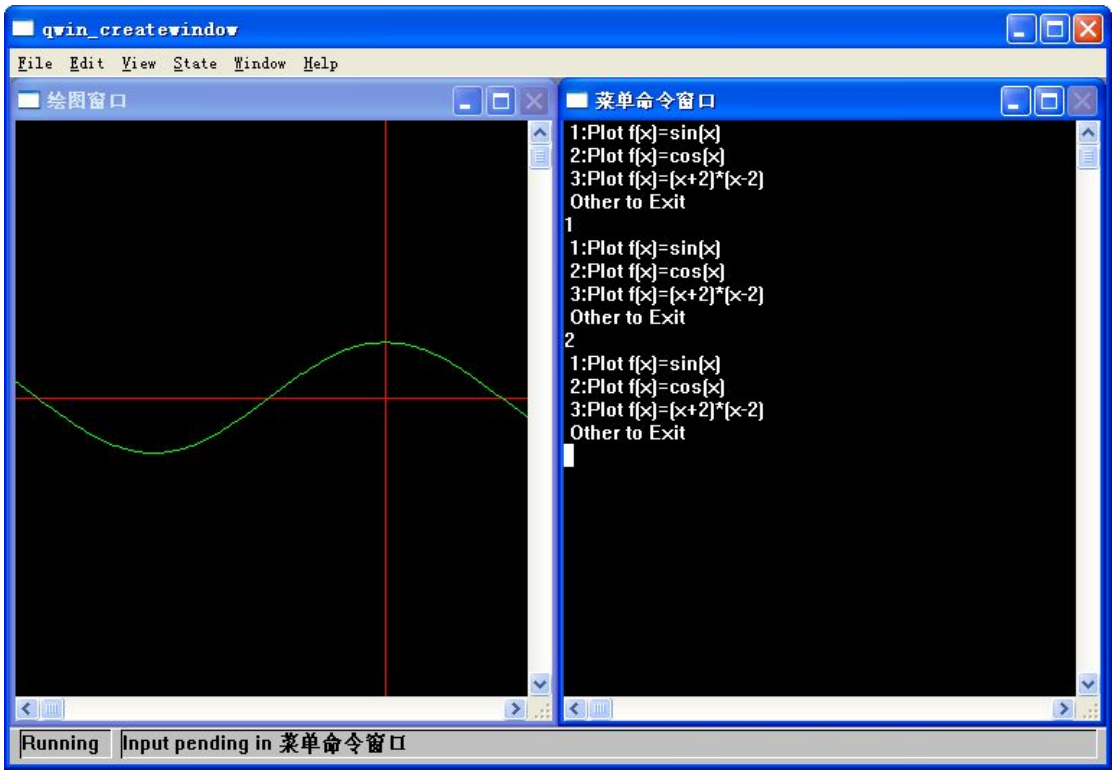



图 3-13 通过子窗口字符菜单控制另一子窗口绘图

*(1)打开绘图和文本输出子窗口*

例 3-15 的主程序实现代码为：

```
Program Ex_15    !主程序
  Use IFQwin
  Use PlotMod
  Implicit None
  Integer::TUnit=17,GUnit=10   !单元号
  Logical::status
  Integer::input=1

  Open(TUnit,File='User',Title="菜单命令窗口"C)    !打开菜单命令窗口
  status = DisplayCursor($GCURSORON)              !打开光标显示
  Open(GUnit,File='User',Title="绘图窗口"C)        !打开绘图窗口
  status = DisplayCursor($GCURSOROFF)             !关闭光标显示
  status = ClickMenuQQ(QWIN$TILE)                 !程序执行Windows菜单中的Tile命令，平铺窗口
  status = FocusQQ(TUnit)

  Do While ( input > 0 .and. input < 4 )
    Write(TUnit,*) '1:Plot f(x)=sin(x)'
    Write(TUnit,*) '2:Plot f(x)=cos(x)'
    Write(TUnit,*) '3:Plot f(x)=(x+2)*(x-2)'
    Write(TUnit,*) 'Other to Exit'
    Read(Tunit,*) input
    status = SetActiveQQ(GUnit)                   !使绘图窗口成为活动窗口
    Select Case(input)
      Case (1)
        Call Draw_Sub(f1)
      Case (2)
        Call Draw_Sub(f2)
      Case (3)
        Call Draw_Sub(f3)
    End Select
  End Do
End Program Ex_15
```

其中，DisplayCursor 函数用来控制子窗口是否显示光标，ClickMenuQQ 函数用来模拟鼠标选取 QuickWin 框架窗口缺省菜单项。

主程序先打开两个子窗口，将菜单命令子窗口的光标显示打开，绘图子窗口的光标显示关闭，接着平铺子窗口，并将菜单命令子窗口设为焦点窗口、绘图子窗口设为活动子窗口；使用循环结构设置文本窗口内的命令菜单，根据用户的输入，调用相应的例程绘图。

程序中有两点值得注意：一是使用 Open 命令打开自窗口后，须对子窗口进行相关操作才能显示，如清屏、展示光标、输入/输出等；二是绘图前，须激活图形输出子窗口。

*(2)绘制图形*

主程序引用的 PlotMod 模块，包含了绘图相关的数据和例程。

绘图模块 PlotMod 的实现代码为：

```
Module PlotMod
  Use IFQWin
  Implicit None
  Integer::lines = 500          !用多少线段来画函数曲线
  Real(8)::X_Start = -5.0       !X轴最小范围
  Real(8)::X_End   =  15.0      !X轴最大范围
  Real(8)::Y_Top   =  5.0       !Y轴最大范围
  Real(8)::Y_Bottom= -15.0      !Y轴最小范围
  Logical(2)::fInvert = .True.
  Contains
    Subroutine Draw_Sub(func)
      Real(8),External::func    !待绘图的函数
      Integer(4)::status        !绘图函数的运行状态
      Integer(4)::color         !颜色值
      Real(8)::step             !循环增量
      Real(8)::x,y,newX,newY
      Type(wxycoord)::wt        !指上一次的窗口坐标位置
      Call ClearScreen($GCLEARSCREEN)     !清屏
      !设定窗口坐标系，第一个参数逻辑真表示Y轴正向朝上
      status = SetWindow(fInvert,X_Start,Y_Top,X_End,Y_Bottom)
      color = RGBToInteger(255,0,0)         !将RGB转换成字节整数
      status = SetColorRGB(color)           !设定画笔颜色为红色
      Call MoveTo_W(X_Start,0.0_8,wt)       !画X轴
      status = LineTo_W(X_End,0.0_8)
      Call MoveTo_W(0.0_8,Y_Top,wt)         !画Y轴
      status = LineTo_W(0.0_8,Y_Bottom)

      status = SetColorRGB(RGBToInteger(0,255,0))       !设定画笔为蓝色
      step = ( X_End - X_Start ) / lines
      Do x=X_Start,X_End-step,step
        y = func(x)
        newX = x + step
        newY = func(newX)
        Call MoveTo_W(x,y,wt)
        status = LineTo_W(newX,newY)
      End Do
    End Subroutine Draw_Sub

    !所要绘图的函数
    Real(8) Function f1(x)
      Real(8),Intent(In)::x
      f1 = sin ( x )
```

```
    End Function f1
    Real(8) Function f2(x)
      Real(8),Intent(In)::x
      f2 = cos ( x )
    End Function f2
    Real(8) Function f3(x)
      Real(8),Intent(In)::x
      f3 =  ( x + 2 ) * ( x - 2 )
    End Function f3
End Module PlotMod
```

其中，绘图子程序 Draw_Sub 的参数为一函数，定义时声明该参数的数据类型(代表函数的返回结果)，调用时将具体的函数名作为实参传入。

在绘图操作前，通常要进行清屏，然后根据绘图数据范围设置合适的窗口坐标系。

## 小　　结

■ 在 Intel Fortan 中，QuickWin 既可在后台运行计算程序，又可同时在前台(子窗口)输出图形和文本，因此它是最常用的工程类型。

■ 在 QuickWin 程序中，使用带 File="User"命名参数的 Open 语句来创建子窗口，通过随后进行的 I/O 操作或图形输出等使子窗口显示在屏幕上。QuickWin 允许打开多达 40 个子窗口。

■ 当使用 Open 语句打开子窗口时，该子窗口即被激活成为活动窗口；某个时刻只能有一个窗口被放置到屏幕最顶层，即拥有焦点。要进行图形输出的子窗口必须是活动的，但可以没有焦点。

■ 文本分为字体文本和字符文本两种：字体文本按图形对待，并通过调用 OutGText 例程来输出；字符文本则按字符队来，并通过 OutText、Write 和 Print 语句来输出。

■ 输出字体文本时，须先调用 InitializeFonts 例程初始化字体，再调用 SetFont 例程从初始化的字体中选取字体，最后调用 OutGText 例程输出特定的字体文本。

■ 在输出图形和文本前，通常利用 windowsConfig 派生类型定义窗口，随后调用 SetWindowConfig 例程设置窗口属性。

■ 不论是输出图形(包括字体文本)还是字符文本，都是在子窗口的视口中进行；只不过，输出图形采用的是像素物理坐标(整数)或窗口坐标(实数)，输出字符文本采用的是行、列坐标。

■ 实际应用中，多采用实数窗口坐标绘图。通过设置不同的窗口坐标系，同样的数据集可以产生不同的图形效果。另外，窗口坐标系独立于显示卡硬件设备。

# 第 4 章　QuickWin 界面定制

在实际应用程序开发中，界面友好是一基本要求。QuickWin 程序的界面元素包括框架窗口的菜单栏和状态栏、框架窗口及其子程序的标题栏和图标、程序运行时出现的提示文字以及在子窗口客户区的鼠标输入。

QuickWin 通过提供运行库 IFQWin,对程序设计人员定制大部分界面元素给与支持。

**本章主要内容：**

- 菜单操作
- 本地化
- 信息提示
- 图标和标题
- 鼠标输入

## 4.1 菜单操作

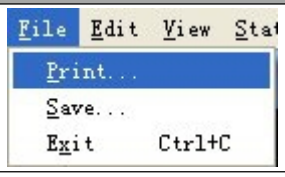
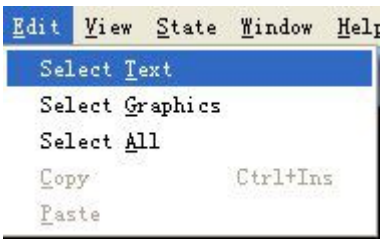
### 4.1.1　设置初始菜单

QuickWin 框架窗口缺省的初始菜单栏包含 6 个菜单：File、Edit、View、State、Window 和 Help,每个菜单包含的菜单项如表 4-1 所示。

QuickWin 程序在创建、展示框架窗口前，需要进行初始化。在初始化过程中，QuickWin 会在当前工程中搜索并调用 InitialSetting 函数，若找不到，就执行其缺省的 InitialSettings 函数，设置如表 4-1 所示的初始菜单。

设计人员若要改变 QuickWin 框架窗口初始菜单或框架窗口的位置及大小，可以提供自己的 InitialSettings 函数，在该函数中进行相应的设置。

表 4-1 QuickWin 框架窗口缺省菜单

| 菜单 | 菜单项 |
|---|---|
| File |  |
| Edit |  |
| View |  |

| | |
|---|---|
| State |  |
| Window |  |
| Help |  |

【例 4-1】将 QuickWin 框架窗口的初始大小设为 400×400，并设置由“Games”和“Help”两个菜单组成的菜单栏。在“Games”菜单下添加“TicTacToe”和“Exit”菜单项，分别执行打印和退出操作；在“Help”菜单下添加“QuickWin Help”菜单项，展示“QuickWin Graphics Application Help”窗口。程序运行结果见图 4-1。




图 4-1 例 4-1 程序运行界面

例 4-1 程序实现代码为：

```
Program Ex_1
  !Simple main program.
  Print *, "Control QuickWin's Window"
End Program Ex_1

Logical(4) Function InitialSettings()
  Use IFQWin
  Implicit None
  Logical(4) result
```

```
  Type(QWInfo) qwi
  Integer i
  !Set window frame size.
  qwi%x = 0
  qwi%y = 0
  qwi%w = 400
  qwi%h = 400
  qwi%type = QWIN$SET
  i = SetWSizeQQ(QWIN$FRAMEWINDOW,qwi)
  !Create first menu called Games.
  result = AppendMenuQQ(1,$MENUENABLED,'&Game'C,NUL)
  !Add item called TicTacToe.
  result = AppendMenuQQ(1,$MENUENABLED,'&TicTacToe'C,WINPRINT)
  !Draw a seperator bar.
  result = AppendMenuQQ(1,$MENUSEPARATOR,''C,NUL)
  !Add item called Exit
  result = AppendMenuQQ(1,$MENUENABLED,'&Exit'C,WINEXIT)
  !Add second menu called Help.
  result = AppendMenuQQ(2,$MENUENABLED,'&Help'C,NUL)
  result = AppendMenuQQ(2,$MENUENABLED,'&QuickWin Help'C,WININDEX)
  InitialSettings = .True.
End Function InitialSettings
```

其中，AppendMenuQQ 函数用来在菜单后面追加菜单项，其具体的使用方法见下一节。

InitialSettings 函数是在创建框架窗口之前的初始化过程中被调用，因此不能在该函数中执行框架窗口被创建、展示之后的动作，如打开子窗口、设置子窗口焦点等。QuickWin 缺省的 InitialSettings 函数返回逻辑真(.True.)，表示成功进行了初始化。在定制该函数时，结果也应返回逻辑真；若返回逻辑假(.False.)，尽管不影响程序功能，但 QuickWin 会弹出错误信息框，提示初始化菜单时产生错误。

## 4.1.2 删除、插入和追加菜单。

要设置 QuickWin 框架窗口的初始化菜单，须在 InitialSettings 函数内完成。若用户没有提供自己的 InitialSettings 函数，QuickWin 程序运行时展示其缺省菜单。

在 QuickWin 框架窗口创建、展示后，仍然可以在现有菜单上删除、插入和追加菜单。不过，在操作前首先要清楚菜单及菜单项的编号次序。QuickWin 菜单编号从左往右，起始编号为 1，依次为 1、2、3、...；菜单项编号从上往下，其实编号为 0，依次为 0、1、2、3、...。例如，File 菜单的菜单号为 1，其菜单项 Print、Save 和 Exit 的编号依次为 1、2、3，顶级 File 菜单项号为 0。

### 4.1.2.1 删除菜单项

要删除一个菜单或菜单项，调用 DeleteMenuQQ 函数，并规定删除菜单项的菜单编号和菜单项编号。DeleteMenuQQ 函数的语法格式为：

```
result = DeleteMenuQQ(menuID,itemID)
```

result 代表 Logical(4)的函数返回结果。函数调用成功，返回.True.；否则，返回.False.。

menuID 为 Logical(4)参数，标识包含删除菜单项的菜单号，最左边菜单编号为 1。

itemID 为 Logical(4)参数，标识要删除的菜单项号，顶级菜单项号为 0。

【例 4-2】删除 QuickWin 缺省菜单 File 中的 Save 菜单项和 Windows 菜单。程序运行结果见图 4-2。

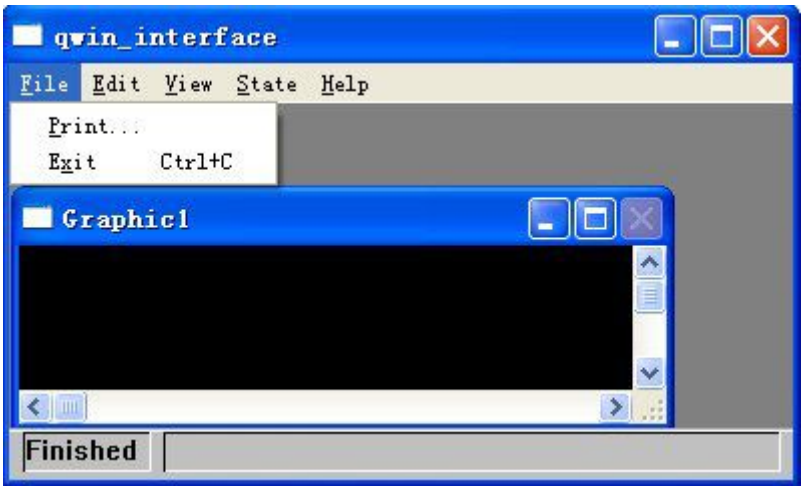



图 4-2 例 4-2 程序运行界面

例 4-2 程序实现代码为：

```
Program Ex_2
  Use IFQWin
  Implicit None
  Logical status
  status = DeleteMenuQQ(1,2)     !Delete the second menu item from menu 1 (the default File menu).
  status = DeleteMenuQQ(5,0)     !Delete the menu 5 (the default Window menu)
  Print *
End Program Ex_2
```

#### 4.1.2.2 插入菜单项

要在现有菜单中插入菜单项，调用 InsertMenuQQ 函数，并规定菜单号、菜单项号，注册其回调例程。InsertMenuQQ 函数的语法格式为：

result = InsertMenuQQ(menuID,itemID,flag,text,routine)

result 代表 Logical(4)的函数返回结果。函数调用成功，返回.True.；否则，返回.False.。

menuID 为 Integer(4)参数，标识包含插入菜单项的菜单号。

itemID 为 Integer(4)参数，标识要插入的菜单项号。

flag 为 Integer(4)参数，标识菜单项状态。QuickWin 提供的菜单项状态常数列于表 4-2。

表 4-4 QuickWin 菜单项状态常数

| 状态常数 | 描述 |
|---|---|
| $MENUGRAYED | 使菜单无效并灰显 |
| $MENUDISABLED | 使菜单无效但不灰显 |
| $MENUENABLED | 使菜单有效 |
| $MENUSEPARATOR | 菜单项为分隔条 |

| | |
|---|---|
| $MENUCHECKED | 在菜单项旁边添加一核取标志(√) |
| $MENUUNCHECKED | 清除菜单项旁边的核取标志 |

text 为 Character*(*)参数，规定菜单项名称。该参数须是 C 语言类型的字符串，即以空格结尾的字符串，在 Intel Fortra 中形如"WORDS OF TEXT"C，即在字符串尾部添加字符 C。对于字符串变量，可使用形如 Trim(StringVar)//''C 的形式。

routine 为外部例程名，代表选择菜单项时执行的回调例程。QuickWin 预定义的菜单回调例程列于表 4-3。

表 4-3 QuickWin 预定义的菜单回调例程

| 回调例程 | 描述 |
|---|---|
| WINPRINT | 打印 |
| WINSAVE | 保存 |
| WINEXIT | 退出 |
| WINSELECTTEXT | 从当前窗口选取文本 |
| WINSELECTGRAPHICS | 从当前窗口选取图形 |
| WINSELECTALL | 选取当前窗口的所有内容 |
| WININPUT | 使接收输入的子窗口获得焦点(成为当前窗口) |
| WINCOPY | 将选取的文本或图形复制到剪贴板 |
| WINPASTE | 在执行 Read 语句时，粘贴剪切板文本到当前活动的文本窗口 |
| WINCLEARPASTE | 清空粘贴缓冲区 |
| WINSIZETOFIT | 调整输出以布满整个窗口 |
| WINFULLSCREEN | 全屏展示输出 |
| WINSTATE | 在文本输出的暂停与恢复状态之间切换 |
| WINCASCADE | 层叠活动子窗口 |
| WINTILE | 平铺活动子窗口 |
| WINARRANGE | 排列活动子窗口图标 |
| WINSTATUS | 展示/隐藏状态栏 |
| WININDEX | 展示 QuickWin 帮助索引 |
| WINUSING | 展示使用 QuickWin 帮助的提示 |
| WINABOUT | 展示当前 QuickWin 应用程序信息 |
| NUL | 无回调例程 |

插入菜单项，须在现有的菜单项之间或紧接最后的菜单项插入。若要插入菜单栏中的菜单项(即菜单本身)，须规定菜单项编号为 0。假如要插入菜单项的菜单位置被占用，在这个位置上的已有菜单及其后面的菜单依次右依一个位置。

【例 4-3】在 QuickWin 的 Window 菜单位置(菜单号为 5)插入一个新的菜单。




图 4-3 例 4-3 程序运行界面

例 4-3 程序实现代码为：

```
Program Ex_3
  Use IFQWin
  Implicit None
  Logical(4) status
  External MyProc
  status = InsertMenuQQ(5,0,$MENUENABLED,'New Menu'C,NUL)
  status = InsertMenuQQ(5,1,$MENUENABLED,'New MenuItem'C,MyProc)
  Do While (.True.)
  End Do
End Program Ex_3

Subroutine MyProc(item_checked)
  Implicit None
  Logical,Intent(In)::item_checked
  Print *, "'NewMenuItem' is clicked"
End Subroutine MyProc
```

其中，MyProc 为自定义的回调例程，须在插入菜单项的程序单元中声明为外部例程，即添加 External 声明语句。

回调例程作为外部子程序来实现，在 Intel Visual Fortran9.0 版本下回调例程带一逻辑参数，实际使用中该参数可以省略。一般情况下，应给每一个菜单项规定一个不同的回调例程，否则有可能导致菜单状态异常。

值得注意的是，在设计 QuickWin 菜单应用程序时，通常须在主程序末尾添加一个空的无限循环，否则会导致菜单出现异常行为。比如，提供了自定义回调例程的菜单项被灰显，用户无法选取该菜单项。

#### 4.1.2.3 追加菜单项

要在现有菜单尾部追加菜单项，调用 AppendMenuQQ 函数，并规定菜单号、注册其回调例程。AppendMenuQQ 函数的语法格式为：

result = AppendMenuQQ(menuID,flags,text,routine)

result 代表 Logical(4)的函数返回结果。函数调用成功，返回.True.；否则，返

回.False.。

menuID 为 Integer(4)参数，标识包含插入菜单项的菜单号。

flags 为 Integer(4)参数，标识菜单项状态。如表 4-2 所示。

text 为 Character*(*)参数，规定菜单项名称。

routine 为外部例程名，代表选择菜单项时执行的回调例程。QuickWin 预定义的菜单回调例程如表 4-3 所示。

因为追加菜单项总是在某一菜单列表的下面或最右边菜单的后面追加，所以无须规定菜单项号，与 InsertMenuQQ 函数相比，它少了一个菜单项编号参数；在使用上，同样要求 menuID 参数取值在现有菜单最大编号+1 的范围内。

【例 4-4】在 QuickWin 缺省菜单的最右边追加新的菜单和菜单项。程序运行结果见图 4-4。




图 4-4 例 4-4 程序运行界面

例 4-4 程序实现代码为：

```
Program Ex_4
  Use IFQWin
  Implicit None
  Logical(4) status
  External MyProc
  status = AppendMenuQQ(7,$MENUENABLED,'New &Menu'C,NUL)
  status = AppendMenuQQ(7,$MENUENABLED,'New MenuI&tem'C,MyProc)
  Do While (.True.)
  End Do
End Program Ex_4

Subroutine MyProc(item_checked)
  Implicit None
  Logical,Intent(In)::item_checked
  Print *, "'NewMenuItem' is clicked"
End Subroutine MyProc
```

因为菜单号 7 代表 Help 菜单后的新菜单，所以首次追加的菜单项被作为顶级菜单项(即菜单栏上的菜单)，其回调例程规定为空(NUL)。在这种情况下，即便规定了顶级菜单项的回调例程，其回调例程也将被忽略。

在给出的菜单项标题时，可以通过在字母前添加“和”号(&)来设置快捷键，程序运行时，&后面的字母出现下划线(图 4-4)；若没有在菜单项标题字符串中添加“和”号,则 QuickWin 自动将首字母设为快捷键(图 4-3)。当同时按下 Alt + 下划线字母时，会选取相应的菜单项，如同在菜单项上点击了鼠标。对于中文的菜单项，可使用“文件(&F)”这样的形式。

与 AppendMenuQQ 相比，InsertMenuQQ 函数的使用更为灵活。InsertMenuQQ 既可以在菜单列表尾部，又可在菜单列表的任何位置插入菜单项。

### 4.1.3 修改菜单

除了上述在现有菜单上进行删除、插入和追加菜单项外，QuickWin 还允许对现有的菜单项进行修改，包括修改菜单项的标题、回调例程和状态，分别使用 ModifyMenuStringQQ、ModifyMenuRoutineQQ 和 ModifyMenuFlagsQQ 函数。3 个函数的语法格式分别为：

result = ModifyMenuStringQQ(menuID,itemID,text)

result = ModifyMenuRoutineQQ(menuID,itemID,routine)

result = ModifyMenuFlagsQQ(menuID,itemID,flag)

result 代表 Logical(4)的函数返回结果。函数调用成功，返回.True.；否则，返回.False.。

menuID 为 Integer(4)参数，标识包含插入菜单项的菜单号。

itemID 为 Integer(4)参数，标识要修改的菜单项号。

text 为 Character*(*)参数，规定菜单项名称，该参数须是 C 语言类型的字符串。

flag(s)为 Integer(4)参数，标识菜单项状态。如表 4-2 所示。

routine 为外部例程名，代表选择菜单项时执行的回调例程。QuickWin 预定义的菜单回调例程如表 4-3 所示。

【例 4-5】在 QuickWin 缺省菜单 File 列表后追加一个菜单项，并修改该菜单项的标题、回调例程和状态。程序运行结果见图 4-5。

例 4-5 程序实现代码为：

```
Program Ex_5
  Use IFQWin
  Implicit None
  Logical(4) status
  Character(20) str
  str='&Add to File Menu'C
  status = AppendMenuQQ(1,$MENUENABLED,str,WINSTATUS)
  status = ModifyMenuStringQQ(1,4,'Tile Windows'C)
  status = ModifyMenuRoutineQQ(1,4,WINTILE)
  status = ModifyMenuFlagsQQ(1,4,$MENUCHECKED)
  Print *, 'Modify appended iten to the bottom of File menu'
  Open(10,File='User')
  Write(10,*) 'Open another child Window'
End Program Ex_5
```

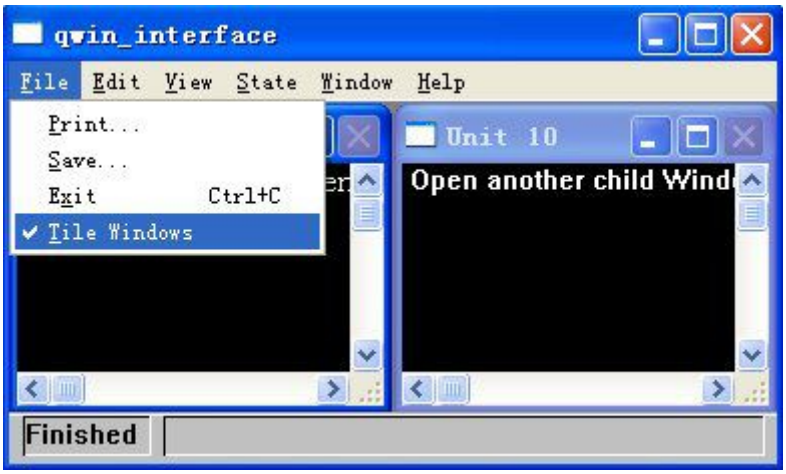



图 4-5 例 4-5 程序运行界面

### 4.1.4 调整子窗口列表的菜单位置

默认情况下，QuickWin 将打开的子窗口列表放置在 Window 缺省菜单下。通过调用 SetWindowMenuQQ 函数，可以将子窗口列表调整到任何菜单下。SetWindowMenuQQ 函数的语法格式为：

result = SetWindowMenuQQ(menuID)

result 代表 Logical(4)的函数返回结果。函数调用成功，返回.True.；否则，返回.False.。

menuID 为 Integer(4)参数，标识包含子窗口列表的菜单号。

【例 4-6】将 QuickWin 子窗口列表由 Window 菜单下调整到 File 菜单下。程序运行结果如图 4-6 所示。

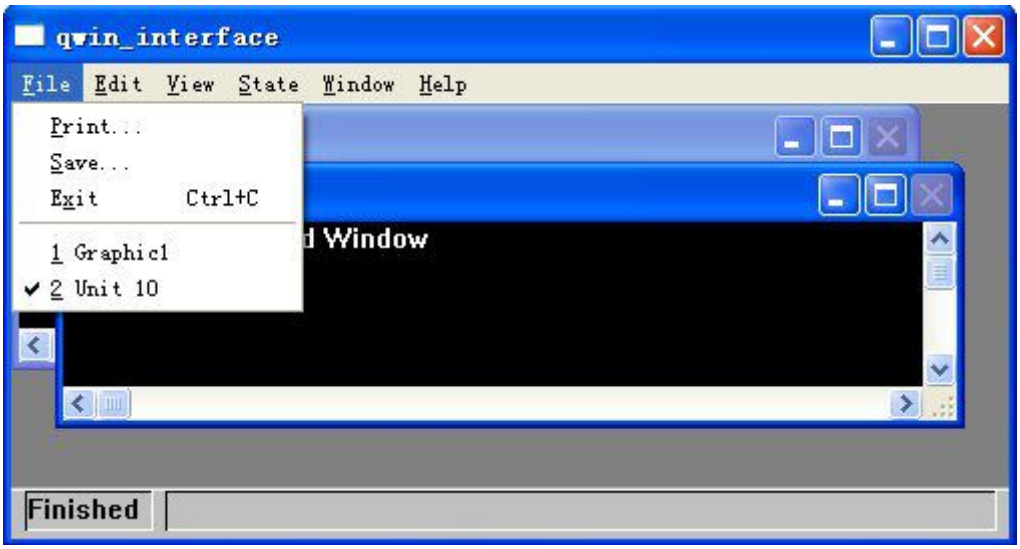



图 4-6 例 4-6 程序运行界面

例 4-6 程序实现代码为：

```
Program Ex_6
  Use IFQWin
  Implicit None
  Logical(4) status
```

```
  status = SetWindowMenuQQ(1)
  Print *, 'Append list of open child windows to File menu'
  Open(10,File='User')
  Write(10,*) 'Open another child Window'
End Program Ex_6
```

### 4.1.5 模拟鼠标选取菜单项的操作

要执行某菜单项命令，通常使用鼠标选取该菜单项。QuickWin 通过提供 ClickMenuQQ 函数，来模拟鼠标选取菜单项的操作。ClickMenuQQ 函数的语法格式为：

result = ClickMenuQQ(item)

result 代表 Integer(4)的函数返回结果。函数调用成功，返回 0；否则，返回非 0。

item 为 Integer(4)参数，代表被选取菜单项执行的回调例程。这里的回调例程必须是 QuickWin 预定义的回调例程，可以采取表 4-3 所列例程的地址形式，也可以采用表 4-4 所列例程符号常量形式。

表 4-4 QuickWin 预定义的菜单回调例程符号常量

| 回调例程 | 描述 |
|---|---|
| QWIN$STATUS | 展示/隐藏状态栏(WINSTATUS) |
| QWIN$TILE | 平铺活动子窗口(WINTILE) |
| QWIN$CASCADE | 层叠活动子窗口(WINCASCADE) |
| QWIN$ARRANGE | 排列活动子窗口图标(WINARRANGE) |

【例 4-7】QuickWin 程序启动后，平铺活动子窗口，并展示“QuickWin Graphics Application Help”窗口(图 4-7)。




图 4-7 QuickWin Graphics Application Help 窗口

例 4-7 程序实现代码为：

```
Program Ex_7
  Use IFQWin
  Implicit None
  Logical(4) status
  Print *, 'Open default Graphic1 Window'
  Open(10,File='User')
  Write(10,*) 'Open another child Window'
  status = ClickMenuQQ(QWIN$TILE)
  status = ClickMenuQQ(LOC(WININDEX))
End Program Ex_7
```

程序中，平铺活动子窗口，传入 ClickMenuQQ 函数的参数为 QuickWin 预定义的菜单回调例程符号常量；而展示“QuickWin Graphics Application Help”窗口时，传入的参数为回调例程的地址。

## 4.2 本地化

所谓本地化或国际化，是指程序界面文字用用户所在国家的语言来表达。

若要对 QuickWin 框架窗口上的菜单进行本地化处理，可直接在 InsertMenuQQ、AppendMenuQQ 和 ModifyMenuStringQQ 函数中使用本地语言的菜单标题；若要对 QuickWin 产生的其它字符信息(如状态栏信息、对话框提示信息等)本地化，可通过 SetMessageQQ 例程进行设置。该例程使用的是规则 Fortran 字符串，而不是以空格结尾的 C 语言类型字符串。SetMessageQQ 例程的语法格式为：

Call SetMessageQQ(msg,id)

msg 为 Character*(*)参数，代表要展示的字符信息。

id 为 Integer(4)参数，指要改变的字符信息号。表 4-5 列出了 QuickWin 程序中可能出现的信息字符串及其信息号。

表 4-5 QuickWin 程序中可能出现的信息字符串及其信息号

| 信息号 | 信息字符串 |
|---|---|
| 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" |

例如，若系统没有安装打印机，用户执行 File 菜单中的 Print 命令，QuickWin 会弹出如图 4-8(a)所示的信息提示框。

【例 4-8】将图 4-8(a)的提示字符由“No Default Printer.”(对应的信息号为 QWIN$MSG_PRINTNODEFAULT)改为“没有安装缺省打印机”。程序运行结果如图 4-8(b)所示。




图 4-8 QuickWin 中的信息提示框

例 4-8 程序实现代码为：

```
Program Ex_8
  Use IFQWin
  Implicit None
  Print *, "Hello"
  Call SetMessageQQ('没有安装缺省打印机',QWIN$MSG_PRINTNODEFAULT)
End Program Ex_8
```

## 4.3 信息提示

### 4.3.1 使用信息对话框

在程序执行过程中,有时需要由用户决定程序执行的特殊操作。在这种情况下，可以使用消息对话框。在 QuickWin 中，通过调用 MessageBoxQQ 函数来展示信息对话框。在该函数中，可以规定要展示的文字、对话框标题、按钮和图标。通过检查函数返回值，可知用户点击了哪一个按钮，并据此进行相应的处理。

MessageBoxQQ 函数的语法格式为：

result = MessageBoxQQ(msg,caption,mType)

msg 为 Character*(*)参数，代表对话框的提示文字，该参数须是以空格结尾的 C 语言类型字符串。

caption 参数和 msg 参数的数据类型相同，代表对话框的标题字符串。

mType 为 Integer(4)参数，是 QuickWin 中预定义的符号常量，规定了对话框要展示的图标、按钮和对话框类型，它可以使表 4-6 所列的不同逻辑组合。

result 代表 Integer(4)的函数返回结果，假如内存不足，函数返回 0；否则，函数返回表 4-7 中的常量。

表 4-6 QuickWin 信息对话框中的图标、按钮符号常量

| 图标、按钮常量 | 描述 |
|---|---|
| MB$ABORTRETRYIGNORE | 中断、重试和忽略按钮 |
| MB$DEFBUTTON1 | 第一个按钮为默认按钮 |
| MB$DEFBUTTON2 | 第二个按钮为默认按钮 |
| MB$DEFBUTTON3 | 第三个按钮为默认按钮 |
| MB$ICONASTERISK,<br>MB$ICONINFORMATION | 信息图标 |
| MB$ICONEXCLAMATION | 感叹号图标 |
| MB$ICONHAND,<br>MB$ICONSTOP | 停止标记图标 |
| MB$ICONQUESTION | 问号图标 |
| MB$OK | 确定按钮 |
| MB$OKCANCEL | 确定和取消按钮 |
| MB$RETRYCANCEL | 重试和取消按钮 |
| MB$SYSTEMMODAL | 模式对话框 |
| MB$YESNO | 是和否按钮 |
| MB$YESNOCANCEL | 是、否和取消按钮 |

表 4-7 MessageBoxQQ 函数返回结果

| 返回结果 | 描述 | 返回结果 | 描述 |
|---|---|---|---|
| MB$IDABORT | 用户点击了中断按钮 | MB$IDOK | 用户点击了确定按钮 |
| MB$IDCANCEL | 用户点击了取消按钮 | MB$IDRETRY | 用户点击了重试按钮 |
| MB$IDIGNORE | 用户点击了忽略按钮 | MB$IDYES | 用户点击了是按钮 |
| MB$IDNO | 用户点击了否按钮 | | |

【例 4-9】提供如图 4-9 所示的信息对话框，包括问号图标和 Yes/No 按钮，将 Yes 按钮设为缺省按钮，并判断用户选取了哪一个按钮。




图 4-9 YES/NO 信息对话框

例 4-9 程序实现代码为：

```
Program Ex_9
  Use IFQWin
  Implicit None
  Integer(4) message
  Character(30) str
  message = MessageBoxQQ('Do you want to continue?'C,'Matrix'C,&
            MB$ICONQUESTION.Or.MB$YESNO.Or.MB$DEFBUTTON1)
  Select Case ( message )
    Case ( MB$IDYES )
      str = "YES button is selected"
    Case ( MB$IDNO )
      str = "NO button is selected"
  End Select
  Print *, str
End Program Ex_9
```

### 4.3.2 定制 About 对话框

若在 QuickWin 缺省的 Help 菜单下执行 About 菜单命令，会弹出 About 信息对话框，展示 QuickWin 缺省的版本信息(图 4-10(a))。用户可以通过 AboutBoxQQ 函数，对 About 对话框展示的信息字符串进行定制。AboutBoxQQ 函数的语法格式为：

result = AboutBoxQQ(cString)

cString 为 Character*(*)参数，代表自定义的信息字符串，该参数须是 C 语言字符串。

result 代表 Integer(4)的函数返回结果，函数调用成功，返回 0；否则，返回非 0。

【例 4-10】对 QuickWin 中的 About 对话框进行定制，替换其缺省字符信息(图 4-10(a))。程序运行结果见图 4-10(b)。

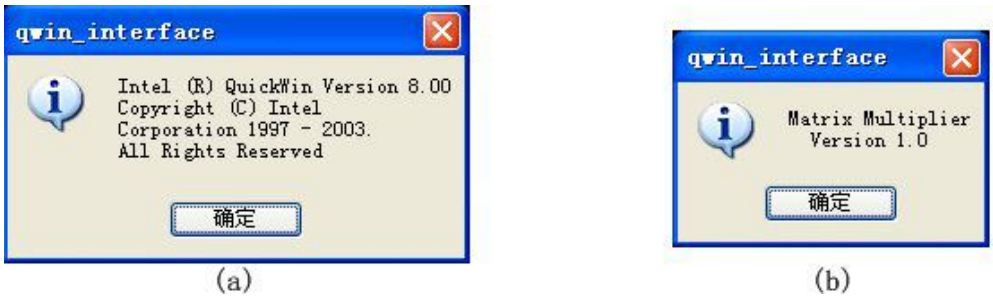



图 4-10 QuickWin 程序中的 About 对话框

例 4-10 程序实现代码为：

```
Program Ex_10
  Use IFQWin
  Implicit None
  Integer(4) dummy
  !Make sure the entire default menu to be displayed
  Print *
  !Set the About Box Message
  dummy = AboutBoxQQ('Matrix Multiplier\r  Version 1.0'C)
End Program Ex_10
```

## 4.4 图标和标题

### 4.4.1 定制图标

在 QuickWin 程序中，可以方便地对框架窗口、子窗口的图标进行定制。图标作为资源保存在资源脚本文件(*.RC)中，资源编辑器负责将图标等资源编译成目标文件，并和其它目标文件链接成可执行文件。

要定制 QuickWin 窗口图标，需要向工程中“添加新项”，插入 Resource 类型的文件(*.RC)，并规定与工程名不同的资源名；然后从“编辑”菜单中选取“添加资源”，在随后出现的“添加资源”对话框中选择 Icon(图 4-11)。若要自己绘制图表，点击“新建”按钮，在随后出现的图标编辑器中进行绘制；若要使用现有的图标，点击“导入”按钮，选取相应的图标文件。




图 4-11 “添加资源”对话框

接下来在 IDE 解决方案资源管理器中双击鼠标，打开资源管理器窗口，设置图标的 ID 号。QuickWin 将框架窗口的图标 ID 规定为“FrameIcon”，子窗口的图标 ID 规定为“ChildIcon”，定制图标时必须按 QuickWin 的规定手工输入窗口的图标 ID 号，如图 4-12 所示。若要同时对框架窗口和子窗口的图标进行定制，须在同一资源文件中分别插入框架窗口和子窗口的图标资源，并规定各自的图标 ID 号(以字符串形式输入)。

图标设置完毕，即可生成拥有自定义图标的 QuickWin 应用程序。例如，图 4-13 是同时对框架窗口和自窗口的图标进行定制的示例程序运行结果。

### 4.4.2 定制标题

QuickWin 子窗口的标题可以在 Open 语句或调用的 SetWindowConfig 函数中进行设置，这里“定制标题”是指对框架窗口的标题进行设置。默认情况下，框架窗口标题为 EXE 文件的主文件名。

QuickWin 运行库 IFQWin 并没有提供设置框架窗口标题的例程，需要引用 User32 模块，并调用其中的 API 函数 SetWindowText，来对框架窗口标题进行设置。该函数的第一个参数为窗口句柄；第二个参数代表要设置的窗口标题，须是 C 语言字符串。

【例 4-11】调用 SetWindowsText 函数，分别设置 QuickWin 框架窗口和子窗口标题。程序运行结果如图 4-14 所示。

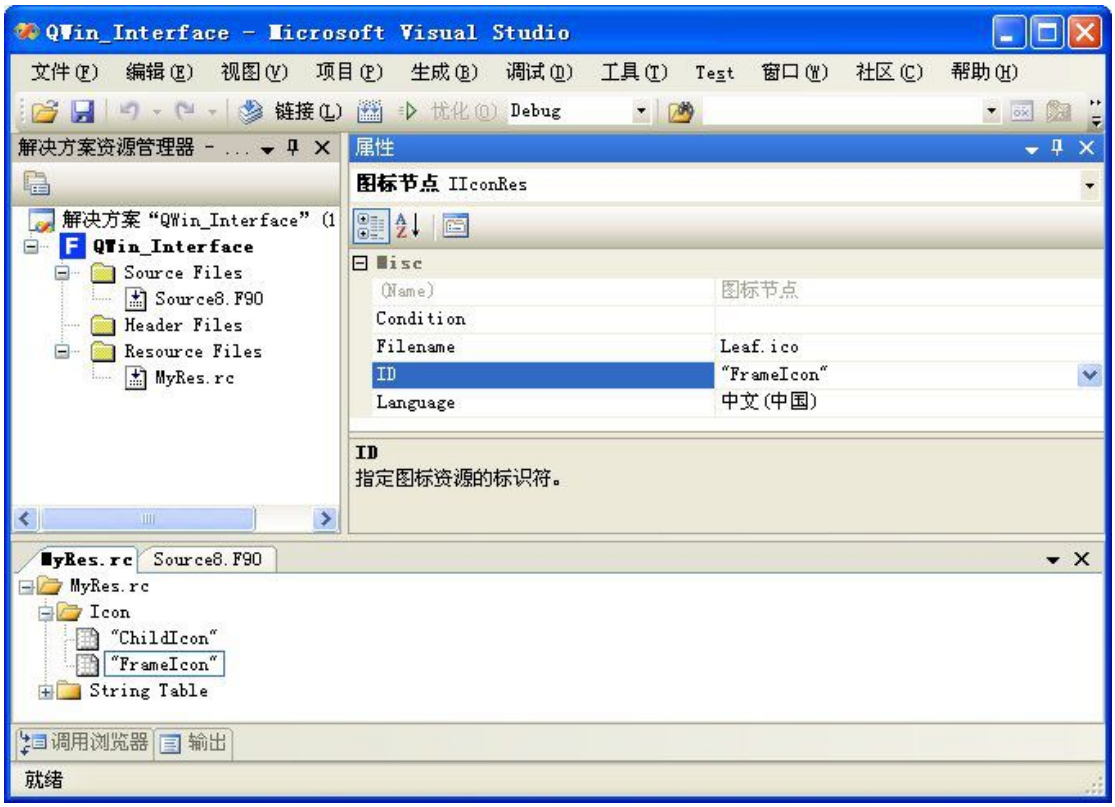



图 4-12 设置图标 ID 的属性窗口




图 4-13 定制了图标的框架窗口和子窗口

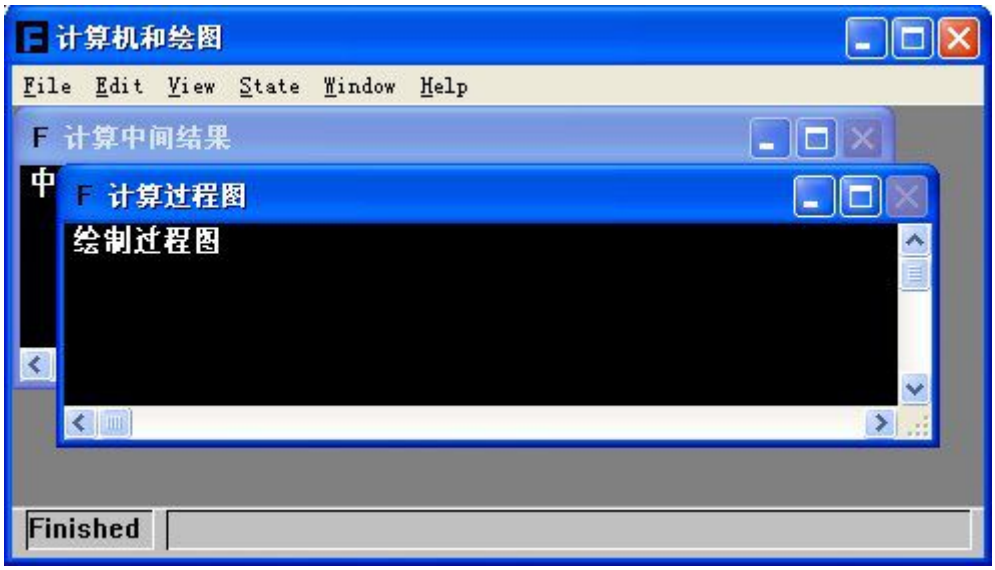



图 4-14 定制标题的框架窗口和自窗口

例 4-11 程序实现代码为：

```
Program Ex_11
  Use IFQWin
  Use User32
  Implicit None
  Integer(4) i4
  i4 = SetWindowText( GetHwndQQ(QWIN$FRAMEWINDOW) , "计算机和绘图"C )
  Open(10,File='User')
  Write(10,*) '中间结果输出'
  i4 = SetWindowText( GetHwndQQ(10) , "计算中间结果"C )
  Open(11,File='User',Title='计算过程图')
  Write(11,*) '绘制过程图'
End Program Ex_11
```

其中，IFQWin 库函数 GetHwndQQ 用来获取窗口句柄，当中的参数为子窗口单元号。

【例 4-12】在例 3-15 的基础上，对 QuickWin 框架窗口的图标、标题、菜单栏和状态栏，以及缺省子窗口(Graphic1)的图标和标题进行定制，并使用菜单交互绘图。程序运行结果如图 4-15 所示。

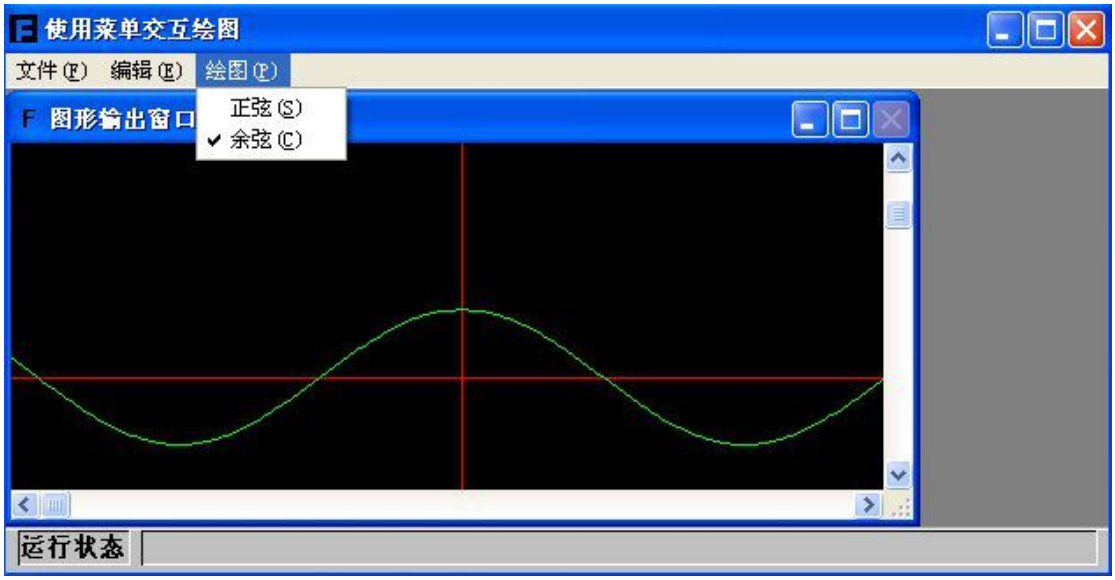



图 4-15 例 4-12 程序运行界面

(1)按 4.4.1 介绍的办法，添加框架窗口及子窗口的图标资源。

(2)编写菜单回调例程。例 3-15 使用字符命令菜单交互绘图程序，已将绘图相关的数据和例程封装在模块 PlotMod 中，这里只需要添加一个引用 PlotMod 模块的新模块(NewPlotMod)，并在该模块中放置“正弦”、“余弦”菜单回调例程(PlotSin 和 PlotCos)。

NewPlotMod 模块的实现代码为：

```
Module NewPlotMod
  Use IFQWin
  Use PlotMod
  Implicit None
  Contains
    Subroutine PlotSin()
```

```
    Integer(4)::i4
    !核取第三组菜单中的Sin菜单项
    i4 = ModifyMenuFlagsQQ(3,1,$MENUCHECKED)
    i4 = ModifyMenuFlagsQQ(3,2,$MENUUNCHECKED)
    Call Draw_Sub(f1)
  End Subroutine PlotSin
  Subroutine PlotCos()
    Integer(4)::i4
    i4 = ModifyMenuFlagsQQ(3,1,$MENUUNCHECKED)
    !核取第三组菜单中的Cos菜单项
    i4 = ModifyMenuFlagsQQ(3,2,$MENUCHECKED)
    Call Draw_Sub(f2)
  End Subroutine PlotCos
End Module NewPlotMod
```

在 PlotSin 和 PlotCos 回调例程中，分别核取相应的菜单项，并调用 PlotMod 模块中的例程绘图。

(3)在主程序中定制框架窗口及子窗口标题，设置状态栏显示文字。

主程序的实现代码为：

```
Program Main    !主程序
  Use IFQwin
  Use User32      !定义了SetWindowText函数
  Use NewPlotMod
  Implicit None
  Integer(4) i4
  Type(QWInfo) qw

  !设置框架窗口和子窗口标题
  i4 = SetWindowText( GetHwndQQ(QWIN$FRAMEWINDOW),&
                      "使用菜单交互绘图"C)
  i4 = SetWindowText( GetHwndQQ(0),&  !缺省子窗口的单元号
                      "图形输出窗口"C)
  !将状态栏的状态提示由"Running"改为"运行状态"
  Call SetMessageQQ('运行状态',QWIN$MSG_RUNNING)

  !极大化框架窗口和绘图子窗口
  qw.type = QWIN$MAX
  i4 = SetWSizeQQ(QWIN$FRAMEWINDOW,qw)
  i4 = SetWSizeQQ(0,qw)

  !使系统处于等待状态
  Do While(.True.)
  End Do
End Program Main
```

为了使定制的菜单正常运行，在主程序末尾设置了一个无限循环；否则，定制的菜单被灰显，用户无法选取菜单。

(4)在初始化函数中定制菜单。

InitialSettings 函数的实现代码为：

```
Logical(4) Function InitialSettings()
  Use IFQWin
  Use NewPlotMod
  Implicit None
  Logical(4)::i4
  !组织文件菜单
  i4 = AppendMenuQQ(1,$MENUENABLED,'文件(&F)'C,NUL)
  i4 = AppendMenuQQ(1,$MENUENABLED,'保存(&S)'C,WINSAVE)
  i4 = AppendMenuQQ(1,$MENUENABLED,'打印(&P)'C,WINPRINT)
  i4 = AppendMenuQQ(1,$MENUENABLED,'退出(&X)'C,WINEXIT)
  !组织编辑菜单
  i4 = AppendMenuQQ(2,$MENUENABLED,'编辑(&E)'C,NUL)
  i4 = AppendMenuQQ(2,$MENUENABLED,'选择文本(&T)'C,WINSELECTTEXT)
  i4 = AppendMenuQQ(2,$MENUENABLED,'选择图形(&G)'C,WINSELECTGRAPHICS)
  i4 = AppendMenuQQ(2,$MENUENABLED,'全选(&A)'C,WINSELECTALL)
  i4 = AppendMenuQQ(2,$MENUENABLED,'复制(&T)'C,WINCOPY)
  i4 = AppendMenuQQ(2,$MENUENABLED,'粘贴(&P)'C,WINPASTE)
  !组织绘图菜单
  i4 = AppendMenuQQ(3,$MENUENABLED,'绘图(&P)'C,NUL)
  i4 = AppendMenuQQ(3,$MENUENABLED,'正弦(&S)'C,PlotSin)
  i4 = AppendMenuQQ(3,$MENUENABLED,'余弦(&C)'C,PlotCos)
  InitialSettings = .True.
End Function InitialSettings
```

程序中，将“正弦”、“余弦”菜单项的回调例程分别设为 PlotSin 和 PlotCos。InitialSettings 函数末尾返回逻辑真，以表明成功进行了初始化。

这里，展示了利用 Fortran90/95 模块进行面向对象程序设计、复用程序资源的方法：将绘图的数据和例程封装在一个专门的模块中，新的应用程序若要对其功能进行扩展(如通过菜单绘图)，可添加引用已有模块的新模块来实现，新的应用程序只需引用新模块，就可以使用已有模块的基本功能及新增模块的扩展功能。

## 4.5 鼠标输入

在 QuickWin 应用程序中，当用鼠标点选菜单、点击窗口控制按钮或窗口中的控件时，系统自动捕获鼠标事件，并执行相应的命令。但在子窗口客户区产生的鼠标事情，系统并不自动捕获，需要编程来捕获，以代替键盘输入或对窗口内的文本、图形进行操作。

鼠标属于异步设备，在程序执行的任何时候用户都可点击鼠标。当产生鼠标事件时，Windows 系统向应用程序发送鼠标消息，应用程序接到消息后执行相应的动作，这是常用的

基于事件的鼠标支持方法。另一种鼠标支持方法是阻塞方法，即暂停程序执行、等待鼠标事件发生，这使得应用程序可以按特定的次序顺序执行。

QuickWin 对上述两种方法都给予支持：事件方法，当窗口内产生鼠标事件时，程序执行自定义的鼠标回调例程；阻塞方法，暂停程序执行，直到产生鼠标事件才继续往下执行。QuickWin 默认的是事件方法。

### 4.5.1 事件方法

该方法首先需要调用 RegisterMouseEvent 函数，注册在特定窗口、产生特定鼠标事件时程序执行的回调例程。RegisterMouseEvent 函数的语法格式为：

result = RegisterMouseEvent(unit,mouseEvents,callBackRoutine)

unit 为 Integer(4)参数，指产生鼠标事件的子窗口单元号。

mouseEvents 为 Integer(4)参数，标识特定的鼠标事件。QuickWin 预定义的鼠标事件符号常量列于表 4-8。

表 4-8 QuickWin 预定义的鼠标事件符号常量

| 鼠标事件 | 描述 |
|---|---|
| MOUSE$LBUTTONDOWN | 按下鼠标左键 |
| MOUSE$LBUTTONUP | 松开鼠标左键 |
| MOUSE$LBUTTONDBLCLK | 双击鼠标左键 |
| MOUSE$RBUTTONDOWN | 按下鼠标右键 |
| MOUSE$RBUTTONUP | 松开鼠标右键 |
| MOUSE$RBUTTONDBLCLK | 双击鼠标右键 |
| MOUSE$MOVE | 移动鼠标 |

callBackRoutine 代表产生鼠标事件时执行的回调例程，回调例程接口为:

```
Subroutine CallBackRoutine(unit,mouseeEvents,keyState,mouseXPos,mouseYPos)
  Integer unit
  Integer mouseEvents
  Integer keyState
  Integer mouseXPos
  Integer mouseYPos
End Subroutine CallBackRoutine
```

其中，unit 和 mouseEvents 参数与 RegisterMouseEvent 函数中的同名参数相同；mouseXPos 和 mouseYPos 参数标识事件发生时鼠标的 X、Y 位置坐标；keyState 参数标识 Shift、Ctrl 键的状态，可以是表 4-9 所列常量的不同逻辑组合。通常，需在调用程序中建立鼠标回调例程的接口块；否则，须显式声明鼠标回调例程为外部程序(External)。

表 4-9 Shift、Ctrl 键的状态常量

| 状态常量 | 描述 |
|---|---|
| MOUSE$KS_LBUTTON | 鼠标左键被按下 |
| MOUSE$KS_RBUTTON | 鼠标右键被按下 |
| MOUSE$KS_SHIFT | Shift 键同时被按下 |
| MOUSE$KS_CONTROL | Ctrl 键同时被按下 |

result 代表 Integer(4)的函数返回结果，函数调用成功，返回 0 或正整数；否则，返

回负整数 MOUSE$BADUNIT 或 MOUSE$BADEVENT。其中，MOUSE$BADUNIT 表示以 unit 标识的子窗口没有打开，或没有与 QuickWin 框架窗口相关联；MOUSE$BADEVENT 表示规定的鼠标事件不被支持。

注册过的鼠标回调例程，可以通过调用 UnRegisterMouseEvent 函数予以撤销。UnRegisterMouseEvent 函数的语法格式为：

result = UnRegisterMouseEvent(unit,mouseEvents)

当中的参数和返回结果表达的含义，与 RegisterMouseEvent 函数的相同。

【例 4-13】为单元号 4 的子窗口注册鼠标左键双击事件回调例程，当用户双击鼠标左键时，打开另一个子窗口，并在该子窗口内输出文本。程序的运行结果如图 4-16 所示。

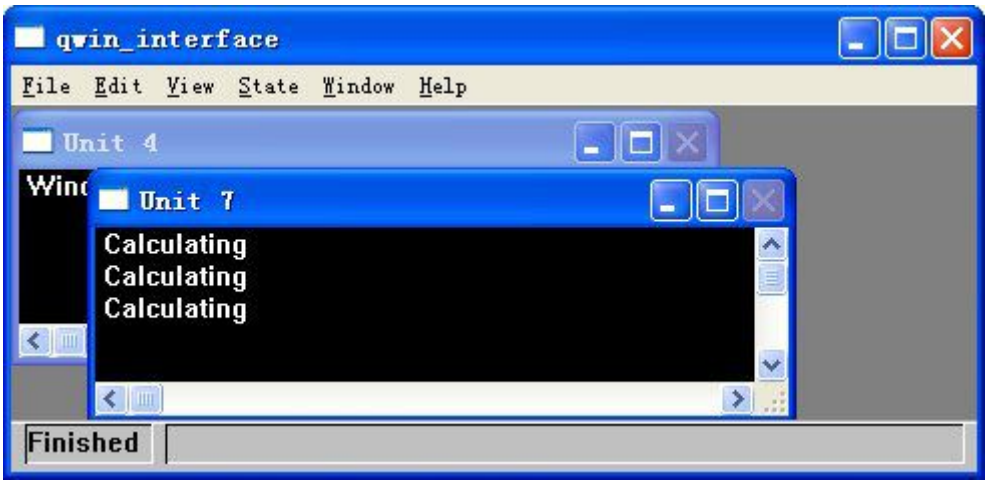



图 4-16 例 4-13 程序运行界面

例 4-13 程序实现代码为：

```
Program Ex_13
  Use IFQwin
  Implicit None
  Integer(4) status
  Interface
    Subroutine Calculate(unit,mouseEvent,keyState,mouseXPos,mouseYPos)
      Integer unit
      Integer mouseEvent
      Integer keyState
      Integer mouseXPos
      Integer mouseYPos
    End Subroutine Calculate
  End Interface
  Open(4,File='User')
  Write(4,*) "Window to receive a mouse event"
  status = RegisterMouseEvent(4,MOUSE$LBUTTONDBLCLK,Calculate)
  !status = UnRegisterMouseEvent(4,MOUSE$LBUTTONDBLCLK)
End Program Ex_13

Subroutine Calculate(unit,mouseEvent,keyState,mouseXPos,mouseYPos)
  Integer unit
```

```
  Integer mouseEvent
  Integer keyState
  Integer mouseXPos
  Integer mouseYPos
  Open(7,File='User')
  Write(7,*) "Calculating"
End Subroutine Calculate
```

### 4.5.2 阻塞方法

上述事件方法适合于非顺序处理流程，而阻塞方法适合于顺序处理流程。QuickWin 提供了 WaitOnMouseEvent 函数来阻塞程序运行，直到完成鼠标输入为止。WaitOnMouseEvent 函数的语法格式为：

result = WatiOnMouseEvent(mouseEvent,keyState,x,y)

当中的参数与鼠标回调例程的对应参数相同。假如函数调用成功，返回与鼠标事件有关的符号常量；否则，返回 MOUSE$BADEVENT 常量，表示规定的事件不被支持。

值得注意的是，一旦调用了 WaitOnMouseEvent 函数，程序执行即被阻塞，此刻在状态栏显示“Mouse input pending in X”,当中的 X 代表调用 WaitOnMouseEvent 函数时拥有焦点的子窗口名(图 4-17)。这就是说，WaitOnMouseEvent 函数参数中规定的事件，发生在函数调用时拥有焦点的子窗口，程序在该子窗口等待捕获发生的鼠标事件。

【例 4-14】打开 Graphic1 和单元号为 4 的两个子窗口。在单元号为 4 的子窗口内采用阻塞方法，捕获鼠标左键或右键按下事件，并判断是否按下了 Shift 键或 Ctrl 键。程序运行结果见图 4-17。




图 4-17 例 4-14 程序运行界面

例 4-17 程序实现代码为：

```
Program Ex_14
  Use IFQwin
  Implicit None
  Integer(4) mouseEvent,keyState,x,y,status
  Print *, "Wait until right or left mouse button clicked"
  Open(4,File='User')
  Write(4,*) "The current focused Window"
  mouseEvent = MOUSE$RBUTTONDOWN.Or.MOUSE$LBUTTONDOWN
  status = WaitOnMouseEvent(mouseEvent,keyState,x,y)
  If ((MOUSE$KS_SHIFT.And.keyState) == MOUSE$KS_SHIFT ) then
    Write(*,*) 'Shift key was down'
  Else If ((MOUSE$KS_CONTROL.And.keyState) == MOUSE$KS_CONTROL ) then
    Write(*,*) 'Ctrl key was down'
  End If
End Program Ex_14
```

不管是事件方法还是阻塞方法，规定的鼠标事件都发生在子窗口的客户区内。因此，当子窗口最小化时，无法接收鼠标输入。

## 小　结

■ 利用 IFQWin 运行库提供的相关例程，可以对 QuickWin 界面元素进行定制。包括框架窗口的菜单栏和状态栏、框架窗口及其子窗口的标题和图标、程序运行时出现的提示文字以及在子窗口客户区的鼠标输入。

■ 若要自定义 QuickWin 初始菜单，可以在初始化函数 InitialSettings 中进行设置，该函数运行时由系统自动调用。

■ 若要在运行时删除、插入和追加菜单，可分别调用 DeleteMenuQQ、InsertMenuQQ、AppendMenuQQ 函数；若要在运行时修改菜单的标题、回调例程和状态，则可分别调用 ModifyMenuStringQQ、ModifyMenuRoutineQQ 和 ModifyMenuFlagsQQ 函数。

■ 默认情况下，QuickWin 的子窗口列表设置在 Window 菜单下。若要将子窗口列表放置到其他菜单下，可调用 SetWindowMenuQQ 函数。

■ 用户通过使用鼠标来选取菜单项；程序可通过调用 ClickMenuQQ 函数，来模拟鼠标选取菜单项的操作。

■ 若要本地化 QuickWin 程序出现的文字信息(如状态栏信息、对话框提示信息等)，可以调用 SetMessageQQ 例程进行处理。

■ 在 QuickWin 程序中，通过调用 MessageBoxQQ 函数来展示信息对话框；调用 AboutBoxQQ 函数对 About 对话框展示的文字进行定制。

■ 若要对 QuickWin 框架窗口及子窗口的图标进行定制，需向工程中插入相应的图标资源，并规定框架窗口的图标 ID 为“FRAMEICON”，子窗口的图标 ID 为“CHILDICON”。

■ 若要设置 QuickWin 框架窗口标题，须直接调用 Windows API 函数 SetWindowText，QuickWIn 运行库 IFQWin 并没有提供这样的例程。

■ 若要捕获 QuickWin 子窗口客户区产生的鼠标事件，可采用事件方法或阻塞方法。事件方法，需调用 RegisterMouseEvent 函数注册鼠标事件发生时执行的回调例程；uze 方法，则需调用 WatiOnMouseEvent 函数阻塞程序执行，等待特定鼠标事件的发生。

# 第 5 章　使用对话框和控件

对话框是 Windows 应用程序与用户进行交互的重要手段。对话框通常放置诸如命令按钮、编辑框、列表框、滚动条等一些控件，正是在这些控件的帮助下，Windows 应用程序才得以与用户交互的。

对话框分为模式对话框和无模式对话框。Intel Fortran 所有类型的应用程序都支持模式对话框，而无模式对话框通常只在 Fortran Windows 程序中使用。本章只介绍模式对话框的使用，无模式对话框的使用将在第 6 章中介绍。

**本章主要内容：**

- 设计对话框
- 编写对话框程序
- 使用控件

## 5.1 设计对话框

设计对话框包括向工程中插入对话框资源、在对话框资源编辑器中设计对话框及其控件的外观、在相应属性对话框中设置对话框及其控件的属性。

【例 5-1】图 5-1 所示，进行摄氏温度与华氏温度的转换。“温度”分组框包含“摄氏”静态文本框、编辑框，“华氏”静态文本框、编辑框，以及滚动条控件。不论哪个控件的值发生改变，其余控件的值都随之发生变化以保持一致。

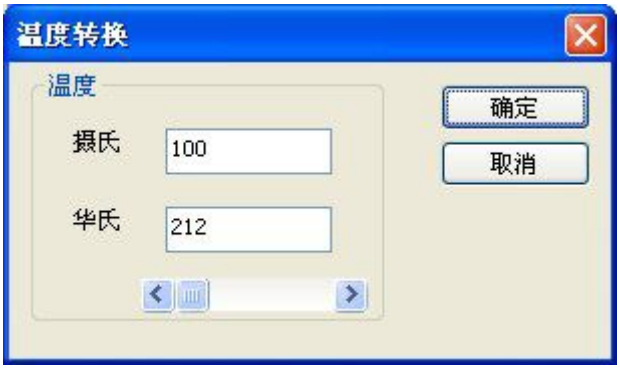



图 5-1 摄氏温度与华氏温度的转换

### 5.1.1　插入对话框资源

在 Intel Fortran 中，图标、对话框等都是资源，被保存在单独的资源文件(.RC)中，编译时连同其他文件一起编译到应用程序执行文件(DLL 和 EXE)中。

添加对话框资源和第 4 章(4.4.1 定制图标)添加图标资源的操作方法相同，只不过此时在“添加资源”对话框中选择“Dialog”资源，点击“新建”按钮后展示图 5-2 所示的初始对话框。




图 5-2 初始对话框和工具箱

如果是.NET IDE 工具箱没有打开，可以从“视图”菜单中打开它。Intel Visual Fortran 支持的控件如表 5-1 所示。

表 5-1 Intel Visual Fortran 所支持的控件

| 控件 | 图标 | 控件 | 图标 |
|---|---|---|---|
| Button | | Check Box | |
| Combo Box | | Edit Control | |
| Group Box | | List Box | |
| Picture Control | | Progress Control | |
| Radio Button | | Horizontal Scroll Bar | |
| Vertical Scroll Bar | | Slider Control | |
| Spin Control | | Static Text | |
| Tab Control | | | |

## 5.1.2 布置对话框

### 5.1.2.1 添加控件

对话框(窗体)是控件的容器，窗体通过控件与用户进行交互。将工具箱中的控件添加至窗体有 3 种方式：一是在工具箱的控件图标上按下鼠标，将控件拖至窗体合适位置，在松开鼠标；二是在工具箱的控件图标上先点击鼠标，再在窗体中用鼠标画一个适当大小的矩形，选取的控件即被放置在该矩形位置上；三是在工具箱的控件图标上双击鼠标，对应的控件随即出现在窗体左上角，再将控件拖至窗体合适位置。

### 5.1.2.2 调整控件的大小和位置

要调整控件的大小和位置，先要在控件上点击鼠标来选取控件(在窗体的非控件位置点击鼠标，选择窗体本身)，或利用 Tab 键在窗体及窗体上的所有控件间切换，被选取的控件周围会出现 8 个伸缩句柄。当鼠标恰好指向句柄时，鼠标指针会变成横向、竖向或对角线方向的箭头图标，此时按下鼠标，在相应方向上伸缩控件，借此调整控件或对话框的大小；当鼠标在 8 个句柄围成的矩形范围内移动时，鼠标指针变成十字箭头，此时按下鼠标将控件拖至适当位置，借此调整控件在对话框中的位置。

控件的大小和位置也可以用键盘来微调。在选取窗体上的控件后，先按下 Shift 键，再按下上、下、左、右方向键，选取的控件即在垂直或水平方向上伸缩；若直接按下方向键，则选取的控件在相应方向上移动。

### 5.1.2.3 统一布置调整多个控件

对话框界面是否美观，和对话框中的控件布置有很大关系。在实际对话框设计中，通常对同一类控件采取统一的大小和对齐方式。对此，.NET IDE 的对话框编辑器提供了相应的快捷按钮，如图 5-3 所示。(如果看不见该工具条，请在工具栏上右击鼠标，在弹出的上下文菜单上勾选对话框编辑器)

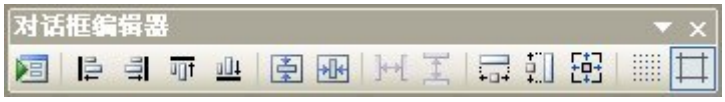



图 5-3 .NET IDE 中的对话框编辑器

在统一布置调整多个控件前，先要选取它们。在窗体中一次选取多个控件有两种方式：一是用鼠标画一矩形框，在矩形框范围内的控件即被选取；二是先按下 Shift 键或 Ctrl 键，在用鼠标点选相应控件。被选取的每个控件周围都会出现 8 个句柄，此时，就可点击图 5-3 所示的快捷按钮，来使多个控件左对齐、右对齐、顶对齐或底对齐，或使多个控件拥有同一个宽度、高度或大小。

另外，通过键盘也可统一调整多个控件的大小和位置，在选取多个控件后其操作与上述单个控件的相同。

示例对话框的布置如图 5-4 所示：1 个分组框、2 个静态文本标签、2 个编辑框、1 个水平滚动条以及 2 个命令按钮。




图 5-4 示例对话框的布置

## 5.1.3 设置对话框及其控件属性

在.NET IDE 中，对话框及控件属性在“属性”窗口中设置。打开属性窗口有两种方式：一是从“试图”菜单执行“属性窗口”菜单命令；二是在对话框或控件上点击鼠标右键，从弹出的上下文菜单中选取“属性”菜单项，打开相应对话框或控件的“属性”窗口。

针对不同的控件，其属性窗口会列出不同的属性项，但整体上都分为外观、行为和杂项三类，如图 5-5 所示。对话框属性还包括“位置”类属性。

其中，控件名是程序当中引用控件的唯一标识，在属性窗口中通过规定控件 ID，浅灰色显示的控件名自动与该控件 ID 保持一致。

在实际开发中，并不是控件的所有属性都要设置，而是视情况需要对控件的某些重要属性进行设置。例如，静态文本控件通常只是用来显示提示文字，作为标签使用，可只设置其 Caption 属性；如果程序中需要动态改变其提示文字，还应设置其 ID 属性，以便程序代码能借此引用它。

示例窗体及其控件的属性设置列于表 5-2 中，经过属性设置和重新布置后的示例对话框外观如图 5-6 所示。

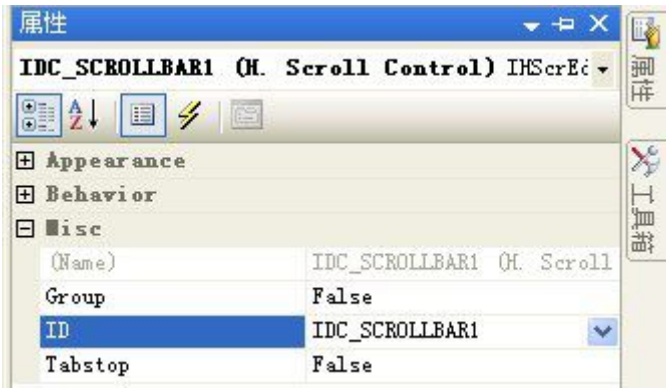



图 5-5 对话框、控件属性设置窗口

表 5-2 例 5-1 窗体及其控件的属性设置

| 窗体及其控件 | 属性类别 | 属性项 | 属性值 |
|---|---|---|---|
| 窗体 | 外观 | Caption | 温度转换 |
| | 杂项 | ID | IDD_TEMP |
| | 位置 | Center | True |
| 分组框 | 外观 | Caption | 温度 |
| 静态文本 1 | 外观 | Caption | 摄氏 |
| | 杂项 | ID | IDC_TEXT_CELSIUS |
| 静态文本 2 | 外观 | Caption | 华氏 |
| 编辑框 1 | 杂项 | ID | IDC_EDIT_CELSIUS |
| 编辑框 2 | 杂项 | ID | IDC_EDIT_FAHRENHEIT |
| 水平滚动条 | 杂项 | ID | IDC_SCROLLBAR_TEMPERATURE |




图 5-6 设计的示例对话框

## 5.1.4 保存对话框资源

当保存添加的图标、对话框等图形资源时，规定与工程名不同的资源脚本文件名(工程名.rc 是主资源脚本文件缺省的文件名)。资源脚本文件是文本文件，可以在解决方案资源管理器窗口列出的资源文件名上单击鼠标右键，打开相关操作的上下文菜单，从中选取“在浏览器中查看”，其中关于示例对话框的描述部分为：

```
IDD_TEMP DIALOGEX 0, 0, 211, 109
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS | DS_CENTER | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "温度转换"
FONT 8, "MS Shell Dlg", 400, 0, 0x1
BEGIN
    DEFPUSHBUTTON   "确定",IDOK,154,7,50,14
    PUSHBUTTON      "取消",IDCANCEL,154,24,50,14
    GROUPBOX        "温度",IDC_STATIC,7,7,127,95
    LTEXT           "华氏",IDC_STATIC,30,61,44,20
    LTEXT           "摄氏",IDC_STATIC,30,28,40,14
    EDITTEXT        IDC_EDIT_FAHRENHEIT,56,61,53,15,ES_AUTOHSCROLL
    SCROLLBAR       IDC_SCROLLBAR_TEMPERATURE,52,88,80,13
    EDITTEXT        IDC_EDIT_CELSIUS,56,28,53,13,ES_AUTOHSCROLL
END
```

开头部分为对话框属性描述，BEGIN 与 END 之间为对话框所包含的控件属性描述。

一个 Visual Studio.NET 工程通常只包含 1 个资源文件。如果工程要使用多个对话框，可将多个对话框资源添加至同一个资源脚本文件(.rc)中。

当第一次保存资源文件时，.NET 资源编辑器创建 Resource.h 文件。该文件采用 C++语法定义了对话框及其控件的标识符常量，并被包含到资源脚本文件中(#include "resource.h")。编译前，需将 Resource.h 文件连同资源脚本文件一起添加到工程中。

示例对话框的 Resource.h 文件为：

```
// Microsoft Visual C++ generated include file used by DlgRes.rc
#define IDD_TEMP                        108
#define IDC_EDIT_CELSIUS                1000
#define IDC_EDIT_FAHRENHEIT             1001
#define IDC_SCROLLBAR_TEMPERATURE       1002

// Next default values for new objects
#ifdef APSTUDIO_INVOKED
        #ifndef APSTUDIO_READONLY_SYMBOLS
        #define _APS_NEXT_RESOURCE_VALUE        109
        #define _APS_NEXT_COMMAND_VALUE         40001
        #define _APS_NEXT_CONTROL_VALUE         1004
        #define _APS_NEXT_SYMED_VALUE           101
        #endif
#endif
```

由此可见，在设置对话框、控件属性时规定的 ID(Name),实际上被 Visual C++.NET 定义为整型符号常量，即用有代表意义的符号代替整型数来引用对话框和组件。

但 Intel Fortran 程序还不能识别 Visual C++语法，鉴于这种情况，Intel Visual Fortran 专门提供了 defToFD 工具，用来将 resource.h 头文件转换成 Fortran 形式的头文件 resource.fd：

```
! resource.fd generated from resource.h
      integer, parameter :: IDC_EDIT_CELSIUS = 1000
      integer, parameter :: $_WIN32 = 1
      integer, parameter :: IDC_SCROLLBAR_TEMPERATURE = 1002
      integer, parameter :: IDD_TEMP = 108
      integer, parameter :: IDC_EDIT_FAHRENHEIT = 1001
```

在命令行使用 defToFD 工具的格式为：

defToFD resource.h resource.FD

这里推荐使用属性命令行设置方法：在解决方案资源管理器中，在 resource.h 文件名上点击鼠标右键，从出现的上下文菜单中打开属性页，从属性页 General 选项卡中规定 Command Line 选项为：“defToFD resource.h resource.FD”,Description 选项为：“Generating Fortran include file...”，Outputs 选项为“resource.FD”，如图 5-7 所示：

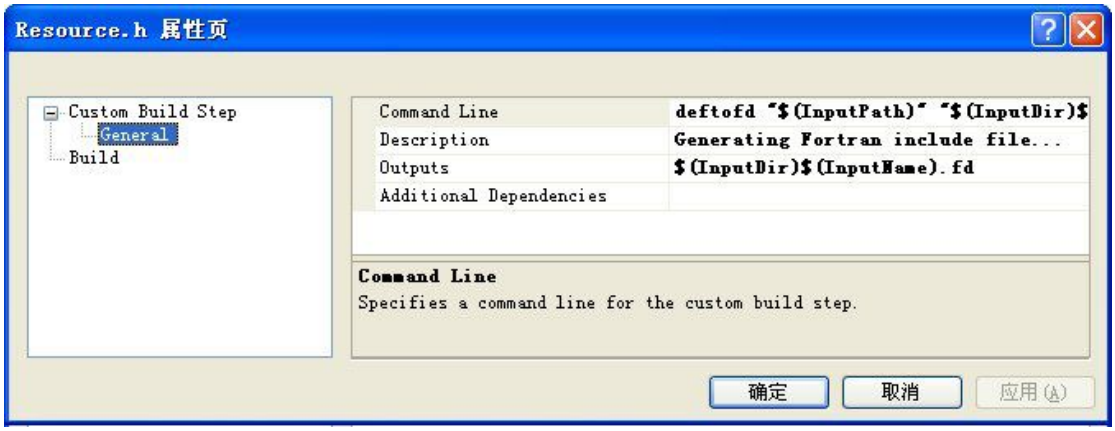



图 5-7 resource.h 属性页

使用属性命令行的好处是：当在资源编辑器中重新设置对话框及控件时，资源编辑器会自动更新 resource.h 文件，编译时 defToFD 工具又会自动更新 resource.fd 文件，这样两个文件总能保持与对话框及控件一致，而不需要从命令行反复执行 defToFD 命令。

## 5.2 编写对话框资源

### 5.2.1 引用对话框资源

要在程序当中操作对话框及其控件，需要引用对话框模块(IFLogM)，并包含资源头文件(resourcd.fd)。类似于 C++中的类，Intel Fortran 提供的对话框模块 IFLogM，将对话框数据类型和相关的操作例程封装在一起，IFLogM 模块一经引用，就可在程序中使用它们；将资源头文件 resource.fd 包含到程序中，就可以使用其定义的符号常量来标识对话框和控件资源。如下列程序代码所示：

```
Program TEMPERATURE
  Use IFLogM
  Implicit None
  Include 'Resource.FD'
  ......
```

```
  !Call dialog routines,such as DlgInit,DlgModal,and DlgUnInit
  ......
End Program TEMPERATURE
```

### 5.2.2 操作对话框及其控件

程序中，通过调用 IFLogM 模块中的对话框例程来操作对话框，其大致步骤为：

(1) 调用 DlgInit 或 DlgInitWithResourceHandle 例程初始化对话框类型，并将对话框及其属性与对话框类型相关联；

(2) 调用 DlgSet 例程初始化对话框上的控件；

(3) 调用 DlgSetSub 例程设置用户操作控件时执行的回调例程；

(4) 调用 DlgModal 例程激活、展示模式对话框；

(5) 调用 DlgGet 例程获取控件的用户输入信息；

(6) 调用 DlgUnInit 例程释放对话框资源。

如下列温度转换主程序所调用的唯一自程序 DoDialog 所示：

```
Program TEMP
  Implicit None
  Call DoDialog()
End Program TEMP
Subroutine DoDialog()
  Use IFLogM
  Implicit None
  Include 'Resource.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 dialog
    retLog = DlgSet( dlg , IDC_SCROLLBAR_TEMPERATURE , 200 , DLG_RANGEMAX )
    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 )
    !Active the modal dialog.
    retInt = DlgModal( dlg )
    !Release dialog resources.
    Call DlgUnInit( dlg )
  End If
End Subroutine DoDialog
```

其中：

DlgInit(IDD_TEMP,dlg)将对话框变量 dlg 与对话框资源 IDD_TEMP 相关联。这样，程序中就可以通过变量 dlg 来引用对话框。

retLog = DlgSet( dlg , IDC_SCROLLBAR_TEMPERATURE , 200 , DLG_RANGEMAX )将对话框上的滚动条(IDC_SCROLLBAR_TEMPERATURE)的最大范围设定为 200，DLG_RANGEMAX 索引代表滚动条最大范围。

retLog = DlgSet( dlg , IDC_EDIT_CELSIUS , "100" )

Call UpdateTemp( dlg , IDC_EDIT_CELSIUS , DLG_CHANGE )

将编辑框(IDC_EDIT_CELSIUS)的显示值设定为 100，调用自定义回调例程 UpdateTemp，向应用程序发出消息通知：编辑框的值发生改变。

retLog = DlgSetSub( dlg , IDC_EDIT_CELSIUS , UpdateTemp )

retLog = DlgSetSub( dlg , IDC_EDIT_FAHRENHEIT , UpdateTemp )

retLog = DlgSetSub( dlg , IDC_SCROLLBAR_TEMPERATURE , UpdateTemp )

设置两个编辑框和一个滚动条控件的回调例程维UpdateTemp。这样，一旦编辑框，、滚动条的值发生改变，程序会自动执行UpdateTemp回调例程。DlgSetSub函数用来设置控件的回调例程，它有4个参数：第一个参数为对话框变量；第二个参数为控件ID；第三个参数为自定义的控件回调例程；第四个(可选)参数为控件回调例程索引(针对一个控件内有多个回调例程的情况)。

retInt = DlgModal(dlg)

用来激活、展示模式对话框dlg。

Call DlgUnInit(dlg)

用来关闭对话框dlg，并释放对话框所占用的内存资源。

IFLogM模块中的对话框操作例程如表5-3所示。

表 5-3 IFLogM 模块中的对话框例程

| 对话框例程 | 描述 |
|---|---|
| DlgExit | 关闭打开的对话框 |
| DlgFlush | 更新对话框显示 |
| DlgGet | 获取控件变量值 |
| DlgGetChar | 获取字符型控件变量值 |
| DlgGetInt | 获取整数型控件变量值 |
| DlgGetLog | 获取逻辑型控件变量值 |
| DlgInit | 初始化对话框 |
| DlgInitWithResourceHandle | 初始化 DLL 中的对话框 |
| DlgIsDlgMessage | 判断消息是否来自于无模式对话框 |
| DlgIsDlgMessageWithDlg | 判断消息是否来自于 DLL 中的无模式对话框 |
| DlgModal | 展示模式对话框 |
| DlgModalWithParent | 展示拥有父窗口的模式对话框 |
| DlgModeless | 展示无模式对话框 |
| DlgSendCtrlMessage | 向对话框上的控件发送消息 |
| DlgSet | 给控件变量赋值 |
| DlgSetChar | 给字符型控件变量赋值 |
| DlgSetCtrlEventHandler | 设置 ActiveX 控件的事件处理柄 |
| DlgSetInt | 给整数型控件变量赋值 |
| DlgSetLog | 给逻辑型控件变量赋值 |

| | |
|---|---|
| DlgSetReturn | (在回调例程内)设置 DlgModal 函数的返回值 |
| DlgSetSub | 设置控件的回调例程 |
| DlgSetTitle | 设置对话框标题 |
| DlgUnInit | 释放对话框占用的内存 |

### 5.2.3 编写控件回调例程

在Windows基于消息驱动的应用程序中，主要任务是编写控件的事件例程；而在Intel Visual Fortran 对话框程序设计中，主要任务则是编写控件的回调例程。尽管事件和回调的实现机制不同，但两者都是当控件发生某个事件时程序执行的动作。

控件的回调例程作为子程序加以实现，其原型为：

Subroutine callBack-Routine-Name(dlg,control_Name,callBackType)

其中，dlg参数指包含设置回调例程控件的对话框；control_Name参数指设置回调例程的控件；callBackType参数指发生的回调或事件类型，如DLG_CLICKED、DLG_CHANGE、DLG_DBLCLICK。

利用回调例程中callBackType和control_Name两个参数，可以为同一个控件设置多个回调例程，在这种情况下，须向函数DlgSetSub传入第四个索引参数，标识哪一类回调例程被调用；也可以使多个逻辑相关的控件共享同一个回调例程。例如，温度转换示例对话框，其摄氏编辑框、华氏编辑框和滚动条三个控件中，不管哪一个控件的值发生改变，都要刷新另外两个控件的值，以使它们保持一致。故此，示例程序为三个控件设置了同一个回调例程 UpdateTemp：

```
Subroutine UpdateTemp(dlg,control_Name,callBackType)
  !DEC$ ATTRIBUTES DEFAULT::UpdateTemp
  Use IFLogM
  Implicit None
  Type(dialog) dlg
  Integer control_Name
  Integer callBackType
  Include 'Resource.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 / ( 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
```

其中：

```
!DEC$ ATTRIBUTES DEFAULT::UpdateTemp
```

通过编译器指令设置外部例程UpdateTemp的属性。Default属性代表覆盖那些影响外部例程的编译器选项，此处用来保持缺省的调用约定、命名约定。

除命令按钮(pushButton)外，对话框上的每一个控件都有一个执行空操作的缺省回调例程。而命令按钮缺省回调例程的行为是，设置展示对话框函数的返回值为被单击的命令按钮名，然后退出对话框。这使得单击任何命令按钮都产生退出对话框的默认行为，调用DlgModal函数的例程可以判断究竟是哪一个命令按钮导致退出对话框的。

当用户改变控件值时，控件的回调例程被执行；而调用DlgSet函数改变控件值时，控件的回调例程并不执行。若要程序执行控件的回调例程，可采用显式调用方式。例如：

```
retLog = DlgSet(dlg,IDC_SCROLLBAR_TEMPERATURE,200,DLG_RANGEMAX)
retLog = DlgSet(dlg,IDC_EDIT_CELSISU,100)
Call UpdateTemp(dlg,IDC_EDIT_CELSISU,DLG_CHANGE)
```

### 5.2.4 控制对话框的退出

默认情况下，当用户点击“确定(OK)”、“取消(Cancel)”或其他任何命令按钮时，对话框退出并返回点击的命令按钮名(ID)，如IDOK、IDCANCEL等。点击对话框关闭按钮等同于点击ID号为IDCANCEL的命令按钮，若没有这样的命令按钮存在，点击关闭按钮对话框不会退出。

通过注册命令按钮的回调例程，可以实现对话框的有条件退出：若满足某种甜椒要求，就调用DlgExit例程退出对话框；否则，就不调用DlgExit函数，使对话框仍呈现在屏幕上。例如：在“确定”按钮的回调过程中，先检查用户输入数据的合法性，若不合法，可弹出一个消息框，提示用户予以改正；若合法，就调用DlgExit例程退出。

在退出对话框时，还可设置特定的整型返回值，而不是缺省的命令按钮ID。例如，我们为上述示例对话框的“确定按钮注册Temp_OK回调例程”。

```
  retLog = DlgSetSub( dlg , IDOK , Temp_OK )
......
Subroutine Temp_OK(dlg,control_Name,callBackType)
  !DEC$ ATTRIBUTES DEFAULT::Temp_OK
  Use IFLogM
  Implicit None
  Type(dialog) dlg
  Integer control_Name,callBackType
  Include 'Resource.FD'
  Integer retInt
  Logical retLog
  retLog = DlgGet( dlg , IDC_SCROLLBAR_TEMPERATURE , retInt , DLG_POSITION )
  Call DlgSetReturn(dlg,retInt)
  Call DlgExit(dlg)
End Subroutine Temp_OK
```

回调例程中，调用DlgSetReturn例程，将retInt变量存放的滚动条当前值设为“确定”按钮的返回值；再调用DlgExit例程，退出对话框。这样，展示对话框的程序就可对返回值做进一步处理，如下列代码所示：

```
    !Active the modal dialog.
    retInt = DlgModal( dlg )
    If (retInt /= IDCANCEL) then
      Print * , retInt
      Read*
    End If
```

若调用DlgModal函数没有打开对话框，其返回值为-1。所以，用户设置的返回值不应是-1。

## 5.3 使用控件

### 5.3.1 控件索引

从上面的示例程序可以看到，在程序中通过控件名(ID)或整型符号常量引用控件。除了控件名外，每个控件还有其他相关的属性以及回调(事件)。标识这些属性和回调的控件变量称为控件索引(control index)，这些索引可以是整型、逻辑型、字符型或外部的。比如普通的命令按钮有3个索引：一是逻辑索引，指按钮当时是否启用；二是字符索引，指按钮标

题；三是外部索引，指鼠标单击事件发生时执行的回调例程(外部子程序)。同一种类型的索引可以有多个，比如滚动条控件就有4个整型索引，分别代表滚动条位置、滚动条最小范围、滚动条最大范围以及用户在滚动条上单击鼠标引起的位置变化量。

温度转换示范对话框上的滚动条，在初始化时就分别使用了最大范围索引和位置索引：

```
retLog = DlgSet( dlg , IDC_SCROLLBAR_TEMPERATURE , 200 , DLG_RANGEMAX )
retLog = DlgSet( dlg , IDC_SCROLLBAR_TEMPERATURE , cel , DLG_POSITION )
```

IFLogM模块中定义的控件索引和对话框索引分别列于表5-4和表5-5，每个控件对应的所有类型索引列于表5-6。

表 5-4 IFLogM 模块中的控件索引

| 控件索引 | 描述 |
|---|---|
| DLG_ADDSTRING | 作为 DlgSetChar 例程的实参，指向列表框或组合框添加一字符串 |
| DLG_BIGSTEP | 指用户在滚动条或滑动条的滑槽内单击鼠标时的位置变换量(缺省值为 10) |
| DLG_CHANGE | 指控件值发生改变事件 |
| DLG_CLICKED | 指控件的鼠标单击事件 |
| DLG_DBLCLICK | 指控件的鼠标双击事件 |
| DLG_DEFAULT | 效果等同于没有规定控件索引 |
| DLG_ENABLE | 指控件当前是否启用(值.True.表示启动，.False.表示没有启用、呈浅灰色) |
| DLG_GAINFOCUS | 指编辑框获得输入焦点事件 |
| DLG_IDISPATCH | 指 ActiveX 控件的对象事件 |
| DLG_LOSEFOCUS | 指编辑框失去输入焦点事件 |
| DLG_NUMITEMS | 指列表框、组合框或选项卡控件项数 |
| DLG_POSITION | 指滚动条、微调按钮、滑动条或进度条的当前位置，以及编辑框的当前光标位置 |
| DLG_RANGEMIN | 指滚动条、微调按钮、滑动条或进度条的位置最小值(滚动条的缺省值为 1，其他为 0) |
| DLG_RANGEMAX | 指滚动条、微调按钮、滑动条或进度条的位置最大值(缺省值为 100) |
| DLG_SELCHANGE | 列表框或组合框的选项发生改变事件 |
| DLG_SELCHANGING | 指选项卡控件的选项将要发生改变事件 |
| DLG_SMALLSTEP | 指用户按下键盘箭头键时滑动条的位置变化量(缺省值为 1) |
| DLG_STATE | 指用户可改变的控件状态 |
| DLG_TEXTLENGTH | 指编辑框内的文本长度 |
| DLG_TICKFREQ | 指滑动条的间隔频率(缺省值为 1) |
| DLG_TITLE | 指控件标题 |
| DLG_UPDATE | 指用户修改了控件状态、需要在屏幕上刷新控件的事件 |

表 5-5 IFLogM 模块中的对话框索引

| 对话框索引 | 描述 |
|---|---|
| DLG_INIT | 指对话框被创建、但还没有显示时进行的初始化事件 |
| DLG_SIZECHANGE | 指对话框大小发生改变的事件 |

表 5-6 控件所有的索引

| 控件 | 整型索引 | 逻辑型索引 | 字符型索引 | 外部(回调)索引 |
|---|---|---|---|---|
| ActiveX control | DLG_IDISPATCH | DLG_ENABLE | | |
| Button | | DLG_ENABLE | DLG_TITLE | DLG_CLICKED |
| Check box | | DLG_STATE*<br>DLG_ENABLE | DLG_TITLE | DLG_CLICKED |
| Combo box | DLG_NUMITEMS | DLG_ENABLE | DLG_STATE*<br>DLG_ADDSTRING<br>and index(1 to n) | DLG_SELCHANGE*<br>DLG_DBLCLICK<br>DLG_CHANGE<br>DLG_UPDATE |
| Drop-down list box | DLG_NUMITEMS*<br>DLG_STATE | DLG_ENABLE | DLG_STATE*<br>DLG_ADDSTRING<br>and index(1 to n) | DLG_SELCHANGE*<br>DLG_DBLCLICK |
| Edit box | DLG_TEXTLENGTH*<br>DLG_POSITION | DLG_ENABLE | DLG_STATE | DLG_CHANGE*<br>DLG_UPDATE<br>DLG_GAINFOCUS<br>DLG_LOSEFOCUS |
| Group box | | DLG_ENABLE | DLG_TITLE | |
| List box | | DLG_ENABLE | DLG_STATE<br>DLG_ADDSTRING<br>and index(1 to n) | DLG_SELCHANGE*<br>DLG_DBLCLICK |
| Picture | | DLG_ENABLE | | |
| Progress bar | DLG_POSITION*<br>and index(1 to n) | DLG_ENABLE | | |
| Radio button | | DLG_STATE*<br>DLG_ENABLE | DLG_TITLE | DLG_CLICKED |
| Scroll bar | DLG_POSITION*<br>DLG_RANGEMIN<br>DLG_RANGEMAX<br>DLG_BIGSTEP | DLG_ENABLE | | DLG_CHANGE |
| Slider | DLG_POSITION*<br>DLG_RANGEMIN<br>DLG_RANGEMAX<br>DLG_SMALLSTEP<br>DLG_BIGSTEP<br>DLG_TICKFREQ | DLG_ENABLE | | DLG_CHANGE |
| Spin controls | DLG_POSITION*<br>DLG_RANGEMIN<br>DLG_RANGEMAX | DLG_ENABLE | DLG_TITLE | DLG_CHANGE |
| Static text | | DLG_ENABLE | DLG_TITLE | |
| Tab control | DLG_NUMITEMS*<br>DLG_STATE | DLG_ENABLE | DLG_STATE*<br>and index(1 to n) | DLG_SELCHANGE*<br>DLG_SELCHANGING |

| | | | | |
|---|---|---|---|---|
| | and index(1 to n) | | | |

注：*表示索引为缺省索引

在 DlgSetSub 例程参数中规定的控件索引也可以是对话框索引，在这种情况下，对话框索引必须是表 5-5 种的索引。

在设置控件值时，假如某种类型的控件索引只有一个，那么不必规定控件索引。例如，要将示例对话框上的静态文本控件 IDC_TEXT_CELSIUS 的标题设为“New Celsius Title”，可以使用下列语句：

```
retLog = DlgSet(dlg,IDC_TEXT_CELSIUS,"New Celsius Title")
retLog = DlgSetChar(dlg,IDC_TEXT_CELSIUS,"New Celsius Title")或
retLog = DlgSet(dlg,IDC_TEXT_CELSIUS,"New Celsius Title",DLG_TITLE)
retLog = DlgSetChar(dlg,IDC_TEXT_CELSIUS,"New Celsius Title",DLG_TITLE)
```

此处，静态文本控件的字符索引只有 DLG_TITLE，所以加不加 DLG_TITLE 索引名均可。

如上列语句所示，针对每种类型的控件索引，可以使用一般的控件设置函数 DlgSet，也可以使用具体的控件设置函数 DlgSetInt、DlgSetLog 和 DlgSetChar。一般的控件设置函数 DlgSet 根据参数类型选择控件索引类型，上列语句的控件索引类型为字符型。

同样，要获取控件某种索引类型的值，既可以使用一般的获取控件值函数 DlgGet，也可以使用具体的获取控件值函数 DlgGetInt、DlgGetLog 和 DlgGetChar。例如：

```
retLog = DlgGet(dlg,IDC_SCROLLBAR_TEMPERATURE,current_val,DLG_POSITION)
retLog = DlgGetInt(dlg,IDC_SCROLLBAR_TEMPERATURE,current_val,DLG_POSITION)
```

上列两语句都用来获取示例对话框上的滚动条控件 IDC_SCROLLBAR_TEMPERATURE 的当前位置值(存放在 current_val 整型变量中)。

若使用具体的获取控件值函数，则函数必须和获取的控件索引类型相匹配。例如，在上列语句中，不能用 DlgGetChar 代替 DlgGetInt 来获取滚动条的当前位置值(整型)。

推荐使用一般函数 DlgSet 和 DlgGet。因为一般函数能够根据传入的参数类型自动执行正确的操作，而不必关心函数与设置获取的控件索引类型是否匹配的问题。

## 5.3.2 使用控件



图 5-8 控件的Tab键顺序

合理利用.NET 工具箱提供的各种控件，可以使应用程序拥有友好的操作界面和强大的功能。

当对话框程序运行时，按下 Tab 键可以使输入焦点在各个控件之间切换。通常情况下，控件的 Tab 键顺序与控件创建(即控件被添加至对话框)顺序相同。如果要调整控件的 Tab 键顺序，可以从“格式”菜单执行“Tab 键顺序”命令(Ctrl+D)，每个控件左上角会出现其 Tab 键顺序数字(图 5-8)。此时，按需要的顺序依次用鼠标点击各控件。调整好各控件的 Tab 键顺序后，按 Esc 键退出该状态。

通常，分组框、静态文本等控件并不要求拥有输入焦点。在设计状态下，可以将其属性页中的 Tabstop 属性设为 False。这样，当对话框程序运行时，按 Tab 键会跳过这些控件。

每个控件都有一个 DLG_ENABLE 控件索引，决定是否启用控件。未启用的控件呈浅灰色，它不响应用户的任何操作。程序设计中，有时会利用这一现象来阻止用户对控件进行操作。

例如：

retLog=DlgSet(dlg,IDC_CHECKBOX1,.False.,DLG_ENABLE)或

retLog=DlgSetLog(dlg,IDC_CHECKBOX1,.False.,DLG_ENABLE)

上列语句使 IDC_CHECKBOX1 控件处于未启用状态，当中的 retLog 为逻辑变量。

下面简要介绍各控件的使用方法。

#### 5.3.2.1 静态文本(Static Text)

静态文本控件用来在对话框上展示文字，用户无法改变其文字，故此称为静态文本。通常，静态文本控件作为标签，来标识其他控件(如编辑框)；或向用户展示信息。

在设计状态下，可在其属性页外观类别下，通过其 Caption 属性来设置展示的文字;在运行状态下，则通过 DlgSet 或 DlgSetChar 函数来设置其 Dlg_Title 控件索引值：

retLog=DlgSet(dlg,IDC_TEXT_CELSIUS,"Celsius Title",DLG_TITLE)或

retLog=DlgSetChar(dlg,IDC_TEXT_CELSIUS,"Celsius Title",DLG_TITLE)

上列语句将 IDC_TEXT_CELSIUS 静态文本控件的展示文字设为“Celsius Title”。DLG_TITLE 为静态文本控件的缺省控件索引，故可省略 DlgSet 或 DlgSetChar 函数的第四个可选参数。

#### 5.3.2.2 编辑框(Edit Control)

编辑框用来展示文字，同时允许用户输入文字，所以它是最常用的控件。

要设置编辑框内容，只能在程序中设置。例如：

Character(20) text/"Send text"/

retLog = DlgSet(dlg,IDC_EDITBOX1,text)或

retLog = DlgSetChar(dlg,IDC_EDITBOX1,text)

要获取编辑框内容，则通过 DlgGet 或 DlgGetChar 函数来实现。例如：

retLog = DlgGet(dlg,IDC_EDITBOX1,text)或

retLog = DlgGetChar(dlg,IDC_EDITBOX1,text)

编辑框内容总是作为字符(串)对待，若要设置、获取数字字符，则需使用内部读、写语句进行字符型和数字型之间的转换。例如，下列程序段先将整型转换为字符，再设为编辑框 IDC_EDITBOX1 中的内容：

```
Integer j
Logical retLog
Character(256) text
Write(text,'(I4)') j
retLog = DlgSet( dlg , IDC_EDITBOX1 , text )
```

而下列程序段先获取编辑框 IDC_EDITBOX1 的字符，再将字符转换成实数：

```
Real x
Logical retLog
Character(256) text
retLog = DlgGet( dlg , IDC_EDITBOX1 , text )
Read(text,*) x
```

要设置、获取编辑框内的字符串长度，须使用 DLG_TEXTLENGTH 控件索引。在下列两种情况下，字符串长度自动被更新：

(1)当调用 DlgSet 函数设置编辑框的字符时(尾部空格被去掉)；

(2)当用户修改编辑框的字符时。

若要保留尾部空格，可先调用 DlgSet 函数设置编辑框的字符，再调用带 DLG_TEXTLENGTH 控件索引参数的 DlgSet 函数设置所需的字符串长度。

使用 DLG_POSITION 控件索引来设置、获取当前光标在编辑框中的位置，同时取消原来的字符选取。

#### 5.3.2.3 分组框(Group Box)

分组框用来将相关的控件分为一个逻辑组，以方便用户操作。通常在设计状态下，在属性页的外观类别下设置 Caption 属性，为分组框提供一个标题。在设置标题时，可以在要加下划线的字母前添加“和”号(&)来设置热键。例如，设置分组框标题为“&Temperature”，当用户按下 Alt + T 键时，对话框上的当前焦点会移至 Tab 键次序在分组框之后的第一个控件。

若使分组框失效(不启用)，其热键也随之失效，但对分组框内的控件没有影响。作为一种好的编程风格，当分组框失效时，应使分组框内的所有控件也失效。

#### 5.3.2.4 核取框(Check Box)和选项按钮(Radio Button)

核取框和选项按钮分别用于多选和单选，即核取框允许一次选取多个选项，而选项按钮一次只能在多个选项中选取一项。核取框和选项按钮只存在两种状态：选取和未选取，其逻辑值分别对应逻辑真(.True.)和逻辑(.False.)假。要获取核取框和选项按钮的状态，使用 DlgSet 或 DlgGetLog 函数，如下列代码段所示：

```
Logical pushed_State,checked_State,regLog
retLog = DlgGet( dlg , IDC_RADIOBUTTON1 , pushed_State )
retLog = DlgGet( dlg , IDC_CHECKBOX1 , checked_State )
```

在初始化或响应用户输入时，可以使用 DlgSet 或 DlgSetLog 函数设置其状态，如下列代码段所示：

```
Logical regLog
retLog = DlgSet( dlg , IDC_RADIOBUTTON1 , .True. )
retLog = DlgGet( dlg , IDC_CHECKBOX1 , .True. )
```

选项按钮一般分组使用，其指导原则是：

(1)其“Auto”属性设为 True(缺省)。

(2)一组当中第一个选项按钮的“Group”属性设为 True。

(3)其他选项按钮的“Group”属性设为 False(缺省)，它们的 Tab 键次序要紧接第一个选项按钮。

(4)当用户选取一个选项按钮时，其状态设为逻辑真(.True.)，其他选项按钮设为逻辑假(.False.)。

(5)实现时，使用 DlgSet 或 DlgSetLog 函数将选取的选项按钮设为逻辑真(.True.)，其他选项按钮自动被设为逻辑假(.False.)。

#### 5.3.2.5 命令按钮(Button)

与核取框和选项按钮不同，命令按钮没有状态。使用命令按钮的唯一目的，是引发一个动作。要使这一目的得以实现，须使用 DlgSetSub 函数先注册其回调例程。这样，当用户用鼠标点击命令按钮时，回调例程规定的动作才能被执行。下列代码段注册 IDC_BUTTON_TIME

命令按钮的回调例程为 DisplayTime：

```
Logical regLog
External DisplayTime
retLog = DlgSetSub( dlg , IDC_BUTTON_TIME , DisplayTime )
```

需要注意的是，Intel Visual Fortran 对话框例程不支持用户绘制的命令按钮。

#### 5.3.2.6 列表框(List Box)和组合框(Combo Box)

类似于选项按钮，列表框和组合框也可从多个选项中选取项，但列表框和组合框中的项可滚动，因此不受对话框空间限制，且列表框和组合框中的项可动态增加或减少。

列表框和组合框的不同在于：列表框只是项的列表，而组合框是列表框和编辑框的组合；列表框允许从列表中一次选取多项，而组合框一次只能选取一项；列表框只能从列表中选择项，而组合框允许编辑选取的项。

需要注意的是，Intel Visual Fortran 对话框例程不支持用户绘制的列表框和组合框。

*1) 列表框*

在列表框和组合框中，DLG_NUMITEMS 控件索引规定了列表项数，因此在添加列表项前，需要先设置 DLG_NUMITEMS 索引。例如：

```
Logical regLog
retLog = DlgSet( dlg , IDC_LIST1 , 3 , DLG_NUMITEMS )
retLog = DlgSet( dlg , IDC_LIST1 , "Moe"   , 1 )
retLog = DlgSet( dlg , IDC_LIST1 , "Larry" , 2 )
retLog = DelSet( dlg , IDC_LIST1 , "Curly" , 3 )
```

上列代码，在添加列表项的同时规定规定每一项的索引。

在任何时候(包括从回调例程内部)，都可重新设置列表的项数和列表框内容。若规定的列表项数比实际的少，多出的列别项被截断；若规定列表项数比实际的多，多出的列表项用空白填充。例如，在上诉列表尾部增加一项：

```
retLog = DlgSet( dlg , IDC_LIST1 , 4 )
retLog = DlgSet( dlg , IDC_LIST1 , "Shemp" , 4 )
```

另一种添加列表框和组合框项的办法，是使用 DLG_ADDSTRING 控件索引。该方法在添加列表项的过程中，其 DLG_NUMITEMS 项数自动增加。如上列语句可改写为：

```
retLog = DlgSet( dlg , IDC_LIST1 , "Moe" , DLG_ADDSTRING )
retLog = DlgSet( dlg , IDC_LIST1 , "Larry" , DLG_ADDSTRING )
retLog = DlgSet( dlg , IDC_LIST1 , "Curly" , DLG_ADDSTRING )
```

缺省情况下，列表框为单选。要使列表框为多选，须在属性页中将其“Selection”属性规定为 Multiple 或 Extended。当按下 Ctrl 或 Shift 键再点列表项时，Multiple 允许选择项多项，而 Extended 还允许拖动鼠标来选取连续的多项。

当用户选取列表中的项时，选取的项被赋以规定的索引。因此，可以通过读取选项的索引，来判断哪些项被选取。如下列代码段所示：

```
Integer j,num,test
Integer,Allocatable::values(:)
Logical regLog
retLog = DlgGet( dlg , IDC_LIST1 , num , DLG_NUMITEMS )
Allocate(values(num))
j = 1
test = - 1
```

```
Do While ( test.NE.0 )
  retLog = DlgGet( dlg , IDC_LIST1 , values(j) , j )
  test = values(j)
  j = j + 1
End Do
```

若用户选取了 Moe 和 Curly，其索引分别为 1 和 3；若只有 Larry 被选取，其索引为 2。

需要注意的是，只有将上述列表框的“Sort”属性设为 False，才能使读出的选项索引与添加列表项时规定的索引一致；若将列表框的“Sort”属性设为 True(缺省)，展示的列表项会自动按 ASCII 顺序排序，这有可能使读出的选项索引与添加列表项时规定的索引不一致。例如，上面添加的列表为 Moe、Larry 和 Curly，展示时次序变为 Curly、Larry 和 Moe。

若要读出单选项或第一选项，可使用 DLG_STATE 控件索引。例如：

retLog = DlgGet( dlg , IDC_LIST1 , str , DLG_STATE )

其中，str 为字符变量，用来存放选项字符串。

*2) 组合框*

组合框是列表框和编辑框的组合，它分为简单(Simple)组合框、下拉(Dropdown)组合框和下拉列表(Drop List)组合框 3 种。

简单组合框上面一个编辑框，可以直接在上面的编辑框内输入项，也可从下面的列表框内选取项，被选取的项自动填入上面的编辑框；下拉组合框和下拉列表组合框平时展示带向下箭头的编辑框，当点击向下箭头时弹出列表框，此时可从列表框中选取项。下拉列表组合框只能从列表框中选取项，而下拉组合框还允许在编辑框中输入项。

在将组合框添加至对话框时，应为组合框的下拉列表框留有足够的空间，否则，列表框无法展开。针对下拉组合框和下拉列表组合框，可点击编辑框的向下箭头，用鼠标拉出一个足够大的矩形。

组合框采用哪一种类型由其“Type”属性决定，该属性有 3 个值：Simple、Dropdown、DropList，分别对应简单组合框、下拉组合框和下拉列表组合框。

因为组合框的输入存在两种方式：从下拉列表框中选取项和从编辑框中直接输入项，所以需要分别注册 DLG_SELCHANGE 和 DLG_UPDATE 两种类型的回调例程。如下列程序代码所示：

retLog = DlgSetSub( dlg , IDC_COMBO1 , UpdateCombo , DLG_SELCHANGE )

retLog = DlgSetSub( dlg , IDC_COMBO1 , UpdateCombo , DLG_UPDATE )

这样，无论用户从下拉列表框中选取还是从编辑框中直接输入，都会引发执行 IDC_COMBO1 组合框的回调例程 UpdateCombo。

要获取组合框的输入项获选项，可使用下列语句：

retLog = DlgGet( dlg , IDC_COMBO1 , str)

其中，str 为字符变量，存放组合框的输入项或选项。

在 3 种组合框中，唯独下拉列表组合框不允许编辑，只能从列表框中选取项。换句话说，下拉列表组合框中的项或索引是固定的，可以使用下列语句获取选项索引：

retLog = DlgSet( dlg , IDC_COMBO1 , num , DLG_STATE )

其中，num 为整型变量，用来存放选项在列表中的索引。

需引起注意的是，组合框展示的列表项也受其“Sort”属性影响。

#### 5.3.2.7 滚动条(Scroll Bar)

滚动条也是获取用户输入的一种控件。滚动条有 4 个整型控件索引：DLG_POSITION、DLG_RANGMIN、DLG_RANGMAX 和 DLG_BIGSTEP,分别标识滑块在滑槽中的位置、滚动条的最小值、滚动条的最大值和在滑槽内点击鼠标引起的位置变化量。

滚动条的最小值、滚动条的最大值缺省分别为 1 和 100。例如，下列语句将滚动条的 IDC_SCROLLBAR1 的最大值设为 212：

```
Logical retLog
retLog = DlgSet( dlg , IDC_SCROLLBAR1 , 212 ,DLG_RANGEMAX )
```

下列语句使用带 DLG_POSITION 控件索引的 DlgGet 函数，来获取滚动条 IDC_SCROLLBAR1 的滑块位置：

```
Integer slide_Position
retLog = DlgGet( dlg , IDC_SCROLLBAR1 , slide_Position , DLG_POSITION )
```

使用 DLG_BIGSTEP 控件索引，来设置在滑槽内点击鼠标引起的位置变化量。例如：

```
retLog = DlgSet( dlg , IDC_SCROLLBAR1 , 20 , DLG_BIGSTEP )
```

当用户在滚动条的上、下(或左、右)箭头上点击鼠标时，其递增量或递减量总是 1。

滚动条的最大位置(DLG_POSITION)由其最大范围(DLG_RANGEMAX 或 DLG_RANGE)和页大小(DLG_BIGSTEP)所决定，具体可通过下列公式计算：

```
MaxScrollPos = MaxRangeValue - ( PageSize - 1 )
```

假设滚动条的 DLG_RANGEMAX=100(缺省值),DLG_BIGSTEP=10(缺省值)，那么，DLG_POSITION 的值为 100-(10-1)=91。

#### 5.3.2.8 图片框(Picture Control)

图片框用来展示静态图像。其展示的图像是在设计状态通过图片框属性页设置的，运行时用户不能改变其中的图像，即图片框不接受用户的输入，因此它不支持任何回调。

以图片框展示现成的位图(.bmp)为例：先将位图文件复制到当前工程目录下，再将位图资源添加至资源脚本文件(.rc)中，在属性页中规定其 ID 属性；向对话框添加图片框控件，在属性页中规定其 Type 属性为 Bitmap，并将其 Image 属性设为上面加入资源脚本文件中的位图 ID。这样，位图就被设置到图片框中了。

图片框支持的图像主要包括 Icon、Bitmap、Frame 和 Rectangle。

#### 5.3.2.9 进度条(Progress Control)

进度条用来标识执行一个长任务的进展情况，当任务执行完毕时，整个矩形槽被填满。其填充方式有两种：一种是平滑填充；另一种是矩形块填充(缺省)。具体通过其 Smooth 属性规定：True 指平滑填充，False 指矩形块填充。

进度条也是一个只供输出的控件，不接受用户输入，因此不支持任何回调。

进度条的范围由 DLG_RANGEMIN 和 DLG_RANGEMAX 控件索引给出，它们的值被限定在 0～65535 的范围内；当前位置则由 DLG_POSITION 控件索引设定。

#### 5.3.2.10 微调按钮(Spin Control)

微调按钮在 Windows 编程文档中称为 Up-down 控件，其外观像是一个微小的垂直滚动条。通常，微调按钮和其合作控件(编辑框、静态文本)联合使用，而在用户看来，微调按钮和其合作控件是一个控件。

在设计状态下，先添加合作(buddy)控件，后添加微调按钮，使微调按钮的 Tab 键次序紧接其合作控件的次序；再在微调按钮的属性页中，将其“Auto buddy”和“Set buddy integer”属性设为 True。这样，当用户点击微调按钮上、下箭头时，其当前值自动设置为合作控件的展示内容。

在程序中，通过 DLG_RANGEMIN、DLG_RANGEMAX 和 DLG_POSITION 控件索引，来分别设置或获取微调按钮的范围和当前值。

若要对微调按钮的行为进行控制，可为其注册 DLG_CHANGE 回调例程。当用户点击微调按钮上、下箭头时，程序自动执行该回调例程。

#### 5.3.2.11 滑动条(Slide Control)

滑动条在 Windows 编程文档中称为 Trackbar 控件。图 5-9 展示了两种形式的滑动条，(a)滑动条为默认情况，(b)滑动条的属性设置列于表 5-7。




图 5-9 滑动条的不同外观

表 5-7 图 5-9(b)滑动条的属性设置

| 属性项 | 属性值(缺省) | 属性项 | 属性值(缺省) |
|---|---|---|---|
| ID | IDC_SLIDER1 | Tick Marks | True(False) |
| Point | Top/Left(Both) | Auto Ticks | True(False) |

在程序中，由 DLG_RANGEMIN 和 DLG_RANGEMAX 控件索引设置或获取滑动条的滑动范围，而滑动条的当前值则由 DLG_POSITION 控件索引标识。另外，还可设置：

(1)用户按下箭头键时的位置变化量(DLG_SMALLSTEP)。

(2)用户按下 PAGE UP 和 PAGE DOWN 键或在滑槽内点击鼠标产生的位置变化量(DLG_BIGSTEP)。

(3)刻度标志的间隔频率(DLG_TICKFREQ)，缺省值为 10。

无论用户采取何种方式改变滑动条的当前值，程序都会执行注册的 DLG_CHANGE 类型的回调例程。

#### 5.3.2.12 选项卡(Tab Control)

如图 5-10 所示，选项卡控件允许将多个页面放置在窗体的同一区域上，既方便了用户操作又节省了空间。事实上，每个选项卡标签对应一个子窗口，当用鼠标点击标签时，相应的子窗口被展示，其他标签对应的子窗口则被隐藏起来。这样，除了要添加主窗口资源外，还要添加选项卡标签对应的子窗口资源。子窗口资源添加完毕，还须在其属性页进行如表 5-8 所示的设置。




图 5-10 选项卡应用示例

表 5-8 选项卡标签对应的子窗口设置

| 属性项 | 属性值 |
| --- | --- |
| Style | Child |
| Border | None |
| Title Bar | False |

初始化选项卡标签时，先规定选项卡的标签数(DLG_NUMITEMS),再设置每个标签对应的索引、标签名，最后将每个标签与其对话框相关联。如下列程序代码所示：

```
!Set initial Tabs
lret = DlgSet( gdlg , IDC_TAB , 3 )
lret = DlgSet( gdlg , IDC_TAB , "Family" , 1 )
lret = DlgSet( gdlg , IDC_TAB , "Style" , 2 )
lret = DlgSet( gdlg , IDC_TAB , "Size" , 3 )
lret = DlgSet( gdlg , IDC_TAB , IDD_TAB_DIALOG1 , 1 )
lret = DlgSet( gdlg , IDC_TAB , IDD_TAB_DIALOG2 , 2 )
lret = DlgSet( gdlg , IDC_TAB , IDD_TAB_DIALOG3 , 3 )
```

其中，gdlg 为主对话框，IDC_TAB 为选项卡控件，IDC_TAB_DIALOG1、IDC_TAB_DIALOG2 和 IDC_TAB_DIALOG3 依次为“Family”、“Style”和“Size”标签相关联的对话框。

选项卡控件初始化完毕，再对选项卡标签对应的对话框初始化。如下列程序代码所示：

```
!Initialize tab dialog boxes
lret = DlgInit( IDD_TAB_DIALOG1 , gdlg_tab1 )
lret = DlgSetSub( gdlg_tab1 , IDD_TAB_DIALOG1 , FamilySub )
lret = DlgSetSub( gdlg_tab1 , IDC_FAMILY_LIST , ChangeFamily )
lret = DlgInit( IDD_TAB_DIALOG2 , gdlg_tab2 )
lret = DlgSetSub( gdlg_tab2 , IDD_TAB_DIALOG2 , StyleSub )
lret = DlgSetSub( gdlg_tab2 , IDC_STYLE_LIST , UpdateFont )
lret = DlgInit( IDD_TAB_DIALOG3 , gdlg_tab3 )
lret = DlgSetSub( gdlg_tab3 , IDC_SIZE_LIST , UpdateFont )
```

其中，gdlg_tab1、gdlg_tab2 和 gdlg_tab3 为对话框变量，IDC_FAMILY_LIST、IDC_STYLE_LIST 和 IDC_SIZE_LIST 分别为三个对话框上的列表框，FamilySub、ChangeFamily、StyleSub 和 UpdateFont 为回调例程。

当显示主对话框时，须先调用 DlgModless 函数将所有的子对话框隐藏起来，再使用 DLG_STATE 控件索引设置初始显示的子对话框。如下列程序代码所示：

```
hwnd = GetDlgItem( gdlg%hwnd , IDC_TAB )
lret = DlgModless( gdlg_tab1 , SW_HIDE , hwnd )
lret = DlgModless( gdlg_tab2 , SW_HIDE , hwnd )
lret = DlgModless( gdlg_tab3 , SW_HIDE , hwnd )
lret = DlgSet( dlg , IDC_TAB , 1 , DLG_STATE )
```

使用完毕，释放对话框所占用的内存资源：

```
!Cleanup dialog box memory
Call DlgUnInit( gdlg )
Call DlgUnInit( gdlg_tab1 )
Call DlgUnInit( gdlg_tab2 )
Call DlgUnInit( gdlg_tab3 )
```

【例 5-2】对话框上布置有静态文本、编辑框、微调按钮、滑动条、滚动条和进度条，见图 5-11 所示。程序运行时，编辑框(微调按钮)、滑动条和滚动条任何一个控件的值发生改变，其余两个控件的值也随之改变，同时静态文本和进度条控件的显示值被刷新。




图 5-11 例 5-2 程序运行界面

*(1)插入对话框资源*

按例 5-1 的操作步骤，向一个 Fortran Console 工程中插入图 5-11 所示的对话框资源(窗体及其控件的属性设置如表 5-9)，并设置命令行工具 defToFD。

为使用户点击关闭按钮也能够退出对话框，在对话框上额外添加一个不可视的命令按钮，规定其 ID 为 IDCANCEL。

表 5-9 例 5-2 窗体及其控件的属性设置

| 窗体及其控件 | 属性类别 | 属性项 | 属性值 |
|---|---|---|---|
| 窗体 | 外观 | Caption | 设置温度 |
| | 杂项 | ID | IDD_THERM_DIALOG |
| 静态文本 1 | 外观 | Caption | 当前温度 |
| 静态文本 2 | 杂项 | ID | IDC_CURRENTVALUE |
| | 外观 | Caption | 0 |
| 编辑框 | 杂项 | ID | IDC_EDIT1 |
| 微调按钮 | 杂项 | ID | IDC_SPIN1 |
| | 行为 | Auto buddy | True |
| | | Set buddy integer | True |
| 滑动条 | 杂项 | ID | IDC_SLIDE1 |
| | 外观 | Point | Both |
| | | Tick Marks | True |
| | | Auto Ticks | True |
| 滚动条 | 杂项 | ID | IDC_SCROLLBAR1 |

| | | | |
|---|---|---|---|
| 进度条 | 杂项 | ID | IDC_PROGRESS1 |
| 命令按钮 | 杂项 | ID | IDCANCEL |
| | 行为 | Visuble | False |

*(2)操作对话框*

包括初始化对话框、设置对话框上的控件、展示模式对话框及释放对话框资源，如下列子程序 DoTherm 所示：

```
Program TEMP
  Implicit None
  Call DoTherm()
End Program TEMP

Subroutine DoTherm()
  Use IFLogM
  Implicit None
  Include 'Resource.FD'
  Type(Dialog)      dlg
  Logical*4         lret
  External UpdateTemp,ThermEdit
  !Initialize dialog box.
  lret = DlgInit( IDD_THERM_DIALOG , dlg )
  !Set up the controls and callbacks.
  lret = DlgSet( dlg , IDC_SPIN1 , 0 )
  lret = DlgSet( dlg , IDC_SPIN1 , 0 , DLG_RANGEMIN )
  lret = DlgSet( dlg , IDC_SPIN1 , 100 , DLG_RANGEMAX )
  lret = DlgSet( dlg , IDC_EDIT1 , '0' )
  lret = DlgSetSub( dlg , IDC_SPIN1 , UpdateTemp )
  lret = DlgSetSub( dlg , IDC_EDIT1 , ThermEdit , DLG_LOSEFOCUS )

  lret = DlgSet( dlg , IDC_SLIDER1 , 0 , DLG_RANGEMIN )
  lret = DlgSet( dlg , IDC_SLIDER1 , 100 , DLG_RANGEMAX )
  lret = DlgSet( dlg , IDC_SLIDER1 , 10 , DLG_TICKFREQ )
  lret = DlgSetSub( dlg , IDC_SLIDER1 , UpdateTemp )

  lret = DlgSet( dlg , IDC_SCROLLBAR1 , 0 , DLG_RANGEMIN )
  lret = DlgSet( dlg , IDC_SCROLLBAR1 , 109 , DLG_RANGEMAX )
  lret = DlgSet( dlg , IDC_SCROLLBAR1 , 10 , DLG_BIGSTEP )
  lret = DlgSetSub( dlg , IDC_SCROLLBAR1 , UpdateTemp )

  lret = DlgSet( dlg , IDC_PROGRESS1 , 0 )
  lret = DlgSet( dlg , IDC_PROGRESS1 , 0 , DLG_RANGEMIN )
  lret = DlgSet( dlg , IDC_PROGRESS1 , 100 , DLG_RANGEMAX )

  !Active the modal dialog.
```

```
  lret = DlgModal( dlg )
  !Cleanup dialog box memory
  Call DlgUnInit( dlg )
End Subroutine DoTherm
```

其中，滚动条的最大值(DLG_RANGEMAX)设为 109。其计算公式如下：

MaxRangeValue = MaxScrollPos + ( PageSize - 1 )

当中的 MaxScrollPos 指滑块位置(DLG_POSITION)最大值，该实例中为 100；PageSize 指在滑槽中点击鼠标或按下 PageUp、PageDown 键时滑块位置移动量(DLG_BIGSTEP)，该实例中设为 10；最后得出 MaxRangeValue 的值为 109。这样，当滑块移至右端时，滚动条的值恰好为 100。

程序为微调按钮、滑动条和滚动条控件注册了回调类型为 DLG_CHANGE(缺省)的回调例程 UpdateTemp，为编辑框注册了回调例程为 DLG_LOSTFOCUS 的回调例程 ThermEdit。

*(3)编写控件回调例程*

UpdateTemp 回调例程的实现代码为：

```
Subroutine UpdateTemp(dlg,control_Name,callBackType)
  Use IFLogM
  Implicit None
  Include 'Resource.FD'
  Type(dialog) dlg
  Integer control_Name,callBackType
  Logical*4 status
  Integer pos
  Character*3 textPos
  status = DlgGet( dlg , control_Name , pos )
  Write(textPos,"(I3)") pos
  Select Case ( control_Name )
    Case ( IDC_SPIN1 )
      status = DlgSet( dlg , IDC_SLIDER1 , pos )
      status = DlgSet( dlg , IDC_SCROLLBAR1 , pos )
    Case ( IDC_SLIDER1 )
      status = DlgSet( dlg , IDC_SPIN1 , pos )
      status = DlgSet( dlg , IDC_SCROLLBAR1 , pos )
    Case ( IDC_SCROLLBAR1 )
      status = DlgSet( dlg , IDC_SLIDER1 , pos )
      status = DlgSet( dlg , IDC_SPIN1 , pos )
  End Select
  status = DlgSet( dlg , IDC_CURRENTVALUE , textPos )
  status = DlgSet( dlg , IDC_PROGRESS1 , pos )
End Subroutine UpdateTemp
```

该例程当微调按钮、滑动条和滚动条控件值发生改变，引发 DLG_CHANGE 类型的回调时被执行，它依据当前控件值刷新其他控件值，以使所有的数据输入、输出控件保持同步。

ThermEdit 回调例程的实现代码为：

```
Subroutine ThermEdit(dlg,id,callBackType)
  Use IFLogM
  Implicit None
  Include 'Resource.FD'
  Type(dialog) dlg
  Integer id,callBackType
  Logical*4 status
  Integer pos
  Character*3 textPos

  If ( callBackType == DLG_LOSEFOCUS ) then
    !Set the values of other controls to match the change
    status = DlgGet( dlg ,id , textPos )
    Read(textPos , "(I3)") pos
  End If
  status = DlgSet( dlg , IDC_CURRENTVALUE , textPos )
  status = DlgSet( dlg , IDC_SLIDER1 , pos )
  status = DlgSet( dlg , IDC_SPIN1 , pos )
  status = DlgSet( dlg , IDC_SCROLLBAR1 , pos )
  status = DlgSet( dlg , IDC_PROGRESS1 , pos )
End Subroutine ThermEdit
```

该例程当用户通过键盘直接在编辑框中输入数据，并引发 DLG_LOSEFOCUS 类型的回调(输入焦点移至别处)时被执行，它依据编辑框的值更新其他控件值。

【例 5-3】在第 4 章例 4-12 中的“绘图”菜单下增加“设置绘图参数”菜单项，当用户选取该菜单项时，暂时“设置图形范围”对话框，如图 5-12 所示。用户通过该对话框调整绘图范围及绘制曲线所用线段数。




图 5-12 例 5-3 展示的对话框

*(1)插入对话框资源*

向例 4-12 的 QuickWin 工程中插入图 5-12 所示的对话框资源，对话框及其控件的属性设置见表 5-10，并设置命令行工具 defToFD。

表 5-10 例 5-3 窗体及其控件的属性设置

| 窗体及其控件 | 属性类别 | 属性项 | 属性值 |
|---|---|---|---|
| 窗体 | 外观 | Caption | 设置图形范围 |

| | | | |
|---|---|---|---|
| | 杂项 | ID | IDD_RANGE_DIALOG |
| 静态文本 1 | 外观 | Caption | X 方向最小值 |
| 编辑框 1 | 杂项 | ID | IDC_XMIN |
| 静态文本 2 | 外观 | Caption | X 方向最大值 |
| 编辑框 2 | 杂项 | ID | IDC_XMAX |
| 静态文本 3 | 外观 | Caption | Y 方向最小值 |
| 编辑框 3 | 杂项 | ID | IDC_YMIN |
| 静态文本 4 | 外观 | Caption | Y 方向最大值 |
| 编辑框 4 | 杂项 | ID | IDC_YMAX |
| 静态文本 5 | 外观 | Caption | 绘图线段数 |
| 编辑框 5 | 杂项 | ID | IDC_LINES |
| 命令按钮 1 | 外观 | Caption | 确定 |
| | 杂项 | ID | IDOK |
| 命令按钮 2 | 外观 | Caption | 取消 |
| | 杂项 | ID | IDCANCEL |

*(2)追加菜单项*

在 QuickWin 的初始化函数 InitialSettings 中增加下列语句：

```
i4 = AppendMenuQQ( 3 , $MENUSEPARATOR , ''C , NUL )
i4 = AppendMenuQQ( 3 , $MENUENABLED , '设置绘图范围(&R)'C , SetRange )
```

这里，为“设置绘图范围”菜单项注册了菜单回调例程 SetRange。追加的菜单项如图 5-13 所示。

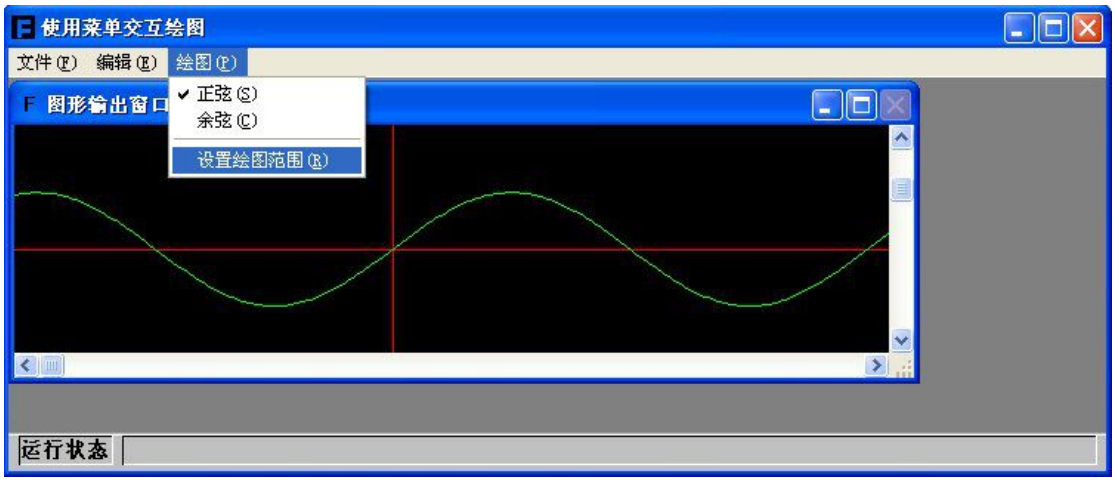



图 5-13 例 5-3 程序运行主界面

*(3)编写菜单回调例程*

“设置绘图范围”菜单项的回调例程 SetRange 包含在新增加的模块 LastMod 中：

```
Module LastMod          !设置绘图范围模块
  Use NewPlotMod        !以前的绘图模块
  Implicit None
  Contains
    Subroutine SetRange()     !绘图范围菜单项的回调例程
      Use IFLogM
      Include 'Resource.FD'   !系统生成的对话窗信息(控件ID)
```

```
    Type(dialog)::dl
    Integer::rlt
    Character(Len=20)::str
    rlt = DlgInit( IDD_RANGE_DIALOG , dl )     !实例化对话框对象
    !将实型变量值转换成字符串，然后设为文本框中的值
    Write(str,'(f6.2)') X_Start
    rlt = DlgSet( dl , IDC_XMIN , str )
    Write(str,'(f6.2)') X_End
    rlt = DlgSet( dl , IDC_XMAX , str )
    Write(str,'(f6.2)') Y_Bottom
    rlt = DlgSet( dl , IDC_YMIN , str )
    Write(str,'(f6.2)') Y_Top
    rlt = DlgSet( dl , IDC_YMAX , str )
    Write(str,'(I5)') Lines
    rlt = DlgSet( dl , IDC_LINES , str )

    rlt = DlgModal( dl )      !展示模式对话框

    If ( rlt == IDOK ) then
      !由字符串转成数值，再获取数值型变量的值
      rlt = DlgGet( dl , IDC_XMIN , str )
      Read(str,*) X_Start
      rlt = DlgGet( dl , IDC_XMAX , str )
      Read(str,*) X_End
      rlt = DlgGet( dl , IDC_YMIN , str )
      Read(str,*) Y_Bottom
      rlt = DlgGet( dl , IDC_YMAX , str )
      Read(str,*) Y_Top
      rlt = DlgGet( dl , IDC_LINES , str )
      Read(str,*) Lines
    End If
  End Subroutine SetRange
End Module
```

当用户设置完对话框并点击“确定”按钮后，程序更新相应的模块变量 X_Start、X_End、Y_Buttom、Y_Top 及 Lines，下次绘图时就会使用这些新的数据，使图形展示新的缩放效果及光滑度。

(4)NewPlotMod 模块替换为 LastMod

最后，将原来主程序引用的操作菜单、进行绘图的模块由 NewPlotMod 替换为 LastMod，而在现在的模块 LastMod 中包含原来模块 NewPlotMod，如上列代码所示。

# 小　结

■对话框是 Windows 应用程序与用户进行交互的重要手段。对话框分为模式对话框和无模式对话框。Intel Fortran 所有类型的应该程序都支持模式对话框，而无模式对话框通常只在 Fortran Windows 程序中使用。

■在设计状态下，需要在资源编辑器中对插入的对话框进行编辑，包括添加控件、布置对话框、设置对话框及其控件属性，以及保存对话框资源。

■在运行状态下，则需引用对话框，初始化对话框及其控件(包括注册控件回调例程)，展示模式对话框，释放对话框所占内存资源。

■对话框程序设计，主要精力花在编写控件回调例程上。当某种类型的回调发生时，为控件注册的该类型的回调例程被执行。

■命令按钮的默认行为是：设置展示模式对话框函数的返回值为被点击的命令按钮名，然后退出对话框。点击对话框关闭按钮等同于点击 ID 号为 IDCANCEL 的命令按钮，若没有这样的命令按钮存在，点击关闭按钮对话框不会退出。

■控件索引是标识控件属性和回调的变量，它分为整型、逻辑型、字符型和外部例程型。通常，使用一般函数 DlgSet 和 DlgGet 来分别设置和获取控件值。

要为控件注册回调例程，须使用 DlgSetSub 函数。

# 第 6 章　设计 Windows 应用程序

典型的 Windows 应用程序，在外观上呈现窗口、菜单等图形用户界面，在内部采取消息驱动的工作机制，程序的运行由用户通过图形界面来控制。目前开发的应用程序大多是 Windows 风格的应用程序。

利用 Intel Visual Fortran 9.0 程序向导，可以开发 3 种类型的 Windows 应用程序：无模式对话框、单文档(SDI)及多文档(MDI)应用程序。

**本章主要内容：**

- Windows 应用程序结构
- 无模式对话框应用程序
- SDI 程序中的无模式对话框、菜单及绘图

## 6.1 Windows 应用程序的结构

使用 Intel Visual Fortran 开发 Windows 程序，了解 Windows 程序的结构是必要的。一个 Windows 程序源程序一般由头文件、源文件、资源文件等几部分组成。下面先对用 C 语言编写的 Windows 程序的源文件作一简单介绍。

### 6.1.1　主函数

为了与 DOS C 程序的主函数名(main)相区别，Windows 程序的主函数名为 WinMain。其函数原型声明为：

```
int APIENTRY WinMain(HINSTANCE hInstance, //当前应用程序实例句柄
                     HINSTANCE hPrevInstance,  //前一个应用程序实例句柄
                     LPTSTR lpCmdline,  //指向本程序命令行的指针
                    int     nCmdShow);   //决定程序窗口显示方式的标志
```

主函数主要完成两个任务：一是创建应用程序的界面——窗口；二是建立消息循环。创建应用程序窗口要用到 4 个 API 函数(RegisterClass、CreateWindow、ShowWindow 和 UpdateWindow)；建立消息循环要用到 3 个 API 函数(GetMessage、TranslateMessage 和 DispatchMessage)。

#### 6.1.1.1 描述窗口属性的数据结构——窗口类

Windows 要求在创建应用程序窗口前，须定义诸如样式、背景颜色、窗口图标等一些窗口相关的属性，这些属性用一个称为窗口类的结构体来描述。该结构体的声明为：

```
typedef struct tag WNDCLASS {
    UINT    style;                      //窗口样式
    WNDPROC lpfnWndProc;                //指向窗口函数的指针
    int     cbClsExtra;                 //分配在窗口类结构后的字节数
    int     cbWndExtra;                 //分配在窗口实例后的字节数
    HANDLE  hInstance;                  //应用程序实例句柄
    HICON   hIcon;                      //窗口图标
```

```
    HCURSOR hCursor;                    //窗口光标
    HBRUSH  hbrBackground;              //窗口背景颜色
    LPCTSTR lpszMenuName;               //窗口菜单资源名
    LPCTSTR lpszClassName;              //窗口类名
} WNDCLASS;
```

其中，第二个成员 lpfnWndProc 是函数指针，系统使用该指针调用窗口函数来处理消息。

程序设计人员首先定义一个窗口类结构体变量，并根据需要给结构体的各个成员赋值。

#### 6.1.1.2 注册窗口类

在创建窗口前，要用窗口类注册函数 RegisterClass 向系统登记定义好的窗口属性。函数 RegisterClass 的原型为：

```
BOOL RegisterClass(WNDCLASS *lpWndClass);
```

其中的参数 lpWndClass 是指向窗口类结构体的指针。

一旦使用 RegisterClass 注册成功，系统中就有了一个以窗口类最后一个成员(lpszClassName)命名的窗口类。

#### 6.1.1.3 创建窗口

窗口类注册成功后，就可以使用 CreateWindow 函数创建窗口了。该函数的原型为：

```
HWND CreateWindow(
    LPCTSTR     lpClassName,        //窗口类名
    LPCTSTR     lpWindowName,       //窗口标题
    DWORD       dwStyle,            //窗口风格
    int         x,                  //窗口左上角x坐标
    int         y,                  //窗口左上角y坐标
    int         nWidth,             //窗口宽度
    int         nHeight,            //窗口高度
    HWND        hWndParent,         //父窗口句柄
    HMENU       hMenu,              //菜单句柄
    HANDLE      hInstance,          //应用程序实例句柄
    LPVOID      lpParam             //该值为NULL
);
```

窗口创建成功后，返回创建的窗口句柄。

#### 6.1.1.4 显示窗口

调用 CreatWindow 函数在内存中创建窗口后，还需使用下列两个函数将窗口显示在计算机屏幕上：

```
BOOL ShowWindow(
    HWND    hWnd,           //窗口句柄
    int     nCmdShow        //窗口的显示方式
);
BOOL UpdateWindow(
    HWND    hWnd            // 窗口句柄
);
```

#### 6.1.1.5 消息循环

窗口一旦创建并显示，应用程序的初始化工作即告完成，随后程序进入等待接收消息状态(利用一个 while 循环)。

事实上，操作系统为应用程序建立一个消息队列，在程序运行过程中如果发生一个事件，Windows 就会把这个事件所对应的消息送入消息队列。应用程序为取得消息队列中的消息，需调用 Windows API 函数 GetMessage，并利用这个函数的返回值组织一个循环来不断获取消息。一旦获取消息，就把这个消息派送给系统。这个消息称为消息循环，其代码为：

```
while ( GetMessage(&msg,NULL,NULL,NULL) )
{
    TranslateMessage(&msg);    //把键盘消息翻译成字符消息
    DispatchMessage(&msg);     //把消息派送给系统
}
```

当消息循环把消息派送给系统后，系统就会以消息作为参数来调用窗口函数，并以消息参数中的 message 成员为依据查找并执行该消息对应的程序段。在处理完与消息对应的程序段后，只要该消息不时终止应用程序的消息，程序控制就会返回到消息循环中，以等待获取下一条消息。这就是 Windows 应用程序的消息驱动(或事件驱动)机制。

Windows 中的消息结构定义为：

```
typedef struct tagMSG {
    HWND        hwnd;          //被检索消息的窗口句柄
    UINT        message;       //消失标识符
    WPARAM      wParam;        //消息的附加信息
    LPARAM      lParam;        //消息的附加信息
    DWORD       time;          //消息进入消息队列的时刻
    POINT       pt;            //发送消息时的光标位置
} MSG;
```

部分常用的 Windows 消息列于表 6-1。

表 6-1 常用的 Windows 消息

| 消息标识 | 描述 |
|---|---|
| WM_LBUTTONDOWN | 按下鼠标左键时产生的消息 |
| WM_LBUTTONUP | 松开鼠标左键时产生的消息 |
| WM_RBUTTONDOWN | 按下鼠标右键时产生的消息 |
| WM_RBUTTONUP | 松开鼠标右键时产生的消息 |
| WM_LBUTTONDBLCLK | 双击鼠标左键时产生的消息 |
| WM_RBUTTONDBLCLK | 双击鼠标右键时产生的消息 |
| WM_CHAR | 按下非系统键时产生的消息。其中，wParam 为键的 ASCII 码 |
| WM_CREATE | 由 CreateWindow 函数产生的消息 |
| WM_CLOSE | 关闭窗口时产生的消息 |
| WM_DESTROY | 消除窗口式产生的消息 |
| WM_QUIT | 退出程序时，由 PostQuitMessage 函数产生的消息 |
| WM_PAINT | 需要窗口重画时产生的消息 |

### 6.1.2 窗口函数

具有窗口界面的 Windows 应用程序，必须要有一个“窗口函数”，各种消息的处理都是

在这里实现的，它是完成用户任务的核心，也是需要程序设计人员编写大量代码的地方。该函数的原型为：

```
LRESULT CALLBACK WndProc(
    HWND      hWnd,              //派送消息的窗口句柄
    UINT      message,           //系统传递来的消息标识
    WPARAM    wParam,            //消息的附加信息
    LPARAM    lParam             //消息的附加信息
)
```

在主函数中定义窗口类时，必须把这个窗口函数的名称赋给 WNDCLASS 结构的 lpfnWndProc 成员，因为在窗口响应消息时，系统就是按照 lpfnWndProc 的指示去调用窗口函数的。

一个最简单的窗口函数代码如下：

```
LRESULT CALLBACK WndProc(HWND hWnd,UINT message,WPARAM wParam,LPARAM lParam)
{
  switch(message)
  {
  case WM_DESTROY:
    PostQuitMessage(0);
    break;
  default:
    return DefWindowProc(hWnd,message,wParam,lParam);
  }
  return 0;
}
```

从函数的参数列表中可以看到，当系统调用该函数时除了要传递的窗口函数相关的窗口句柄外，还要传递消息号 message 和消息附加信息 wParam 和 lParam，以便需要时在函数中使用这些参数。

从上列代码可以看到，窗口函数的主体是一个 switch-case 选择结构，每一个 case 对应一个消息处理程序。也就是说，函数是根据消息号 message 转向相应的消息处理程序段的。

在上面的窗口函数中只对一个消息 WM_DESTROY 进行了处理，WM_DESTROY 是在关闭窗口时产生的消息，在其程序段中调用了 API 函数 PostQuitMessage，该函数在关闭窗口时做一些善后处理工作。而在 default 程序段中调用了 DefWindowProc 函数，该函数是系统提供的窗口消息处理函数，它使系统对程序没有处理的消息进行默认处理，以保证所有发送到窗口的消息均能得到合适的处理。

程序设计人员可以在窗口函数体内添加任何消息处理过程，以使应用程序能够响应消息，并在消息处理过程中完成用户所要求的任务。

### 6.1.3 Windows 系统、主函数、窗口函数之间的关系

理解 Windows 系统、主函数、窗口函数这三者之间的关系，对于编写 Windows 程序来讲是极为重要的。

主函数和窗口函数都是由 Windows 系统调用的函数，只不过主函数是在程序启动后，系统首先调用的函数，故主函数又称为应用程序的入口点；而窗口函数则是主函数在获得消息并把消息发送给系统后，由系统调用的函数。

不同消息对应的操作是由窗口函数完成的，Windows 应用程序设计的主要工作是编写窗口函数中的 case 结构代码。

Windows 系统、主函数、窗口函数三者之间的关系如图 6-1 所示。

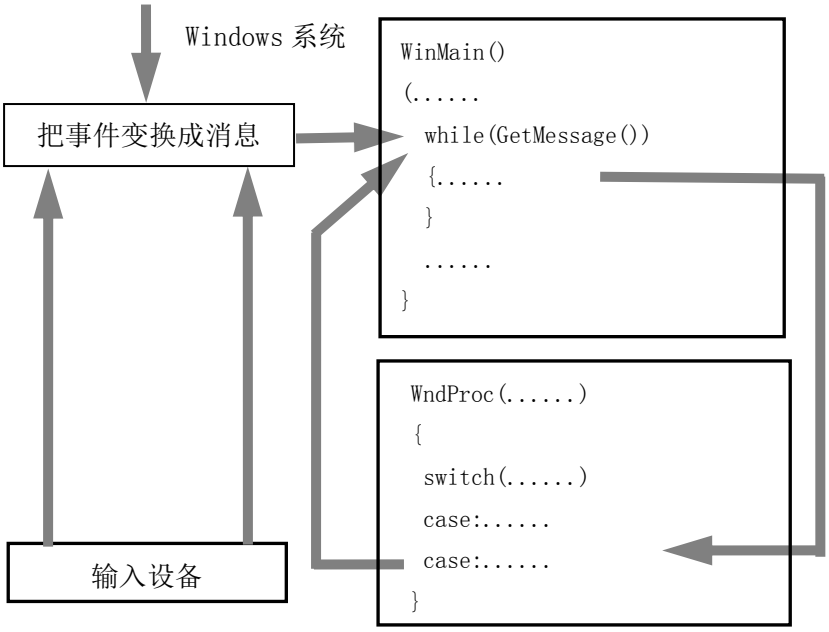



图 6-1 Windows 系统、主函数、窗口函数三者之间的关系

【例 6-1】用 C 语言编写一个简单的 Windows 程序，当用鼠标左键单击程序窗口的客户区时，计算机的扬声器发出“叮”的声音。

程序代码如下：

```
#include <windows.h>

//声明窗口函数原型
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);

//主函数
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE PreInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow)
{
  HWND hwnd;
  MSG  msg;
  TCHAR lpszClassName[3] ;
  TCHAR szWindowsTitle[3] ;
  WNDCLASS wc;
  wc.style = 0 ;
  wc.lpfnWndProc = WndProc ;
  wc.cbClsExtra = 0 ;
  wc.cbWndExtra = 0 ;
```

```
    wc.hInstance = hInstance ;
    wc.hIcon = LoadIcon( NULL , IDI_APPLICATION ) ;
    wc.hCursor = LoadCursor( NULL , IDC_ARROW ) ;
    wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH) ;
    wc.lpszMenuName = NULL ;
    wc.lpszClassName = lpszClassName ;

    RegisterClass(&wc);

    hwnd = CreateWindow(lpszClassName,
                        szWindowsTitle,
                        WS_OVERLAPPEDWINDOW,
                        120,50,800,600,
                        NULL,
                        NULL,
                        hInstance,
                        NULL);
    ShowWindow(hwnd,nCmdShow);
    UpdateWindow(hwnd);

    while(GetMessage(&msg,NULL,0,0))
    {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
    return (int) msg.wParam;
}

//处理消息的窗口函数
LRESULT CALLBACK WndProc(HWND hwnd,
                         UINT message,
                         WPARAM wParam,
                         LPARAM lParam)
{
    switch(message)
    {
    case WM_LBUTTONDOWN:
      {
        MessageBeep(0);
      }
    case WM_DESTROY:
      PostQuitMessage(0);
      break;
    default:
```

```
    return DefWindowProc(hwnd,message,wParam,lParam);
  }
  return 0;
}
```

从程序中可以看到，主函数共做了 3 件事情：注册窗口类、创建并显示窗口和执行消息循环。在窗口函数中编写了一段鼠标左键按下时的处理代码：调用 API 函数 MessageBeep，这样，程序启动后如果用户在客户区按下鼠标左键，计算机的扬声器就会发出“叮”的声音。

在程序中定义 WNDCLASS 结构变量 wc 时，还调用了 3 个 API 函数：LoadIcon、LoadCursor 和 GetStockObject，它们分别用来装载鼠标、图标资源和获取设置窗口客户区背景颜色的系统画刷。

### 6.1.4 Fortran Windows 源程序的结构

不论是基于对话框的还是单文档(SDI)、多文档(MDI)应用程序，利用 Intel Visual Fortran 9.0 向导开发的 Windows 程序结构都是相同的，如表 6-2 所示。

它们的不同体现在文档内容上，比如在主程序(WinMain)中，单文档、多文档程序包含注册窗口类、创建窗口及显示窗口的代码，而对话框程序使用现成的窗口资源，因此不包含创建窗口的代码；在资源文件中，对话框程序包含对话框资源，而单文档、多文档程序则包含菜单资源。

表 6-2 Fortran Windows 程序中包含的文件(FWApp 为工程名)

| 文件 | 描述 |
|---|---|
| FWApp.F90 | Fortran Windows 主程序源码文件，它包含程序入口点 |
| FWAppGlobals.F90 | 文件中的模块(包含模块中全局变量声明) |
| FWApp.FI | Fortran 包含文件，当中包含 Windows 例程接口声明 |
| FWApp.Rc | Windows 资源文件(程序主资源文件) |
| Resource.h(Resource.FD) | 定义资源标识符(ID)的头文件 |
| FWApp.Ico | 包含在主资源文件中的图标文件 |

## 6.2 无模式对话框应用程序

利用 Intel Fortran 工程向导，设计基于对话框的 Windows 应用程序，实际上是设计无模式对话框 Windows 程序。模式对话框可以垄断所有的用户消息，因此只有在关闭模式对话框之后，用户才能进行其他工作；无模式对话框没有自己独立的消息循环，而是与主程序使用同一个消息循环，所以它不能垄断用户消息。因此，用户在打开无模式对话框时，仍可在应用程序的其他窗口中工作。

本节结合实例讲解无模式对话框的 Windows 程序设计。

【例 6-2】将第 5 章例 5-2 的模式对话框设计为无模式对话框，并将进度条放置在单独的对话框中，来模拟红色“温度计”，见图 6-2。当点击主对话框中的“应用”按钮时，“温度计”指示器会上升或下降，以和以前的温度值保持一致。




图 6-2 无模式对话框 Windows 程序示例

*(1)利用向导生成工程骨架*

在“新建项目”对话框中，选择“Windows Application”工程类型；在随后出现的应用设置对话框中，选择“Dialog-Based sample code”应用类型，如图 6-3 所示。

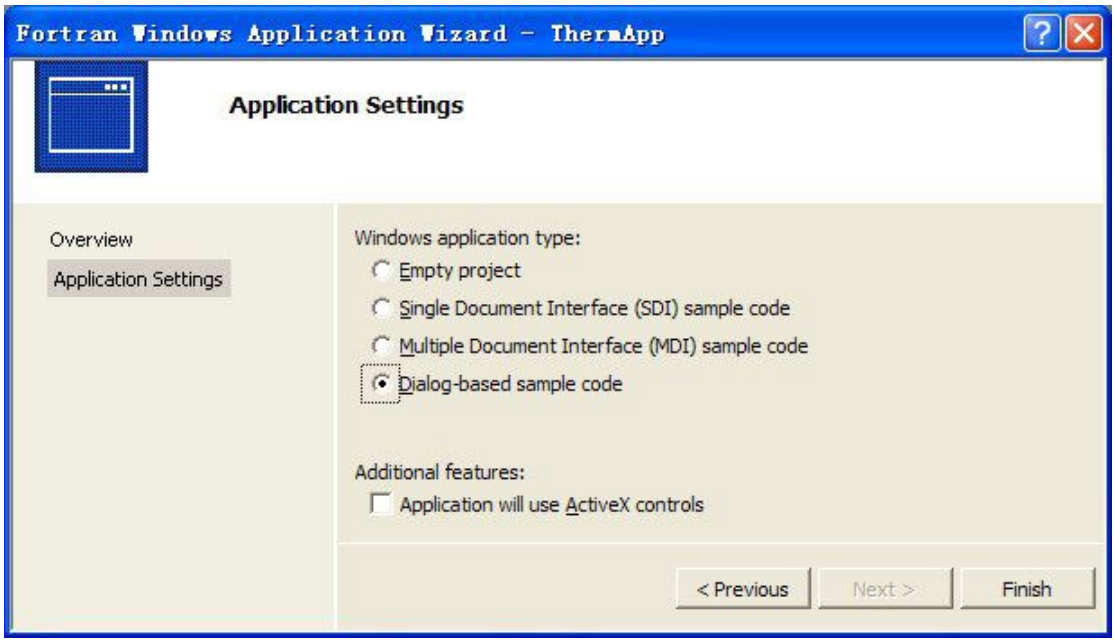



图 6-3 Fortran Windows 向导的程序设置窗口




图 6-4 基本的对话框程序界面

①主函数

向导生成的主函数骨架代码为：（已删除注释行）

```
integer*4 function WinMain( hInstance, hPrevInstance, lpszCmdLine, nCmdShow )
!DEC$ IF DEFINED(_X86_)
!DEC$ ATTRIBUTES STDCALL, ALIAS : '_WinMain@16' :: WinMain
!DEC$ ELSE
!DEC$ ATTRIBUTES STDCALL, ALIAS : 'WinMain' :: WinMain
!DEC$ ENDIF

    use user32
    use kernel32
    use iflogm
    use ThermGlobals

    implicit none

    integer*4 hInstance
    integer*4 hPrevInstance
    integer*4 lpszCmdLine
    integer*4 nCmdShow

    include 'resource.fd'

    external ThermSub
    external ThermApply

    ! Variables
    type (T_MSG)            mesg
    integer*4               ret
    logical*4               lret


    ghInstance = hInstance
    ghModule = GetModuleHandle(NULL)
    ghwndMain = NULL

    lret = DlgInit(IDD_Therm_DIALOG, gdlg)
    if (lret == .FALSE.) goto 99999
    lret = DlgSetSub(gdlg, IDD_Therm_DIALOG, ThermSub)
    lret = DlgSetSub(gdlg, IDM_APPLY, ThermApply)
    lret = DlgModeless(gdlg, nCmdShow)
    if (lret == .FALSE.) goto 99999

    ! Read and process messsages
```

```
    do while( GetMessage (mesg, NULL, 0, 0) )
       if ( DlgIsDlgMessage(mesg) .EQV. .FALSE. ) then
           lret = TranslateMessage( mesg )
           ret  = DispatchMessage( mesg )
       end if
    end do
    call DlgUninit(gdlg)

    WinMain = mesg.wParam
    return

99999 &

    ret = MessageBox(ghwndMain, "Error initializing application Therm"C, &
                     "Error"C, MB_OK)
    WinMain = 0

end
```

主函数开头部分的条件编译语句，规定了 32 位系统下主函数使用的调用约定为 STDCALL，产生的目标代码函数名为_WinMain@16。

与 Windows API 分布在一系列 DLL 中的情况相对应，Intel Fortran 采用一系列同名模块对 Windows API 重新进行了封装，并将它们统一包含在 IFWin 模块中。这里只引用了相关模块 User32 和 Kernel32，以减少产生的应用程序执行文件(EXE)的大小。(事实上，录入人个人认为执行文件大小并不会减少，仅仅是加快了编译链接的速度,进一步加快速度的办法是使用 Use 语句的 Only 修饰词)

T_MSG 为 Windows 消息派生类型，它对应 MSG 消息结构类型。

主函数中，调用 DlgModeless 函数来显示无模式对话框。在消息循环中，调用函数 DlgIsDlgMessage 判断当前消息是否来自无模式对话框，如果是(该函数返回逻辑假)，就需要对来自无模式对话框的消息做进一步处理。

```
    do while( GetMessage (mesg, NULL, 0, 0) )
       if ( DlgIsDlgMessage(mesg) .EQV. .FALSE. ) then
           lret = TranslateMessage( mesg )
           ret  = DispatchMessage( mesg )
       end if
    end do
```

**②对话框、按钮回调例程**

主函数中。分别为对话框 IDD_THERM_DIALOG 和命令按钮 IDM_APPLY 注册了 ThermSub 和 ThermApply 回调例程。其中，ThermSub 回调例程的实现代码为：

```
SUBROUTINE ThermSub( dlg, id, callbacktype )
!DEC$ ATTRIBUTES DEFAULT :: ThermSub
  use user32
  use iflogm
  implicit none
  type (dialog) dlg
  integer id, callbacktype
  if (callbacktype == dlg_destroy) then
    call PostQuitMessage(0)
  endif
END SUBROUTINE ThermSub
```

该回调例程是在初始化(DLG_INIT)或销毁(DLG_DESTROY)对话框时被调用。

事实上，对话框上所有的命令按钮都有一个缺省的回调例程(DLG_CLICKED)，其默认行为是引发对话框的 DLG_DESTROY 回调，并调用 DlgExit 函数退出对话框。此处，相当于注册了“Exit”按钮的回调例程。它调用 API 函数 PostQuitMessage，发出 WM_QUIT 的 Windows 消息，来结束程序运行。

对话框关闭按钮执行的是 ID 号为 IDCANCEL 的命令按钮的回调例程，由于对话框上没有 IDCANCEL 命令按钮，所以点击关闭按钮并不退出对话框。

ThermApply 回调例程的代码骨架为：

```
SUBROUTINE ThermApply( dlg, id, callbacktype )
!DEC$ ATTRIBUTES DEFAULT :: ThermApply
  use iflogm
  implicit none
  type (dialog) dlg
  integer id, callbacktype
  if (callbacktype == dlg_clicked) then
    ! TO DO; Add your APPLY logic here
  endif
END SUBROUTINE ThermApply
```

该例程被注册了“Apply”命令按钮的回调例程。这里，只提供了发生 DLG_CLICKED 回调时执行的代码骨架，需要用户填充具体的逻辑代码。

**③全局变量模块**

向导声明的全局变量放置在下列模块中：

```
module ThermGlobals
  use iflogm
  implicit none
  !  Parameters
  integer*4, parameter, public :: SIZEOFAPPNAME = 100
  !  Global data
  integer     ghInstance
  integer     ghModule
  integer     ghwndMain
  type (dialog) gdlg
end module
```

*(2)重新布置对话框*

该实例需重新布置向导生成的对话框，并添加一新的“温度计”对话框资源。现将调整后的对话框及其控件的属性设置列于表 6-3(“温度计”对话框中额外增加了一个图标)。

表 6-3 例 6-2 的对话框及其控件属性设置

| 窗体及其控件 | 属性类别 | 属性项 | 属性值 |
|---|---|---|---|
| 窗体 2 | 外观 | Caption | T |
| | 杂项 | ID | IDD_THERMOMETER |
| 命令按钮 1 | 外观 | Caption | 应用(&A) |
| | 杂项 | ID | IDM_APPLY |
| 命令按钮 2 | 外观 | Caption | 退出(&E) |
| | 杂项 | ID | IDM_EXIT |

注：表中只列出了与第 5 章例 5-2 不同的控件属性

向导以自动设置了命令行的 defToFD 工具，将资源文件 resource.h 转换成对应的 Fortran 头文件 Resource.FD，如表 6-4 所示。当资源中定义的符号常量发生改变时，头文件 resource.h 和 Resource.FD 会自动更新，以保持一致。

表 6-4 向导设置的命令行 defToFD

| 选项 | 属性设置 |
|---|---|
| Command Line | deftofd "$(InputPath)" "$(InputDir)$(InputName).fd" |
| Description | Generating Fortran include file... |
| Outputs | $(InputDir)$(InputName).fd |

*(3)完善程序代码*

**①声明对话框全局变量**

作为参数传递到 DlgModeless 函数中的对话框变量，须在对话框生命周期内一直保留在内存中，因此在向导生成的模块文件中声明无模式“温度计”对话框变量：

```
Type (dialog) dlg_thermometer
```

**②扩充主函数**

主函数要扩充的代码主要是声明回调例程，初始化两个对话框及其控件，设置回调例程及调用 DlgModeless 函数展示“温度计”对话框。因为两个无模式对话框共用应用程序的主消息循环，所以无须改变原来的消息循环。主函数添加的代码如下所列：

```
......
External ThermEdit
External UpdateTemp
External ThermometerSub

......
!Set up the controls of the main dialog
lret = DlgSet( gdlg , IDC_CURRENTVALUE , '0' )
lret = DlgSet( gdlg , IDC_SPIN1 , 0 )
lret = DlgSet( gdlg , IDC_SPIN1 , 0 , DLG_RANGEMIN )
lret = DlgSet( gdlg , IDC_SPIN1 , 100 , DLG_RANGEMAX )
```

```
  lret = DlgSet( gdlg , IDC_EDIT1 , '0' )
  lret = DlgSetSub( gdlg , IDC_SPIN1 , UpdateTemp )
  lret = DlgSetSub( gdlg , IDC_EDIT1 , ThermEdit , DLG_LOSEFOCUS )
  lret = DlgSet( gdlg , IDC_SLIDER1 , 0 , DLG_RANGEMIN )
  lret = DlgSet( gdlg , IDC_SLIDER1 , 100 , DLG_RANGEMAX )
  lret = DlgSet( gdlg , IDC_SLIDER1 , 10 , DLG_TICKFREQ )
  lret = DlgSetSub( gdlg , IDC_SLIDER1 , UpdateTemp )

  lret = DlgSet( gdlg , IDC_SCROLLBAR1 , 0 , DLG_RANGEMIN )
  lret = DlgSet( gdlg , IDC_SCROLLBAR1 , 109 , DLG_RANGEMAX )
  lret = DlgSet( gdlg , IDC_SCROLLBAR1 , 10 , DLG_BIGSTEP )
  lret = DlgSetSub( gdlg , IDC_SCROLLBAR1 , UpdateTemp )
  ......
  lret = DlgModeless( gdlg , nCmdShow )
  if (lret == .FALSE.) goto 99999
  ......
  !Create the thermometer dialog box and set up the controls
  lret = DlgInit( IDD_THERMOMETER , dlg_thermometer )
  lret = DlgSet( dlg_thermometer , IDC_PROGRESS1 , 0 )
  lret = DlgSet( dlg_thermometer , IDC_PROGRESS1 , 0 , DLG_RANGEMIN )
  lret = DlgSet( dlg_thermometer , IDC_PROGRESS1 , 100 , DLG_RANGEMAX )
  lret = DlgSetSub( dlg_thermometer , IDD_THERMOMETER , ThermometerSub )
  lret = DlgModeless( dlg_thermometer , nCmdShow )
  if (lret == .FALSE.) goto 99999
  ......
```

**③完善“应用”按钮回调例程**

向导生成的“应用”按钮回调例程 ThermApply 只有一个代码骨架，此处添加从主对话框控件上获取当前温度，设置“温度计”温度的处理代码：

```
SUBROUTINE ThermApply( dlg, id, callbacktype )
!DEC$ ATTRIBUTES DEFAULT :: ThermApply
  use iflogm
  Use ThermGlobals,Only:dlg_thermometer
  implicit none
  type (dialog) dlg
  integer id, callbacktype
  Integer status,pos
  Include 'Resource.FD'
  If ( CallBackType == DLG_CLICKED ) then
    !Set the value of the thermometer to match the controls
    status = DlgGet( dlg , IDC_SPIN1 , pos )
    status = DlgSet( dlg_thermometer , IDC_PROGRESS1 , pos )
  End If
END SUBROUTINE ThermApply
```

④编写主对话框新增控件的回调例程

当用户点击主对话框上的微调按钮、滑动条及滚动条时，都会引发 DLG_CHANGE 类型的回调，且要求的行为都是相同的；依据当前控件值刷新其他控件值，以使所有的数据输入、输出控件保持同步。这样，我们就可以为上述三个控件设置一个回调例程(UpdateTemp)，并依据控件 ID 来组织代码段。UpdateTemp 回调例程代码如下：

```
Subroutine UpdateTemp(dlg,control_Name,callBackType)
  Use IFLogM
  Implicit None
  Include 'Resource.FD'
  Type(dialog) dlg
  Integer control_Name,callBackType
  Logical*4 status
  Integer pos
  Character*3 textPos
  status = DlgGet( dlg , control_Name , pos )
  Write(textPos,"(I3)") pos
  Select Case ( control_Name )
    Case ( IDC_SPIN1 )
      status = DlgSet( dlg , IDC_SLIDER1 , pos )
      status = DlgSet( dlg , IDC_SCROLLBAR1 , pos )
    Case ( IDC_SLIDER1 )
      status = DlgSet( dlg , IDC_SPIN1 , pos )
      status = DlgSet( dlg , IDC_SCROLLBAR1 , pos )
    Case ( IDC_SCROLLBAR1 )
      status = DlgSet( dlg , IDC_SLIDER1 , pos )
      status = DlgSet( dlg , IDC_SPIN1 , pos )
  End Select
  status = DlgSet( dlg , IDC_CURRENTVALUE , textPos )
End Subroutine UpdateTemp
```

主对话框上的编辑框的值在两种情况下发生改变：一是点击编辑框旁边的微调按钮；二是通过键盘直接在编辑框中输入数据。第一种情况下注册的回调例程为 UpdateTemp；这里为第二种情况注册 DLG_LOSEFOCUS 类型的回调例程 ThermEdit。该回调当用户输入数据完毕，且输入焦点移动到别处时被引发，其实现代码如下：

```
Subroutine ThermEdit(dlg,id,callBackType)
  Use IFLogM
  Implicit None
  Include 'Resource.FD'
  Type(dialog) dlg
  Integer id,callBackType
  Logical*4 status
  Integer pos
  Character*3 textPos
```

```
  If ( callBackType == DLG_LOSEFOCUS ) then
    !Set the values of other controls to match the change
    status = DlgGet( dlg ,id , textPos )
    Read(textPos , "(I3)") pos
    status = DlgSet( dlg , IDC_CURRENTVALUE , textPos )
    status = DlgSet( dlg , IDC_SLIDER1 , pos )
    status = DlgSet( dlg , IDC_SPIN1 , pos )
    status = DlgSet( dlg , IDC_SCROLLBAR1 , pos )
  End If
End Subroutine ThermEdit
```

**⑤编写“温度计”对话框回调例程**

进度条指示器默认的颜色为蓝色，要想将其设置为红色，不能在设计状态通过设置属性来实现，但可以在运行状态通过向控件发送相应的消息来实现。

前面已经提过：在对话框初始化和销毁时会自动调用对话框回调例程。而设置进度条指示器颜色应在对话框显示前的初始化中完成。故此，我们添加如下的“温度计”对话框回调例程：

```
Subroutine ThermometerSub(dlg,id,callBackType)
  Use User32
  Use IFLogM
  Implicit None
  Include 'Resource.FD'
  Type(dialog) dlg
  Integer id , callBackType
  Integer*4    cref
  Integer*4    lret
  If ( callBackType == DLG_INIT ) then
    cref = #FF    !Red
    lret = DlgSendCtrlMessage( dlg , IDC_PROGRESS1 ,PBM_SETBARCOLOR , 0 , cref )
  End If
End Subroutine ThermometerSub
```

其中的 DlgSendCtrlMessage 为 IFLogM 模块中定义的对话框例程，用来向对话框上的控件发送 Windows 消息。其语法为：

result = DlgSendCtrlMessage(dlg,controlID,msg,wParam,lParam)

参数 dlg 指包含控件的对话框变量，controlID 指接收消息的控件 ID，msg 值向控件发送的消息(T_Msg)，wParam 和 lParam 用来规定消息的附加信息；函数结果(result)为 Integer(4)，表示消息处理结果，其具体值依赖于所发送的消息。

从 Platform SDK 中可以获知：PBM_SETBARCOLOR 代表设置进度条指示器颜色的 Windows 消息。调用发送消息的 API 函数 SendMessage 的语法为：

```
LRESULT SendMessage(          //返回 LRESULT 类型结果
    HWND hWnd,                //接收消息控件的句柄
    UINT Msg,                 //消息 ID 号
    WPARAM wParam,            //第一个消息参数
```

```
        LPARAM lParam              //第二个消息参数
    );
```

API 函数 SendMessage 被封装在 User32 模块中，所以也可以直接调用该函数向进度条发送 PBM_SETBARCOLOR 消息：

```
lret = SendMessage(GetDlgItem(dlg%hwnd,IDC_PROGRESS1),PBM_SETBARCOLOR,0,cref)
```

其中的 GetDlgItem 也是封装在 User32 模块中的 API 函数，用来返回特定控件的句柄。

所有的 Windows 消息都在 IFWinTY 模块中予以声明，而 User32 模块又引用了 IFWinTY 模块，并将众多界面操作例程封装其中。关于 Windows 消息的使用，还需参照 SDK 文档。

（事实上，设置进度条颜色在具有主题样式的 Windows 应用程序中已经不能体现出来，参见图 6-2，录入人在 WindowsXP 主题下没有看到预期的效果。为了美观，没有加入红色图标控件。需要查看效果，可以切换到 Windows 经典主题，效果截图保存在 Code 文件夹下 Chapter6\Ex_2\ClassicTheme.jpg。建议最好不要在程序设计中利用控件的外观。）

## 6.3 SDI 和 MDI Windows 程序

### 6.3.1 SDI 和 MDI 界面

用户使用应用程序时，如果程序一次只能打开一个文档，那么这种程序就称为 SDI(单文档界面)程序，反之就称为 MDI(多文档界面)程序。运用 Intel Visual Fortran 9.0 的 Fortran Windows 程序向导，生成的 SDI 和 MDI 程序分别如图 6-5 和图 6-6 所示。

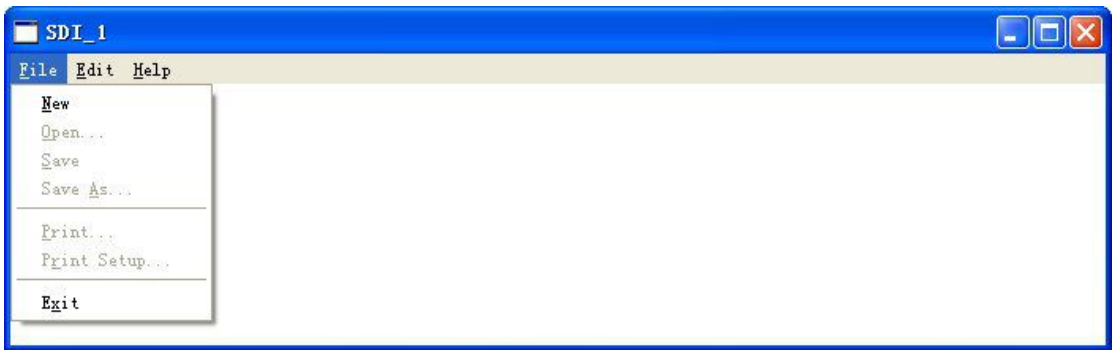



图 6-5 基本的 SDI 程序界面

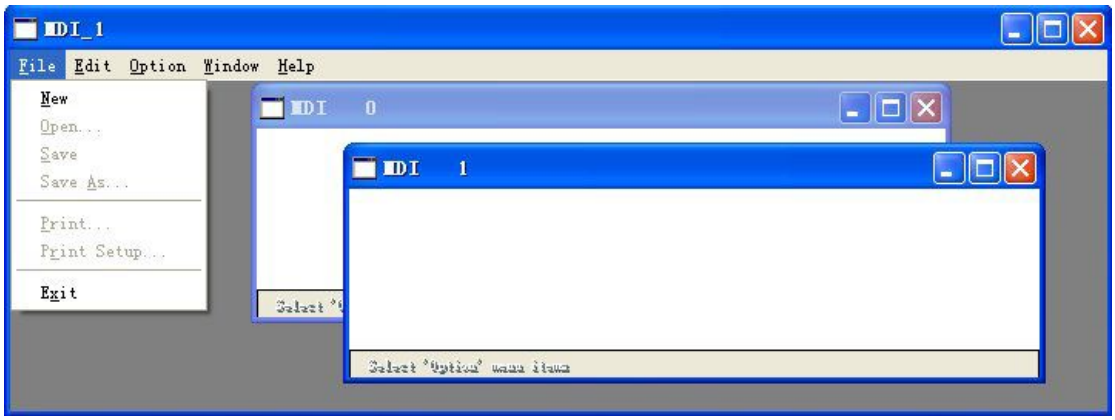



图 6-6 基本的 MDI 程序界面

Fortran 向导生成的 MDI 程序运行时只显示框架窗口(父窗口)，当执行“File”菜单的“New”命令时，会在框架窗口中生成若干子窗口，如图 6-6 所示。

现在，已经很少有人再设计 MDI 程序了，甚至连 Microsoft 也建议：尽量不要开发多文档界面程序。相比之下，SDI 程序更简单也更易于实现。基于这个原因，本书不对 MDI 程序做更多的介绍。

下面结合实例讲解 SDI 程序中如何使用无模式对话框、菜单及如何绘制图形。

## 6.3.2 使用无模式对话框

无模式对话框大多在 Windows 程序中使用。在前面基于对话框的 Windows 程序中，主对话框本身就是无模式对话框。这里探讨在 SDI 程序中使用无模式对话框的问题。

在图 6-7 所示的实例中，使用无模式对话框比模式对话框更为有利：可以一边在对话框中设置数据；一边观察新设置的数据所产生的图形缩放效果及曲线光滑程度。

【例 6-3】将第 5 章例 5-3 的 QuickWin 程序设计为 SDI 程序，提供菜单及对话框，并在 SDI 客户区绘制函数图形，见图 6-7 所示。

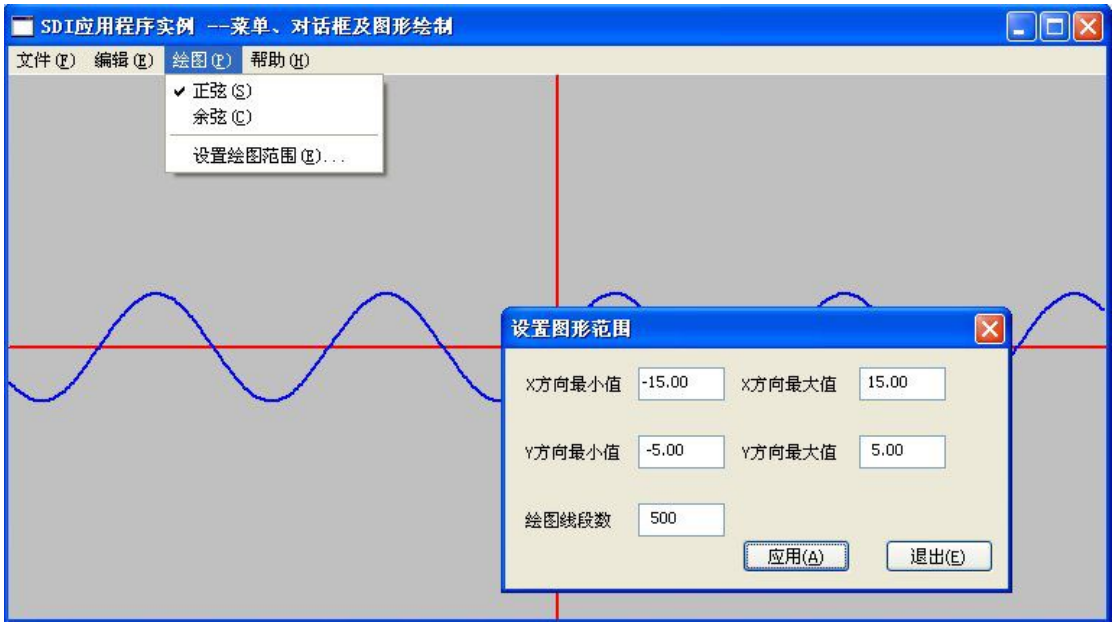



图 6-7 SDI 应用程序实例

*(1)处理无模式对话框产生的消息*

启动 Fortran Windows 程序向导，在图 6-3 所示的程序设置窗口中选择“Single Document Interface(SDI) sample code”应用程序，生成 SDI 工程骨架。

类似于基于对话框的 Windows 程序，在 SDI 程序中使用无模式对话框，也要在主消息循环中处理无模式对话框产生的消息，如下列完善后的主函数代码所示：

```
integer*4 function WinMain( hInstance, hPrevInstance, lpszCmdLine, nCmdShow )
!DEC$ IF DEFINED(_X86_)
!DEC$ ATTRIBUTES STDCALL, ALIAS : '_WinMain@16' :: WinMain
!DEC$ ELSE
!DEC$ ATTRIBUTES STDCALL, ALIAS : 'WinMain' :: WinMain
!DEC$ ENDIF

    use user32
    use kernel32
    use DrawSDIGlobals
```

```
implicit none

integer*4 hInstance
integer*4 hPrevInstance
integer*4 lpszCmdLine
integer*4 nCmdShow

include 'DrawSDI.fi'

! Variables
type (T_WNDCLASS)       wc
type (T_MSG)            mesg
integer*4              ret
logical*4              lret
integer                haccel

character(SIZEOFAPPNAME) lpszClassName
character(SIZEOFAPPNAME) lpszIconName
character(SIZEOFAPPNAME) lpszAppName
character(SIZEOFAPPNAME) lpszMenuName
character(SIZEOFAPPNAME) lpszAccelName

ghInstance = hInstance
ghModule = GetModuleHandle(NULL)
ghwndMain = NULL
lpszClassName ="DrawSDI"C
lpszAppName ="SDI应用程序实例--菜单、对话框及图形绘制"C   !<<<<----
lpszIconName ="DrawSDI"C
lpszMenuName ="DrawSDI"C
lpszAccelName ="DrawSDI"C

!  If this is the first instance of the application, register the
!  window class(es)
if (hPrevInstance .eq. 0) then
    !  Main window
     wc%lpszClassName = LOC(lpszClassName)
     wc%lpfnWndProc = LOC(MainWndProc)
     wc%style = IOR(CS_VREDRAW , CS_HREDRAW)
     wc%hInstance = hInstance
     wc%hIcon = LoadIcon( hInstance, LOC(lpszIconName))
     wc%hCursor = LoadCursor( NULL, IDC_ARROW )
     wc%hbrBackground = GetStockObject(LTGRAY_BRUSH)      !<<<<----
     wc%lpszMenuName = NULL
```

```
        wc%cbClsExtra = 0
        wc%cbWndExtra = 0
        if (RegisterClass(wc) == 0) goto 99999
    end if

    ! Load the window's menu and accelerators and create the window
    !
    ghMenu = LoadMenu(hInstance, LOC(lpszMenuName))
    if (ghMenu == 0) goto 99999
    haccel = LoadAccelerators(hInstance, LOC(lpszAccelName))
    if (haccel == 0) goto 99999

    ghwndMain = CreateWindowEx(  0, lpszClassName,                &
                                 lpszAppName,                     &
                                 INT(WS_OVERLAPPEDWINDOW),        &
                                 CW_USEDEFAULT,                   &
                                 0,                               &
                                 CW_USEDEFAULT,                   &
                                 0,                               &
                                 NULL,                            &
                                 ghMenu,                          &
                                 hInstance,                       &
                                 NULL                             &
                               )
    if (ghwndMain == 0) goto 99999

    lret = ShowWindow( ghwndMain, nCmdShow )

    ! Read and process messsages
    do while( GetMessage (mesg, NULL, 0, 0) )
      If (DlgIsDlgMessage(mesg,gdlg).EQV.False) then     !<<<<----
        if ( TranslateAccelerator (mesg%hwnd, haccel, mesg) == 0) then
          lret = TranslateMessage( mesg )
          ret  = DispatchMessage( mesg )
        end if                                           !<<<<----
      End If
    end do
    WinMain = mesg.wParam
    return
99999 &
    ret = MessageBox(ghwndMain, "Error initializing application DrawSDI"C, &
                     "Error"C, MB_OK)
    WinMain = 0
end
```

类似于例 6-1 用 C 语言编写的 Windows 程序，SDI 主函数也要进行下列处理：注册窗口类、创建并显示窗口和执行消息循环。这里，在消息循环中添加了 DlgIsDlgMessage 函数语句，用来检测要使用的无模式对话框产生的消息。

在给窗口类(T_WNDCLASS)变量 wc 的成员赋值时，程序调用了 3 个 API 函数：LoadIcon、LoadCursor 和 GetStockObject，分别用来装载光标、装载图标和获取设置窗口客户区背景颜色的系统画刷；在创建窗口前，程序调用了 2 个 API 函数：LoadMenu 和 LoadAccelerators，分别用来装载菜单资源和快捷键字符表。

主函数中引用的全局变量模块 DrawSDIGlobals 为：

```
module DrawSDIGlobals
  Use GDI32
  Use IFLogM
  implicit none
  !  Parameters
  integer*4, parameter, public :: SIZEOFAPPNAME = 100
  integer*4, parameter, public :: MAXPOINTS     = 10000
  !  Global data
  integer     ghInstance
  integer     ghModule
  integer     ghwndMain
  integer     ghMenu
  Type(dialog) gdlg
  Type(T_RECT)::recTP
  Type(T_POINT)::svPoints(MAXPOINTS),xPoints(2),yPoints(2)
  Type (T_PAINTSTRUCT)  ps
  Integer::Lines = 500
  Real*8 :: X_Start = - 15.0
  Real*8 :: X_End   =   15.0
  Real*8 :: Y_Top   =   5.0
  Real*8 :: Y_Bottom= - 5.0
  Integer::hPenAxis , hdc , hPenSin , hPenCos , hPen , width , height , num , which = 1
  Logical::DlgIsExist = .False.
end module
```

该模块在向导声明的变量基础上，添加了对话框以及用于绘制图形的全局变量。其中，派生类型 T_RECT 定义了一个矩形的左上角和右下角坐标，T_Point 则定义了一个点的 X、Y 坐标，它们对应的结构体 RECT 和 POINT 的定义分别为：

```
typedef struct tagRECT {
    LONG left;          //左上角X坐标
    LONG top;           //左上角Y坐标
    LONG right;         //右下角X坐标
    LONG bottom;        //右下角Y坐标
} RECT;

typedef struct tagPOINT {
    LONG x;             //X坐标
```

```
    LONG y;                 //Y坐标
} POINT;
```

*(2)插入对话框资源*

将第 5 章例 5-3 所用的对话框插入本实例资源文件(工程名.rc)中。Fortran SDI 向导已自动设置了命令行的 defToFD 工具，插入的对话框资源 ID 会在编译时自动添加到 Resource.FD 头文件中。至于对话框及其控件的属性设置，参见第 5 章例 5-3。

*(3)操作无模式对话框*

操作无模式对话框与操作模式对话框类似，都是使用 IFLogM 模块中的对话框例程来初始化对话框、设置控件或获取控件值；所不同的是，无模式对话框和模式对话框分别适用 DlgModless 和 DlgModel 函数来展示对话框。本实例将对话框的相关操作统一封装在 SetGetDlg 模块中：

```
Module SetGetDlg
  Use DrawSDIGlobals
  Implicit None
  Contains
    Subroutine SetRange()   !设置对话框
      Include 'Resource.FD'
      Integer*4 :: rlt
      Character(Len=20) :: str
      rlt = DlgInit( IDD_RANGE_DIALOG , gdlg )
      rlt = DlgSetSub( gdlg , ID_EXIT , RangeExit )
      rlt = DlgSetSub( gdlg , IDM_APPLY , RangeApply )
      !将实型变量值转换成字符串，然后设为文本框中的值
      Write(str,'(f6.2)') X_Start
      rlt = DlgSet( gdlg , IDC_XMIN , str )
      Write(str,'(f6.2)') X_End
      rlt = DlgSet( gdlg , IDC_XMAX , str )
      Write(str,'(f6.2)') Y_Bottom
      rlt = DlgSet( gdlg , IDC_YMIN , str )
      Write(str,'(f6.2)') Y_Top
      rlt = DlgSet( gdlg , IDC_YMAX , str )
      Write(str,'(I5)') Lines
      rlt = DlgSet( gdlg , IDC_LINES , str )
      DlgIsExist = .True.
      rlt = DlgModeless( gdlg , SW_SHOWNA )
    End Subroutine SetRange

    Subroutine GetRange()  !获取对话框中的控件值
      Include 'Resource.FD'
      Integer*4 :: rlt
      Character(Len=20) :: str
        rlt = DlgGet( gdlg , IDC_XMIN , str )
```

```
        Read(str,*) X_Start
        rlt = DlgGet( gdlg , IDC_XMAX , str )
        Read(str,*) X_End
        rlt = DlgGet( gdlg , IDC_YMIN , str )
        Read(str,*) Y_Bottom
        rlt = DlgGet( gdlg , IDC_YMAX , str )
        Read(str,*) Y_Top
        rlt = DlgGet( gdlg , IDC_LINES , str )
        Read(str,*) Lines
    End Subroutine GetRange

    Subroutine RangeExit(dlg,id,callBackType)
    !DEC$ ATTRIBUTES DEFAULT :: RangeExit
      Type(dialog) dlg
      Integer id , callBackType
      If ( callBackType == DLG_CLICKED ) then
        DlgIsExist = .False.
        Call DlgExit( dlg )
      End If
    End Subroutine RangeExit

    Subroutine RangeApply(dlg,id,callBackType)
    !DEC$ ATTRIBUTES DEFAULT :: RangeApply
      Type(dialog) dlg
      Integer id , callBackType
      If ( callBackType == DLG_CLICKED ) then
        Call GetRange()
      End If
    End Subroutine RangeApply

End Module SetGetDlg
```

模块共包含了 SetRange、GetRange、RangeApply 和 RangeExit 四个例程。其中，RangeApply 和 RangeExit 例程分别为“设置图形范围”对话框中“应用”按钮和“退出”按钮的回调例程；GetRange 例程由回调例程 RangeApply 调用；SetRange 例程则由“设置范围”菜单例程调用。

展示无模式对话框函数 DlgModeless 的语法格式为：

result = DlgModeless(dlg[ , nCmdShow , hwndParent ])

参数 dlg 代表对话框变量；可选参数 nCmdShow 规定对话框显示方式，它须是表 6-5 中所列的值，其缺省值为 SW_SHOWNORMAL；可选参数 hwndParent 规定对话框的父窗口，其缺省值按下列次序确定：

- 如果 DlgModeless 函数是从一个无模式对话框的回调例程中调用，那么调用它的无模式对话框为其父窗口；
- Windows 桌面窗口为其父窗口。

假如 DlgModeless 函数调用成功，返回值为逻辑真(.True.)；否则为逻辑假(.False.)。

表 6-5 规定对话框显示方式参数(nCmdShow)的取值

| 符号常量 | 描述 |
|---|---|
| SW_HIDE | 隐藏对话框 |
| SW_MINIMIZE | 最小化对话框 |
| SW_RESTORE | 激活并显示对话框。如果对话框被最小化或最大化，系统将其还原到正常大小和位置 |
| SW_SHOW | 激活并显示对话框 |
| SW_SHOWMAXIMIZED | 激活并显示对话框，并显示为最大化窗口 |
| SW_SHOWMINIMIZED | 激活并显示对话框，并显示为最小化窗口 |
| SW_SHOWMINNOACTIVE | 以图标形式显示对话框，当前激活的窗口仍保持激活 |
| SW_SHOWNA | 以当前状态显示对话框 |
| SW_SHOWNOACTIVE | 以前一次的大小和位置来显示对话框，当前激活的窗口仍保持激活 |
| SW_SHOWNORMAL | 激活并显示对话框。如果对话框被最小化或最大化，系统将其还原到正常大小和位置 |

(4)无模式对话框的关闭

在基于对话框的 Windows 程序中，主对话框本身就是无模式对话框，通过注册对话框的 DLG_DESTORY 回调，来间接注册“退出”按钮的回调例程(已为“应用”按钮注册了专门的回调例程)，并在其中调用函数 PostQuitMessage、发出 WM_QUIT 的 Windows 消息，来结束程序运行。

此处，无模式对话框是 SDI 程序的一个子窗口，点击“退出”按钮并不引发对话框的 DLG_DESTORY 回调。因此，须注册专门的“退出”按钮回调例程，并在其中通过调用 DlgExit 例程来关闭无模式对话框；如果调用函数 PostQuitMessage，就将退出整个应用程序。

一个值得注意的问题是：当选取“设置范围”菜单项、执行 Call SetRange 语句时，会打开多个无模式对话框，除了最新打开的对话框外，其他对话框不能通过点击“退出”按钮、执行 RangeExit 回调来关闭。况且，实际情况下也不必要同时打开多个对话框运行实例。

对此，程序在全局模块中设置了一个逻辑变量(DlgIsExist)，并初始化为逻辑假。当调用 SetRange 例程、展示“设置图形范围”对话框时，将其设为逻辑真；当点击“退出”按钮、执行 RangeExit 回调时，将其设为逻辑假。对话框不存在时，才允许打开新的对话框：

```
If(DlgIsExist.EQV..False.) Call SetRange()
```

## 6.3.3 使用菜单

菜单是 Windows 程序界面的重要组成部分，也是几乎所有图形界面应用程序都要使用的一种资源。菜单可以使用户直观了解和使用应用程序所提供的各项功能，因此一个菜单设计的如何，将直接影响一个应用程序的质量。

### 6.3.3.1 编辑菜单

在 Intel Visual Fortran 9.0 中，可以像插入对话框资源那样插入菜单资源，然后在.NET 菜单编辑器上直接编辑菜单。实例程序在 Fortran Windows 向导为 SDI 程序设置的缺省菜单基础上，添加了“绘图”菜单栏，如图 6-8 所示。

在资源描述文件(.rc)中，菜单部分的定义为：

```
// Menu
DRAWSDI MENU
BEGIN
    POPUP "文件(&F)"
    BEGIN
        MENUITEM "新建(&N)",                    IDM_NEW
        MENUITEM "打开(&O)...",                 IDM_OPEN, GRAYED
        MENUITEM "保存(&S)",                    IDM_SAVE, GRAYED
        MENUITEM "另存为(&A)...",               IDM_SAVEAS, GRAYED
        MENUITEM SEPARATOR
        MENUITEM "打印(&P)...",                 IDM_PRINT, GRAYED
        MENUITEM "打印设置(&R)...",             IDM_PRINTSETUP, GRAYED
        MENUITEM SEPARATOR
        MENUITEM "退出(&X)",                    IDM_EXIT
    END
    POPUP "编辑(&E)"
    BEGIN
        MENUITEM "撤销(&U)\tCtrl+Z",            IDM_UNDO, GRAYED
        MENUITEM SEPARATOR
        MENUITEM "剪切(&T)\tCtrl+X",            IDM_CUT, GRAYED
        MENUITEM "复制(&C)\tCtrl+C",            IDM_COPY, GRAYED
        MENUITEM "粘贴(&P)\tCtrl+V",            IDM_PASTE, GRAYED
    END
    POPUP "绘图(&P)"
    BEGIN
        MENUITEM "正弦(&S)",                    IDM_SIN, CHECKED
        MENUITEM "余弦(&C)...",                 IDM_COS
        MENUITEM SEPARATOR
        MENUITEM "设置绘图范围(&E)...",         IDM_SCALE
    END
    POPUP "帮助(&H)"
    BEGIN
        MENUITEM "内容(&C)",                    IDM_HELPCONTENTS
        MENUITEM "搜索(&S)...",                 IDM_HELPSEARCH
        MENUITEM "如何使用帮助(&H)",            IDM_HELPHELP
        MENUITEM SEPARATOR
        MENUITEM "关于DrawSDI(&A)...",                  IDM_ABOUT
    END
END
```




图 6-8 .NET 菜单编辑器

编辑菜单需要注意的是:要生成中文菜单,必须将菜单的 Language 属性规定为"中文(中国)",如图 6-9 所示。否则,输入的是中文,但显示的不是中文。(建议所有资源全部设定该属性)。




图 6-9 设置菜单的 Language 属性

### 6.3.3.2 加载菜单资源

为了创建带菜单栏的主窗口,须首先加载菜单资源。其语句如下:

```
ghMenu = LoadMenu( hInstance , LOC(lpszMenuName) )
```

在调用 CreateWindowEx 函数创建主窗口时,使用上面返回的菜单句柄(ghMenu):

```
ghwndMain = CreateWindowEx(  0,  lpszClassName,                &
                             lpszAppName,                      &
                             INT(WS_OVERLAPPEDWINDOW),         &
                             CW_USEDEFAULT,                    &
                             0,                                &
                             CW_USEDEFAULT,                    &
                             0,                                &
                             NULL,                             &
                             ghMenu,                           & !<<<<----
                             hInstance,                        &
                             NULL                              &
                              )
```

### 6.3.3.3 设置菜单标志

在本实例中,为了区分当前绘制的是正弦还是余弦,使用了菜单项核取标志(√)。

在设计状态下,要设置某一菜单项的核取标志,直接在属性页中将其 Checked 属性设为

True；在运行时，则通过调用 API 函数 CheckMenuItem 来实现。该函数的原型为：

DWORD CheckMenuItem(HMENU hmenu,UINT uIDCheckItem,UINT uCheck);

参数 hmenu 为菜单句柄，它可以是整个菜单栏句柄(上述 ghMenu)，也可以是某一菜单(如“绘图”菜单)句柄。菜单序号从左往右依次为 0、1、2、…，所以“绘图”菜单序号为 2。

uIDCheckItem 为要设置标志的菜单项，它可以用菜单项 ID 表示，也可以用菜单项序号表示。菜单项序号从上往下一次为 0、1、2、…，所以“余弦”菜单项序号为 1。

uCheck 规定菜单项核取标志。通常，它是 MF_BYCOMMAND 与 MF_CHECKED 或 MF_UNCHECKED 的组合，或者 MF_BYPOSITION 与 MF_CHECKED 或 MF_UNCHECKED 的组合。前者通过菜单项 ID 来标志；后者则通过菜单项序号来标志。如果采用单一的 MF_CHECKED 或 MF_UNCHECKED，那么默认为 MF_BYCOMMAND，即通过菜单项 ID 来标志。

例如，下列语句通过序号来标志“正弦”菜单，同是取消“余弦”菜单项标志：

```
hMenu = GetSubMenu( GetMenu(hWnd) , 2 )
ret = CheckMenuItem( hMenu , 0 , IOR(MF_BYPOSITION,MF_CHECKED) )
ret = CheckMenuItem( hMenu , 1 , IOR(MF_BYPOSITION,MF_UNCHECKED) )
```

这种方式须先获取“绘图”菜单项句柄。若通过菜单 ID 来标志，则可使用整个菜单栏句柄：

```
ret=CheckMenuItem(ghMenu,IDM_SIN,IOR(MF_BYCOMMAND,MF_CHECKED))
ret=CheckMenuItem(ghMenu,IDM_COS,IOR(MF_BYCOMMAND,MF_UNCHECKED))
```

函数返回先前的菜单项标志(MF_CHECKED 或 MF_UNCHECKED)。若设置标志的菜单项不存在，函数返回-1。

#### 6.3.3.4 处理菜单消息

当用户选取菜单项时，Windows 向主窗口发送 WM_COMMAND 消息。WM_COMMAND 消息的 wParam 参数包含下列信息：

(1)其低位字节规定菜单项标识，即菜单项 ID。

(2)假如 WM_COMMAND 消息来自菜单项，其高位字节规定为 0；假如 WM_COMMAND 消息来自快捷键。其高位字节规定为 0。通常，无须判断高位字节，只需比较低位字节。

例如，下列为窗口函数(MainWndProc)处理“正弦”、“余弦”和“设置范围”菜单消息的代码段：

```
! WM_COMMAND: user command
     case (WM_COMMAND)
       select case ( IAND(wParam, 16#ffff ) )
           case (IDM_SIN)
               ret = CheckMenuItem(ghMenu,IDM_SIN,IOR(MF_BYCOMMAND,MF_CHECKED))
               ret = CheckMenuItem(ghMenu,IDM_COS,IOR(MF_BYCOMMAND,MF_UNCHECKED))
               which = 1
               hPen = hPenSin
               Call FuncPoints()
               ret = InvalidateRect( hwnd , recTP , .True. )
               MainWndProc = 0
               return
           case (IDM_COS)
               ret = CheckMenuItem(ghMenu,IDM_SIN,IOR(MF_BYCOMMAND,MF_UNCHECKED))
               ret = CheckMenuItem(ghMenu,IDM_COS,IOR(MF_BYCOMMAND,MF_CHECKED))
```

```
            which = 2
            hPen = hPenCos
            Call FuncPoints()
            ret = InvalidateRect( hwnd , recTP , .True. )
            MainWndProc = 0
            return
        case (IDM_SCALE)
            If ( DlgIsExist .EQV. .False. ) then
              Call SetRange
            End If
            MainWndProc = 0
            return
```

其中，FuncPoints 为获取绘图点例程，它与获取坐标轴点例程一起置于 GetPlotPoints 模块中：

```
Module GetPlotPoints        !获取绘图点
  Use DrawSDIGlobals
  Use User32
  Implicit None
  Contains
    Subroutine AxisPoints(hwnd)
      Integer hwnd,ret
      ret = GetClientRect( hwnd ,recTP )
      width  = recTP.right - recTP.left
      height = recTP.bottom - recTP.top
      xPoints(1).x = - width / 2
      xPoints(1).y = 0
      xPoints(2).x = width / 2
      xPoints(2).y = 0

      yPoints(1).x = 0
      yPoints(1).y = - height / 2
      yPoints(2).x = 0
      yPoints(2).y = height / 2
    End Subroutine AxisPoints

    Subroutine FuncPoints()
      Integer i
      Real(8) :: x , step , f1 , f2
      f1 = width / ( X_End - X_Start )
      f2 = - height / ( Y_Top - Y_Bottom )
      step = ( X_End - X_Start ) / Lines
      i = 1
      Do x = X_Start , X_End - step , step
```

```
        svPoints(i).x = f1 * x
        svPoints(i+1).x = f1 * ( x + step )
        If ( which ==1 ) then
          svPoints(i).y = f2 * sin(x)
          svPoints(i+1).y = f2 * sin(x+step)
        Else
          svPoints(i).y = f2 * cos(x)
          svPoints(i+1).y = f2 * cos(x+step)
        End If
        i = i + 1
      End Do
      num = i
    End Subroutine FuncPoints
End Module GetPlotPoints
```

## 6.3.4 绘制图形

### 6.3.4.1 GDI 和设备描述环境

Windows 把与绘图工作相关的操作都制作成了函数，GDI(Graphical Device Interface，图形设备接口)就是用户可以调用的一套用于绘图的函数集。例如，调用 GDI 中的 Rectangle 函数绘制矩形，调用 Polyline 函数绘制折线，调用 TextOut 函数绘制文本等。除了这些绘图函数外，GDI 还包含一套用于绘图的工具(对象)，如画笔、画刷、字体等。

由于计算机显示设备的多样性，要求程序设计人员具有对所有图形显示设备的编程能力是不现实的。为此，Windows 设置了虚拟的图形显示对象，设计人员可以在这个虚拟的图形显示对象上绘图，而把图形最终转换为实际物理设备上图形的工作交由系统完成。这个虚拟的图形显示对象就叫做设备描述环境(DC)，它用来决定如何显示图形的属性和对象。换句话说，DC 可以看做是系统预置的具有一套绘图方法及显示方法的画板。

在应用程序中，如果需要在某个显示设备上绘图，那么首先应该获得这个显示设备的 DC。为了方便，系统已预置了一套 DC 的属性和对象，所以获得了该设备的 DC 后，就可以在这个默认的环境下绘图了。如果对系统预置的属性和对象不满意的话，也可以按自己的需要对它们进行设置。例如，DC 在初始化时自动匹配了一枝黑色画笔，如果想使用其它颜色的画笔，则可创建相应颜色的画笔来替换它。

### 6.3.4.2 窗口客户区

除去标题栏、菜单栏和边框剩下的窗口部分称为窗口客户区。在窗口上绘图，实际是在窗口客户区中绘图。API 函数 GetClientRect 专门用来获取窗口客户区，该函数原型为：

```
BOOL GetClientRect(HWND hWnd,LPRECT lpRect);
```

参数 hWnd 代表包含客户区的窗口句柄；lpRect 代表获取的客户区坐标，它是一个 RECT 结构体。若调用成功，函数返回非 0 值；否则，函数返回 0。

窗口客户区代表一个显示设备，在这个显示设备上绘图采用的是视口设备坐标(像素)。默认情况下，窗口客户区左上角规定为视口坐标原点，视口坐标范围从左上角到右下角。若要重新规定视口原点，可调用 API 函数 SetViewportOrgEx，该函数原型为：

```
BOOL SetViewportOrgEx(
    HDC hdc,          //设备环境句柄
```

```
    int X,          //视口原点的 X 窗口坐标，单位像素
    int Y,          //视口原点的 Y 窗口坐标，单位像素
    LPPOINT lpPoint //指向原来视口原点的指针。若规定为 NULL，该参数不使用
);
```

若调用成功，函数返回非 0 值；否则，函数返回 0。

#### 6.3.4.3 处理 WM_PAINT 消息——绘制图形

当系统或程序请求重画窗口时，就会发送 WM_PAINT 消息。窗口重画时，在窗口上绘制的图形也得重画。因此，选择在 WM_PAINT 消息处理中绘制图形最为合适。

程序请求重画窗口的一个常用手段，是调用 InvalidateRect 函数。该函数的原型为：

```
BOOL InvalidateRect(
    HWND hWnd,          //窗口句柄
    CONST RECT *lpRect, //需要重画的矩形窗口区域
    BOOL bErase         //是否擦除重画区域上原来的图形
  );
```

若调用成功，函数返回非 0 值；否则，函数返回 0。

该函数的功能是积累要刷新的区域。假如有刷新区域存在，而消息队列中又没有其它消息要处理，系统就会向窗口发送 WM_PAINT 消息。

本实例程序中，正是采用了这一手段：在“正弦”、“余弦”菜单消息及窗口大小改变消息(WM_SIZE)中，调用 InvalidateRect 函数来引发 WM_PAINT 消息，在该消息处理中统一绘制图形。下列为窗口函数中与绘图操作相关的消息处理代码：

```
    select case ( mesg )
      case (WM_CREATE)
          hPenAxis = CreatePen( PS_SOLID , 2 , 16#020000FF )
          hPenSin  = CreatePen( PS_SOLID , 2 , 16#02FF0000 )
          hPenCos  = CreatePen( PS_SOLID , 2 , 16#0200FF00 )
          hPen = hPenSin
          Call AxisPoints( hwnd )
          MainWndProc = 0
          return
    ! WM_DESTROY: PostQuitMessage() is called
      case (WM_DESTROY)
          ret = DeleteObject( hPenAxis)
          ret = DeleteObject( hPenSin)
          ret = DeleteObject( hPenCos)
          Call PostQuitMessage( 0 )
          MainWndProc = 0
          return
      case (WM_PAINT)
          hdc = BeginPaint( hwnd , ps )
          ret = SetViewportOrgEx( hdc , width / 2 , height / 2 , NULL_POINT )
          !Plot axis
          ret = SelectObject( hdc , hPenAxis )
          ret = PolyLine( hdc , xPoints(1) , 2 )
```

```
            ret = PolyLine( hdc , yPoints(1) , 2 )
            !Plot sin or cos
            ret = SelectObject( hdc , hPen )
            ret = PolyLine( hdc , svPoints(1) ,num )
            ret = EndPaint( hwnd , ps )
            MainWndProc = 0
            return
        case (WM_SIZE)
            Call AxisPoints( hwnd )
            Call FuncPoints()
            ret = InvalidateRect( hwnd , recTP , .True. )
            MainWndProc = 0
            return
```

在窗口创建时(WM_CREATE)创建自定义画笔，绘图时(WM_PAINT)将其选入设备环境，窗口销毁时(WM_DESTROY)删除自定义画笔。

在 WM_PAINT 消息处理中，绘图前先调用 BeginPaint 函数获取设备环境，绘图完成后调用 EndPaint 函数标志绘图工作结束。两个函数具有相同的参数表，这里只介绍 BeginPaint 函数：

```
HDC BeginPaint(
    HWND hwnd,                //窗口句柄
    LPPAINTSTRUCT lpPaint     //绘图信息
);
```

该函数执行绘图前的初始化，并用绘图相关信息填充 PAINTSTRUCT 结构体。若调用成功，函数返回绘图窗口的设备环境句柄；否则，返回 NULL。PAINTSTRUCT 结构体的定义如下：

```
typedef struct tagPAINTSTRUCT {
    HDC  hdc;
    BOOL fErase;
    RECT rcPaint;
    BOOL fRestore;
    BOOL fIncUpdate;
    BYTE rgbReserved[32];
} PAINTSTRUCT;
```

该结构体参数为输出参数，实际使用时无须进行初始化。

绘图函数 PolyLine 的功能是，通过连接数组中的点来绘制一系列线段。该函数原型为：

```
BOOL Polyline(
    HDC hdc,              //设备环境句柄
    CONST POINT *lppt,    //绘图点(POINT)数组
    int cPoints           //数组中的点数
);
```

若调用成功，函数返回非 0 值；否则，函数返回 0。

其中，代表绘图点的第二个参数是指针参数，即传递点数组的首地址，在 Fortran 中只需传入数组的第一个元素；代表绘图点数的第三个参数须大于或等于 2。

# 小　结

■在 Windows 应用程序的主函数中，首先要注册窗口类，然后创建并显示窗口。窗口创建后程序就进入消息循环，在消息循环中，程序不断地获得消息并将消息派送给对应的窗口函数进行处理。

■Intel Visual Fortran 程序向导支持开发三种类型的 Windows 程序：无模式对话框、单文档界面和多文档界面程序。目前，多文档界面程序已不常用。

■无模式对话框 Windows 程序，其主窗口是由对话框资源创建的，并以无模式对话框方式展示；无模式对话框没有自己独立的消息循环，需要共享应用程序主消息循环。

■Intel Visual Fortran 单文档界面程序，需要注册窗口类，创建、显示窗口，获取、派送消息，并在窗口函数中对相应的消息进行处理。

■单文档程序使用无模式对话框，需要设置全局的对话框句柄变量，在主消息循环中对无模式对话框产生的消息进行处理，调用 DlgExit 例程关闭对话框。

■单文档程序使用菜单，需要插入菜单资源，在菜单资源编辑器中编辑菜单、在程序中加载、设置菜单，并对菜单消息(WM_COMMAND)进行处理。

■单文档程序绘图，通常在窗口重绘消息(WM_PAINT)中处理，并在窗口客户区设备环境中绘制。需要绘图时，显式调用 InvalidateRect 函数来触发 WM_PAINT 消息。

# 第 7 章　使用 COM 组件

COM(Component Object Model,组件对象模型)是一种开发软件组件的规范，同时也是一种软件组件开发技术。遵照 COM 规范编写的组件可以毫无阻碍地被所有应用程序所使用,即使编写应用程序使用的语言与编写组件的语言各不相同,也丝毫不影响组件与应用程序之间的通信，从而能以组件组装方式开发大型软件。

Intel Visual Fortran 9.0 通过提供模块向导，来简化对 COM 组件的操作，极大地提高了 COM 组件客户程序的开发效率。

**本章主要内容：**

- 相关的 COM 技术概述
- Intel Fortran 模块向导
- 使用自动化服务器
- 使用 ActiveX 控件

## 7.1 相关的 COM 技术

从面向过程程序设计方法，到面向对象程序设计方法，再到现在的 COM 组件软件设计方法，软件工程的发展经历了一个较长的过程。在这个过程中，人们追求的一个重要目标,就是希望软件开发能够像汽车制造那样，先按标准制造零件和部件，最后再用这些零件和部件组装成产品。人们经过若干年不懈的努力，终于取得了可喜的成就，这就是 COM。

### 7.1.1 组件对象模型(COM)

COM 把组件应用程序分为两大部分：COM 服务器和 COM 客户，如图 7-1 所示。COM 服务器的任务就是提供组件模块，它可以是 DLL，也可以是 EXE；而 COM 客户就是使用 COM 服务器组件的程序。客户必须通过组件接口才能获得 COM 服务器提供的服务，客户要知道从服务器可以获得什么服务，但无须知道该服务的实现细节。显然，为了使客户和服务器可以毫无障碍地进行沟通(通信)，接口就必须有一个双方都能识别的标准格式和相关的规范。它们所要遵守的这个规范就是组件对象模型(COM)，按照这个规范设计的软件部件就称为组件。

除了规范，COM 还提供有运行库。这个运行库以 DLL 形式置于 Windows 操作系统中，库中的 API 函数主要用来完成查找注册表、对组件进行定位及返回接口指针等基础工作。

按照组件在运行时所处的位置，可以将其分为以下几种类型：

(1)进程内组件。这种组件被实现为 DLL，运行在客户程序的地址空间内。

(2)进程外组件。这种组件被实现为 EXE，它又分为两种：①本地组件，与客户程序运行在同一台计算机上，但有自己的地址空间；②远程组件，运行在客户机联网的远程计算机上。

下面从开发的角度，结合实例进一步讲解 COM 的相关知识。

【例 7-1】利用 Visual C++ 6.0 中的活动模板库 ATL，创建一个进程内的 Math 组件。该组件实现了 IMath 和 IMath2 两个接口,如图 7-2 所示。IMath 接口有两个方法(函数成员)。分别将两个整数相加和相减；IMath2 接口在 IMath 接口基础上添加了求整数数组各元素和的办法。

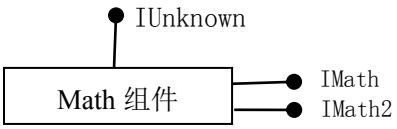



图 7-1 COM 应用程序的组成

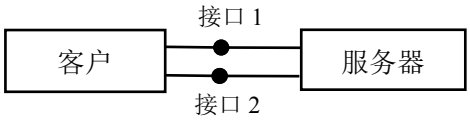



图 7-2 实例组件 Math 中的接口

活动模板库(ATL)是一套基于模板的 C++类，可以用来创建小型、快速的 COM 对象。它对主要的 COM 功能具有特殊支持，这些功能包括常用实现、双重接口、标准 COM 枚举数接口、连接点、分开的接口和 ActiveX 控件。

COM 规范规定：一个接口一旦发布，就不允许更改；若要扩充现有的组件功能、进行组件版本升级，可添加新的接口。新接口除了包含原接口已有的方法外，还包含代表扩展功能的新方法。

*(1)定义接口*

为了符合 COM 的二进制标准，必须使用一种独立的语言来描述接口，微软选用的语言是 IDL。IDL 是开放软件基金会(OSF)为分布式计算环境 RPC 软件包开发的一种语言，IDL 帮助 RPC 工程的客户机和服务器都遵守同一接口。为了将 IDL 语言应用于 COM 系统，微软对 IDL 的语法进行了扩充。IDL 本身不是一种编程语言，它只是用来定义接口的一种工具，至于对 IDL 语言的解释则由使用它的系统来决定。

描述实例组件 Math 及其接口的 IDL 文件为：

```
//Match_Server.idl : IDL source for Math_Server.dll

import "oaidl.idl";
import "ocidl.idl";

[
  object,
  uuid(4CF4F598-BB9D-4DCA-BCF0-C0AD32EBCC47),

  helpstring("IMath Interface"),
  pointer_default(unique)
]
interface IMath:IUnknown
{
  [helpstring("method Add")] HRESULT Add([in]long lOp1,[in]long lOp2,[out,retval]long
*plResult);
  [helpstring("method Subtract")] HRESULT Subtract([in]long lOp1,[in]long
lOp2,[out,retval]long *plResult);
};
[
  object,
  uuid(32FCCD40-78F6-457B-A799-12A73A9D1B4D),
```

```
  helpstring("IMath2 Interface"),
  pointer_default(unique)
]
interface IMath2:IUnknown
{
  [helpstring("method Add")] HRESULT ADD([in]long IOp1,[in]long IOp2,[out,retval]long
*plResult);
  [helpstring("method Subtract")] HRESULT Substract([in]long IOp1,[in]long
IOp2,[out,retval]long *plResult);
  [helpstring("method Sum")] HRESULT Sum([in]short sArraySize,[in,size_is(sArraySize)]short
sArray[],[out,retval]long *plResult);
};
[
  uuid(E466D266-A753-430D-9E0E-924749184B70),
  version(1.0),
  helpstring("Math_Server 1.0 Type Library")
]
library MATH_SERVERLib
{
  import("stdole32.tlb");
  import("stdole2.tlb");

  [
    uuid(61AE7459-0E1C-45B1-909C-90CDBBEE768A),
    helpstring("Math Class")
  ]
  coclass Math
  {
    [default] interface IMath;
    interface IMath2;
  };
};
```

**①基接口 IUnknown**

IMath 和 IMath2 接口均派生自 IUnknown 基接口。为了保证组件接口在基本功能上的标准化，COM 预定义了一个基本接口 IUnknown。同时规定，组件其他所有的接口必须由这个接口直接或间接派生而来，这样就保证了组件的所有接口都有统一的基本功能。基接口 IUnknown 的 IDL 描述为：

```
interface IUnknown{
    HRESULT _stdcall QueryInterface([in] GUID *riid, [out]void **ppvObj);
    unsigned long _stdcall AddRef();
    unsigned long _stdcall Release();
};
```

其中，QueryInterface 函数成员是用来查询同一组件其他接口的。通常，组件一般不

止一个接口，所以 COM 规定，在客户程序取得组件的任意一个接口后，该接口必须具有提供该组件其他接口的能力。换句话说，在客户得到组件的任意一个接口后，就应该可以通过这个接口得到该组件的其他接口。组件的这个能力，就要由基接口中的函数 QueryInterface 来实现。

该函数中的第一个参数 riid 指需要查询的接口 ID(IID),如果查询成功，第二个参数 ppvObj 存放按第一个参数查询得到的接口指针,该指针指向这个接口的虚函数表 vtable(图 7-3),虚函数表中存放的是接口中各个函数的函数指针,函数指针又指向各自的函数实现体。显然，只有通过接口指针客户程序才能获得接口所提供的服务。

AddRef 和 Release 这两个函数则用来管理组件对象生存期。客户程序在使用组件前，要申请创建组件对象，如果客户程序不再需要这个组件了，则应及时释放它。那么在多个客户程序使用同一组件对象的情况下，应该在什么时候由谁来释放组件对象呢？显然，对象的释放应由组件自己来控制。COM 使用了一个相当简单的方法：在组件中设置一个计数器，当客户程序引用组件对象时，就调用 AddRef 函数把计数器的值加 1；当客户程序不再引用该组件对象时，就调用 Release 函数把计数器的值减 1。这样，组件对象随时知道有多少个客户程序正在引用自己，当计数器的值为 0 时，就意味着已经没有客户程序再引用它了，这时组件对象就可自行销毁了。

基接口由系统实现，用户只需实现自定义的接口。

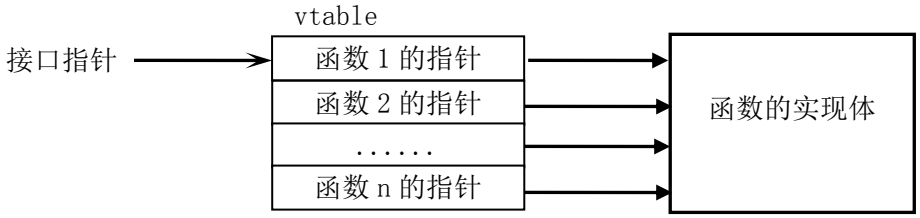



图 7-3 接口指针指向接口的 vtable

**②调度接口 IDispatch 与调度**

由上述可知,客户程序须通过接口指针才能调用接口的函数,从而获得组件提供的服务。但是，并不是所有的计算机语言都提供指针，典型的如 Office 的自动化语言 VBA、脚本语言 VBScript 及 JavaScript 等就不支持指针,那么这些语言所使用的组件必须支持调度接口 IDispatch(又称自动化接口)。该接口的 IDL 描述为：

```
interface IDispatch : IUnknown{
  HRESULT _stdcall GetTypeInfoCount([out]unsigned int *pctinfo);
  HRESULT _stdcall GetTypeInfo(
    [in]unsigned int itinfo,
    [in]unsigned long lcid,
    [out]void **pptinfo);
  HRESULT _stdcall GetIDsOfNames(
    [in]GUID *riid,
    [in]char **rgszNames,
    [in]unsigned int cNames,
    [in]unsigned long lcid,
    [out]long *rgdispid);
  HRESULT _stdcall Invoke(
```

```
    [in]long dispidMember,
    [in]GUID *riid,
    [in]unsigned long lcid,
    [in]unsigned short wFlags,
    [in]DISPPARAMS *pdispparams,
    [out]VARIANT *pvarResult,
    [out]EXCEPINFO *pexcepinfo,
    [out]unsigned int *puArgErr);
};
```

调度接口把组件中的每个方法(函数)及属性都编上号，当客户程序要调用这些方法和属性时，就把它们的编号传给 IDispatch 接口，IDispatch 接口再根据这些编号调用相应的方法和属性。当然实际的过程远比这复杂，因为调用方法和属性时还传递参数、参数类型及返回类型等，这些调度的工作由 IDispatch 接口中的 Invoke 函数来完成。

**③接口标识**

在应用程序中，标识是识别对象的依据。COM 规定，不仅组件(类)应该有标识，而且接口也应该有标识，并且该标识不仅能被不用语言编写的客户程序所识别，而且不能出现重复问题。为此，COM 对组件及组件接口的标识进行了特殊的规定。

众所周知，二进制数据是所有计算机程序设计语言都能识别的信息。因此，COM 规定；组件及组件接口必须使用一个 128 位的二进制数字来标识；同时，为使标识不出现重复现象，这个二进制标识不是由组件设计者来定义，而是由编译器按照产生唯一数字的特殊算法随即产生。这个 128 位的二进制标识就称为全局唯一标识符(Globally Unique Identifier,GUID)或通用唯一标识符(Universally Unique Identifier,UUID)。例如，前面列出的标识 IMath 接口的 UUID 为：

uuid(4CF4F598-BB9D-4DCA-BCF0-C0AD32EBCC47)

用于标识接口的标识符称为 IID(Interface Identifier)，而用来标识组件(类)的标识符称为 CLSID(Class Identifier)。

*(2)实现接口*

将组件接口的定义和实现分离开来是 COM 的一个重要特征。定义了接口后，支持 COM 的任何语言工具都可用来实现接口、创建组件。本文开发的 Math 组件实例，统一使用 ATL 框架定义接口和实现接口。

ATL 向导提供的用来实现组件接口的 C++类 CMath 的头文件定义为：

```
//Math.h : Declaration of the CMath

#ifndef __MATH_H_
#define __MATH_H_

#include "resource.h"       //main symbols

//CMath
class ATL_NO_VTABLE CMath :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CMath,&CLSID_Math>,
```

```
  public IMath,
  public IMath2
{
public :
  CMath()
  {
  }

  DECLARE_REGISTER_RESOURCEID(IDR_MATH)
  DECLARE_PROJECT_FINAL_CONSTRUCT()

  BEGIN_COM_MAP(CMath)
    COM_INTERFACE_ENTRY(IMath)
    COM_INTERFACE_ENTRY(IMath2)
  END_COM_MAP()

  // IMath
  public :
    STDMETHOD(Subtract)(/*[in]*/long IOp1,/*[in]*/long IOp2,/*[out,retval] */long *plResult);
    STDMETHOD(Add)(/*[in]*/long IOp1,/*[in]*/long IOp2,/*[out,retval] */long *plResult);
  // IMath2
    STDMETHOD(Sum)(/*[in]*/short sArraySize,/*[in]*/short sArray[],/*[out,retval] */long
*plResult);
};

#endif //__MATH_H_
```

对应的实现文件为：

```
//Math.cpp : Implementation of Cmath

#include "stdafx.h"
#include "Math_Server.h"
#include "Math.h"

//IMath methods
STDMETHODIMP CMath::Add(long IOp1,long IOp2,long *plResult)
{
  *plResult=IOp1 + IOp2;
  return S_OK;
}

STDMETHODIMP CMath::Subtract(long IOp1,long IOp2,long *plResult)
{
  *plResult=IOp1 - IOp2;
```

```
  return S_OK;
}

//IMath New method

STDMETHODIMP CMath::Sum(short sArraySize,short sArray[],long *plResult)
{
  *plResult= 0 ;
  while(sArraysize)
  {
    *plResult += sArray [--sArraySize]
  }
  return S_OK;
}
```

*(3)生成并注册类型库*

类型库被公认为是 IDL 文件的二进制版本，它包含组件、组件接口及接口方法的二进制描述，诸如 Visual Basic、Visual J++、Visual C++、Delphi 等许多语言环境可以识别、访问类型库中的信息。在编译组件工程时，开发工具自动将接口定义的 IDL 文件转换成对应的类型库文件。这样，客户程序使用组件时就可从类型库中查找组件接口的相关信息。

但是，实现接口的组件与使用组件的客户程序有可能不在同一台计算机上，为此 COM 规定，组件接口定义(类型库)及实现接口的组件位置等相关信息须存放到系统注册表中。这样，客户程序请求创建组件时，COM 运行库就可从系统注册表中查找组件的相关信息，并据此创建客户所需的组件。

ATL 在编译、链接生成组件文件(DLL 或 EXE)时，自动完成对组件相关信息的系统注册工作。

(4)编写组件客户程序

该实例编写的客户程序为一简单的 Visual C++控制台程序，其程序代码如下：

```
//Math_Client.cpp
#include <windows.h>
#include <iostream.h>
//Guids,interface declarations,etc.
#include "..\Math_Server\Math_Server_i.c"
#include "..\Math_Server\Math_Server.h"
int main(int argv,char *argc[])
{
  cout<<"Initializing COM" <<endl;
  if (FAILED(CoInitialze(NULL)))
  {
    cout<<"Unable to initialize COM"<<endl;
    return -1;
  }
//Use CoCreateInstance
```

```
  IMath *pMath;
  HRESULT hr = CoCreateInstance(CLSID_Math,
                                NULL,
                                CLSCTX_INPROC,
                                IID_IMath,
                                (void**)&pMath);
  if (FAILED(hr))
  {
    cout.setf(ios::hex,ios::basefield);
    cout<<"Failed to create server instance. HR="<<hr<<endl;
    CoUnitialize();
    return -1;
  }
  cout<<"Instance created"<<endl;

  //Access IMath interface
  long result;
  pMath->Add(134,353,&result)
  cout<<"134 + 353 = "<<result<<endl;
  pMath->Subtract(134,353,&result)
  cout<<"134 - 353 = "<<result<<endl;

  //Access IMath2
  IMath2 *pMath2;
  hr=pMath->QueryInterface(IID_IMath2,
                           (void**)&pMath2);
  if(SUCCEEDED(hr))
  {
    short sArray[3]={3,4,5};
    pMath2->Sum(3,sArray,&result);
    cout<<"3 + 4 + 5 ="<<result<<endl;
  }

  //Release all of interfaces
  if(pMath)
    pMath->Release();
  if(pMath2)
    pMath2->Release();
  cout<<"Shutting down COM"<<endl;
  CoUninitialize();

  return 0;
}
```

使用组件前，需初始化 COM 运行库(CoInitialize)；使用完毕，需退出 COM 运行库

(CoUnInitialize)。Intel Visual Fortran 9.0 在使用自动化服务器和 ActiveX 控件时，采取了相同的处理步骤。

使用组件时，先调用 COM API 函数 CoCreateInstance 创建组件对象，并获取组件对象的缺省接口指针，通过该指针调用接口的办法；若要调用另一个接口的办法，则先通过 QueryInterface 函数获得该接口指针。AddRef 函数的调用由系统完成。

### 7.1.2 自动化

自动化(以前称为 OLE 自动化)允许一个应用程序驱动另一个应用程序。驱动其他程序的程序称为自动化客户端或者自动化控制器，被驱动的程序称为自动化服务器或自动化组件。像 Excel、Word、AutoCAD、ActiveX 控件等都可用作自动化服务器；而 Visual Basic、Visual C++、Intel Visual Fortran 等都可作为自动化客户端。

要使一个应用程序成为自动化服务器，必须通过自动化接口 IDispatch 向外公开其自动化对象的方法和属性。换句话说，自动化客户端通过 IDispatch 接口访问自动化服务器所提供的服务。尽管允许为自动化对象实现 vtable 接口或 IDispatch 接口，或者同时实现这两个接口(双重接口)，但实际开发中，几乎总是为自动化对象实现双重接口，以便充分利用双重接口的下列优点：

(1)接口方法既可在编译时通过 vtable 接口静态绑定，又可在运行时通过 IDispatch 接口动态绑定；

(2)支持函数指针的自动化控制器可以使用 vtable 接口，从而获得较高的执行性能；

(3)先前使用 IDispatch 接口的自动化控制器仍可正常工作；

(4)C++可高效访问 vtable 接口；

(5)兼顾 Visual Basic 创建的自动化服务器(双重接口)。

### 7.1.3 Active 控件

ActiveX 是指 COM 对象的封装技术。ActiveX 控件是 VBX 的后继产品，曾称为 OLE 定制控件 OCX(OLE Custom Control)，现在它是一种 COM 组件，作为进程内服务器(通常是一个小型 COM 对象)加以实现，可以嵌入到容器程序中运行。ActiveX 控件通常封装在 DLL 中，其扩展名为 OCX。

ActiveX 控件的容器程序通过使用 ActiveX 控件的属性和方法来操作控件，并可通过 ActiveX 控件事件与其进行交互。ActiveX 控件的重要用途之一是为 Web 页提供计算能力，如果将某个应用程序设计成 ActiveX 控件，那么就可以把它嵌入 Web 页，当用户登录该 Web 页时，系统会自动下载 ActiveX 控件并在 Web 页中运行它。

## 7.2 Intel Fortran 模块向导

Intel Fortran 模块向导的作用是：依据 COM 组件注册的类型库信息，创建相应的 Fortran 包装程序(模块)。

在.NET 环境中，Intel Fortran 模块向导位于“工具”菜单中，如图 7-4 所示。




图 7-4 工具菜单下的 Intel Fortran 模块向导

Intel Fortran 模块向导分 2 步进行设置：

**A.第一步**

向导启动后，第一步展示图 7-5 所示的选取组件类型库对话框。用户可从对话框“COM”选项卡列表中直接选取在系统中注册的组件，也可点击“Browse for Type Library...”按钮，查找系统中的特定组件类型库。

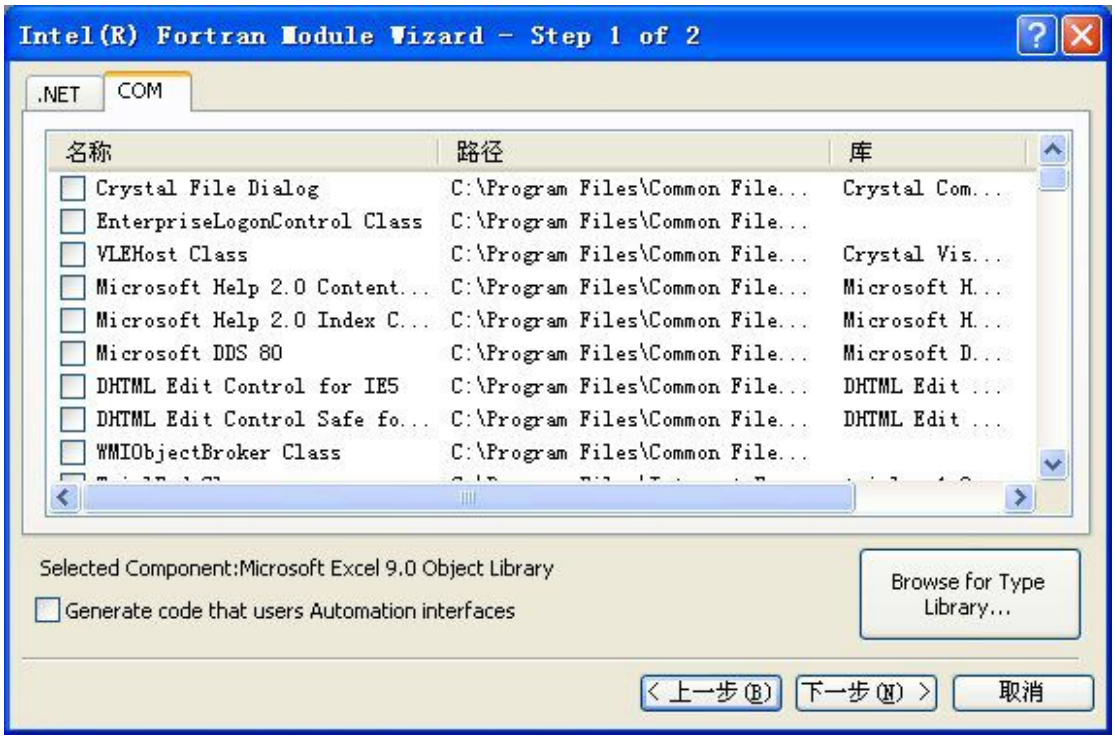



图 7-5 Intel Fortran 模块向导第 1 步选取组件类型库

自动化服务器和 ActiveX 控件都支持自动化接口接口(IDispatch)，为使向导生成自动化接口程序，应该核取“Generate code that users Automation interfaces”选项。

**B.第 2 步**

向导第二步列出第 1 步选取组件的所有接口供用户选择，如图 7-6 所示。用户可从中选取特定的接口，也可点击“Select All”按钮选取组件的所有接口。

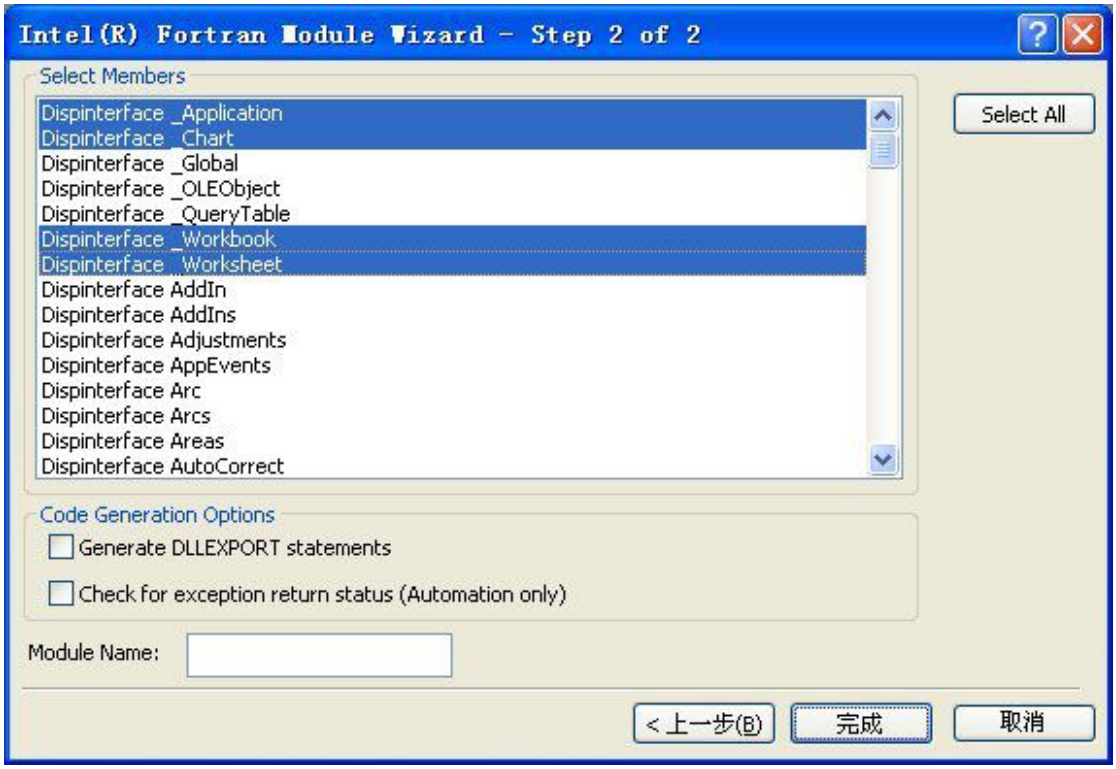



图 7-6 Intel Fortran 模块向导第 2 步选取组件窗口

如果计划将向导生成的模块置于动态链接库 DLL 中供其它应用程序使用，那么就应该核取“Generate DLLEXPORT statements”选项，以便在生成的模块里程中插入 DLLEXPORT 语句，使其他应用程序可以访问 DLL 中的模块例程。

假如第 1 步核取了“Generate code that users Automation interfaces”选项，第 2 步中的“Check for exception return status (Automation only)”选项被激活。如果要向导生成检查自动化对象方法、属性调用的返回状态代码，那么就核取该选项，以便在调用失败时展示错误信息对话框。

向导最后生成包含一个或多个模块的源文件(.F90)，文件内容包括：

(1)派生类型(Derived-type)声明，用来声明类型库中的数据结构；

(2)常量(PARAMETER)声明，包括类型库中的各种标识符和枚举数据；

(3)接口、模块例程定义，用来包装类型库中代表属性、方法或事件的所有例程。

Fortran 直接调用模块中的包装例程，而包装例程又调用相应的系统例程，这样一来，就大大简化了使用 COM、自动化对象的编程工作。事实上，在模块向导的帮助下，Intel Visual Fortran 开发 COM、自动化客户程序的效率甚至可以和 Visual Basic 6.0 相媲美。

## 7.3 使用自动化服务器

本节结合实例，讲解自动化服务器的使用。

【例 7-2】Intel Visual Fortran 作为自动化客户，控制 Excel 2000 自动化服务器绘制图表，程序运行结果见图 7-7。

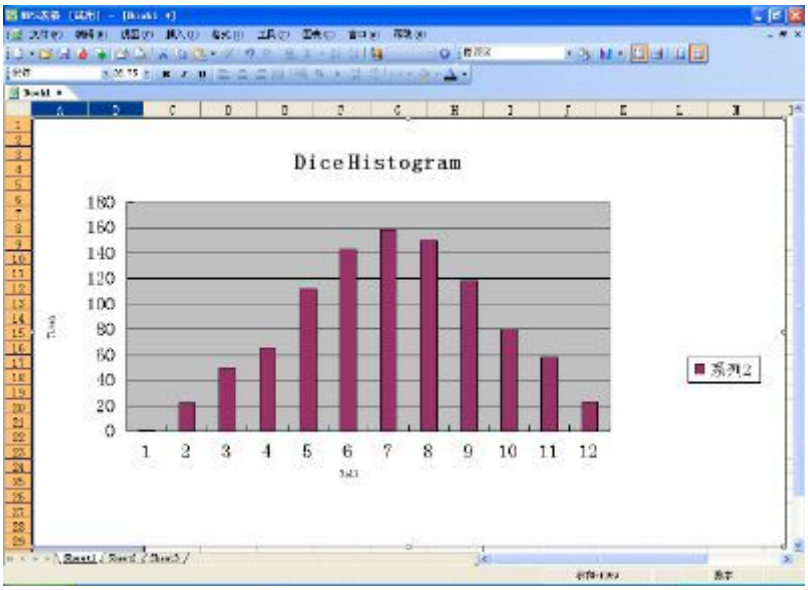



图 7-7 Intel Fortran 控制 Excel 2000 绘制的图表(不好意思，用 WPS 顶替)

*(1)利用模块向导生成 Excel 2000 包装程序*

启动 Intel Fortran 模块向导，从对话框“COM”选项卡列表中选取“Microsoft Excel 9.0 Object Library”组件，并核取“Generate code that users Automation interfaces”选项，从随后展示的接口列表中选取下列接口：

_Application
_Chart
_Worksheet
Axes
Charts
Range
Workbooks
Worksheets

同时核取“Check for exception return status (Automation only)”选项。最后，命名生成的模块文件为 Excel9.F90。

由于篇幅所限，下面仅列出其中的一个包装例程$Application_GetWorkbooks，以对 Intel Fortran 模块向导生成的包装例程的结构有一个大致的了解：

```
Integer(INT_PTR_KIND()) Function $Application_GetWorksheets($OBJECT,$STATUS)
  Implicit None
  Integer(INT_PTR_KIND()),Intent(In) :: $OBJECT     !Object Pointer
  !DEC$ ATTRIBUTES VALUE :: $OBJECT
  Integer(4),Intent(OUT),Optional :: $STATUS      !Method status
  !DEC$ ATTRIBUTES REFERENCE     :: $STATUS
  Integer(INT_PTR_KIND()),Volatile :: $RETURN
  Integer(4) $$STATUS
  Integer(INT_PTR_KIND()) invokeArgs
  invokeArgs = AUTOALLOCATEINVOKEATGS()
```

```
  Call AUTOADDARG( invokeArgs , 'Workbooks', $RETURN , AUTO_ARG_OUT , VT_DISPATCH )
  $$STATUS = AUTOGETPROPERTYBYID( $OBJECT , 572 , invokeArgs )
  If ( $$STATUS == DISP_E_EXCEPTION ) Call $$DisplayError( invokeArgs )
  If ( PRESENT($STATUS) ) $STATUS = $$STATUS
  $Application_GetWorkbooks = $RETURN
  Call AUTODEALLOCATEINVOKEARGS( invokeArgs )
End Function $Application_GetWorksheets
```

包装例程的第一个参数总是$OBJECT,它是一个指向自动化对象 IDispatch 接口的指针。针对 COM 和自动化，Intel Visual Fortran 分别提供了 IFCOM 和 IFAuto 运行库，包装例程中使用的诸如 AUTOALLOCATEINVOKEATGS、AUTOADDATG 等均为 IFAuto 运行库例程。正因为有包装例程对自动化对象接口的转换,使得 Fortran 客户程序可以直接获取或设置自动化对象的属性并调用其方法，而不必关心像参数的封送、调度这样一些自动化细节问题。

*(2)编写 Excel 的客户程序*

操作 Excel 的客户程序可以是任何类型的 Intel Fortran 工程，实例采用的是 Fortran 控制台客户程序。在利用 Intel Fortran 工程向导生成骨架工程后,将上述向导生成的 Excel 模块插入骨架工程中。在编写客户程序代码前，有必要先了解 Excel 的对象模型。

作为自动化服务器，Excel 提供了类似于目录树的对象层次模型，如图 7-8 所示。访问对象时，先要创建根对象(Application)，然后才能逐级访问子对象，这个过程类似于目录中的文件查找。例如，要访问 Range 对象，先要获取应用对象的下一级对象——工作簿集合，由工作簿集合获取当前工作簿；再由工作簿对象获取其下一级对象——工作表集合，由工作表集合获取当前工作表；最后，才能通过工作表对象访问 Range 对象。清除对象时，则要由下到上进行：先释放 Range 对象，然后是工作表对象……最后才是 Excel 应用对象。

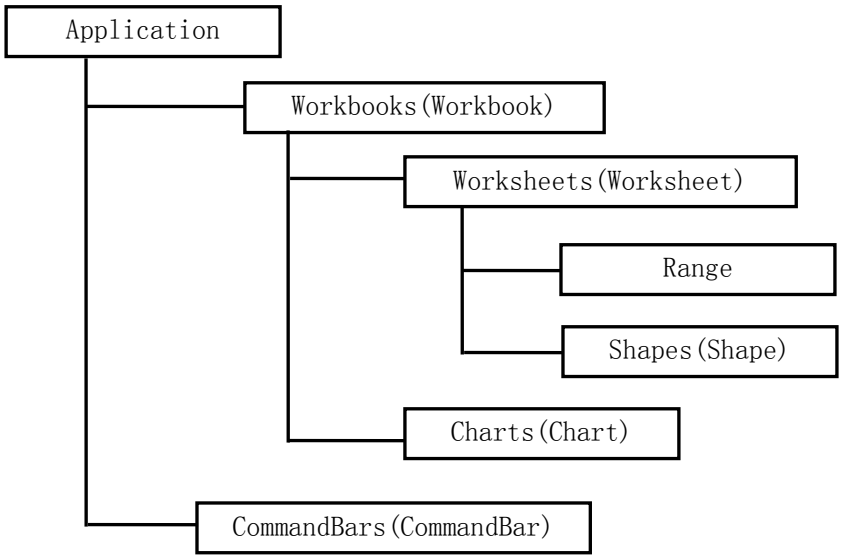



图 7-8 Excel 对象层次模型简图

操作 Excel 2000 的 Intel Fortran 客户程序的主程序为：

```
Program ExcelSample
  Use IFCOM
  Use IFAuto
  Use IFCore
```

```
Use GOBJECTS
Use Excel9    !Execl module generated by the Module Wizard
Implicit None
!Variables
Integer*4 status
Integer*4 loopCount
Integer*4 die1,die2
Integer*4 roll
Integer*4 maxScale
Character(Len=32) :: loopc
Integer i
Real*4 rnd
Integer*2,DIMENSION(1:12) :: cellCounts
!Variant arguments
Type(VARIANT) :: vBSTR1 , vBSTR2 , vBSTR3
Type(VARIANT) :: vInt
!Initialize object pointers
Call InitObjects()
!Create an Excel object
Call COMInitialize(status)
Call COMCreateObject( "Excel.Application.9" , excelApp , status )
If ( excelApp == 0 ) then
  Write(*,'("Unable to create Excel object;Aborting")')
End If
Call $Application_SetVisible( excelApp , .True. )
!Get the WORKBOOKS object
workBooks = $Application_GetWorkboos( excelApp , $STATUS = status )
!Open the specified spreadsheet file(note:specify the full file path)
workBook = Workbooks_Open( workBooks , "X:\path\FileName.xls" , $STATUS = status )
!Get the worksheet
vInt.VT          = VT_I4
vInt.VU.LONG_VAL = 1
workSheet = $Workbook_GetActiveSheet( workBook , status )
!Initialize the call counts
Do i = 1 ,12
  range = 0
  cells(i) = range
  cellCounts(i) = 0
End Do
!Create a new chart
Call VariantInit(vBSTR1)
vBSTR1.VT = VT_BSTR
bstr1 = ConvertStringToBSTR("A1")
vBSTR1.VU.PTR_VAL = bstr1
```

```
  Call VariantInit( vBSTR2 )
  vBSTR2.VT = VT_BSTR
  bstr2 = ConvertStringToBSTR("L1")
  vBSTR2.VU.PTR_VAL = bstr2
  range = $Worksheet_GetRange( worksheet , vBSTR1 ,vBSTR2 , status )
  status = VariantClear( vBSTR1 )
  bstr1 = 0
  status = VariantClear( vBSTR2 )
  bstr2 = 0

  status = AUTOSETPROPERTY( range , "VALUE" , cellCounts )
  vInt = Range_Select(range , status )
  charts = $Workbook_GetCharts( workbook , $STATUS = status )
  chart = Charts_Add( charts , $STATUS = status )

  !Invoke the ChartWizard to format the chart
  Call VariantInit( vInt )
  vInt.VT = VT_I4
  vInt.VU.LONG_VAL = 11
  Call VariantInit( vBSTR1 )
  vBSTR1.VT = VT_BSTR
  bstr1 = ConvertStringToBSTR("Dice Histogram")
  vBSTR1.VU.PTR_VAL = bstr1
  Call VariantInit(vBSTR2)
  vBSTR2.VT = VT_BSTR
  bstr2 = ConvertStringToBSTR("Roll")
  vBSTR2.VU.PTR_VAL = bstr2
  Call VariantInit(vBSTR3)
  vBSTR3.VT = VT_BSTR
  bstr3 = ConvertStringToBSTR("Times")
  vBSTR3.VU.PTR_VAL = bstr3
  Call $Chart_ChartWizard( chart , Gallery = vInt , Title = vBSTR1 , CategoryTitle = vBSTR2 ,
ValueTitle = vBSTR3 , $STATUS = status )
  status = VariantClear( vBSTR1 )
  bstr1 = 0
  status = VariantClear( vBSTR2 )
  bstr2 = 0
  status = VariantClear( vBSTR3 )
  bstr3 = 0
  !Determine the number of times to roll the dice
  If ( NARGS > 1 ) then
    Call GetArg( 1_2 , loopc )
    Read( loopc , * ) loopCount
  Else
```

```
    loopCount = 1000
  End If
  !Set some chart properties
  Call VariantInit(vInt)
  vInt.VT = VT_I4
  vInt.VU.LONG_VAL = xlValue
  valueAxis = $Chart_Axis( chart , vInt , xlPrimary , $STATUS = status )
  maxScale = loopCount / 5
  status = AUTOSETPROPERTY( valueAxis , "MaximumScale" , maxScale )
  !Loop the specified number of times
  Call Seed(RND$TIMESEED)
  Do i = 1 , loopCount
    Call Random(rnd)
    die1 = NINT( (rnd*6) + 0.5 )
    Call Random(rnd)
    die2 = NINT( (rnd*6) + 0.5 )
    roll = die1 + die2
    cellCounts(roll) = cellCounts(roll) + 1
    If ( MOD(i,200) == 0 ) then
      status = AUTOSETPROPERTY( range , "VALUE" , cellCounts )
    End If
  End Do
  !Release all objects
  Call ReleaseObjects()
  Call COMUnInitialize()
End Program ExcelSample
```

从中可以看出，Intel Fortran 使用自动化服务的一般步骤为：

● 调用 COM 库例程 COMInitialize，初始化 COM 库；

● 调用 COM 库例程 COMCreateObject，获取自动化对象的 IDispatch 接口指针；

● 利用向导生成的包装例程，调用自动化对象接口的属性和方法(设置属性则通过调用自动化例程 AUTOSETPROPERTY 来实现)；

● 调用 COM 库例程 COMReleaseObject，由下到上逐级释放对象接口指针；

● 调用 COM 库例程 COMUnInitialize，退出 COM 库。

在创建了 Excel 应用对象 excelApp 后，操作 Excel 绘制图表的骨架代码(Visual Basic 格式)如下：

```
workbooks = excelApp.GetWorkbooks()
workbook = workbooks.Open(spreadsheet)
worksheet = workbook.GetActiveSheet
range = worksheet.GetRange("A1","L1")
range.Select()
charts = workbook.GetCharts()
chart = charts.Add()
chart.ChartWizard(gallery=chartType,title=title,categoryTitle=title,valueTitle=title)
```

```
valueAxis=chart.Axes(type=XlValue,axisGroup=xlPrimary)
valueAxis.MaximumScale(loopCount/5)
```

在初始化 COM 库后、退出 COM 库前，分别调用的 InitObjects 和 ReleaseObjects 例程包含在模块 GOBJECTS 中。另外，该模块还包含了对象指针等变量声明：

```
Module GOBJECTS
  Implicit None
  !Object Pointers
  Integer*4 execelApp
  Integer*4 workBooks
  Integer*4 workBooks
  Integer*4 workSheets
  Integer*4 workSheet
  Integer*4 range
  Integer*4 charts
  Integer*4 chart
  Integer*4,DIMENSION(1:12) :: cells
  Integer*4 categoryAxis
  Integer*4 valueAxis
  !BSTRs
  Integer*4 bstr1,bstr2,bstr3
  Contains
    Subroutine InitObject()
      Integer i
      execelApp = 0
      workBooks = 0
      workBoos  = 0
      workSheets= 0
      workSheet = 0
      range     = 0
      charts    = 0
      chart     = 0
      Do i = 1 , 12
        cells(i) = 0
      End Do
      categoryAxis = 0
      valueAxis    = 0
      bstr1        = 0
      bstr2        = 0
      bstr3        = 0
    End Subroutine InitObject

    Subroutine ReleaseObjects()
      Use IFCOM
```

```
    Integer*4 status
    Integer*4 i
    If ( range /= 0 ) status = COMReleaseObject( range )
    If ( chart /= 0 ) status = COMReleaseObject( chart )
    If ( charts /= 0 ) status = COMReleaseObject( charts )
    If ( workSheet /= 0 ) status = COMReleaseObject( workSheet )
    If ( workSheets /= 0 ) status = COMReleaseObject( workSheets )
    If ( workBook /= 0 ) status = COMReleaseObject( workBook )
    If ( workBooks /= 0 ) status = COMReleaseObject( workBooks )
    Do i = 1 , 12
      If ( cells(i) /= 0 ) status = COMReleaseObject(cells(i))
    End Do
    If ( categoryAxis /= 0 ) status = COMReleaseObject( categoryAxis )
    If ( valueAxis /= 0 ) status = COMReleaseObject( valueAxis )
    If ( excelApp /= 0 ) status = COMReleaseObject( excelApp )
    If ( bstr1 /= 0 ) status = COMReleaseObject( bstr1 )
    If ( bstr2 /= 0 ) status = COMReleaseObject( bstr2 )
    If ( bstr3 /= 0 ) status = COMReleaseObject( bstr3 )
  End Subroutine ReleaseObjects

End module GOBJECTS
```

由于篇幅所限，没有列出主程序引用的、由模块向导生成的、完整的 Excel 包装程序。

（由于录入人水平及使用计算机内软件等多方面的限制，没有办法将本章的实例实现，也没有验证手工敲入的代码是否正确，甚至本章内容的截图都是伪造的。很抱歉~~如果对以上内容有疑问或发现了错误，请翻阅原书及其他资料，并以其他资料为准，感谢你的谅解）

## 7.4 使用 AcitveX 控件

在第 5 章使用对话框和控件中，没有讲解 ActiveX 控件的使用。事实上，对话框例程不但支持普通控件，而且也支持 ActiveX 控件，因此对话框也可以作为 ActiveX 控件的容易。在使用上，ActiveX 控件有下列几点特别之处：

(1)在向对话框插入控件资源时，普通控件可直接从工具箱中选取；ActiveX 控件则需在对话框上点击鼠标右键，从弹出的上下文菜单中选择“插入 ActiveX 控件”，再从系统注册的 ActiveX 控件列表中选取特定的 ActiveX 控件。

(2)在初始化对话框(DlgInit)前，须调用 COMInitialize 例程初始化 COM 库；在清除对话框(DlgUnInit)时，还需调用 COMUnInitialize 例程退出 COM 库。在利用 Intel Fortran Windows 程序向导设置工程时，如果核取了“Application will use ActiveX controls”选项，那么就会在主程序中自动添加这两条语句。

(3)如果需要调用 ActiveX 控件的方法，或设置其事件处理柄，则需要使用 Intel Fortran 模块向导生成 ActiveX 控件的包装程序。

【例 7-3】编写一个基于对话框的 Windows 程序，控制 Windows 媒体播放器(Active 控件)播放音视频文件。要求：提供“打开”和“关闭”按钮，分别打开和关闭音视频文件；

设置事件处理柄，响应媒体播放器控件上发生的鼠标双击事件。程序运行结果如图 7-9 和图 7-10 所示。(此实例可参阅 CVF6.5 安装光盘\INFO\DF\SAMPLES\DIALOG\MMPLAYER 目录下示范代码，录入人对此实例尚未实现，所有截图均属伪造，代码未曾验证，敬请谅解)

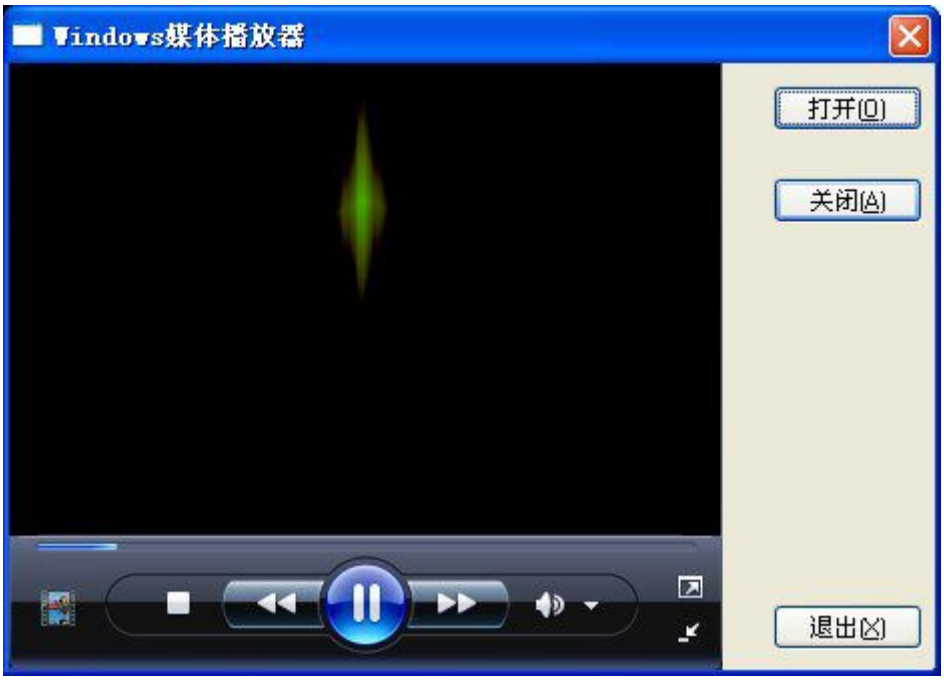



图 7-9 Windows 媒体播放器




图 7-10 双击控件时展示的信息框

*(1)插入 ActiveX 控件*

先利用 Intel Fortran Windows 程序向导生成对话框骨架工程，在设置时核取“Application will use ActiveX controls”选项；然后，在对话框上插入“Windows Media Player”控件(wmp.dll)，并重新布置对话框。各控件的属性设置如表 7-1 所示。

表 7-1 对话框及其控件属性设置

| 对话框及其控件 | 属性类别 | 属性项 | 属性值 |
|---|---|---|---|
| 窗体 | 外观 | Caption | Windows 媒体播放器 |
| | 杂项 | ID | IDD_MMPlayer_DIALOG |
| 按钮 1 | 外观 | Caption | 打开(&O) |
| | 杂项 | ID | IDM_OPEN |
| 按钮 2 | 外观 | Caption | 关闭(&C) |
| | 杂项 | ID | IDM_CLOSE |
| 按钮 3 | 外观 | Caption | 退出(&X) |
| | 杂项 | ID | IDM_EXIT |
| ActiveX 控件 | 杂项 | ID | IDC_PLAYER |

*(2)生成 ActiveX 控件的包装程序*

实例程序要用到 Windows Media Player 控件的 Colse 方法，并要相应控件的双击鼠标事件，因此须利用 Intel Fortran 模块向导生成该控件的包装程序(ActiveMovie 模块)。ActiveX 控件本质上是一种自动化服务器，所以，在模块向导设置时核取“Generate code that users Automation interfaces”和“Check for exception return status (Automation only)”选项。

Windows Media Player 控件的 Close 方法包装例程为：

```
!Closes the Media
Subroutine IWMPPlayer2_close($OBJECT,$STATUS)
!DEC$ ATTRIBUTES DLLEXPORT :: IWMPPlayer2_close
  Implicit None
  Integer(INT_PTR_KIND()),Intent(In) :: $OBJECT    !Object Pointer
  !DEC$ ATTRIBUTES VALUE :: $OBJECT
  Integer(4),Intent(OUT),Optional :: $STATUS     !Method status
  !DEC$ ATTRIBUTES VALUE :: $STATUS
  Integer(4) $$STATUS
  Integer(INT_PTR_KIND()) invokeArgs
  invokeArgs = AUTOALLOCATEINVOKEARGS()
  $$STATUS = AUTOINVOKE( $OBJECT , 3 , invokeArgs )
  If ( Present($STATUS) ) $STATUS = $$STATUS
  Call AUTODEALLOCATEINVOKEARGS( invokeArgs )
End Subroutine IWMPPlayer2_close
```

*(3)调用 COM 库*

在主函数中，Intel Fortran Windows 程序向导已添加了初始化 COM 库的 COMInitialize 语句和退出 COM 库德 COMUnInitialize 语句，如下列主函数代码所示：

```
integer*4 function WinMain( hInstance, hPrevInstance, lpszCmdLine, nCmdShow )
!DEC$ IF DEFINED(_X86_)
!DEC$ ATTRIBUTES STDCALL, ALIAS : '_WinMain@16' :: WinMain
!DEC$ ELSE
!DEC$ ATTRIBUTES STDCALL, ALIAS : 'WinMain' :: WinMain
!DEC$ ENDIF
    Use User32
    Use kernel32
    Use MMplayerGlobals
    implicit none
    integer*4 hInstance
    integer*4 hPrevInstance
    integer*4 lpszCmdLine
    integer*4 nCmdShow
    include 'resource.fd'
    external MMplayerSub
    external MMplayerOpen
    external MMplayerClose
```

```
    external MMplayerDblClk
     ! Variables
    type (T_MSG)            mesg
    integer*4               ret
    logical*4               lret
    ghInstance = hInstance
    ghModule = GetModuleHandle(NULL)
    ghwndMain = NULL
    call COMINITIALIZE(ret)
    lret = DlgInit(IDD_MMPLAYER_DIALOG, gdlg)
    if (lret == .FALSE.) goto 99999
    lret = DlgSetSub(gdlg, IDD_MMPLAYER_DIALOG, MMplayerSub)
    lret = DlgSetSub(gdlg, IDM_OPEN, MMplayerOpen)
    lret = DlgSetCtrlEventHandler(gdlg, IDC_PLAYER,MMPlayerDblClk,6506)
    lret = DlgModeless(gdlg, nCmdShow)
    if (lret == .FALSE.) goto 99999
    ! Read and process messsages
    do while( GetMessage (mesg, NULL, 0, 0) )
       if ( DlgIsDlgMessage(mesg) .EQV. .FALSE. ) then
           lret = TranslateMessage( mesg )
           ret  = DispatchMessage( mesg )
       end if
    end do
    call DlgUninit(gdlg)
    call COMUNINITIALIZE()
    WinMain = mesg.wParam
    return
99999 &
    ret = MessageBox(ghwndMain, "Error initializing application mmplayer"C, &
                     "Error"C, MB_OK)
    call COMUNINITIALIZE()
    WinMain = 0
end
```

程序中，对包含 ActiveX 控件的对话框进行的操作，与普通控件情况下的错作没有什么两样。这里，为对话框、“打开”按钮和“关闭”按钮分别注册了 MMPlayerSub、MMPlayerOpen 和 MMPlayerClose 回调例程；为媒体播放器控件设置了 MMPlayerDblClk 事件处理柄。

*(4)设置 ActiveX 控件的属性、调用其方法*

下面先给出三个回调例程的实现代码：

```
SUBROUTINE MMplayerSub( dlg, id, callbacktype )
!DEC$ ATTRIBUTES DEFAULT :: MMPlayerSub
  Use User32
  Use IFLogM
```

```
  Use IFAuto
  implicit none
  type (dialog) dlg
  integer id, callbacktype
  include 'resource.fd'
  integer obj, iret
  logical lret
  if (callbacktype == dlg_destroy) then
     !  Reset the filename to "" to "close" the current file
      lret = DlgGet(dlg, IDC_PLAYER, obj)
      iret = AUTOSETPROPERTY(obj, "URL", "")
      call PostQuitMessage(0)
  endif
END SUBROUTINE MMplayerSub

SUBROUTINE mmplayerOpen( dlg, id, callbacktype )
!DEC$ ATTRIBUTES DEFAULT :: MMPlayerOpen
  Use IFLogM
  Use IFCOM
  Use IFAuto
  Use IFWin
  implicit none
  type (dialog) dlg
  integer id, callbacktype
  include 'resource.fd'
  type(T_OPENFILENAME) fileinfo
  character * MAX_PATH filename
  character * MAX_PATH filefilter
  integer obj, iret
  logical lret

  if (callbacktype == dlg_clicked) then
    ! Display a file open dialog box to allow the user to choose the
    ! multimedia file to play
    fileinfo % lStructSize = sizeof(fileinfo)
    fileinfo % hwndOwner = dlg % hwnd
    fileinfo % hInstance = 0
      iret = lstrcpy(filefilter, &
          "All files (*.*)|*.*|&
          DirectShow files (*.mpg;*.mpa;*.mpv;*.mov;*.mpeg;*.enc;*.m1v;*.mp2)|&
          *.mpg;*.mpa;*.mpv;*.mov;*.mpeg;*.enc;*.m1v;*.mp2|&
          Audio files (.wav)|*.wav|&
          Video for Windows files (.avi)|*.avi||"C)
      call convertfilterstring(filefilter)
```

```
    fileinfo % lpstrFilter = loc(filefilter)
    fileinfo % lpstrCustomFilter = 0
    fileinfo % nMaxCustFilter = 0
    fileinfo % nFilterIndex = 0
    filename = ""C
    fileinfo % lpstrFile = loc(filename)
    fileinfo % nMaxFile = MAX_PATH
    fileinfo % lpstrFileTitle = 0
    fileinfo % nMaxFileTitle = 0
    fileinfo % lpstrInitialDir = 0
    fileinfo % lpstrTitle = 0
    fileinfo % Flags = OFN_HIDEREADONLY
    fileinfo % nFileOffset = 0
    fileinfo % nFileExtension = 0
    fileinfo % lpstrDefExt = 0
    fileinfo % lCustData = 0
    fileinfo % lpfnHook = 0
    fileinfo % lpTemplateName = 0

    lret = GetOpenFileName(fileinfo)
    if (lret .ne. 0) then
        lret = DlgGet(dlg, IDC_PLAYER, obj)
        iret = AUTOSETPROPERTY(obj, "URL", filename)
    endif
  endif
END SUBROUTINE mmplayerOpen
```

```
SUBROUTINE MMplayerClose( dlg, id, callbacktype )
!DEC$ ATTRIBUTES DEFAULT :: MMPlayerClose
  Use User32
  Use IFLogM
  Use IFAuto
  Use AvtiveMovie
  implicit none
  type (dialog) dlg
  integer id, callbacktype
  include 'resource.fd'
  integer obj, iret
  logical lret
  if (callbacktype == dlg_destroy) then
      lret = DlgGet(dlg, IDC_PLAYER, obj)
      call IWMPPlayer2_close( obj )
  endif
END SUBROUTINE MMplayerClose
```

在设置 ActiveX 控件属性和调用 AcitveX 控件方法之前，先要通过对话框例程 DlgGet 获取 ActiveX 控件的 IDispatch 接口指针。例如，上面 3 个回调例程都使用了下列语句：

lret = DlgGet(dlg, IDC_PLAYER, obj)

DlgGet 例程中，代表控件索引的第四个参数(可选)为 DLG_DISPATCH，由于它是 ActiveX 控件的缺省属性，故可以省略；实参 obj 存放的即位 IDispatch 接口指针。

设置 ActiveX 控件属性，需利用自动化库(IFAuto)例程 AUTOSETPROPERTY 来实现。例如，MMPlayerOpen 回调例程中采用下列语句设置 Media Player 控件的 URL 属性：

iret = AUTOSETPROPERTY(obj, "URL", filename)

调用 ActiveX 控件方法，则通过模块向导生成的相应包装例程来实现。例如，MMPlayerClose 回调例程中采用下列语句调用 Media Player 控件的 close 方法：

Call IWMPPlayer2_close( obj )

当中的 IWMPPlayer2_close 例程，即为 close 方法的包装例程。

*(5)响应 ActiveX 控件引发的事件*

除了属性、方法外，ActiveX 控件还拥有事件通知的能力，允许用户编写相应的事件处理程序，对 ActiveX 控件引发的事件做出响应。

实现控件的回调，需要使用 DlgSetSub 对话框例程为控件注册回调例程；而实现 ActiveX 控件的事件处理，则需要使用 DlgSetCtrlEventHandler 对话框例程为控件注册事件处理柄。DlgSetCtrlEventHandler 例程原型(当中的参数如表 7-2 所示)为：

Integer DlgSetCtrlEventHandler(dlg,controllID,handler,dispID,iID)

表 7-2 DlgSetCtrlEventHandler 例程中的参数

| 参数 | 描述 |
|---|---|
| dlg | 对话框类型变量，指 ActiveX 控件的容器 |
| controllID | 整型，指 ActiveX 控件 ID(来自.FD 文件) |
| handler | (外部的)事件处理柄，事件发生时被调用 |
| dispID | 整型，指事件接口中的方法 ID 号 |
| iID | T_GUID 类型的可选参数，指事件接口 iID。如果省略该参数，则使用 ActiveX 控件的缺省 iID |

实例程序为 Media Player 控件设置事件处理柄的语句为：

lret = DlgSetCtrlEventHandler(gdlg, IDC_PLAYER,MMPlayerDblClk,6506)

这里，为 Media Player 控件设置的事件处理柄为 MMPlayerDblClk；6506 是与该事件相对应控件方法成员(DoubleClick)ID 号，即该处理柄要处理的事件实际上是由控件事件接口(_WMPOCXEvents)下编号为 6506 的 DoubleClick 方法引发的。

要获取 ActiveX 控件的事件方法成员 ID 号，可以从 Intel Fortran 模块向导生成的包装程序的接口声明中查找；也可使用“OLE/COM 对象查看器”工具 查看 Windows Media Player 控件的类型库：该控件的缺省事件接口为_WMPOCXEvents，DoubleClick 是该接口下的一个方法成员，其 dispID 为 0x0000196a(十六进制)，十进制下为 6506；在包装程序中声明的接口为：

```
INTERFACE
    !Occurs when a user double-clicks the mouse
    !MEMBERID = 6506
    SUBROUTINE $WMPOCXEvents_DoubleClick($OBJECT,nButton,nShiftState,fX,fY)
        INTEGER(INT_PTR_KIND()), INTENT(IN)   :: $OBJECT      ! Object Pointer
```

```
        !DEC$ ATTRIBUTES VALUE :: $OBJECT
        INTEGER(2),INTENT(IN) :: nButton
        !DEC$ ATTRIBUTES VALUE :: $nButton
        INTEGER(2),INTENT(IN) :: nShiftState
        !DEC$ ATTRIBUTES VALUE :: $nShiftState
        INTEGER(4),INTENT(IN) :: fX
        !DEC$ ATTRIBUTES VALUE :: fX
        INTEGER(4),INTENT(IN) :: fY
        !DEC$ ATTRIBUTES VALUE :: fY
        !DEC$ ATTRIBUTES STDCALL    :: $WMPOCXEvents_DoubleClick
    END SUBROUTINE $WMPOCXEvents_DoubleClick
END INTERFACE
```

从中可以发现：MEMBERID = 6506。

由于_WMPOCXEvents 接口是 Windows Media Player 控件的缺省事件接口(前导下划线即代表缺省接口)，所以，程序中省略了 DlgSetCtrlEventHandler 例程中代表事件接口 iID 的最后一个参数。在模块向导生成的包装程序中，该事件接口的常量声明为：

```
    TYPE (GUID), PARAMETER :: IID__WMPOCXEvents = &
        GUID(#6BF52A51, #394A, #11D3, &
            CHAR('B1'X)//CHAR('53'X)//CHAR('00'X)//CHAR('C0'X)// &
            CHAR('4F'X)//CHAR('79'X)//CHAR('FA'X)//CHAR('A6'X))
```

这里，若将 IID__WMPOCXEvents 用作 DlgSetCtrlEventHandler 例程的最后一个实参，反而会导致编译失败。究其原因，表 7-2 所列的虚参 iID 的类型为 T_GUID，而实参 IID__WMPOCXEvents 的类型为 GUID。尽管这两种类型本质上是相同的，都代表事件接口的全局唯一标识符，但由于两者形式上不统一，所以不能在 Intel Fortran 编译器上通过。这不能不说是 Intel Fortran 当前版本中的一个疏忽。

该实例编写的事件处理柄 MMPlayerDblClk 的处理代码为：

```
SUBROUTINE MMPlayerDblClk($OBJECT,nButton,nShiftState,fX,fY)
!DEC$ ATTRIBUTES DEFAULT    :: MMPlayerDblClk
  Use User32
  Implicit None
    INTEGER(INT_PTR_KIND()), INTENT(IN)  :: $OBJECT     ! Object Pointer
    !DEC$ ATTRIBUTES VALUE :: $OBJECT
    INTEGER(2),INTENT(IN) :: nButton
    !DEC$ ATTRIBUTES VALUE :: $nButton
    INTEGER(2),INTENT(IN) :: nShiftState
    !DEC$ ATTRIBUTES VALUE :: $nShiftState
    INTEGER(2),INTENT(IN) :: fX
    !DEC$ ATTRIBUTES VALUE :: fX
  INTEGER(4),INTENT(IN) :: fY
    !DEC$ ATTRIBUTES VALUE :: fY
    Integer*4 ret
    ret = MessageBox( 0 , "用户在媒体播放器上双击了鼠标!"C , "事件通知"C , MB_ICONINFORMATION
+ MB_OK )
END SUBROUTINE MMPlayerDblClk
```

事件处理柄的接口，必须与包装程序中对应的事件方法接口相统一。

## 小　结

■ COM(组件对象模型)为开发组件软件提供了统一的标准和规范，按此标准和规范创建的组件模块，可以被支持 COM 的任何语言所调用。组件软件的运行，通常需要有 COM 运行库的系统注册表的帮助。

■ 基于 COM 的自动化技术，允许自动化客户端通过 IDispatch 接口访问自动化服务器对象提供的方法和属性。自动化服务器通常为进程外的 COM 组件，可以在自己的地址空间单独运行，如 Excel、Word、AutoCAD 等。

■ ActiveX 控件作为小型的进程内 COM 组件，可以嵌入到容器程序中运行。ActiveX 控件通过 IDispatch 接口向外公开其属性、方法及事件。

■ Intel Fortran 模块向导，依据 COM 组件注册的类型库信息，生成相应的 Fortran 90/95 模块，对自动化服务器、ActiveX 控件的接口进行相应的转换，极大地便利了 Intel Fortran 环境下对 COM 组件的操作。

■ 使用自动化服务器和 ActiveX 控件，均需要初始化 COM 库；使用完毕，还得清理、退出 COM 库。

■ 要访问自动化服务器对象提供的属性和方法，需要先获取 IDispatch 接口指针，然后按自动化对象层次模型由上到下逐级访问；使用完毕，还需由下到上逐级清除对象、释放内存资源。

■ 对话框例程即可用于普通控件，也可用于 ActiveX 控件。如果要访问 ActiveX 控件的方法，或对其事件做出响应，则需要利用 Intel Fortran 模块生成向导生成包装程序，然后直接调用控件的方法，或设置与事件方法同接口的事件处理柄，并在其中编写事件处理代码。

# 第 8 章　创建多线程应用程序

Windows 是一个多任务操作系统，由于它采用了较完善的进程与线程管理技术，从而使同时运行的多个应用程序可以无冲突、协调地进行工作。Intel Visual Fortran 9.0 通过提供运行库 IFMT 机 IFWin，支持开发多线程应用程序。

**本章主要内容：**

- 进程、线程及其优先性
- 线程操作
- 线程同步
- 线程局部存储
- 进程操作
- 多线程操作的相关例程

## 8.1 进程、线程及其优先性

### 8.1.1 进程、线程的基本概念

进程是拥有程序所有资源的对象，从程序设计角度可以把它看做是一个正在运行的应用程序的实例。进程由一个或多个线程组成，而线程是进程中的一个独立执行路径。CPU 按照一定的规则，分时运行线程来实现多个程序的“同时”运行。

一个应用程序(进程)至少需要一个线程，这个线程叫做主线程。当然，根据需要一个进程可以创建任意数目的从线程，用这些从线程执行这个进程的并发任务。

### 8.1.2 进程和线程的优先级

线程是以抢占的方式来取得 CPU 这个共享资源的。由于每个进程和线程的重要程度不同，所以让每个进程和线程以相同的机会和时间占用 CPU 是不合理的。因此，在 Windows 中允许为进程和线程赋予不同的优先级别，使其在抢占 CPU 时具有不同优先权。

Windows 用两步来确定线程的优先级，第一步先确定进程的优先权，第二步在进程所具有的优先级别基础上，再确定该进程中线程的相对优先级。

#### 8.1.2.1 进程的优先级

进程可以使用的优先级别见表 8-1。从表中可以看到，Windows 把进程的优先权分为 4 个级别：空闲(IDLE)、正常(NORMAL)、高级(HIGH)和实时(REALTIME)，其中实时级别最高，空闲级别最低。进程的优先级别告诉系统该进程与其它正在运行的进程相比较的优先级。

表 8-1 进程的优先级别

| 优先级别 | 说明 |
|---|---|
| REALTIME_PRIORITY_CLASS | 最高级 |
| HIGH_PRIORITY_CLASS | ↓ |
| ABOVE_NORMAL_PRIORTY_CLASS | ↓ |
| NORMAL_PRIORT_CLASS | ↓ |

| | |
|---|---|
| BELOW_NORMAL_PRIORT_CLASS<br>IDLE__PRIORT_CLASS | ↓<br>最低级 |

#### 8.1.2.2 线程的优先级

一个进程可由若干个线程组成，在创建线程时，需要规定线程的优先级，线程可以使用的优先级见表 8-2。线程的这些级别并不表示线程的绝对优先级，而是表示线程的绝对优先级与其所在进程的优先级之间的偏移量。

表 8-2 线程的优先级别

| 优先级别 | 偏移量 |
|---|---|
| THREAD_PRIORITY_TIME_CRITICAL | 15 |
| THREAD_PRIORITY_HIGHEST | 2 |
| THREAD_PRIORITY_ABOVE_NORMAL | 1 |
| THREAD_PRIORITY_NORMAL | 0 |
| THREAD_PRIORITY_BELOW_NORMAL | -1 |
| THREAD_PRIORITY_LOWEST | -2 |
| THREAD_PRIORITY_IDLE | -15 |

CPU 是一个由各个进程竞争使用的共享资源,通常情况下,高优先级的线程首先获得 CPU 时间，然后这个进程一直运行，直到没有数据需要处理时，Windows 才会调度其他线程。因此，不应把进程和线程的优先级别设置得过高，以使其他进程运行得非常缓慢。大多数情况下，将进程和线程的优先级别分别设为 NORMAL_PRIORT_CLASS 和 THREAD_PRIORITY_NORMAL 就比较合适。

在 Intel Fortran 中，进程和线程的优先级声明位于 IFWinTY 模块。

## 8.2 线程操作

Windows 按线程是否拥有用户界面，把线程分为两种：工作线程和用户界面线程。用户界面线程在运行时会有一个窗口界面和与其相对应的窗口函数,所以它可以通过相应消息来和用户进行交互；而工作线程没有窗口，不处理用户消息，通常用来执行一些后台任务,比如数值计算等。Intel Fortran 只支持创建工作线程。

### 8.2.1 线程的创建

线程的创建使用 CreateThread 函数，该函数返回 Integer(4)的线程句柄，当与创建的线程进行通信或关闭线程时，需要使用该线程句柄。CreateThread 函数原型为：

CreateThread (security, stack, thread_func, argument, flags, thread_id)

除了 thread_func 参数代表创建的线程所运行的例程外，其他函数均为 Integer(4)变量。各参数的描述见表 8-3。

表 8-3 创建线程函数 CreateThread 的参数

| 参数 | 描述 |
|---|---|
| security | 该参数为 IFMT 模块中声明的 SECURITY_ATTRIBUTES 类型变量。若其值为 0，则表示创建的进程有与父进程相同的安全属性 |
| stack | 该参数规定创建线程的堆栈大小。Windows 在应用程序(进程)的虚拟 |

| | |
|---|---|
| | 地址空间为创建的线程分配线程堆栈，默认情况下，Windows 将缺省的堆栈空间全部分配给第一个执行进程。若该参数为 0，则表示创建进程堆栈大小与主线程的相同。视情况需要，线程的堆栈大小可动态增加(最大为 1MB) |
| thread_func | 该参数表示创建的线程所运行的函数地址 |
| argument | 该参数(指针)代表线程运行函数 thread_func 的唯一参数，指向线程通信的数据结构 |
| flags | 该参数规定创建的线程是否立即执行。它有 0 和 CREATE_SUSPEND 两个取值:0 表示立即执行;CREATE_SUSPEND 表示知道调用 ResumeThread 函数才执行 |
| thread_id | 该参数为输出参数，用来标识所创建的线程。指代一个线程可以通过线程句柄，也可以通过线程标识符。 |

### 8.2.2 线程的终止

若要终止线程的执行，则需在线程运行的函数中调用 ExitThread 例程。该例程的原型为：

ExitThread ( [ Termination Status ] )

Termination Status 参数取值 0，表示线程正常终止。当然，也可以赋予其他的值和所代表的意义。

当被调用线程不再需要时，调用线程需要使用 CloseHandle 例程来关闭被调用线程的句柄，释放由被调用线程所占用的内存资源。有时，会出现多个线程句柄对一个线程开放的情况。例如，一个程序创建两个线程，其中的一个线程等待来自另一个线程的信息，在这种情况下，两个线程句柄同时对第一个线程开放，线程对象直到最后的线程句柄关闭才被删除。当父进程终止时，当中所有的线程句柄自动被关闭。

假如安全属性设置合理的话，TerminationThread 例程允许一个线程终止另一个线程，此时，依附于被终止线程的 DLL 得不到通知，它所占用的堆栈也不能被清理。所以，一般情况下不使用 TerminationThread 例程来终止线程。

### 8.2.3 线程的挂起和恢复

要挂起一个线程，需使用 SuspendThread 函数。该函数对同步线程并不十分有效，因为它不能控制线程挂起时的代码执行点。可是，当创建一个单独的线程用来确认用户输入时，却可以使用该函数挂起线程：一旦证实用户的输入是有效的，就终止该线程的执行；否则，就恢复该线程的执行。

当使用 CreateThread 函数创建线程，并使线程处于挂起状态(CREATE_SUSPENDED)时，该线程直到调用 ResumeThread 函数才恢复执行。这种操作方式，既有利于线程执行前的初始化工作，又便于进行一次性的线程同步——因为 ResumeThread 函数能够确保被挂起的线程在代码开始点处恢复执行。

【例 8-1】设计一 QuickWin 多线程程序(图 8-1)。在“File”菜单下插入“StartWork”和“StopWork”两个菜单项，当用户选择执行“StartWork”菜单命令时，创建并启动一个工程线程，在状态栏内输出一连串的“*”字符，以此来模拟执行长任务；当用户选择执行“StopWork”菜单命令时，终止工作线程的执行，提前结束任务。

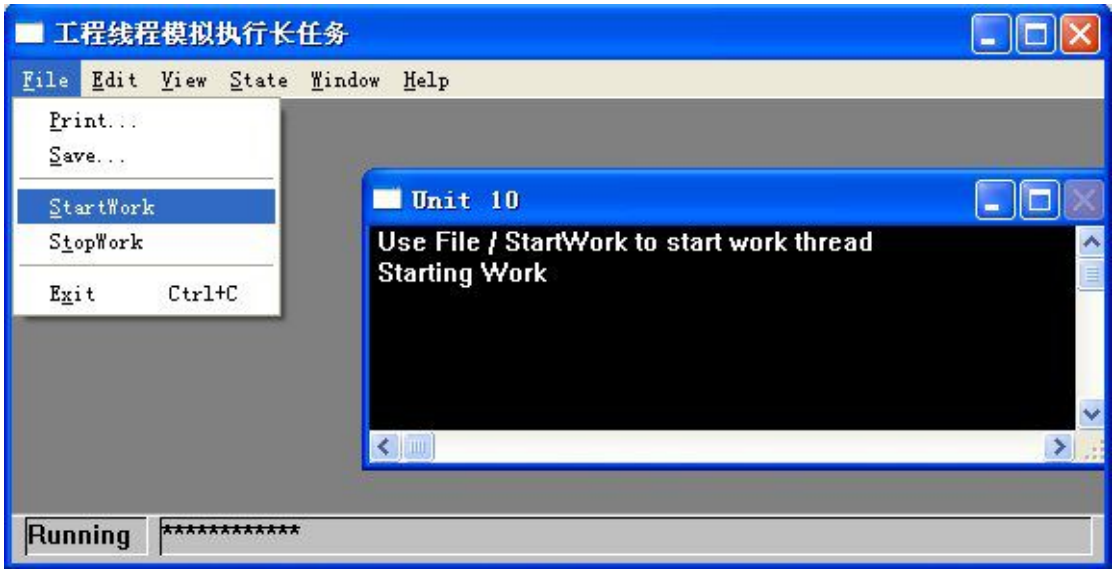



图 8-1 多线程 QuickWin 程序实例

*(1)编写 QuickWin 主程序(代表主线程)*

主程序主要执行框架窗口、子窗口及 I/O 操作，如设置框架窗口标题、插入菜单、打开子窗口、输出到子窗口等。当涉及自定义菜单、多线程等问题时，须在主程序末尾设置一个空循环。这样，系统才能正确设置自定义菜单，并对用户操作做出及时响应。

主程序代码如下：

```
Program MThread
  Use IFQWin
  Use User32
  Implicit None
  Integer(4) status
  External StartWorkProc,StopWorkProc
  status = SetWindowText( GetHwndQQ(QWIN$FRAMEWINDOW) , "工程线程模拟执行长任务"C )
  Open(10,File='User')
  !put in start and stop handlers
  status = InsertMenuQQ( 1 , 3 , $MENUSEPARATOR , ''C , NUL )
  status = InsertMenuQQ( 1 , 4 , $MENUENABLED , '&StartWork'C , StartWorkProc )
  status = InsertMenuQQ( 1 , 5 , $MENUENABLED , 'S&topWork'C , StopWorkProc )
  status = InsertMenuQQ( 1 , 6 , $MENUSEPARATOR , ''C , NUL )
  Write(10,*) 'Use File / StartWork to start work thread'
  Do While(.True.)
  End Do
End Program MThread
```

程序中，分别为“StartWork”和“StopWork”菜单注册了 StartWorkProc 和 StopWorkProc 菜单回调例程。

*(2)创建和启动工作线程*

下面先给出菜单例程 StartWorkProc 的程序代码：

```
Subroutine StartWorkProc(item_Checked)
  Use IFQWin
  Use IFMT
  Implicit None
  Logical item_Checked
  Integer*4 isRunning , stopRun
  COMMON isRunning , stopRun
  External WorkProc
  Integer(INT_PTR_KIND()) ThreadHandle
  Logical(4) retLog
  Integer(4) retInt , i , j ,val
  If ( isRunning /= 0 ) then
    i = MessageBoxQQ( 'Already Running'C , 'OK'C , MB$OK + MB$DEFBUTTON2 )
    return
  Else
    stopRun   = 0
    isRunning = 1
    ThreadHandle = CreateThread( 0 , 0 , WorkProc , Loc(val) , CREATE_SUSPENDED , j )
    retLog       = SetThreadPriority( ThreadHandle , THREAD_PRIORITY_BELOW_NORMAL )
    retInt       = ResumeThread( ThreadHandle )
  End If
End Subroutine StartWorkProc
```

程序中，采用下列语句来创建和启动工作线程：

```
ThreadHandle = CreateHand( 0 , 0 , WorkProc , Loc(val) , CREATE_SUSPENDED , j )
retLog       = SetThreadPriority( ThreadHandle , THREAD_PRIORITY_BELOW_NORMAL )
retInt       = ResumeThread( ThreadHandle )
```

CreateHand 函数第一个参数取值为 0，规定创建线程的安全属性与父进程的相同；第二个参数取值为 0，规定创建线程的堆栈大小与主线程的相同；第三个参数规定线程执行的例程为 WorkProc；第四个参数(指针)用做线程例程的实参，在线程间传递一个数据结构，在该实例中只起占用作用；第五个参数取值 CREATE_SUSPENDED，规定创建的线程直到调用 ResumeThread 函数才恢复执行；最后一个输出参数代表创建线程的标识符，在该实例中只起占位作用。

该实例先调用 CreateThread 函数创建线程并挂起，待调用 SetThreadPriority 函数设置了线程的优先级后，再调用 ResumeThread 函数恢复线程的执行。

*(3)执行线程*

工作线程一旦启动，便执行 CreateThread 函数第三个参数规定的线程例程(WorkProc)。线程执行的例程要求是函数格式，且具有一个 Integer(4)指针类型的可选参数。其函数原型为：

```
Integer(4) Function WorkProc( dumArg )
```

其中的参数 dumArg 可用来与 CreateThread 函数的第四个参数交换数据，即从调用线程向被调用线程传递数据。

该实例的线程例程 WorkProc 为：

```
Integer(4) Function WorkProc(dumArg)
  Use IFMT
  Implicit None
  Integer(4) dumArg , i , retInt
  Logical retLog
  Character(50) prog
  Integer*4 isRunning , stopRun
  COMMON isRunning , stopRun

  !Internal QuickWin routine to set status bar at screen bottom
  Interface
    Integer*4 Function SetStatusBar(msg)
      !DEC$ IF DEFINED(_X86_)
      !DEC$ ATTRIBUTES C,ALIAS : '__QWINTSetStatusBar' :: SetStatusBar
      !DEC$ ELSE
      !DEC$ ATTRIBUTES C,ALIAS : '_QWINTSetStatusBar' :: SetStatusBar
      !DEC$ ENDIF
      Integer msg  !to hold address of string
    End Function SetStatusBar
  End Interface
  Do i = 1 , 49
    prog(i:i) = ""
  End Do
  prog(50:50) = Char(0) !null terminate
  Write(10,*) 'Starting Work'
  !Show progress
  Do i = 1 , 50
    Call SleepQQ(500) !sleep a half a second
    prog(i:i) = "*"
    retInt = SetStatusBar(Loc(prog))
    If ( stopRun /= 0 ) then
      Write(10,*) "Aborting Work"
      Exit
    End If
  End Do
  isRunning = 0 !indicate completion
  stopRun = 0
  WorkProc = 0
  retInt = SetStatusbar(0) !free any string storage
  Write(10,*) 'Stopping Work'
  Call ExitThread(0)
End Function WorkProc
```

程序中，没有利用线程例程的参数来传递数据，而是利用共用区在菜单例程

StartWorkProc、StopWorkProc 和线程例程 WorkProc 间共享线程执行与停止的标志数据(isRunning 和 stopRun)。(推荐使用新的 Module 语法代替 COMMON)

在线程执行过程中，不断检测停止标志数据。如果该数据被菜单例程 StopWorkProc 设为“停止”标志，那么就中断任务的执行，并调用 ExitThread 例程退出线程。

程序中调用的 SetStatusBar 函数，实际上为 QuickWin 的内部函数 QWINSetStatusBar。此处，利用别名(alias)属性将其重命名为 SetStatusBar。

*(4)线程终止例程*

在线程例程外部——这里是菜单例程 StopWorkProc，设置线程停止标志(stopRun=1)。该标志通过共用区传递给线程例程，工作例程在执行下一步循环前予以检测，并适时退出线程。

菜单例程 StopWorkProc 的代码为：

```
Subroutine StopWorkProc(item_Checked)
  Implicit None
  Logical item_Checked
  Integer*4 isRunning , stopRun
  COMMON isRunning , stopRun
  stopRun = 1
End Subroutine StopWorkProc
```

## 8.3 线程同步

每一个线程都有自己的堆栈和 CPU 寄存器拷贝，但诸如文件、单元、静态数据、堆内存等其他资源则被进程内的所有线程所共享，使用这些共享资源的线程间必须进行某种同步、协调。

Intel Fortran 提供了临界段、互斥体、信号计数器和事件对象四种线程同步对象。这些对象要么处于有信号状态，要么处于无信号状态。有信号状态，表明等待的线程可以使用共享资源；无信号状态，则表明共享资源正在被某个进程使用。

### 8.3.1 临界段

临界段对象就像一把钥匙，哪个线程获得了它就获得了运行线程的权力，而其他线程全部被阻塞，所以它可以防止多个线程同时访问某个共享资源。比如，编辑共用区的数据。

在使用临界段同步线程前，先要调用 InitializeCriticalSection 例程初始化临界段，如下列语句所示：

```
Type(RTL_CRITICAL_SECTION) DrawLock
......
Call InitializeCriticalSection( Loc(DrawLock) )
```

在需要编辑共享资源时，调用 EnterCriticalSection 例程进入临界段；编辑完毕，调用 LeaveCriticalSection 例程离开临界段。如下列语句所示：

```
Call EnterCriticalSection( Loc(DrawLock) )
Call LeaveCriticalSection( Loc(DrawLock) )
```

EnterCriticalSection 和 LeaveCriticalSection 例程允许多次被调用，即可反复进入、

离开临界段。

### 8.3.2 互斥体

互斥体对象的使用方法与临界段对象的使用方法极为相似。不同的是，临界段对象只能在同一个进程中对线程进行同步，而互斥体对象却可以在不同的进程间进行线程同步控制。所以，互斥体对象典型地被用来限制一次只能由一个线程访问的系统资源。比如，使用一台系统打印机。

创建互斥体对象使用 CreateMutex 函数，例如：

```
hRunMutex = CreateMutex( NULL_security_attributes , .True. , NULL_CHARACTER )
```

该函数的第一个参数是规定安全属性；第二个参数指定互斥体对象初始状态是锁定(.True.)还是非锁定(.False.)；第三个参数用来指定互斥体的名称。CreateMutex 函数调用成功，返回互斥体对象句柄；假如同名称的互斥体已经存在，函数返回一个错误码，若要获悉这个错误码(ERROR_ALREADY_EXISTS)，可调用 GetLastError 函数。

要判断特定名称的互斥体对象是否存在，可使用 OpenMutex 函数。假如存在，韩淑返回互斥体对象句柄；否则，返回空指针。

当获得的互斥体对象的线程不再需要互斥体时，应该调用 ReleaseMutex 释放互斥体，使互斥体从无信号状态变为有信号状态，以便等待的线程可以获取互斥体，执行其任务。

等待获取互斥体的线程，需调用 WaitForSingleObject 或 WaitForMultipleObjects 函数。例如：

```
iRet = WaitForSingleObject( hRunMutex , INFINITE )
```

该函数的第一个参数代表要获取的互斥体对象句柄;第二个参数规定等待的时间(单位为μS)，取值 INFINITE 表示无限等待下去，直至获得互斥体对象句柄。

WaitForMultipleObjects 函数的使用方法与 WaitForSingleObject 的使用方法类似，不同的是，WaitForMultipleObjects 函数的第一个参数代表对象句柄个数；第二个参数是包含对象句柄的数组。

### 8.3.3 信号计数器

信号计数器对象的作用，是对访问某个共享资源的线程数目进行限制。换句话说，信号计数器允许多个线程访问同一资源,但同时访问该资源的线程总数不能超过信号计数器对象的最大计数值。因此，创建信号计数器对象函数 CreateSemaphore 有两个特别参数，分别用来设定计数器的初始值和最大值。例如：

```
ButtonSem= CreateSemaphore( NULL_SECURITY_ATTRIBUTES , 0 , 1 , NULL_CHARACTER )
```

这里，将计数器的初始值和最大值分别设为 0 和 1。

在调用 CreateSemaphore 函数创建信号计数器对象，进行初始化时，将计数器初值设为 0 是一种惯用处理方法。这样，可在初始化期间有效保护资源，待初始化完成后再调用 ReleaseSemaphore 函数，将计数器的值增至最大值。例如：

```
i = ReleaseSemaphore( ButtonSem , 1 , 0 )
```

ReleaseSemaphore 函数通过第二个参数将计数器的值增至规定的数目，其第三个参数指以前的计数器数目。

类似于 OpenMutex 函数，OpenSemaphore 函数返回指定名称的信号计数器对象句柄(如果存在)，该句柄可用做 WaitForSingleObject 或 WaitForMultipleObjects 函数的实参。调用这两个函数的同时，计数器的值增加 1，当计数器的值为 0 时，就不再允许其他的线程访

问共享资源了。

### 8.3.4 事件对象

当一个线程正在运行时，为了使其他正在等待的线程能够安全地启动运行，而不至于与正在运行的线程发生冲突，最简单的办法是设立一个由正在运行的线程操控的标志，用这个标志通知系统是否可以启动正在等待运行的线程。所谓事件对象就相当于一个标志，这个标志有两个状态；TRUE 和 FALSE。当标志为 TRUE(发信)时，允许系统启动正在等待的线程；而当标志为 FALSE(未发信)时，则禁止系统启动正在等待的线程。

事件对象的创建使用 CreateEvent 函数。例如：

```
hEvent = CreateEvent(NULL_SECURITY_ATTRIBUTES, &        !No security
                .True.,                         &        !Manual reset
                .False., &    !Initially Event set to non-signaled
                ""C)                                     !No name
```

该函数的第三个参数规定事件对象的初始状态是有信号还是无信号,即是发信还是未发信；第二个参数规定事件对象是手工事件对象还是自动事件对象。

手工事件对象与自动事件对象的差别是：当一个等待线程被释放时，自动事件对象被系统自动设置为“有信号”状态，待恢复一个被挂起的线程后，自动事件对象又被系统自动设为“无信号”状态；而手工事件对象一旦由函数 CreateEvent 或 SetEvent 设为“有信号”状态，就一直保持这种状态，除非又使用 PulseEvent 或 ResetEvent 函数把它重新设为“无信号”状态。所以，手工事件对象可以用来恢复多个处在等待状态线程的运行。

OpenEvent 函数返回事件对象句柄，该句柄可用于其他函数的调用。

## 8.4 线程局部存储(TLS)

线程局部存储(TLS)允许存储每线程数据，即允许为进程内的每个线程动态分配内存、内存线程特定的数据。

使用 TLS 方法，首先需要调用 TlsAlloc 函数为存储数据分配一个索引(指针)。例如：

```
TlsIndex = TlsAlloc()           !create TLS index
```

然后，调用 TlsSetValue 函数存储数据到指定索引。例如：

```
lret = TlsSetValue( TlsIndex , pString )        !Set TLS value
```

要访问制定索引存储的数据，则使用 TlsGetValue 函数。例如：

```
tlsValue = TlsGetValue( TlsIndex )              !Get TLS value
```

当所有线程都不再需要动态存储时,调用 TlsFree 函数释放 TLS 索引占用的内存。例如：

```
lret = TlsFree( TlsIndex )                      !free TLS index
```

## 8.5 进程操作

假如程序需要私有地址空间、访问私有资源而不与其他线程发生冲突，那么可以选择创建多进程应用程序。多个进程共享互斥体、信号计数器及事件对象，但不能共享临界段对象。通常，在执行多任务情况下，创建一个进程的多个线程较之创建多个进程更为高效。这是因为：

(1)系统创建多线程快于创建多进程。在编译时，线程代码已被映射到进程的地址空间；而进程代码则需要在运行时动态加载。

(2)一个系统内的所有线程共享同一个地址空间，可以访问进程的全局变量，这可以有效地简化线程间的通信工作。

(3)一个进程内的所有线程都可以使用诸如文件、管道这样的共享资源。

CreateProcess 函数用来创建新的进程，该函数的原型为：

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName,     //指向可执行模块名的指针
    LPTSTR lpCommandLine,          //指向命令行字符串的指针
    LPSECURITY_ATTRIBUTES lpProcessAttributes,   //被创建进程的安全属性
    LPSECURITY_ATTRIBUTES lpThreadAttributes,    //被创建进程的主线程安全属性
    BOOL bInheritHandles,          //是否继承调用进程的句柄
    DWORD dwCreationFlags,         //创建标志
    LPVOID lpEnvironment,          //指向新环境的指针
    LPCTSTR lpCurrentDirectory,    //指向当前目录的指针
    LPSTARTUPINFO lpStartupInfo,   //指向结构体STARTUPINFO的指针
    LPPROCESS_INFORMATION lpProcessInformation   //指向结构体PROCESS_INFORMATION的指针
);
```

GetCurrentProcess 函数用来获取当前正在运行的进程信息；GetCurrentProcessID 函数用来返回当前进程标识符(进程 ID)，该标识符可用来与其他进程进行通信；GetExitCodeProcess 函数则用来返回指定进程的终止状态，如果指定进程没有终止，那么就返回 STILL_ACTIVE 状态码，否则返回具体的终止状态码；OpenProcess 函数则用来打开进程句柄，访问指定的进程，并允许规定进程句柄的访问权限和继承性。

下列任何一种情况发生，都会终止进程的运行：

(1)进程的任何线程调用了 ExitProcess 函数；

(2)进程的主线程退出；

(3)进程的最后一个线程终止；

(4)TerminateProcess 函数被调用。

通常，调用 ExitProcess 函数来终止运行的进程。在这种情况下，所有依附的 DLL 都能得到通知，确保进程的所有线程都能正常终止；而调用 TerminateProcess 函数来终止进程，所有依附的 DLL 得不到通知，进程的线程所占用的资源也不能及时被释放。

【例 8-2】设计一个基于模式对话框的 Windows 多线程程序(图 8-2)。对话框上设有“创建进程...”和“终止进程”命令按钮。当用户点击“创建进程...”按钮时，会弹出系统打开文件对话框，让用户选择要运行的应用程序实例(EXE)；当用户点击“终止进程”按钮时，选定的应用程序被撤销，其代表的进程被终止。

(1)利用 Intel Fortran 程序向导创建一个基于对话框的 Windows 工程

对缺省的对话框模版按图 8-2 所示重新进行设计：在两个命令按钮之外，增加两个列表框控件。

调整后的全局模块如下所示：

```
module procinc
  use dfwin
  integer, parameter, public :: DID_CREATE     = 16#0065
  integer, parameter, public :: DID_TERMINATE  = 16#0066
```

```
  integer, parameter, public :: DID_LISTBOX     = 16#0067
  integer, parameter, public :: DID_HEADER      = 16#0068
  integer, parameter, public :: MAXCHARS        = MAX_PATH
  integer, parameter, public :: SIZEOPENFILENAME       = 76
  integer, parameter, public :: SIZESTARTUPINFO        = 68
  integer ghInstance
end module procinc
```




图 8-2 多线程 Windows 程序实例

*(2)编辑 Windows 主函数(WinMain)*

调用 Dialog 函数创建模式对话框，并设置模式对话框的窗口函数。

主函数代码如下：

```
Integer Function WinMain( hInstance, hPrevInstance, lpszCmdLine, nCmdShow )
  !DEC$ IF DEFINED(_X86_)
  !DEC$ ATTRIBUTES STDCALL, ALIAS : '_WinMain@16' :: WinMain
  !DEC$ ELSE
  !DEC$ ATTRIBUTES STDCALL, ALIAS : 'WinMain' :: WinMain
  !DEC$ ENDIF
  use IFWin
  use procinc
  integer hInstance, hPrevInstance, lpszCmdLine, nCmdShow
  interface
    integer(4) function DlgProc( hwnd, mesg, wParam, lParam )
      !DEC$ IF DEFINED(_X86_)
      !DEC$ ATTRIBUTES STDCALL, ALIAS : '_DlgProc@16' :: DlgProc
      !DEC$ ELSE
      !DEC$ ATTRIBUTES STDCALL, ALIAS : 'DlgProc' :: DlgProc
      !DEC$ ENDIF
      integer  hwnd
      integer mesg
      integer wParam
      integer lParam
```

```
    end function DlgProc
  end interface
  integer ret
  character*100  lpszDlgName
  hPrevInstance    = hPrevInstance
  lpszCmdLine      = lpszCmdLine
  nCmdShow         = nCmdShow
  ghInstance = hInstance
  lpszDlgName = "processDlg"C
  ret = DialogBoxParam(ghInstance,LOC(lpszDlgName),NULL, LOC(DlgProc), 0)
  WinMain = ret
  return
End Function WinMain
```

在 Windows 主函数中，通常需建立模式对话框的窗口函数接口，如上列代码所示。

创建模式对话框的 DialogBox 函数原型为：

```
int DialogBox(
    HINSTANCE hInstance,       //应用程序实例句柄
    LPCTSTR lpTemplate,        //对话框模版标识符
    HWND hWndParent,           //父窗口句柄
    DLGPROC lpDialogFunc       //指向对话框窗口函数的指针
);
```

程序中，模式对话框的父窗口为空，窗口函数规定为 DlgProc。

*(3)编写模式对话框的窗口函数并对特定的消息进行处理*

该实例需要处理的消息包括 WM_INITDIALOG、WM_SYSCOMMAND 和 WM_COMMAND 三类：在 WM_INITDIALOG 消息处理代码段中，调用 SendDlgItemMessage 函数向对话框的列表框控件发送消息，初始化列表框控件；在 WM_SYSCOMMAND 消息处理代码段中，调用 EndDialog 函数终止模式对话框运行，以响应用户点击系统关闭按钮发出的 SC_CLOSE 消息；在 WM_COMMAND 消息代码处理段中，调用相应的回调例程来分别创建和终止进程，以响应用户点击“创建进程...”和“终止进程”发出的命令消息。

该实例的窗口函数代码如下：

```
Integer*4 Function DlgProc( hDlg, message, wParam, lParam )
  !DEC$ IF DEFINED(_X86_)
  !DEC$ ATTRIBUTES STDCALL, ALIAS : '_DlgProc@16' :: DlgProc
  !DEC$ ELSE
  !DEC$ ATTRIBUTES STDCALL, ALIAS : 'DlgProc' :: DlgProc
  !DEC$ ENDIF
  use procinc
  integer     hDlg        !window handle of the dialog box
  integer     message     !type of message
  integer     wParam      !message-specific information
  integer     lParam      !message-specific information
```

```
interface
  subroutine doCreate (hDlg)
    !DEC$ ATTRIBUTES VALUE :: hDlg
    integer*4 hDlg
  end subroutine
  subroutine doTerminate (hDlg)
    !DEC$ ATTRIBUTES VALUE :: hDlg
    integer*4 hDlg
  end subroutine
end interface
integer*4   tabs(4)
integer*4   ret
lParam = lParam
select case (message)
  case (WM_INITDIALOG)    ! message: initialize dialog box
    tabs(1) = 10 * 4             !(WORD)(sizeof("00000000 ") *4);
    tabs(2) = tabs(1) + 10 * 4   !(WORD)(sizeof("00000000 ") *4);
    tabs(3) = tabs(2) + 13 * 4   !(WORD)(sizeof("dwProcessID ") *4);
    tabs(4) = tabs(3) + 12 * 4   !(WORD)(sizeof("dwThreadID ") *4);
    ret = SendDlgItemMessage (hDlg, DID_LISTBOX,               &
                              LB_SETTABSTOPS, 4, LOC(tabs(1)))
    ret = SendDlgItemMessage (hDlg, DID_HEADER,                &
                              LB_SETTABSTOPS, 4, LOC(tabs(1)))
    ret = SendDlgItemMessage (hDlg, DID_HEADER, LB_ADDSTRING,0,&
              LOC("hProcess \thThread \tdwProcessID \tdwThreadID \tImage file"C))
    DlgProc = 1
    return
  case (WM_SYSCOMMAND)
    if (wParam == SC_CLOSE) then
       ret = EndDialog (hDlg, TRUE)
       DlgProc = 1
       return
    else
       DlgProc = 0
    end if
  case (WM_COMMAND)                      ! message: received a command
    ! if the list box sends back messages, return.
    if (LoWord(wParam) == DID_LISTBOX) then
      DlgProc = 1
      return
    end if
    select case ( INT4(LoWord(wParam)))
      case ( DID_CREATE)
```

```
            call doCreate(hDlg)
        case ( DID_TERMINATE)
            call doTerminate(hDlg)
      end select
      DlgProc = 1
      return
  end select  ! message
  DlgProc = 0 ! Didn't process the message
end function DlgProc
```

其中，向对话框控件发送消息的 SendDlgItemMessage 函数原型为：

```
LONG SendDlgItemMessage(
    HWND hDlg,        //包含该控件的对话框句柄
    int nIDDlgItem, //控件标识符(ID)
    UINT Msg,             //发送的控件消息
    WPARAM wParam,       //消息附加参数
    LPARAM lParam        //消息附加参数
);
```

程序中，向列表框控件发送了 LB_SETTABSTOPS 和 LB_ADDSTRING 消息，分别设置列表项的 Tab 间隔和显示的字符串。

至于“创建进程...”和“终止进程”命令按钮的回调例程(doCreate 和 doTerminate)，则需要显示调用(call)。这里的回调例程，相当于 Visual Basic 程序为控件编写的事件过程。

*(4)创建进程*

在上面的窗口函数中，“创建窗口...”按钮的回调例程规定为 doCreate。在 doCreate 回调例程中，主要是设置 CreateProcess 函数的参数，然后调用 CreateProcess 函数创建新的进程。

doCreate 回调例程代码为：

```
subroutine doCreate(hDlg)
  !DEC$ ATTRIBUTES VALUE :: hDlg
  use procinc
  integer*4 hDlg
  type (T_OPENFILENAME)       of
  character*256               buffer
  character*256               buffer1
  character*56                buffer2
  character*56                buffer3
  character*56                buffer4
  character*256               szFileTitle
  type (T_STARTUPINFO)            sui
  type (T_PROCESS_INFORMATION)    pi
  integer*4          ret
  logical(4)         bret
  szFileTitle = ""C
```

```
buffer = szFileTitle
!set up the OPENFILE structure,
!then use the appropriate common dialog
call ZeroMemory(LOC(buffer),256)
iret = lstrcpy(buffer,"*.exe"C)
of%lStructSize       = SIZEOPENFILENAME    !76
of%hwndOwner         = NULL
of%hInstance         = ghInstance
of%lpstrFilter       = LOC("Executables\0,*.EXE\0\0"C)
of%lpstrCustomFilter = NULL
of%nMaxCustFilter    = 0
of%nFilterIndex      = 0
of%lpstrFile         = LOC(buffer)
of%nMaxFile          = MAXCHARS
of%lpstrFileTitle    = NULL
of%nMaxFileTitle     = 0
of%lpstrInitialDir   = NULL
of%lpstrTitle        = NULL
of%Flags             = OFN_HIDEREADONLY
of%nFileOffset       = 0
of%nFileExtension    = 0
of%lpstrDefExt       = NULL
of%lCustData         = 0
of%lpfnHook          = NULL
of%lpTemplateName    = NULL

if ( GetOpenFileName(of) .EQV. .FALSE. ) then
  return
endif

!set up the STARTUPINFO structure,
!then call CreateProcess to try and start the new exe.
sui%cb               = SIZESTARTUPINFO  ! 68 ( sizeof(StartUpInfo)
sui%lpReserved       = 0
sui%lpDesktop        = NULL
sui%lpTitle          = NULL
sui%dwX              = 0
sui%dwY              = 0
sui%dwXSize          = 0
sui%dwYSize          = 0
sui%dwXCountChars    = 0
sui%dwYCountChars    = 0
sui%dwFillAttribute  = 0
sui%dwFlags          = 0
```

```
  sui%wShowWindow      = 0
  sui%cbReserved2      = 0
  sui%lpReserved2      = 0
  iret = ConvertFToCString (buffer, of%lpstrFile)

  bret = CreateProcess (                          &
                        buffer,                   &
                        NULL_CHARACTER,           &
                        NULL_SECURITY_ATTRIBUTES, &
                        NULL_SECURITY_ATTRIBUTES, &
                        .FALSE.,                  &
                        DETACHED_PROCESS,         &
                        NULL,                     &
                        NULL_CHARACTER,           &
                        sui,                      &
                        pi)


  if (bret .eqv. .TRUE.) then
    write(buffer1,10) pi%hProcess
    10       format(Z8)
    buffer1(9:9) = char(0)
    ret = lstrcat(buffer1, "\t"C)
    write(buffer2,20) pi%hThread
    20       format(Z8)
    buffer2(9:9) = char(0)
    ret = lstrcat(buffer2, "\t"C)
    ret = lstrcat(buffer1, buffer2)
    ret = pi%dwProcessId
    write(buffer3,30) ret
    30       format(Z8)
    buffer3(9:9) = char(0)
    ret = lstrcat(buffer3, "\t"C)
    ret = lstrcat(buffer1, buffer3)
    write(buffer4,40) pi%dwThreadId
    40       format(Z8)
    buffer4(9:9) = char(0)
    ret = lstrcat(buffer4, "\t"C)
    ret = lstrcat(buffer1, buffer4)
    ret = lstrcat(buffer1, buffer)
    ret = SendDlgItemMessage (hDlg, DID_LISTBOX, LB_ADDSTRING, 0, LOC(buffer1))
  else
    !report failure to the user.
    ret = GetLastError ()
```

```
    write(buffer,100) ret
100 format ('0x',Z5)
    ret = MessageBox( hDlg, buffer, "GetLastError..."C,IOR(MB_ICONSTOP,MB_OK ))
  end if
end Subroutine doCreate
```

CreateProcess 函数的第一个参数为新进程运行的执行文件名(EXE)。程序中，执行文件名通过调用 GetOpenFile 函数、展示系统通用打开文件对话框，从用户选择的文件名中获取。用户选择的文件名存放在 GetOpenFile 函数参数为 T_OPENMFILENAME 派生类型)的 lpstrFile 成员中。

T_OPENFILENAME 派生类型对应的结构体 OPENFILENAME 的定义为：

```
    typedef struct tagOFN { // ofn
        DWORD        lStructSize;    //结构体长度(字节)
        HWND         hwndOwner;      //拥有对话框的窗口句柄
        HINSTANCE    hInstance;      //包含对话框的程序实例句柄
        LPCTSTR      lpstrFilter;    //过滤器字符串指针
        LPTSTR       lpstrCustomFilter;    //指向运行时用户规定的过滤器
        DWORD        nMaxCustFilter; //lpstrCustomFilter的字符串长度(字节)
        DWORD        nFilterIndex;   //当前选取的过滤器索引
        LPTSTR       lpstrFile;      //初始化用于存放文件名的缓冲区
        DWORD        nMaxFile;       //lpstrFile缓冲区长度
        LPTSTR       lpstrFileTitle; //用于存放文件名和扩展名的缓冲区，该参数可以为NULL
        DWORD        nMaxFileTitle;  //lpstrFileTilter缓冲区长度
        LPCTSTR      lpstrInitialDir;//初始文件路径，如果该参数为NULL，初始路径为当前路径
        LPCTSTR      lpstrTitle;     //文件对话框标题，如果该参数为NULL，则使用缺省标题
        DWORD        Flags;          //初始化对话框的位标志
        WORD         nFileOffset;    //lpstrFile参数中文件名偏移量(从路径首字符算起)
        WORD         nFileExtension; //lpstrFile参数中扩展名偏移量(从路径首字符算起)
        LPCTSTR      lpstrDefExt;    //缺省文件扩展名
        DWORD        lCustData;      //传递给挂钩例程(lpfnHook)的数据
        LPOFNHOOKPROC lpfnHook;      //指向挂钩例程
        LPCTSTR      lpTemplateName; //指向由hInstance成员标识的对话框模版资源
    } OPENFILENAME;
```

CreateProcess 函数的最后两个参数为 T_STARTUPINFO 派生类型，其对应的结构体 STARTUPINFO 的定义为：

```
typedef struct _STARTUPINFO { // si
    DWORD   cb;              //结构体的大小(字节)
    LPTSTR  lpReserved;      //保留字，设为NULL
    LPTSTR  lpDesktop;       //运行进程的计算机名
    LPTSTR  lpTitle;         //进程创建窗口(指控制台程序，下同)的标题
    DWORD   dwX;             //进程创建窗口的X屏幕坐标(像素)
    DWORD   dwY;             //进程创建窗口的Y屏幕坐标(像素)
    DWORD   dwXSize;         //进程创建窗口的宽度(像素)
    DWORD   dwYSize;         //进程创建窗口的高度(像素)
    DWORD   dwXCountChars;   //进程创建窗口时的屏幕缓冲区字符列数
```

```
    DWORD   dwYCountChars;    //进程创建窗口时的屏幕缓冲区字符行数
    DWORD   dwFillAttribute;  //进程创建窗口时展示的文本及背景颜色
    DWORD   dwFlags;          //位标志，决定进程创建窗口时是否使用STARTUPINFO的某些成员
    WORD    wShowWindow;      //进程创建窗口的展示方式
    WORD    cbReserved2;      //保留字，设为
    LPBYTE  lpReserved2;      //保留字，设为NULL
    HANDLE  hStdInput;        //用做进程的标准输入句柄
    HANDLE  hStdOutput;       //用做进程的标准输出句柄
    HANDLE  hStdError;        //用做进程的标准错误句柄
} STARTUPINFO, *LPSTARTUPINFO;
```

CreateProcess 函数的最后一个参数为 T_PROCESS_INFORMATION 派生类型，其对应的结构体 PROCESS_INFORMATION 的定义为：

```
typedef struct _PROCESS_INFORMATION { // pi
    HANDLE hProcess;                  //被创建进程的句柄
    HANDLE hThread;                   //进程中的主线程句柄
    DWORD dwProcessId;                //被创建进程的标识符
    DWORD dwThreadId;                 //进程中的主线程标识符
} PROCESS_INFORMATION;
```

一旦进程创建成功，就可以从 CreatProcess 函数最后一个参数中获取新创建进程及其主线程的句柄和标识符。然后，调用 SendDlgItemMessage 函数向列表框发送 LB_ADDSTRING 消息，来设置列表项的字符串。

*(5)终止进程*

在前面的窗口函数中，“终止进程”命令按钮的回调例程规定为 doTerminate。在 doTerminate 回调例程中，先获取要终止的进程句柄，然后调用函数 TerminateProcess 来终止进程。

doTerminate 回调例程代码为：

```
subroutine doTerminate(hDlg)
  !DEC$ ATTRIBUTES VALUE :: hDlg
  use procinc
  integer*4 hDlg
  character*256   buffer5
  character*256   buffer6
  integer*4       sel
  integer*4       ret
  logical(4)      bret
  integer*4       hProcess
  !determine which item is selected in the list box, and get the text
  sel= SendDlgItemMessage (hDlg, DID_LISTBOX, LB_GETCURSEL, 0, 0)
  if (sel == LB_ERR) then
    ret = MessageBox (hDlg, "No listbox item is selected."C, "Incorrect use"C, IOR(MB_ICONSTOP,
MB_OK))
    return
  end if
```

```
  ret = SendDlgItemMessage (hDlg, DID_LISTBOX, LB_GETTEXT, sel, LOC(buffer5))
  !pick the process handle out of the string.
  read(buffer5, 50) hProcess
50  format(Z8)
  bret = TerminateProcess (hProcess, 0)
  if (bret .eqv. .TRUE.) then
    ret = SendDlgItemMessage (hDlg, DID_LISTBOX, LB_DELETESTRING, sel, 0)
  else
    ret = GetLastError ()
    write(buffer6,200) ret
200 format ('0x',Z5)
    ret = MessageBox( hDlg, buffer6, "GetLastError..."C, IOR(MB_ICONSTOP,MB_OK))
  endif
end subroutine doTerminate
```

程序中，为了获取要终止的进程句柄，先向列表框控件发送 LB_GETCURSEL 消息，以获取用户选择的列表项；再向列表框控件发送 LB_GETTEXT 消息，来获取用户选择的列表项文本，并从中分离出进程句柄。

进程终止后，向列表框控件发送 LB_DELETESTRING 消息，来清除终止进程的列表项文本。

终止进程函数 TerminateProcess 的原型为：

```
BOOL TerminateProcess(
    HANDLE hProcess,          //要终止的进程句柄
    UINT uExitCode            //进程的退出状态码
);
```

函数调用成功，返回非 0；否则，返回 0。

## 8.6 多线程操作的相关例程

表 8-4 列出了 IFMT 模块中的多线程操作相关例程(不包含位于 IFWin 模块中的相关例程)，这些例程的详细信息需查阅 SDK 文档。

表 8-4 多线程操作的相关例程

| 线程操作例程 | 说明 | 线程操作例程 | 说明 |
|---|---|---|---|
| CloseHandle | 关闭一个打开的对象句柄 | PulseEvent | 将事件对象设为无信号状态 |
| CreateEvent | 创建一个事件对象 | ReleaseMutex | 交出互斥体对象的拥有权 |
| CreateMutex | 创建一个互斥体对象 | ReleaseSemaphore | 将信号计数器对象的计数器增加一个指定值 |
| CreateProcess | 创建一个进程及其主线程 | ResetEvent | 将事件对象状态设为无信号 |
| CreateSemaphore | 创建一个信号计数器对象 | ResumeThread | 将线程挂起数递减 1，当减至 0 时，挂起线程被恢复执行 |
| CreateThread | 创建一个执行在调用进程地址空间内的线程 | SetEvent | 将事件对象状态设为有信号 |
| DeleteCriticalSection | 删除一个临界段对象 | SetLastError | 设置调用线程的最后错误码 |
| DuplicateHandle | 复制一个对象句柄 | SetPriorityClass | 设置指定进程的优先级 |

| | | | |
|---|---|---|---|
| EnterCriticalSection | 等待拥有临界段对象 | SetThreadPriority | 设置线程的优先级相对值 |
| ExitProcess | 结束进程及其所有线程 | SuspendThread | 挂起指定的线程 |
| ExitThread | 结束一个线程 | TerminateProcess | 终止指定的进程及其线程 |
| GetCurrentProcess | 获取当前进程的伪句柄 | TerminateThread | 终止一个线程 |
| GetCurrentProcessID | 获取当前进程的标识符 | TlsAlloc | 分配线程局部存储索引 |
| GetCurrentThread | 获取当前线程的伪句柄 | TlsFree | 释放线程局部存储索引 |
| GetCurrentThreadID | 获取当前线程的标识符 | TlsGetValue | 获取与线程局部存储索引相关联的数据 |
| GetExitCodeProcess | 获取指定进程的终止状态 | TlsSetValue | 设置与线程局部存储索引相关联的数据 |
| GetExitCodeThread | 获取指定线程的终止状态 | WaitForMultipleObjects | 当指定对象有一个处于有信号状态，或者过了规定的超时，等待的线程恢复执行 |
| GetLastError | 获取当前线程最近一次出现错误的码值 | WaitForSingleObject | 当指定对象处于有信号状态或过了规定的超时，等待的线程恢复执行 |
| GetPriorityClass | 获取指定进程的优先级 | OpenEvent | 获取现存的事件对象句柄 |
| GetThreadPriority | 获取线程的优先级相对值 | OpenMutex | 获取现存的互斥体对象句柄 |
| InitializeCriticalSelction | 初始化一个临界段对象 | OpenProcess | 获取现存的进程对象句柄 |
| LeaveCriticalSection | 交出临界段对象的拥有权 | OpenSemaphore | 获取信号计数器对象句柄 |

# 小　结

■进程是一个正在运行的应用程序实例，拥有应用程序的所有资源，它由一个或多个线程组成。线程是进程中一个独立的执行路径，Windows 按照一定的规则给线程分配 CPU 时间。

■Intel Fortran 只支持创建 Windows 中的工作线程。工作线程没有窗口，不处理用户消息，通常用来执行一些后台任务。

■使用 CreateThread 函数创建并启动一个线程，该函数的第三个参数规定线程执行的例程；ResumeThread 函数恢复并执行创建时挂起的线程；ExitThread 子程序终止线程的执行。

■线程同步对象包括临界段、互斥体、信号计数器和事件对象四种：如果线程必须等待某个事件的发生才能访问资源，则使用事件对象；如果同一时刻允许多个线程访问资源，则使用信号计数器；要防止多个线程同时访问资源，使用临界段和互斥体。临界段只能用于同一进程内线程间的同步控制，而互斥体可用于进程间线程的同步控制。

■线程局部存储(TLS),运行为进程内的线程动态分配内存，存储线程特定的数据。

■使用 CreateProcess 函数创建并启动一个进程,该函数的第一个参数规定进程运行的执行文件；TerminateProcess 函数终止指定的进程。

# 第 9 章　创建和使用动态链接库

动态链接库(Dynamic-Link Library:DLL)作为 Windows 特有的可执行模块，包含其它执行模块(DLL 文件或 EXE 文件)可调用的例程、数据及资源。DLL 文件平时驻留在磁盘上，只有当正在运行的应用程序需要调用 DLL 模块的情况下，系统才会将其装载到内存空间中。这种方式不但减少了应用程序 EXE 文件的大小和对内存空间的需要，而且 DLL 模块可以同时被多个(即便由不同语言编写的)应用程序所共享。

**本章主要内容：**

- 动态链接库的概念
- 动态链接库的创建
- 动态链接库的使用

## 9.1 动态链接库概述

### 9.1.1 动态链接库的基本概念

目前，比较大的应用程序都是由若干个模块组成的，这些模块分别实现了程序中相对独立的功能，它们通过彼此之间的相互协作共同完成整个软件系统的任务。在这些程序模块中，必然有一些功能较为通用、在设计其他软件系统时仍希望能被使用的模块。这样，人们就希望能有一种方法，使这些具有通用功能的模块代码可以为不同的应用程序所共享。

以前，曾采用静态库(Lib)实现代码的共享。也就是把一些具有通用功能的模块编写成例程，然后在编译时将例程代码静态链接到用户的应用程序上，从而使之成为应用程序的一部分。但是，这种做法存在一些缺陷：一个缺陷是增加了应用程序的代码量，因而占用了更多的磁盘空间，同时在程序运行时也会占用较大的内存空间，从而造成了系统资源的浪费；另一个缺陷是，在编写大的 EXE 程序时，在每次修改重建时都必须调整、编译所有源代码，增加了编译过程的复杂性。

为了解决上述问题，Windows 允许把独立的程序模块创建成 DLL 文件。这种文件与静态链接库相比较有下列为人们所欢迎的优点：

(1)节省内存空间。多个应用程序(进程)可以同时使用一个 DLL，在内存中共享该 DLL 的一个副本。相反，对于每个用静态链接库生成的应用程序，Windows 都必须在内存中加载库代码的一个副本。

(2)节省磁盘空间。多个应用程序可在磁盘上共享 DLL 的一个副本。相反，每个用静态链接库生成的应用程序均具有作为单独的副本链接到其可执行文件中的库代码。

(3)升级 DLL 更为容易。DLL 中的函数更改时，只要函数的参数和返回值没有改变，就不需要重新编译或链接使用它们的应用程序。相反，静态链接的对象代码要求在函数更改时重新编译应用程序。

(4)支持多语言编程。只要程序遵循相同的调用约定，用不同语言编写的程序都可以调用相同的 DLL 函数。

(5)创建不同语言版本的程序更为容易。通过将不同语言版本的字符串放置到 DLL 中，使不同语言的版本加载不同的字符串资源，从而使创建国际版本的应用程序变得更为容易。

(6)自动卸载。当没有应用程序需要调用 DLL 时，DLL 会自动卸载、释放所占用的内存

资源。

## 9.1.2 动态链接库的组成

通常，一个 DLL 由例程(函数和子程序的统称)、数据和资源组成；当需要对 DLL 进行初始化和清理工作时，还应包含一个动态链接库入口函数(DllMain)。

### 9.1.2.1 动态链接库的入口函数

我们知道，应用程序是通过 Windows 系统来调用动态链接库的，如果存在 DLL 的入口函数，那么系统在调用 DLL 时会首先调用其入口函数。DLL 入口函数的主要作用是在系统调用该 DLL 时，为 DLL 进行一些初始化工作。例如，当创建一个新的线程时，可以进行创建线程局部存储的初始化工作；当线程终止时，可以进行清理线程局部存储的善后工作。

DLL 入口函数形如：

```
BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD  ul_reason_for_call,
                      LPVOID lpReserved)
{
  //编写DLL的初始化代码
  return TRUE;
}
```

其中，第一个参数 hModule 是指向 DLL 本身的句柄；第二个参数 ul_reason_for_call 常用来指明 DLL 被调用的原因；第三个参数 lpReserved 是系统保留参数。

由于 DLL 的初始化和清理工作常常出现在表 9-1 所列的 4 种情况下，因此 Windows 定义了与这 4 种情况相对应的常数。

表 9-1 ul_reason_for_call 参数的取值

| 常数 | 说明 | 常数 | 说明 |
|---|---|---|---|
| DLL_PROCESS_ATTACH | 进程被调用 | DLL_THREAD_DETACH | 线程被终止 |
| DLL_THREAD_ATTACH | 线程被调用 | DLL_PROCESS_DETACH | 进程被终止 |

这样，便可以根据调用 DLL 的不同原因，书写不同的初始化代码：

```
BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD  ul_reason_for_call,
                      LPVOID lpReserved)
{
  switch(ul_reason_for_call)
  {
  case DLL_PROCESS_ATTACH:
    ...
  case DLL_THREAD_ATTACH:
    ...
  case DLL_THREAD_DETACH:
    ...
  case DLL_PROCESS_DETACH:
```

```
    ...
  }
  return TRUE:
}
```

#### 9.1.2.2 导出实体和内部实体

我们将 DLL 中的例程和数据统称为实体(Entity)。DLL 实体分为导出实体(Export Entity)和内部实体(Internal Entity)：如果实体是外部应用程序可以调用的，那么这种实体就称为导出实体；相反，如果实体只是在 DLL 内部调用，那么这种实体就称为内部实体。导出实体在客户端称为导入实体。

在 Intel Fortran 中，可导出的数据可以是共用区中的数据，也可以是模块中的数据；可导出的例程包括外部例程和模块例程。

值得注意的是，Intel Fortran QuickWin 应用程序不能使用 DLL 中的例程，也不能将 QuickWin 程序建造成一个 DLL 文件。

#### 9.1.2.3 DLL 和应用程序之间的差异

尽管 DLL 和应用程序都是可执行的程序模块，但它们之间有若干不同之处。对于最终用户来说，最明显的差异在于 DLL 不是可直接执行的程序。从系统角度讲，应用程序和 DLL 之间存在两个基本差异：

(1)应用程序可有多个同时在系统上运行的实例，而 DLL 只能有一个实例；

(2)应用程序可以拥有像堆栈、共用内存、文件句柄、消息队列这样的事物，而 DLL 没有这样的事物。

## 9.2 动态链接库的创建

### 9.2.1 导出声明

DLL 文件结构类似于 EXE 文件，但有一个重要的不同：DLL 文件包含一个输出表。输出表列出了 DLL 向外部程序导出的例程和数据(导出实体)，只有输出表列出的例程和数据才能由外部程序导入并访问；输出表没有列出的例程和数据(内部实体)只能在 DLL 内部使用。

利用 Visual C++自动的 DUMPBIN 工具，使用/EXPORTS 选项，可以对 DLL 的输出表进行察看。DLL 可以使用模块定义文件和 DLLEXPORT 编译器指令两种方法来导出实体。

#### 9.2.1.1 模块定义文件

编写一个模块定义文件(.DEF)，列出要导出的例程和数据，当创建 DLL 时将该文件插入 DLL 工程中。当通过序号而不是名称导出 DLL 实体时选择这种方法，但该方法要求按模块定义语法规则来组织、编写文本文件。

#### 9.2.1.2 DLLEXPORT 编译器指令

在 Visual C++中，需在导出的 DLL 实体前添加“_ _declspec(dllexport)”关键字；在 Intel Fortran 中，则使用编译器指令声明导出的 DLL 实体具有 DLLEXPORT 属性。该方法简洁明快，本书的 DLL 实例程序一律使用这种方法。

使用 DLL 的外部程序(DLL 或 EXE 文件)在有些情况下需要导入 DLL 实体。导入 DLL 实体

只能采取编译器指令方法：Visual C++需在导入的 DLL 实体前添加“__declspec (dllimport)”关键字;Intel Fortran 则使用编译器指令声明导入的 DLL 实体具有 DLLIMPORT 属性。

### 9.2.2 共享 DLL 中的例程

Fortran 90/95 存在 4 种程序单元：主程序、外部例程、模块和数据块。除主程序单元外、其他 3 种程序单元均可置于 DLL 中，被编译成二进制可执行代码。

这样，Fortran DLL 可导出的例程除了外部例程还包含模块例程。

#### 9.2.2.1 外部例程

【例 9-1】创建一个 Intel Fortran DLL，在其中插入一个实现 $c = ( a^2 + b^2 )^{1/2}$ 算法的函数，a,b,c 均为实数。

利用 Intel Fortran 应用程序向导，创建一个 Dynamic-Link Library 类型的工程(EProc_DLL)。在 DLL 工程中插入下列函数代码：

```
Function GetRSqrt(a,b) result(r)
  Implicit None
  !DEC$ ATTRIBUTES DLLEXPORT::GetRSqrt
  Real a,b,r
  r = SQRT( a*a + b*b )
End Function GetRSqrt
```

源文件缺省情况下被保存为自由格式的.F90 文件，DLLEXPORT 属性声明的前缀为“!”。如果将源文件保存为固定格式的.for 文件，那么 DLLEXPORT 属性声明的前缀须改为“C”或“*”，且应位于第 1 列至第 5 列的语句标号区

在.NET 环境中，执行“生成”菜单下的“生成 EProc_DLL”命令，将源文件编译、链接成 DLL 文件。

为了测试创建的 DLL，利用 Intel Fortran 应用程序向导，创建一个 Console 类型的工程(EProc_Test)。其主程序文件如下所示：

```
Program EProc_Test
  Implicit None
  Interface
    Function GetRSqrt(a,b) result(r)
      Real a,b,r
    End Function GetRSqrt
  End Interface
  Real :: a = 3.0 , b = 4.0
  Write(*,*) GetRSqrt( a , b )
End Program EProc_Test
```

外部例程的使用，通常需在调用程序中建立其接口块，如上列代码所示。对于简单的接口，可只声明函数的数据类型、子程序的外部(External)类型。

设置链接器选项，在测试工程属性页的 Link/Input/Additional Dependencies 编辑框中添加 EProc_DLL.Lib(图 9-1)，以将前面生成的 DLL 导入文件链接到工程中。

将前面生成的EProc_DLL.dll和EProc_DLL.Lib两个库文件复制到测试程序所在的目录下。

测试程序运行结果如图 9-2 所示。

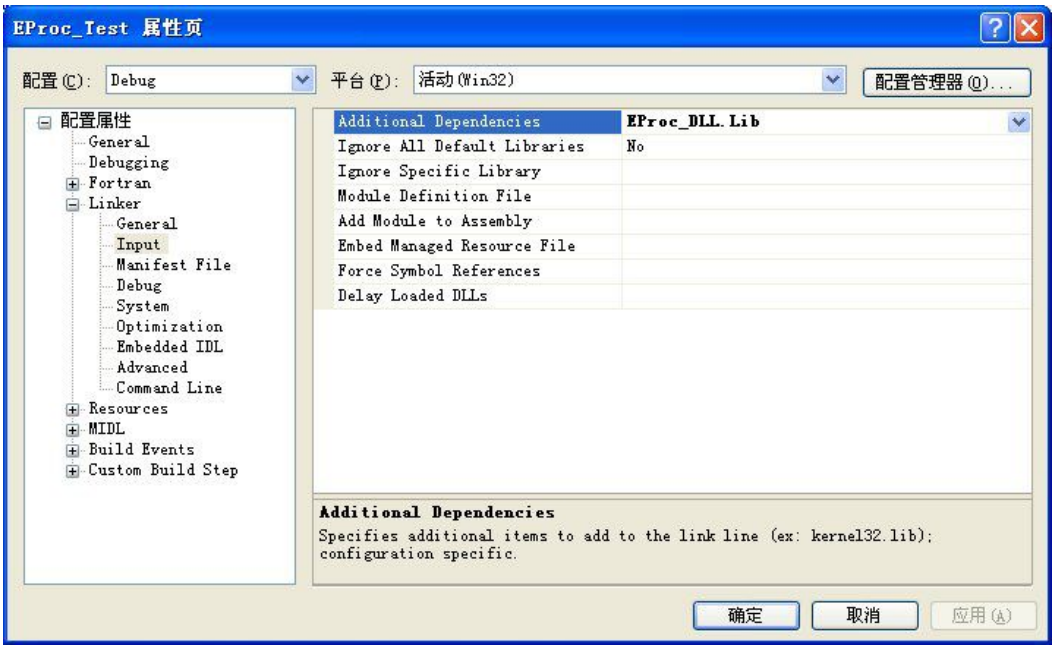



图 9-1 在属性页中添加 DLL 的导入文件

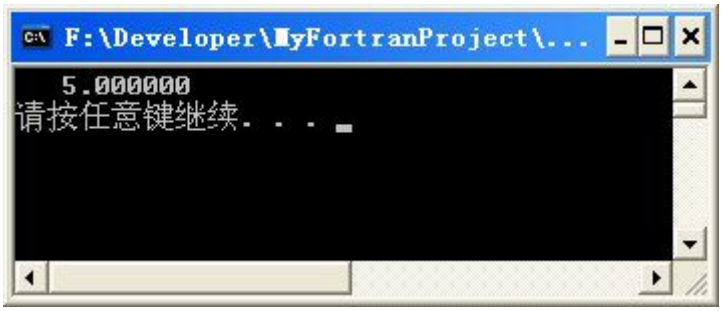



图 9-2 例 9-1 测试程序运行结果

### 9.2.2.2 模块例程

【例 9-2】将例 9-1 的外部例程(函数)改写为模块例程(子程序),然后将模块插入 Intel Fortran DLL 中。

利用 Intel Fortran 应用程序向导，创建一个 Dynamic-Link Library 类型的工程(MProc_DLL)。在 DLL 工程中插入下列模块：

```
Module MProc
  Implicit None
  Contains
  Subroutine GetRSqrt(a,b,r)
    !DEC$ ATTRIBUTES DLLEXPORT::GetRSqrt
    Real a,b,r
    r = SQRT( a*a + b*b )
  End Subroutine GetRSqrt
End Module MProc
```

在.NET 环境中，执行“生成”菜单下的“生成 MProc_DLL”命令，将源文件编译、链接成 DLL 文件。

为了测试创建的 DLL，利用 Intel Fortran 应用程序向导，创建一个 Console 类型的工程(MProc_Test)。其主程序文件如下所示：

```
Program MProc_Test
  Use MProc
  Implicit None
  Real :: a = 3.0 , b = 4.0 , c
  Call GetRSqrt( a , b , c )
  Write(*,*) c
End Program MProc_Test
```

在使用上，DLL 中的模块和外部程序自身的模块没有什么两样。

设置链接器选项，在测试工程属性页的 Linker/Input/Additional Dependencies 编辑框中添加 MProc_DLL.Lib，以将前面生成的 DLL 导入文件链接到工程中。

将前面生成的 MProc_DLL.dll、MProc_DLL.Lib、MPro_DLL.Mod 三个文件复制到测试程序所在的目录下。

生成、运行测试程序。

从例子 9-1 和例 9-2 可以看出，Intel Fortran 共享 DLL 中的例程，必须的设置是声明导出的例程拥有 DLLEXPORT 属性，而使用 DLL 的外部程序无须进行例程的导入声明，但在隐式链接时需将 DLL 的导入文件(Lib)链接到外部程序的执行文件中。

## 9.2.3 共享 DLL 中的数据

### 9.2.3.1 模块数据

【例 9-3】查询学生某门课程的基本信息(学号和成绩)，要求找出最高分的学生，并统计出该班学生的平均分。

该实例的操作步骤同例 9-2，下面仅给出 Intel Fortran DLL 中的模块源文件和 Intel Fortran 控制台测试程序的源代码。

DLL 中的模块文件：

```
Module MData
  Implicit None
  Real :: Aver
  !DEC$ ATTRIBUTES DLLEXPORT::Aver
  Type Student
    Integer n
    Real mark
  End Type Student
  Contains
    Function FindMax(Stud,M)
      !DEC$ ATTRIBUTES DLLEXPORT::FindMax
      Integer M
      Type(Student) Stud(M),FindMax
      Integer :: i , j = 0
```

```
    Real :: t = 0.0 , s = 0.0
    Do i = 1 , M
      If ( t <= Stud(i).Mark ) then
        j = Stud(i).n
        t = Stud(i).mark
      End If
    End Do
    FindMax.n = j
    FindMax.Mark = t
    Do i = 1 , M
      s = s + Stud(i).mark
    End Do
    Aver = s / M
  End Function FindMax
End Module MData
```

在模块文件中，定义了一个学生(Student)派生类型，并声明了代表平均分的全局变量 Aver，查找最高分的学生和统计平均分的算法由模块函数 FindMax 实现。

需要注意的是，外部程序要访问的模块变量和模块例程均需进行导出声明，但派生类型例外。

使用共享 DLL 的外部程序如下：

```
Program MData_Test
  Use MData
  Implicit None
  Type(Student) :: s(3) = (/Student(1001,70),Student(1002,80),Student(1003,90)/),Fs
  Fs = FindMax( s , 3 )
  Write(*,*) Fs.n , Fs.mark , Aver
End Program MData_Test
```

在缺省情况下，模块中定义的派生类型的成员访问属性为 Public。所以，引用模块的外部程序可直接使用派生类型构造子初始化变量。如上列代码所示。

#### 9.2.3.2 共用区数据

【例 9-4】将例 9-1 的 DLL 函数计算所用到的数据通过共用区传递。

该实例的操作步骤同例 9-1，下面仅给出 Intel Fortran DLL 中的导出函数和 Intel Fortran 控制台测试程序的源代码。

DLL 中的导出函数如下：

```
Function GetRSqrt() result(r)
  Implicit None
  Real r
  Real i , j
  COMMON /cb/ i , j
  !DEC$ ATTRIBUTES DLLEXPORT::/cb/ , GetRSqrt
  r = SQRT( i*i + j*j )
End Function GetRSqrt
```

程序中，函数和其包含的共用区均需进行导出声明。

使用共享 DLL 的外部程序如下：

```
Program CData_Test
  Implicit None
  Real a , b , GetRSqrt
  COMMON /cb/ a , b
  !DEC$ ATTRIBUTES DLLIMPORT :: /cb/
  a = 3.0 ; b = 4.0
  Write(*,*) GetRSqrt()
End Program CData_Test
```

测试程序声明，在 Intel Fortran 中，唯有共享 DLL 中的共用区数据才需要在外部程序中进行导入声明，如上列代码所示。

### 9.2.4 跨进程共享 DLL 中的共用区数据

在加载 Win32 DLL 时，系统总是将 DLL 映射到调用程序(进程)的地址空间中。默认情况下，每个使用 DLL 的进程都有自己的所有 DLL 全局变量和静态变量的实例。如果需要在多个进程的 DLL 实例间共享数据，则可使用共享数据段或内存映射文件方法。这里，仅介绍使用共享数据段方法。

使用共享数据段存在下列限制：

(1)必须静态初始化共享数据段中的所有变量；

(2)所有共享变量放在编译 DLL 的指定数据段中；

(3)不应将特定于进程的信息存储在共享数据段中；

(4)不应将指针存储在共享数据段包含的变量中。

【例 9-5】创建一个仅包含数据块的 DLL，并在两个进程间共享数据块共用区中的数据(采用共享数据段方法)。

利用 Intel Fortran 应用程序向导，创建一个 Dynamic-Link Library 类型的工程(Process_DLL)。在 DLL 工程中插入下列数据块：

```
Block Data
  Implicit None
  Real i,j
  COMMON /cb/ i , j
  !DEC$ ATTRIBUTES DLLEXPORT :: /cb/
  Data i , j / 1.0 , 1.0 /
End Block Data
```

要在多个进程的 DLL 实例间共享共用区数据，除了对共用区进行导出声明外，还必须静态初始化共用区中的变量，且赋给变量的值不能使缺省值。例如，上列初始化语句将 1.0 赋给共用区中的两个变量；若将两个变量初始化为 0.0，系统将按没有初始化对待，因为 0.0 恰好是共用区实型变量的缺省值。

在生成 DLL 前，须在工程属性页的 Linker/Command Line/Additional Options 编辑框中添加“/section:.data,RWS”选项，如图 9-3 所示。

事实上，在 Intel Fortran DLL 中，共用区中的数据位于 DLL 的.data 标准数据段。因

此，程序中没有定义专门的共享数据节，而是将 DLL 中的.data 标准数据段规定为可读写的共享数据段(/section:.data,RWS)。

其中，属性字符 R 表示允许对数据进行读操作；W 表示允许对数据进行写操作；S 指在所有加载 DLL 的进程中共享段。

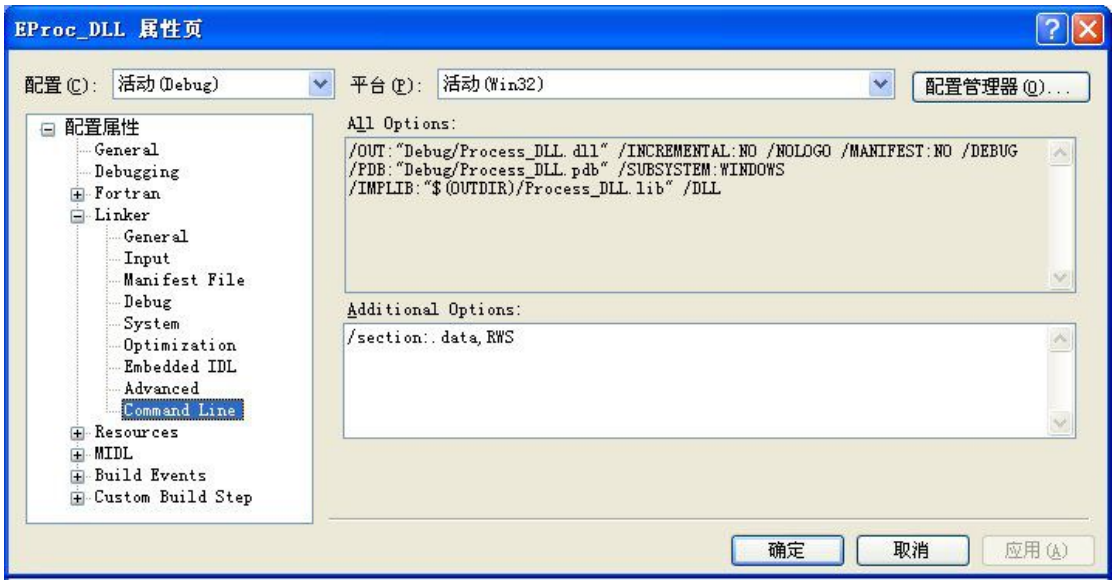



图 9-3 在属性页设置 DLL 的共享数据段选项

DLL 生成成功后，在另一目录下同时创建两个控制台测试程序 Process_Test1 和 Process_Test2。Process_Test1 的程序代码为：

```
Program Process_Test1
  Implicit None
  Real a , b
  COMMON /cb/ a , b
  !DEC$ ATTRIBUTES DLLIMPORT::/cb/
  Write(*,*) '暂停以接收Process_Test2通过共用区传递来的数据'
  Read(*,*)
  Write(*,*) Sqrt( a*a + b*b )
  a = 30.0
  b = 40.0
End Program Process_Test1
```

Process_Test2 的程序代码为：

```
Program Process_Test2
  Implicit None
  Real c , d
  COMMON /cb/ c , d
  !DEC$ ATTRIBUTES DLLIMPORT::/cb/
  c = 3.0 ; d = 4.0
  Write(*,*) '暂停以接收Process_Test1通过共用区传递来的数据'
  Read(*,*)
  Write(*,*) c , d
End Program Process_Test2
```

设置两个测试程序的链接器选项，以将前面生成的 DLL 导入文件链接到工程中。

将前面生成的Process_DLL.dll和Process_DLL.Lib两个库文件复制到测试程序所在的同一目录下，以确保两个进程加载的是同一份DLL。

两个测试程序的运行结果分别如图9-4和图9-5所示。

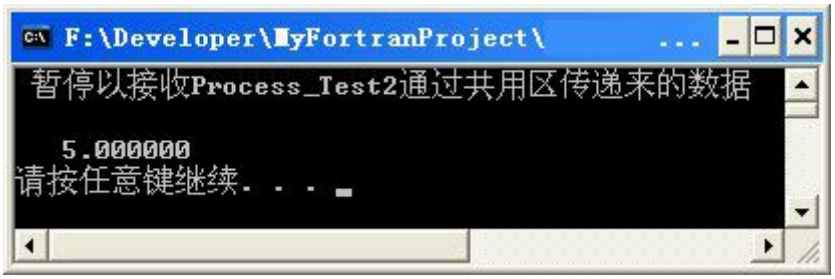



图9-4 测试程序Process_Test1运行结果

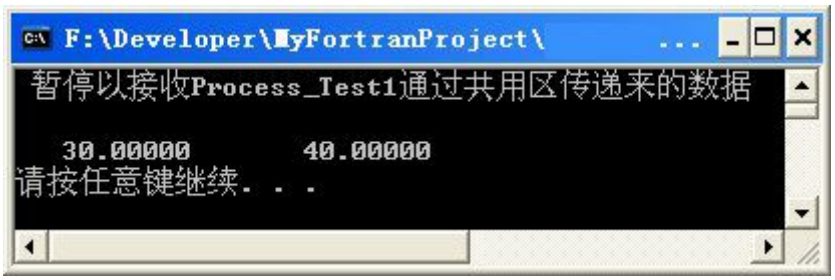



图9-5 测试程序Process_Test2运行结果

运行时，先启动Process_Test1并暂停，然后启动Process_Test2。Process_Test2将共用区的数据设为3.0和4.0后暂停；此时，共用区的两个数据已被更新为3.0和4.0，所以恢复执行Process_Test1,求得这两个数平方和的二次根为5.0；接下来Process_Test1将这两个数分别设为30.0和40.0，所以恢复执行Process_Test2，输出的正是这两个数。

实际上,如果有多个线程或进程同时向共用区写数据,那么就存在线程或进程同步问题。对于进程间同步，可使用第8章介绍的互斥体对象。例9-5采取了一种简单处理方式：当一个进程改写共用区数据时，另一个进程被暂停执行，以此来避免访问共用区数据出现冲突。

### 9.2.2 共享DLL中的对话框资源

事实上，不仅外部程序调用的例程和数据可置于DLL中，而且访问的对话框资源也可置于DLL中。

默认情况下，对话框例程在主应用程序中寻找对话框资源。如果将对话框资源置于DLL中，那么须调用DlgInitWithResourceHandle例程初始化对话框，并通知对话框例程在何处寻找对话框资源。与DlgInit例程相比,DlgInitWithResourceHandle例程多处一个“hinst”句柄参数，该参数代表包含对话框资源的模块实例句柄。针对DLL模块实例，该句柄需传递给DLL的入口点函数DllMain。

【例9-6】将图9-6所示的对话框资源插入DLL中，并提供相应的接口例程，以便外部程序可以直接使用对话框。




图9-6 例子9-6中的DLL所包含的对话框资源

利用 Intel Fortran 应用程序向导，创建一个 DLL 工程(PRGRS_DLL)，并在其中插入图 9-6 所示的对话框资源。资源脚本文件(PB.rc)命名避免使用工程缺省资源文件名 resource.rc，利用 defToFD 工具生成相应的头文件 resource.fd(具体设置参见 5.1 节)；

```
  integer, parameter :: IDD_PROGRESS_DIALOG = 101
  integer, parameter :: $_WIN32 = 1
  integer, parameter :: IDC_PROGRESS1 = 1000
```

对话框上的“取消”按钮 ID 为常量 IDCANCEL。

在模块中声明全局变量、常量、派生类型和接口；

```
!  Module containing definitions used in the public interfaces
module ProgressBoxTypes
  use iflogm
  !  Global variables
  integer ghinst
  !  PBContext type used in calls
  type :: PBContext
    sequence
    type (dialog), pointer :: PBDlg
    integer                hwndParent
    logical                bCancelButton
    integer                iProgress
    logical                bEnableParent
    logical                bCancelled
  end type
  !  Status return values
  integer, parameter, public :: PBStatus_OK         = 1
  integer, parameter, public :: PBStatus_Failed     = -1
  integer, parameter, public :: PBStatus_Aborted    = -2
end module ProgressBoxTypes

!  Module containing the public interfaces

module ProgressBox
  use ProgressBoxTypes
  interface
    integer function DisplayProgressBox(    &
                     context,           &
                     hwndParent,        &
                     title,             &
                     bCancelButton )
    !DEC$ ATTRIBUTES DLLEXPORT::DisplayProgressBox
      use ProgressBoxTypes
      ! Arguments
      type(PBContext) context
```

```
      integer hwndParent
      character*(*) title
      logical bCancelButton
    end function
    integer function SetProgress ( context, value )
    !DEC$ ATTRIBUTES DLLEXPORT::SetProgress
      use ProgressBoxTypes
      ! Arguments
      type(PBContext) context
      integer value
    end function
    integer function DestroyProgressBox( context )
    !DEC$ ATTRIBUTES DLLEXPORT::DestroyProgressBox
      use ProgressBoxTypes
      ! Arguments
      type(PBContext) context
    end function
  end interface
end module ProgressBox
```

这里，将变量、常量、派生类型和接口的声明分别放置在 ProgressBoxTypes 和 ProgressBox 两个模块中。

其中，全局变量 ghinst 用来存放 DLL 实例句柄；在 PBContext 派生类型中，包含了一个 dialog 类型的指针成员，该成员在初始化时需要动态分配内存；外部程序要调用的三个函数 DisplayProgressBox、SetProgress 和 DestroyProgressBox，分别用来展示进度条对话框、更新进度条控件值和销毁对话框，它们被声明具有 DLLEXPORT 属性。

实现导出例程。导出函数 DisplayProgressBox 的实现代码为：

```
integer function DisplayProgressBox(    &
                 context,               &
                 hwndParent,            &
                 title,                 &
                 bCancelButton )
  !DEC$ ATTRIBUTES DLLEXPORT::DisplayProgressBox
    use ProgressBoxTypes
    use user32

    implicit none

  ! Arguments
    type(PBContext) context
    integer hwndParent
    character*(*) title
    logical bCancelButton
```

```
! Variables
  include 'resource.fd'
  external PBDlgSub
  external CancelSub
  logical lret
  integer hwnd

! Body of DisplayProgressBox

  DisplayProgressBox = PBStatus_OK

! Initialize the context structure
  allocate( context % PBDlg )
  context % bCancelButton = bCancelButton
  context % hwndParent = hwndParent
  context % bCancelled = .FALSE.
  context % iProgress = 0

! Initialize the progress dialog box
  lret = DlgInitWithResourceHandle( IDD_PROGRESS_DIALOG, ghinst, context % PBDlg )
  if ( .not. lret ) then
      DisplayProgressBox = PBStatus_Failed
      return
  end if
  CALL DLGSETTITLE( context % PBDlg, title )
  lret = DlgSetSub( context % PBDlg, IDD_PROGRESS_DIALOG, PBDlgSub)
  if ( .not. lret ) then
      DisplayProgressBox = PBStatus_Failed
      return
  end if
  lret = DlgSetSub( context % PBDlg, IDCANCEL, CancelSub)
  if ( .not. lret ) then
      DisplayProgressBox = PBStatus_Failed
      return
  end if

!  Disable the parent window
   context % bEnableParent = .FALSE.
   if ( hwndParent .ne. 0 ) then
      lret = IsWindowEnabled( hwndParent )
      if ( lret ) then
           lret = EnableWindow( hwndParent, .FALSE. )
       context % bEnableParent = .TRUE.
      end if
```

```
    end if

  !  Display the progress dialog box
    lret = DlgModeless( context % PBDlg, SW_SHOWNORMAL, hwndParent)
    if ( .not. lret ) then
        DisplayProgressBox = PBStatus_Failed
        return
    end if

  !  Display the Cancel button if the caller requested it
    if ( bCancelButton ) then
         hwnd = GetDlgItem( context % PBDlg % hwnd, IDCANCEL )
        lret = ShowWindow( hwnd, SW_SHOWNA )
    end if

  !  Save the address of the context block as a property of the window
  !  We will retrieve it from there in our callbacks
    lret = SetProp( context % PBDlg % hwnd, "PBContext", %loc(context) )

end function DisplayProgressBox
```

该函数用来创建和显示无模式对话框 IDD_PROGRESS_DIALOG。其实现的相关功能包括：

● 初始化 context 派生类型，使用 allocate 命令为其对话框指针成员 PBDlg 动态分配内存；

● 调用 DlgInitWithResourceHandle 函数初始化 DLL 中的对话框，DLL 实例句柄通过其第二个参数传入。

● 调用 DlgSetSub 函数，将对话框 IDD_PROGRESS_DIALOG 和“取消”按钮 IDCANCEL 的回调例程分别设为 PBDlgSub 和 CancelSub；

● 在展示无模式对话框前，调用 EnableWindow 函数使对话框的父窗口失效。这样，行为上无模式对话框像是模式对话框；

● 调用 DlgModeless 函数，来展示无模式对话框；

● 若外部程序要求对话框具有“取消”按钮，则调用 ShowWindow 函数予以展示；

● 调用 SetProp 函数，将设置好的 context 派生类型存储在 PBDlg 窗口的属性链表中，以供对话框及其控件的回调例程使用。

导出函数 SetProgress 的实现代码为：

```
integer function SetProgress ( context, value )
  !DEC$ ATTRIBUTES DLLEXPORT::SetProgress
    use ProgressBoxTypes
    use ifwinty
    use user32
    implicit none
    ! Arguments
    type(PBContext) context
    integer value
```

```
    ! Variables
    include 'resource.fd'
    type (T_MSG) mesg
    logical lret
    logical lNotQuit
    integer iret

    ! Body of SetProgress
    SetProgress = PBStatus_OK

    ! Set the value of the Progress Bar
    lret = DlgSet( context % PBDlg, IDC_PROGRESS1, value )
    context % iProgress = value

    ! Dispatch all pending window messages
    lNotQuit = .true.
    do while ( lNotQuit .and. (PeekMessage(mesg, 0, 0, 0, PM_NOREMOVE) .ne. 0) )
         lNotQuit = GetMessage(mesg, NULL, 0, 0)
         if (lNotQuit) then
              if ( DlgIsDlgMessage(mesg) .EQV. .FALSE. ) then
                 lret = TranslateMessage( mesg )
                 iret = DispatchMessage( mesg )
              end if
        end if
       ! Check to see if the user pressed the Cancel button
        if ( context % bCancelled ) then
            context % bCancelled = .FALSE.      ! Reset
            SetProgress = PBStatus_Aborted
            return
        end if
    end do

end function SetProgress
```

该函数用来更新进度条控件值。值得注意的是，函数中插入了一个消息循环，用来不断分发当前队列中的各种消息，以便程序能够对用户点击“取消”按钮的操作做出及时响应。

PeekMessage 和 GetMessage 函数的作用类似，两者都用来检查消息队列，将符合过滤条件的消息拷贝至 MSG 结构；所不同的是，GetMessage 函数直到发现队列中有匹配的消息才返回，而 PeekMessage 函数在调用后立即返回。

当用户点击“取消”按钮时，SetProgress 函数返回 PBStatus_Aborted 状态码。

导出函数 DestroyProgressBox 的实现代码为：

```
integer function DestroyProgressBox( context )
  !DEC$ ATTRIBUTES DLLEXPORT::DestroyProgressBox
  use ProgressBoxTypes
  use user32
  implicit none
  ! Arguments
  type(PBContext) context
  ! Variables
  logical lret
  ! Body of DestroyProgressBox
  DestroyProgressBox = PBStatus_OK
  ! Enable the parent window
  if ( context % bEnableParent ) then
    lret = EnableWindow( context % hwndParent, .TRUE. )
  end if
  ! Free the dialog structure
  call DlgExit( context % PBDlg )
  call DlgUninit( context % PBDlg )
  deallocate( context % PBDlg )
end function DestroyProgressBox
```

该函数用来销毁对话框。销毁前，先恢复对话框父窗口的有效状态。对话框销毁的步骤是：先调用 DlgExit 例程关闭对话框；再调用 DlgUnInit 例程释放 DlgInitWithResourceHandle 例程初始化对话框所分配的内存；最后使用 Deallocate 命令释放 Allocate 为 context 派生类型的 PBDlg 指针成员动态分配的内存。

在导出函数 DisplayProgressBox 中，分别注册了对话框回调例程 PBDlgSun 和“取消”按钮回调例程 CancelSub。其中，回调例程 PBDlgSub 的实现代码为：

```
subroutine PBDlgSub( dlg, id, callbacktype )
  use ProgressBox
  use user32
  implicit none
  ! Arguments
  type (dialog) dlg
  integer id, callbacktype
  ! Variables
  type(PBContext) context
  pointer ( pc, context )
  integer iret
  if (callbacktype == dlg_destroy) then
    !  Remove the window property
    pc = GetProp( dlg % hwnd, "PBContext" )
    iret = RemoveProp( context % PBDlg % hwnd, "PBContext" )
  endif
end subroutine PBDlgSub
```

该回调例程是在用户关闭对话框时被调用。它获取对话框窗口的属性链表，从中删除添加的“PBContext”项。

回调例程 CancelSub 的实现代码为：

```
subroutine CancelSub( dlg, id, callbacktype )
  use ProgressBox
  use user32
  implicit none
  ! Arguments
  type (dialog) dlg
  integer id, callbacktype
  ! Variables
  type(PBContext) context
  pointer ( pc, context )
  integer iret
  ! Set the bCancelled flag in the context block.
  ! SetProgress checks this flag and returns PBStatus_Aborted
  if (callbacktype == dlg_clicked) then
    !  Get the "context" window property
    pc = GetProp( dlg % hwnd, "PBContext" )
    context % bCancelled = .TRUE.
  endif
end subroutine CancelSub
```

该回调例程是在用户点击“取消”按钮时被调用。它获取对话框窗口的属性链表，将 context 的 bCancelled 成员设为.TRUE.。导出函数 SetProgress 在检测到该设置后，将其返回状态码设为 PBStatus_Aborted。

实现 DLL 入口函数，建造、生成 DLL 二进制文件。入口函数 DllMain 的实现代码为：

```
integer(4) function DllMain (hInst, ul_reason_being_called, lpReserved)
!DEC$ IF DEFINED(_X86_)
!DEC$ ATTRIBUTES STDCALL, ALIAS : '_DllMain@12' :: DllMain
!DEC$ ELSE
!DEC$ ATTRIBUTES STDCALL, ALIAS : 'DllMain' :: DllMain
!DEC$ ENDIF
  use ProgressBox
  implicit none
  ! Arguments
  integer(4) hInst
  integer(4) ul_reason_being_called
  integer(4) lpReserved
  !  Save the module instance handle in a global variable
  ghinst =  hInst
  !  Return 1 to indicate success
  DllMain = 1
  return
end function DllMain
```

入口函数DllMain是系统在加载DLL时首先调用的函数。该DLL提供入口函数的目的是，将DLL实例句柄(hInst)存储在全局变量(ghinst)中。这样，DlgInitWithResourceHandle例程在初始化对话框时，就会依据这个句柄在DLL中寻找对话框资源。

利用Intel Fortran程序向导，创建一个基于对话框的Windows程序，来访问前面创建的DLL中的对话框。客户程序中的对话框及其控件ID声明如下：

```
!Resource.fd
  integer, parameter :: $_WIN32                 = 1
  integer, parameter :: IDD_CALLDLG_DIALOG      = 101
  integer, parameter :: IDC_MESSAGE             = 1006
  integer, parameter :: IDM_EXIT                = 30001
  integer, parameter :: IDC_START               = 30002
```

客户程序对话框IDD_CALLDLG_DIALOG，包含IDC_START和IDM_EXIT命令按钮，以及IDC_MESSAGE静态文本控件。

客户程序中的全局模块如下：

```
module CallDlgGlobals
  use IFLogM
  implicit none
  !  Parameters
  integer*4, parameter, public :: SIZEOFAPPNAME = 100
  !  Global data
  integer     ghInstance
  integer     ghModule
  integer     ghwndMain
  type (dialog) gdlg
end module CallDlgGlobals
```

客户程序中的主函数WinMain调整如下：

```
integer*4 function WinMain( hInstance, hPrevInstance, lpszCmdLine, nCmdShow )
!DEC$ IF DEFINED(_X86_)
!DEC$ ATTRIBUTES STDCALL, ALIAS : '_WinMain@16' :: WinMain
!DEC$ ELSE
!DEC$ ATTRIBUTES STDCALL, ALIAS : 'WinMain' :: WinMain
!DEC$ ENDIF
  use user32
  use kernel32
  use iflogm
  use calldlgGlobals
  implicit none
  integer*4 hInstance
  integer*4 hPrevInstance
  integer*4 lpszCmdLine
  integer*4 nCmdShow
  include 'resource.fd'
```

```
  external calldlgSub
  external calldlgStart
  ! Variables
  type (T_MSG)            mesg
  integer*4               ret
  logical*4               lret
  ghInstance = hInstance
  ghModule = GetModuleHandle(NULL)
  ghwndMain = NULL
  lret = DlgInit(IDD_CALLDLG_DIALOG, gdlg)
  if (lret == .FALSE.) goto 99999
  lret = DlgSetSub(gdlg, IDD_CALLDLG_DIALOG, calldlgSub)
  lret = DlgSetSub(gdlg, IDC_START, calldlgStart)
  lret = DlgModeless(gdlg, nCmdShow)
  if (lret == .FALSE.) goto 99999
  ! Read and process messsages
  do while( GetMessage (mesg, NULL, 0, 0) )
   if ( DlgIsDlgMessage(mesg) .EQV. .FALSE. ) then
     lret = TranslateMessage( mesg )
     ret  = DispatchMessage( mesg )
   end if
  end do
  call DlgUninit(gdlg)
  WinMain = mesg.wParam
  return
99999 &
  ret = MessageBox(ghwndMain, "Error initializing application calldlg"C,"Error"C, MB_OK)
  WinMain = 0
end function WinMain
```

主函数中，无模式对话框共享主消息循环，对话框 IDD_CALLDLG_DIALOG 的回调例程注册为 CallDlgSub，IDC_START 按钮的回调例程注册为 CallDlgStart。

实现客户程序中的对话框及控件回调例程。其中，对话框回调例程 CallDlgSub 的实现代码为：

```
SUBROUTINE calldlgSub( dlg, id, callbacktype )
  use user32
  use iflogm
  implicit none
  type (dialog) dlg
  integer id, callbacktype
  if (callbacktype == dlg_destroy) then
    call PostQuitMessage(0)
  endif
END SUBROUTINE calldlgSub
```

该例程在用户点击“退出”按钮(IDM_EXIT)时被执行，它调用 PostQuitMessage 例程向消息队列发送 WM_QUIT 消息，停止应用程序的运行。

“启动”按钮(IDC_START)回调例程 CallDlgStart 的实现代码为：

```
SUBROUTINE calldlgStart( dlg, id, callbacktype )
  use iflogm
  use ProgressBox
  implicit none
  ! Arguments
  type (dialog) dlg
  integer id, callbacktype
  ! Variables
  include 'resource.fd'
  type (PBContext) context
  integer iret
  logical lret
  integer progress
  logical bCancelled
  if (callbacktype == dlg_clicked) then
    ! Change the dialog message
    lret = DlgSet( dlg, IDC_MESSAGE, "执行..." )
    call DlgFlush( dlg )
    ! Create the progress dialog box
    ! We specify the main dialog as the parent window
    iret = DisplayProgressBox( context, dlg % hwnd,"DLL中的对话框", .TRUE.)
    if ( iret == PBStatus_OK ) then
      bCancelled = .FALSE.
      progress = 0
      ! Perform the work, updating the progress bar as we go along...
      do while ( ( bCancelled == .FALSE. ) .and. ( progress < 101 ) )
        iret = SetProgress ( context, progress )
        if ( iret == PBStatus_Aborted ) then
          bCancelled = .TRUE.
        else
          call sleepqq(250)      ! Wait .25 seconds, 100 times
        end if
        progress = progress + 1
      end do
      ! Destroy the progress box
      iret = DestroyProgressBox( context )
    end if
    ! Change the dialog message
    lret = DlgSet( dlg, IDC_MESSAGE, "点击'启动'按钮以执行任务" )
  endif
END SUBROUTINE calldlgStart
```

该例程在用户点击“启动”按钮(IDC_START)时被执行。它依次调用 DLL 的三个导出函数 DisplayerProgressBox、SetProgress 和 DestoryProgressBox，分别用来展示 DLL 中的对话框、设置进度条及销毁对话框。

在设置进度条值时，使用一个循环结构，且每隔 0.25s 设置依次，以提供用户随时取消执行任务的机会。在调用 SetProgress 函数后，检测其返回值状态码，如果状态码是 PBStatus_Aborted，则表示用户按下了“取消”按钮，作为回应，程序在执行下一步循环前跳出循环，并调用 DestoryProgressBox 函数销毁对话框。

将前面生成的 DLL 库、Lib 库连同模块中间文件 MOD 一起复制到客户程序所在目录下，编译、链接客户程序，程序运行结果见图 9-7。




图 9-7 访问放置在 DLL 中的对话框

## 9.3 动态链接库的使用

应用程序可以调用的 DLL 例程，在 DLL 中叫做导出例程，而在应用程序中叫做导入例程。应用程序中的导入例程与 DLL 中的导出例程进行链接有隐式链接和显式链接两种方式。

### 9.3.1 隐式链接

在建立一个 DLL 文件时，编译器会自动生成一个与该文件对应的导入库文件(扩展名为 Lib)。该文件包含了 DLL 的输出表，输出表中提供了所有导出例程及 DLL 库德相关信息，应用程序可以根据这个文件寻找并加载 DLL。因此，在需要使用 DLL 的应用程序中可以把该 DLL 对应的导入文件链接到应用程序工程中。这样，当应用程序在运行过程中需要加载 DLL 时，Windows 就会根据导入文件提供的相关信息，按照某种顺序搜索并加载 DLL，然后再通过例程名实现对 DLL 例程的动态链接。由于程序中没有出现加载 DLL 的代码，所以把这种链接方式叫做隐式链接。

Windows 搜索 DLL 的顺序为：

(1)包含应用程序 EXE 文件的目录；

(2)应用程序的当前工作目录；

(3)Windows 系统目录；

(4)Windows 目录；

(5)列在 Path 环境变量中的一系列目录。

本章前面例题的测试程序，都是使用隐式链接来调用 DLL。

### 9.3.2 显式链接

如果应用程序使用 Windows API 例程直接完成 DLL 库及其例程的调用，那么这种方式就叫做 DLL 的显式链接方式。当然，显式链接方式下不必再使用导入库文件。具体操作步骤如下：

#### 9.3.2.1 加载 DLL

调用 Win32 API 函数 LoadLibrary 加载 DLL 库。如果加载成功，函数将指定的 DLL 映射到调用进程的地址空间并返回此 DLL 的句柄，该句柄可被后面将要调用的显式链接函数 GetProcAddress 和 FreeLibrary 所使用。

LoadLibrary 函数的原型如下：

```
HINSTANCE LoadLibrary(
    LPCTSTR lpLibFileName    //可执行模块文件名
);
```

函数返回值为 DLL 的句柄。

#### 9.3.2.2 获取 DLL 例程

调用 Win32 API 函数 GetProcAddress 来获取 DLL 导出例程，然后使用返回的例程指针调用 DLL 例程。

GetProcAddress 函数的原型如下：

```
FARPROC GetProcAddress(
    HMODULE hModule,      //DLL的句柄
    LPCSTR lpProcName   //目标例程名
);
```

函数返回值为 DLL 导出例程地址。

#### 9.3.2.3 释放 DLL

使用完 DLL 后，调用 Win32 API 函数 FreeLibrary 来释放动态链接库。

FreeLibrary 函数的原型如下：

```
BOOL FreeLibrary(
    HMODULE hLibModule  //DLL的句柄
);
```

函数调用成功返回非 0；否则，函数返回 0。

【例 9-7】创建一个 Fortran 控制台程序，采用显式链接方式调用例子 9-1 的 DLL 导出例程(函数)GetRSqrt。

*(1)定义 DLL 导出例程的接口*

例程名不必相同，但例程参数表必须和导出例程的参数表相同；如果例程是函数，那么函数的数据类型也必须与导出函数的数据类型相同。该例定义的接口为：

```
Interface
  Function Func(a,b) result(r)
    Real a , b , r
  End Function Func
End Interface
```

*(2)声明 DLL 句柄指针及 DLL 导出例程指针*

该例的声明语句为：

```
Integer(4) i
Pointer( p , i )
Pointer( q , Func )
```

在 32 位系统下，地址编码使用 4 字节的整型数，上例第二条语句初始化一个整型指针 p，以存放 DLL 句柄的地址；同理，上列第三条语句初始化指向 DLL 导出例程的指针 q，使之先指向与 DLL 导出例程相对应的 Func。

值得注意的是，这里的 Pointer 句法是 Intel Fortran 编译器增加的扩展句法，Fortran 90/95 并没有这样的 Pointer 句法。

(3)显式链接

依次调用 LoadLibrary、GetProcAddress 和 FreeLibrary 函数。这里要特别注意：GetProcAddress 函数中的第二个参数要求使用目标例程名。通常，Fortran 语言不区分字母的大小写，但当程序中用到目标例程名时就必须区分字母的大小写。大多数 Fortran 编译器(包括 Intel Fortran 编译器)在将源代码转换成目标代码时，都将例程名统一转换成大写形式。所以，该例采取了下列调用形式：

```
q = GetProcAddress( p , "GETRSQRT"C )
```

其中，p 指 DLL 句柄指针；q 指针存放函数返回的例程地址；引号内的目标例程名为大写，后缀 C 表示该字符串是以空格结尾的 C 语言字符串。

完整的程序代码如下：

```
Program EProc_Link
  Use Kernel32
  Implicit None
  !Declare an interface block to the routine to be called.
  Interface
    Function Func(a,b) result(r)
      Real a , b , r
    End Function Func
  End Interface
  Integer(4) i
  Pointer( p , i )
  Pointer( q , Func )
  Logical status
  Real :: a = 3.0 , b = 4.0
  !First,locate the dll and load it into memory
  p = LoadLibrary( "EProc_DLL.dll"C )
  If ( p == 0 ) then
```

```
    Type *, "Error occurred opening EProc_DLL.dll"
    Type *, "Program aborting"
    Read(*,*)
    Stop
  End If
  !Set up a pointer to the routine of interest
  q = GetProcAddress( p , "GETRSQRT"C )
  If ( q == 0 ) then
    Type *, "Error occurred finding GetRSqrt in EProc_DLL.dll"
    Type *, "Program aborting"
    Read(*,*)
    Stop
  End If
  !Now,the routine can simply be called
  Write(*,*) Func( a , b )
  status = FreeLibrary( p )
  Type *, "FreeLibrary status was:",status
End Program EProc_Link
```

## 小　　结

■ 动态链接库(DLL)是程序运行时装载和连接的一种二进制文件，主要是通过它的导出例程、数据来向外界提供服务，并允许同时被多个应用程序所共享。

■ Intel Fortran DLL 可导出的例程包括外部例程和模块例程；可导出的数据有模块中的全局变量、派生类型和外部例程(或数据块)中的共用区数据。

■ 将对话框置于 DLL 中，需要调用 DlgInitWithResourceHandle 例程初始化对话框，并通过入口函数 DllMain 传入 DLL 实例句柄。

■ 除模块中的派生类型外，导出的例程和数据必须声明具有 DLLEXPORT 属性。

■ 利用共享数据段方法，共用区中的数据可以在多个应用程序(进程)调用的 DLL 实例间实现共享。

■ 应用程序调用 DLL 中的导出例程(和数据)有隐式链接和显式链接两种方式。

■ 在隐式链接方式下，需要将 DLL 的导入库(Lib)链接到调用程序中；若要访问 DLL 中的共用区导出数据，还需在调用程序中声明对应的共用区具有 DLLIMPORT 属性。

■ 在显式链接方式下，需要在调用程序中建立 DLL 导出例程的接口块。