

目 录

试验的注意事项	2
利用 Visual Studio 2005.net 开发 XSBASE270 应用程序说明	3
实验一 Windows CE 平台的构建与配置	8
实验二 线程同步调试实验	28
实验三 驱动程序结构实验	37
实验四 进程通信调试实验	50
实验五 SQL 数据库编程	56
实验六 IO 接口控制实验之七段数码管和 LED 显示实验	67
实验七 IO 接口控制实验——电机控制实验	75
实验八 动态链接库（DLL）实验	84
实验九 无线网络通信实验	91
实验十 UDP 和 Ping 实验	99
实验十一 驱动程序实验	109
实验十二 串口编程实验—GSM 和 GPS 实验	124
实验十三 CAN 总线实验	136
实验十四 SD/CF 存储卡读写实验	146
实验十五 USB 摄像头驱动和应用实验	153
实验十六 应用程序集成实验	164

试验的注意事项

- 1、VS2005 一定要是英文的才行，中文的会下载不成功，有些试验无法进行。
- 2、ActiveSync 要用比较新的版本，推荐使用本光盘中的。
- 3、请严格按照操作步骤做，通过 VS2005 的运行菜单来完成下载可执行文件和，和直接把可执行文件复制到目标板上执行的结果是不一样的。
- 4、SDK 推荐使用本光盘上的。
- 5、安装 VS2005 的 SP1 补丁的时候保证 C 盘上至少留有 2.5G 的空间，这样才能保证 2 到 3 个小时内完成安装。

利用 Visual Studio 2005.net 开发 XSBase270 应用程序说明

- 1、安装 Visual Studio 2005.net，最好采用全部安装
- 2、安装 XSBase270 开发平台的随机提供的 SDK
- 3、打开 VS2005(如图 1)

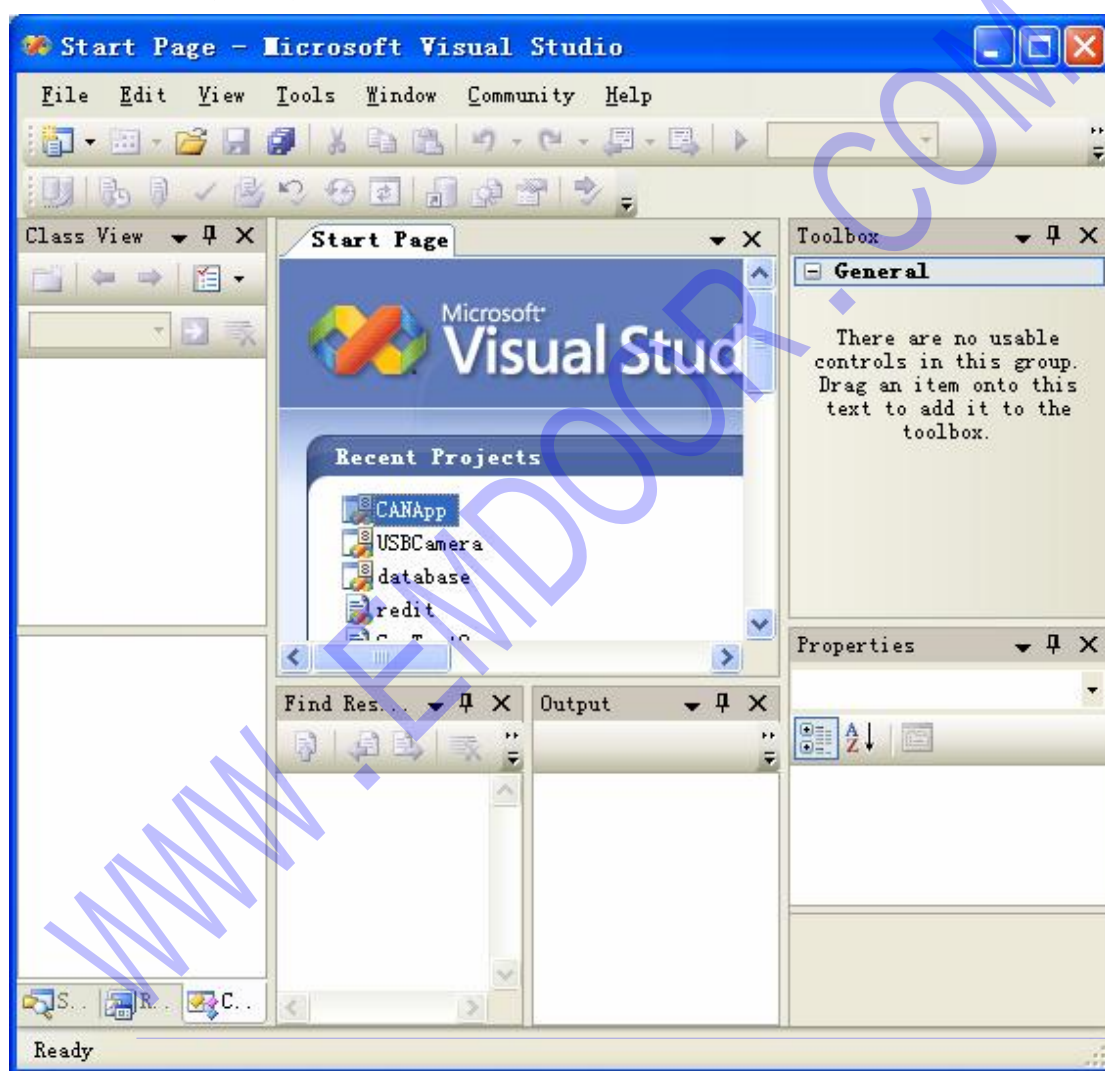


图 1

- 4、新建工程(File→New->Project 如图 2)，VS2005 弹出创建新工程的对话框（图 3），对于 XSBase270 开平台，选中“Smart Device”项（具体模板有基于 MFC 的 MFC Smart Device Application、基于 Win32 的 Win32 Smart Device Project 等），输入工程路径和工程名，按“OK”按钮

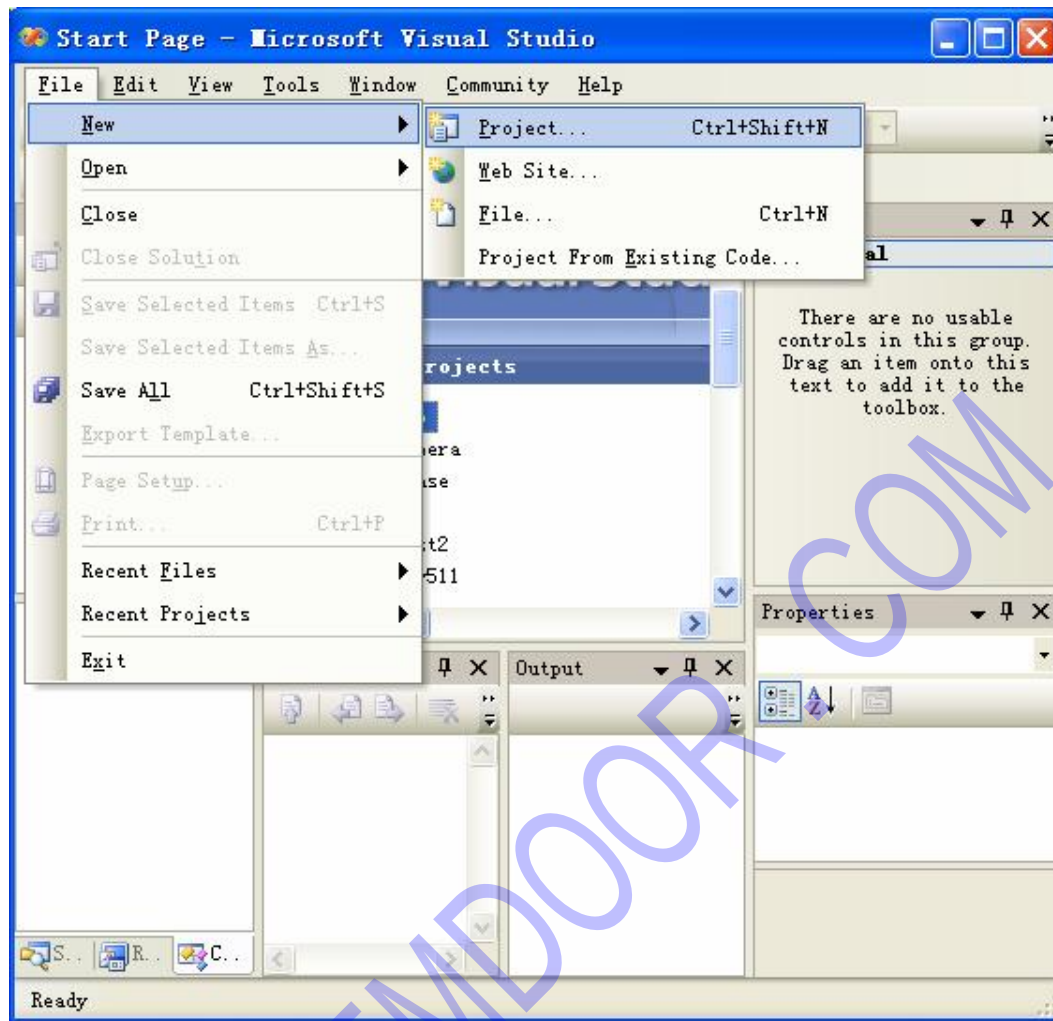


图 2

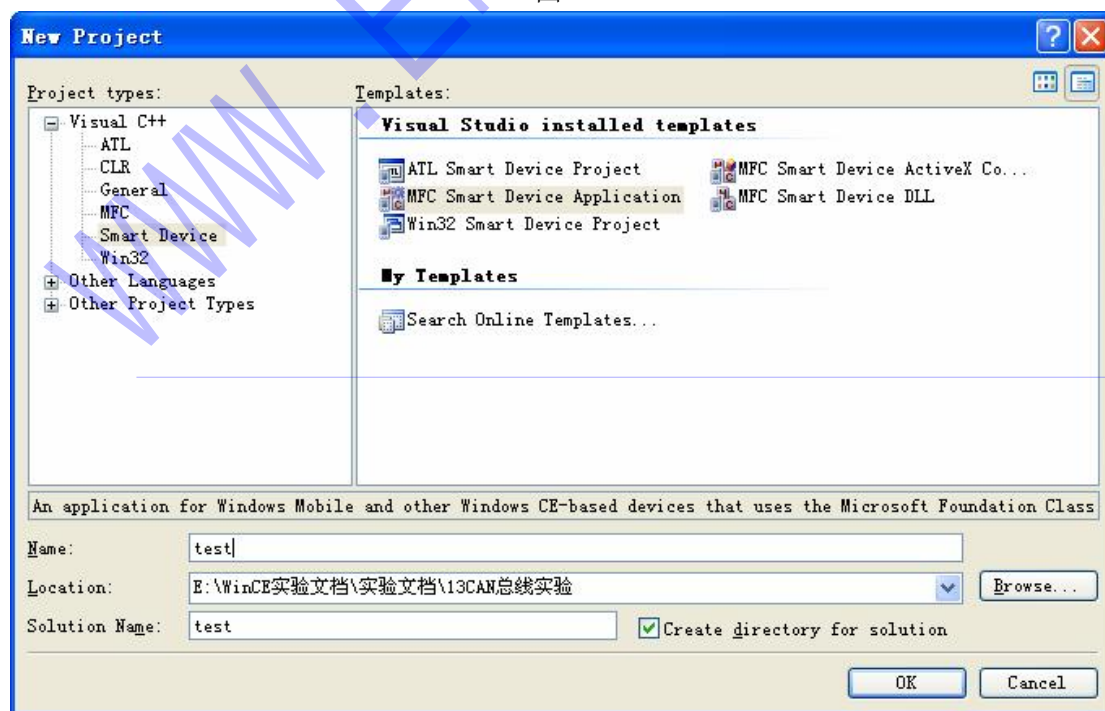


图 3

5、按“OK”按钮后，进入应用程序配置向导（如图 4），按“next”，进入应用平台选择

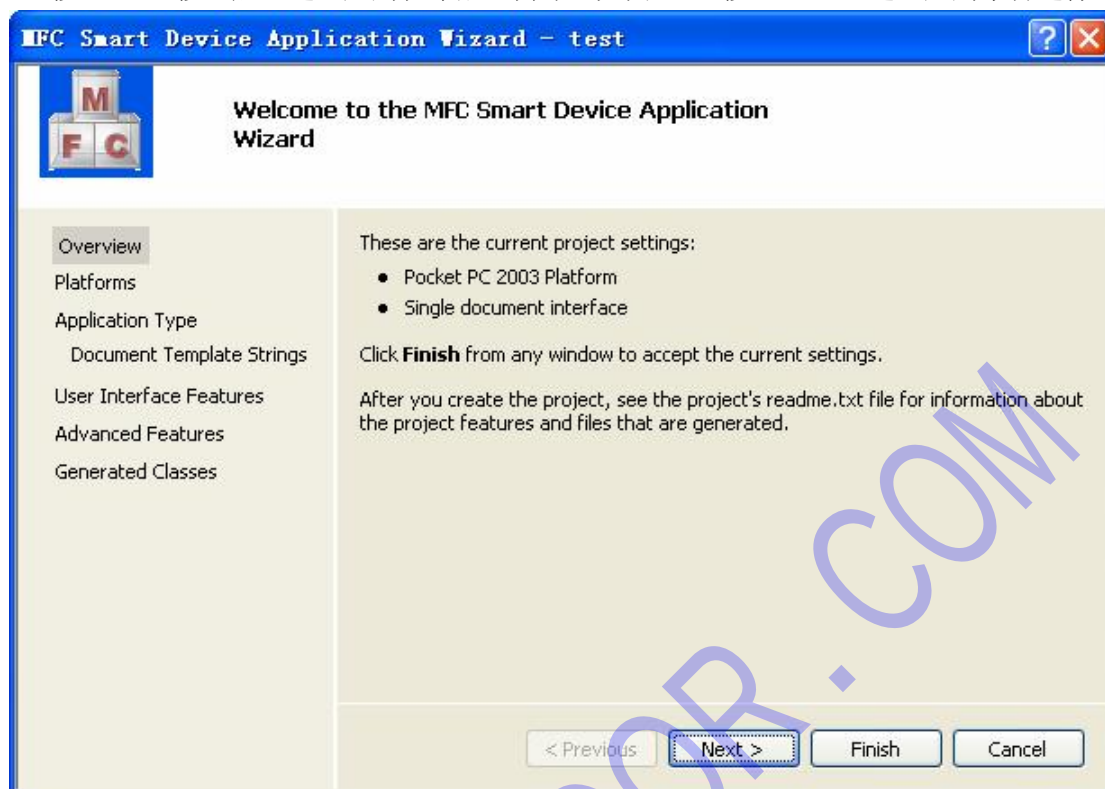


图 4 应用程序配置向导

6、平台选择：选择所需要的 SDK（如图 5），选好后按“next”，进入应用类型选择

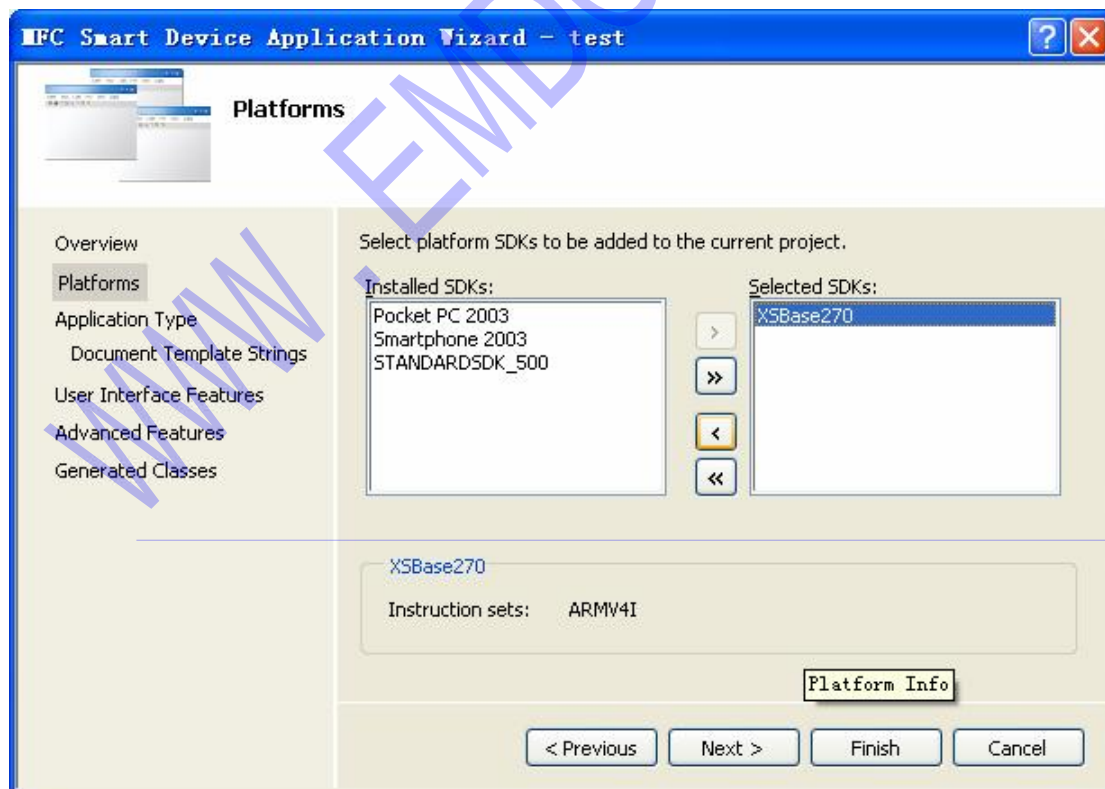
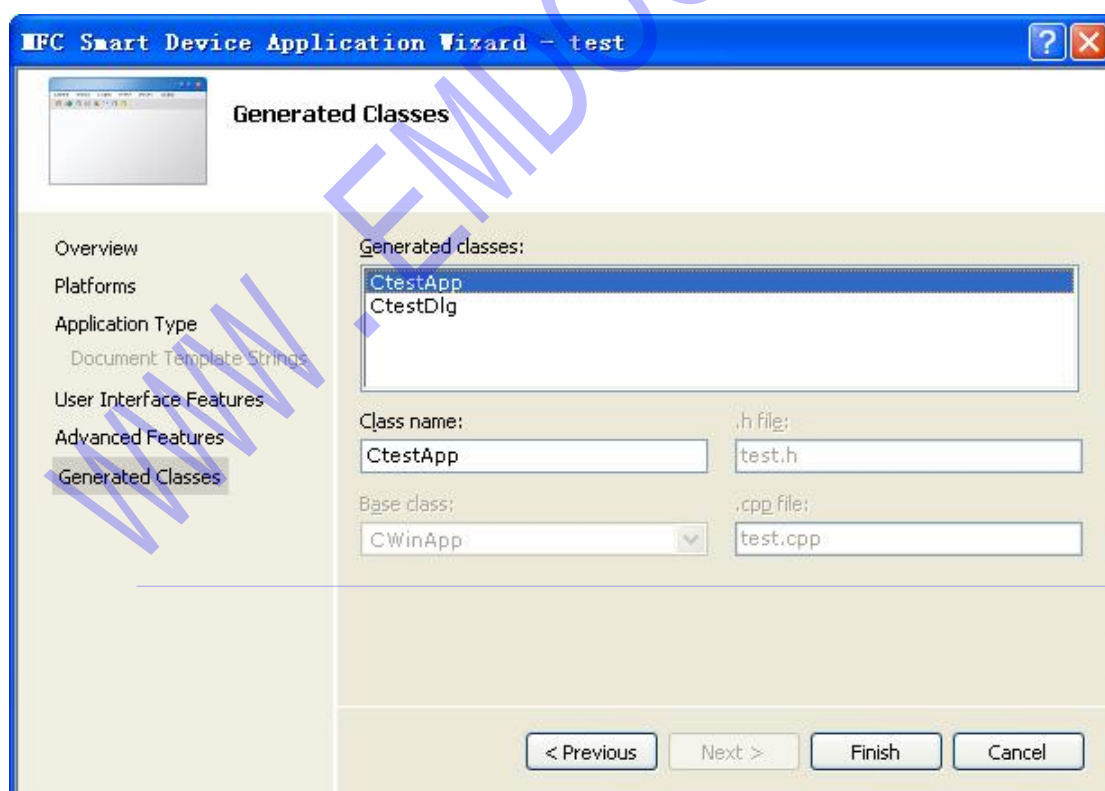
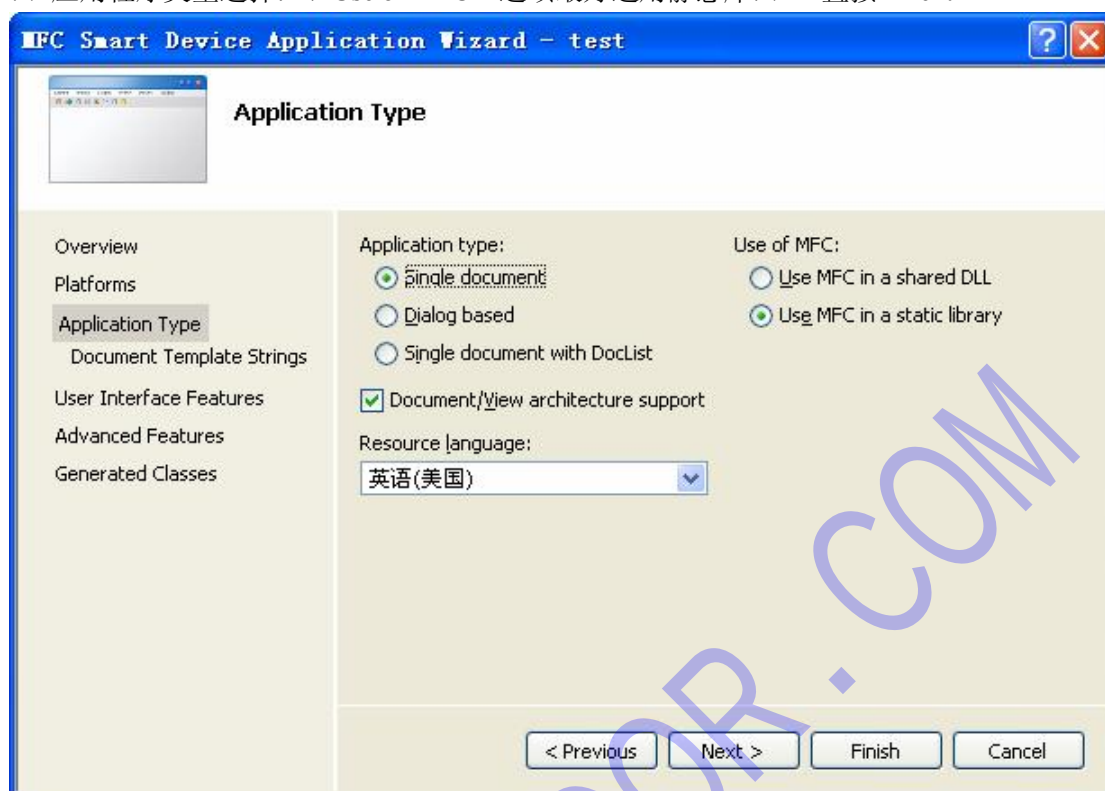
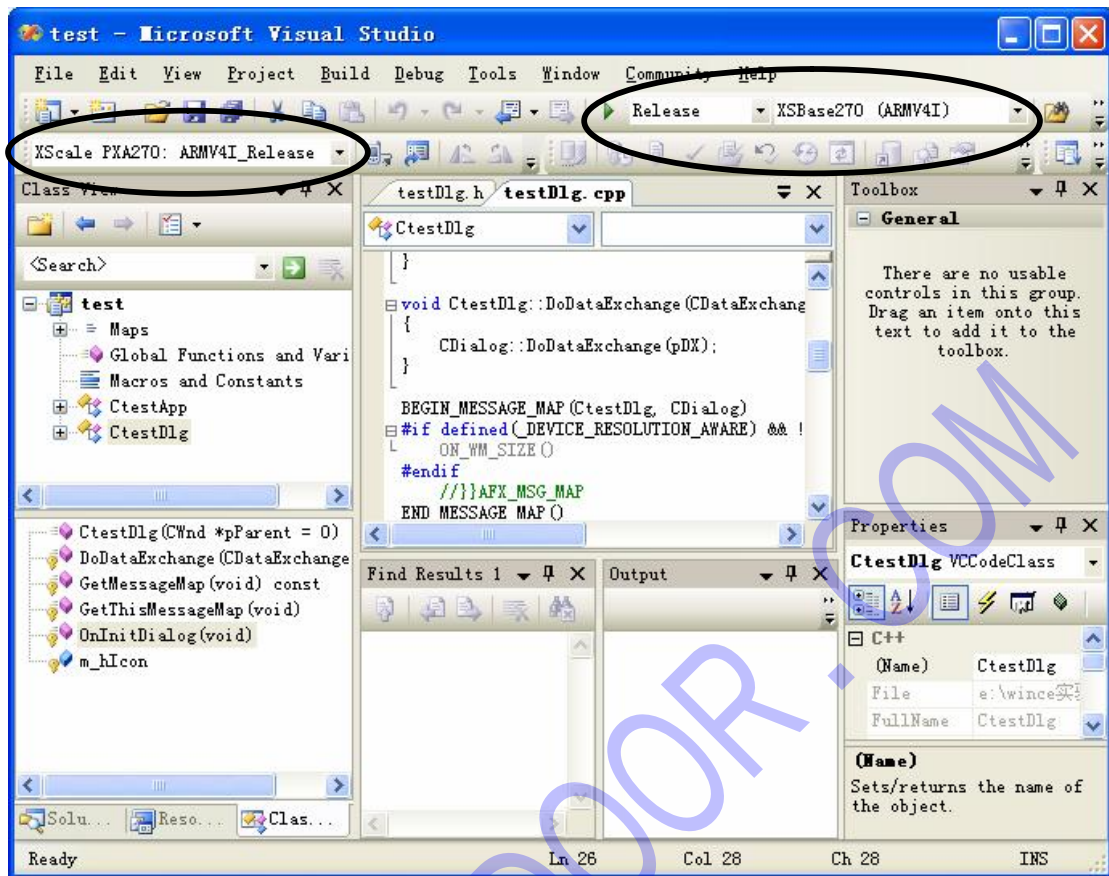


图 5 平台选择

7、应用程序类型选择：“Use of MFC” 选项最好选用静态库）、一直按 “next”



8、最后生成的应用程序



9、通过 ActiveSync 连好开发板，按运行按钮或 F5，程序进行编译并自动下载到目标平台中运行。

实验一 Windows CE 平台的构建与配置

[实验目的]

- 1、掌握 Windows CE 内核的配置、编译方法；
- 2、掌握构建一个适合特定开发平台的 Windows CE 系统方法；
- 3、熟悉 Platform Builder 开发工具使用方法。

[实验仪器]

- 1、装有 Platform Builder 开发环境的 PC 机一台
- 2、XSBase270 实验开发平台一套

[实验原理]

1、Windows CE 操作系统简介

Microsoft Windows CE 是一个开放的、可裁剪的、32 位的实时嵌入式窗口操作系统。和其他桌面窗口操作系统（Windows XP/2000）相比，它具有可靠性好、实时性高、内核体积小等特点，适用于各种嵌入系统和产品。它拥有多线程、多任务、确定性的实时、完全抢先式优先级的操作系统环境，专门面向只有有限资源的嵌入式硬件系统。同时，开发人员可以根据特定硬件系统对 Windows CE 操作系统进行裁剪、定制，所以目前 Windows CE 被广泛用于各种嵌入式智能设备的开发，是当今应用最多、增长最快的嵌入式操作系统。

Windows CE 被设计成为一种高度模块化的操作系统，每一模块都提供特定的功能，这些模块中的一部分被划分成组件，系统设计者可以根据设备的性质只选择那些必要的模块或模块中的组件包含进操作系统映像，从而使 Windows CE 变得非常紧凑（只占不到 200 KB 的 RAM），因此只占用了运行设备所需的最小的 ROM、RAM 以及其它硬件资源。

Windows CE 被分成不同的模块，其中最主要的模块有内核模块（Kernel）、对象存储模块、图形窗口事件子系统（GWES）模块以及通信（Communication）模块。另外 Windows CE 还包含一些附加的可选择模块，这些模块可支持的任务有管理可安装设备驱动程序、支持 COM 等。一个最小的 Windows CE 系统至少由内核和文件系统模块组成。

1.1 内核模块

内核模块是 Windows CE 操作系统的核心，它为任何基于 Windows CE 的设备提供处理器调度、内存管理、异常处理以及系统内通信等系统功能，并为应用程序使用这些核心功能提供内核服务。Windows CE 的内核模块通过 CoreDLL 模块表示。所有的操作系统定制设计都必须包含这个模块，但并不是这个模块的所有组件都必需的，有一些内核组件是可选的。

1.2 对象存储

对象存储是 Windows CE 的默认文件系统，它相当于 Windows CE 设备上的硬盘。对象存储是由共享一个内核堆的文件系统、系统数据库和系统注册表组成，即使在没有系统主电

源时,对象存储也能维持应用程序及相关数据不会丢失。对象存储可将用户数据和应用程序数据存入文件或注册器。在操作系统创建进程(该进程中只包括那些必需选项)的过程中,对于这些不同的对象存储组件,可以选取,也可以忽略。

1.3 图形窗口和事件系统模块

图形窗口和事件系统模块(GWES)包含大部分的核心 Windows CE 功能,它集成了图形设备接口(GDI)、窗口管理器和事件管理器。GWES 模块是 Windows CE 操作系统高度组件化的部分,它分别由 USER 和 GDI 两部分组成,USER 用来处理消息、事件及鼠标和键盘等用户输入,而 GDI 用于处理图形的屏幕和打印输出等。GWES 是用户、应用程序和操作系统之间的图形用户接口。GWES 通过处理键盘、鼠标动作与用户交互,并选择传送到应用程序和操作系统的信息。GWES 通过创建并管理在显示设备和打印机上显示的窗口、图形以及文本来处理输出。

GWES 的中心是窗口。所有应用程序都通过窗口接收来自操作系统的消息,即使那些为缺少图形显示的设备创建的应用程序也是如此。GWES 提供控制器、菜单、对话框以及图形显示的设备资源,还提供 GDI 以控制文本与图形显示。

1.4 通信模块

通信模块为基于 Windows CE 的设备提供有线或无线通信能力,使 Windows CE 设备能够与其他设备或计算机进行连接与通信,通信组件提供对下列通信硬件和数据协议的支持:

- 串行 I/O 支持
- 远程访问服务(RAS)
- 传输控制协议/Internet 协议(TCP/IP)
- 局域网(LAN)
- 电话技术 API(TAPI)
- WinCE 的无线服务

可选组件

除上述主要模块之外,还可使用其它的操作系统模块。这些模块与组件主要有:

- 设备管理器和设备驱动程序
- 多媒体(声音)支持模块
- COM 支持模块
- WinCE 外壳模块

WinCE 提供的每一模块或组件都支持一组可用的相关 API 函数。

2 Platform Builder 开发工具介绍

Platform Builder(PB)是微软提供给 Windows CE 开发人员进行基于 Windows CE 平台下嵌入式操作系统定制的综合开发环境。它提供了所有进行设计、创建、编译、测试和调试 Windows CE 操作系统平台的工具。它运行在桌面 Windows 下,开发人员可以通过交互式的环境来设计和定制内核、选择系统特性,然后进行编译和调试。该工具能够根据用户的需求,选择构建具有不同内核功能的 CE 系统。同时,它也是一个集成的编译环境,可以为所有 CE 支持的 CPU 目标代码编译 C/C++ 程序。一旦成功地编译了一个 CE 系统,就会得到一个名为 nk.bin 的映像文件。将该文件下载到目标板中,就能够运行 CE 了。

Platform Builder 提供了开发人员快速建立基于 Windows CE 嵌入式系统所需的各种工具。Platform Builder 的集成开发环境(IDE)允许开发人员配置、建立并调试能够借助 Windows 和 Web 强大功能为嵌入式系统带来灵活性与可靠性的新一代高度模块化设计方案。

Platform Builder 提供的主要特性包括：

- 平台开发向导（Platform Wizard）和 BSP 开发向导：开发向导用于引导开发人员创建简单的系统平台或 BSP（板级支持软件包），然后再根据要求进一步修改。开发向导提高了平台和 BSP 创建效率；
- 特性目录（Catalog）：操作系统可选特性均在特性目录（Catalog）中列出，开发人员可以选择相应的特性来定制操作系统；
- 导出向导（Export Wizard）：可以向其他 Platform Builder 用户导出自定义的目录（Catalog）特性；
- 导出 SDK 向导（Export SDK Wizard）：使用户可以导出一个自定义的软件开发工具包（SDK），可以将客户定制的 SDK 导出到特定的开发环境中（如 EVC）。
- 远程工具：可以执行同基于 Windows CE 的目标设备有关的各种调试任务和收集任务；
- 仿真器（Emulator）：通过硬件仿真加速和简化了系统的开发，使用户可以在开发工作站内对平台和应用程序进行调试，大大简化了系统的开发流程，缩短了开发时间。
- 应用程序调试器：可以在自定义的操作系统映像上对应用程序进行调试；
- 内核调试器：可以对自定义的操作系统映像进行调试，并且向用户提供有关映像性能的信息；
- 驱动测试工具包（Windows CE.net Test Kit）：系统为驱动程序开发提供了基本的测试工具集；
- 基础配置：为各种流行的设备类别预置的可操作系统基础平台，为自定义操作系统的创建提供了一个起点。

Platform Builder 的开发界面如图 1-1 所示。

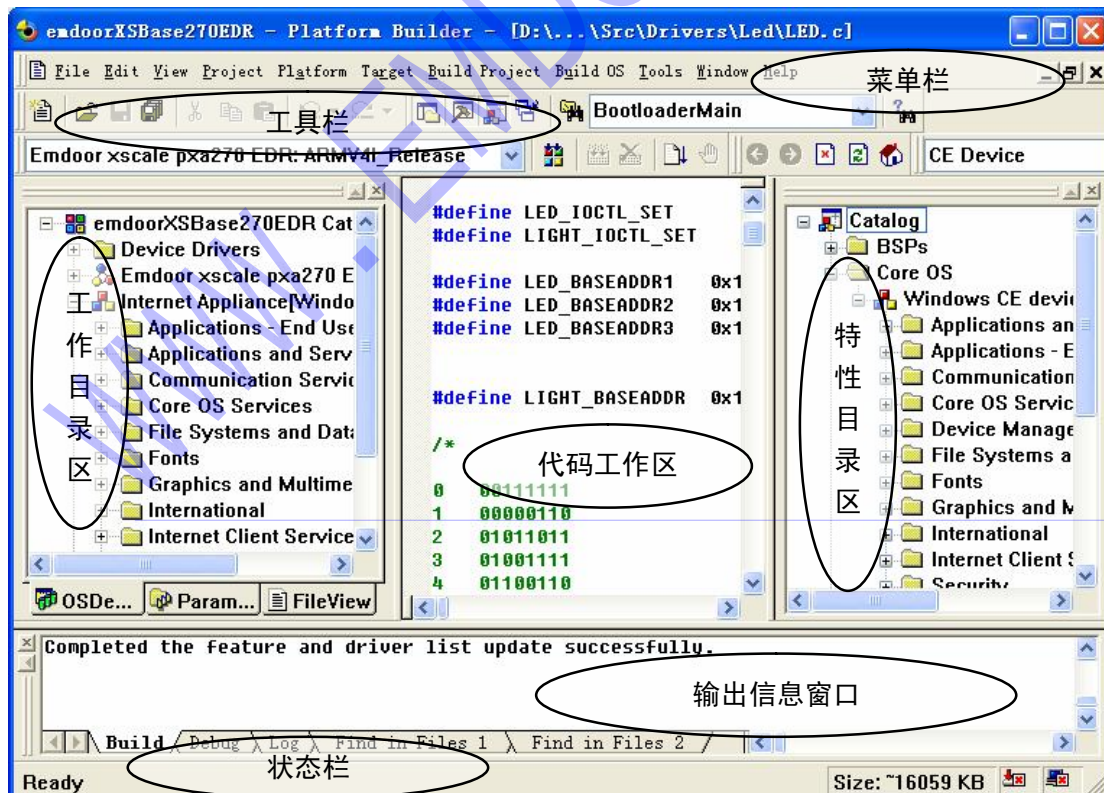


图 1-1 Platform Builder 开发界面

3 BSP 原理简述

BSP 又称板级支持软件包，它是一个包括启动程序、OEM 适配层程序（OAL）、标准开发板（SDB）和相关硬件设备驱动程序的软件包。BSP 是基于 Windows CE 平台系统的主要部分，它是由一些源码和二进制文件组成。

对于嵌入式系统来说，它没有像 PC 机那样具有广泛使用的各种工业标准，各种嵌入式系统的不同应用需求决定了它选用的各自定制的硬件环境，这种多变的硬件环境决定了无法完全由操作系统来实现上层软件与底层硬件之间的无关性。因此各种商用实时操作系统都采用了分层设计的方法，它将系统中与硬件直接相关的一层软件独立出来，称之为 Board Support Package (板级支持软件包，简称 BSP)。顾名思义，BSP 是针对某个单板而设计的，它对于用户(开发者)是开放的，用户可以根据不同的硬件需求对其作改动或二次开发，而操作系统本身仅仅提供了 CPU 内核的无关性。BSP 在系统中的角色，很相似于 BIOS 在 PC 统中的地位。BSP 在系统中所处的位置，如图 1-2 所示，它位于硬件平台与操作系统或应用软件之间，用于屏蔽上层软件对各种硬件的相关性。

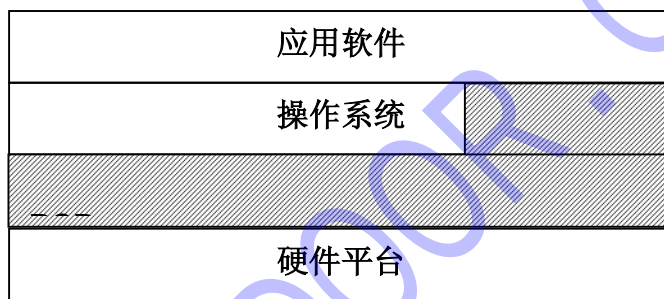


图 1-2 BSP 在系统中的位置

BSP 的主要功能在于配置系统硬件使其工作在正常状态，并且完成硬件与软件之间的数据交互，为 OS 及上层应用程序提供一个与硬件无关的软件平台。

在 Platform Builder 中，微软提供了对十几种标准开发板（SDB）支持的 BSP，这些 BSP 覆盖了所有 Windows CE 可支持的处理器类型，它可使开发者快速地评估各种操作系统特性并减少新产品开发时间。

[实验内容]

- 1、安装 XSBase270 实验开发平台的 BSP；
- 2、根据 XSBase 目标平台的特点和系统的需要，配置 Windows CE 操作系统的特性和功能；
- 3、编译、链接操作系统内核，生成系统映像文件；
- 4、下载并运行编译好的 Windows CE 系统。

[实验步骤]

1、安装 XSBase270 的 BSP

为了使 Platform Builder 支持 XSBase270 开发板的硬件资源系统, 在安装 Windows CE 之后需要安装 XSBase270 的板级支持软件包 (BSP)。具体安装步骤:

双击随开发板提供的 BSP 安装文件 XSBase270.msi, 进入 BSP 安装向导如图 1-3 所示。



图 1-3 XSBase270 BSP 安装向导

按“Next”按钮, 进入安装过程, BSP 自动选择 Windows CE 安装目录 (D:\WINCE500), 如图 1-4 所示。

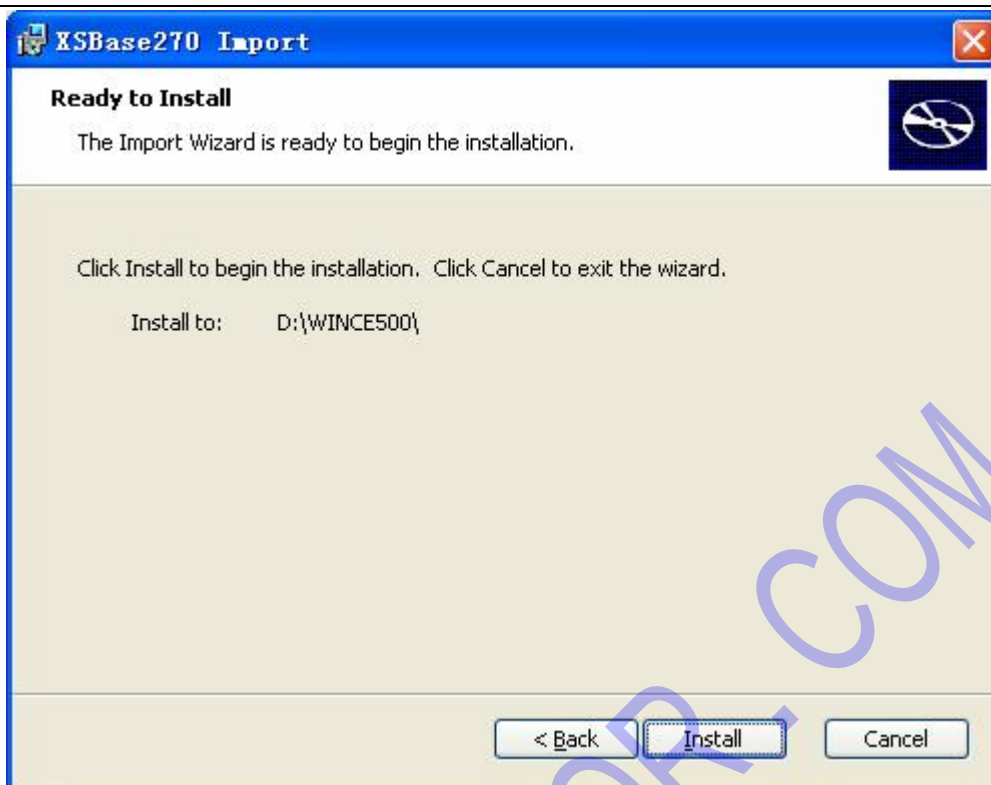


图 1-4 BSP 安装目录界面

按“Install”按钮，进行 BSP 的安装。如图 1-5 所示。

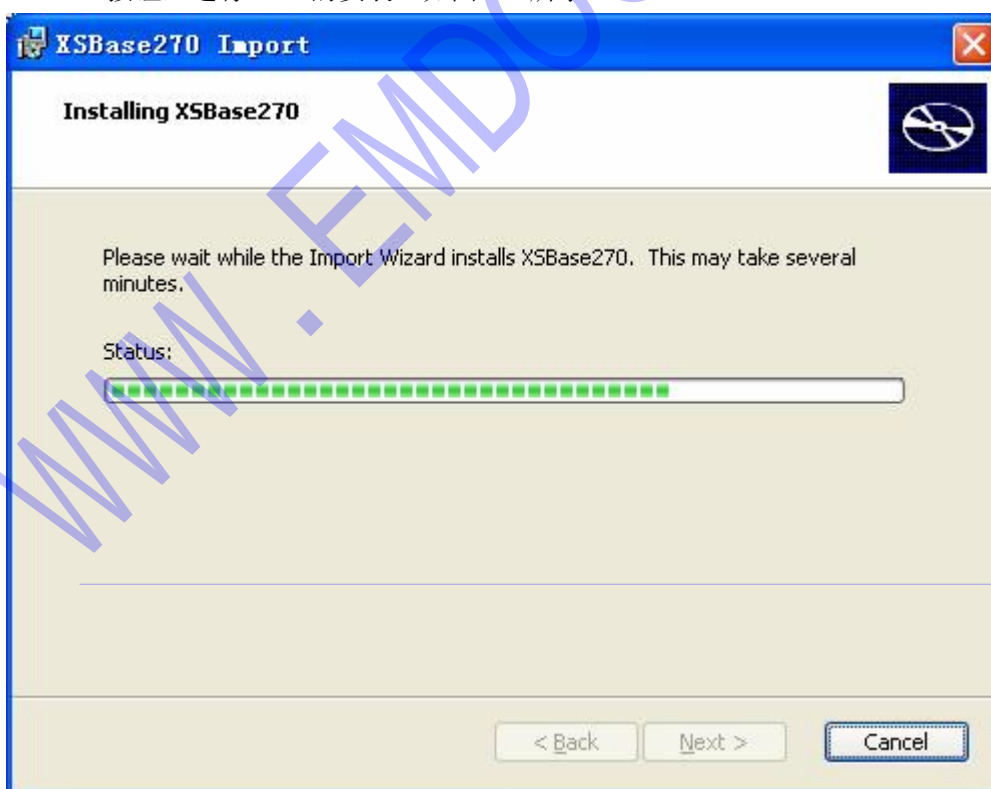


图 1-5 BSP 安装过程界面

BSP 安装完成后，安装向导提示安装完成界面，按“Finish”按钮，完成 BSP 的安装。如图 1-5 所示。

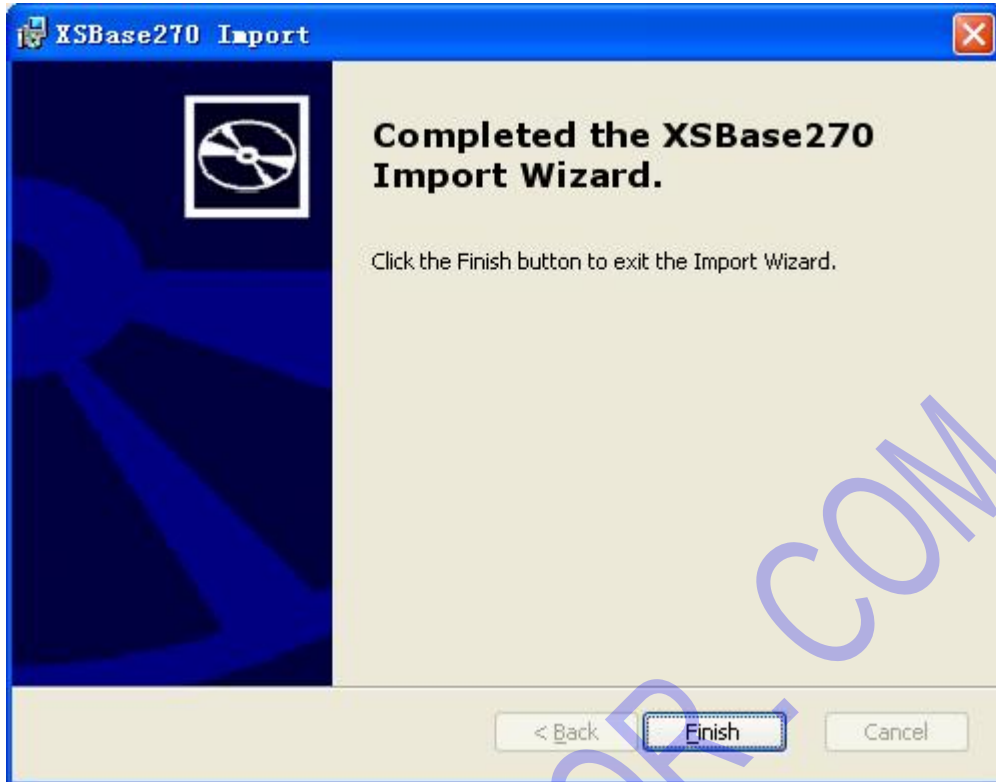


图 1-5 BSP 安装完成界面

XSBase270 的 BSP 安装完成后, 启动 Platform Builder, 在 Platform Builder 的特性目录区(Catalog)的第三方 BSP (Third Party) 节点下出现所安装的 XSBase270 的 BSP (Emdoor System XSBase270: ARMV4I), 如图 1-6 所示。

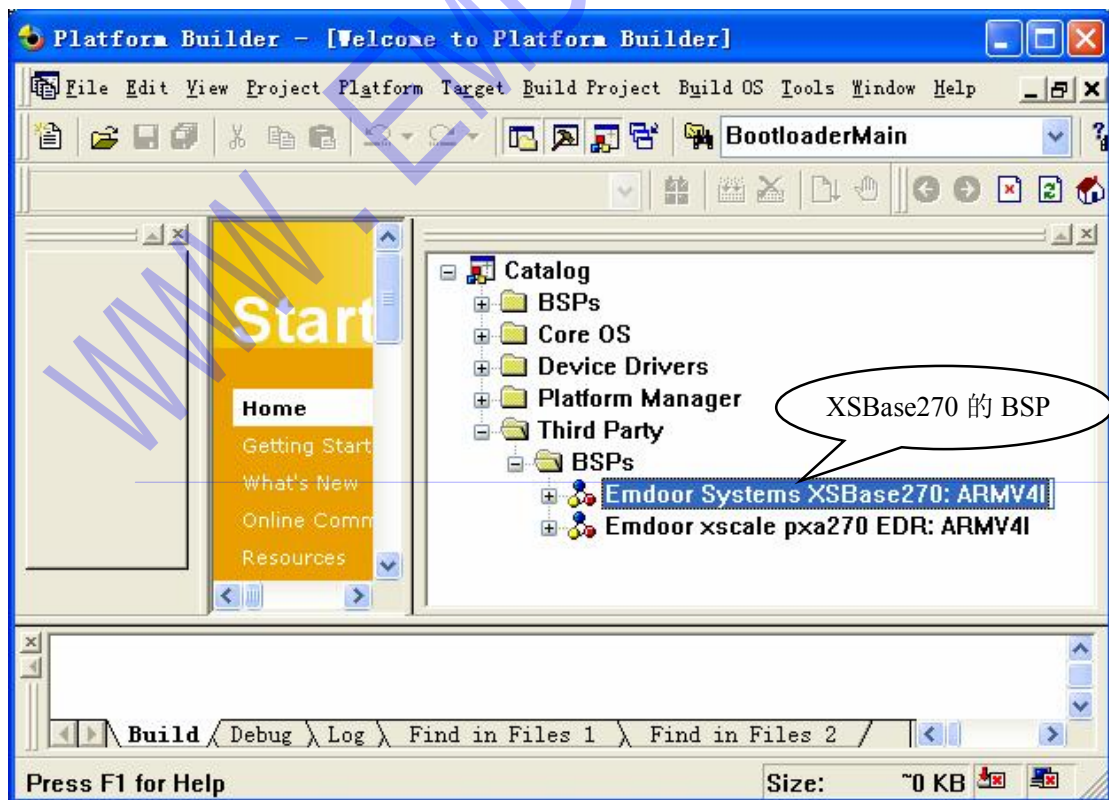


图 1-6 Platform Builder 中 XSBase270 的 BSP

2、实验平台的配置

第一步：新建一个目标平台：

- 选择菜单项“File” -> “New Platform Wizard.”;
- 输入目标平台名称和保存的位置，如图1-7所示

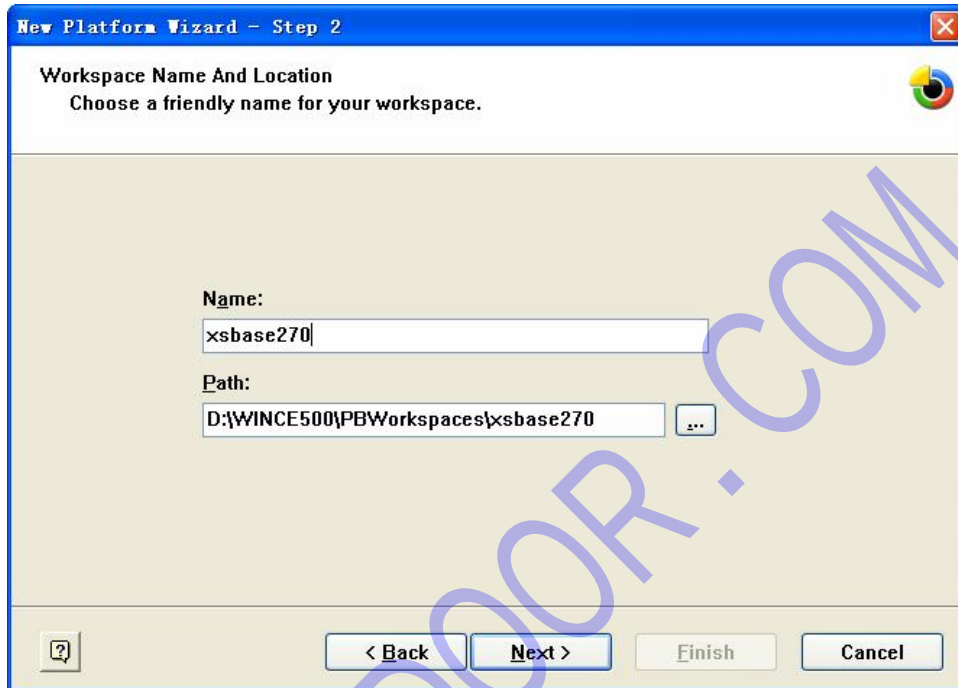


图 1-7 目标平台名称和保存路径界面

- 为目标平台选择所支持的BSP，XSBase270开发板选择选择“EMDOOR SYSTEM XSBA SE270:ARMV4I”平台，如图1-8所示；

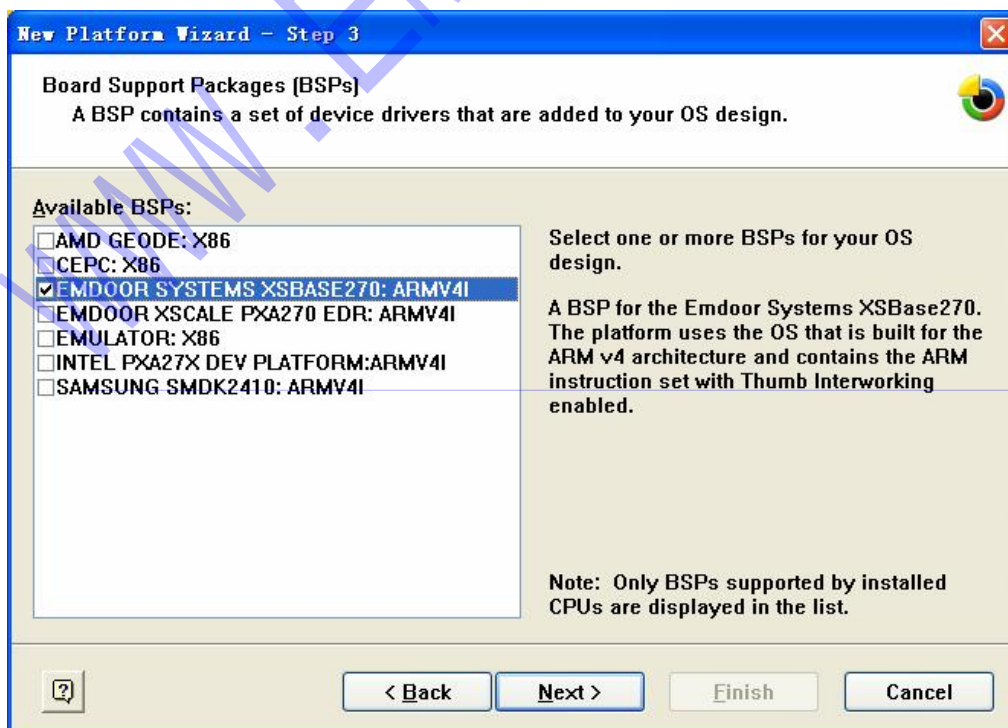


图 1-8 BSP 选择界面

d) 为新建的目标平台选择一个合适的模板配置，如图1-9所示；

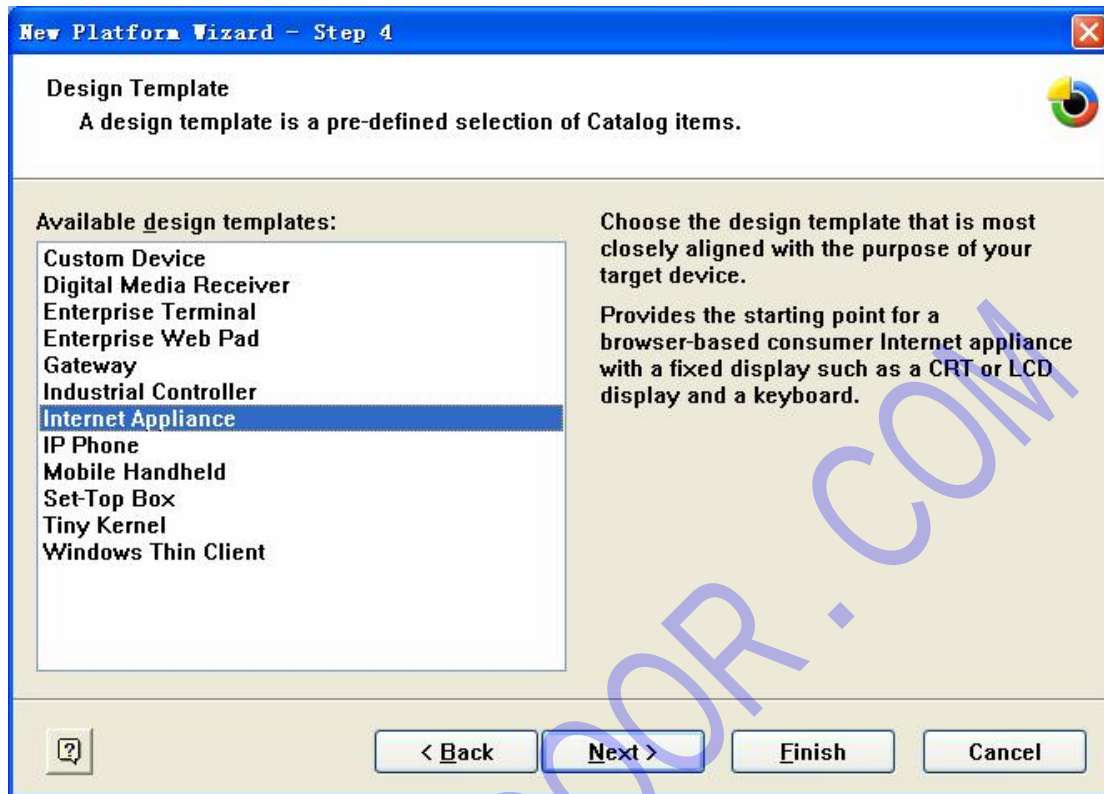


图 1-9 设计模板配置

e) 在 “Application & Media” 选择有关特性（本实验选则默认项）；如图1-10所示

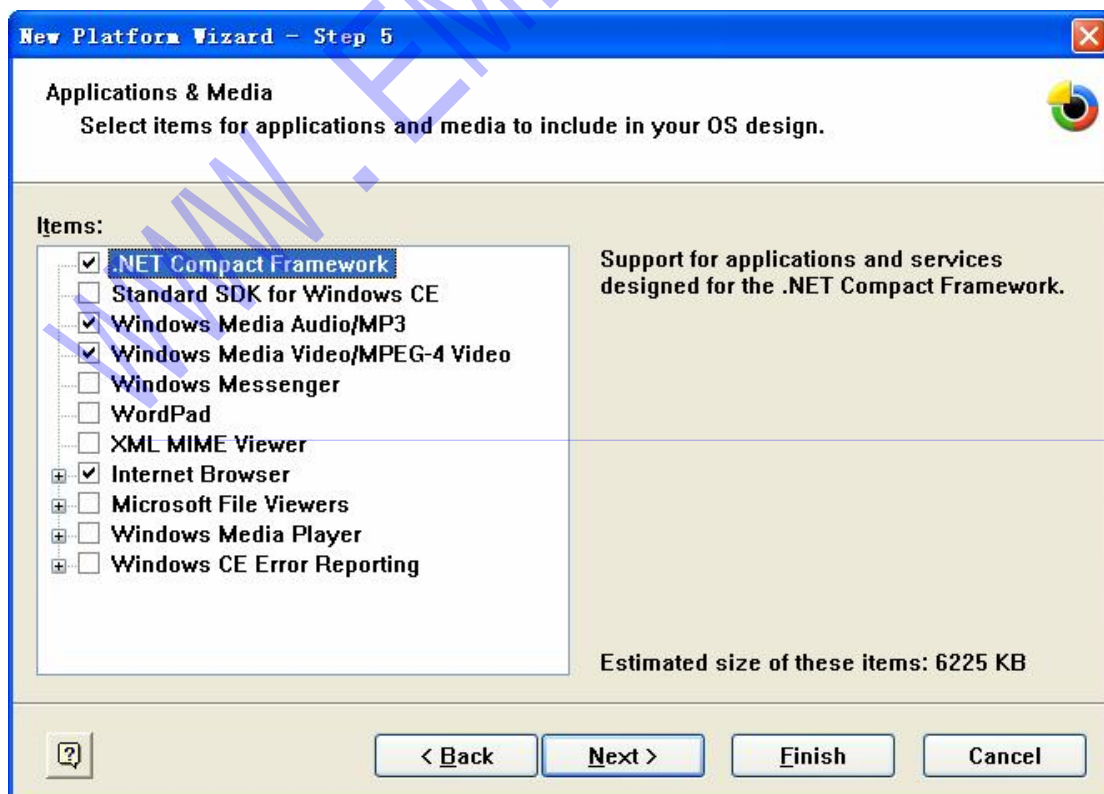


图 1-10 Applications & Media 选择界面

f) 在 “Networking & Communications” 中选择可用特性（默认项）；如图1-11所示

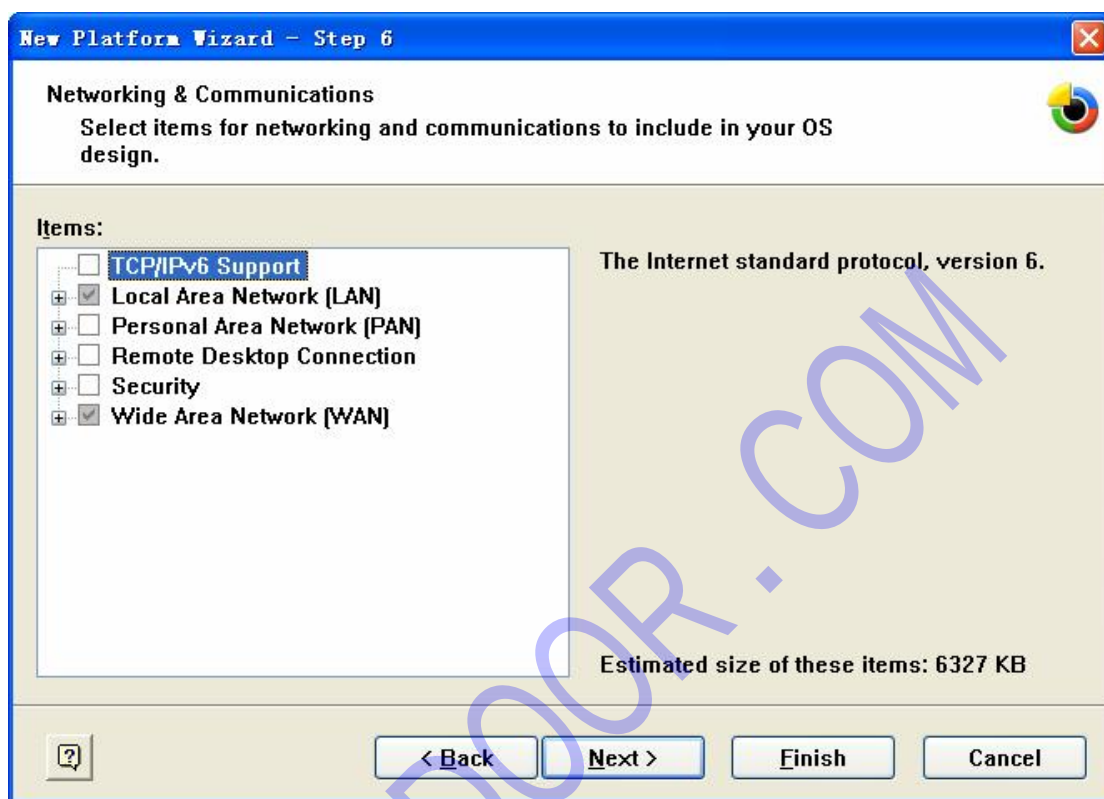


图1-11 Networking & Communications 配置界面

g) 完成 “New Platform Wizard.”，如图 1-12所示



图1-12 平台配置完成界面

第二步：平台的配置

- 选择菜单项“Platform”→“Setting”。
- 在“Platform Settings”对话框中设置“General”选项卡。选中“Release”，如图 1-13 所示

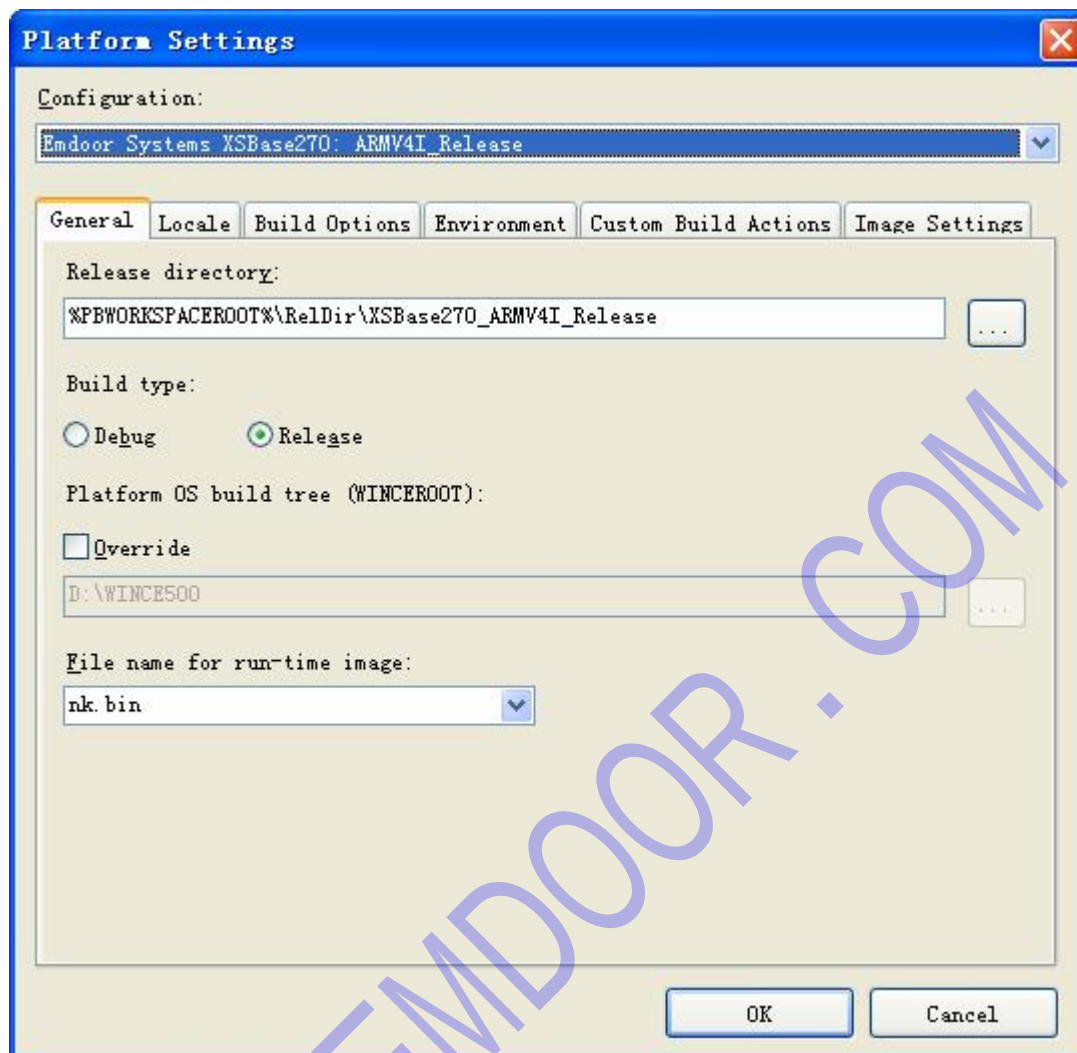


图 1-13 编译类型设置

- c) 在“Platform Settings”对话框中设置“Build Options”选项卡；针对该 XSBASE270 开发板实验平台的选项配置如图 1-14 所示

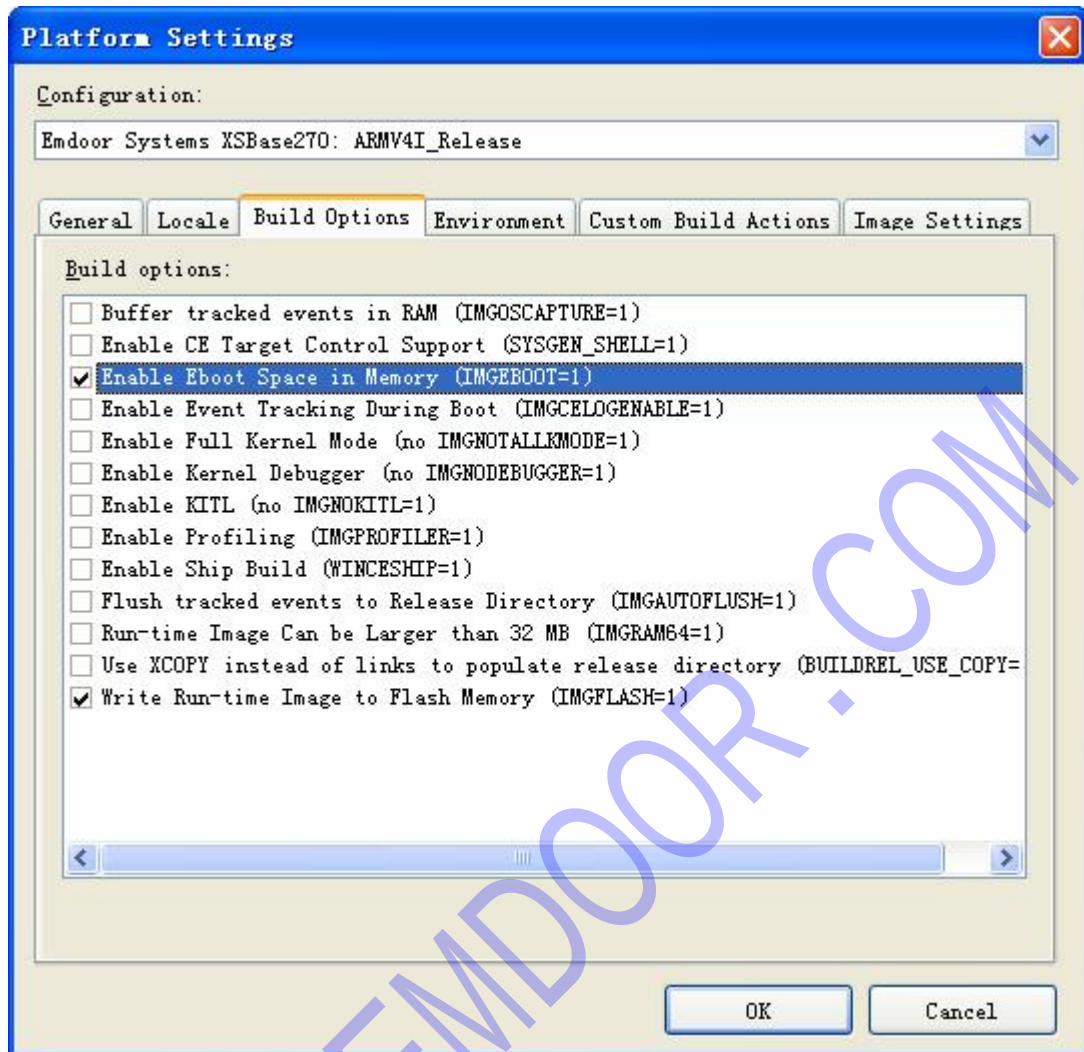


图 1-14 编译选项配置

第三步：配置系统组件和特性

如图 1-15 所示，左边的 workspace 区显示的是所配置的 WinCE 操作系统所具有的组件，右边的 Catalog 区显示的则是 PB 提供的能够加入到 WinCE 操作系统的所有组件。实验者可以在右边的 catalog 区中选择自己需要的设备驱动程序和功能组件。如果决定要添加它到左边的 workspace 区以参加编译，通过右击选项再选择“Add to OS Design”。

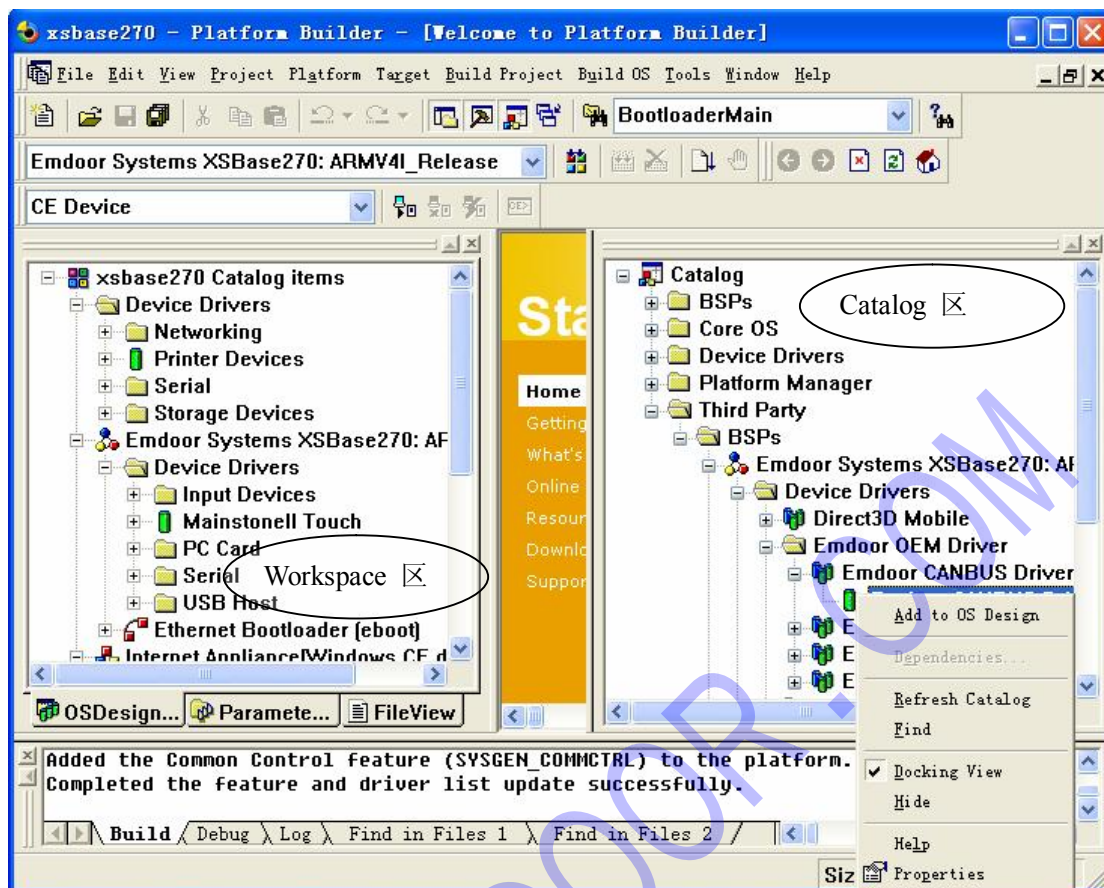


图 1-15 系统组件特性界面

a) 添加 XSBase270 实验平台提供的设备驱动程序

XSBase270 实验平台的驱动程序由平台 BSP 提供，其中包括输入设备（Input Devices）、网络设备（Networking）、PC 卡（PC Card）、SD 卡、串口、触摸屏、USB 设备等；另外还包括 Emdoor OEM 驱动程序（主要包括：CAN 总线驱动、IDE 接口驱动、LED 显示驱动、电机接口驱动等）。如果用户想向新建平台添加设备驱动程序，通过右击选项再选择“Add to OS Design”即可。如图 1-16 所示

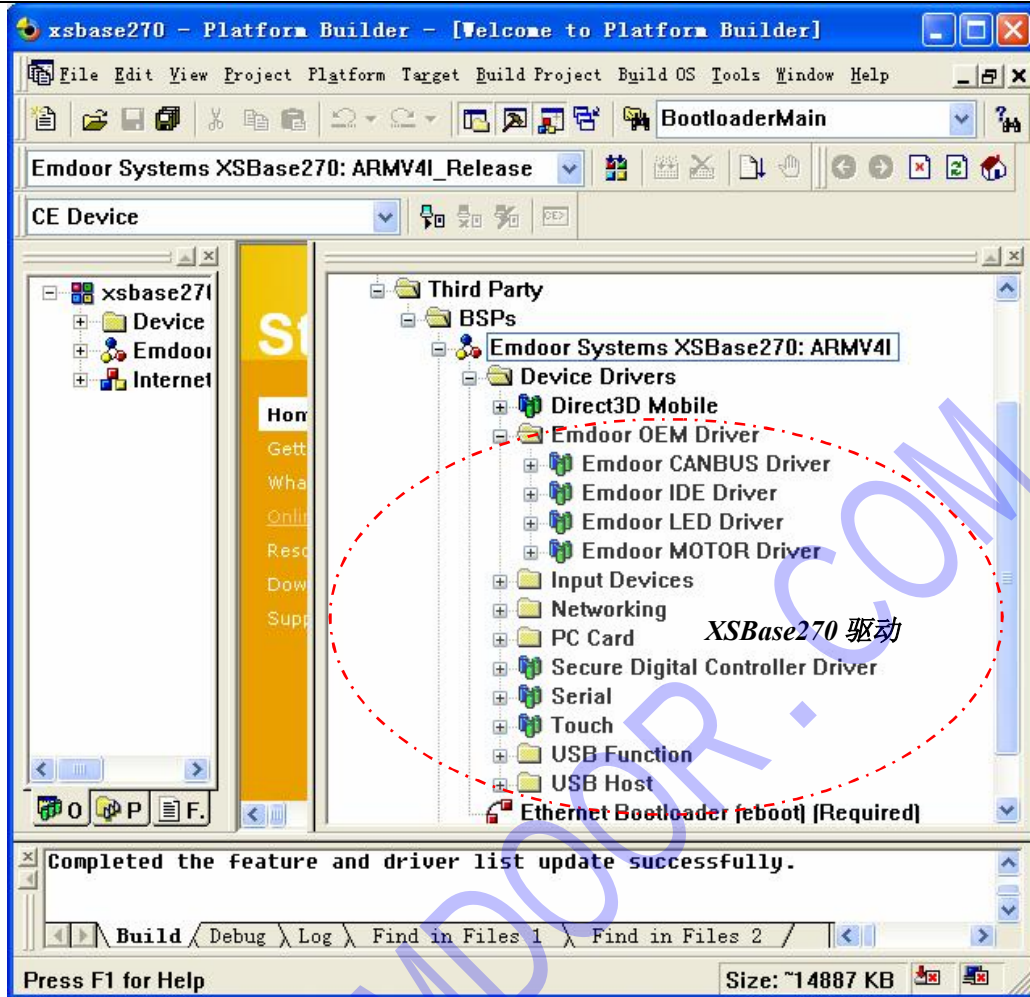


图 1-16 XSBase270 的 BSP 提供的驱动程序

b) 添加系统功能组件

系统的功能组件属于平台无关部分，由 Platform Builder 开发环境统一提供。它们均是在 Core Os 目录下进行添加，如图 1-17 所示。用户可以根据实验的需要添加系统功能组件，本实验中添加以下系统功能组件：

- ① 添加数据同步的组件：Core OS → Windows CE devices → Applications-End User → ActiveSync → File Sync;
- ② 图片浏览器的添加：Core OS → Windows CE devices → Applications-End User → File Viewers → Microsoft Image Viewer
- ③ 大的软键盘组件的添加：Core OS → Windows CE devices → Shell and User Interface → Software Input Panel → Software-based Input Panel (SIP) (Choose 1 or more) → SIP for Large Screens
- ④ 文件系统的添加：Core OS → Windows CE devices → File Systems and Data Store → Storage Manager → FAT File System;
- ⑤ 配置信息保存模块添加：Core OS → Windows CE devices → File Systems and Data Store → Registry Storage (Choose 1) → Hive-based Registry;
- ⑥ CF 卡模块的添加：Catalog → Device Drivers → Storage Devices → Compact Flash / PC Card Storage (ATADISK)
- ⑦ 添加 802.11b 无线 CF 卡：Core OS → Windows CE devices → Communication Services and Networking → Networking - Local Area Network (LAN) → Wireless LAN (802.11)

STA - Automatic Configuration and 802.1x

- ⑧ EVC 调试程序组件模块的添加: Catalog → Platform Manager → Platform Manager

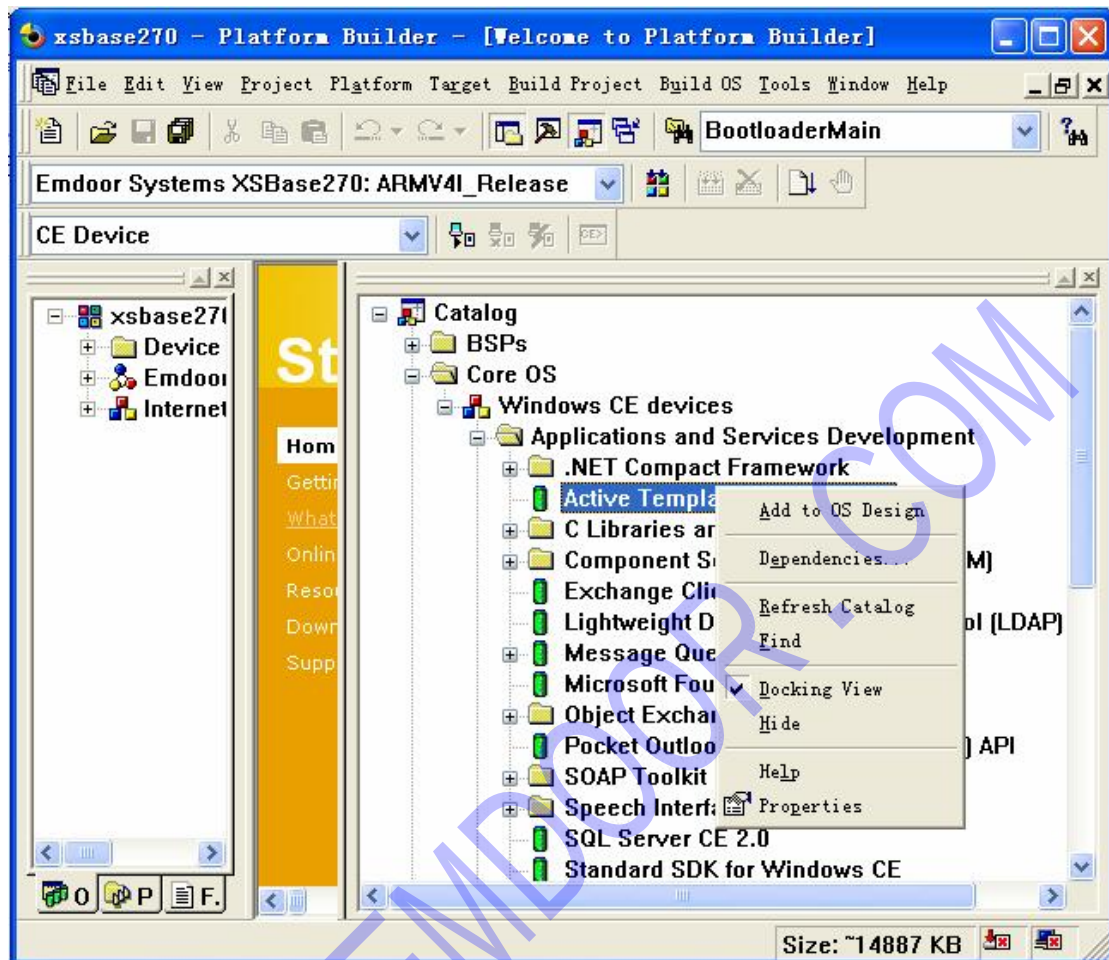


图 1-17 添加系统功能组件

3、平台的编译，构建系统，生成映像文件

选择菜单项 Build OS → Sysgen 对平台进行编译（如图 1-18 所示），如果没出任何错误，编译完成后将生成一个系统映像文件 NK.bin，该文件被放在所建工程所保存的目录下的/ReIDir/工程名_ARMV4I_Release 目录下（本实验的编译后的映像文件 NK.bin 保存在 D:\WINCE500\PBWorkspaces\xsbase270\ReIDir\XSBase270_ARMV4I_Release 目录下）。具体编译链接花费的时间根据你的宿主机的速度和你所配置平台的大小而定。一般需要 20~30 分钟。

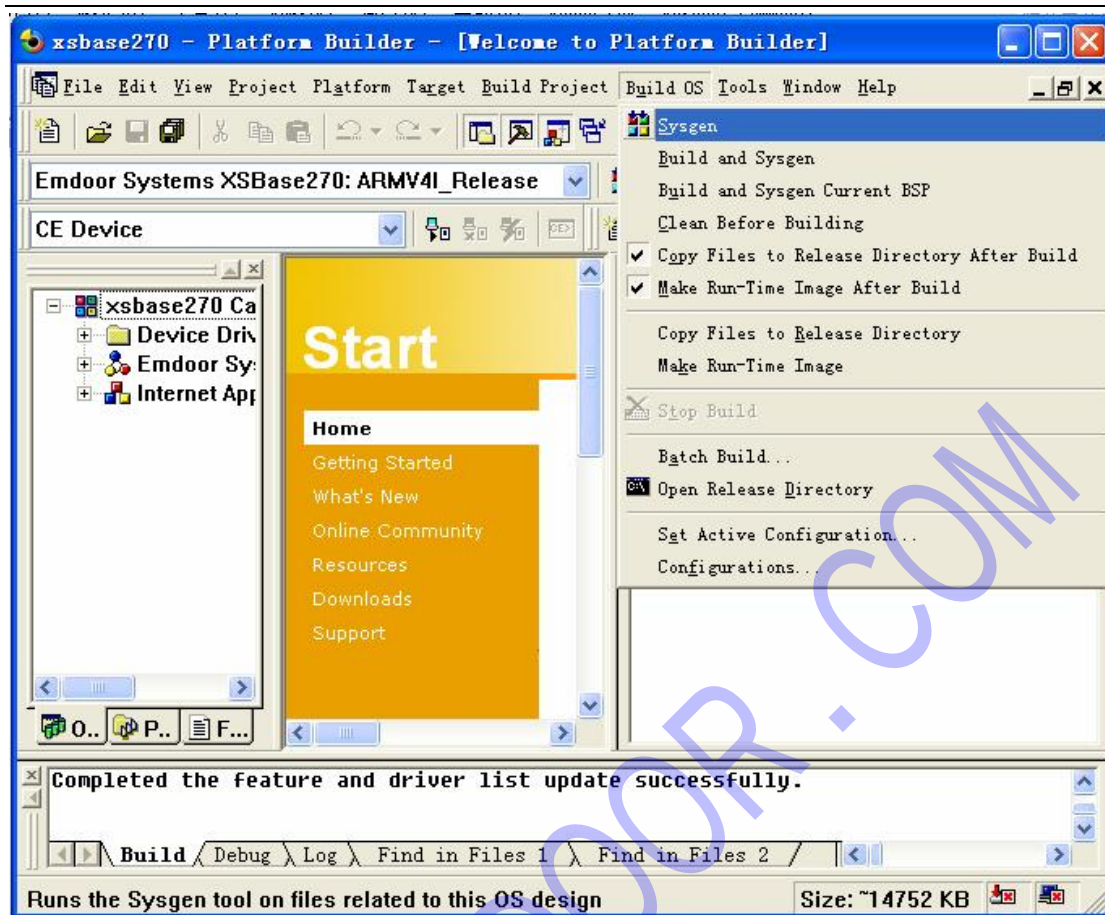


图 1-18 平台的编译

4、代码下载测试

4.1 超级终端的配置 — 超级终端的通信参数设置（如图 1-19 所示）



图 1-19 超级终端设置

启动目标平台，超级终端显示 Boot Loader 配置选项（如图 1-20），如果需要下载新的映像文件，则第三项必须为：3) DHCP: Disable，第五项必须为：5) Download new image at startup

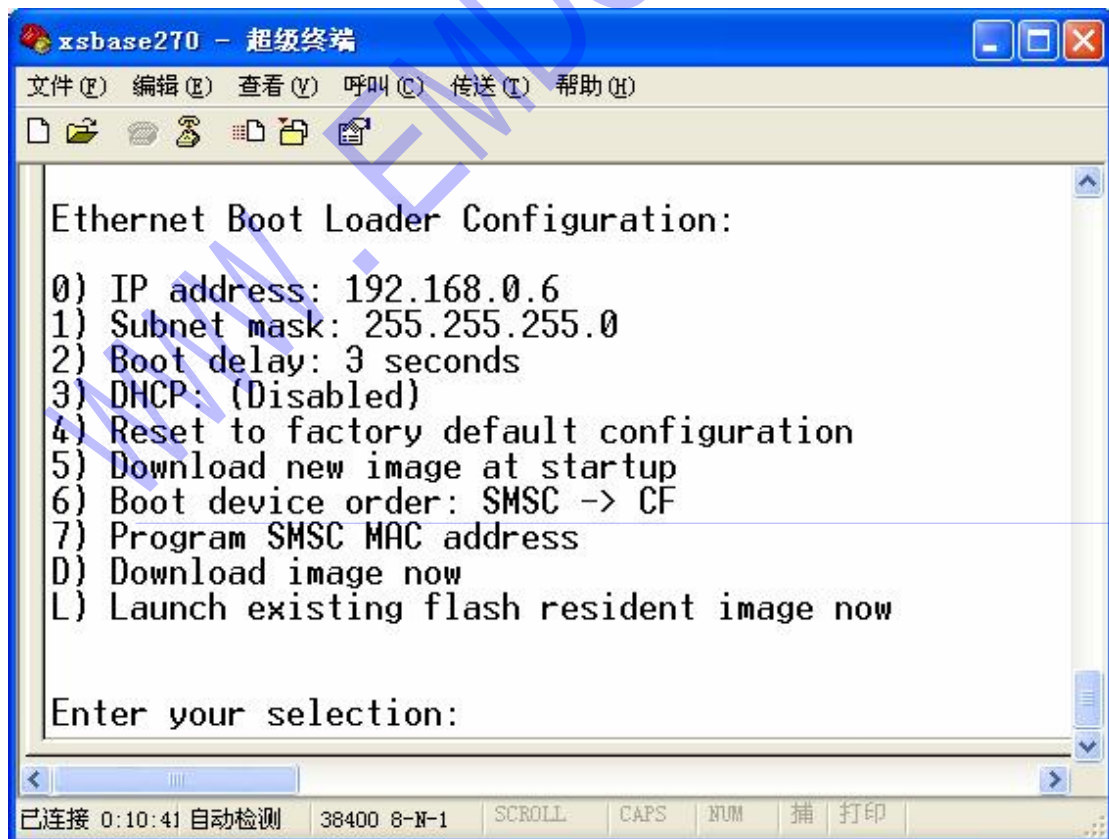


图 1-20 目标板的 Boot Loader 菜单配置选项

4.2 Platform Builder 配置:

选择菜单项 Target → Connectivity Options 进行连接选项配置, 如图 1-21 所示。下载方式采用以太网下载, 单击“Settings”按钮, 对以太网进行配置, 如图 1-22 所示。假如没有接收到设备名: XSBASE2960 的话, 检查网线是否连接好, 可以查看板子上, 以太网端口旁的指示灯是否点亮。然后检查 XSBASE2960 的 IP 是否跟你的 PC 机在同一网段。

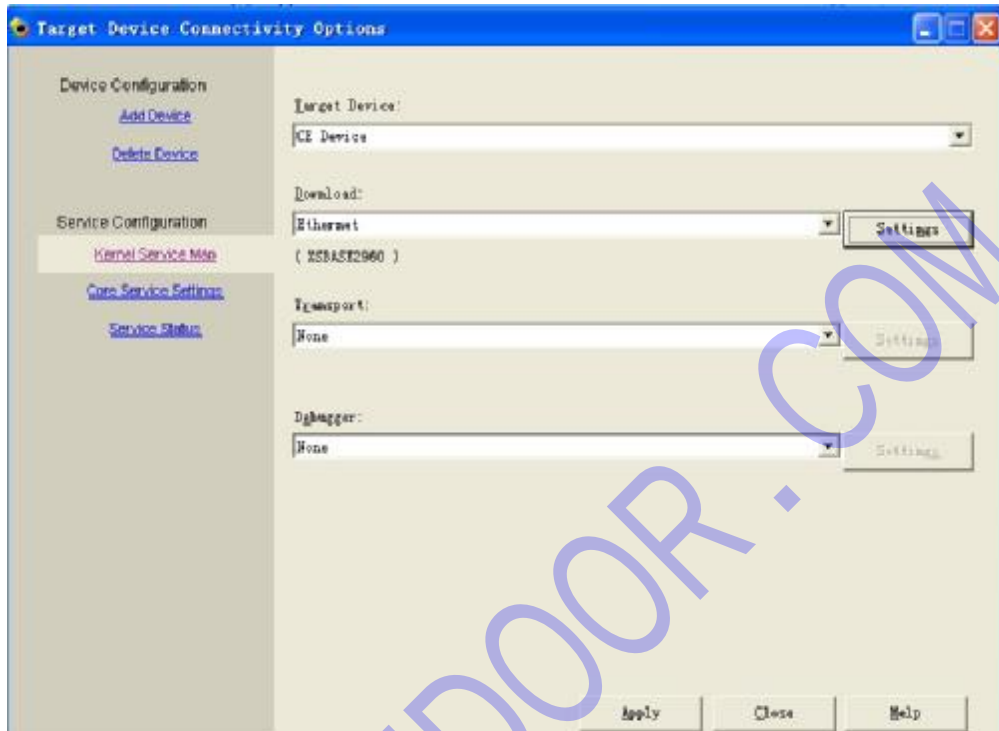


图 1-21 目标板连接配置

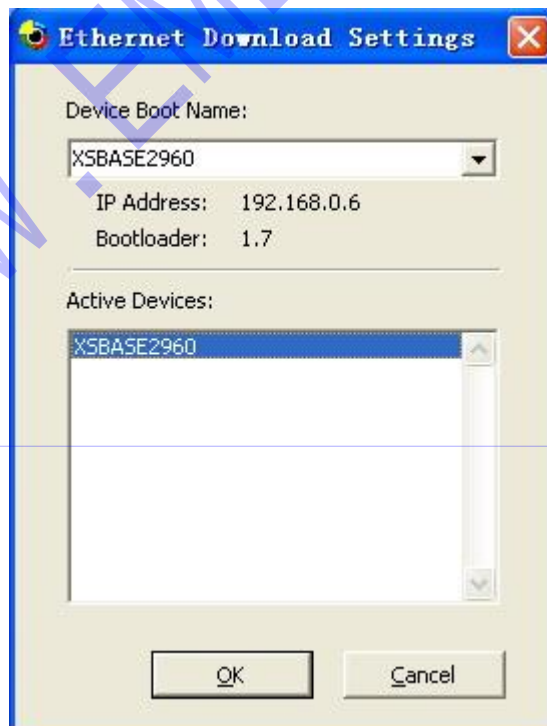


图 1-22 下载以太网配置

4.3 下载测试

选择菜单项 Target → Attach Device, 进行设备连接, 如果设备连接正常, Platform Builder 将编译好的映像文件 NK.bin 通过以太网下载到目标板中, 图 1-23 为 Platform Builder 下载进程。



图 1-23 映像文件下载进程

同时超级终端显示映像文件下载情况, 当超级终端出现下面的字样的时候, 表示映像文件已经下载到目标板的 Flash 存储器中。

```
FlashWrite: Saved end sector(s) First 0x206 (Length=0x1FA).  
FlashErase: Unlocking flash block(s) [0x1, 0x1] (please wait): Done.  
Erasing flash block(s) [0x1, 0x1] (please wait): Done.  
FlashWrite: Restored end sector(s) First 0x206 (Length=0x1FA).  
Writing to flash (please wait): Done.
```

随后断电或按复位键重新启动, 并将超级终端的第 5 项改为 “Launch existing flash resident image at startup.” 然后按 “L” 字母。系统将启动起来。

[习题与思考题]

- 1、如何使用 PB 编译一个 Mobile Handheld 的平台?
- 2、请描述 Platform Builder 的功能;
- 3、如何确定组件已经添加到 PB 中?
- 4、请编译一个在 PC 机仿真 (Emulator) 运行的 Windows CE 系统?

实验二 线程同步调试实验

[实验目的]

- 1、了解 WinCE 下 IO 访问机制和原理；
- 2、掌握 WinCE 线程编程方法；
- 3、掌握线程同步原理和实现线程同步的方法
- 4、熟悉 EVC 和 VS.Net 的使开发环境；

[实验仪器]

- 1、装有 Platform Builder、EVC 和 VS.Net 开发平台的 PC 机一台
- 2、XSBase270 实验开发平台一套

[实验原理]

1、线程概述

WinCE 是有优先级的多任务操作系统，它允许重功能、进程在相同时间的系统中运行，WinCE 支持最大的 32 位同步进程。一个进程包括一个或多个线程，每个线程代表进程的一个独立部分，而一个线程被指定为进程的基本线程。

WinCE 以抢先方式来调度线程。线程以“时间片”为单位来运行，WinCE 的“时间片”通常为 25 毫秒。过来那个时间后，如果线程没有放弃它的时间片，并且线程并不紧急，系统就会挂起线程并调度另一个线程来运行。WinCE 将根据优先级方法来决定要运行的线程，高优先级的线程将在低优先级的线程前面调度。

2、线程 API 函数

2.1 创建线程

WinCE 提供了 CreateThread 函数来创建线程，其声明如下：

```
HANDLE CreateThread(  
    LPSECURITY_ATTRIBUTES lpThreadAttributes, //线程安全指针, 不支持  
    DWORD dwStackSize, //为自己所使用堆栈分配的地址空间大小, 不支持  
    LPTHREAD_START_ROUTINE lpStartAddress, //线程函数地址  
    LPVOID lpParameter, //传入线程函数的参数  
    DWORD dwCreationFlags, //控制线程创建的附加标志  
    LPDWORD lpThreadId //新线程的 ID 值  
);  
WinCE 不支持 lpThreadAttributes 和 dwStackSize 参数，将它们设置成 NULL 和 0
```

即可。lpStartAddress 指向线程函数的地址；lpParameter 被传递到线程中的参数；dwCreationFlags 线程创建参数，可以设置成 0 或 CREATE_SUSPENDED，如果为 0，表示线程立即执行，如果参数为 CREATE_SUSPENDED，则被创建的线程将处于挂起状态，而且必须要调用 ResumeThread 函数将其唤醒。

2.2 挂起和恢复线程

正在运行的线程可以被挂起、暂停执行。同他使用 SuspendThread 函数即可实现以上功能，该函数的声明如下：

```
DWORD SuspendThread(
HANDLE hThread);
```

参数 hThread 代表要挂起线程的句柄。由于 SuspendThread 函数的调用将增加挂起计数，因而在实际调度线程运行之前，对 SuspendThread 函数的多次调用必须与对 ResumeThread 函数的多次调用相匹配。ResumeThread 函数的定义

```
DWORD ResumeThread(
HANDLE hThread);
```

参数 hThread 同样代表要恢复线程的句柄。

3 线程同步

在使用线程时，会经常遇到两个概念，即线程冲突和线程死锁。

线程冲突：如果线程 A 读写数据 G，线程 B 也正在读取数据 G，那么很显然，该操作将导致数据冲突，引起数据混乱。这里需要使用同步技术，以保证线程 A 和线程 B 依次读写数据 G，避免数据冲突。

线程死锁：例如 A 工人为加工 III 零件在等待 B 提供的 I 零件，而 B 正好在等待应由 A 加工提供的 II 零件来装配 I 零件。由于他们之间再没有其他的任何人帮助通信或其他通信手段。所以他们一直在等对方的零件而进入死锁状态。死锁属于逻辑错误，无法通过线程同步来解决。

WinCE 实现线程同步的常用方法：事件（Event）、互斥（Mutex）、信号量（Semaphore）、临界区（CriticalSection）。

3.1 利用事件同步

“事件对象”是实现线程同步最基本的方法之一，一个事件对象可以处于“已标示”和“未标示”两种状态，如果将事件对象设置为“已标示”状态，表示可以执行同步操作，事件对象处于“未标示”状态，则表示需要等待事件对象变为“已标示”状态才可以进行同步操作。下面介绍利用事件同步所需要的 API 函数。

（1）CreateEvent 函数。创建事件对象函数 CreateEvent，其声明如下：

```
HANDLE CreateEvent(
LPSECURITY_ATTRIBUTES lpEventAttributes, //CE 不支持，设为 NULL
BOOL bManualReset, //设置是否手动设置事件对象状态
BOOL bInitialState, //事件对象初始状态
LPTSTR lpName //事件对象名称
);
```

参数 bManualReset 表示是否手动设置事件对象状态，当其值为 TRUE 时，在调用完等待函数（WaitForSingleObject，WaitForMultipleObject）后，则必须调用 ResetEvent 函数，以设置事件对象没有被标示，当其值为 FALSE 时，系统调用完等待函数，会自动将事件对象设置为未标示状态。

参数 `bInitialState` 表示事件对象初始状态，当其值为 `TRUE` 是，事件对象初始化状态为已标示，当其值为 `FALSE` 时，事件对象初始状态为未标示。

如果创建事件函数对象 `CreateEvent` 执行成功，将返回事件对象句柄。若失败，则返回 0，在不用事件句柄时，需要使用 `CloseHandle()` 将其关闭，以释放资源。

(2) `SetEvent` 函数和 `ResetEvent` 函数。函数 `SetEvent()` 的功能是将事件对象设置为已标示状态。该函数的声明如下：

```
BOOL SetEvent (HANDLE hEvent);
```

参数 `hEvent` 表示事件对象句柄。

函数 `ResetEvent` 函数功能将事件对象设置成未标示状态，该函数的声明如下：

```
BOOL ResetEvent (HANDLE hEvent);
```

(3) 使用事件同步的一般使用流程

通常情况，在主线程中，用户利用 `CreateEvent` 函数创建一个事件对象，并且将参数 `bManualReset` 设为 `FALSE`，参数 `bInitialState` 也设为 `FALSE`，此时事件对象状态未标示。然后在线程里通过 `WaitForSingleObject` 函数来等待事件被标示。此时，只要在主线程中调用 `SetEvent` 函数，将事件对象设置成已标示。那么线程里的 `WaitForSingleObject` 函数便会返回，继续执行，同时将事件对象状态设置成未标示。

3.2 利用互斥同步

互斥同步类似于事件对象同步。互斥同步也将创建一个互斥对象，该互斥对象也有“被线程拥有”和“不被线程拥有”两种状态；当互斥对象处于“不被线程拥有”状态，表示可以执行相关操作；当互斥对象处于“被线程拥有”状态，表示此时不可以执行相关操作。通过等待函数请求互斥对象实现同步。

(1) `CreateMutex` 函数。通过 `CreateMutex` 函数创建互斥对象，该函数定义如下：

```
HANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES lpMutexAttributes, //CE 不支持
    BOOL bInitialOwner, //初始化拥有状态，TRUE 表示拥有，FALSE 表示未被拥有
    LPCTSTR lpName //互斥名称
);
```

如果创建互斥函数对象 `CreateMutex` 执行成功，将返回互斥对象句柄。若失败，则返回 `ERROR_INVALID_HANDLE`，在不用互斥句柄时，需要使用 `CloseHandle()` 将其关闭，以释放资源。

(2) `ReleaseMutex` 函数。在使用等待函数请求互斥对象时，如果请求到互斥对象的拥有权，则等待函数将自动设置互斥对象状态为“未被拥有”。`ReleaseMutex` 函数负责释放某个线程对象互斥对象的拥有权，也就是将互斥对象设置为“未被线程拥有”状态。`ReleaseMutex` 函数定义如下：

```
BOOL ReleaseMutex (HANDLE hHandle);
```

`hHandle` 表示互斥对象句柄；

(3) 利用互斥同步的一般使用流程

利用互斥同步的一般使用流程是：首先利用 `CreateMutex` 函数创建互斥对象，并将 `CreateMutex` 中的参数 `bInitialOwner` 设置为 `FALSE`，使互斥对象处于“未被线程拥有”状态。然后利用 `WaitForObject` 等待互斥对象，执行相关操作。处理完成后，利用 `ReleaseMutex` 函数释放线程对互斥对象的拥有权。当所有线程执行完毕后，需要使用 `CloseHandle()` 将其关闭。

3.3 利用临界区同步

“临界区”是进行线程同步的另一种方法，它能够阻止两个或多个不同的线程在同一时间内访问同一个代码区域。它通过调用 `EnterCriticalSection` 函数来指出已经进入代码的

临界区,如果另一线程也调用了 EnterCriticalSection 函数,并且参数指向同一临界区对象,那么另一线程将阻塞,直到第一个线程调用了 LeaveCriticalSection 函数离开临界区为止。临界区同步所需要的 API 函数:

(1) InitializeCriticalSection 函数。如果要使用临界区,首先要使用 InitializeCriticalSection 函数创建临界区,该函数定义如下:

```
void InitializeCriticalSection( LPCRITICAL_SECTION lpCriticalSection );
```

(2) DeleteCriticalSection 函数,当结束使用临界区对象时,必须调用 DeleteCriticalSection 函数释放临界区对象所占有的资源。该函数定义如下:

```
void DeleteCriticalSection( LPCRITICAL_SECTION lpCriticalSection );
```

(3) EnterCriticalSection 函数,在创建了临界区对象后,需要调用 EnterCriticalSection 函数进入临界区,以保护代码,该函数定义如下:

```
void EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

如果临界区对象已经属于另一个线程,那么此函数将阻塞直到另一线程离开临界区才返回。

(4) LeaveCriticalSection 函数。如果要离开临界区,只需要调用 LeaveCriticalSection 函数即可。该函数定义如下:

```
void LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

3.4 利用信号量同步

信号量是建立在互斥基础上,并增加了资源计数的功能。它允许预定数目的线程同时进入要同步的代码。通过设置信号量计数为 1,只允许一个线程同时访问同步代码,而实现线程同步。信号量同步所需要的 API 函数:

(1) CreateSemaphore 函数。在使用信号量实现同步时,需要调用 CreateSemaphore 函数创建信号量对象。该函数定义如下:

```
HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, //CE 不支持
    LONG lInitialCount, //信号量初始化计数值
    LONG lMaximumCount, //信号量计数最大值
    LPCTSTR lpName //信号量对象名称
);
```

如创建信号量成功,函数返回信号量对象句柄,否则返回 NULL 值。

(2) ReleaseSemaphore 函数。在使用等待函数请求信号量时,等待函数自动给信号量计数减 1,那么当计数减到 0 时,信号量对象将不能被请求。ReleaseSemaphore 函数负责给信号量计数加 1,使信号量可以被请求。此函数定义如下:

```
BOOL ReleaseSemaphore(
    HANDLE hSemaphore, //信号量句柄
    LONG lReleaseCount, //信号量计数增加的值
    LPLONG lpPreviousCount //输出量,表示上一次信号量计数
);
```

4、线程同步实验原理

(1) 在 PXA270 目标板上具有 8 个 LED 发光二极管,利用线程函数实现在 LED 点亮左移后便进行点亮右移,并循环进行。其中 LeftShiftThread(LPVOID param) 线程函数完成左移,RightShiftThread(LPVOID param) 线程函数完成右移。由于两线程函数对同一物理地址和同

一输入变量outdata进行操作，所以需要利用线程同步技术进行线程同步。下面为左移线程代码：（右移代码参考源程序）

```

DWORD CThreadSyncDlg::LeftShiftThread(LPVOID param)
{
    CThreadSyncDlg *pDlg=(CThreadSyncDlg*)param;
    if(pDlg->m_CtrlEvent.GetCheck()){//利用事件同步
        while(1)
        {
            WaitForSingleObject(pDlg->m_hSyncEvent,INFINITE);//等待事件标示
            pDlg->outdata =0x01;
            for(int i=0;i<8;i++) {
                *pLightReg=~(pDlg->outdata);
                Sleep(m_ShiftTime);
                pDlg->outdata =(pDlg->outdata )<<1;
                if(pDlg->m_bStop) //停止
                    return 0;
            }
            SetEvent(pDlg->m_hSyncEvent);//标示事件
        }
    }

    if(pDlg->m_CtrlMutex.GetCheck()){//互斥同步
        while(1)
        {
            WaitForSingleObject(pDlg->m_hSyncMutex,INFINITE);//等待互斥
            pDlg->outdata =0x01;
            for(int i=0;i<8;i++)
            {
                *pLightReg=~(pDlg->outdata);
                Sleep(m_ShiftTime);
                pDlg->outdata =(pDlg->outdata )<<1;
                if(pDlg->m_bStop) {
                    ReleaseMutex(pDlg->m_hSyncMutex);
                    return 0;
                }
            }
            ReleaseMutex(pDlg->m_hSyncMutex);//释放互斥对象
        }
    }

    if(pDlg->m_CtrlSemaphore.GetCheck()){//信号量同步
        while(1)
        {
            WaitForSingleObject(pDlg->m_hSynSemaphore,INFINITE);
            pDlg->outdata =0x01;
            for(int i=0;i<8;i++) {
                *pLightReg=~(pDlg->outdata);
                Sleep(m_ShiftTime);
            }
        }
    }
}

```

```

        pDlg->outdata =(pDlg->outdata )<<1;
        if(pDlg->m_bStop) {
            ReleaseSemaphore(pDlg->m_hSynSemaphore,1,NULL);
            return 0;
        }
    }
    ReleaseSemaphore(pDlg->m_hSynSemaphore,1,NULL);
}

if(pDlg->m_CtrlCritical.GetCheck()){//临界区同步
while(1)
{
    EnterCriticalSection(&(pDlg->m_critical_Section));//进入临界区
    pDlg->outdata =0x01;
    for(int i=0;i<8;i++){
        *pLightReg=~(pDlg->outdata);
        Sleep(m_ShiftTime);
        pDlg->outdata =(pDlg->outdata )<<1;
        if(pDlg->m_bStop) {
            LeaveCriticalSection(&(pDlg->m_critical_Section));
            return 0;
        }
    }
    LeaveCriticalSection(&(pDlg->m_critical_Section));//离开临界区
}
}

if(pDlg->m_CtrlNone.GetCheck()){//没有使用同步技术
while(1) {
    pDlg->outdata =0x01;
    for(int i=0;i<8;i++)
    {
        *pLightReg=~(pDlg->outdata);
        Sleep(m_ShiftTime);
        pDlg->outdata =(pDlg->outdata )<<1;
        if(pDlg->m_bStop)
            return 0;
    }
}
return 0;
}
}

```

(2) 线程开启过程

```

void CThreadSyncDlg::OnbtnExecute()
{
    HANDLE m_hLeftThread;

```

```

HANDLE m_hRighthread;
m_bStop=FALSE;
if(m_CtrlEvent.GetCheck())
    ::SetEvent(m_hSyncEvent);
m_hLeftThread=::CreateThread(NULL,0,LeftShiftThread,this,0,NULL);//创建左移线程
m_hRighthread=::CreateThread(NULL,0,RightShiftThread,this,0,NULL); //创建右移线程
if(!m_hLeftThread)
    MessageBox(_T("Create LeftShiftThread Failed"),_T("System
Information"),MB_OK|MB_ICONERROR);

if(!m_hRighthread)
    MessageBox(_T("Create RightShiftThread Failed"),_T("System
Information"),MB_OK|MB_ICONERROR);

CloseHandle(m_hLeftThread);
CloseHandle(m_hRighthread);
}

```

(3) 线程同步量初始化

```

BOOL CThreadSyncDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    SetIcon(m_hIcon, TRUE); // Set big icon
    SetIcon(m_hIcon, FALSE); // Set small icon
    CenterWindow(GetDesktopWindow());// center to the hpc screen
    if(!InitLightReg()){
        MessageBox(L"Init Light Register Failed",L"System
Inform",MB_OK|MB_ICONERROR);
        return FALSE;
    }

    m_CtrlEvent.SetCheck(1);
    m_hSyncEvent=::CreateEvent(NULL,false,false,NULL);//创建事件同步对象
    m_hSyncMutex=::CreateMutex(NULL,false,NULL);//创建互斥同步对象
    m_hSynSemaphore = CreateSemaphore(NULL,1,1,NULL);//创建信号量同步对象
    InitializeCriticalSection(&m_critical_Section); //创建临界区同步对象
    return TRUE;
}

```

(4) 同步变量的释放

```

void CThreadSyncDlg::OnDestroy()
{
    if(m_hSyncEvent)
        CloseHandle(m_hSyncEvent);
    if(m_hSyncMutex)
        CloseHandle(m_hSyncMutex);
}

```

```
if(m_hSynSemaphore)
    CloseHandle(m_hSynSemaphore);
DeleteCriticalSection(&m_critical_Section);
CDialog::OnDestroy();
}
```

[实验内容]

- 1、了解在 Windows CE 下线程编程方法；
- 2、掌握利用线程同步实现线程同步的基本原理；
- 3、掌握线程同步编程；
- 4、掌握 EVC 或 VS.net 编程方法

[实验步骤]

第一步：连接好实验系统，打开实验箱电源。

第二步：利用 Visual Studio 2005.net 打开线程同步调试工程文件 ThreadSync.sln，进行编译；

第三步：编译该代码，点击运行按钮，这样程序就会下载到 XSBase270 目标板板上运行。

运行界面如图 2-1 所示。



图 2-1 线程同步调试实验运行界面

第四步：线程调试实验程序操作过程

在分别选中事件同步（Event）、互斥同步（Mutex）、信号量同步（Semaphore）、临界区同步（CriticalSection）后，按下“Execute”按钮（Execute），观察 LED 发光二极管的点亮现象；按下“Stop”按钮使 LED 发光二极管停止移动。

选中没有利用线程同步方法“None”，按下“Execute”按钮，观察 LED 发光二极管的点亮现象；

[习题与思考题]

- 1、如果在线程运行过程中，需要采用事件的方法停止线程的运行，该怎样修改程序？
- 2、如果只需要每次完成发光二极管一次左移和右移操作，随即自动停止，怎样修改源程序？
- 3、在编写线程程序时，需要注意哪些问题？

WWW.EMDOOR.COM

实验三 驱动程序结构实验

[实验目的]

- 1、了解驱动程序的原理和功能；
- 2、掌握流式接口驱动程序的结构；
- 3、掌握编写流式接口的驱动程序的方法
- 4、熟悉 EVC 和 VS.Net 的开发环境；

[实验仪器]

- 1、装有 Platform Builder、EVC 和 VS.Net 开发平台的 PC 机一台
- 2、XSBase270 实验开发平台一套

[实验原理]

1、流式接口驱动程序简介

驱动程序是对底层硬件的抽象。应用程序开发者不需要真正理解底层驱动的工作原理，他们只需要通过 Windows CE 提供的 API 函数，就可以直接与硬件进行交互。

WinCE 的流式接口驱动程序以动态链接库的形式存在，由设备管理器（通常是 device.exe 或者 gwes.exe）统一加载、管理和卸载。与具有单独目的的内部设备驱动程序相比，所有流式接口驱动程序都是用同一接口并调用同一个函数集。

每个流式接口驱动程序必须实现一组标准的函数，用来完成标准的文件 I/O 函数和电源管理函数，这些函数提供给 WinCE 操作系统的内核使用。这些函数通常叫做流式接口驱动程序的 DLL 接口，如表 3-1 所示：

表 3-1 流式接口驱动程序要实现的 DLL 接口：

函数名称	描述
XXX_Close	在驱动程序关闭时应用程序通过 CloseHandle 函数调用这个函数
XXX_Deinit	当设备管理器卸载一个驱动程序时调用这个函数
XXX_Init	当设备管理器初始化一个具体设备时调用这个函数。
XXX_IOControl	上层的软件通过 DeviceIoControl 函数可以调用这个函数
XXX_Open	在打开一个设备驱动程序时应用程序通过 CreateFile 函数调用这个函数
XXX_PowerDown	在系统调用前调用这个函数
XXX_PowerUp	在系统从新启动前调用这个函数

XXX_Read	在一个设备驱动程序处于打开状态时由应用程序通过 ReadFile 函数调用
XXX_Seek	对设备的数据指针进行操作，由应用程序通过 SetFilePointer 函数调用
XXX_Write	在一个设备驱动程序处于打开状态时由应用程序通过 WriteFile 函数调用。

在实际开发中接口名称中的 XXX 三个字母由设备驱动的设备名前缀代替，例如，如果一个流式接口驱动程序的设备文件名前缀为“STR”，那么它相应要实现的 DLL 接口为 STR_Close, STR_Deinit, STR_Init, STR_Read 等。

下面介绍几个主要的流式接口驱动接口函数：

(1) DWORD XXX_Open (DWORD hDeviceContext,
DWORD AccessCode,
DWORD ShareMode)

参数：DWORD hDeviceContext，设备驱动的句柄，由 XXX_Init 函数创建式返回；
DWORD AccessCode，传给驱动程序使用的地址，这个地址跟读和写有关；
DWORD ShareMode，共享模式，

返回值：返回驱动程序引用实例句柄。

描述：这个函数用于打开一个设备驱动程序，当应用程序准备对某个设备进行读或写操作时，系统必须先执行 CreateFile() 这个函数用于打开这个设备，这个函数执行后才能执行读和写操作；

(2) BOOL XXX_Close (DWORD hOpenContext)

参数：DWORD hOpenContext，设备驱动的句柄，由 XXX_Open 创建。

返回值：调用成功返回 TRUE，失败返回 FALSE

描述：这个函数用于关闭一个驱动程序的引用实例，应用程序通过 CloseHandle () 函数来调用这个函数，当执行完这个函数后，hOpenContext 将不再有效；

(3) DWORD XXX_Read (DWORD hOpenContext,
LPVOID pBuffer,
DWORD Count)

参数：DWORD hOpenContext，由 CreateFile () 函数返回的句柄；
LPVOID pBuffer，一个缓冲区地址，用于从驱动读数据；
DWORD Count，需要读缓冲区的长度

返回值：实际读缓冲区的长度

描述：当一个流式接口驱动程序已经打开后，可以使用 ReadFile () 函数对这个设备进行读操作。ReadFile () 中的 hFile 参数对应设备的引用实例句柄 hOpenContext，二参数 lpBuffer 将传给 pBuffer，用于表示要读缓冲区的地址，参数 nNumberOfByteToRead 将传给 Count，表示读取缓冲区的长度。

(4) DWORD XXX_Write (DWORD hOpenContext,
LPVOID pBuffer,
DWORD Count)

参数：DWORD hOpenContext，由 CreateFile () 函数返回的句柄；
LPVOID pBuffer，一个缓冲区地址，用于从驱动写数据；
DWORD Count，需要写缓冲区的长度

返回值：实际写缓冲区的长度

描述：当一个流式接口驱动程序已经打开后，可以使用 WriteFile () 函数对这个设备进行写

操作。

```
(5) BOOL XXX_IOCTL (DWORD hOpenContext,
                    WORD dwCode,
                    PBYTE pBufIn,
                    DWORD dwLenIn,
                    PBYTE pBufOut,
                    DWORD dwLenOut,
                    PDWORD pdwActualOut)
```

参数: DWORD hOpenContext, 由 CreateFile () 函数返回的句柄;

WORD dwCode, 用于描述这次 IOCTL 操作的语义, 由用户自己定义;

PBYTE pBufIn, 缓冲区指针指向需要传送给驱动程序使用的数据;

DWORD dwLenIn, 要传送给驱动程序使用数据的长度;

PBYTE pBufOu, 缓冲区指针指向驱动程序传给应用程序使用的数据;

DWORD dwLenOut, 要传送给应用程序使用数据的长度;

PDWORD pdwActualOut, 返回实际处理数据的长度;

返回值: 调用成功返回 TRUE, 失败返回 FALSE

描述: 这个函数通常用于向设备发送一个命令, 应用程序使用 DeviceIOControl 函数来通知操作系统调用这个函数, 通过参数 dwCode 来通知驱动程序要执行的操作。这个函数扩展了流式接口驱动程序的功能。

```
(6) DWORD XXX_Init (DWORD dwContext)
```

参数: DWORD dwContext, 指向字符串的指针, 通常这个参数都为一个流式接口驱动在注册表内的设置。

返回值: 如果调用成功返回一个驱动程序的句柄;

描述: 当用户使用一个设备的时候, 设备管理器调用这个函数来对设备进行初始化。这个函数并不是由应用程序调用的, 而是通过设备管理器提供的 ActiveDeviceEx () 来调用。函数执行成功后返回一个设备的句柄。

2、流式接口驱动程序的实现

由于在 WinCE 中流式接口驱动程序以 DLL 的形式存在, 是运行在用户模式的动态链接库, 所以既可以使用微软提供的 EVC 或 VS.net 编写流式接口驱动程序, 也可以使用 Platform Builder (PB) 来进行编写。为了方便调试, 通常采用 PB 来开发流式驱动程序。下面以在 X86 仿真平台下为例介绍采用 PB 编写和调试流式接口设备的方法和步骤。

2.1 流式驱动程序的创建步骤:

- (1) 打开 Platform Builder。在 Platform Builder 中选择 “File” -> “New Project or File”, 创建一个 “Windows CE Dynamic link library” 项目, 项目的名称填写 “My Driver” (如图 3-1 所示)

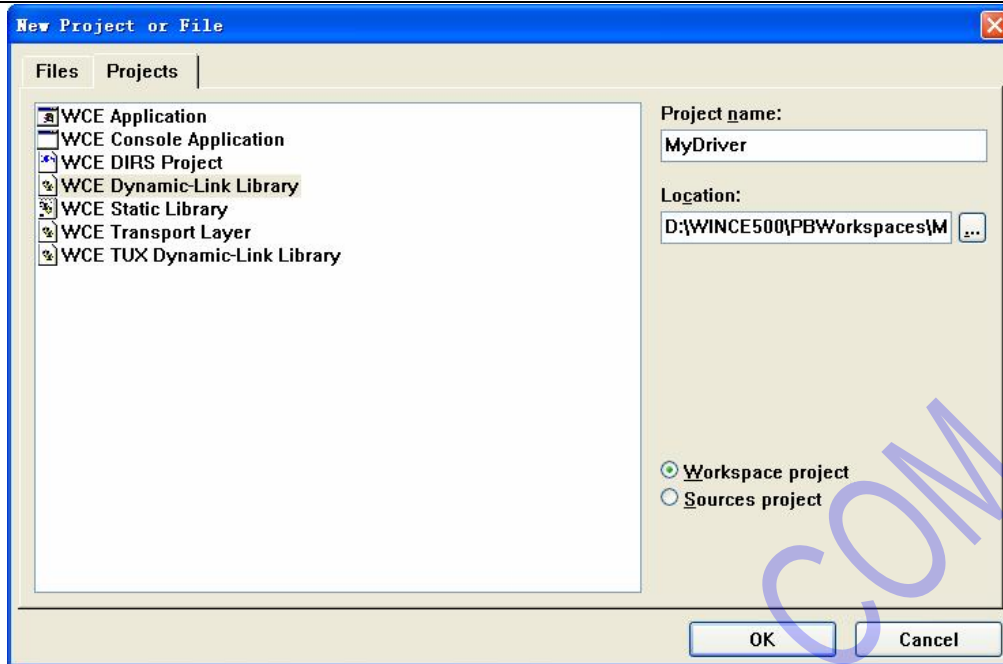


图 3-1 WinCE 驱动创建项目界面

- (2) 按 OK 按钮，在 DLL 的类型界面中（如图 3-2 所示）选中 A Simple Windows CE DLL projects，Platform Builder 将生成 DLL 框架代码。

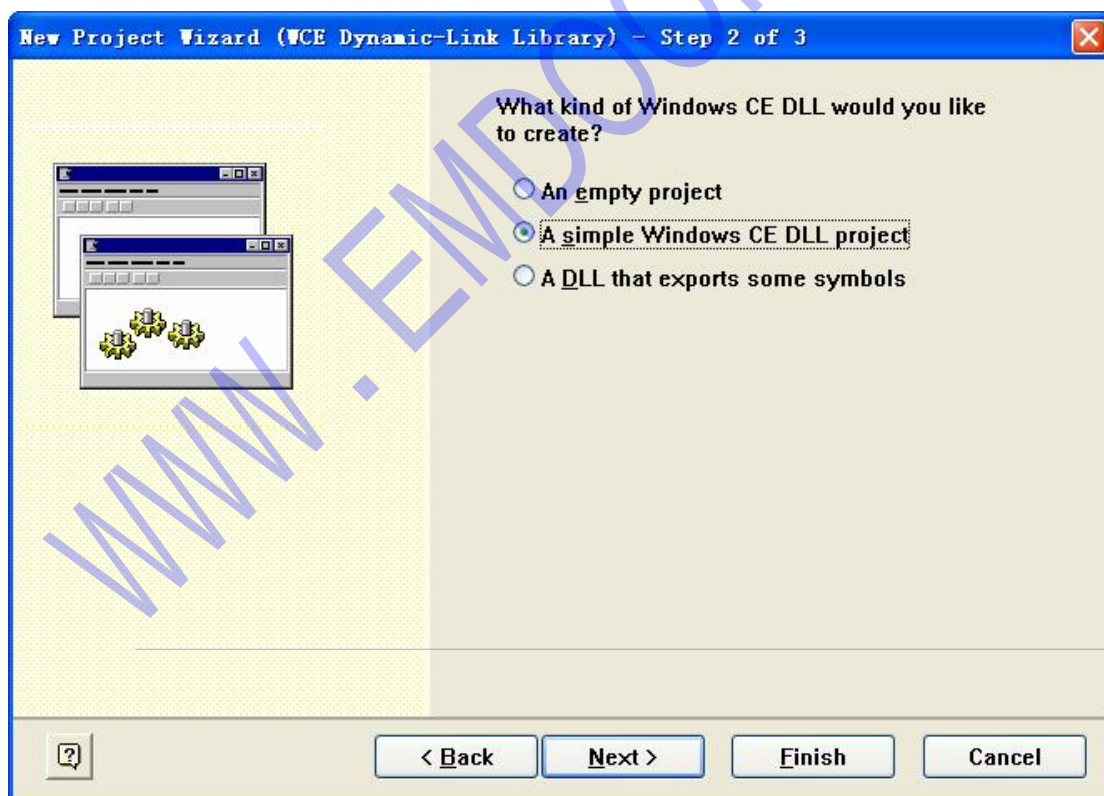


图 3-2 驱动程序的 DLL 类型选择

- (3) 修改 Platform Builder 生成的DllMain 函数，如下：

```
BOOL WINAPI DllMain( HANDLE hModule,
                    DWORD  ul_reason_for_call,
```

```

        LPVOID lpReserved )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        RETAILMSG (1, (_T("MyDriver Demo: DLL_PROCESS_ATTACH \n")));
        return TRUE;
    case DLL_THREAD_ATTACH:
        RETAILMSG (1, (_T("MyDriver Demo: DLL_THREAD_ATTACH \n")));
        break;
    case DLL_THREAD_DETACH:
        RETAILMSG (1, (_T("MyDriver Demo: DLL_THREAD_DETACH \n")));
        break;
    case DLL_PROCESS_DETACH:
        RETAILMSG (1, (_T("MyDriver Demo: DLL_PROCESS_DETACH \n")));
        break;
    }
    return TRUE;
}

```

(4) 添加导出函数的具体实现，代码如下：

```

DWORD STR_Init( DWORD dwContext )
{
    DWORD dwRet = 0;
    RETAILMSG ( 1, (_T("MyDriver Demo: STR_Init\n")) );
    //Init the driver
    memset ( achBuffer, 0, BUFSIZE*sizeof ( WCHAR ) );
    //Return a nonzero number
    dwRet = 1;
    return dwRet;
}

BOOL STR_Deinit ( DWORD hDeviceContext )
{
    BOOL bRet = TRUE;
    RETAILMSG ( 1, (_T("MyDriver Demo: STR_Deinit\n")) );
    return bRet;
}

DWORD STR_Open( DWORD hDeviceContext, DWORD AccessCode, DWORD ShareMode )
{
    DWORD dwRet = 0;
    RETAILMSG ( 1, (_T("MyDriver Demo: STR_Open\n")) );
    //Return a nonnull handle
    dwRet = 1;
}

```

```

        return dwRet;
    }

    BOOL STR_Close ( DWORD hOpenContext )
    {
        BOOL bRet = TRUE;
        RETAILMSG ( 1, ( _T("MyDriver Demo: STR_Close\n") ) );
        return bRet;
    }

    BOOL STR_IOControl ( DWORD hOpenContext,
                        DWORD dwCode,
                        PBYTE pBufIn,
                        DWORD dwLenIn,
                        PBYTE pBufOut,
                        DWORD dwLenOut,
                        PDWORD pdwActualOut)
    {
        BOOL bRet = TRUE;
        RETAILMSG ( 1, ( _T("MyDriver Demo: STR_IoControl\n") ) );
        return bRet;
    }

    void STR_PowerDown ( DWORD hDeviceContext )
    {
        RETAILMSG ( 1, ( _T("MyDriver Demo: STR_PowerDown\n") ) );
    }

    void STR_PowerUp ( DWORD hDeviceContext )
    {
        RETAILMSG ( 1, ( _T("MyDriver Demo: STR_PowerUp\n") ) );
    }

    DWORD STR_Read( DWORD hOpenContext, LPVOID pBuffer, DWORD Count )
    {
        DWORD dwRet = 0;
        RETAILMSG ( 1, ( _T("MyDriver Demo: STR_Read\n") ) );
        //Get the size of the buffer
        DWORD cbBuffer = wcslen ( achBuffer );
        dwRet = min ( cbBuffer, Count );
        wcsncpy ( ( LPWSTR ) pBuffer, achBuffer, dwRet);
        //Return the number of the actual bytes
        return dwRet;
    }

```

```

DWORD STR_Seek( DWORD hOpenContext, long Amount, DWORD Type )
{
    DWORD dwRet = 0;
    RETAILMSG ( 1, ( _T("MyDriver Demo: STR_Seek\n") ) );
    return dwRet;
}

DWORD STR_Write(  DWORD hOpenContext,  LPVOID pSourceBuffer,  DWORD
NumberOfBytes )
{
    DWORD dwRet = 0;
    RETAILMSG ( 1, ( _T("MyDriver Demo: STR_Write\n") ) );
    //Get the size of the data to write
    dwRet = min ( BUFSIZE, NumberOfBytes );
    wcsncpy ( achBuffer, ( LPWSTR )pSourceBuffer, dwRet);
    //Return the number of the actual bytes
    return dwRet;
}

```

- (5) 添加导出函数的定义。在 Platform Builder 菜单中，选择 File->New Project or File -> File Tab ->Text File，文件名叫 “MyDriver.def”，内容如下：

```

LIBRARY MyDriver

```

```

EXPORTS

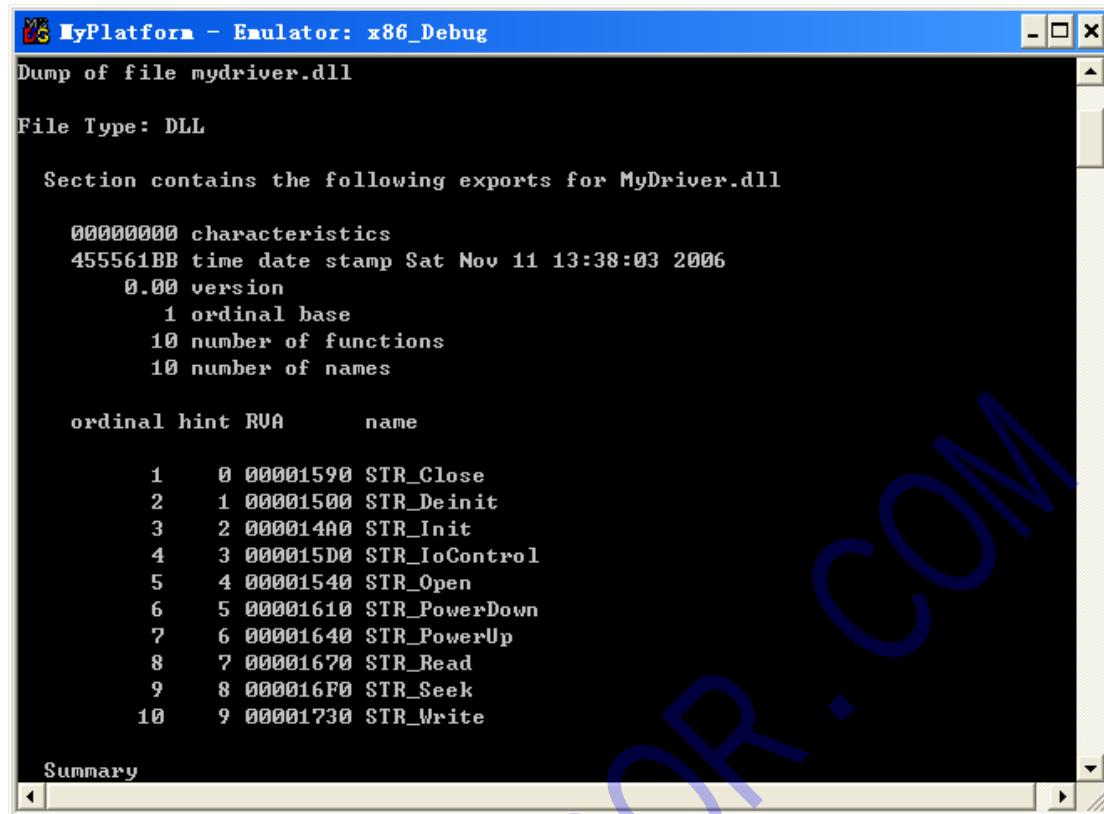
```

```

    STR_Init
    STR_Deinit
    STR_Open
    STR_Close
    STR_IOControl
    STR_PowerUp
    STR_PowerDown
    STR_Read
    STR_Write
    STR_Seek

```

- (6) 编译驱动程序项目。结束后选择 “Build” -> “Open Build Release Directory”，然后输入命令：dumpbin /exports mydriver.dll，确保输出结果如图 3-3，表示 DLL 函数被正确导出。



(7) 在 PB 的 Parameter View 中打开 project.reg, 在文件中添加如下部分:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Sample]
"Dll" = "mydriver.Dll"
"Prefix" = "STR"
"Index" = dword:1
"Order" = dword:0
"FriendlyName" = "Demo Driver"
"Ioctl" = dword:0
```

这段注册表信息告诉系统, 在系统启动的时候把 mydriver.dll 加载到 device.exe, 驱动的前缀是 STR, 下标索引是 1, 这样, 我们就可以使用 CreateFile(L"STR1".....)这样的方式来访问驱动程序了。

2.2 驱动程序的加载测试

(1) 在 Platform Builder 里面选择 "Target" -> "Attach Device", 观察系统启动的时候, 系统加载驱动程序时 Platform Builder 的 Output 窗口的输出如图 4-4 所示:

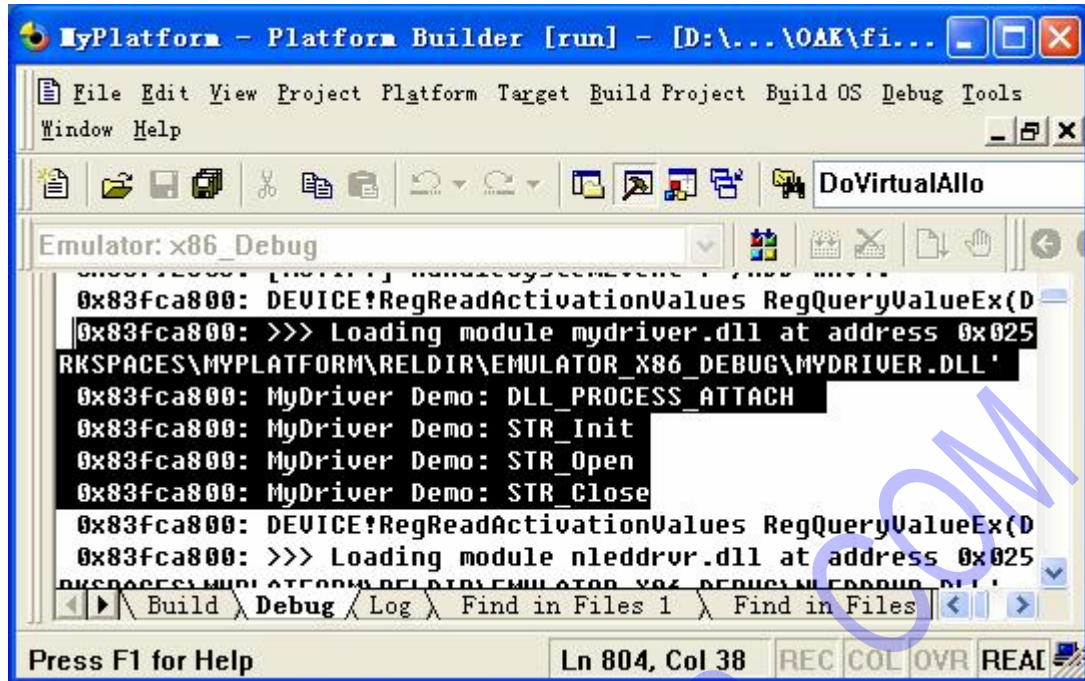


图 4-4 驱动程序加载时的输出图

这说明在系统启动的时候，系统已经根据设置的注册表项，找到了 MyDriver.dll 并且成功加载。为了进一步验证，我们可以利用几个远程工具进一步验证。

(2) 打开 Remote Process Viewer，定位到 device.exe，从图 4-5 我们可以看到，mydriver.dll 已经在 device.exe 的模块列表中了。这说明 mydriver.dll 已经被作为一个驱动加载了。

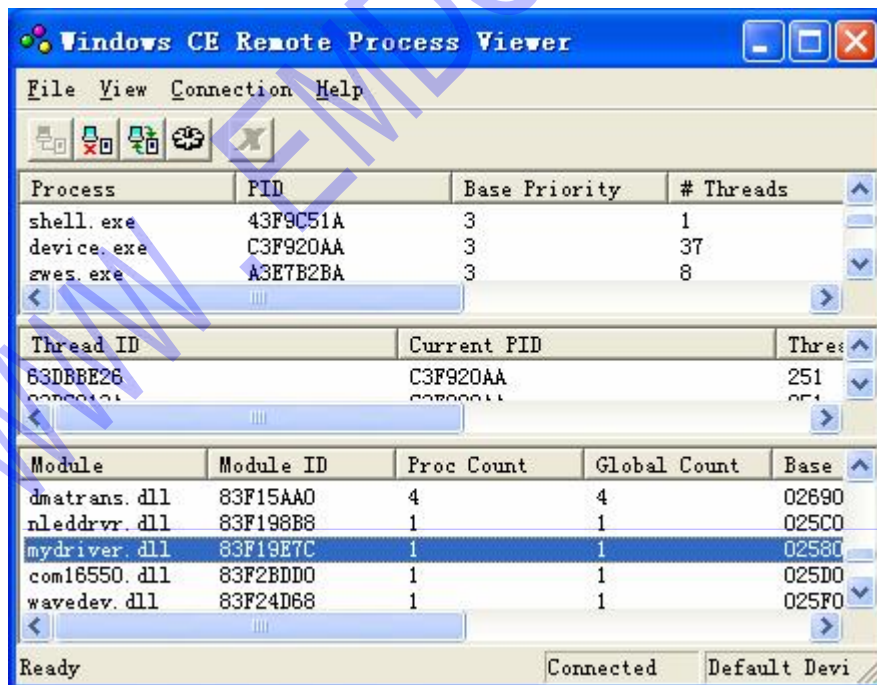


图 4-5 mydriver.dll 在 Device.exe 中加载情况

(3) 打开 Remote Registry Editor，定位到 HKEY_LOCAL_MACHINE\Drivers\Active，寻找所有的已加载驱动，我们可以在找到某个目录下包含下图的信息，这也说明 Driver 已经成功加载。

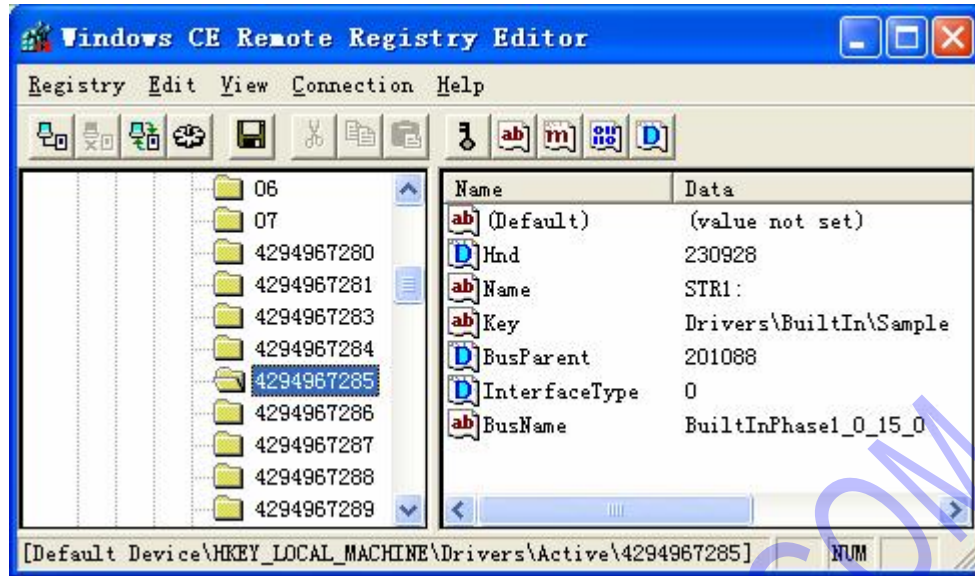


图 4-6 Remote Registry Editor 工具显示成功 mydriver.dll 加载

2.3 驱动程序的应用测试

(1) 利用 VS.net 编写一个驱动程序的应用测试程序 DriverApp, 操作的函数, 如下:

```
void CDriverAppDlg::OnBnClickedbtnopen()
{
    // Open the stings driver
    //-----
    hStr = CreateFile ( TEXT ( "STR1:" ), GENERIC_READ | GENERIC_WRITE, 0 , NULL,
    OPEN_EXISTING, 0 , 0);
    if ( INVALID_HANDLE_VALUE == hStr )
        ::MessageBox( NULL, _T("Cannot open STR1:"), _T( "StringApp" ), MB_OK);
    else
        ::MessageBox( NULL, _T("Open STR1:Succeed"), _T( "StringApp" ), MB_OK);
}

void CDriverAppDlg::OnBnClickedbtnread()
{
    // Read a string from the driver
    //-----
    WCHAR wch[BUFFER_SIZE];
    DWORD dwBytesRead = BUFFER_SIZE;
    memset(&wch, '\0', BUFFER_SIZE*sizeof( WCHAR ) );
    ::ReadFile(hStr, wch, sizeof( wch ), &dwBytesRead, NULL);
    ::MessageBox( NULL, wch, TEXT( "StringApp" ), MB_OK);
}

void CDriverAppDlg::OnBnClickedbtnwrite()
{
    // Write a string to the driver
    //-----
    DWORD dwWritten = 0;
    WCHAR* pString = TEXT ( "This is a test of the String Driver" );
    ::WriteFile (hStr, pString, (_tcslen (pString) + 1), &dwWritten, NULL);
}

void CDriverAppDlg::OnBnClickedbtnciocontrol()
```

```
{
    ::DeviceIoControl (hStr, NULL,  NULL, NULL, NULL, NULL, NULL, NULL);
}
void CDriverAppDlg::OnBnClickedbtnclose()
{
    // Disconnect from the driver
    //-----
    if ( hStr != NULL )
    {
        CloseHandle ( hStr );
        hStr = NULL;
    }
}
```

(2) 驱动程序测试程序运行界面

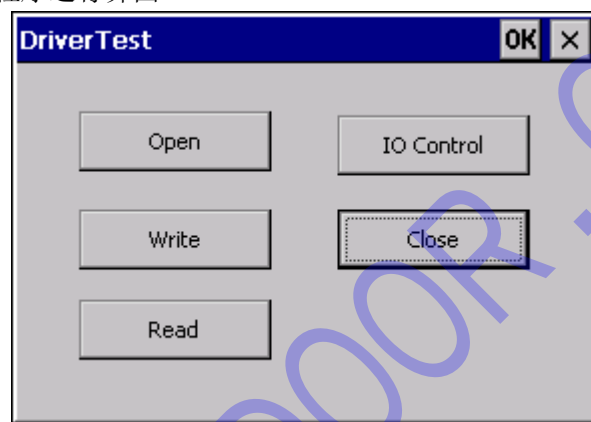


图 4-7 驱动程序测试运行界面

(3) 应用程序对驱动程序进行操作时 PB 调试信息,如图 4-8 所示

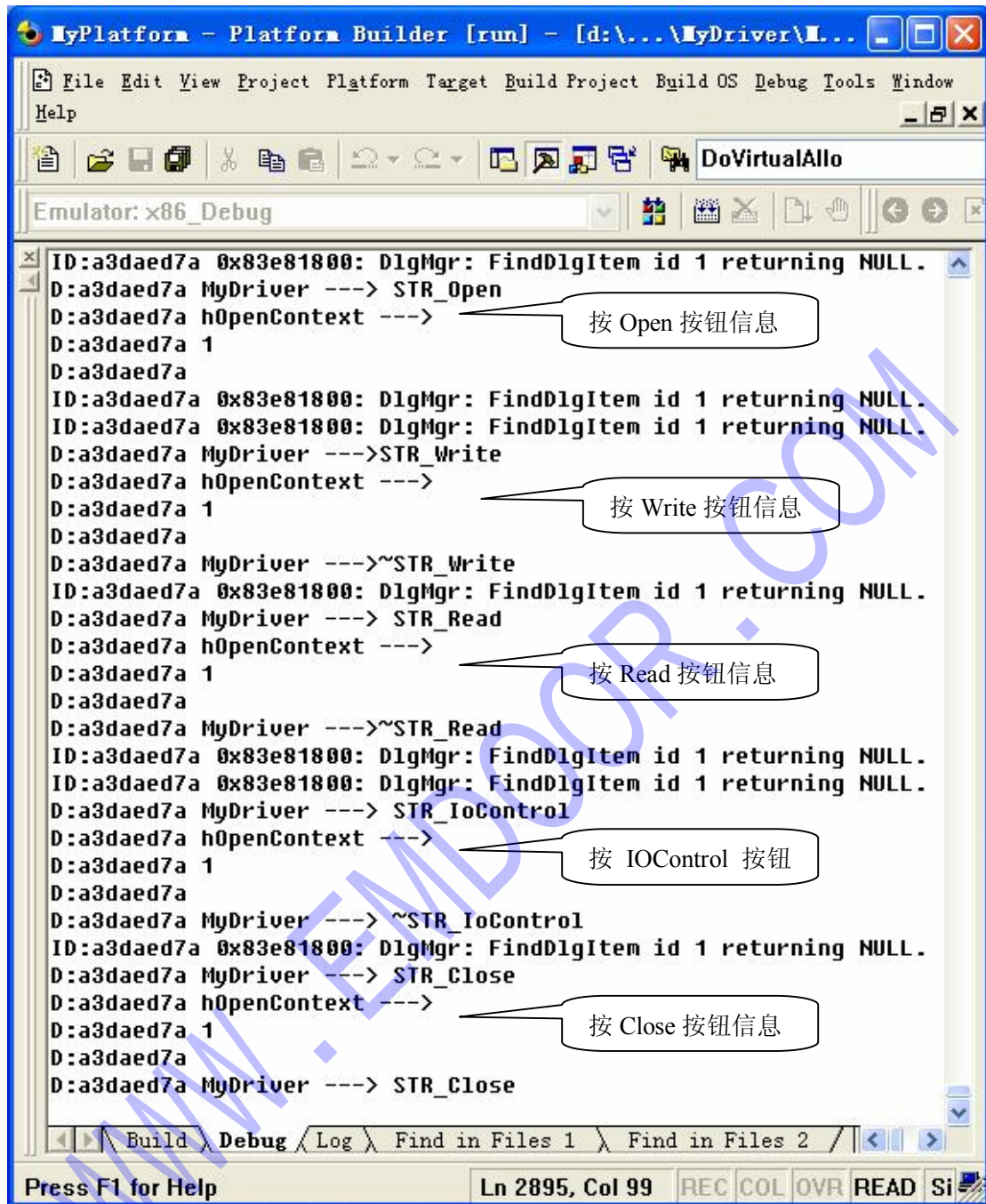


图 4-8 应用程序对驱动程序进行操作时 PB 调试信息

[实验内容]

- 1、新建并编译一个 WinCE 操作系统的仿真平台；
- 2、编译并调试一个流式接口驱动程序，同时利用远程调试工具观察驱动程序加载和测试过程中输出信息
- 3、编写一个流式驱动程序的测试程序，对上述编译好的驱动程序进行应用测试。
- 4、基本掌握流式驱动程序的编写方法和编译调试过程

[实验步骤]

第一步：利用 PB 新建一个基于 X86 的仿真平台，并编译成调试版 (debug)，如图 4-10 所示：

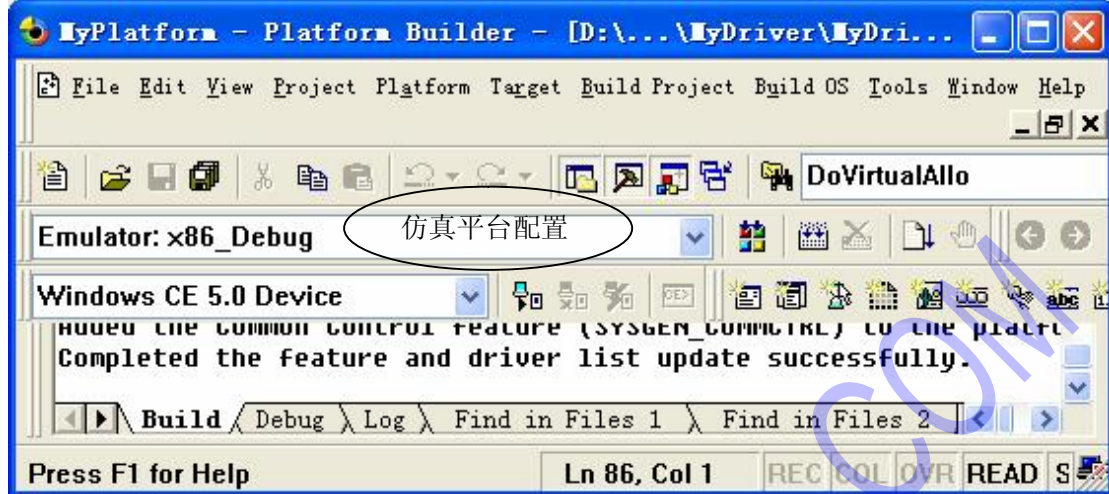


图 4-10 仿真平台配置

第二步：在新建的 PB 平台中，根据流式驱动程序的创建步骤新建一个流式驱动程序（见流式驱动程序的创建步骤），并对 PB 平台进行配置。

第三步：运行 WinCE 操作系统仿真平台，观测新建驱动程序的加载信息和过程；

第四步：在 VS2005.net 编译驱动程序的测试程序 DriverTest，将编译后的可执行文件 DriverApp.exe 复制新建的 WinCE 操作系统的仿真平台的目录下（如 D:\WINCE500\PBWorkspaces\MyPlatform\ReIDir\Emulator_x86_Debug）。

第五步：在仿真平台中运行驱动程序测试程序，并观察单击每个按钮时，PB 调试输出的信息；

[习题与思考题]

- 1、分析流式接口驱动程序的 DLL 接口函数集每个参数的作用
- 2、修改驱动程序，完善 IOControl 接口函数功能，使驱动程序在某个语义下完成一个动作（如数据的输出）；
- 3、如果需要 XSBASE270 目标平台中调试实验中的流式驱动程序，该怎样实现？

实验四 进程通信调试实验

[实验目的]

- 1、掌握 WinCE5.0 下进程间通信机制;
- 2、掌握通过对消息传递和共享内存的实现进程间通信的方法;
- 3、了解实现进程通信的编程方法

[实验仪器]

- 1、装有 Platform Builder、EVC 和 VS.Net 开发平台的 PC 机一台
- 2、XSBase270 实验开发平台一套

[实验原理]

1、软件原理:

在项目开发和系统集成中,进程间通信的应用非常广泛,进程间的通信有以下几种实现方法:利用 Windows 消息(WM_COPYDATA)、全局原子、内存映射、命名管道以及邮曹。下面介绍利用 WM_COPYDATA 和内存映射实现进程通信的方法。

1.1 WM_COPYDATA 方法

使用 WM_COPYDATA 消息可以很方便地传递进程间的数据,具体定义如下:

SendMessage (hwnd, WM_COPYDATA, wParam, lParam);

其中 wParam 设置为包含数据的窗体的句柄,lParam 指向一个 COPYDATASTRUCT 的结构:

```
typedef struct tagCOPYDATASTRUCT {
    DWORD dwData;    //用户定义数据
    DWORD cbData;    //数据大小
    PVOID lpData;    //指向数据的指针
} COPYDATASTRUCT
```

也就是通过 COPYDATASTRUCT 的结构来传递数据。

需要注意的是,WinCE 提供的 WM_COPYDATA 消息来传递进程间的数据,并没有提供同步机制,所以必须要使用 SendMessage 函数来发送消息,且等待直到对方处理完后返回,而不能使用 PostMessage 函数来发送 WM_COPYDATA。

1.2 内存映射文件的方法

内存映射文件提供了一种完全不同的读写文件的方法,它是将文件内容映射到内存的某个区域,读写文件直接操作内存即可。下面介绍内存映射文件所需的 API 函数:

(1) 创建用于内存映射访问的文件

在 WinCE 中,要想创建或打开一个用于内存映射访问的文件,需要通过 CreateFileForMapping 函数来实现,它是 CreateFile 函数的一个特殊版本,专门提供给内存

映射文件使用，CreateFileForMapping 函数的定义如下：

```
HANDLE CreateFileForMapping (
    LPCTSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
);
```

(2) 创建或打开内存映射文件对象

在使用内存映射文件时，需要创建或打开内存映射文件对象，它实际上是于已经创建或打开的文件建立连接。创建或打开内存映射对象通过 CreateFileMapping 函数实现，此函数定义如下：

```
HANDLE CreateFileMapping(
    HANDLE hFile,
    LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
    DWORD flProtect,
    DWORD dwMaximumSizeHigh,
    DWORD dwMaximumSizeLow,
    LPCTSTR lpName
);
```

成功创建内存映射文件对象，函数将返回一个内存映像对象句柄，如果在创建内存映射文件对象已存在，将直接返回已打开的内存映像文件对象句柄。

(3) 获取内存映像文件对象视图

当创建或打开了内存映射对象之后。接着就需要得到内存映射文件对象的数据内存指针，通过此指针则可以读写文件中的内容，也就是获取内存映射文件对象视图。得到内存映射文件对象的数据内存指针通过 MapViewOfFile 函数实现，此函数定义如下：

```
LPVOID MapViewOfFile(
    HANDLE hFileMappingObject,
    DWORD dwDesiredAccess,
    DWORD dwFileOffsetHigh,
    DWORD dwFileOffsetLow,
    DWORD dwNumberOfBytesToMap
);
```

函数返回值指向映射文件的内存指针。

(4) 取消文件视图的映射

在使用完内存映射文件之后，还要调用 UnmapViewOfFile 函数取消文件视图，进行内存清理工作。此函数定义如下：

```
BOOL UnmapViewOfFile(
    LPCVOID lpBaseAddress
);
```

2、进程通信的实现

2.1 WM_COPYDATA 方法

Client端的实现

```
void CClientDlg::OnBnClickedbtnsendmessage()
{
    CString str=_T("Server");
    CWnd *pWnd=CWnd::FindWindow(NULL,str);
    UpdateData(TRUE);
    if(pWnd && m_msg!=_T(""))
    {
        COPYDATASTRUCT buf;
        buf.lpData =m_msg.GetBuffer(m_msg.GetLength());
        buf.cbData =m_msg.GetLength()*2;
        pWnd->SendMessage(WM_COPYDATA,0,LPARAM(&buf));
    }
    else
        MessageBox(_T("No such Message"));
}
```

Server端的实现方法:

```
BOOL CServerDlg::OnCopyData(CWnd* pWnd, COPYDATASTRUCT* pCopyDataStruct)
{
    m_RecvData=LPCTSTR(pCopyDataStruct->lpData );
    UpdateData(FALSE);
    return CDialog::OnCopyData(pWnd, pCopyDataStruct);
}
```

2.2 内存映射方法

Client端的实现

```
void CClientDlg::OnBnClickedbtntomem()
{
    HANDLE hMapping;
    LPCTSTR lpData;
    hMapping=CreateFileMapping(HANDLE(0xFFFFFFFF),NULL,PAGE_READWRITE,0,1024,_T("MyShare"));
    if(hMapping==NULL)
    {
        AfxMessageBox(_T("Create FileMapping Failed"));
        return;
    }
    lpData=(LPCTSTR)MapViewOfFile(hMapping,FILE_MAP_ALL_ACCESS,0,0,0);
    if(lpData==NULL)
    {

```

```

        AfxMessageBox(_T("MapViewOfFile Failed"));
        return;
    }
    UpdateData(TRUE);
    if(m_msg!=_T(""))
        wsprintf(lpData,m_msg);
    else
        AfxMessageBox(_T("Message is empty!"));
}

```

Sever端的实现:

```

void CServerDlg::OnBnClickedbtnopen()
{
    HANDLE hMapping;
    LPTSTR lpData;
    hMapping=CreateFileMapping((HANDLE)0xFFFFFFFF,NULL,PAGE_READWRITE,0,1024,_T("MyShare"));
    if(hMapping==NULL)
    {
        AfxMessageBox(_T("CreateFileMapping failed"));
        return;
    }
    lpData=(LPTSTR)MapViewOfFile(hMapping,FILE_MAP_ALL_ACCESS,0,0,0);
    if(lpData==NULL)
    {
        AfxMessageBox(_T("MapViewOfFile failed"));
        return;
    }
    m_MapFileMessage=lpData;
    UpdateData(FALSE);
    ::UnmapViewOfFile(lpData);
    CloseHandle(hMapping);
}

```

[实验内容]

- 1、掌握 WinCE5.0 下进程间通信机制;
- 2、掌握通过对消息传递和共享内存的实现进程间通信的方法;
- 3、了解实现进程通信的编程方法;
- 4、掌握 EVC 或 VS.net 编程方法

[实验步骤]

第一步：连接好实验系统，打开实验箱电源。

第二步：分别利用 Visual Studio 2005.net 打开进程通信的 Client 和 Sever 工程文件 Client.sln 和 Server.sln，进行编译：

第三步：分别编译 Client 和 Sever 代码，点击运行按钮，这样程序就会下载到 XSBase270 目标板板上运行。Client 和 Sever 运行界面分别如图 4-1、4-2 所示。

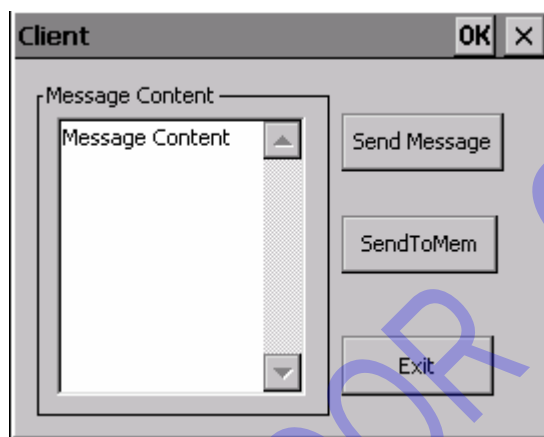


图 4-1 Client 端运行界面

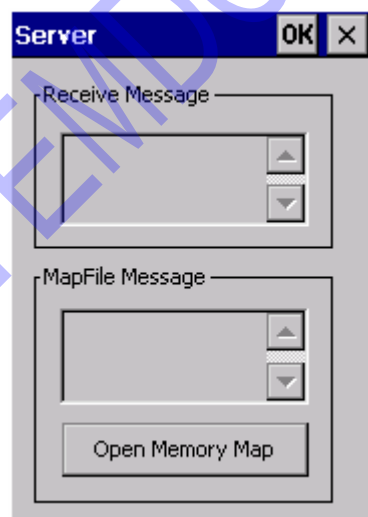


图 4-2 Sever 端运行界面

第四步：进程通信调试实验程序操作过程

在 XSBase270 目标板上同时运行 Sever 和 Client 程序，当单击 Client 端的“Send Message”按钮时，程序采用 WM_COPYDATA 消息实现进程通信，此时 Sever 端将接受 Client 发送的消息内容；

单击 Client 端的“SendToMeme”按钮时，程序利用内存映射实现进程通信，Client 端将通信内容映射到内存文件中，此时单击 Sever 端的“Open Memory Map”按钮，通过打开内存映射文件读出通信内容。

[习题与思考题]

- 1、如果需要采用具体文件实现进程通信，怎样实现？
- 2、比较利用 SendMessage 发送消息和 PostMessage 发送消息的区别？分析为什么进程通信不能采用 PostMessage 来实现？

WWW.EMDOOR.COM

实验五 SQL 数据库编程

[实验目的]

- 1、了解 ADOCE 组件；
- 2、掌握在 Windows 下的 DLL 等组件的注册；
- 3、掌握嵌入式数据库的开发过程；
- 4、掌握 SQL 语句的使用方法
- 5、熟悉 EVC 和 VS.Net 的使开发环境；

[实验仪器]

- 1、装有 Platform Builder、EVC 和 VS.Net 开发平台的 PC 机一台
- 2、XSBase270 实验开发平台一套

[实验原理]

1、ADOCE 简介

ADO 是 Microsoft 公司为各类型数据所提供的的一个策略性的高级界面。使用 ADO 的应用程序可以通过 OLE 数据库提供者存取和操作数据库服务器中的数据。ADO 主要的好处是容易使用、高速、占用内存少及占用磁盘空间小。ADO 提供一致高效的数据存取，用以建立前端数据库客户端或是建立应用程序、工具甚至是 Internet 浏览器的中间层商业对象。

ADO 是 Microsoft Universal Data Access 关键组件。Universal Data Access 提供对各种信息来源的高效存取，包括关系型和非关系型的数据源以及一个容易使用的程序化界面。该界面不但独立于开发工具，也不限定使用哪一种程序语言。这些技术使得公司能够综合不同的数据源，建立容易维护的解决方案并选择使用最好的工具、应用程序和应用平台。

ADOCE 提供 ADO 对 Windows CE 操作系统的子集合，这个子集合包括对 Recordset 对象和 Field 对象的支持。ADOCE 对 Windows CE 来说增加了新的数据库功能，可以存取保存在本机中的数据库并且提供网络数据库的数据同步。ADOCE 在任何 COM- capable 环境下都提供对 Windows CE 数据库引擎的存取。

ADOCE 提供了 ADOCE.Connection 对象，用来创建和存取数据库、数据表。ADOCE 还提供了一个 Errors 对象，它用来代表各类运行中产生的错误。此外，ADOCE 提供的名为 ADOCE.Recordset 的对象，用来表示从实际数据库、数据表里提取组合而成的虚拟记录集。在 ADOCE.Recordset 对象中含有很多的方法与属性，它们主要用于管理已经获得的记录集合，此外它还含有一个 Field 对象，用于表示字段。

1.1 Connection 对象：

Connection 对象的方法共有 7 种，如表 5.1 所示。

表 5.1 Connection 对象的方法

方法	描述
BeginTrans	在目前的数据库连接上建立一个新的事务(transaction)
Close	关闭正在使用的数据库连接，同时中断与实际数据库间的沟通渠道
CommmmitTrans	将目前事务的内容写入
Open	打开与实际数据源间渠道沟通
OpenSchema	从数据源的提供者处获取关于数据库的 schema 信息
Execute	执行一个不会返回记录集的命令
RollbackTran	取消当前的事务

1.2 Recordset 对象：

Recordset 对象的方法共有 17 种，如表 5.2

表 5.2 Recordset 对象方法

方法	描述
Open	用来取得虚拟记录集，执行 SQL 语句
Close	关闭虚拟记录集
Move	移动指向虚拟记录集记录的指针
MoveFirst	将指向虚拟记录集记录的指针移到第一笔的位置
MoveLast	将指向虚拟记录集记录的指针移到最后一笔的位置
MoveNext	将指向虚拟记录集记录的指针移到目前位置的下一笔
MovePrevious	将指向虚拟记录集记录的指针移到目前位置的前一笔
Supports	判断虚拟记录集是否支持某些特性
AddNew	针对目前的虚拟记录集新建一笔记录
Delete	从目前的虚拟记录集中删除一笔记录
Update	将虚拟记录集中被更新的记录写入数据库
CancelUpdate	取消保留在内存中关于记录的修改
Requery	以数据源更新虚拟记录集的内容 t from the data
Find	返回符合特定条件的虚拟记录集
Seek	从建有索引的虚拟记录集中查找一笔记录，并将记录指针指向该记录使之成为当前的记录
clone	复制一份虚拟记录集

1.3 Field 对象：

Field 对象是组成数据表的最基本组件。Field 对象的方法共有 2 种，如表 5.3 所示

表 5.3 Field 对象的方法

方法	描述
AppendChunk	在一个二进制数据或大量文字数据的字段中新建数据
GetChunk	取得全部或部分二进制数据或大量文字数据

2、SQL 简介

SQL 是一种专门用来处理关系型数据库的语句(statement)，也有人将 SQL 称作是第 4 代程序语言，它是 Structured Query Language3 个单词的缩写。

主要功能包括：

- (1) 定义、创建和修改数据库及数据表
- (2) 新增、删除、修改、查询实际数据表中的记录的功能
- (3) 保持数据库的安全性数据的整合性

SQL 语句分成两种：DDL 与 DML。DDL 是 Data Definition Language (数据定义语言) 的缩写，它支持上述第 1 项功能，而 DML 是 Data Manipulation Language(数据操作语言) 的缩写，它支持上述第 2 项功能。ADOCE 目前支持的 DDL 如表 5.4 所示，ADOCE 目前支持的 DDL 如表 5.5 所示，所有的 DML 语句支持 Where 条件。

表 5.4 ADOCE 控件支持的 DDL 语句

DDL 语句	描 述
Create Database	创建新的数据库文件
Create Table	在数据库文件中创建新的数据表
Alter Table	修改现存数据表的结构
Create Index	根据数据表的字段建立索引
Drop Database	删除数据库
Drop Table	删除数据表 o
Drop Index	删除索引

表 5.5 ADOCE 控件支持的 DML 语句

DML 语句	描 述
Select	查询数据表中记录
Delete	删除数据表中记录
Insert	向数据表中新增一条记录
Update	修改更新数据表记录

3 ADOCE 编程实现

3.1 注册 ADOCE 的 DLL 组件

ADOCE 属于标准的 COM 组件，采用动态连接库的形式进行发布，在使用 ADOCE 进行数据编程，必须对 ADOCE 的动态连接库下载到目标板，并利用 REGSVRCE.EXE 对部分需要注册的 DLL 组件进行注册。ADOCE 需要的组件如表 5.6

表 5.6 ADOCE 控件的 DLL 组件

组件名称	是否需要注册
Adoce31.dll	是
Adoxce31.dll	是
Adocedb31.dll	否

Adoceedb31.dll	否
Msdaer.dll	是
Msdaeren.dll	否
msdadc.dll	是
msdaosp.dll	否

用户可以将上述 DLL 组件下载到目标板，然后利用 REGSVRCE.EXE 命令对它们进行注册。本实验程序在运行时自动将上述 DLL 组件下载到目标板的 WinCE 操作系统的 \windows 目录中，然后利用 CreateProcess 函数运行 REGSVRCE.EX 命令对 DLL 组件进行注册。

```

BOOL CdatabaseDlg::RegisterDLL(void)
{
    BOOL ok1, ok2, ok3, ok4, ok5, ok6, ok7, ok8;
    ok1=ok2=ok3=ok4=ok5=ok6=ok7=ok8=FALSE;
    //1. 注册DLL
    ok1=CreateProcess(_T("\\RegSvrCe"), _T("/s \\Windows\\adoce31.dll"),
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL);
    ok2=CreateProcess(_T("\\RegSvrCe"), _T("/s \\Windows\\Adoxce31.dll"),
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL);
    ok3=CreateProcess(_T("\\RegSvrCe"), _T("/s \\Windows\\Adocedb31.dll"),
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL);
    ok4=CreateProcess(_T("\\RegSvrCe"), _T("/s \\Windows\\Adoceedb31.dll"),
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL);
    ok5=CreateProcess(_T("\\RegSvrCe"), _T("/s \\Windows\\Msdaer.dll"),
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL);
    ok6=CreateProcess(_T("\\RegSvrCe"), _T("/s \\Windows\\Msdaeren.dll"),
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL);
    ok7=CreateProcess(_T("\\RegSvrCe"), _T("/s \\Windows\\msdadc.dll"),
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL);
    ok8=CreateProcess(_T("\\RegSvrCe"), _T("/s \\Windows\\msdaosp.dll"),
        NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL);
    if(! (ok1&&ok2&&ok3&&ok4&&ok5&&ok6&&ok7&&ok8))
        return FALSE;
    else
        return TRUE;
}

```

3.2 ADOCE 的初始化步骤:

由于 ADOCE 属于标准的 COM 组件，必须初始化一个 COM，然后再进行 ADO 数据源连接的相关编程，具体实现过程如下：

```

BOOL CdatabaseDlg::CreateConnection(void)
{
    CLSID tClsid;
    HRESULT hr;
}

```

```
//0 初始化COM库
hr=CoInitializeEx(NULL, COINIT_MULTITHREADED); //初始化COM库
if (FAILED(hr))
{
    MessageBox(_T("初始化COM库失败!"), _T("系统信息"), MB_OK | MB_ICONINFORMATION);
    return FALSE;
}

//1, 得到ADO连接对象对应ClassID
hr = CLSIDFromProgID( g_szADOCE31ConnProgID, &tClsid );

if (FAILED(hr))
    return FALSE;
//2, 创建ADO连接对象
hr = CoCreateInstance (tClsid, NULL,
    CLSCTX_INPROC_SERVER | CLSCTX_LOCAL_SERVER,
    IID__Connection, (LPVOID *)&m_pADOCEConn);
if ((!m_pADOCEConn) || FAILED(hr))
    return FALSE;
//3. 设置连接数据库的Provider
hr = m_pADOCEConn->put_Provider(TEXT("cedb"));
if FAILED(hr)
    return FALSE;
//4. 得到记录集对象的ClassID
hr = CLSIDFromProgID( g_szADOCE31RSPProgID, &tClsid );
if FAILED(hr)
    return FALSE;
//5. 创建结果集对象
hr = CoCreateInstance (tClsid, NULL, CLSCTX_INPROC_SERVER | CLSCTX_LOCAL_SERVER,
    IID__Recordset, (LPVOID *)&m_pADOCERS);
if ((!m_pADOCERS) || FAILED(hr))
    return FALSE;
return TRUE;
}
```

3.3 ADOCE 的编程实现

ADOCE 进行数据库编程基本上属于对 SQL 语句的执行，下面分别对建立数据库、打开数据库、建立数据表以及增加数据记录的编程进行分析。

(1) 执行 SQL 的函数：

```
BOOL CdatabaseDlg::ExecSql(CString strSql, BOOL View)
{
    HRESULT hr;
    VARIANT varStrSql, varEmpty;
    VariantInit(&varStrSql); //初始化VARIANT变量
    VariantInit(&varEmpty);;
```

```
varStrSql.bstrVal = SysAllocString(strSql);
varStrSql.vt = VT_BSTR; //定义字符串
hr = m_pADOCERS->Open(varStrSql, varEmpty, MSADOCE::adOpenDynamic,
    MSADOCE::adLockOptimistic, MSADOCE::adCmdText);
SysFreeString(varStrSql.bstrVal);
if (FAILED(hr))
    return FALSE;
if(View)
{
    long iCount = 0;
    hr = m_pADOCERS->get_RecordCount(&iCount);
    if(iCount>0)
        AddRecordToView();
}
m_pADOCERS->Close();
return TRUE;
}
```

(2) 利用“CREATE DATABASE 数据库名”建立数据库

```
BOOL CdatabaseDlg::CreateDatabase(LPCTSTR szDbName)
{
    CString strSql;
    strSql.Format(_T("CREATE DATABASE '%s'"), szDbName);
    if(!ExecSql(strSql))
        return FALSE;
    else
        return TRUE;
}
```

(3) 利用 CREATE TABLE 建立数据表

```
void CdatabaseDlg::OnBnClickedbtncreatetabe()
{
    CString m_strSql;
    m_strSql="CREATE TABLE student(StuID varchar(20),Name varchar(20),age int,class
varchar(10))";
    if(!ExecSql(m_strSql))
        MessageBox(_T("建立数据表失败!"), _T("系统信息"), MB_OK|MB_ICONINFORMATION);
    else
        MessageBox(_T("建立数据表成功!"), _T("系统信息"), MB_OK|MB_ICONINFORMATION);
}
```

(4) 打开数据库函数

```
BOOL CdatabaseDlg::OpenDatabase(LPCTSTR szDbName)
{
    HRESULT hr;
```

```

VARIANT varConn1;
//与指定的数据库建立连接
hr = m_pADOCEConn->Open(LPTSTR(szDbName), TEXT(""), TEXT(""), MSADOCE::adOpenUnspecified);
if FAILED(hr)
    return FALSE;
VariantInit(&varConn1);
varConn1.pdispVal=m_pADOCEConn;
varConn1.vt=VT_DISPATCH;
//设置结果集对象到已建立的连接上
hr=m_pADOCERS->put_ActiveConnection(varConn1);
if (FAILED(hr))
    return FALSE;
return TRUE;
}

```

(5) 增加数据记录

```

void CdatabaseDlg::OnBnClickedbtnaddrecord()
{
    CAddRecordsetDlg dlg(_T("增加记录"));
    if(dlg.DoModal() ==IDOK)
    {
        HRESULT hr;
        CString m_strSql, strValue;
        VARIANT varStrSql, varEmpty, varFieldValue;
        VariantInit(&varStrSql);
        VariantInit(&varEmpty);
        m_strSql="SELECT * FROM student";
        varStrSql.bstrVal =SysAllocString(m_strSql);
        varStrSql.vt = VT_BSTR;
        hr = m_pADOCERS->Open(varStrSql, varEmpty, MSADOCE::adOpenDynamic,
            MSADOCE::adLockOptimistic, MSADOCE::adCmdText);
        SysFreeString(varStrSql.bstrVal);
        if (FAILED(hr))
        {
            MessageBox(_T("打开数据库错误"), _T("系统信息"), MB_OK|MB_ICONINFORMATION);
            return;
        }
        m_pADOCERS->AddNew(varEmpty, varEmpty);
        strValue=dlg.m_studentID;
        varFieldValue.bstrVal =SysAllocString(strValue);
        varFieldValue.vt =VT_BSTR;
        SetFieldValue(0, varFieldValue);

        strValue=dlg.m_studentName;
        varFieldValue.bstrVal =SysAllocString(strValue);
        varFieldValue.vt =VT_BSTR;
    }
}

```

```
SetFieldValue(1, varFieldValue);

varFieldValue.iVal =dlg.m_studentAge ;
varFieldValue.vt =VT_I2;
SetFieldValue(2, varFieldValue);

strValue=dlg.m_studentClass;
varFieldValue.bstrVal =SysAllocString(strValue);
varFieldValue.vt =VT_BSTR;
SetFieldValue(3, varFieldValue);
m_pADOCERS->Close();
ShowData(); //显示数据函数
}
}
```

[实验内容]

- 1、掌握在 Windows 下的 DLL 等组件的注册；
- 2、掌握利用 ADOCE 开发嵌入式数据库的过程；
- 3、掌握 SQL 语句的使用方法

[实验步骤]

第一步：连接好实验系统，打开实验箱电源；

第二步：利用 VS2005.net 打开数据库工程文件 database.sln，由于本实验程序实现 ADOCE 组件的自动下载到目标板，必须在 VS2005.net 设置 DLL 组件的所在路径以及下载到目标板的路径，如图 5-1 所示。

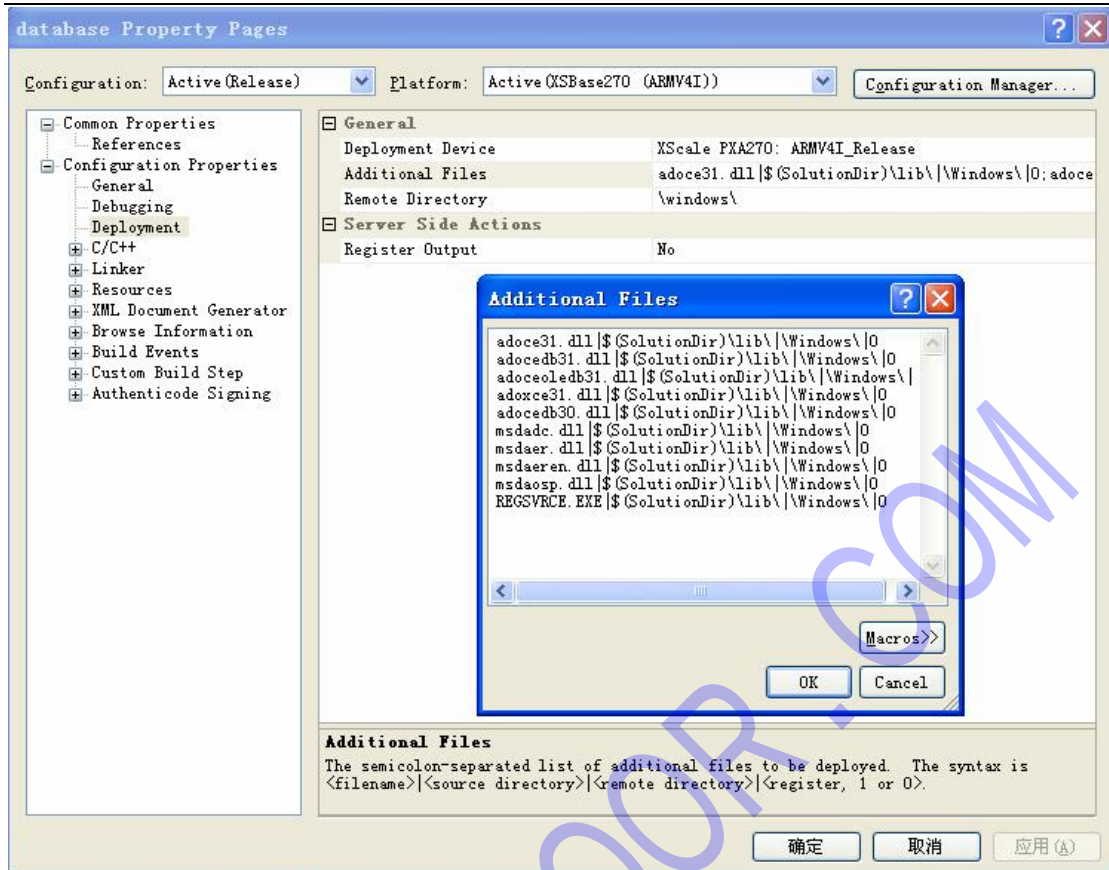


图 5-1 ADOCE 组件的自动下载设置

第三步：编译 database 工程文件；点击运行按钮自动将所需要的文件下载（database.exe 和 ADOCE 组件），运行界面如图 5-2 所示：



图 5-2 ADOCE 数据库测试程序运行界面

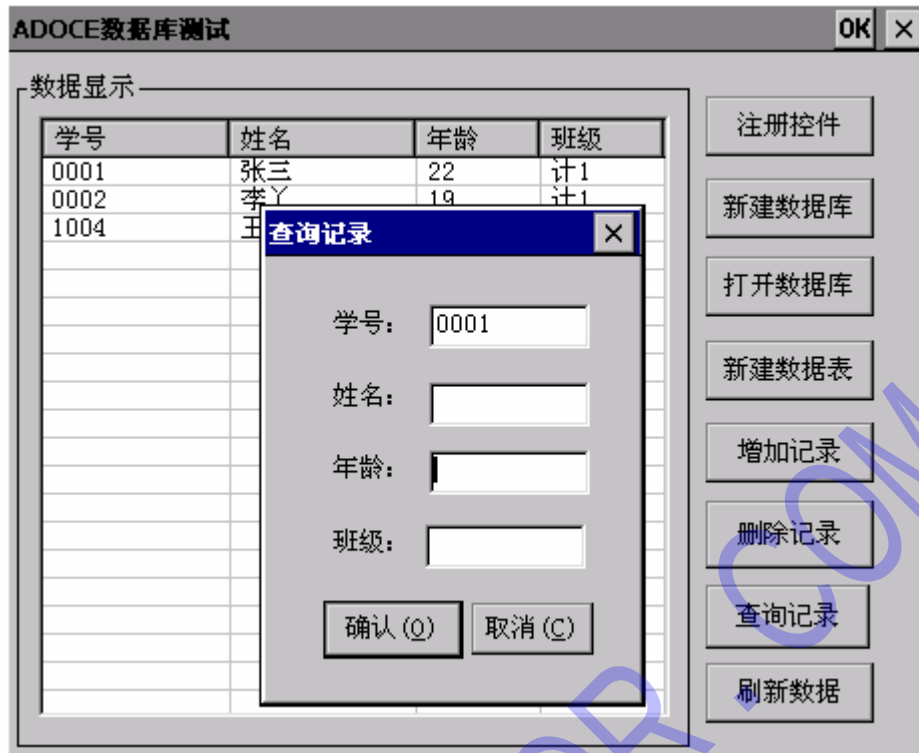


图 5-4 数据记录查询界面

[习题与思考题]

- 1、修改源程序，增加对数据记录进行修改的操作；
- 2、源程序中查询记录和增加记录窗体采用同一个窗体文件，但其 Caption 显示分别为“查询记录”和“增加记录”，分析其实现过程。
- 3、如果需要对数据库中的数据表进行删除操作，应怎样实现？
- 4、修改源程序，增加一个“或”条件查询操作。

实验六 IO 接口控制实验之七段数码管和 LED 显示实验

[实验目的]

- 1、掌握在 Windows CE 下访问硬件 I/O 寄存器的一般方法；
- 2、了解 WinCE 下 IO 访问机制和原理；
- 3、了解数码管(LED)的显示及控制原理；
- 4、熟悉 EVC 和 VS.Net 的使开发环境；

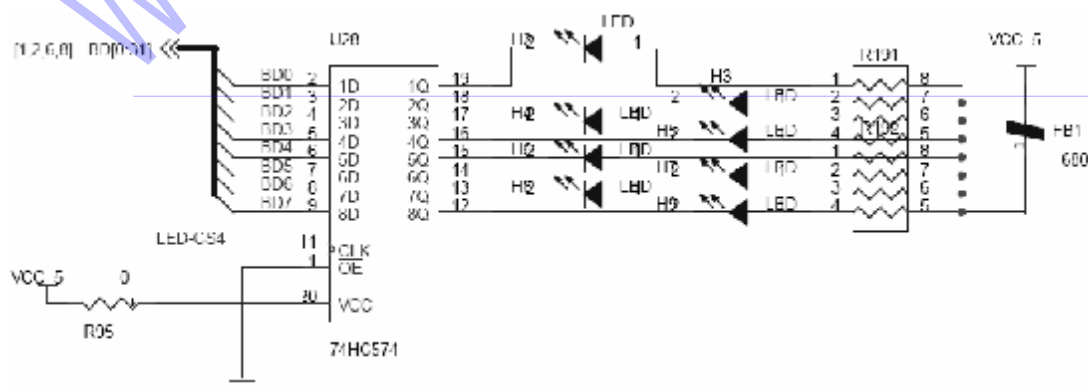
[实验仪器]

- 1、装有 Platform Builder、EVC 和 VS.Net 开发平台的 PC 机一台
- 2、XSBase270 实验开发平台一套

[实验原理]

1、硬件接口电路分析

XSBase 目标板 LED 和七段数码显示接口电路如图 6-1 所示，74HC574 为 D 锁存器，在时钟信号 CLK 作用下，该锁存器将输入信号进行锁存，即 $xQ = xD$ ($x=1\sim 8$)。从电路图中可以看出，LED 和八段数码显示电路将 74HC574 的时钟信号输入端作为片选信号，其中 LED 显示的片选信号为 LED_CS4、七段数码显示的片选信号为 LED_CS1（另外还有 2 组七段数码显示未画出，参考原理图）。在七段数码显示电路中，数据的高位（D7、D15：即数码管的小数点 dp 段）用作七段数码的公共选通信号，通过控制 PNP 三极管来控制数码管的显示。



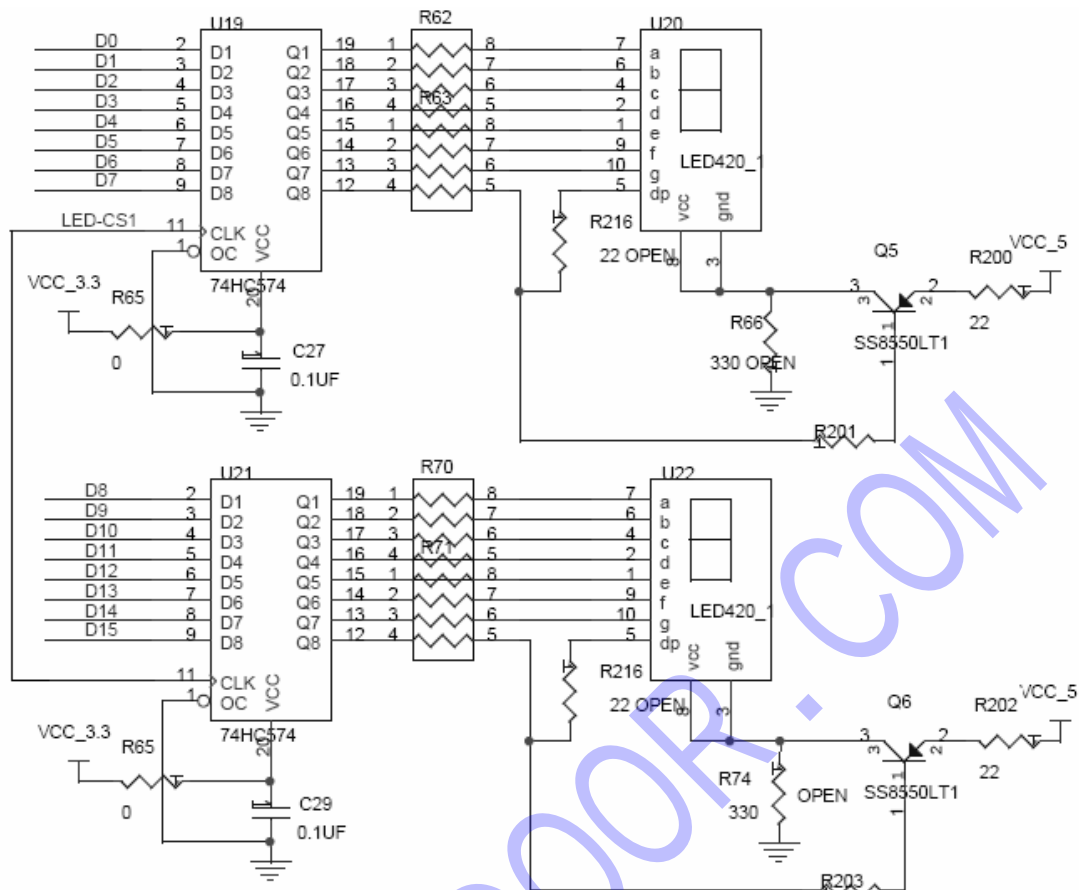


图 6-1 LED 和七段数码显示接口电路

显示电路中的片选信号 LED_CS_x($x=1-4$), 由 XSBase270 目标板系统的处理器 PXA270x 的地址信号 BA22~BA20 通过 3-8 译码器 LC138 产生 (如图 2-2 所示)。

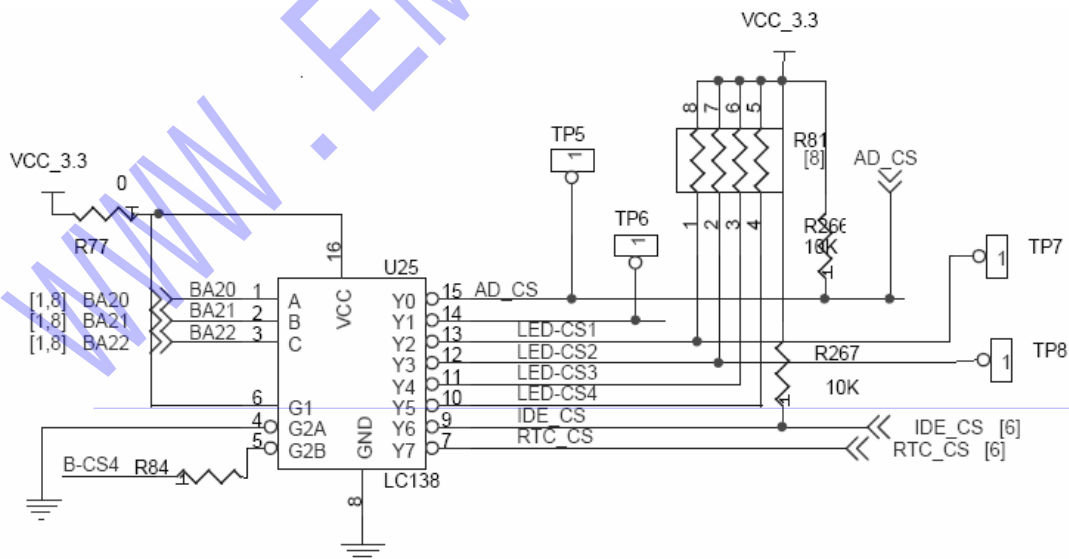


图 6-2 片选信号产生电路

由 3-8 控制功能可知, 当 BA22、BA21、BA20=101 时产生 LED 显示电路的片选信号 LED_CS4, 当 BA22、BA21、BA20=010、011、100 时分别产生七段数码显示电路的片选信号 LED_CS1、LED_CS2、LED_CS3 (另外 2 组八段显示电路参考系统提供的总电路图)。根据分析, 可以得出七段数码管和 LED 的片选信号为: LED_CS1=0x10200000、LED_CS2

=0x10300000、LED_CS3=0x10400000，LED_CS4=0x10500000。

2、Windows CE 下访问物理地址的方法

2.1 虚拟内存函数介绍

IO 接口属于 XScale 处理器的外部设备，类似于通常所使用的 PC 的串口、并口或者 PC 喇叭。对于一般单片机，可以直接操作硬件，即通过直接读写 IO 端口来访问硬件。而对于 Windows CE 操作系统，不允许直接访问硬件，尽管知道物理地址，我们仍不能直接访问它，因此必须使用特殊的方法来访问硬件。

Windows CE 操作系统出于稳定性和安全的要求，屏蔽了用户应用程序对硬件访问的权限，只有内核应用程序才可以访问硬件资源。如果要直接访问某一个地址的物理内存，可以采用内存映射的方法，将该硬件寄存器映射到普通的内存空间，像操作内存地址一样操作硬件寄存器。Windows CE 提供了 VirtualAlloc()和 VirtualCopy()函数，VirtualAlloc 负责在虚拟内存空间内保留一段虚拟内存，而 VirtualCopy 负责把一段物理内存和虚拟内存绑定，这样，最终对物理内存的访问还是通过虚拟地址进行。下面简单介绍这些函数的作用和使用方法。

VirtualAlloc()函数用于分配一块内存空间。如果要映射一个硬件寄存器，必然要将其映射到一块确定的内存空间。在这里我们分配一个内存空间。这个内存空间并不占用实际内存空间，而是留待后面进行分配。

VirtualAlloc()函数原型为：

```
LPVOID VirtualAlloc(
    LPVOID lpAddress,           // 希望的虚拟内存起始地址
    DWORD dwSize,               // 以字节为单位的大小
    DWORD flAllocationType,     // 申请类型，分为 Reserve 和 Commit
    DWORD flProtect             // 访问权限
);
```

其参数 lpAddress 包含一个内存地址，用于定义待分配区域的首地址。通常可将此参数设置为 NULL，由系统通过搜索地址空间来决定满足条件未保留的地址空间。这时系统可从地址空间的任意位置处开始保留一个区域。如果存在一个足够大的空闲区域，那么系统将会保留此区域并返回此保留区域的虚拟地址，否则将导致分配的失败而返回 NULL。这里需要特别指出的是，在指定 lpAddress 的内存地址时，必须确保从分配粒度的边界处开始。

VirtualCopy()负责将硬件设备寄存器的物理地址与 VirtualAlloc()分配的虚拟地址做一个映射关系，这样驱动程序访问虚拟地址即是访问对应的物理地址处的第一个寄存器。

VirtualCopy()函数原型为：

```
BOOL VirtualCopy(
    LPVOID lpvDest,             // 虚拟内存的目标地址
    LPVOID lpvSrc,              // 物理内存地址
    DWORD cbSize,               // 要绑定的大小
    DWORD fdwProtect            // 访问权限
);
```

由于 EVC 里没有提供没有 VirtualCopy()的头文件和库文件（这些是由 Platform Builder 提供的），因此需要将它们作为外部函数调用导入。如下：

```
extern "C" __declspec(dllimport) BOOL VirtualCopy(LPVOID lpvDest, LPVOID lpvSrc,
```

```
DWORD cbSize, DWORD fdwProtect );
```

如此调用 VirtualCopy 函数后，物理地址和虚拟内存空间就对应起来了。

对于分配的内存空间，虽然没有占用实际的内存空间，但是还是占用了系统的存储器地址，因此在退出程序时需要将其释放掉。使用的函数是 VirtualFree()。其函数原型为：

```
BOOL VirtualFree(
    LPVOID lpAddress, // 虚拟内存的地址
    DWORD dwSize,     // 释放的地址空间区域的大小
    DWORD dwFreeType  // 释放内存类型
);
```

其中，参数 lpAddress 为指向待释放页面区域的指针。参数 dwSize 指定了要释放的地址空间区域的大小，如果参数 dwFreeType 指定了 MEM_RELEASE 标志，则将 dwSize 设置为 0，由系统计算在特定内存地址上的待释放区域的大小。参数 dwFreeType 为所执行的释放操作的类型。

2.2 虚拟内存函数使用方法

首先使用 VirtualAlloc 分配出一个虚拟的地址空间，代码如下：

```
VirtualAlloc(0,0x1000,MEM_RESERVE,PAGE_READWRITE)
```

这样就分配出一个 MEM_RESERVE 类型的存储器空间，它并没有占用实际内存空间，而是虚拟的地址空间。

接着将实际的硬件地址（例如 LED 的片选控制信号地址）映射到前面分配的虚拟地址空间，使用 VirtualCopy 函数建立起两个地址间的映射关系。具体代码如下：

```
VirtualCopy((PVOID)pLightReg,(PVOID)(pLightIoBaseAddress>>8),0x1000,PAGE_READWRITE|PAGE_NOCACHE|PAGE_PHYSICAL)
```

这里的 pLightReg 即前面分配的虚拟地址空间，而 pLightIoBaseAddress 为实际的硬件地址，需要将它右移 8 位，因为在函数中存储器分配是以 256 位为单位的。而后面的选项则指定了映射地址的属性——可读写、不缓冲以及硬件物理地址。

现在就可以使用虚拟地址访问原本不能直接访问的硬件地址了。在程序退出的时候，出于安全与妥善考虑，还应使用 VirtualFree 将已经分配的空间释放掉。实际的代码如下：

```
VirtualFree((PVOID) pLightReg,0,MEM_RELEASE);
```

[实验内容]

- 1、了解在 Windows CE 下访问一般硬件寄存器的方法，掌握 EVC 下对硬件 I/O 寄存器的访问方法，以及使用 EVC 对硬件设备编程的一般方法。
- 2、使用 EVC 或 VS.net 编写对 XSBASE270 目标板的 LED 和七段数码控制程序。

[实验步骤]

第一步：连接好实验系统，打开实验箱电源。

第二步：打开对应的工程文件，我们可以看到硬件地址映射到内存的关键代码如下：

```
BOOL CLEDDlg::SetMemoryMap()
{
    if(!(v_pLEDBaseAddr1=(USHORT*)VirtualAlloc(0,0x400,MEM_RESERVE,PAGE_READWRITE))) {
```

```

        MessageBox(TEXT("VirtualAlloc() failed!\r\n"),NULL,MB_OK);
        return FALSE;
    }
    if(!VirtualCopy((PVOID)v_pLEDBaseAddr1,(PVOID)(LED_BASEADDR1>>8),0x400,PAGE_READWRITE|PAGE_NOCACHE|PAGE_PHYSICAL)) {
        VirtualFree((PVOID)v_pLEDBaseAddr1,0,MEM_RELEASE);
        pLightReg = NULL;
        MessageBox(TEXT("VirtualCopy() failed!\r\n"),NULL,MB_OK);
        return FALSE;
    }
    //////////////////////////////////
    if(!(v_pLEDBaseAddr2=(USHORT*)VirtualAlloc(0,0x400,MEM_RESERVE,PAGE_READWRITE))) {
        MessageBox(TEXT("VirtualAlloc() failed!\r\n"),NULL,MB_OK);
        return FALSE;
    }
    if(!VirtualCopy((PVOID)v_pLEDBaseAddr2,(PVOID)(LED_BASEADDR2>>8),0x400,PAGE_READWRITE|PAGE_NOCACHE|PAGE_PHYSICAL)) {
        VirtualFree((PVOID)v_pLEDBaseAddr2,0,MEM_RELEASE);
        pLightReg = NULL;
        MessageBox(TEXT("VirtualCopy() failed!\r\n"),NULL,MB_OK);
        return FALSE;
    }
    //////////////////////////////////
    if(!(v_pLEDBaseAddr3=(USHORT*)VirtualAlloc(0,0x400,MEM_RESERVE,PAGE_READWRITE))) {
        MessageBox(TEXT("VirtualAlloc() failed!\r\n"),NULL,MB_OK);
        return FALSE;
    }
    if(!VirtualCopy((PVOID)v_pLEDBaseAddr3,(PVOID)(LED_BASEADDR3>>8),0x400,PAGE_READWRITE|PAGE_NOCACHE|PAGE_PHYSICAL)) {
        VirtualFree((PVOID)v_pLEDBaseAddr3,0,MEM_RELEASE);
        pLightReg = NULL;
        MessageBox(TEXT("VirtualCopy() failed!\r\n"),NULL,MB_OK);
        return FALSE;
    }
    //////////////////////////////////
    if(!(pLightReg=(char*)VirtualAlloc(0,0x400,MEM_RESERVE,PAGE_READWRITE))) {
        MessageBox(TEXT("VirtualAlloc() failed!\r\n"),NULL,MB_OK);
        return FALSE;
    }
    if(!VirtualCopy((PVOID)pLightReg,(PVOID)(pLightIoBaseAddress>>8),0x400,PAGE_READWRITE|PAGE_NOCACHE|PAGE_PHYSICAL)) {
        VirtualFree((PVOID)pLightReg,0,MEM_RELEASE);
        pLightReg = NULL;
        MessageBox(TEXT("VirtualCopy() failed!\r\n"),NULL,MB_OK);
    }

```

```

        return FALSE;
    }
    return TRUE;
}

```

物理地址、外部函数引用以及 0-9 的七段数码显示声明代码：

```

char* pLightReg = NULL;
USHORT*v_pLEDBaseAddr1=NULL;
USHORT*v_pLEDBaseAddr2=NULL;
USHORT*v_pLEDBaseAddr3=NULL;
#define BIT7 (0x1<<7)
#define BIT15 (0x1<<15)
extern "C" __declspec(dllimport) BOOL VirtualCopy(LPVOID lpvDest, LPVOID lpvSrc,
DWORD cbSize, DWORD fdwProtect );

#define LED_BASEADDR1 0x10200000
#define LED_BASEADDR2 0x10300000
#define LED_BASEADDR3 0x10400000
#define pLightIoBaseAddress 0x10500000
BYTE NumData[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};

```

编写的写内存函数：

```

UINT16 READ_PORT_USHORT(UINT16 *pAddr)
{
    UINT32 nAlignedAddr = ((UINT32)pAddr & ~0x3);
    if ((UINT32)pAddr & 0x2) // Is the address an "odd" word within a longword?
    { // Yes - read a longword and data is in the upper word.
        return((UINT16) ((* (volatile unsigned long *)nAlignedAddr) >> 16));
    }
    else
    {
        // No - read single word.
        return(* (volatile unsigned short *)nAlignedAddr);
    }
}

void WRITE_PORT_USHORT(UINT16 *pAddr, UINT16 Data)
{
    UINT32 nAlignedAddr = ((UINT32)pAddr & ~0x3);
    if ((UINT32)pAddr & 0x2) // Is the address an "odd" word within a longword?
    { // Yes - read a longword and data is in the upper word.
        *((volatile UINT32 *)nAlignedAddr) = (READ_PORT_USHORT(pAddr) | (Data << 16));
    }
    else
    {

```

```

*(volatile UINT16 *)pAddr = Data;
}
}

```

第三步：编译、下载与调试

(1) IO 接口控制 LED 工程的调试和发行版的编译配置如图 6-3 所示。

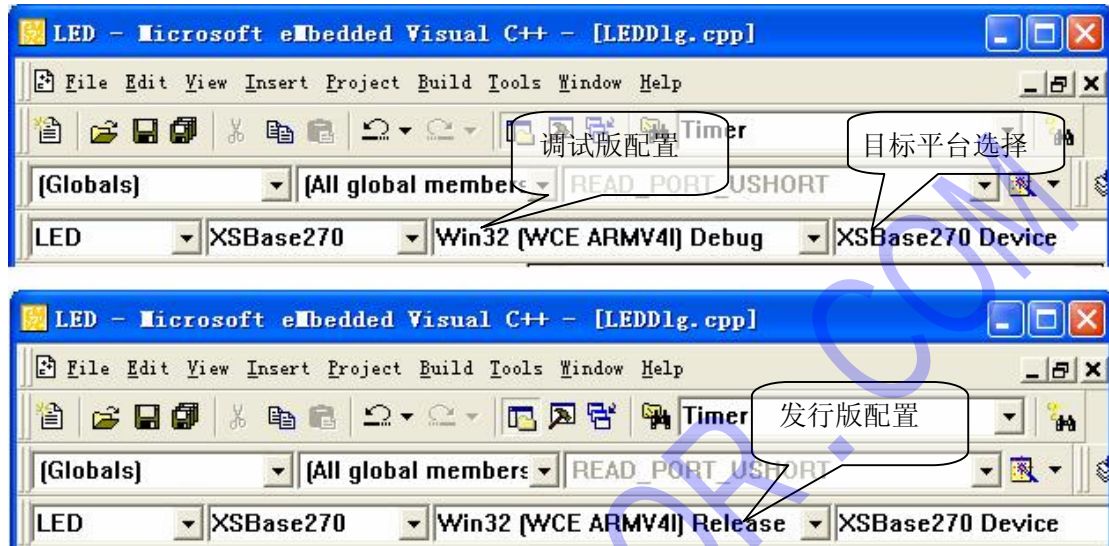


图 6-3 IO 接口控制的 EVC 编译配置图

(2) 编译该代码，点击运行按钮，这样程序就会下载到 XSBase270 目标板板上运行。运行界面如图 6-4 所示。

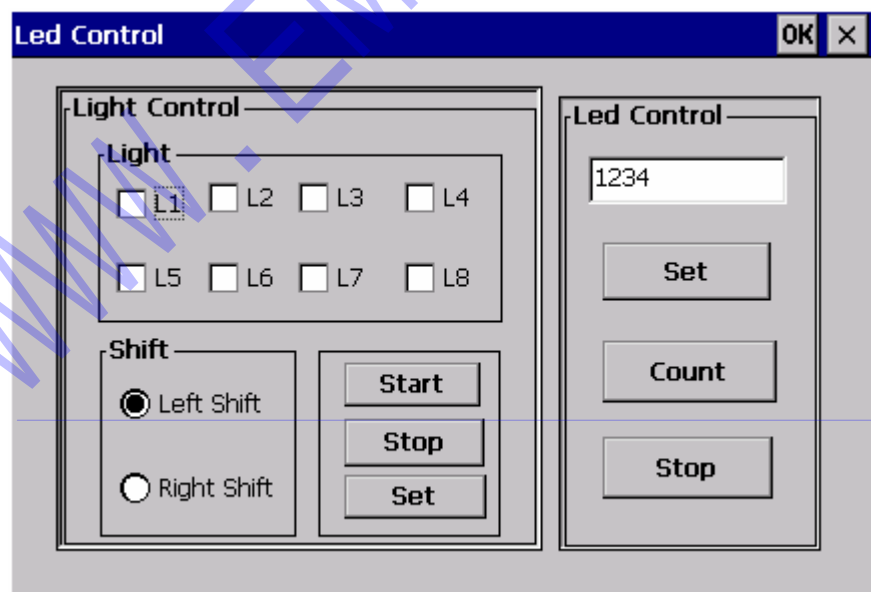


图 6-4 IO 接口 LED 和七段数码管实验运行界面

(3) 应用程序操作过程：

- Light Control 表示对 XSBase270 目标板的 8 个发光二极管控制，按“Start”按钮，发光

二极管左移或右移；L1~L8 对应目标板的发光二极管，勾选后按“Set”按钮，对应的二极管点亮。按“Stop”按钮，二极管的左移或右移停止。

- Led Control 表示对目标板的 4 个七段数码管的控制，在文本框中输入 4 位（0~9）数字，按“Set”按钮，七段数码管将显示输入的数字；按“Count”按钮，数码管进行计数操作，按“Stop”按钮停止计数。

（4）修改程序，实现其他不同的显示实验效果（如 2 个二极管一起左或右移等）。

[习题与思考题]

1、根据系统电路图，分析下列程序源代码

```
void CLEDDlg::SetLedValue(UINT indata)
{
    USHORT Data;
    UINT buf;
    buf=indata;
    buf=buf%1000000;
    Data=NumData[buf/100000];
    buf=buf%100000;
    Data|=NumData[buf/10000]<<8;
    WRITE_PORT_USHORT(v_pLEDBaseAddr1,~(Data|BIT7|BIT15));
    .....
}
```

指明代码中 BIT7 和 BIT15 的作用；

2、修改源代码，使数码管显示十六进制 0~F 并进行计数处理；

实验七 IO 接口控制实验——电机控制实验

[实验目的]

- 1、了解 WinCE 下 IO 访问机制和原理；
- 2、掌握 GPIO 的控制寄存器的控制方法；
- 3、掌握线程通信的编程方法；
- 4、熟悉 EVC 和 VS.Net 的使开发环境；

[实验仪器]

- 1、装有 Platform Builder、EVC 和 VS.Net 开发平台的 PC 机一台
- 2、XSBase270 实验开发平台一套

[实验原理]

1、硬件接口电路分析

1.1 步进电机控制接口：

XSBase270 的步进电机控制电路如图 7-1 所示，UCN4202 为四相步进电机控制器，其控制逻辑如图 7-2 所示，在 XSBase270 目标板的步进电机控制电路中，步进电机的方向控制信号和步进输入信号分别由 CPU 的通用 IO 口 GPIO84 和 GPIO83 控制，当方向控制信号为逻辑低电平时，步进电机旋转输出相位时序为 A-B-C-D，当方向控制信号为逻辑高电平时，步进电机旋转输出相位时序为 A-D-C-B；步进电机的步进输入时序由 GPIO83 控制，相应的输入逻辑如图 7-2 所示；UCN4202 输出使能信号由 PXA270 微处理器的 GPIO53 控制，低电平有效。

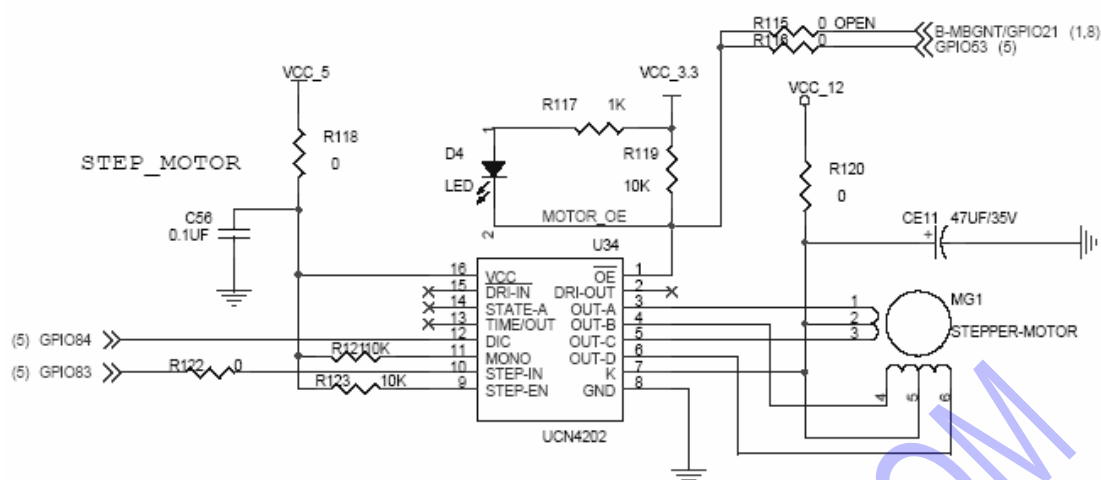


图 7-1 步进电机接口控制电路

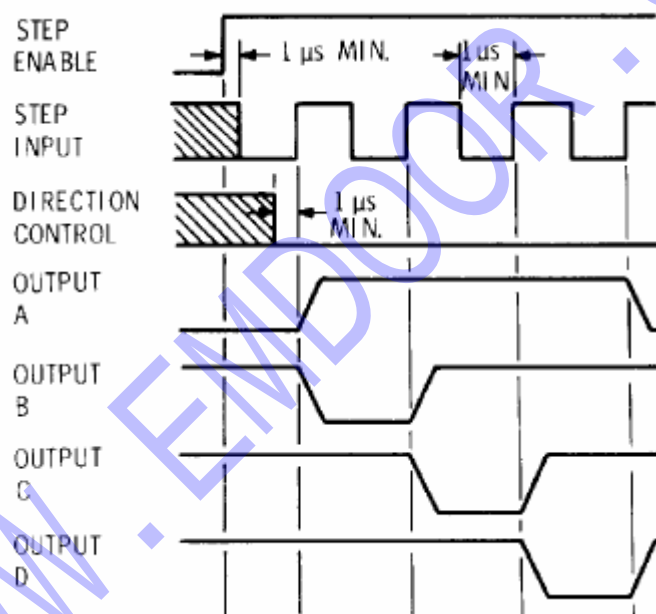


图 7-2 UCN4202 输出时序图

1.2 直流电机接口控制电路

XSBase270 的步进电机控制电路如图 7-3 所示，Q7、Q8、Q9、Q10 组成双稳态电路，当 GPIO82 为逻辑低电平、GPIO82 逻辑高电平时，Q7、Q10 导通，Q8、Q9 截止，直流电机的端口 1 为高，端口 2 为低，直流电机顺时针旋转；当 GPIO81 为逻辑低电平、GPIO81 逻辑高电平时，Q8、Q9 导通，Q7、Q10 截止，直流电机的端口 2 为高，端口 1 为低，直流电机逆时针旋转；当 GPIO81、GPIO8 都为高电平时，Q7、Q8、Q9、Q10 全部截止，电机停止运转。当 GPIO81、GPIO8 都为低电平时，双稳态电路进入随机状态。

Physical Address 0x40E0_002C										GPCR2										GPIO Controller																	
User Settings																																					
Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
	PC95	PC94	PC93	PC92	PC91	PC90	PC89	PC88	PC87	PC86	PC85	PC84	PC83	PC82	PC81	PC80	PC79	PC78	PC77	PC76	PC75	PC74	PC73	PC72	PC71	PC70	PC69	PC68	PC67	PC66	PC65	PC64					
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0					
Bits		Access		Name		Description																															
<31:0>		W		PCx		GPIO Pin Clear 'x' (where x = 64 to 95) This field configures the output level of each GPIO. 0 = Pin level unaffected. 1 = If pin configured as an output, clear pin level low (zero).																															

图 7-5 GPIO64~95 清零控制寄存器 GPCR2

2、程序实现分析

2.1 寄存器结构体定义

```

#define GPIO_81      ( 1u << 17 )
#define GPIO_82      ( 1u << 18 )
#define GPIO_83      ( 1u << 19 ) //0x00080000
#define GPIO_84      ( 1u << 20 ) //0x00100000
#define GPIO_53      ( 1u << 21 )

#define GPIO_BASE_U_VIRTUAL 0x40E00000 // GPIO Virtual Base address
// GPIO definition start
typedef struct {
    unsigned int GPLR_x; // 0x40E00000 Pin Level Registers
    unsigned int GPLR_y; // 0x40E00004
    unsigned int GPLR_z; // 0x40E00008
    unsigned int GPDR_x; // 0x40E0000C Pin Direction Registers
    unsigned int GPDR_y; // 0x40E00010
    unsigned int GPDR_z; // 0x40E00014
    unsigned int GPSR_x; // 0x40E00018 Pin Output Set Registers
    unsigned int GPSR_y; // 0x40E0001C
    unsigned int GPSR_z; // 0x40E00020
    unsigned int GPCR_x; // 0x40E00024 Pin Output Clear Registers
    unsigned int GPCR_y; // 0x40E00028
    unsigned int GPCR_z; // 0x40E0002C
    unsigned int GRER_x; // 0x40E00030 Rising Edge Detect Enable Registers
    unsigned int GRER_y; // 0x40E00034
    unsigned int GRER_z; // 0x40E00038
    unsigned int GFER_x; // 0x40E0003C Falling Edge Detect Enable Registers
    unsigned int GFER_y; // 0x40E00040
    unsigned int GFER_z; // 0x40E00044
    unsigned int GEDR_x; // 0x40E00048 Edge Detect Status Registers

```

```

unsigned int GEDR_y; // 0x40E0004C
unsigned int GEDR_z; // 0x40E00050
unsigned int GAFR0_x; // 0x40E00054 Alternate Function Registers
unsigned int GAFR1_x; // 0x40E00058
unsigned int GAFR0_y; // 0x40E0005C
unsigned int GAFR1_y; // 0x40E00060
unsigned int GAFR0_z; // 0x40E00064
unsigned int GAFR1_z; // 0x40E00068
} GPIO_REGS, *PGPIO_REGS

```

2.2 GPIO 控制的宏定义

```

#define GPIO_81_PullHigh() v_pGPIOReg->GPSR_z|=GPIO_81 //用于直流电机
#define GPIO_81_PullLow() v_pGPIOReg->GPCR_z|=GPIO_81
#define GPIO_82_PullHigh() v_pGPIOReg->GPSR_z|=GPIO_82 //用于直流电机
#define GPIO_82_PullLow() v_pGPIOReg->GPCR_z|=GPIO_82
#define GPIO_83_PullHigh() v_pGPIOReg->GPSR_z|=GPIO_83 //用于产生步进电机脉冲
#define GPIO_83_PullLow() v_pGPIOReg->GPCR_z|=GPIO_83
#define GPIO_84_PullHigh() v_pGPIOReg->GPSR_z|=GPIO_84 //用于控制步进电机方向
#define GPIO_84_PullLow() v_pGPIOReg->GPCR_z|=GPIO_84
#define GPIO_53_PullHigh() v_pGPIOReg->GPSR_y|=GPIO_53 //用于步进电机输出使能
#define GPIO_53_PullLow() v_pGPIOReg->GPCR_y|=GPIO_53

```

2.3 GPIO 初始化函数

```

BOOL CMotorDlg::InitGPIO()
{
    int retvalue;
    retvalue=0;
    if(!v_pGPIOReg){
        if(!(v_pGPIOReg=(volatile
GPIO_REGS
*)VirtualAlloc(0,0x1000,MEM_RESERVE,PAGE_NOACCESS)))
        {
            MessageBox(TEXT("VirtualAlloc() failed!\r\n"),NULL,MB_OK);
            return FALSE;
        }
    }
    else
    {
        retvalue=VirtualCopy((PVOID)v_pGPIOReg,(PVOID)(GPIO_BASE_U_VIRTUAL>>8),0x
1000,PAGE_READWRITE|PAGE_NOCACHE|PAGE_PHYSICAL);
        if(!retvalue) {
            VirtualFree((PVOID)v_pGPIOReg, 0, MEM_RELEASE);
            v_pGPIOReg = NULL;
            MessageBox(TEXT("VirtualCopy() failed!\r\n"),NULL,MB_OK);
            return FALSE;
        }
    }
}

```

```

else
{
    //set GPIO pin direction
    v_pGPIOReg->GPDR_z |= GPIO_81;
    v_pGPIOReg->GPDR_z |= GPIO_82;
    v_pGPIOReg->GPDR_z |= GPIO_83;
    v_pGPIOReg->GPDR_z |= GPIO_84;
    v_pGPIOReg->GPDR_y |= GPIO_53;

    g_uPWMRunTimes=0xFFFFFFFF;
    g_DCERunTime=1000;
    g_Positive=0xFFFFFFFF;

    GPIO_84_PullLow();
    GPIO_83_PullLow();
    GPIO_81_PullHigh();
    GPIO_82_PullHigh();
    GPIO_53_PullHigh();
    return TRUE;
}
}
return TRUE;
}

```

2.4 控制步进电机运转线程

```

DWORD WINAPI CMotorDlg::StepMotorThread(LPVOID lpParam)    //步进电机驱动
{
    CMotorDlg *pDlg=(CMotorDlg*)lpParam;
    CWait waitTime;
    while(1) {
        if(WaitForSingleObject(pDlg->g_hPWMOpenEvent, INFINITE) == WAIT_OBJECT_0)
        {
            GPIO_53_PullLow();// Step Motor Power on
            if(pDlg->g_ZLDJPositive)
                GPIO_84_PullLow();
            else
                GPIO_84_PullHigh();
            while(pDlg->g_uPWMRunTimes)
            {
                if(pDlg->g_uPWMRunTimes!=0xFFFFFFFF)
                    pDlg->g_uPWMRunTimes--;
                GPIO_83_PullHigh();
                waitTime.usWait(pDlg->g_HighTimeA);
                GPIO_83_PullLow();
                waitTime.usWait(pDlg->g_LowTimeA);
            }
        }
    }
}

```

```

    }
    GPIO_84_PullLow();
}
GPIO_53_PullHigh();
}
return 0;
}

```

2.5 直流电机控制函数

```

void CMotorDlg::OnDCMotorStart()
{
    CWait wait;
    //SetEvent(g_hDCEOpenEvent);
    UpdateData(TRUE);
    if(m_DCMotorDir)    //Positive Run
    {
        GPIO_81_PullLow();
        GPIO_82_PullHigh();
        if(!m_DCContinue.GetCheck()){ //定时运行
            Sleep(m_DCRunTime);//
            //wait.msWait(m_DCRunTime);
            GPIO_81_PullHigh();
            GPIO_82_PullHigh();
        }
    }
    else
    {
        GPIO_81_PullHigh();
        GPIO_82_PullLow();
        if(!m_DCContinue.GetCheck())//定时运行
        {
            Sleep(m_DCRunTime);
            //wait.msWait(m_DCRunTime);
            GPIO_81_PullHigh();
            GPIO_82_PullHigh();
        }
    }
}

```

[实验内容]

- 1、了解在 Windows CE 下访问一般硬件寄存器的方法；
- 2、掌握 EVC 下对硬件 I/O 寄存器的访问方法，以及使用 EVC 对硬件设备编程的一般方法；
- 3、掌握 GPIO 控制寄存器的使用方法；

4、了解线程编程方法；

5、使用 EVC 或 VS.net 编写对 XSBASE270 目标板步进电机和直流电机的控制方法。

[实验步骤]

第一步：连接好实验系统，打开实验箱电源。

第二步：打开电机控制的工程文件 Motor.vcw，进行编译：

第三步：编译该代码，点击运行按钮，这样程序就会下载到 XSBASE270 目标板上运行。

运行界面如图 6-4 所示。

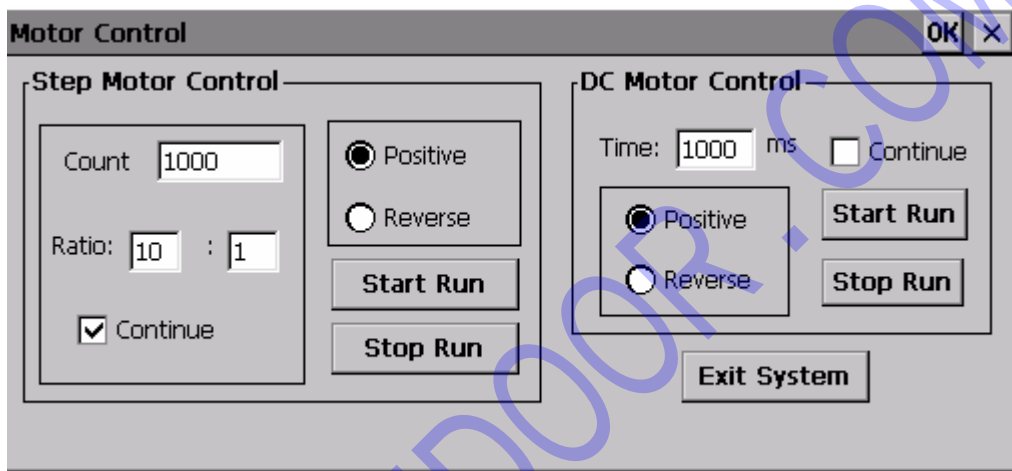


图 7-6 电机控制实验运行界面

第三步：电机控制程序操作过程：

- Step Motor Control 表示对 XSBASE270 目标板的步进电机进行控制操作，按“Start Run”按钮，启动步进电机，当“Positive”表正转，“Reverse”表示反转，当“Continue”被勾选后，步进电机将连续运转，直到按下“Stop Run”按钮。“Count”表示步进电机的运行脉冲数，在“Continue”没有被选中时，步进电机运行到设定的脉冲数后停止运行；“Ratio”表示 PWM 的占空比。
- DC Motor Control 表示对目标板的直流电机运行进行控制，按“Start Run”按钮，启动步进电机，当“Positive”表正转，“Reverse”表示反转，当“Continue”被勾选后，电机将连续运转，直到按下“Stop Run”按钮。“Time”表示直流电机的运行时间，在“Continue”没有被选中时，直流电机运行到设定时间后停止运行；

第四步：修改应用程序，使电机正转运行到设定时间后自动进入反转运行：

[习题与思考题]

1、在 Motordef.h 文件中，有下列定义，根据 PXA270 用户手册，分析定义的作用

```
// OSSR Bits
#define OST_BASE_PHYSICAL 0x40A00000
#define OSSR_M0 (0x1 << 0)
#define OSSR_M1 (0x1 << 1)
#define OSSR_M2 (0x1 << 2)
#define OSSR_M3 (0x1 << 3)
#define TIMERTICK 4
// OIER Bits
#define OIER_E0 (0x1 << 0)
#define OIER_E1 (0x1 << 1)
#define OIER_E2 (0x1 << 2)
#define OIER_E3 (0x1 << 3)
// OST (os timer)
typedef struct
{
    unsigned long    osmr0;        //OS timer match register 0
    unsigned long    osmr1;        //OS timer match register 1
    unsigned long    osmr2;        //OS timer match register 2
    unsigned long    osmr3;        //OS timer match register 3
    unsigned long    oscr;         //OS timer counter register
    unsigned long    ossr;         //OS timer status register
    unsigned long    ower;         //OS timer watchdog enable register
    unsigned long    oier;         //OS timer interrupt enable register
} OST_REGS, *POST_REGS;
```

2、EVC 或 VS.net 中的定时器最小定时为 1 毫秒 (ms)；如果实验中需要微秒 (us) 级定时，应该怎样进行处理？

3、编写一个控制直流电机的线程函数；

实验八 动态链接库 (DLL) 实验

[实验目的]

- 1、了解 WinCE 动态链接库的基本原理；
- 2、掌握 WinCE 动态链接库程序的编程方法；
- 3、掌握采用静态和动态调用动态链接库方法
- 4、熟悉 EVC 和 VS.Net 的使开发环境；

[实验仪器]

- 1、装有 Platform Builder、EVC 和 VS.Net 开发平台的 PC 机一台
- 2、XSBase270 实验开发平台一套

[实验原理]

1 动态链接库简介

1.1 DLL 概述

动态链接库 (Dynamic Link Library, 简称 DLL) 是一些编译过的可执行的程序模块, 可以在应用程序中或其他 DLL 中被调用。DLL 的应用非常广泛, 可以实现多个应用程序的代码和资源共亨, 是 WinCE 程序设计中的一个非常重要的组成部分。

DLL 设计程序的优点:

- 共享代码、资源和数据。DLL 作为一种基于 Windows 的程序模块, 不仅可以包含可执行的代码, 还可以包括数据和各种资源等, 扩大了库文件的使用范围;
- 可将系统模块化, 方便升级。
- 隐藏实现的细节。在某些情况下, 用户可能想隐藏例程的细节, 可以采用 DLL 来实现, DLL 的例程可以被应用程序访问, 而不显示其中代码的细节。
- DLL 与语言无关

1.2 DLL 的调用

不论使用何种语言对编译好的 DLL 进行调用时, 基本上都有两种调用方式, 即静态调用方式和动态调用方式。静态调用方式由编译系统完成对 DLL 的加载和应用程序结束时 DLL 卸载的编码 (如还有其它程序使用该 DLL, 则 Windows 对 DLL 的应用记录减 1, 直到所有相关程序都结束对该 DLL 的使用时才释放它), 简单实用, 但不够灵活, 只能满足一般要求。动态调用方式是由编程者用 API 函数加载和卸载 DLL 来达到调用 DLL 的目的, 使用上较复杂, 但能更加有效地使用内存, 是编制大型应用程序时的重要方式。

(1) DLL 的静态调用

DLL 的静态调用由编译系统完成对 DLL 的加载和应用程序结束时 DLL 卸载, 在 EVC 或 VS.net 中静态调用 DLL 非常简单, 首先将动态链接库的.LIB 文件加入到应用程序的工程

中，然后在使用 DLL 中的函数文件里引用 DLL 的头文件（.h）即可。

当开发人员通过静态方式编译并生成应用程序时，应用程序中的调用函数与 LIB 文件中的导出符号相匹配，这些符号或标示进入到生成的 EXE 文件中。当应用程序运行过程中需要加载 DLL 文件时，操作系统将根据这些信息查寻并加载 DLL，然后通过符号或标示实现对 DLL 函数的动态链接。当加载应用程序的 EXE 文件时，所有被应用程序调用的 DLL 文件都被加载到内存中，这时可执行程序直接通过函数名调用 DLL 的输出函数，其调用方法与调用程序内部函数相同。

（2）DLL 的动态调用

动态调用方式是由编程者用 API 函数加载和卸载 DLL 来达到调用 DLL 的目的，动态调用是指在应用程序中使用 LoadLibrary 函数或 MFC 提供的 AfxLoadLibrary 函数显式调用自己所需要的动态链接库，动态链接库的文件名就是上面两个函数的参数，然后在使用 GetProcAddress() 函数获取所需要引入的函数。完成上述操作后，应用程序可以调用引入的函数。在应用程序退出之前，应该使用 FreeLibrary 函数或 MFC 提供的 AfxFreeLibrary 函数来释放动态链接库。

2 动态链接的实现

2.1 动态链接库的创建

（1）新建一个基于 Smart Device 的 MFC Smart Device DLL，将项目名称设为 DllTest，如图 8-1 所示。

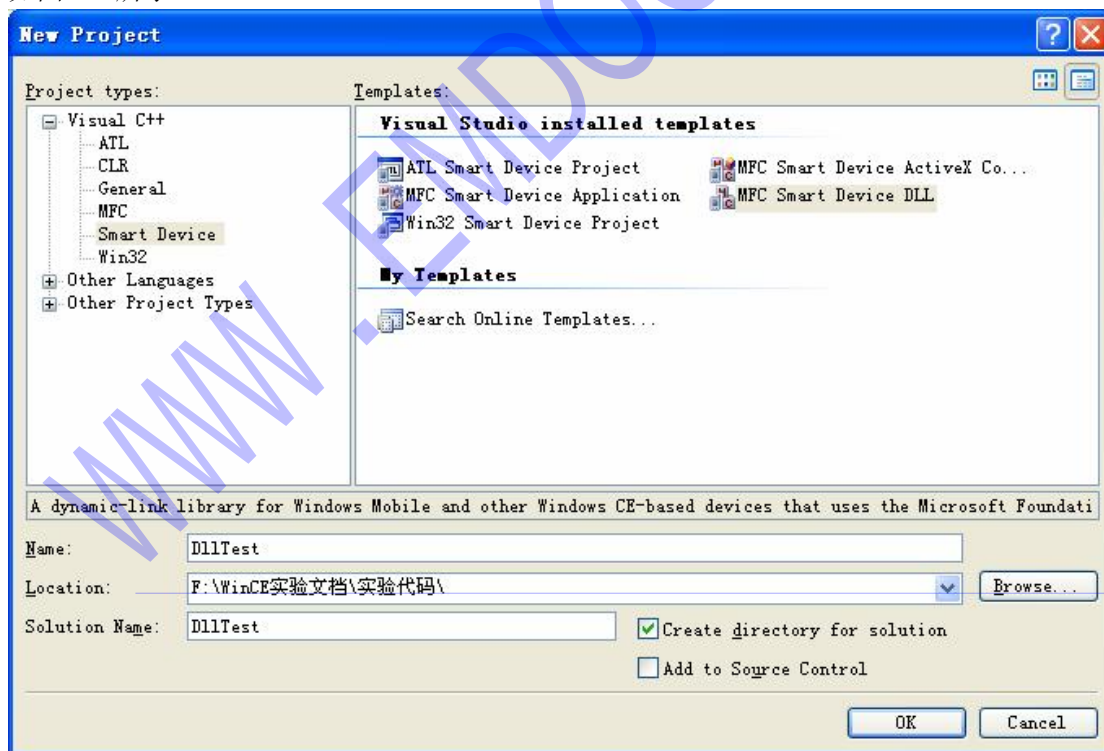


图 8-1 新建 DLL 项目

单击 OK，此时将出现 DLL 的支持平台对话框，选择所支持的平台，由于本实验的 DLL 动态库使用 XBASE270 目标板，所以选择 XBASE270 平台。如图 8-2 所示。

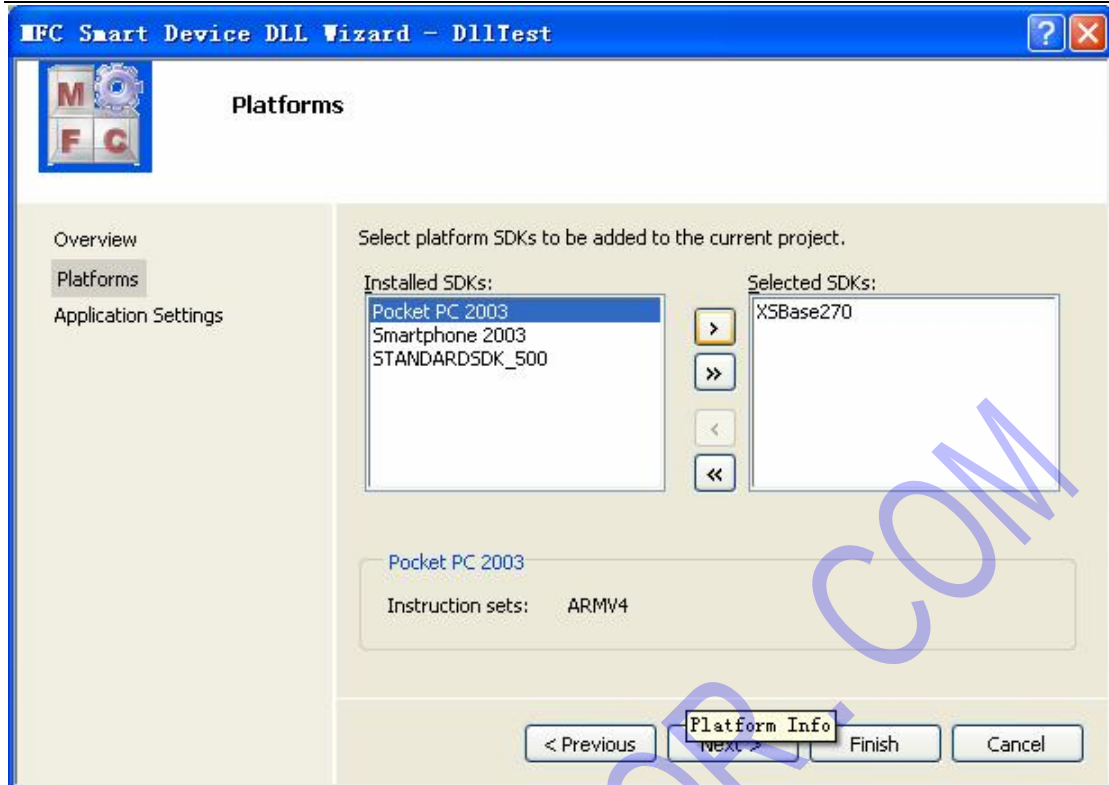


图 8-2 DLL 项目的平台选择

单击 Next，将出现 DLL 的类型选择（如图 8-3 所示），选中 MFC Extension DLL，按 Finish 可完成 DLL 向导工作。此时向导将自动生成 DLL 基本框架。

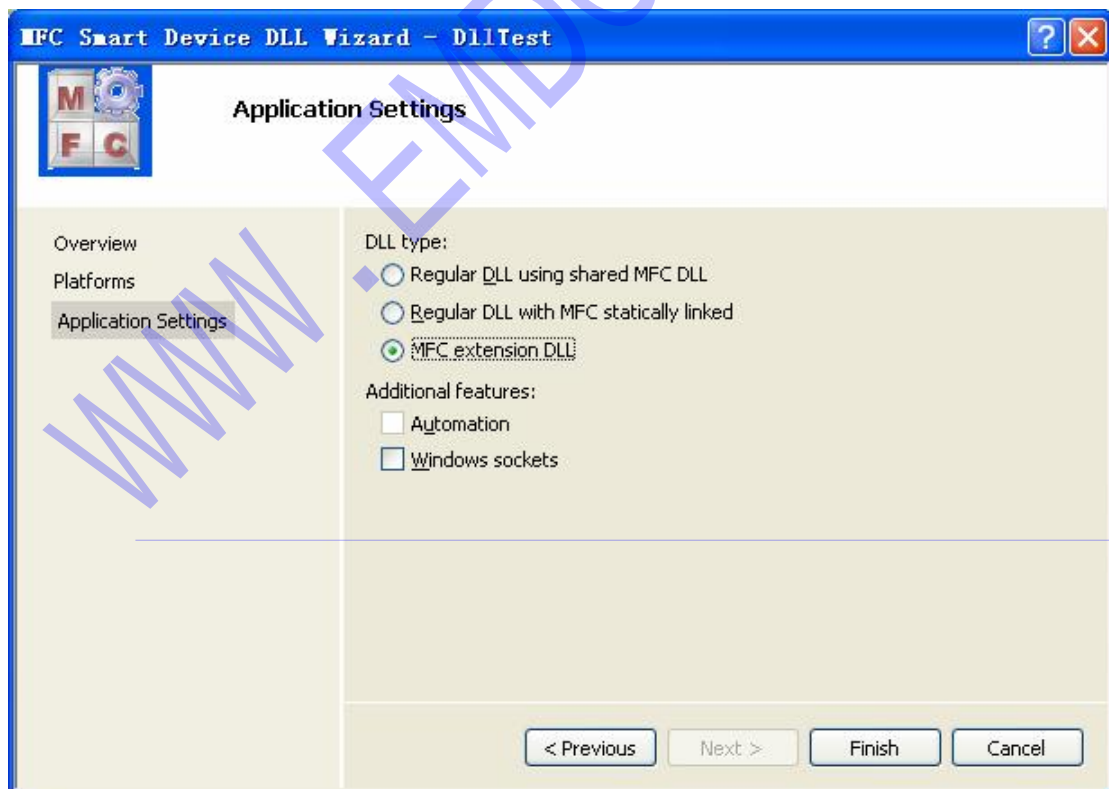


图 8-3 DLL 类型选择对话框

(2) 每个 DLL 必须有一个入口点，这就如同使用 C 语言编写的应用程序必须有一个 main 函数一样，DllMain 是一个默认的入口函数，它负责初始化和结束工作，当一个新的线程访

问 DLL 时，都会调用 DllMain 函数。

打开利用向导生成的 DllTest.cpp 文件，即可看到 DllMain 函数的实现。

```
extern "C" BOOL APIENTRY DllMain(HANDLE hInstance, DWORD dwReason, LPVOID lpReserved)
{
    // Remove this if you use lpReserved
    UNREFERENCED_PARAMETER(lpReserved);
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        TRACE0("DllTest.DLL Initializing!\n");
        // Extension DLL one-time initialization
        if (!AfxInitExtensionModule(DllTestDLL, reinterpret_cast<HMODULE>(hInstance)))
            return 0;
        new CDynLinkLibrary(DllTestDLL);
    }
    else if (dwReason == DLL_PROCESS_DETACH)
    {
        TRACE0("DllTest.DLL Terminating!\n");
        // Terminate the library before destructors are called
        AfxTermExtensionModule(DllTestDLL);
    }
    return 1;    // ok
}
```

(3) 输出函数的实现方法

DLL 中导出函数的声明有两种方式：一种方法在函数名称声明中加上修饰符 `__declspec(dllexport)`，表示输出，此外，还有一种修饰符 `extern "C" __declspec(dllexport)`，也表示输出，而且该类 DLL 不仅可以被 C++ 调用，还可以被 C 调用。在 C++ 下定义 C 函数时，需要加上 `extern "C"` 关键字。下面为 DLL 测试程序的输出函数的声明。（在 DllTest.h 文件中）。

```
#ifdef DLLTEST_EXPORTS
#define DLLTEST_API __declspec(dllexport)
#else
#define DLLTEST_API __declspec(dllimport)
#endif
extern "C"
{
    void DLLTEST_API SetLight(unsigned char data);
    void DLLTEST_API Ledshift(int shiftDir, int count);
    void DLLTEST_API LedControl(DWORD dwCode, int shiftDir=-1);
}
```

导出函数另外一种方式是采用模块定义(.def) 文件声明，.def 文件为链接器提供了有关被链接程序的导出、属性及其他方面的信息。（DllTest.def 的内容）

```
; DllTest.def : Declares the module parameters for the DLL.
LIBRARY
    "DllTest"
```

EXPORTS

SetLight
Ledshift
LedControl

采用模块定义.def 导出函数声明, 如果要求导出函数能够被 C 语言调用, 必须在函数的实现前加 `extern "C"` 进行修饰。

```
extern "C" void SetLight(unsigned char data)
{
    if(m_bShiftRunning)
    {
        m_bStop=TRUE;
        Sleep(500);
    }
    *pLightReg=~data;
}
```

2.2 动态链接库的调用步骤

(1) 静态调用 DLL 的步骤

利用 VS2005.net 生成一个 DLL 调用测试应用程序 TestDll.sln, 并将上述编译好的 DllTest.dll 和 DllTest.lib 文件拷贝到本工程的目录下。然后使用 VS2005.net 的 Project|DllTest Properties 属性设置中, 选中 Linker, 在 Linker 的 Input 选项的 Additional Dependencies 中输入 DllTest.lib 如图 8-4 所示

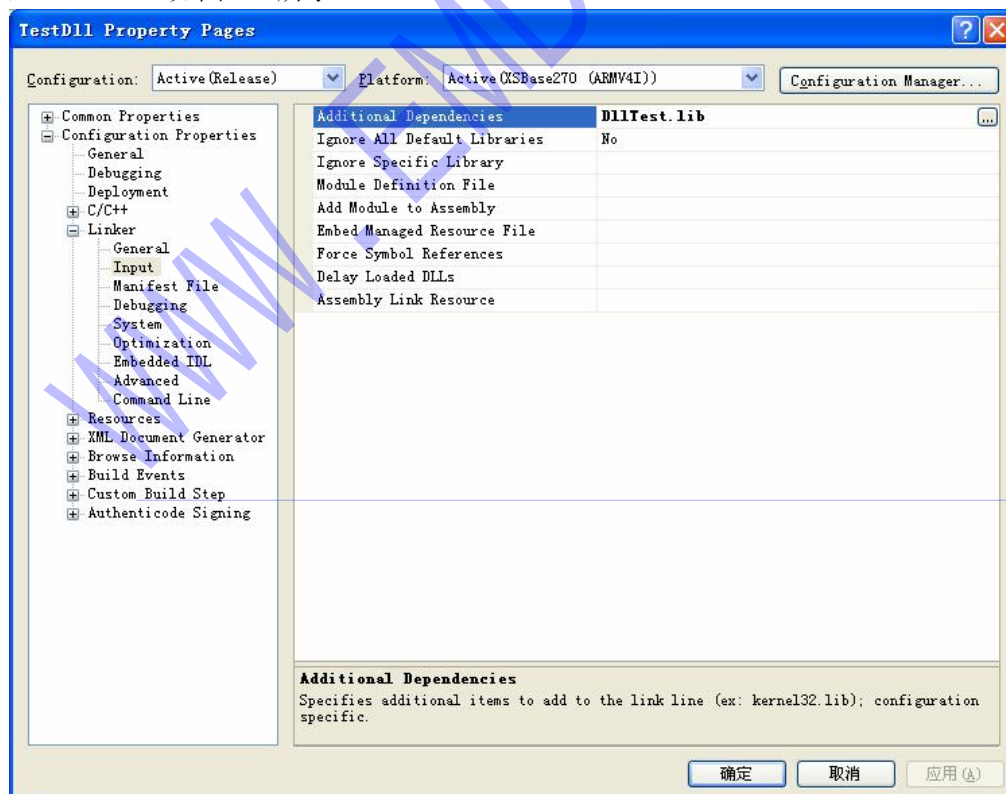


图 8-4 静态调用操作界面

如果在 DLL 的函数的导出采用头文件的实现方法, 必须将 DLL 的头文件 DllTest.h 拷

贝到在调用 DLL 的工程中，并在实现文件中引用 DllTest.h 文件，代码如下：

```
#include "DllTest.h"
```

这样就可以使用 DLL 的导出函数。

如果采用模块 (.def) 导出 DLL 中实现函数，则必须在调用 DLL 的实现文件声明导入函数，代码如下：

```
extern "C" void __declspec(dllimport) SetLight(unsigned char data);
extern "C" void __declspec(dllimport) Ledshift(int shiftDir, int count);
extern "C" void __declspec(dllimport) LedControl(DWORD dwCode, int shiftDir=-1);
```

(2) 动态调用 DLL 的步骤

动态调用方式是使用 LoadLibrary API 函数加载 DLL，然后在使用 GetProcAddress() 函数获取所需要引入的函数。

具体实现方式：

- 添加 DllTest 动态链接库工程的实现函数的定义，代码如下：

```
typedef void (*pLedControl) (DWORD, int);
typedef void (*pSetLight) (unsigned char);
typedef void (*pLedShift) (int, int);
```

- 利用 LoadLibrary API 函数动态加载动态链接库，代码的黑体部分：

```
BOOL CTestDllDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    SetIcon(m_hIcon, TRUE); // Set big icon
    SetIcon(m_hIcon, FALSE); // Set small icon
    hModule=LoadLibrary(_T("DllTest.dll"));
    if(hModule==NULL)
        MessageBox(_T("Load Dll file failed"), _T("System Information"), MB_OK|MB_ICONERROR);
    return TRUE; // return TRUE unless you set the focus to a control
}
```

- 利用 GetProcAddress API 函数获取需要引用的函数，代码如下：

```
void CTestDllDlg::OnBnClickedbtnstop()
{
    pLedControl LedControl = (pLedControl)GetProcAddress (hModule, _T("LedControl"));
    if(LedControl==NULL)
        MessageBox(_T("Load LedControl function failed"), _T("System Information"), MB_OK|MB_ICONERROR);
    else
        LedControl(2, -1);
}
```

[实验内容]

- 1、掌握 Windows CE 下 DLL 编程方法；
- 2、掌握静态和动态调用 DLL 的步骤和方法
- 3、使用 EVC 或 VS.net 编写对 XSBASE270 目标板硬件进行控制的动态链接库。

[实验步骤]

- 第一步：**连接好实验系统，打开实验箱电源；
- 第二步：**打开动态链接库工程文件 DllTest.sln，进行编译；然后点击运行按钮将 DllTest.dll 文件下载到目标板中。（点击运行按钮自动将所需要的文件下载）。
- 第三步：**打开静态调用 DLL 的测试工程文件 TestDll.sln，并将编译好的 DllTest.dll 和 DllTest.lib 文件复制到本工程目录下，然后进行工程属性设置；
- 第四步：**编译该工程文件，并把目标代码复制到 XSBase270 目标板上运行，运行界面如图 8-5 所示；

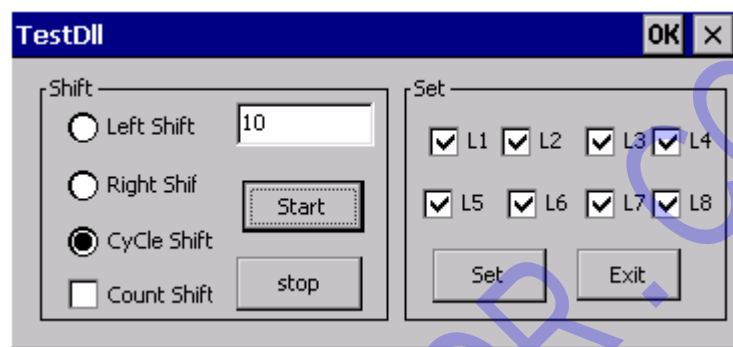


图 8-5 DLL 调用测试程序运行界面

- 第五步：**打开动态调用 DLL 的测试工程文件 TestDll.sln，并将编译好的 DllTest.dll 和 DllTest.lib 文件复制到本工程目录下；
- 第六步：**编译该动态调用 DLL 的工程文件，并把目标代码复制到 XSBase270 目标板上运行，运行界面如图 8-5 所示；
- 第七步：**程序界面的操作
- shift 操作：Left Shift 和 Right Shift 分别表示 LED 发光二极管的左移和右移；Cycle Shift 表示循环移动，按 Start 按钮启动；当 Count Shift 选中时，二极管的移动次数由文本框内的数值决定；
- Set 操作：L1~L8 对应目标板 8 个发光二极管，勾选后按 Set 按钮，相应的发光二极管点亮；

[习题与思考题]

- 1、比较动态和静态调用 DLL 的两种方法的区别，并分析采用动态调用 DLL 的好处？
- 2、如果 DLL 的导出函数既能被 C 语言调用，也能被 C++ 语言调用，在采用用模块定义时（.def）应该怎样定义导出函数？
- 3、如果需要导出 DLL 中的一个变量，应该怎样定义？

实验九 无线网络通信实验

[实验目的]

- 1、掌握 Socket 编程方法;
- 2、掌握 WinCE 无线网络的配置过程;
- 3、了解 TCP/IP 协议
- 4、熟悉 EVC 和 VS.Net 的开发环境;

[实验仪器]

- 1、装有 Platform Builder、EVC 和 VS.Net 开发平台的 PC 机一台
- 2、XSBase270 实验开发平台一套
- 3、SL-2511 CF 无线网卡一块
- 4、无线 AP 一套

[实验原理]

1、软件原理

1.1 服务器端操作 socket (套接字)

(1) 在初始化阶段调用 WSStartup()

此函数在应用程序中初始化 Windows Sockets DLL，只有此函数调用成功后，应用程序才可以再调用其他 Windows Sockets DLL 中的 API 函数。

(2) 建立 Socket

初始化 WinSock 的动态连接库后，需要在服务器端建立一个监听的 Socket，为此可以调用 `Socket()` 函数用来建立这个监听的 Socket，并定义此 Socket 所使用的通信协议。此函数调用成功返回 Socket 对象，失败则返回 `INVALID_SOCKET` (调用 `WSAGetLastError()` 可得知原因，所有 WinSocket 的函数都可以使用这个函数来获取失败的原因)。

```
SOCKET PASCAL FAR socket( int af, int type, int protocol )
```

参数: `af`: 目前只提供 `PF_INET(AF_INET)`;

`type`: Socket 的类型 (`SOCK_STREAM`、`SOCK_DGRAM`);

`protocol`: 通讯协定(如果使用者不指定则设为 0);

如果要建立的是遵从 TCP/IP 协议的 socket，第二个参数 `type` 应为 `SOCK_STREAM`，如为 UDP (数据报) 的 socket，应为 `SOCK_DGRAM`。

(3) 绑定端口

接下来要为服务器端定义的这个监听的 Socket 指定一个地址及端口 (Port)，这样客

户端才知道待会要连接哪一个地址的哪个端口，为此我们要调用 `bind()` 函数，该函数调用成功返回 0，否则返回 `SOCKET_ERROR`。

```
int PASCAL FAR bind( SOCKET s, const struct sockaddr FAR *name, int namelen );
```

参 数: s: Socket 对象名;

name: Socket 的地址值，这个地址必须是执行这个程式所在机器的 IP 地址;

namelen: name 的长度;

如果使用者不在意地址或端口的值，那么可以设定地址为 `INADDR_ANY`，及 Port 为 0，Windows Sockets 会自动将其设定适当之地址及 Port (1024 到 5000 之间的值)。此后可以调用 `getsockname()` 函数来获知其被设定的值。

(4) 监听

当服务器端的 Socket 对象绑定完成之后，服务器端必须建立一个监听的队列来接收客户端的连接请求。`listen()` 函数使服务器端的 Socket 进入监听状态，并设定可以建立的最大连接数 (目前最大值限制为 5，最小值为 1)。该函数调用成功返回 0，否则返回 `SOCKET_ERROR`。

```
int PASCAL FAR listen( SOCKET s, int backlog );
```

参 数: s: 需要建立监听的 Socket;

backlog: 最大连接个数;

服务器端的 Socket 调用完 `listen()` 后，如果此时客户端调用 `connect()` 函数提出连接申请的话，Server 端必须再调用 `accept()` 函数，这样服务器端和客户端才算正式完成通信程序的连接动作。

(5) 服务器端接受客户端的连接请求

当 Client 提出连接请求时，Server 端 hwnd 视窗会收到 Winsock Stack 送来我们自定义的一个消息，这时，我们可以分析 `lParam`，然后调用相关的函数来处理此事件。为了使服务器端接受客户端的连接请求，就要使用 `accept()` 函数，该函数新建一 Socket 与客户端的 Socket 相通，原先监听之 Socket 继续进入监听状态，等待他人的连接要求。该函数调用成功返回一个新产生的 Socket 对象，否则返回 `INVALID_SOCKET`。

```
SOCKET PASCAL FAR accept( SOCKET s, struct sockaddr FAR *addr, int FAR *addrlen );
```

参数: s: Socket 的识别码;

addr: 存放来连接的客户端的地址;

addrlen: addr 的长度

(6) 结束 socket 连接

结束服务器和客户端的通信连接是很简单的，这一过程可以由服务器或客户机的任一端启动，只要调用 `closesocket()` 就可以了，而要关闭 Server 端监听状态的 socket，同样也是利用此函数。另外，与程序启动时调用 `WSAStartup()` 整数相对应，程式结束前，需要调用 `WSACleanup()` 来通知 Winsock Stack 释放 Socket 所占用的资源。这两个函数都是调用成功返回 0，否则返回 `SOCKET_ERROR`。

```
int PASCAL FAR closesocket( SOCKET s );
```

参 数: s: Socket 的识别码;

```
int PASCAL FAR WSACleanup( void );
```

参 数: 无

1.2 客户端 Socket 的操作

(1) 建立客户端的 Socket

客户端应用程序首先也是调用 `WSAStartup()` 函数来与 Winsock 的动态连接库建立关系，然后同样调用 `socket()` 来建立一个 TCP 或 UDP socket（相同协定的 sockets 才能相通，TCP 对 TCP，UDP 对 UDP）。与服务器端的 socket 不同的是，客户端的 socket 可以调用 `bind()` 函数，由自己来指定 IP 地址及 port 号码；但是也可以不调用 `bind()`，而由 Winsock 来自动设定 IP 地址及 port 号码。

(2) 提出连接申请

客户端的 Socket 使用 `connect()` 函数来提出与服务器端的 Socket 建立连接的申请，函数调用成功返回 0，否则返回 `SOCKET_ERROR`。

```
int PASCAL FAR connect( SOCKET s, const struct sockaddr FAR *name, int namelen );
```

参 数: s: Socket 的识别码;

name: Socket 想要连接的对方地址;

namelen: name 的长度

1.3 数据的传送

虽然基于 TCP/IP 连接协议（流套接字）的服务是设计客户机/服务器应用程序时的主流标准，但有些服务也是可以通过无连接协议（数据报套接字）提供的。先介绍一下 TCP socket 与 UDP socket 在传送数据时的特性：Stream (TCP) Socket 提供双向、可靠、有次序、不重复的资料传送。Datagram (UDP) Socket 虽然提供双向的通信，但没有可靠、有次序、不重复的保证，所以 UDP 传送数据可能会收到无次序、重复的资料，甚至资料在传输过程中出现遗漏。由于 UDP Socket 在传送资料时，并不保证资料能完整地送达对方，所以绝大多数应用程序都是采用 TCP 处理 Socket，以保证资料的正确性。一般情况下 TCP Socket 的数据发送和接收是调用 `send()` 及 `recv()` 这两个函数来达成，而 UDP Socket 则是用 `sendto()` 及 `recvfrom()` 这两个函数，这两个函数调用成功发挥发送或接收的资料的长度，否则返回 `SOCKET_ERROR`。

```
int PASCAL FAR send( SOCKET s, const char FAR *buf, int len, int flags );
```

参数: s: Socket 的识别码

buf: 存放要传送的资料的暂存区

len buf: 的长度

flags: 此函数被调用的方式

对于 Datagram Socket 而言，若是 datagram 的大小超过限制，则将不会送出任何资料，并会传回错误值。对 Stream Socket 言，Blocking 模式下，若是传送系统内的储存空间不够存放这些要传送的资料，`send()` 将会被 block 住，直到资料送完为止；如果该 Socket 被设定为 Non-Blocking 模式，那么将视目前的 output buffer 空间有多少，就送出多少资料，并不会被 block 住。flags 的值可设为 0 或 `MSG_DONTROUTE` 及 `MSG_OOB` 的组合。

```
int PASCAL FAR recv( SOCKET s, char FAR *buf, int len, int flags );
```

参数: s: Socket 的识别码

buf: 存放接收到的资料的暂存区

len buf: 的长度

flags: 此函数被调用的方式

1.4 TCP 服务器端和客户端编程

TCP 服务器端编程的一般流程为：首先 TCP 服务器端调用 `socket` 函数建立一个流式套接字，然后调用 `bind` 函数绑定本地地址，接着调用 `Listen` 函数进行监听客户端连接，一旦监听到客户端连接请求后，服务器套接字将调用 `Accept` 函数接受客户端连接请求，并建立连接，同时服务器端会新加一个单独的套接字与客户端进行通讯。

对于 TCP 客户端，客户端首先调用 `socket` 函数建立流式套接字，然后调用 `connect` 函数，请求与服务器端 TCP 建立连接，成功建立连接后，即可通服务器端进行通讯。TCP 服务器端和客户端的流程如图 9-1 所示。

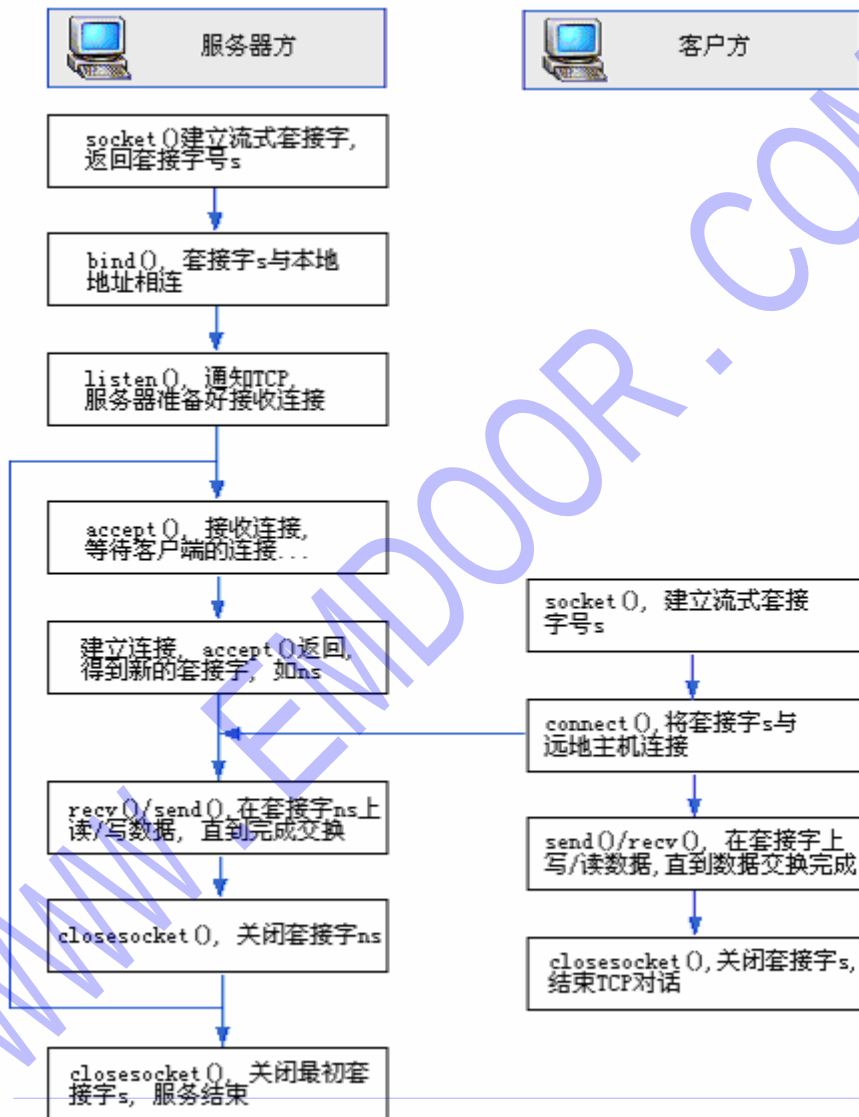


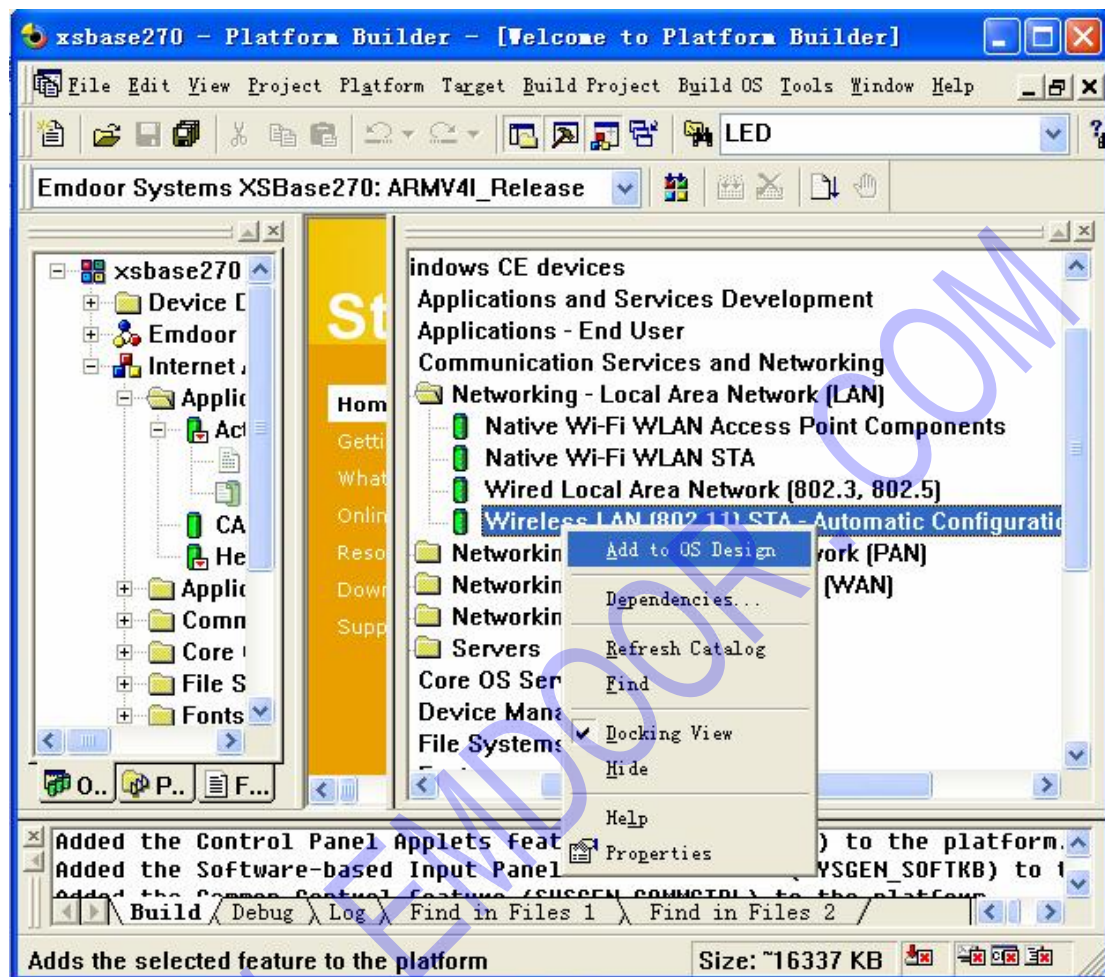
图 9-1 TCP 服务器端和客户端流程图

2、无线网卡的配置

为了开发平台支持 CF 无线网卡，在编译 WinCE 操作系统映像文件时，必须对 CF 无线网卡的支持进行配置，具体配置如图 9-2 所示。

在 WinCE 中没有 SL-2511 无线网卡的驱动，必须要安装 SL-2511CF 无线网卡的驱动程序（可以从网上下载），安装完成后，驱动程序会自动下载到目标板。（具体驱动程序文件为

WLANNDS.dll)。当无线网卡插入到目标板上，系统弹出要求输入 CF 无线网卡的驱动程序对话框。(如图 9-3 所示)。如果驱动程序存在，按照图 9-3 输入驱动程序文件后，无线网络便进入自动配置和连接过程。(如图 9-4)



◆ 图 9-2 CF 无线网卡的配置

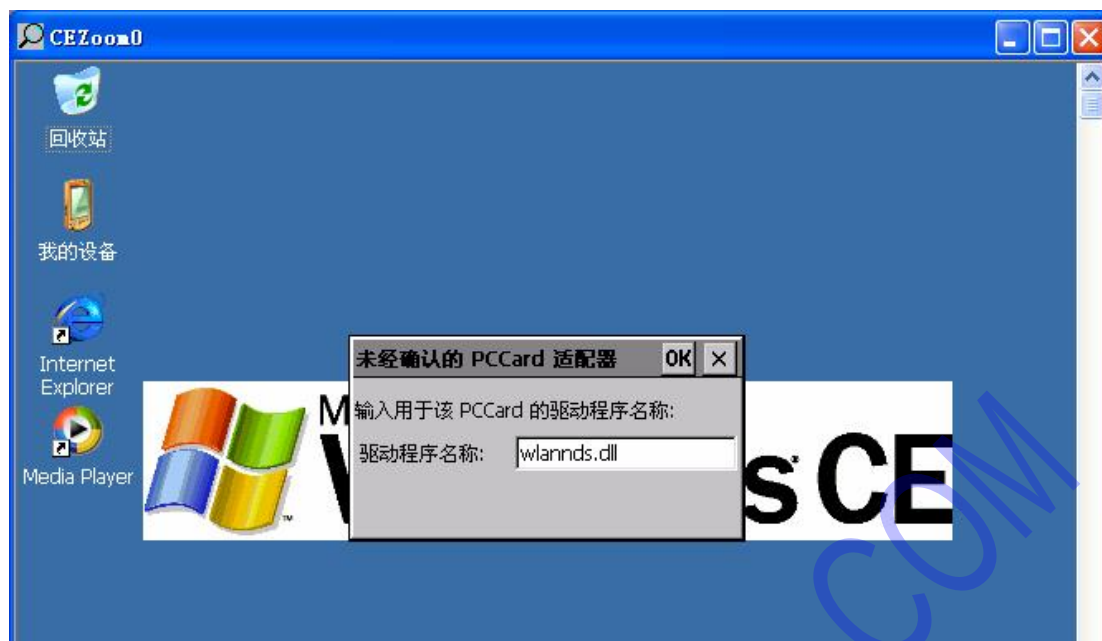


图 9-3 CF 无线网卡驱动输入对话框

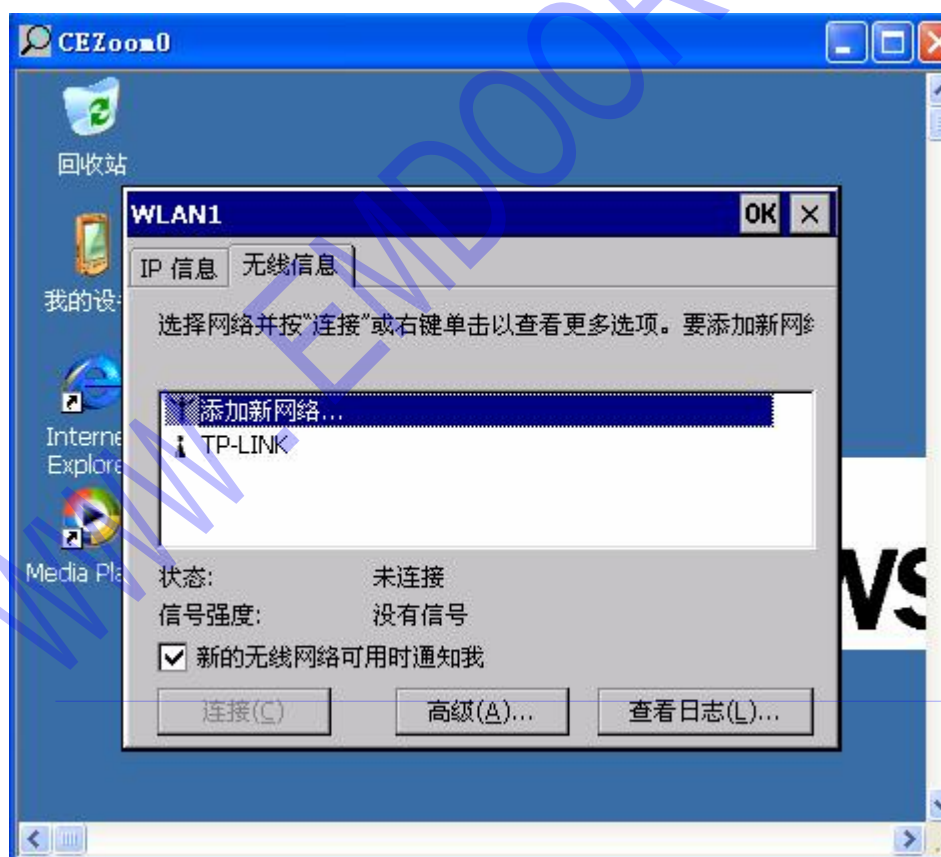


图 9-4 CF 无线网络的连接

[实验内容]

- 1、无线网络的配置;
- 2、TCP/IP 无线网络数据传输的编程方法;
- 3、了解面向连接的网络编程方法

[实验步骤]

第一步: 连接好实验系统, 打开实验箱电源;

第二步: 插入无线网卡, 安装驱动程序

第三步: 利用 VS2005.net 打开 TCP 客户端程序 TcpClient.sln。进行编译: 点击运行按钮, 将 TCP 客户端测试程序下载到 XSBase270 目标板运行, 运行界面如图 9-5 所示;

第四步: 利用 VS2005.net 打开 TCP 服务器端程序 TcpSever.sln。进行编译: 点击运行按钮, 在 PC 机上运行的 TCP 服务器端测试程序界面如图 9-6 所示。

第五步: 连上无线 AP, 进行无线通信测试

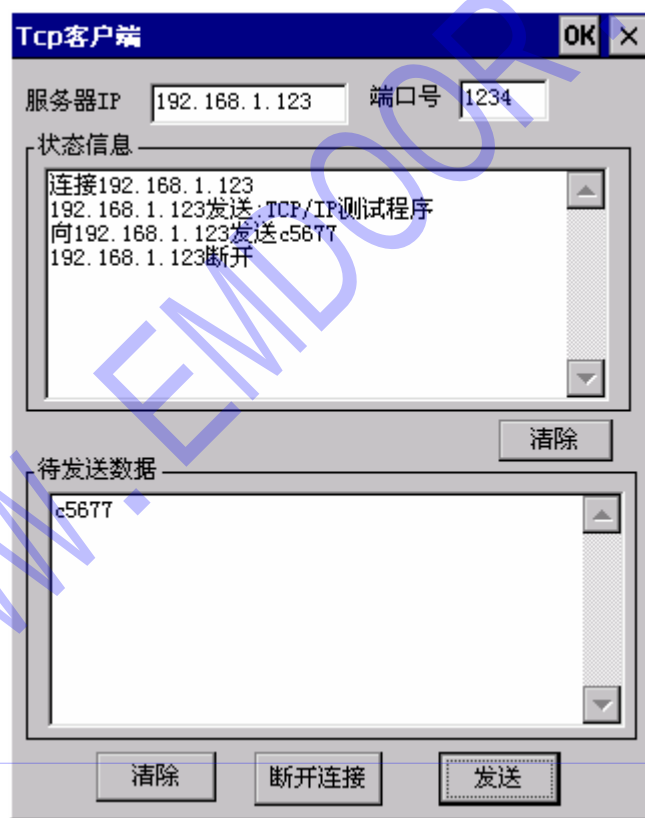


图 9-5 TCP 客户端测试程序运行界面

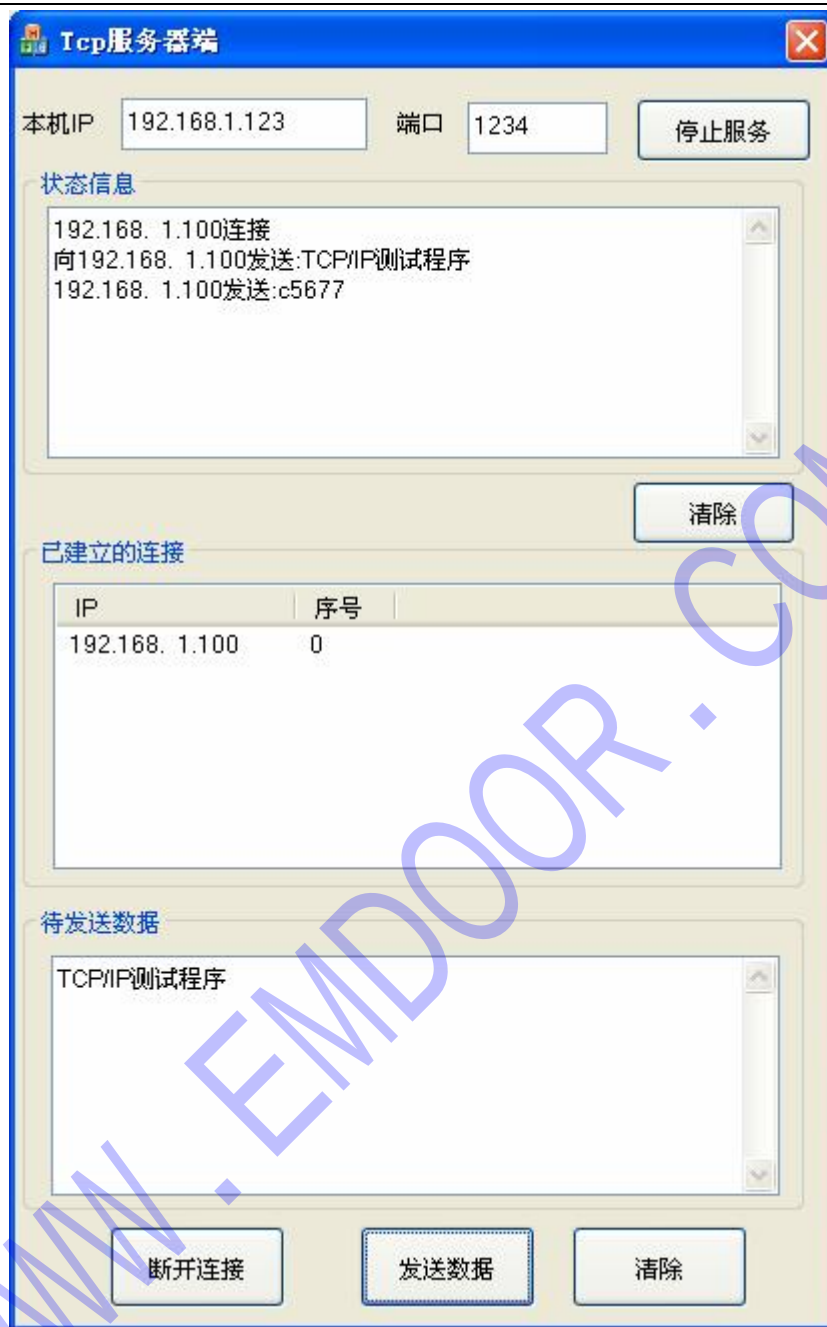


图 9-6 TCP 客户端测试程序运行界面

[习题与思考题]

- 1、修改 TCP 服务器端程序，使其能在目标板上运行。
- 2、修改 TCP 客户端程序，使其在 PC 端运行，然后完成无线通信实验

实验十 UDP 和 Ping 实验

[实验目的]

- 1、了解 UDP 网络通信的基本内容；
- 2、掌握 PING 实现的基本原理；
- 3、了解 TCP/IP 和 ICMP 协议
- 4、熟悉 EVC 和 VS.Net 的开发环境；

[实验仪器]

- 1、装有 Platform Builder、EVC 和 VS.Net 开发平台的 PC 机一台
- 2、XSBase270 实验开发平台一套

[实验原理]

1、UDP 编程概述

无连接通信是通过“用户数据报协议”(User Datagram Protocol, UDP)来完成,UDP 不保障可靠数据的传输,但能够向若干个目标发送数据,接收发自若干个源的数据。简单的说就是,如果一个客户机向服务器发送数据,这一数据会立即发出,不管服务器是否已准备接收数据,如果服务器收到客户机的数据,它不会确认收到与否。数据传输方法采用的是数据报。

UDP 由于不需要建立连接,所以它的传输效率要比 TCP 高。但它不能保证所有数据都能准确有序地到达目的地,不保证顺序性、可靠性和无重复性。它是无连接的服务。以独立的信息报进行传输。通信端点使用 UDP 对应的 Internet 地址,双方不需要互连,按固定的最大长度进行传输,因而适用于单个报文传输。

UDP 的编程过程:UDP 首先调用了 Socket 函数创建数据报套接字,然后再调用 bind 函数绑定本地址后,即可调用 sendto 和 recvfrom 函数来直接发送数据和接受数据。在 sendto 函数中直接指定接受方的地址和端口号,同时 recvfrom 函数里的参数可以直接得到接收的数据来源。

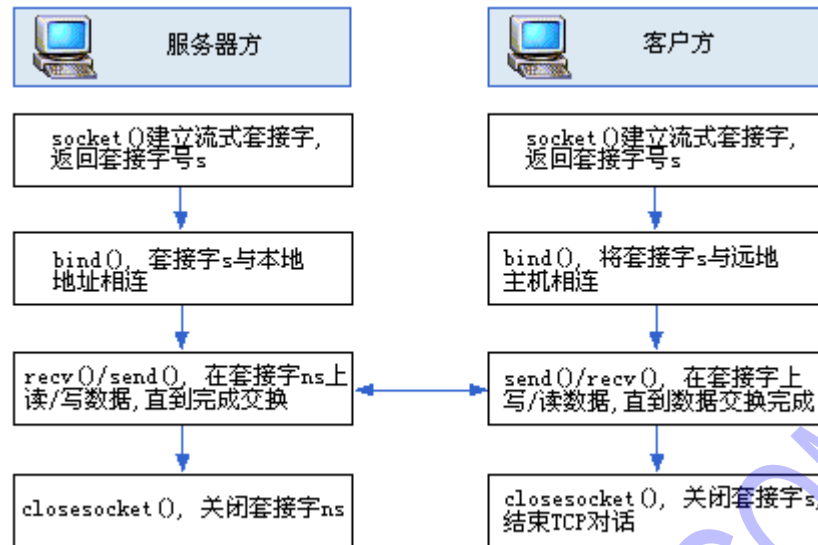


图 10-1 UDP 服务器端和客户端编程流程图

2、UDP 实现中主要函数介绍：

初始化和打开套接字

```

DWORD CUdpSocket::Open(CWnd* pWnd, int localPort, LPCTSTR remoteHost, int remotePort)
{
    WSADATA wsa;
    m_pOwnerWnd = pWnd;
    //加载winsock动态链接库
    if (WSAStartup(MAKEWORD(1, 1), &wsa) != 0)
        return -1; //代表失败
    //创建UDP套接字
    m_UDPSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (m_UDPSocket == INVALID_SOCKET)
        return -2;
    struct sockaddr_in localAddr;
    localAddr.sin_family = AF_INET;
    localAddr.sin_port = htons(localPort);
    localAddr.sin_addr.s_addr = INADDR_ANY;
    //绑定地址
    if (bind(m_UDPSocket, (sockaddr*)&localAddr, sizeof(localAddr)) != 0)
        return -3;
    //设置非堵塞通讯
    DWORD ul = 1;
    ioctlsocket(m_UDPSocket, FIONBIO, &ul);
    //创建一个线程退出事件
    m_ExitThreadEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
    //创建通讯线程
    AfxBeginThread(RecvThread, this);
}
    
```

```

        m_RemoteAddr.sin_family = AF_INET;
    m_RemoteAddr.sin_port = htons(remotePort);
    //此处要将双字节转换成单字节
    char ansiRemoteHost[255];
    ZeroMemory(ansiRemoteHost, 255);
    WideCharToMultiByte(CP_ACP, WC_COMPOSITECHECK, remoteHost, wcslen(remoteHost)
        , ansiRemoteHost, wcslen(remoteHost), NULL, NULL);
    m_RemoteAddr.sin_addr.s_addr=inet_addr(ansiRemoteHost);
    return 1;
}

```

接收数据线程

```

UINT CUDPSocket::RecvThread(LPVOID lparam)
{
    CUDPSocket *pSocket;
    pSocket = (CUDPSocket*)lparam;
    fd_set fdRead;
    int ret;
    TIMEVAL aTime;
    char * recvBuf=NULL;
    aTime.tv_sec = 1;
    aTime.tv_usec = 0;
    SOCKADDR_IN tmpAddr;
    int tmpRecvLen;
    int recvLen;
    int iErrorCode;
    char * recvedBuf = NULL;
    int recvedBufLen;
    while (TRUE)
    {
        //收到退出事件，结束线程
        if (WaitForSingleObject(pSocket->m_ExitThreadEvent, 0) == WAIT_OBJECT_0)
            break;
        //将set初始化空集合
        FD_ZERO(&fdRead);
        //将pSocket->m_UDPSocket套接字添加到集合中
        FD_SET(pSocket->m_UDPSocket, &fdRead);
        //调用select函数，判断套接字I/O状态
        ret = select(0, &fdRead, NULL, NULL, &aTime);
        if (ret == SOCKET_ERROR)
        {
            iErrorCode = WSAGetLastError();
            pSocket->m_OnUpdError(pSocket->m_pOwnerWnd, iErrorCode);
            break;
        }
    }
}

```

```

        if (ret > 0)
        {
            if (FD_ISSET(pSocket->m_UDPSocket, &fdRead))
            {
                tmpAddr.sin_family=AF_INET;
                tmpAddr.sin_port = htons(pSocket->m_RemoteAddr.sin_port);
                tmpAddr.sin_addr.s_addr =INADDR_ANY;
                tmpRecvLen = sizeof(tmpAddr);
                recvBuf = new char[1024];
                recvedBuf = new char[1024];
                ZeroMemory(recvBuf, 1024);
                ZeroMemory(recvedBuf, 1024);
                recvLen = recvfrom(pSocket->m_UDPSocket, recvBuf, 1024, 0, (SOCKADDR*)&tmpAddr, &tmpRecvLen);
                if (recvLen == SOCKET_ERROR)
                {
                    iErrorCode = WSAGetLastError();
                    pSocket->m_OnUdpError(pSocket->m_pOwnerWnd, iErrorCode);
                    break;
                }
                else if (recvLen == 0)
                {
                    iErrorCode = WSAGetLastError();
                    pSocket->m_OnUdpError(pSocket->m_pOwnerWnd, iErrorCode);
                    break;
                }
                else
                {
                    if (pSocket->HandlePackage(recvBuf, recvLen, recvedBuf, recvedBufLen))

                    pSocket->m_OnUdpRecv(pSocket->m_pOwnerWnd, recvedBuf, recvedBufLen, (SOCKADDR*)&tmpAddr);
                    delete []recvBuf;
                    recvBuf = NULL;
                    delete []recvedBuf;
                    recvedBuf = NULL;
                }
            }
        }
    }
    return 0;
}

```

发送数据

DWORD CUDPSocket::SendData(const char * buf, int len)

```

{
    int nBytes = 0;
    int nSendBytes=0;
    int nSumBytes =0;
    int nErrorCode;

    UDPData sendData;
    CopyMemory (sendData.FrameHead, FRAMEHEAD, 4);
    sendData.DataPackageLen = len;
    CopyMemory (sendData.FrameTail, FRAMETAIL, 4);

    nSumBytes = len + 12;

    char * sendBuf;
    sendBuf = new char[nSumBytes];
    CopyMemory (sendBuf, sendData.FrameHead, 4);
    CopyMemory (sendBuf+4, &(sendData.DataPackageLen), 4);
    CopyMemory (sendBuf+8, buf, len);
    CopyMemory (sendBuf+8+len, sendData.FrameTail, 4);

    //发送数据
    while (nSendBytes < nSumBytes)
    {
        nBytes =
sendto(m_UDPSocket, sendBuf+nSendBytes, nSumBytes-nSendBytes, 0, (sockaddr*)&m_RemoteAddr, sizeof
(m_RemoteAddr));
        if (nBytes==SOCKET_ERROR )
        {
            nErrorCode = WSAGetLastError();
            m_OnUdpError(m_pOwnerWnd, nErrorCode);
            return -1;
        }
        if (nSendBytes == nSumBytes)
        {
            break;
        }
        Sleep(1000);
        nSendBytes = nSendBytes + nBytes;
    }
    delete[] sendBuf;
    return nSendBytes;
}

```

3、PING 编程概述

ICMP (Internet Control Messages Protocol, 网间控制报文协议) 允许主机或路由器报告差错情况和提供有关异常情况的报告。

一般来说, ICMP 报文提供针对网络层的错误诊断、拥塞控制、路径控制和查询服务四项大的功能。如, 当一个分组无法到达目的站点或 TTL 超时后, 路由器就会丢弃此分组, 并向源站点返回一个目的站点不可到达的 ICMP 报文。

PING 的实现可以使用一个 ICMPECHO 数据包来探测主机地址是否存活, 当发送一个 ICMPECHO (Type 8) 数据包到目标机时, 如果 ICMPECHOREPLY(ICMPType0)数据包被收到, 则说明主机是存活的。如果没有则可以初步判断主机没有在线或者使用了某些过滤设备过滤了 ICMP 的 REPLY。这种机制就是使用 ping 命令检测目标机的依据。

在实现 ping 的过程中, 需要用到 IcmpCreateFile、IcmpSendEcho 和 IcmpCloseHandle 三个函数, 下面分别介绍这三个函数

HANDLE WINAPI IcmpCreateFile(VOID);

IcmpCreateFile 函数创建一个 ping 句柄, 该句柄将在发送 ping 数据和接收 ping 应答中用到, 如果 IcmpCreateFile 函数返回 INVALID_HANDLE_VALUE, 表示创建 ping 句柄失败, 否则将返回 ping 句柄。

BOOL WINAPI IcmpCloseHandle(HANDLE IcmpHandle);

IcmpCloseHandle 关闭创建的 ping 句柄。

DWORD WINAPI IcmpSendEcho(
 HANDLE IcmpHandle,
 IPAddr DestinationAddress,
 LPVOID RequestData,
 WORD RequestSize,
 PIP_OPTION_INFORMATION RequestOptions,
 LPVOID ReplyBuffer,
 DWORD ReplySize,
 DWORD Timeout
);

IcmpSendEcho 函数的功能是发送 ping 数据包和接收 ping 应答。其参数的含义:

【IcmpHandle】表示已经创建的 ping 句柄;

【DestinationAddress】表示 ping 的目标 IP 地址;

【RequestData】表示要发送的 ping 数据包;

【RequestSize】表示要发送的 ping 数据包长度

【RequestOptions】表示请求 IP 头的方式, 可以设为 NULL

【ReplyBuffer】表示执行 ping 方法后收到的应答数据

【ReplySize】表示收到的应答数据大小

【Timeout】表示指定等待应答的最大时间。

该函数返回值表示接收到应答的数量, 如果返回 0 表示执行 ping 失败。

4、ping 的实现

```
DWORD CWinPingDlg::Ping(LPVOID param)
{
```

```

CWinPingDlg *pDlg=(CWinPingDlg*)param;
WaitForSingleObject(pDlg->m_hEvent, INFINITE);
WSADATA wsaData;
//初始化Socket动态链接库
if (WSAStartup(MAKEWORD(1, 1), &wsaData) != 0)
    return 0;
//将IP地址转换成单字节
USES_CONVERSION;
char *szDestIPAddr=W2A(pDlg->m_ipAddr);
IPAddr ipAddr;
//将目标字符串IP地址转换成IPAddr结构
ipAddr = inet_addr(szDestIPAddr);
if (ipAddr == INADDR_NONE) {
    AfxMessageBox(TEXT("地址无效"));
    return 0;
}
// 打开ping服务
HANDLE hIP = IcmpCreateFile();
if (hIP == INVALID_HANDLE_VALUE) {
    AfxMessageBox(TEXT("不能打开Ping服务"));
    return 0;
}
// 构造ping数据包
char acPingBuffer[64];
memset(acPingBuffer, '*', sizeof(acPingBuffer));
PICMP_ECHO_REPLY pIpe = (PICMP_ECHO_REPLY)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
    sizeof(ICMP_ECHO_REPLY) + sizeof(acPingBuffer));
if (pIpe == 0) {
    AfxMessageBox(TEXT("分配ping包缓冲区失败"));
    return 0;
}
CString m_recv;
// 发送ping数据包
m_recv.Format(_T("Ping %s with 64 Bytes"), pDlg->m_ipAddr);
int nCnt=pDlg->m_lstRecv.GetItemCount();
pDlg->m_lstRecv.InsertItem(nCnt, m_recv, 1);
for (UINT i=0; i<pDlg->m_retry; i++) {
    //发送ping服务包, 等待接收时间为秒
    DWORD dwStatus = IcmpSendEcho(hIP, ipAddr,
        acPingBuffer, sizeof(acPingBuffer), NULL, pIpe,
        sizeof(ICMP_ECHO_REPLY) + sizeof(acPingBuffer), 1000);
    //当dwStatus不等于0, 代表接收到回应
    if (dwStatus != 0) {
        m_recv.Format(TEXT("Reply From %d.%d.%d.%d :bytes=%d time=%d TTL=%d"),

```

```
, int(LOBYTE(LOWORD(pIpe->Address)))
, int(HIBYTE(LOWORD(pIpe->Address)))
, int(LOBYTE(HIWORD(pIpe->Address)))
, int(HIBYTE(HIWORD(pIpe->Address)))
, int(pIpe->DataSize)
, int(pIpe->RoundTripTime)
, int(pIpe->Options.Ttl));

nCnt = pDlg->m_lstRecv.GetItemCount();
pDlg->m_lstRecv.InsertItem(nCnt, m_recv, 2);
Sleep(500);

}

else
{
    nCnt = pDlg->m_lstRecv.GetItemCount();
    pDlg->m_lstRecv.InsertItem(nCnt, TEXT("Error obtaining info from ping packet."), 3);
}

}

pDlg->m_bProgress = FALSE;
GlobalFree(pIpe); //释放已分配的内存
IcmpCloseHandle(hIP); //关闭Ping服务
WSACleanup(); //释放Socket资源
return 0;
}
```

[实验内容]

- 1、UDP 数据通信原理和过程;
- 2、PING 实现的基本原理和方法;
- 3、了解非连接的网络编程方法

[实验步骤]

第一步：连接好实验系统，打开实验箱电源；

第二步：利用 VS2005.net 打开 UDP 客户端程序 UDPClient.sln 和 UDP 服务器端程序 UDPServer.sln，并进行编译：点击运行按钮，将 UDP 客户端和服务端测试程序下载到 XSBase270 目标板运行，UDP 客户端和服务端运行界面如图 10-1 所示；

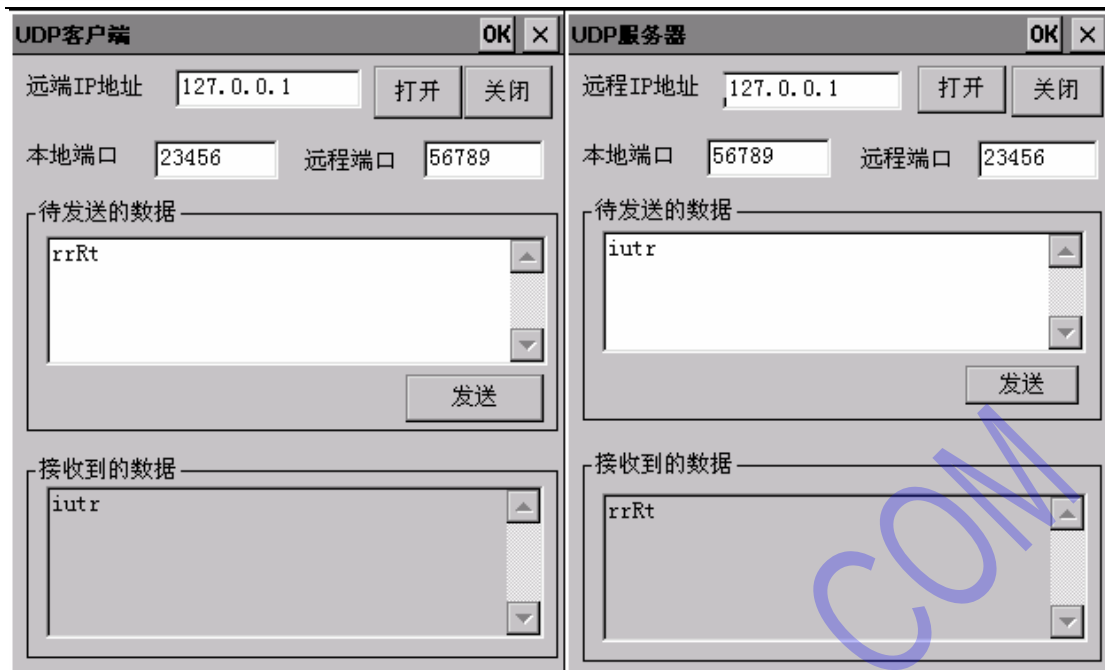


图 UDP 网络通信的客户端和服务端运行界面

第四步：利用 VS2005.net 打开 PING 测试程序 WinPing.sln。进行编译：点击运行按钮，将 PING 测试程序下载到 XSBASE270 目标板运行，其运行界面如图 10-2 所示；



图 10-2 PING 测试程序运行界面

[习题与思考题]

- 1、分析 ICMP 和 UDP 协议，写出其数据包格式。
- 2、在 UDP 程序中，如果实现大数据包的通信，怎样完成？

- 3、分析 PING 程序的中 IcmpCreateFile、IcmpCloseHandle 和 IcmpSendEcho 三个函数的使用顺序。

WWW.ENDOOR.COM

实验十一 驱动程序实验

[实验目的]

- 1、了解驱动程序的原理和功能；
- 2、掌握流式接口驱动程序的结构；
- 3、掌握编写流式接口的驱动程序的方法
- 4、熟悉 EVC 和 VS.Net 的开发环境；

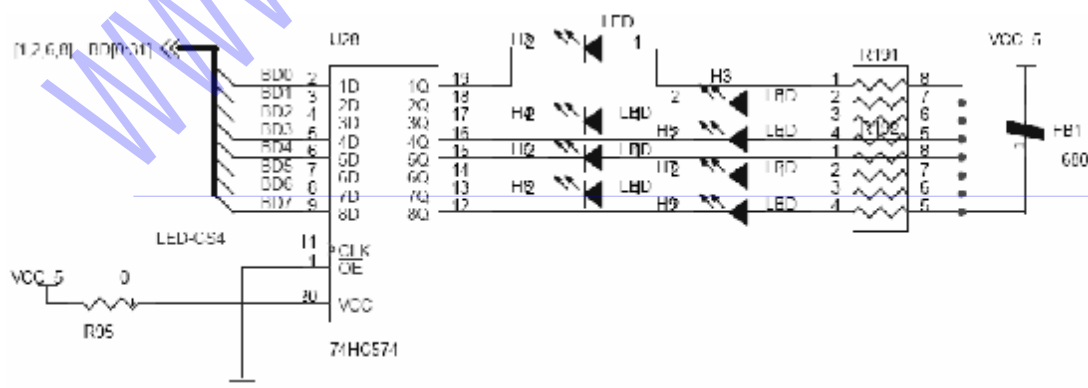
[实验仪器]

- 1、装有 Platform Builder、EVC 和 VS.Net 开发平台的 PC 机一台
- 2、XSBase270 实验开发平台一套

[实验原理]

1、硬件接口电路分析

XSBase 目标板 LED 和七段数码显示接口电路如图 6-1 所示，74HC574 为 D 锁存器，在时钟信号 CLK 作用下，该锁存器将输入信号进行锁存，即 $xQ=xD$ ($x=1\sim 8$)。从电路图中可以看出，LED 和八段数码显示电路将 74HC574 的时钟信号输入端作为片选信号，其中 LED 显示的片选信号为 LED_CS4、七段数码显示的片选信号为 LED_CS1（另外还有一组七段数码显示未画出，参考原理图）。在七段数码显示电路中，数据的高位（D7、D15：即数码管的小数点 dp 段）用作七段数码的公共选通信号，通过控制 PNP 三极管来控制数码管的显示。



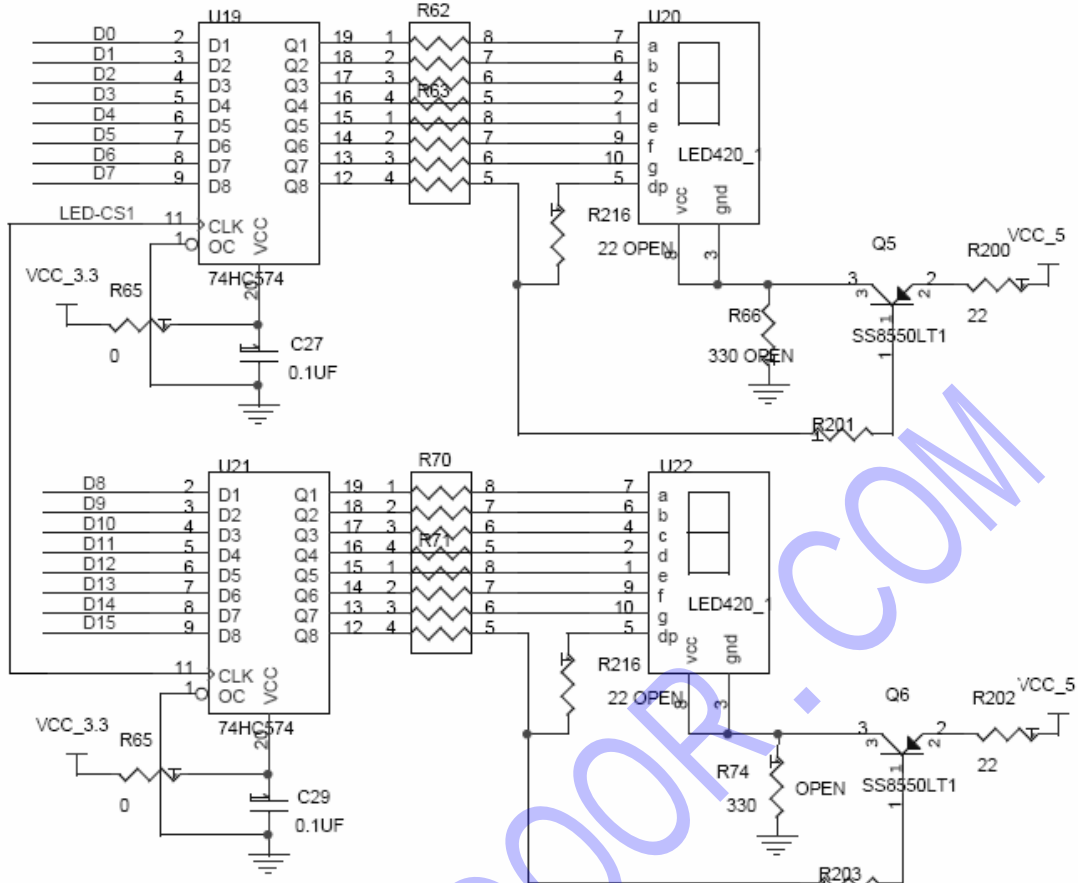


图 6-1 LED 和七段数码显示接口电路

显示电路中的片选信号 LED_CS_x(x=1-4), 由 XSBase270 目标板系统的处理器 PXA270x 的地址信号 BA22~BA20 通过 3-8 译码器 LC138 产生 (如图 2-2 所示)。

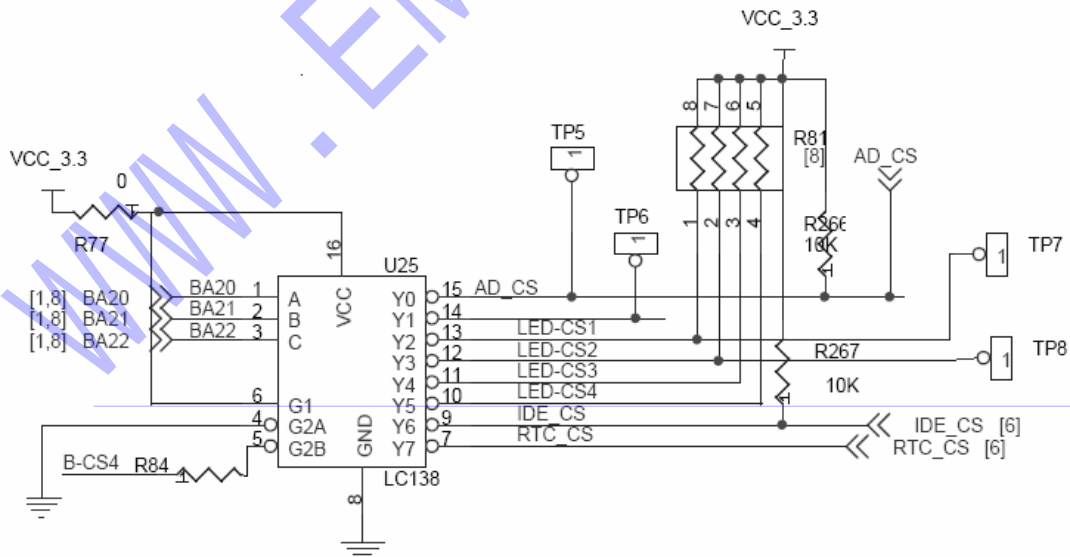


图 6-2 片选信号产生电路

由 3-8 控制功能可知, 当 BA22、BA21、BA20=101 时产生 LED 显示电路的片选信号 LED_CS4, 当 BA22、BA21、BA20=010、011、100 时分别产生七段数码显示电路的片选信号 LED_CS1、LED_CS2、LED_CS3 (另外 2 组八段显示电路参考系统提供的总电路图)。根据分析, 可以得出七段数码管和 LED 的片选信号为: LED_CS1=0x10200000、LED_CS2

=0x10300000、LED_CS3=0x10400000，LED_CS4=0x10500000。

2、流式接口驱动程序简介

驱动程序是对底层硬件的抽象。应用程序开发者不需要真正理解底层驱动的工作原理，他们只需要通过 Windows CE 提供的 API 函数，就可以直接与硬件进行交互。

WinCE 的流式接口驱动程序以动态链接库的形式存在，由设备管理器（通常是 device.exe 或者 gws.exe）统一加载、管理和卸载。与具有单独目的的内部设备驱动程序相比，所有流式接口驱动程序都是用同一接口并调用同一个函数集。

每个流式接口驱动程序必须实现一组标准的函数，用来完成标准的文件 I/O 函数和电源管理函数，这些函数提供给 WinCE 操作系统的内核使用。这些函数通常叫做流式接口驱动程序的 DLL 接口，如表 11-1 所示：（详细介绍参考实验三）

表 11-1 流式接口驱动程序要实现的 DLL 接口：

函数名称	描述
XXX_Close	在驱动程序关闭时应用程序通过 CloseHandle 函数调用这个函数
XXX_Deinit	当设备管理器卸载一个驱动程序时调用这个函数
XXX_Init	当设备管理器初始化一个具体设备时调用这个函数。
XXX_IOControl	上层的软件通过 DeviceIoControl 函数可以调用这个函数
XXX_Open	在打开一个设备驱动程序时应用程序通过 CreateFile 函数调用这个函数
XXX_PowerDown	在系统调用前调用这个函数
XXX_PowerUp	在系统从新启动前调用这个函数
XXX_Read	在一个设备驱动程序处于打开状态时由应用程序通过 ReadFile 函数调用
XXX_Seek	对设备的数据指针进行操作，由应用程序通过 SetFilePointer 函数调用
XXX_Write	在一个设备驱动程序处于打开状态时由应用程序通过 WriteFile 函数调用。

3、LED 流式接口驱动程序的实现

由于在 WinCE 中流式接口驱动程序以 DLL 的形式存在，是运行在用户模式的动态链接库，所以既可以使用微软提供的 EVC 编写流式接口驱动程序，也可以使用 Platform Builder (PB) 来进行编写。为了方便调试，通常采用 PB 来开发流式驱动程序。下面具体介绍 LED 驱动程序为例介绍采用 PB 编写流式接口设备驱动程序的方法和步骤。

3.1 流式驱动程序的创建步骤：

(1) 打开 Platform Builder。在 Platform Builder 中选择“File”->“New Project or File”，创建一个“Windows CE Dynamic link library”项目，项目的名称填写“LedDriver”（如图 11-1 所示）

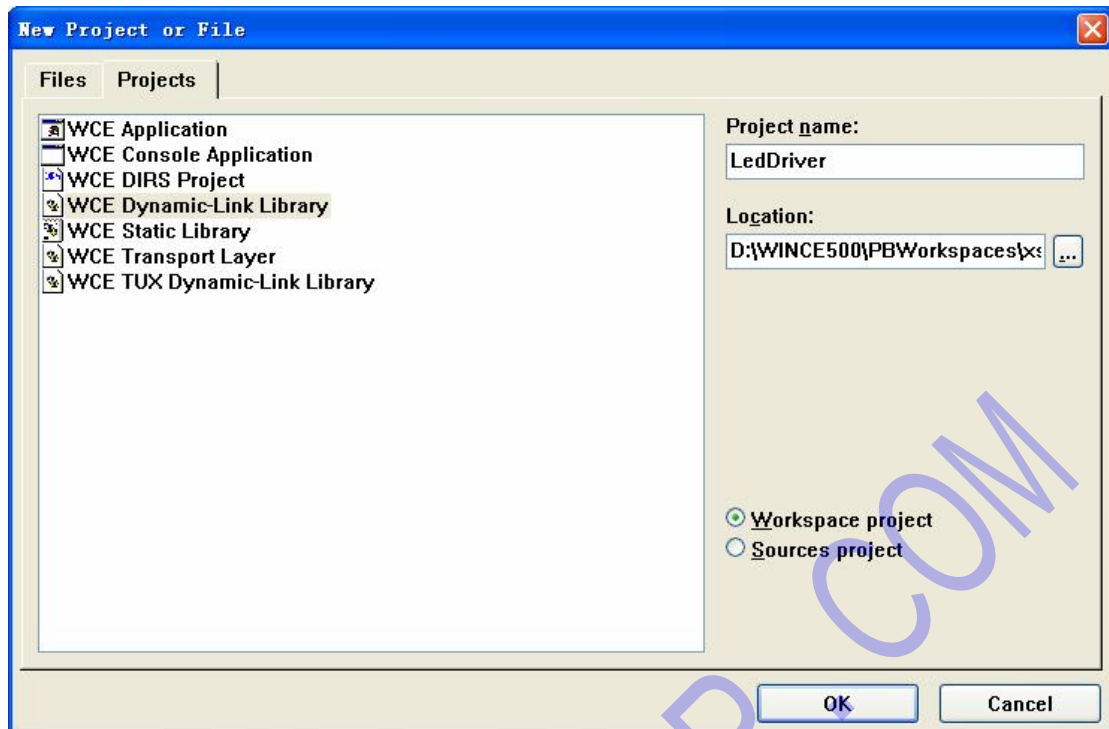


图 11-1 WinCE 驱动创建项目界面

(2) 按 OK 按钮，在 DLL 的类型界面中（如图 11-2 所示）选中 A Simple Windows CE DLL projects，Platform Builder 将生成 DLL 框架代码。

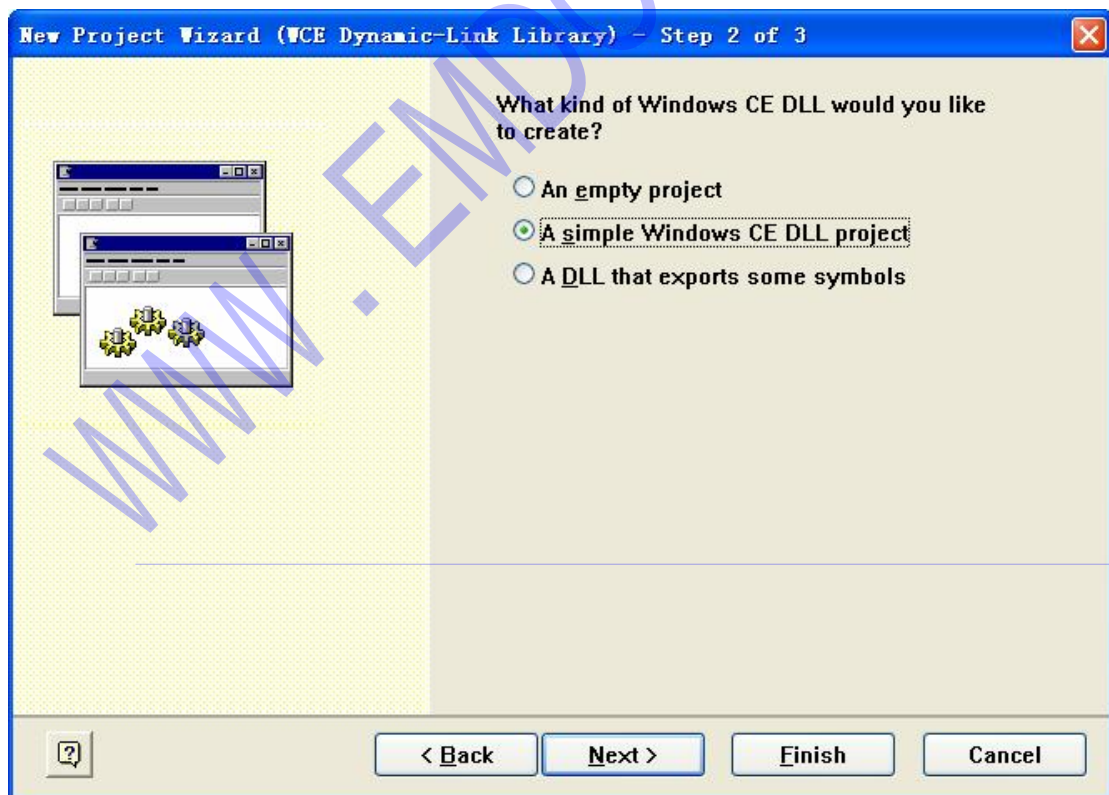


图 11-2 驱动程序的 DLL 类型选择

3.2 LED 流式驱动程序的源代码分析：

(1) 变量定义

```
#define LED_IOCTL_SET      0x00000001      //IO 控制接口代号码
#define Light_IOCTL_Set    0x00000002
#define Light_IOCTL_LShift 0x00000003
#define Light_IOCTL_RShift 0x00000004
#define Light_IOCTL_LRShift 0x00000005
#define Light_IOCTL_ShiftStop 0x00000006
#define Light_IOCTL_SetTime 0x00000007

#define LED_BASEADDR1 0x10200000      //七段数码的物理地址
#define LED_BASEADDR2 0x10300000
#define LED_BASEADDR3 0x10400000
#define pLightIoBaseAddress 0x10500000      //Led 物理地址

#define BIT7 (0x1<<7)
#define BIT15 (0x1<<15)
HANDLE m_hShiftEvent=NULL;      //LED 流水灯移动事件变量
BOOL m_bStop;      //LED 流水灯停止变量
UINT m_ShiftDir;      // LED 流水灯移动方向
UINT ShiftTime;      // LED 流水灯时间间隔
unsigned char ShiftData;      // LED 流水灯赋值变量
```

(2) 驱动程序初始化函数 DEM_Init

驱动程序的初始化函数 XXX_Init 主要完成当用户使用一个设备的时候，设备管理器调用这个函数来对设备进行初始化。这个函数并不是由应用程序调用的，而是通过设备管理器提供的 ActiveDeviceEx () 来调用。该初始化函数主要完成 LED 和七段数码的物理地址到虚拟地址的转换。

```
DWORD DEM_Init(DWORD dwContext)
{
    char black=0xff;
    RETAILMSG(1,(TEXT("Led_Init\r\n")));//从串口输出调试信息
    if(!LEDInit())
        return FALSE;
    SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_NORMAL);
    m_hShiftEvent=CreateEvent(NULL,FALSE,FALSE,NULL);
    if(m_hShiftEvent==NULL)
    {
        RETAILMSG(1, (TEXT("Failed to Create thread\r\n")));
        return FALSE;
    }
    *pLightReg=black;
    ShiftTime=500;
    m_bStop=FALSE;
    m_ShiftDir=1;
    return TRUE;
}
```

```

BOOL LEDInit(void)
{
    char black=0xff;
    PHYSICAL_ADDRESS ioPhysicalBase = {LED_BASEADDR1, 0 };
    v_pLEDBaseAddr[0]=(USHORT *)MmMapIoSpace(ioPhysicalBase,0x400,FALSE);
    ioPhysicalBase.QuadPart = LED_BASEADDR2;
    v_pLEDBaseAddr[1]=(USHORT *)MmMapIoSpace(ioPhysicalBase,0x400,FALSE);
    ioPhysicalBase.QuadPart = LED_BASEADDR3;
    v_pLEDBaseAddr[2]=(USHORT *)MmMapIoSpace(ioPhysicalBase,0x400,FALSE);

    if(!(pLightReg=(char*)VirtualAlloc(0,0x400,MEM_RESERVE,PAGE_READWRITE)))
    {
        RETAILMSG(1,(TEXT("VirtualAlloc() failed!\r\n")));
        return FALSE;
    }
    if(!VirtualCopy((PVOID)pLightReg,(PVOID)(pLightIoBaseAddress>>8),0x400,PAGE_READWRITE|PAGE_NOCACHE|PAGE_PHYSICAL))
    {
        VirtualFree((PVOID)pLightReg,0,MEM_RELEASE);
        pLightReg = NULL;
        RETAILMSG(1,(TEXT("VirtualCopy() failed!\r\n")));
        return FALSE;
    }

    if (!v_pLEDBaseAddr||!pLightReg)
    {
        DEBUGMSG(1,(TEXT("Virtual copy Error ,Error code=%u\r\n"),GetLastError()));
        return (FALSE);
    }

    WRITE_PORT_USHORT(v_pLEDBaseAddr[0],0xffff);
    WRITE_PORT_USHORT(v_pLEDBaseAddr[1],0xffff);
    WRITE_PORT_USHORT(v_pLEDBaseAddr[2],0xffff);
    *pLightReg =black;
    return TRUE;
}

```

(3) 驱动程序卸载函数 DEM_Deinit:

当设备管理器卸载一个驱动程序时调用驱动程序卸载函数 XXX_Deinit，该函数完成对系统资源的回收。

```

BOOL DEM_Deinit(DWORD hDeviceContext)
{
    UINT i=0;
    RETAILMSG(1,(TEXT("DEM_Deinit\r\n")));
}

```

```

for(i=0;i<3;i++)
{
    MmUnmapIoSpace(v_pLEDBaseAddr[i],0);
    v_pLEDBaseAddr[i] = NULL;
}

if (pLightReg)
{
    MmUnmapIoSpace(pLightReg,0);
    pLightReg = NULL;
}

if(m_hShiftEvent)
    CloseHandle(m_hShiftEvent);
return TRUE;
}

```

(4) 驱动程序的读写操作 DEM_Read、DEM_Write:

当一个流式接口驱动程序已经打开后,可以使用 ReadFile () 函数和 WriteFile () 函数对这个设备进行读写操作。该驱动程序的读操作完成将 LED 的亮灭的情况输出,应用程序通过 ReadFile () 函数,写操作函数对四个七段数码管进行点亮。具体实现函数如下:

```

DWORD DEM_Read(DWORD hOpenContext, LPVOID pBuffer, DWORD Count)
{
    UINT temp;
    temp=(UINT)ShiftData;
    RETAILMSG(1,(TEXT("LED_Read      \r\n      hOpenContext=%d;*pBuffer=%d;
Count=%d\r\n"),hOpenContext,temp,Count));
    *(UINT*)pBuffer=temp;
    return TRUE;
}

```

```

DWORD DEM_Write(DWORD hOpenContext, LPCVOID pSourceBytes, DWORD
NumberOfBytes)
{
    INT buf=*(INT*)pSourceBytes;
    USHORT Data;
    RETAILMSG(1,(TEXT("DEM_Write\r\n
hOpenContext=%d;*pSourceBytes=%d;NumberOfBytes=%d\r\n"),hOpenContext,buf,NumberOf
Bytes));
    buf=buf%10000;
    Data=NumData[buf/1000];
    buf=buf%1000;
    Data|=NumData[buf/100]<<8;
    WRITE_PORT_USHORT(v_pLEDBaseAddr[1],~(Data|BIT7|BIT15));
}

```

```

    buf=buf%100;
    Data=NumData[buf/10];
    buf=buf%10;
    Data|=NumData[buf]<<8;
    WRITE_PORT_USHORT(v_pLEDBaseAddr[2],~(Data|BIT7|BIT15));
    return TRUE;
}

```

(5) IO 控制函数 DEM_IOControl

XXX_IOControl 通常用于向设备发送一个命令，应用程序使用 DeviceIOControl 函数来通知操作系统调用该函数。LED 驱动程序的 DEM_IOControl 函数主要完成对八个 LED 发光二极管的流水左移、流水右移、流水循环、流水停止、点亮设置、流水移动时间间隔进行控制，采用线程完成流水操作。

```

BOOL DEM_IOControl(DWORD hOpenContext, DWORD dwCode, PBYTE pBufIn, DWORD
dwLenIN, PBYTE pBufOut, DWORD dwLenOut, PDWORD pdwActualOut)
{
    char Data;
    HANDLE m_hShiftThread=NULL;
    RETAILMSG(1,(TEXT("LED_IOControl\r\n
hOpenContext=%d;dwCode=%d\r\n"),hOpenContext,dwCode));
    m_hShiftThread=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)LedShiftThread,
NULL,0,NULL);
    switch(dwCode)
    {
    case Light_IOCTL_LShift:
        m_ShiftDir=1;
        SetEvent(m_hShiftEvent);
        break;
    case Light_IOCTL_RShift:
        m_ShiftDir=2;
        SetEvent(m_hShiftEvent);
        break;
    case Light_IOCTL_LRShift:
        m_ShiftDir=3;
        SetEvent(m_hShiftEvent);
        break;
    case Light_IOCTL_Set:
        Data=*(char*)pBufIn;
        *pLightReg=~Data;
        break;
    case Light_IOCTL_ShiftStop:
        m_bStop=TRUE;
        break;
    case Light_IOCTL_SetTime:

```

```

        ShiftTime=*(UINT*)pBufIn;
        break;
    }
    CloseHandle(m_hShiftThread);
    return TRUE;
}

INT WINAPI LedShiftThread(void)
{
    UINT i=0;
    WaitForSingleObject(m_hShiftEvent,INFINITE);
    while(1)
    {
        switch(m_ShiftDir)
        {
            case 1:
                ShiftData=0x01;
                for(i=0;i<8;i++)
                {
                    *pLightReg=~ShiftData;
                    Sleep(ShiftTime);
                    ShiftData=ShiftData<<1;
                    if(m_bStop)
                    {
                        *pLightReg=0xff;
                        m_bStop=FALSE;
                        return 0;
                    }
                }
                break;
            case 2:
                ShiftData=0x80;
                for(i=0;i<8;i++)
                {
                    *pLightReg=~ShiftData;
                    Sleep(ShiftTime);
                    ShiftData=ShiftData>>1;
                    if(m_bStop)
                    {
                        *pLightReg=0xff;
                        m_bStop=FALSE;
                        return 0;
                    }
                }
        }
    }
}

```

```

        break;
    case 3:
        ShiftData=0x01;
        for(i=0;i<8;i++)
        {
            *pLightReg=~ShiftData;
            Sleep(ShiftTime);
            ShiftData=ShiftData<<1;
            if(m_bStop)
            {
                *pLightReg=0xff;
                m_bStop=FALSE;
                return 0;
            }
        }
        ShiftData=0x80;
        for(i=0;i<8;i++)
        {
            *pLightReg=~ShiftData;
            Sleep(ShiftTime);
            ShiftData=ShiftData>>1;
            if(m_bStop)
            {
                *pLightReg=0xff;
                m_bStop=FALSE;
                return 0;
            }
        }
        break;
    default:
        m_bStop=FALSE;
        return 0;
        break;
    }
}
return 0;
}

```

其他实现函数如：DEM_Open、DEM_PowerDown、DEM_PowerUp、DEM_Seek、DEM_Close 参考提供的源代码文档。

(6) 添加导出函数的定义。在 Platform Builder 菜单中，选择 File->New Project or File -> File Tab ->Text File，文件名叫“LedDriver.def”，内容如下：

```

LIBRARY      "LedDriver"

```

```
EXPORTS
    DEM_Init
    DEM_Deinit
    DEM_Open
    DEM_Read
    DEM_Write
    DEM_PowerDown
    DEM_PowerUp
    DEM_Seek
    DEM_IOControl
    DEM_Close
```

(7) 编译驱动程序项目。结束后选择“Build OS”->“Open Build Release Directory”，然后输入命令：dumpbin /exports LedDriver.dll，确保输出结果如图 11-3，表示驱动程序的 DLL 函数被正确导出。

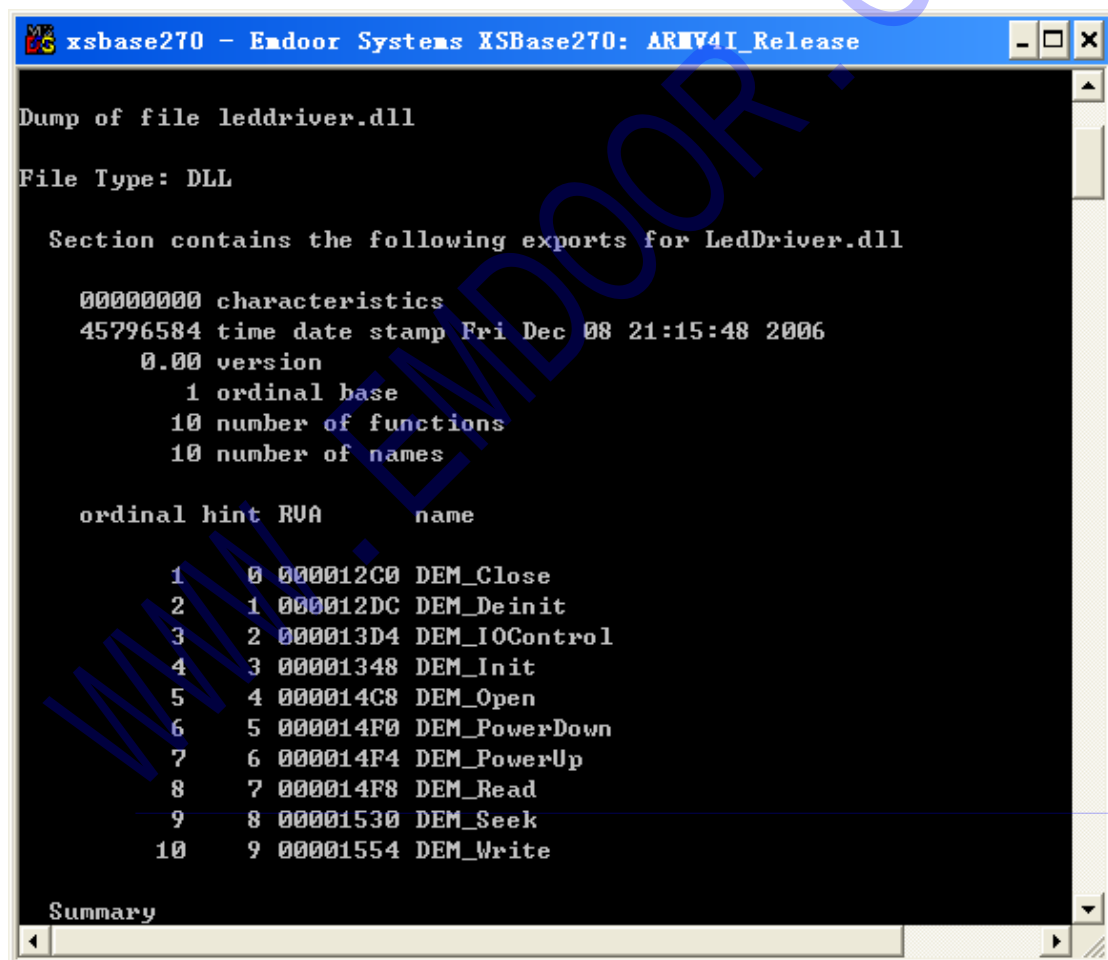


图 11-3 驱动程序的函数导出图

3.2 LED 流式驱动程序的加载：

(1) 选择“Build OS”->“Open Build Release Directory”，利用文本编辑器 notepad 修改平

台的注册文件 platform.reg，在文件最后添加如下内容：

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\LedDriver]
"Dll" = "LedDriver.Dll"
"Prefix" = "DEM"
"Index" = dword:1
"Order" = dword:0
"FriendlyName" = "LED Driver"
```

这段注册表信息告诉系统，在系统启动的时候把 LedDriver.dll 加载到 device.exe，驱动的前缀是 DEM，下标索引是 1，这样，应用程序中可以使用 CreateFile(L"DEM".....)这样的方式来访问驱动程序了。

(2) 驱动程序固化

选择“Build OS”->“Open Build Release Directory”，利用文本编辑器 notepad 修改平台的配置文件 platform.bib，将驱动程序 LedDriver.dll 文件固化到映像文件中在文件，在 platform.bib 文件中具体添加内容如下：

```
LedDriver.dll $( _FLATRELEASEDIR )\LedDriver.dll NK SH
```

③、生成映像文件

选择“Build OS”->“Make Run-Time Image”生成映像文件，然后下载到目标板中。

4、LED 流式接口驱动程序的应用测试

利用 VS.net 编写一个驱动程序的应用测试程序 DriverApp。

具体实现如下：

● 打开设备

```
void CDriverAppDlg::OnBnClickedbtnopendevic()
{
    TCHAR FileName[10]=TEXT("DEM1:");
    m_hFile=CreateFile((LPCTSTR)&FileName, GENERIC_WRITE|GENERIC_READ, FILE_SHARE_READ,
        NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    if(m_hFile==NULL){
        MessageBox(_T("打开LED设备失败!"), _T("系统信息"), MB_OK|MB_ICONINFORMATION);
    }
}
```

● LED流水灯左移

```
void CDriverAppDlg::OnBnClickedbtnlshift()
{
    if(m_hFile)
    {
        if(!DeviceIoControl(m_hFile, Light_IOCTL_RShift, NULL, 0, NULL, 0, NULL, NULL))
        {
            MessageBox(_T("设备操作失败!"), _T("系统信息"), MB_OK|MB_ICONINFORMATION);
        }
    }
}
```

```

        CloseHandle(m_hFile);

        return ;
    }
}

else
    MessageBox(_T("设备文件没打开!"), _T("系统信息"), MB_OK|MB_ICONINFORMATION);
}

```

● LED流水灯右移

```

void CDriverAppDlg::OnBnClickedbtnrshift()
{
    if(m_hFile)
    {
        if(!DeviceIoControl(m_hFile, Light_IOCTL_LShift, NULL, 0, NULL, 0, NULL, NULL))
        {
            MessageBox(_T("设备操作失败!"), _T("系统信息"), MB_OK|MB_ICONINFORMATION);
            CloseHandle(m_hFile);
            return ;
        }
    }
    else
        MessageBox(_T("设备文件没打开!"), _T("系统信息"), MB_OK|MB_ICONINFORMATION);
}

```

● LED流水灯循环移

```

void CDriverAppDlg::OnBnClickedbtnlrshift()
{
    if(m_hFile)
    {
        if(!DeviceIoControl(m_hFile, Light_IOCTL_LRShift, NULL, 0, NULL, 0, NULL, NULL))
        {
            MessageBox(_T("设备操作失败!"), _T("系统信息"), MB_OK|MB_ICONINFORMATION);
            CloseHandle(m_hFile);
            return ;
        }
    }
    else
        MessageBox(_T("设备文件没打开!"), _T("系统信息"), MB_OK|MB_ICONINFORMATION);
}

```

● 设备读操作和定时器

```

void CDriverAppDlg::OnBnClickedbtnread()
{
    SetTimer(1, 200, NULL);
}

```

```
void CDriverAppDlg::OnTimer(UINT_PTR nIDEvent)
{
    UINT buf=0;
    UINT tmp=0;
    DWORD nBytesRead;
    if(m_hFile)
        ReadFile(m_hFile,&buf, 1,&nBytesRead, NULL);
    CStatic *pStatic=NULL;
    for(int index=0;index <8;index++)
    {
        tmp=buf&0x1;
        pStatic=(CStatic*)GetDlgItem(IDC_BMPL1+index);
        pStatic->SetBitmap( tmp==1 ? m_BmpRed:m_BmpBlack);
        buf>>=1;
    }
    CDialog::OnTimer(nIDEvent);
}
```

● 设备写操作

```
void CDriverAppDlg::OnBnClickedbtnwrite()
{
    UpdateData(TRUE);
    DWORD dwWritten=0;
    if(m_hFile)
        WriteFile(m_hFile,&m_SegValue, 1,&dwWritten, NULL);
    else
        MessageBox(_T("设备文件没打开!"),_T("系统信息"),MB_OK|MB_ICONINFORMATION);
}
```

其他操作函数参考源代码。

[实验内容]

- 1、在 PB 环境中编写编译 LedDriver 驱动程序;
- 2、将编译后的驱动程序加载到映像文件中
- 3、编写驱动的测试程序,对上述编译好的驱动程序进行应用测试。
- 4、基本掌握流式驱动程序的编写方法和编译调试过程

[实验步骤]

第一步: 连接好实验系统, 打开实验箱电源;

第二步: 运行 PB, 打开一个工作区, 并打开驱动程序工程文件 LedDriver.pbpxml; 然后编译当前工程。

第三步：参照实验原理修改平台注册和配置文件，然后生成映像文，并将映像文件下载到目标板上

第四步：打开驱动程序测试工程文件 DriverAppl.sln，编译该工程文件，点击运行按钮，将驱动程序测试程序下载到 XSBase270 目标板运行，运行界面如图 11-4 所示；



图 11-4 驱动程序测试运行界面

第五步：驱动测试程序界面的操作步骤

- (1)、按“打开设备”，打开设备文件；
- (2)、设备控制操作：设备控制操作包括八个 LED 的流水灯左移、右移、循环移、点亮设置、流水灯的时间间隔设置、停止等。从一种流水灯操作切换到另一种方式，必须先按“停止”按钮后才进行切换
- (3)、驱动读写操作：当 LED 在进行流水灯操作、如果按“读操作”旁边的八个图标与目标板的流水灯同步移动。
- (4)、驱动写操作：在文本框中输入数字，然后按“写操作”，目标板的四个七段数码管将显示文本框中的数字。
- (5)、设备关闭：按“关闭设备”，关闭设备文件；
- (6)、系统退出：按“退出系统”，中止应用程序。

[习题与思考题]

- 1、分析流式接口驱动程序的 DLL 接口函数集每个参数的作用；
- 2、在什么情况下，驱动程序需要编写 XXX_PowerDown 和 XXX_PowerUp 功能？
- 3、完善驱动程序，使 LED 流水灯在进行方式切换时不需按“停止”按钮。

实验十二 串口编程实验 -GSM 和 GPS

实验

[实验目的]

- 1、掌握串口编程方法；
- 2、掌握 GSM 的 SMS 数据格式以及 AT 指令集；
- 3、了解 GPS 数据格式
- 4、熟悉 EVC 和 VS.Net 的开发环境；

[实验仪器]

- 1、装有 Platform Builder、EVC 和 VS.Net 开发平台的 PC 机一台
- 2、XSBase270 实验开发平台一套
- 3、MC35 GSM 模块一块
- 4、GPS 模块一块

[实验原理]

1、GPS 数据格式

NMEA-0183 是美国国家海洋电子协会(National Marine Electronics Association)制定的 GPS 接口协议标准。这种接口协议采用 ASCII 码输出，协议定义了若干代表不同含义的语句，语句格式如表 13-1 所示。

表 12-1 NMEA-0183 语句格式^[38]

符号 (ASCII)	定义	HEX	DEX	说明
\$	起始位	24	36	语句起始位
aaccc	地址域			前两位为位识别符，后三位为语句名
“,”	域分隔符	2C	44	域分隔符
ddd.....ddd	数据块			发送的数据内容
“* ”	校验和符	2A	42	星号分隔符，表明后面的两位数是校验和
Hh	校验和			校验和
<CR>/<LF>	终止符	0D,0A	13,10	回车，换行

NMEA-0183 接口协议定义的主要语句有：GGA、GLL、GSA、GSV、MSS、RMC、VTG、ZDA 等。表 13-2 介绍这些语句所包含的具体内容。

表 12-2 常见 NMEA-0183 语句内容^[38]

语句	语句内容
GGA	UTC 时间、纬度值、经度值、定位状态（无效、单点定位、差分）、观测的 GPS 卫星个数、HDOP 值、GPS 椭球高、天线架设高度、差分数据龄期、差分基准站编号、校验和
GLL	UTC 时间、纬度值、经度值、定位状态（无效、单点定位、差分）、校验和
GSA	定位模式（M—手动，强制二维或三维定位；A—自动，自动二维或三维定位）、定位中使用的卫星 ID 号、PDOP 值、HDOP 值、VDOP 值
GSV	视野中的 GPS 卫星颗数、PRN 编号、卫星仰角、距正北的角度（方位角）、信噪比
MSS	信标台的信号强度、信噪比、信标频率、波特率、通道号
RMC	UTC 时间、定位状态（A—可用，V—可能有错误）、纬度值、经度值、对地速度、日期等
VTG	对地速度等
ZDA	UTC 时间、年、月、日、当地时区、时区的分钟值等

2、PDU 格式

GSM 规范¹对短消息传输定义了三种控制协议：即二进制协议(块模式)，基于字符的 AT 命令接口协议(文本模式)和基于字符的十六进制编码二进制传输块接口协议(PDU 模式)。

块模式（Block mode）是使用二进制编码来传输用户数据的接口协议。为了提高可靠性，它带有差错保护，适合于链接不完全可靠的地区，尤其是要求控制远程设备的情况。它属于 GSM 第一阶段的短消息传输接口协议。目前，PDU 已取代了块模式。

文本模式（Text mode）是使用 AT 命令传输文本数据的接口协议。该模式适合于非智能终端、终端仿真器等。

PDU 模式相当于计算机网络中的分组交换接口协议。这种传送方式能够很平稳地过渡到 GPRS，因此 GSM 规范要求用户尽可能地使用 PDU 模式处理短消息。

PDU 格式分为收短消息和发短消息¹两种基本格式如图 12-1 和 12-2 所示。

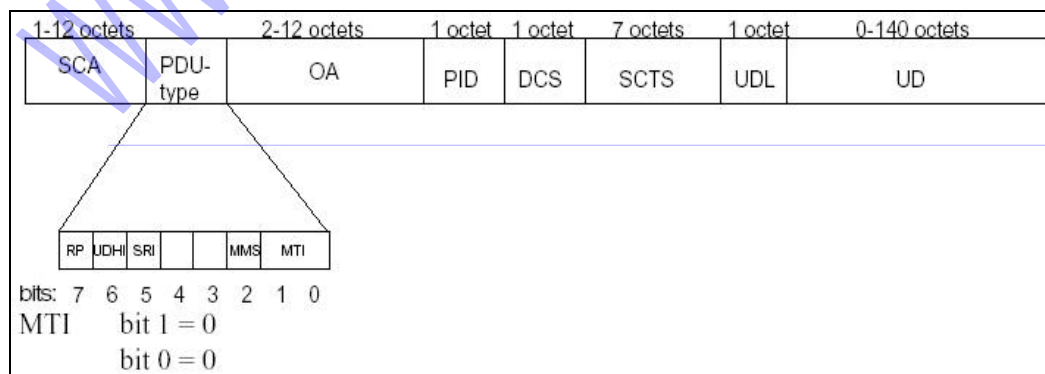


图12-1 收短消息报文格式

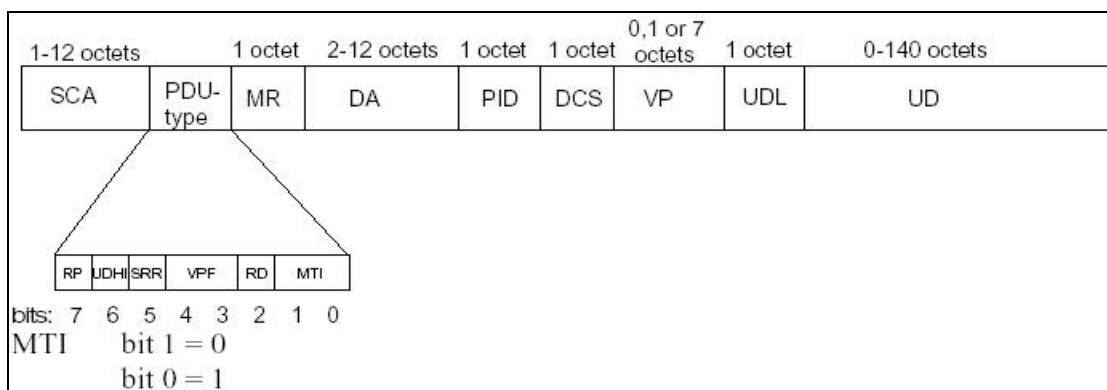


图12-2 发短消息报文格式

SCA: 短消息服务中心的号码。

PDU Type: PDU的消息类型。

MR: 短消息的连续编号（一般短消息终端会对每次发送的SMS-SUBMIT类型的短消息进行编号，从1开始到255结束，然后重新从1开始）

OA: 短消息发送端的地址，即电话号码。

DA : 短消息发送目的端的地址。

PID : 协议指示参数。这个参数主要是提供给SMSC用于判断如何处理这条短消息。

DCS : 数据编码方案。指示了用户数据(User Data)所使用的编码方式。

SCTS: 端消息中心的时间戳。指示了SMSC收到这条短消息的时间。

VP : 有效期。超过这段时间后，短消息在SMSC中不再有效。

UDL : 用户数据长度。指示了UD段的长度。

UD : 用户数据段。

RP: 回复路径指示。标示存在着回复路径。

UDHI : 用户数据段报头指示。标示UD段存在着数据报头。

SRI : 状态报告指示。标示对方端消息实体是否要求有状态报告。

SRR : 状态报告请求。标示本端要求有状态报告。

VPF: 有效期的格式。标示有效期字段是否出现。

MMS : 更多消息指示。标示是否有更多短消息要发送。

RD 拒绝重复指示。后文会有详细说明。

MTI : 消息类型指示。标示消息的类型。

00 为 SMS-DELIVER类型。

01 为 SMS-SUBMIT 类型。

下面以实例说明 PDU 串的结构和编码方式：

(1) **发送时**：短消息中心号码是：+8613800210500，对方号码是：13818413649，消息内容是“Hello!”。从终端发出的 PDU 串为：08 91 68 31 08 20 01 05 F0 11 00 0B 91 31 58 81 27 64 F8 00 00 00 06 C8 32 9B FD 0E 01。

对照规范，具体的分析如表 12-3 所示：

表 12-3 发送 PDU 时的数据格式

分段	含义	说明
08	SMSC 地址信息的长度	共 8 个八位字节(包括 91)
91	SMSC 地址格式(TON/NPI)	用国际格式号码(在前面加 ‘+’)
68 31 08 20 01 05 F0	SMSC 地址	8613800250500, 补 ‘F’ 凑成偶数个

11	基本参数(TP-MTI/VFP)	发送, TP-VF 用相对格式
00	消息基准值(TP-MR)	0
0B	目标地址数字个数	共 11 位, 不包括补足的 ‘F’
91	目标地址格式(TON/NPI)	用国际格式号码(在前面加 ‘+’)
68 31 18 48 31 46 F9	目标地址(TP-DA)	8613818413649, 补 ‘F’ 凑成偶数个
00	协议标识(TP-PID)	是普通 GSM 类型, 点到点方式
00	用户信息编码方式(TP-DCS)	7-bit 编码
00	有效期(TP-VP)	5 分钟
06	用户信息长度(TP-UDL)	实际长度 6 个字节
C8 32 9B FD 0E 01	用户信息(TP-UD)	“Hello!”

(2) 接收时: 短消息中心号码是: +8613800210500, 对方号码是 13851872468, 消息内容是 “华东! ”。手机接收到的 PDU 串 08 91 68 31 08 20 01 05 F0 84 0D 91 68 31 58 81 27 64 F8 00 08 30 30 21 80 63 54 80 06 53 4E 4E 1C 00 21。

对照规范, 具体的分析如表 12-4 所示:

表 12-4 接收 PDU 时的数据格式

分段	含义	说明
08	SMSC 地址信息的长度	共 8 个八位字节(包括 91)
91	SMSC 地址格式(TON/NPI)	用国际格式号码(在前面加 ‘+’)
68 31 08 20 05 05 F0	SMSC 地址	8613800250500, 补 ‘F’ 凑成偶数位
84	基本参数(TP-MTI/MMS/RP)	接收, 无更多消息, 有回复地址
0B	回复地址数字个数	共 11 个十进制数(不包括 ‘F’)
91	回复地址格式(TON/NPI)	用国际格式号码(在前面加 ‘+’)
68 31 58 81 27 64 F8	回复地址(TP-RA)	8613851872468, 补 ‘F’ 凑成偶数位
00	协议标识(TP-PID)	是普通 GSM 类型, 点到点方式
08	用户信息编码方(TP-DCS)	UCS2 编码
30 30 21 80 63 54 80	时间戳(TP-SCTS)	2003-3-12 08:36:45 +8 时区
06	用户信息长度(TP-UDL)	实际长度 6 个字节
53 4E 4E 1C 00 21	用户信息(TP-UD)	“华东!”

3、串口编程 API 函数介绍

串行设备的接口是通用驱动 I/O 调用和特殊通讯相关函数的集合。串行设备被对待为通用, 可安装的流设备(可以开, 关, 读和写串口端口)。为了配置端口, Win32 API 支持一个通讯函数的集合。

3.1 打开、关闭串口

(1) 打开串口

```
hHandle=CreateFile(TEXT("COM1:"),
    GENERIC_READ|GENERIC_WRITE,
    0,
    NULL,
    OPEN_EXISTING,
    0,
    NULL);
```

返回的句柄或者是已打开的串行端口的句柄，或者是 INVALID_HANDLE_VALUE，具体打开哪个端口可以修改第一个参数。

(2) 关闭串口

```
CloseHandle(hHandle);
```

调用 CloseHandle 可以关闭一个串行端口，CloseHandle 只有一个参数，就是调用 CreateFile 打开一个端口时返回的句柄

3.2 配置串行端口

在实际使用串口时，必须对端口配置好正确的波特率、字符长度等等。可以通过 I/O 设备控制调用来配置串行驱动程序，但此操作需要对低层的了解，并且要有相应的“嵌入工具包”（ETK），SDK 不能实现该操作。除此之外，还有一种更简单的方法，就是使用两个函数 GetCommState 和 SetCommState 来配置串行端口。

函数原型如下：

```
BOOL SetCommState(HANDLE hRle,LPDCB lpDCB);
BOOL GetCommState(HANDLE hFile,LPDCB lpDCB);
```

这两个函数均包含两个参数，已打开的串行端口的句柄和指向 DCB 结构的指针。扩展的 DCB 结构的定义如下：

```
typedef struct _DCB {
    DWORD DCBlength;
    DWORD BaudRate;
    DWORD fBinary :1;
    DWORD fParity :1;
    DWORD fOutxCtsFlow :1;
    DWORD fOutxDsrFlow :1;
    DWORD fDtrControl :2;
    DWORD fDsrSensitivity :1;
    DWORD fTXContinueOnXoff :1;
    DWORD fOutX :1;
    DWORD fInX :1;
    DWORD fErrorChar :1;
    DWORD fNull :1;
    DWORD fRtsControl :2;
    DWORD fAbortOnError :1;
    DWORD fDummy2 :17;
    WORD wReserved;
```

```

WORD XonLim;
WORD XoffLim;
BYTE ByteSize;
BYTE Parity;
BYTE StopBits;
char XonChar;
char XoffChar;
char ErrorChar;
char EofChar;
char EvtChar;
WORD wReserved1;
} DCB;

```

从该结构可以看出，SetCommState 能设置多种状态。最好不要从头填写整个结构，而应该使用修改串行端口的办法，即调用 GetCommState 来接收默认的 DCB 结构，之后修改必须的区域，然后调用 SetCommState 来配置新的串行端口。

3.3 读写串口

可以使用 ReadFile 和 WriteFile 函数读取串口数据和向串口中写入数据。

3.4 异步串口 I/O

虽然 Windows CE 不支持重叠 I/O，但可以使用多个线程来执行同样类型的重叠操作。当主线程忙时，需要做的就是运行单独的线程来处理同步 I/O 操作。除了使用用于读和写的单线程以外，Windows CE 还支持 Win32 WaitCommEvent 函数，该函数将阻塞线程，直到预先选择的串行事件中的一个事件发生。

通过以下三个函数，可以使线程等待串行驱动程序事件：

```

BOOL SetCommMask(HANDLE hFile,DWORD dwEvtMask);
BOOL GetCommMask(HANDLE hFile,LPDWORD lpEvtMask);
BOOL WaitCommEvent(HANDLE hFile,LPDWORD lpEvtMask,
                  LPOVERLAPPED lpOverlapped);

```

3.5 设置串口读写超时

在调用 ReadFile 和 WriteFile 从串读取数据和写入数据时，WinCE 提供了超时机制，也就是设置了等待它们返回的时间长度。设置串口超时函数 SetCommTimeouts 的定义：

```

BOOL SetCommTimeouts(
    HANDLE hFile,
    LPCOMMTIMEOUTS lpCommTimeouts
);

```

其中 lpCommTimeouts 指向 COMMTIMEOUTS 结构，设置新的超时值。COMMTIMEOUTS 结构的定义：

```

typedef struct _COMMTIMEOUTS {
    DWORD ReadIntervalTimeout;
    DWORD ReadTotalTimeoutMultiplier;
    DWORD ReadTotalTimeoutConstant;
    DWORD WriteTotalTimeoutMultiplier;
}

```

```
DWORD WriteTotalTimeoutConstant;
}
```

4、串口类的实现

4.1 串口的初始化

```
BOOL CSerial::InitializePort(HWND hWnd, CString PortNum, UINT Baud, UINT Parity, UINT DataBits,
UINT StopBits)
```

```
{
    CString Nnn;
    InitializeCriticalSection(&m_cs); // initialize critical section
    EnterCriticalSection(&m_cs); // now it critical!
    if(m_hPort!=NULL) // Empty Port Handle.
    {
        CloseHandle(m_hPort);
        m_hPort = NULL;
    }

    if(m_hThread!=NULL)
    {
        CloseHandle(m_hThread);
        m_hThread = NULL;
    }

    if(m_hWnd!=NULL)
    {
        CloseHandle(m_hWnd);
        m_hWnd = NULL;
    }

    // format port num.
    //Nnn = PortNum + ":"; 李外云修改
    PortNum+=":";
    Nnn=PortNum;
    // set message Handle.
    m_hWnd=hWnd;
    m_hPort = CreateFile (Nnn, // Pointer to the name of the port
        GENERIC_READ | GENERIC_WRITE,
        // Access (read-write) mode
        0, // Share mode
        NULL, // Pointer to the security attribute
        OPEN_EXISTING, // How to open the serial port
        0, // Port attributes
        NULL); // Handle to port with attribute
```

```

// to copy
if (m_hPort == INVALID_HANDLE_VALUE)
{
    return FALSE;
}
// Get Comm DCB
if(!GetCommState(m_hPort,&m_dcb)) return FALSE;
// Set Comm DCB
m_dcb.BaudRate = Baud;

if(Parity == 0) m_dcb.Parity = 0;
if(Parity == 1) m_dcb.Parity = 1;
if(Parity == 2) m_dcb.Parity = 2;
m_dcb.ByteSize = DataBits;
if(StopBits == 1) m_dcb.StopBits = ONESTOPBIT;
if(StopBits == 1.5) m_dcb.StopBits = ONE5STOPBITS;
if(StopBits == 2) m_dcb.StopBits = TWOSTOPBITS;

if(!SetCommState(m_hPort,&m_dcb)) return FALSE;

// Get Comm Timeouts
if(!GetCommTimeouts(m_hPort,&m_CommTimeouts)) return FALSE;
m_CommTimeouts.ReadIntervalTimeout = MAXDWORD;
m_CommTimeouts.ReadTotalTimeoutMultiplier = 0;
m_CommTimeouts.ReadTotalTimeoutConstant = 0;
m_CommTimeouts.WriteTotalTimeoutMultiplier = 0;
m_CommTimeouts.WriteTotalTimeoutConstant = 0;
if(!SetCommTimeouts(m_hPort,&m_CommTimeouts))
    return FALSE;
// Set Comm Mask
if(!SetCommMask(m_hPort,EV_RXCHAR | EV_CTS | EV_DSR | EV_RING)) // modify nick
// if(!SetCommMask(m_hPort,EV_RXCHAR ))
    return FALSE;
// flush the port
PurgeComm(m_hPort, PURGE_RXCLEAR | PURGE_TXCLEAR | PURGE_RXABORT | PURGE_TXABORT);
// release critical section
LeaveCriticalSection(&m_cs);
return TRUE;
}

```

4.2 写串口

```

BOOL CSerial::WriteChar(BYTE Byte)
{
    DWORD dwNumBytesWritten;

```

```

BOOL bResult;

// Clear Comm Error
PurgeComm(m_hPort, PURGE_RXCLEAR | PURGE_TXCLEAR | PURGE_RXABORT | PURGE_TXABORT);

// write to port.
bResult = WriteFile(m_hPort,                                // Handle to COMM Port
                    &Byte,                                // Pointer to message buffer in calling finction
                    1,                                     // Length of message to send
                    &dwNumBytesWritten, // Where to store the number of bytes sent
                    NULL);

if(!bResult || dwNumBytesWritten == 0) return FALSE;
return TRUE;
}

```

4.3 写串口（采用线程）

```

DWORD WINAPI CSerial::ReadThread(LPVOID lpParam)
{
    CSerial *p=(CSerial *)lpParam;
    DWORD dwCommStatus = 0;           // comm event
    BOOL bResult = FALSE;             // waitcommevent result
    BOOL bClear = FALSE;              // clearcomm result
    BOOL bReadResult = FALSE;         // readfile result
    DWORD dwNumBytesRead = 0;         // number readed
    COMSTAT comstat;                  // COMSTAT struct
    DWORD dwError;                    // error message
    char RXBuff;                      // receive buffer
    char ReceiveBuf[258];             //缓冲区的大小
    int iCounter = 0;
    // check if the port is opened, then clear buffer first.
    if (p->m_hPort)
        PurgeComm(p->m_hPort, PURGE_RXCLEAR | PURGE_TXCLEAR | PURGE_RXABORT | PURGE_TXABORT);
    else
        return 0;
    while (1)
    {
        // Wait for an event to occur for the port.
        memset(ReceiveBuf, 0, 256);
        iCounter=0;
        bResult = WaitCommEvent(p->m_hPort, &dwCommStatus, NULL);
        if (bResult && (dwCommStatus & EV_RXCHAR))
        {
            do

```

```

{
    EnterCriticalSection(&p->m_cs); // now it critical!
    // Clear port Error message.
    bClear = ClearCommError(p->m_hPort, &dwError, &comstat);
    if(comstat.cbInQue ==0) break;
    // clear RX Buffer and count
    dwNumBytesRead = 0;
    RXBuff = 0;
    bReadResult = ReadFile(p->m_hPort ,           // Handle to COMM port
                           &RXBuff,             // RX Buffer Pointer
                           1,                     // Read one byte
                           &dwNumBytesRead, // Stores number of bytes read
                           NULL);

    // release critical section
    LeaveCriticalSection(&p->m_cs);
    // display data.
    if(bReadResult && dwNumBytesRead == 1)
    {
        ReceiveBuf[iCounter++] = RXBuff;
        //接收一个，处理一个
        ::SendMessage(p->m_hWnd, WM_COMM_RXCHAR, (WPARAM) RXBuff, (LPARAM) p->m_hPort);
    }
} while(bReadResult && dwNumBytesRead == 1);
//处理完一帧
::SendMessage(p->m_hWnd, WM_COMM_RXCHAR_ALL, (WPARAM) ReceiveBuf, (LPARAM) iCounter);
}
}
return 1;
}

```

[实验内容]

- 1、掌握串口编程方法;
- 2、掌握 SMS 数据格式和 AT 指令集的使用方法;
- 3、了解 GPS 数据格式

[实验步骤]

GSM 测试实验步骤:

第一步: 连接好实验系统, 打开实验箱电源;

第二步: 利用 Visual Studio 2005.net 打开 SMS 测试工程文件 SMSTest.sln

进行编译：点击运行按钮，将 SMS 测试程序下载到 XSBase270 目标板运行，运行界面如图 12-3 所示；



图 12-3 GSM 测试程序界面

第三步：将 GSM 模块接到目标板的 BT 串口上，在手机号对应的文本框中输入对方手机号，然后进行 SMS 测试实验。（注意：服务号要根据不同地域进行设置，图示的服务号 1380210500 为上海移动的短信服务号）。

GPS 测试实验步骤：

第一步：利用 Visual Studio 2005.net 打开 GPS 测试工程文件 GPSTest.sln 进行编译：点击运行按钮，将 GPS 测试程序下载到 XSBase270 目标板运行，运行界面如图 12-3 所示；

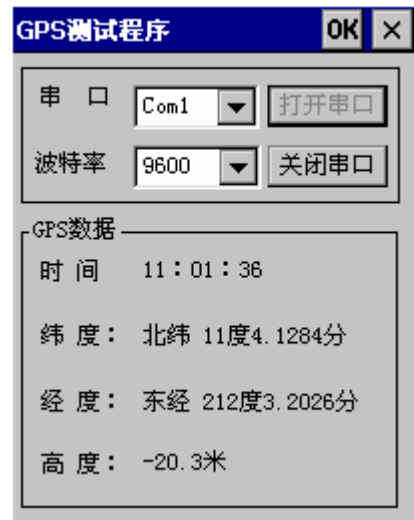


图 12-4 GPS 测试程序运行界面

第二步：将 GPS 模块接到目标板的 BT 串口上进行 GPS 测试实验。

[习题与思考题]

- 1、分析串口类程序，完成利用多线程实现串口的写操作；
- 2、GSM 模块的波特率可以通过 AT 指令集进行设置，修改应用程序完成自动测试 GSM 模块的波特率。
- 3、在 GPS 程序中，修改源程序，完成利用其他的 GPS 格式来实现上述参数的读取。

实验十三 CAN 总线实验

[实验目的]

- 1、了解流式设备驱动的编写过程；
- 2、了解 CAN 总线的基本原理
- 3、熟悉 EVC 和 VS.Net 的开发环境；

[实验仪器]

- 1、装有 Platform Builder、EVC 和 VS.Net 开发平台的 PC 机一台
- 2、XSBase270 实验开发平台一套
- 3、CAN232MB 模块 一块或两台 XSBase270 平台

[实验原理]

1、CAN 总线简介：

CAN 总线是德国 BOSCH 公司在 20 世纪 80 年代初，为了解决现代汽车中众多的控制与测试仪器之间的数据交换而开发的一种串行数据通讯协议。它的短帧数据结构、非破坏性总线性仲裁技术以及灵活的通讯方式适应了汽车的实时性和可靠性要求。

1.1 CAN 总线的特点

CAN 作为一种多主总线，支持分布式实时控制的通讯网络。其通讯介质可以是双绞线、同轴电缆或光纤。在汽车发动机控制部件、传感器、抗滑系统等应用中，总线的位速率最大可达 1Mbit/s。CAN 总线属于总线式串行通讯网络，由于其采用了许多新技术及独特的设计，与一般的通讯总线相比，CAN 总线的数据通讯具有突出的可靠性、实时性和灵活性。其特点可以概括如下：

- (1) CAN 为多主方式工作，网络上任一节点均可在任意时刻主动地向网络上其他节点发送信息，而不分主从，通信方式灵活，且无需站地址等节点信息。利用这一点可方便地构成多机备份系统。
- (2) CAN 网络上的节点信息分成不同的优先级，可满足不同的实时要求，高优先级的数据最多可在 134us 内得到传输。
- (3) CAN 采用非破坏性总线性仲裁技术，当多个节点同时向总线发送信息时，优先级较低的节点会主动地退出发送，而最高优先级的节点可不受影响地继续传输数据，从而大大节省了总线冲突仲裁时间。尤其是在网络负载很重的情况下也不会出现网络瘫痪情况（以太网则可能）。
- (4) CAN 只需通过帧滤波即可实现点对点、一点对多点及全局广播等几种方式传送接受数据，无需专门的“调度”。

- (5) CAN 采用 NRZ 编码, 直接通信距离最远可达 10km (速率 5kbps); 通信速率最高可达 1Mbps(此时通信距离最长为 40m)。
- (6) CAN 上的节点数主要取决于总线驱动电路, 目前可达 110 个; 标示符可达 2032 种 (CAN2.0A), 而扩展标准(CAN2.0B)的标示符几乎不受限制。
- (7) CAN 采用短帧结构, 传输时间短, 受干扰概率低, 具有极好的检错效果, 每帧信息都有 CRC 效验及其他检错措施, 保证数据出错率极低。
- (8) CAN 的通信介质可为双绞线、同轴电缆或光纤, 选择灵活。
- (9) CAN 节点在错误严重的情况下具有自动关闭输出功能, 以使总线上其他节点的操作不受影响。

1.2 CAN 总线技术的优点

使用 CAN 总线后, 对其优点进行了总结, 得出以下结论:

- (1) 如果数据扩展以增加新的信息, 只需升级软件即可。
- (2) 控制单元对所传输的信息进行实时检测, 检测到故障后存储故障码。
- (3) 使用小型控制单元及小型控制单元插孔可节省空间。
- (4) 使传感器信号线减至最少, 控制单元可做到高速数据传输。
- (5) CAN 总线符合国际标准, 因此可应用不同型号控制单元间的数据传输。

1.3 CAN 典型系统框图

图 13-1 为 CAN 总线系统的框图

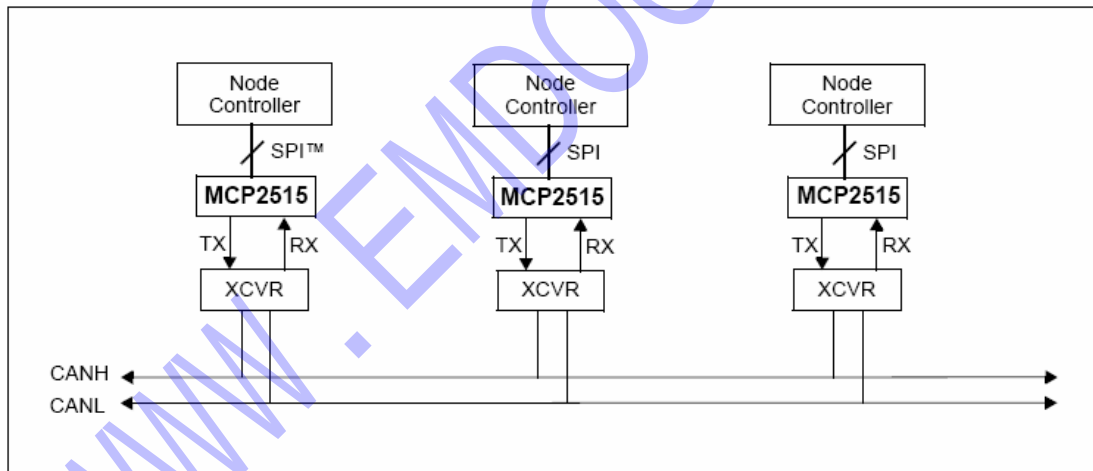


图 13-1 can 典型系统框图

1.4 XSBASE270 目标板的 CAN 总线接口硬件图 (如图 13-2)

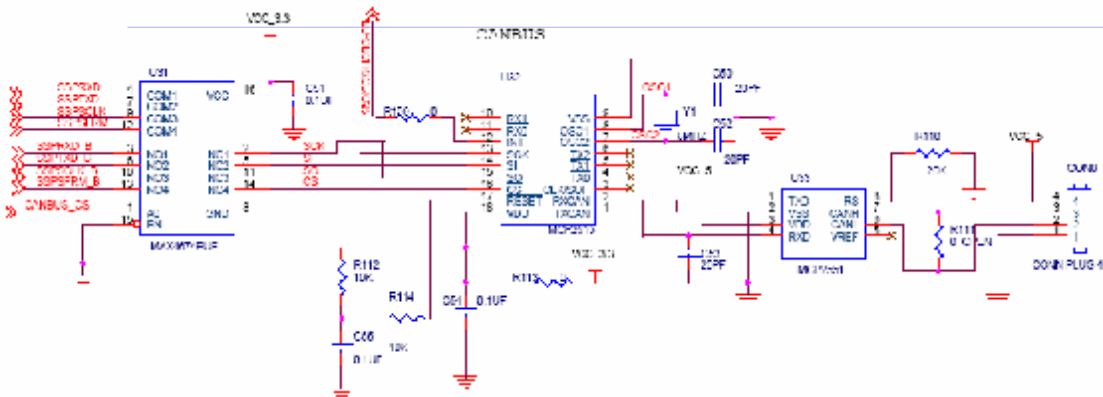


图 13-2 目标板的 CAN 总线的硬件接口图

2、Wince 下的 CAN 总线驱动

在 wince 下，CAN 驱动采用标准的流驱动接口来实现。从硬件接口图（图 13-2）可知，利用 PXA270 的 SPI 控制 CAN 总线驱动芯片 MCP2515，所以驱动程序包括 PXA27x 的 SPI 编程控制和 CAN 总线驱动芯片 MCP2515 的传输协议的控制。

2.1 PXA27x 的 SPI 传输协议

PXA27x 的 SPI 传输协议比较简单，SPI 协议的传输时序如图 13-3 所示。

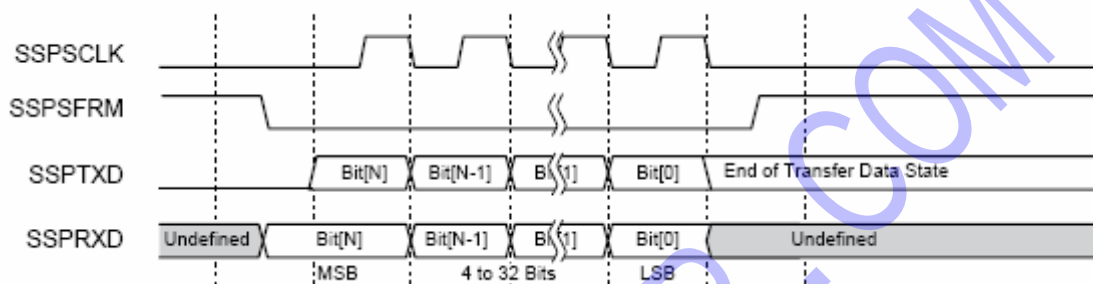


图 13-3 PXA270 的 SPI 传输协议时序图

2.2 MCP2515 的传输协议

CAN 总线驱动芯片 MCP2515 的传输协议比较复杂，主要包括 CAN 协议报文的接收和发送。图 13-4 和图 13-5 分别为 CAN 总线驱动芯片 MCP2515 的发送和接收流程图。

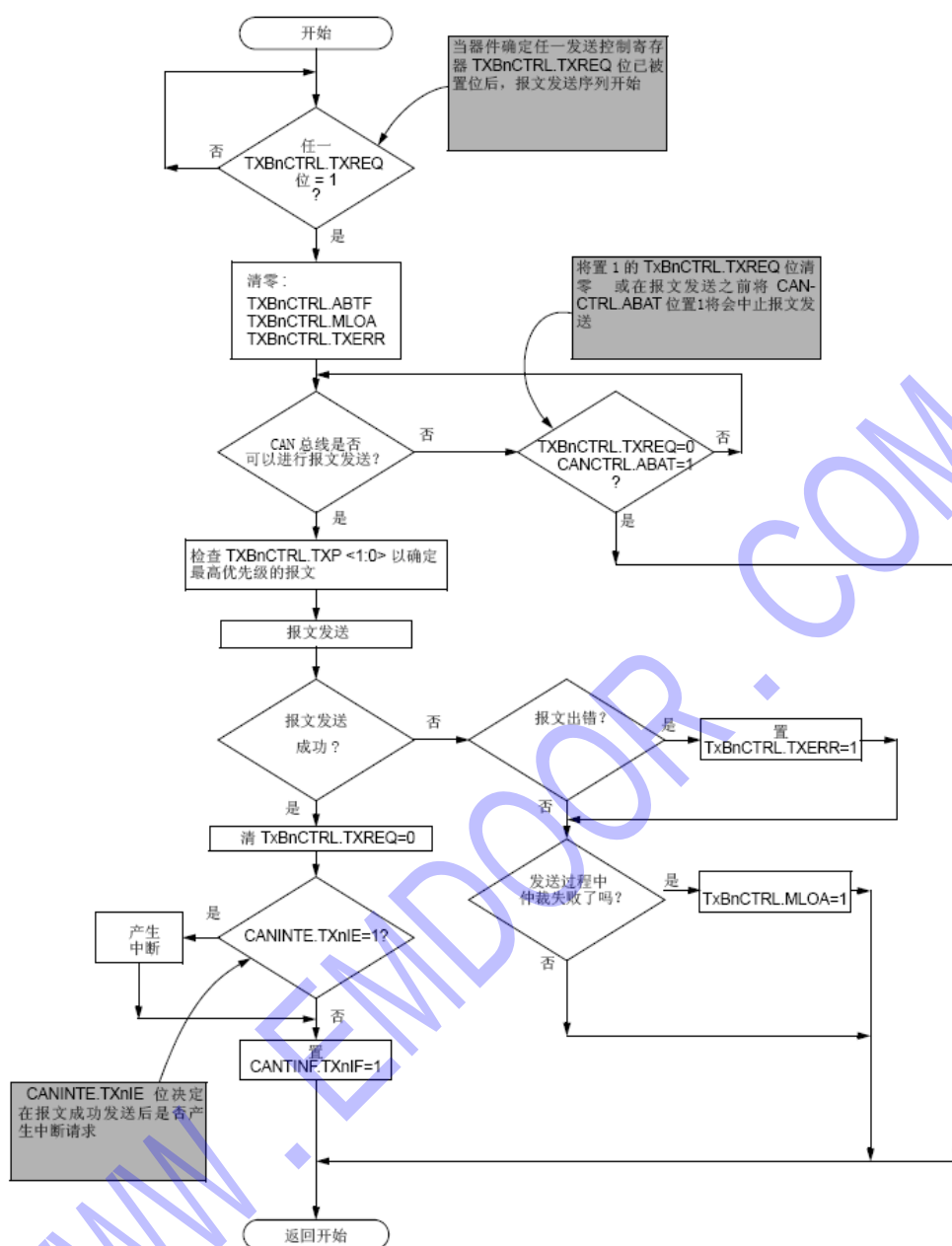


图 13-4 CAN 总线报文发送流程

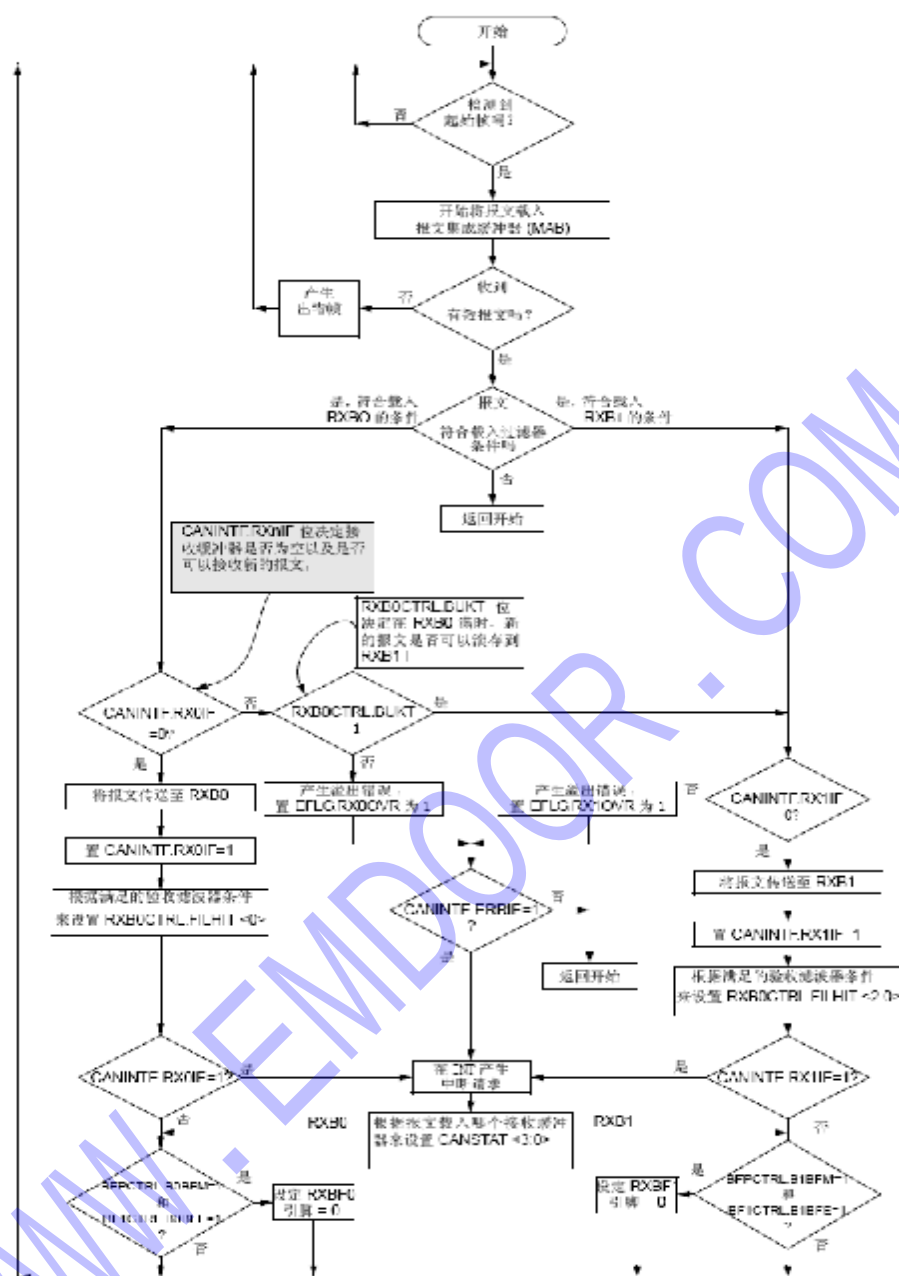


图 13-5 CAN 总线报文接收流程

2.3 CAN总线驱动分析

CAN 总线的驱动分为两部分，一部分是实现 SPI 控制传输，另一部分用来实现 CAN 总线数据的传输。其中 SPI 控制传输实际上完成对 SPI 控制协议的数据的串行传输，相对比较简单，主要包括 SPI 的 1~4 个字节的发送函数，即 OneByteSPITransaction、TwoByteSPITransaction、ThreeByteSPITransaction 和 FourByteSPITransaction。具体参考系统提供的 BSP 源代码中的 CAN 驱动源代码。

(1) Can 驱动的入口函数

由于驱动采用 wince 的流驱动模式，所以入口函数应该是 CAN Init()。

```
if (!hUserEvent = CreateEvent(NULL, FALSE, FALSE, sUserEventName)))
    goto InitFail;//这里创建一个有名事件，提供给应用程序使用
if (!InitGPIO()) //设置好 GPIO，使能 CPU 的 SPI 接口
```

```

        goto InitFail;
    if (!InitSSP()) //初始化 SSP，设置 SSP 工作在 SPI 模式下，用来控制 MCP2515
        goto InitFail;
    InitMCP2510(NOTINPOWERHANDLER); //初始化 MCP2515

```

(2) 读写函数

在初始化完成后，就可以开始传输数据了。

```

DWORD CAN_Read(DWORD hOpenContext, PBYTE pBuffer, DWORD Count)
{ //这是读数据函数
    CAN_MESSAGE *message; //这个结构包含了从 CAN 读出数据的格式
    if ((pBuffer == NULL) || (Count != sizeof(CAN_MESSAGE)))
        return (ULONG)-1;
    message = (CAN_MESSAGE *)pBuffer;
    CanGetMessage(message, gRXChannel, NOTINPOWERHANDLER); //读数据
    return (sizeof(CAN_MESSAGE));
}

```

```

DWORD CAN_Write(DWORD hOpenContext, PBYTE pBuffer, DWORD Count)
{
    CAN_MESSAGE *message; //这个结构包含了从 CAN 读出数据的格式
    UINT flg=0;
    if ((pBuffer == NULL) || (Count != sizeof(CAN_MESSAGE)))
        return (ULONG)-1;
    message = (CAN_MESSAGE *)pBuffer;
    flg = CanTransmitMessage(message, gTXChannel, NOTINPOWERHANDLER);
    if (!flg)
        return (sizeof(CAN_MESSAGE));
    else
        return (ULONG)-1;
}

```

2.4 CAN 测试程序分析

(1) 打开设备

```

void CCANAppDlg::OnBnClickedbtnopen()
{
    m_hFile = CreateFile(L"CAN1:",
                        GENERIC_READ | GENERIC_WRITE,
                        0, NULL,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        NULL);
    if (m_hFile == INVALID_HANDLE_VALUE)
        MessageBox(_T("打开设备错误!"), _T("系统提示"), MB_OK);
}

```

(2) 发送数据

```
void CCANAppDlg::OnBnClickedbtnsend()
{
    CString str;
    DWORD dw;
    int loopnum, datanum, temp;
    if(m_hFile) {
        GetDlgItemText(IDC_SendData, str);
        CAN_MESSAGE SendMsgData = {0};
        SendMsgData.receive_dlc = SendMsgData.send_dlc = 8;
        SendMsgData.send_id = GetDlgItemInt(IDC_ReceiveID);
        SendMsgData.receive_id = GetDlgItemInt(IDC_SendID);
        if((datanum = str.GetLength()) > 8)
            datanum = 8;
        loopnum = str.GetLength() / 8;
        if((str.GetLength() % 8) == 0) loopnum -= 1;
        temp = loopnum;
        while(loopnum >= 0) {
            for(int i = 0; i < datanum; i++)
            {
                SendMsgData.send_buf[i] = str.GetAt(i + 8 * (temp - loopnum));
            }
            WriteFile(m_hFile, &SendMsgData, sizeof(CAN_MESSAGE), &dw, NULL);
            loopnum--;
        }
    }
    else
        MessageBox(_T("CAN设备没有打开"), _T("系统提示"), MB_OK);
}
```

(3) 接收数据

```
void CCANAppDlg::OnBnClickedbtnreceive()
{
    if(m_hFile)
    {
        DWORD dw;
        CAN_MESSAGE RecvMsgData = {0};
        ReadFile(m_hFile, &RecvMsgData, sizeof(CAN_MESSAGE), &dw, NULL);
        CString str;
        str.Format(L"receive_id=0x%x\r\nreceive_dlc=0x%x\r\n",
            RecvMsgData.receive_id, RecvMsgData.receive_dlc);
        str.Format(L"%sreceive_buf[8]=%c%c%c%c%c%c%c\r\n", str,
            RecvMsgData.receive_buf[0],
            RecvMsgData.receive_buf[1],
            RecvMsgData.receive_buf[2],
```

```
        RecvMsgData.receive_buf[3],
        RecvMsgData.receive_buf[4],
        RecvMsgData.receive_buf[5],
        RecvMsgData.receive_buf[6],
        RecvMsgData.receive_buf[7]);
    SetDlgItemText(IDC_Receive, str);
}
else
    MessageBox(_T("CAN设备没有打开"), _T("系统提示"), MB_OK);
}
```

[实验内容]

- 1、了解 CAN 总线的基本原理;
- 2、掌握流式驱动程序的基本编程和 CAN 总线驱动的配置过程;
- 3、掌握利用 VS.net 实现 CAN 协议数据传输的编程过程和方法

[实验步骤]

CAN 总线驱动配置:

第一步: 连接好实验系统, 打开实验箱电源;

第二步: 编译带 CAN 总线驱动的镜像, 在 P B 的 catalog 中, 把 CAN 驱动加到工程中去 (如图 13-6)

第三步: 重新生成映像文件, 并将映像文件下载到目标板

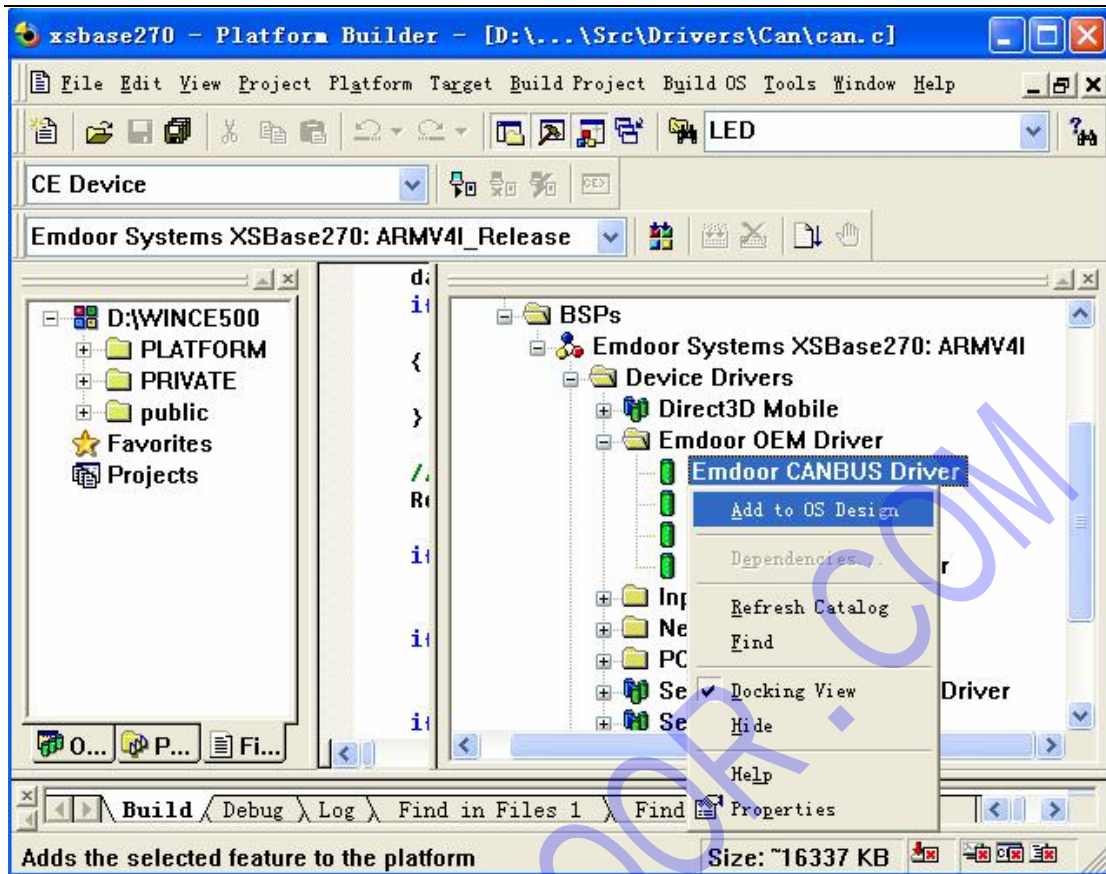


图 13-6 CAN 驱动的配置

CAN 驱动测试程序:

第一步: 利用 VS2005.net 打开 CAN 应用测试工程文件 CANApp.sln, 进行编译: 点击运行按钮, 将 CAN 总线应用测试程序下载到 XSBase270 目标板运行, 运行界面如图 13-7 所示:



图 13-7 CAN 测试程序运行界面

第二步：用双绞线把目标平台和 CAN232MB 连接好，并利用串口线将 CAN 模块连到 PC 机，打开串口调试助手。

第三步：按“打开设备”按钮，测试程序将打开 CAN 设备。然后按“发送数据”和“接收按钮”进行数据的接收和发送。

（注：如果没有 CAN 模块，可以将两个平台的 can 接口连接好，然后分别将测试程序下载到两个平台上进行测试）。

[习题与思考题]

- 1、如何使用中断方式来发送接收数据？
- 2、如何设计驱动，才能使得在应用程序中不需要判断数据是否大于 8 字节？

WWW.EMDOOR.COM

实验十四 SD/CF 存储卡读写实验

[实验目的]

- 1、了解 SD/CF 卡存储基本知识;
- 2、掌握 WinCE 中 SD/CF 卡的配置方法;
- 3、掌握对 SD/CF 卡进行文件读写操作的编程方法
- 4、熟悉 Platform Builder 配置系统环境和 EVC、VS.Net 的开发环境;

[实验仪器]

- 1、装有 Platform Builder、EVC 和 VS.Net 开发平台的 PC 机一台
- 2、XSBase270 实验开发平台一套
- 3、SD 和 CF 存储卡各一块

[实验原理]

1、SD/CF 存储卡简介

1.1 Secure Digital Memory Card 简介:

SD 卡 (Secure Digital Memory Card) 是一种基于半导体快闪记忆器的新一代记忆设备。SD 卡由日本松下、东芝及美国 SanDisk 公司于 1999 年 8 月共同开发研制。大小犹如一张邮票的 SD 记忆卡,重量只有 2 克 (如图 14-1 所示),但却拥有高记忆容量、快速数据传输率、极大的移动灵活性以及良好的安全性。

SD 卡在 $24\text{mm} \times 32\text{mm} \times 2.1\text{mm}$ 的体积内结合了 SanDisk 快闪记忆卡控制与 MLC (Multilevel Cell) 技术和 Toshiba (东芝) 0.16u 及 0.13u 的 NAND 技术,通过 9 针的接口界面与专门的驱动器相连接,不需要额外的电源来保持其上记忆的信息。而且它是一体化固体介质,没有任何移动部分,所以不用担心机械运动的损伤。



图 14-1 SD 存储卡

SD 卡数据传送和物理规范由 MMC 发展而来,大小和 MMC 差不多,尺寸为 $32\text{mm} \times 24\text{mm} \times 2.1\text{mm}$ 。长宽和 MMC 一样,只是厚了 0.7mm,以容纳更大容量的存储单元。SD 卡与 MMC 卡保持着向上兼容,也就是说,MMC 可以被新的 SD 设备存取,兼容性则取决于

应用软件，但 SD 卡却不可以被 MMC 设备存取。

SD 接口除了保留 MMC 的 7 针外，还在两边加多了 2 针，作为数据线。采用了 NAND 型 Flash Memory，基本上和 SmartMedia 的一样，平均数据传输率能达到 2MB/s。

SD 卡的结构保证了数字文件传送的安全性，也很容易被重新格式化，所以有着广泛的应用领域，音乐、电影、新闻等多媒体文件都可以方便地保存到 SD 卡中。因此不少数码相机也开始支持 SD 卡。

1.2 Compact Flash 简介:

CF 卡 (Compact Flash) (如图 14-2 所示) 是 1994 年由 SanDisk 最先推出的。CF 卡具有 PCMCIA-ATA 功能，并与之兼容；CF 卡重量只有 14g，仅纸板火柴般大小 (43mm×36mm×3.3mm)，是一种固态产品，也就是工作时没有运动部件。CF 卡采用闪存 (Flash) 技术，是一种稳定的存储解决方案，不需要电池来维持其中存储的数据。对所保存的数据来说，CF 卡的安全性和保护性较传统的磁盘驱动器都更高；可靠性比传统的磁盘驱动器及 III 型 PC 卡高 5 到 10 倍，而且 CF 卡的用电量仅为小型磁盘驱动器的 5%。这些优异的条件使得大多数数码相机选择 CF 卡作为其首选存储介质。



图 14-2 CF 卡示例

虽然最初 CF 卡是采用 Flash Memory 的存储卡，但随着 CF 卡的发展，各种采用 CF 卡规格的非 Flash Memory 卡也开始出现，CFA 后来又发展出了 CF+ 的规格，使 CF 卡的范围扩展到非 Flash Memory 的其它领域，包括其它 I/O 设备和磁盘存储器，以及一个更新物理规格的 Type II 规格 (IBM 的 Microdrive 就是 Type II 的 CF 卡)，Type II 和原来的 Type I 相比不同之处在于 Type II 厚 5mm。

CF 卡同时支持 3.3 伏和 5 伏的电压，任何一张 CF 卡都可以在这两种电压下工作，这使得它具有广阔的使用范围。CF 存储卡的兼容性还表现在它把 Flash Memory 存储模块与控制器结合在一起，这样使用 CF 卡的外部设备就可以做得比较简单，而且不同的 CF 卡都可以用单一的设备来读写，不用担心兼容性问题，特别是 CF 卡升级换代时也可以保证旧设备的兼容性。

2、实验平台的配置

为了开发平台支持 SD 和 CF 存储卡，在编译 WinCE 操作系统映像文件时，必须对 SD/CF 卡的支持进行配置。

2.1 SD 配置:

将 Device Drivers->SDIO->SD Memory 选项添加到平台中，具体配置如图 14-3 所示。

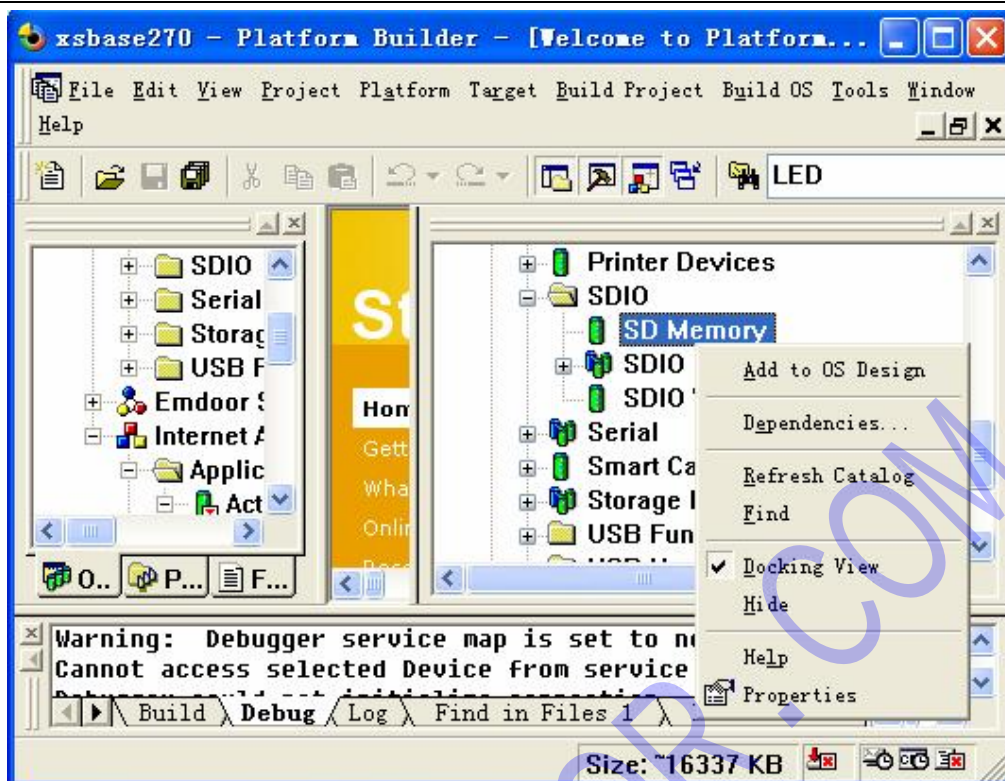


图 14-3 SD 卡的配置

2.2 CF 卡的配置

将 Device Drivers->Storage Devices->Compact Flash/PC Card Storage 选项添加到平台中，具体配置如图 14-4 所示。

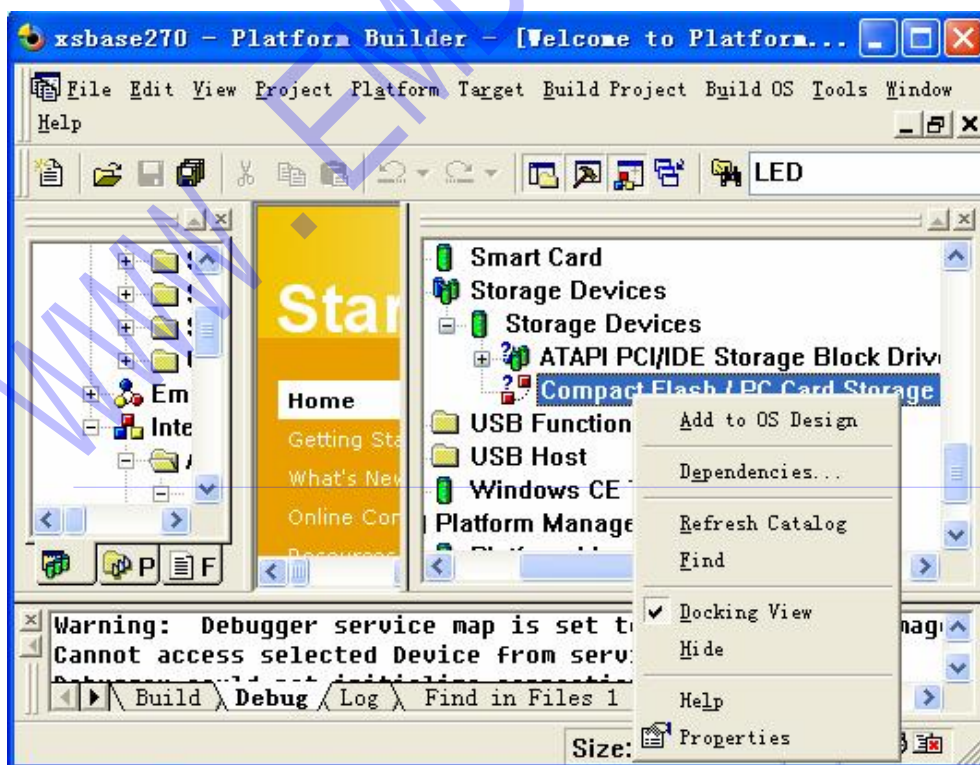


图 14-4 CF 卡的配置

配置完成后，重新生成 WinCE 操作系统的映像文件，并下载到目标板上。当 CF 和 SD

卡插入到开发平台时，SD/CF 存储卡能够被 Windows CE 操作系统识别并在根目录下以“Storage Card”或“存储卡”这个特殊的设备目录存在如图 14-5 所示。双击该目录图标可以打开该目录



图 14-5 SD/CF 卡的在 WinCE 中的目录

3、SD/CF 存储卡的读写操作实现

SD/CF 存储卡得到了业界众多厂商的支持，Windows CE 也提供了对 CF 卡的支持。插入的 SD/CF 存储卡能够被 Windows CE 操作系统识别并在根目录下以“Storage Card”这个特殊的设备目录存在。双击该目录图标可以打开该目录。

下面将通过 VS.net 来实现利用程序打开 SD/CF 卡对应的目录，并完成对 SD/CF 卡的读写操作。打开 SD/CF 卡对应目录，将使用系统 API 为 ShellExecuteEx，其函数原型如下：

```
WINSHELLAPI BOOL WINAPI ShellExecuteEx(
    LPSHELLEXECUTEINFO lpExecInfo
);
```

它执行指定的命令，这个命令可以打开一个文件、目录、应用程序或者其他文档，命令由参数 *lpExecInfo* 来指定。在这里定义一个数据结构：SHELLEXECUTEINFO *sei*，SHELLEXECUTEINFO 的类型如下：

```
typedef struct _SHELLEXECUTEINFO {
    DWORD cbSize;
    ULONG fMask;
    HWND hwnd;
    LPCTSTR lpVerb;
    LPCTSTR lpFile;
    LPCTSTR lpParameters;
    LPCTSTR lpDirectory;
    int nShow;
    HINSTANCE hInstApp;
    LPVOID lpIDList;
    LPCTSTR lpClass;
    HKEY hkeyClass;
    DWORD dwHotKey;
    HANDLE hIcon;
    HANDLE hProcess;
} SHELLEXECUTEINFO, FAR* LPSHELLEXECUTEINFO;
```

必须设置的属性有 *cbSize*、*lpVerb* 等，设置完毕后就可以调用函数 *ShellExecuteEx* 来执行相

关的命令了。具体实现函数：

```
void CSDCFTestDlg::OnBnClickedbtnview()
{
    UpdateData(TRUE);
    CString strPath=_T("\\");
    int ret;
    strPath+=m_strPathName;
    LPTSTR lpPath = new TCHAR[strPath.GetLength()+1];
    _tcscpy(lpPath, strPath);
    SHELLEXECUTEINFO sei;
    sei.cbSize =sizeof (SHELLEXECUTEINFO);
    sei.lpVerb = TEXT("open");
    sei.lpFile =lpPath;
    ret = ShellExecuteEx(&sei);
}
```

将通过 VS.net 的 CFile 类对 SD/CF 卡的进行读写操作，Cfile 类是 Windows CE 的 MFC 的基类，Cfile 类和它的派生类为操作系统提供通用磁盘 I/O。Cfile 类提供了无缓冲的、二进制磁盘 I/O 服务，并且通过它的派生类间接的提供对文本文件、内存文件的支持。

对 SD/CF 卡的读写操作函数：

读操作：

```
void CSDCFTestDlg::OnBnClickedbtnopen()
{
    char buf[1024];
    CString strFileName;
    TCHAR szFilter[] = _T("Txt File (*.txt)|*.txt|All Files (*.*)|*.*||");
    CFileDialog dlg(FALSE, _T("txt"), _T("Test"), OFN_PATHMUSTEXIST, szFilter, this);
    if(dlg.DoModal()==IDOK)
        strFileName=dlg.GetPathName();
    else
        return;
    CFile file;
    file.Open(strFileName, CFile::modeRead);
    file.Read(buf, sizeof(buf));
    m_strContent=buf;
    UpdateData(FALSE);
    file.Close();
}
```

写操作：

```
void CSDCFTestDlg::OnBnClickedbtnwrite()
{
    UpdateData(TRUE);
    CString strFileName;
    TCHAR szFilter[] = _T("Txt File (*.txt)|*.txt|All Files (*.*)|*.*||");
    CFileDialog dlg(FALSE, _T("txt"), _T("Test"), OFN_PATHMUSTEXIST, szFilter, this);
```

```
if(dlg.DoModal()==IDOK)
    strFileName=dlg.GetPathName();
else
    return;
CFile file;
USES_CONVERSION;
char* strRead=W2A(m_strContent);
file.Open(strFileName,CFile::modeCreate|CFile::modeWrite );
file.Write(strRead,strlen(strRead));
file.Close();
}
```

[实验内容]

- 1、在 PB 环境中对 SD/CF 进行配置；
- 2、掌握利用程序对 CF/SD 卡进行读写操作；
- 3、了解利用程序预览系统目录的方法；

[实验步骤]

- 第一步：**连接好实验系统，打开实验箱电源；
- 第二步：**运行 PB，打开一个工作区，配置 SD/CF，编译系统生成系统映像文件，并下载到目标板。
- 第三步：**将 SD 和 CF 存储卡插入开发平台 SD 和 CF 插槽中，查看 WinCE 的 SD/CF 目录。
- 第四步：**打开 SD/CF 存储卡测试工程文件 SDCFTTest.sln，编译该工程文件，点击运行按钮，测试程序的运行界面如图 14-6 所示；



图 14-6 SD/CF 存储卡测试程序运行界面

第五步：SD/CF 存储卡测试程序界面的操作方法

- (1)、存储卡目录的预览：在文件夹的文本框中输入 WinCE 操作系统中 SD/CF 存储卡的目录，然后点击“预览文件夹”按钮，程序将打开 SD/CF 存储卡的目录；
- (2)、文件的读写：单击“读文件”按钮，利用打开文件对话框打开一个文本文件，并将文件内容显示在文本框中；单击“写文件”按钮，利用打开文件对话框打新建一个文件，并将文本框中内容写入新建的文件中。

[习题与思考题]

- 1、在测试中，文件的读写大小只有 1K，如果需要对不同大小的文件进行读写操作，应该怎样进行文件操作；
- 2、如果需要读二进制文件进行读写操作，应该怎样进行文件操作；

实验十五 USB 摄像头驱动和应用实验

[实验目的]

- 1、了解流式设备驱动的编写过程；
- 2、了解 USB 总线的基本概念和 USB 摄像头驱动的基本过程
- 3、掌握对 USB 摄像头的控制方法
- 4、熟悉 EVC 和 VS.Net 的开发环境；

[实验仪器]

- 1、装有 Platform Builder、EVC 和 VS.Net 开发平台的 PC 机一台
- 2、XSBase270 实验开发平台一套
- 3、罗技 Pro5000 摄像头一个

[实验原理]

1、USB 总线简介：

USB 的全称是 Universal Serial Bus，中文含义是“通用串行总线”，它广泛应用于计算机领域的各个方面。1994 年 11 月，Intel、Compaq、Microsoft 等 7 家公司推出了 USB 协议规范的第一个草案，在以后的 10 年时间里，USB 协议版本从 1.1 过渡到 2.0，设备传输速度得到了很大的提高，而 USB 也广泛应用于计算机外设和 PC 领域。

USB 被称为“通用”串行总线，具有很多优秀的特点，例如：

- 热插拔、自动检测、自动配置、即插即用；
- 低功耗设计；
- 接口相同，标准统一；
- 总线供电，可以减少设备端的体积与应用难度；
- 具有很好的扩展性，可以接入基于标准的大量不同种类设备，可以在一个 USB 总线上接入 127 个设备；
- 提供四种不同的数据传输类型，适合各种不同外围设备的要求。

1.1 USB 总线拓扑结构

USB 系统主要被分为三部分：USB 的互连、USB 的设备、USB 的主机。其总线布局为有层次的星型结构。每个 USB 系统中，Host 也就是主机控制器是整个系统中唯一的控制部分，作为整个总线的根节点。而根集线器为 Host 提供了扩展，使它可以连接多个 Slave 和集线器。而集线器进一步扩展了这个层次结构。在每个层次中，集线器作为星型结构的中心，为下一级 Slave（设备）和集线器提供了扩展的可能。具体的总线拓扑结构如图 1：

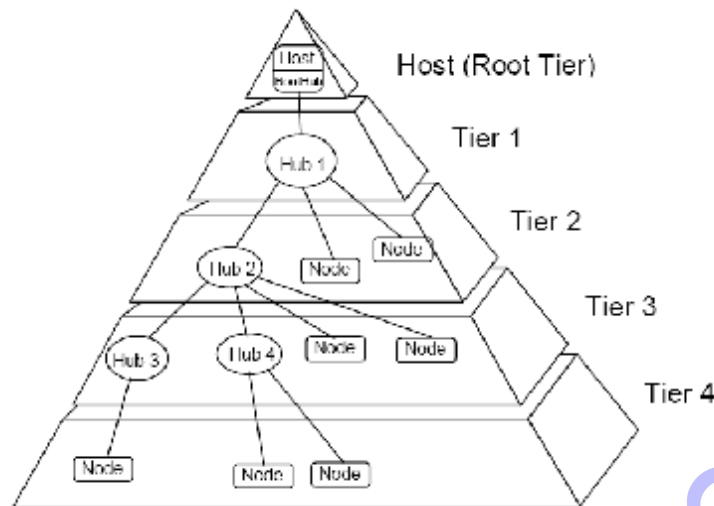


图 1 USB 总线拓扑结构

1.2 USB 设备端的基本属性

USB Slave，也就是 USB 设备，是 USB 总线上的从属部分，USB 标准将 USB 集线器也归入 USB 设备的一种，因为它具有 USB 设备的普遍特性，但是由于它的功能还具有一定特殊性，因此我们这里的 USB Slave 端暂时不包括集线器。

USB 设备通过总线与 USB 主机相连，它们根据属性不同完成不同的功能，如键盘、显示器、鼠标、扬声器等。它们以从属的方式与 USB 主机进行通信，并受 USB 主机的控制，按照 USB 主机的要求发送与接收数据。

为了体现 USB 的通用性，协议为 USB 设备定义了若干属性：描述符(Descriptor)、类(Class)、功能(Function)/接口(Interface)、端点(Endpoint)、管道(Pipe)和设备地址(Device Address)。USB 设备就是利用这些属性各自区别并被 USB Host 识别与控制。

下面简要介绍一下各种属性的作用：

- 描述符 (Descriptor)：用来描述设备的属性与特点，USB Host 就是通过描述符来识别不同种类的设备。
- 类 (Class)：USB 协议以其通用性支持多种外围设备，为了驱动这些设备，USB 主机端需要为这些设备提供符合协议的驱动。为了减少驱动开发的困难，协议将设备归纳划分为几种不同的设备类，把功能相近的设备归为一类。
- 功能 (Function) / 接口 (Interface)：功能就是具有某种能力的设备，传统的设备一般具有单一的功能，而随着技术的发展，一个物理设备可能具有多种“功能”。从设备硬件角度来说，“功能”又称为“接口”，接口描述符中提供了接口所属的设备类。
- 端点 (Endpoint)：端点位于 USB 设备中与 USB 主机进行通信的基本单元。端点的作用与网络协议中的 SAP (服务访问点) 类似，设备通过端点完成和 USB 主机端的数据通信。每个设备允许有多个端点，而每个端点只支持一种传输方式，不同的端点标示了不同的端点号。因此，对于四种不同的传输方式来说，每一种传输方式都有若干端点来完成传输。
- 管道 (Pipe)：管道为 USB 设备与 USB 主机之间数据通信的逻辑通道，管道的物理介质就是 USB 系统中的数据线。
- 设备地址 (Device Address)：USB 主机的客户端驱动程序通过描述符区别不同种类的设备，而 USB 主机控制器通过设备地址来区分不同设备。

1.3 USB 设备端的识别与通信

USB 设备的识别过程，包括 USB 总线枚举过程、设备类配置过程等。而 USB 的通信过程，则按层次依次分为信号层、协议层和数据传输层进行。下面分别对上述过程进行描述。

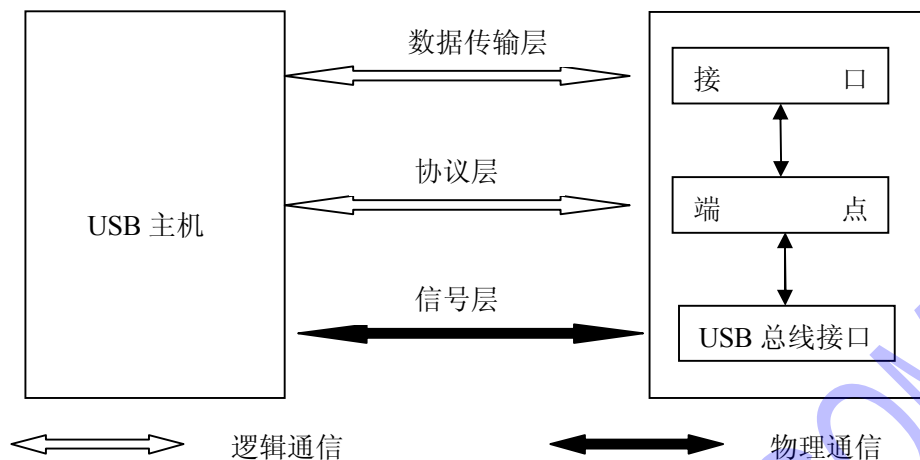


图 2 USB 通信的逻辑结构

a) USB 设备的识别过程:

总线枚举过程:

当一个新的 USB 设备接入集线器 (HUB) 的某个端口上，集线器就会通过“状态改变管道”向 USB 主机 (USB Host) 报告新的设备的接入。主机询问集线器确认新设备的接入后，等待一段时间后，向端口发出复位命令并使能该端口。

在端口复位完毕后，该端口就有效了，此时 USB 设备出于默认状态，地址为 0。接着主机给设备分配一个唯一的地址，设备进入有地址状态。

主机从设备中读取所有配置描述符，并且根据读取的配置描述符为设备指定一个配置。这样设备就可以得到所需要的电量和资源，设备已经准备就绪。

设备类配置:

总线枚举完毕后，从设备的角度讲，它已经可以正常工作了，但是主机尚未为该设备的不同接口分配具体的客户端驱动程序。因此此时主机端协议软件找到设备中每个接口所需要的驱动程序，然后驱动程序从接口的选择设置中选出最合适的，为接口中端点创建管道。

如此完成所有接口的配置，设备的配置过程就彻底完成了。USB 设备就像非 USB 设备一样传输数据了。

USB 通信原理

USB 的通信就是指 USB 设备与 USB 主机之间的通信。物理上，总线上的设备通过一条物理连线和主机通信，所有的设备共享这个物理链路。逻辑上，主机给每个设备提供了一条逻辑的连接，每个设备都有这样一条点对点的连接。

为了细化 USB 的通信机制，协议开发者采用了分层的概念。USB 通信逻辑上分成三层：信号层、协议层和数据传输层。

信号层用来传输位信息流的信息，在这里传输的数据称为包 (Packet)；协议层用来实现包字节流的信息，它们在信号层被编码成 NRZI 位信息传送出去，这里的包信息流称为事务处理 (Transaction)；数据传输层用来实现功能接口间传输有一定意义的信息，这些信息在协议层被打包为包格式，这里的信息流称为传输 (Transfer)。下面分析各个层次进行数据传输的过程。

首先是传输的基本单位——包。包由 5 个部分组成，其中包括设备地址、端点号与数据等信息，以及其他的头、尾和 CRC 校验等信息。通过这些信息我们可以在端点与端点间传输指定的包。

在此之上一个层次传输的就是事务。以输入（IN）事务为例，在数据输入时，USB 主机首先发送令牌包，通知设备准备向 USB 主机发送数据。当指定设备接收到令牌包并验证正确后，将需要发送的数据组装为数据包向 USB 主机发送。最后当 USB 主机接收到数据并校验无误后，创建一个 ACK 握手包返回设备。这样一个正确的数据输入（IN）事务就完成了。

而在数据传输层上传输的则是传输（Transfer）。传输包括四种，我们这里以批量传输为例。当批量传输开始的时候，USB 主机向总线广播发出一个输入令牌包，当设备收到令牌包后，根据令牌包中的信息将指定的数据返回；如果一个数据包不能容纳下全部数据，就会发送多次数据包。而主机在收到数据的时候返回 ACK 来确认接收到数据。

1.4 USB Host

USB Host（USB 主机）是 USB 总线的核心部分，它负责整个 USB 总线的所有信息；USB 设备是 USB 总线上的功能模块节点，它总是与特定应用关联在一起。USB 主机包括 USB 软件驱动和 USB 主机硬件功能接口。USB 主机分为主机控制器、USB 协议软件和设备的客户端驱动程序三个部分。下面简要介绍三个部分的功能：

- USB 总线接口层：主机端的物理层，提供了与系统互连的总线接口，处理电气信号层及协议层的互连。
- USB 协议软件：通过主机控制器来管理和控制 USB 主机与设备之间的数据传输，相对于主机控制器而言，它主要处理客户端驱动程序看到的数据传输以及他们同设备之间的交互。USB 协议软件和主机控制器共同完成对 USB 系统中数据传输的处理。
- 设备客户端驱动程序：主要实现对特定设备功能的管理和配置等操作。

2、USB 摄像头软件实现

2.1 打开 USB 摄像头设备

```
int CCameraCode::InitCamera ()
{
    int rc = 0;
    if (hCam == INVALID_HANDLE_VALUE)
    {
        // Open Driver
        hCam = CreateFile (TEXT("CAM1:"), GENERIC_WRITE | GENERIC_READ,
            0, NULL, OPEN_EXISTING, 0, NULL);

        if (hCam == INVALID_HANDLE_VALUE)
        {
            rc = GetLastError();
        }
    }

    return rc;
}
```

2.2 关闭 USB 摄像头设备

```
int CCameraCode::ShutdownCamera ()
{
    if (fCont)
        StopStreaming ();
    if (hCam != INVALID_HANDLE_VALUE)
    {
        CloseHandle (hCam);
    }
    hCam = INVALID_HANDLE_VALUE;
    return 0;
}
```

2.3 视频捕获函数与其捕获线程

```
int CCameraCode::StartStreaming (HDC hdc, RECT *prect, WORD wFormat, WORD wFrame, DWORD
dwInterval)
{
    PTHREADSTRUCT pThd = (PTHREADSTRUCT)LocalAlloc (LPTR, sizeof (THREADSTRUCT));
    if (!pThd) return ERROR_NOT_ENOUGH_MEMORY;
    // Load parameter passing struct
    pThd->wFormat = wFormat;
    pThd->wFrame = wFrame;
    pThd->rect = *prect;
    pThd->dwInterval = dwInterval;
    pThd->hdc = hdc;
    pThd->pCamercode =this;

    if (hCam != INVALID_HANDLE_VALUE)
    {
        fCont = TRUE;
        hThread = CreateThread (NULL, 0, ReadFrameThread, (PVOID)pThd, 0, NULL);
        if (hThread)
        {
            Sleep (0); //Let the thread start...
            return 0;
        }
        else
            return GetLastError();
    }
    return 0;
}
```

```

DWORD WINAPI CCameraCode::ReadFrameThread (PVOID pArg)
{
    int rc = 0;
    BOOL f;
    DWORD dwBytes;
    THREADSTRUCT Thd;
    FORMATPROPS fmtData;
    int nFrameCnt = 0;
    DWORD dwTick = 0;
    DWORD dwErr = 0;
    if (!pArg)
        return -1;
    // Copy over params
    Thd = *(PTHREADSTRUCT)pArg;
    CCameraCode *pCameracode=(CCameraCode*)Thd.pCameracode;
    LocalFree (pArg);
    rc = pCameracode->GetFormatInformation (Thd.wFormat, Thd.wFrame, &fmtData);
    if (rc)
        return rc;
    // Initialize the conversion library
    rc =pCameracode->pMjpe2bmp-> InitDisplayFrame (NULL);
    RECT rect;
    if ((Thd.rect.right == 0) && (Thd.rect.bottom == 0))
        SetRect (&rect, Thd.rect.left, Thd.rect.top,
            Thd.rect.left + fmtData.dwWidth, Thd.rect.top + fmtData.dwHeight);
    else
        rect = Thd.rect;
    // Parameters needed to start a stream
    STARTVIDSTRUCT svStruct;
    dwBytes = 0;
    svStruct.cbSize = sizeof (STARTVIDSTRUCT);
    svStruct.wFormatIndex = Thd.wFormat;
    svStruct.wFrameIndex = Thd.wFrame;
    svStruct.dwInterval = Thd.dwInterval;
    svStruct.dwNumBufs = NUMBUFFS;
    svStruct.dwPreBuffSize = PREBUFFSIZE;
    svStruct.dwPostBuffSize = 0;
    // Start the video stream
    f = DeviceIoControl ( pCameracode->hCam, IOCTL_CAMERA_DEVICE_STARTVIDEOSTREAM,
        (LPVOID)&svStruct, sizeof (STARTVIDSTRUCT),
        0, 0, &dwBytes, NULL);
    if (f)
    {
        GETFRAMESTRUCT gfsIn;
    }
}

```

```

GETFRAMESTRUCTOUT gfsOut;
memset (&gfsIn, 0, sizeof (GETFRAMESTRUCT));
gfsIn.cbSize = sizeof (GETFRAMESTRUCT);
gfsIn.dwFlags = GETFRAMEFLAG_GET_LATESTFRAME;
gfsIn.dwFlags |= GETFRAMEFLAG_TIMEOUT_VALID;
gfsIn.dwTimeout = 10000;
memset (&gfsOut, 0, sizeof (GETFRAMESTRUCTOUT));
gfsOut.cbSize = sizeof (GETFRAMESTRUCTOUT);
// Get the next frame of video
f = DeviceIoControl ( pCameracode->hCam, IOCTL_CAMERA_DEVICE_GETNEXTVIDEOFRAME,
                    &gfsIn, sizeof (GETFRAMESTRUCT),
                    &gfsOut, sizeof (GETFRAMESTRUCTOUT), &dwBytes, NULL);
pCameracode->fCont = f;
while ( pCameracode->fCont)
{
    nFrameCnt++;
    if (pCameracode->fDraw)
    {
        rc = pCameracode->pMjpe2bmp-> DisplayFrame (gfsOut.pFrameData, PREBUFFSIZE,
gfsOut.dwFrameSize, Thd.hdc, &rect);
        if (rc) dwErr++;
    }
    dwTick = GetTickCount();
    gfsIn.dwFlags = GETFRAMEFLAG_GET_LATESTFRAME | GETFRAMEFLAG_FREEBUFF_VALID;
    gfsIn.pFrameDataRet = gfsOut.pFrameData;
    gfsIn.dwFlags |= GETFRAMEFLAG_TIMEOUT_VALID;
    gfsIn.dwTimeout = 10000;
    // Call the driver
    f = DeviceIoControl ( pCameracode->hCam, IOCTL_CAMERA_DEVICE_GETNEXTVIDEOFRAME,
                        &gfsIn, sizeof (GETFRAMESTRUCT),
                        &gfsOut, sizeof (GETFRAMESTRUCTOUT), &dwBytes, NULL);
    if (!f)
        pCameracode->fCont = FALSE;
}
// Stop the stream
f = DeviceIoControl ( pCameracode->hCam, IOCTL_CAMERA_DEVICE_STOPVIDEOSTREAM,
                    0, 0, 0, 0, &dwBytes, NULL);
}
else
    pCameracode->fCont = FALSE;
// Clean up translation code
pCameracode->pMjpe2bmp-> ReleaseDisplayFrame ();
return 0;
}

```

2.4 获取静态图像

```
int CCameraCode::GetStillImage (WORD wFormat, WORD wFrame, PFORMATPROPS pFmt, PBYTE *ppData,
DWORD *pdwSize)
{
    int rc = 0;
    VIDFORMATSTRUCT vf;
    DWORD dwBytes, dwBuff;
    BOOL f;
    memset (&vf, 0, sizeof (vf));
    vf.cbSize = sizeof (VIDFORMATSTRUCT);
    vf.wFormatIndex = wFormat;
    vf.wFrameIndex = wFrame;
    *pdwSize = 0;
    // See if we should allocate the buffer
    if (*ppData == 0)
    {
        // Call to get the size of the buffer
        f = DeviceIoControl (hCam, IOCTL_CAMERA_DEVICE_GETSTILLIMAGE,
                             &vf, sizeof (VIDFORMATSTRUCT), 0, 0, &dwBytes, NULL);

        if (!f)
        {
            rc = GetLastError();
            return rc;
        }

        // Allocate the buffer
        *ppData = (PBYTE) LocalAlloc (LPTR, dwBytes);
        if (*ppData == 0)
            return ERROR_NOT_ENOUGH_MEMORY;
        dwBuff = dwBytes;
    }
    // Call to get the image
    f = DeviceIoControl (hCam, IOCTL_CAMERA_DEVICE_GETSTILLIMAGE,
                         &vf, sizeof (VIDFORMATSTRUCT), *ppData, dwBuff, &dwBytes, NULL);

    if (!f)
        rc = GetLastError();
    else
        *pdwSize = dwBytes;

    return rc;
}
```

[实验内容]

- 1、了解 USB 总线基本原理；
- 2、掌握 USB 摄像头编程控制方法；
- 3、了解流式驱动程序架构和 USB 摄像头驱动的编译

[实验步骤]

USB 摄像头驱动编译：

第一步：连接好实验系统，打开实验箱电源；

第二步：利用 EVC4.0 打开 USB 摄像头驱动工程 WebCam.vcw，按 F7 编译 USB 摄像头驱动程序。如图 15-1 所示。编译成功后 EVC 自动将 USB 摄像头驱动程序 WebCam.dll 序下载到 XSBase270

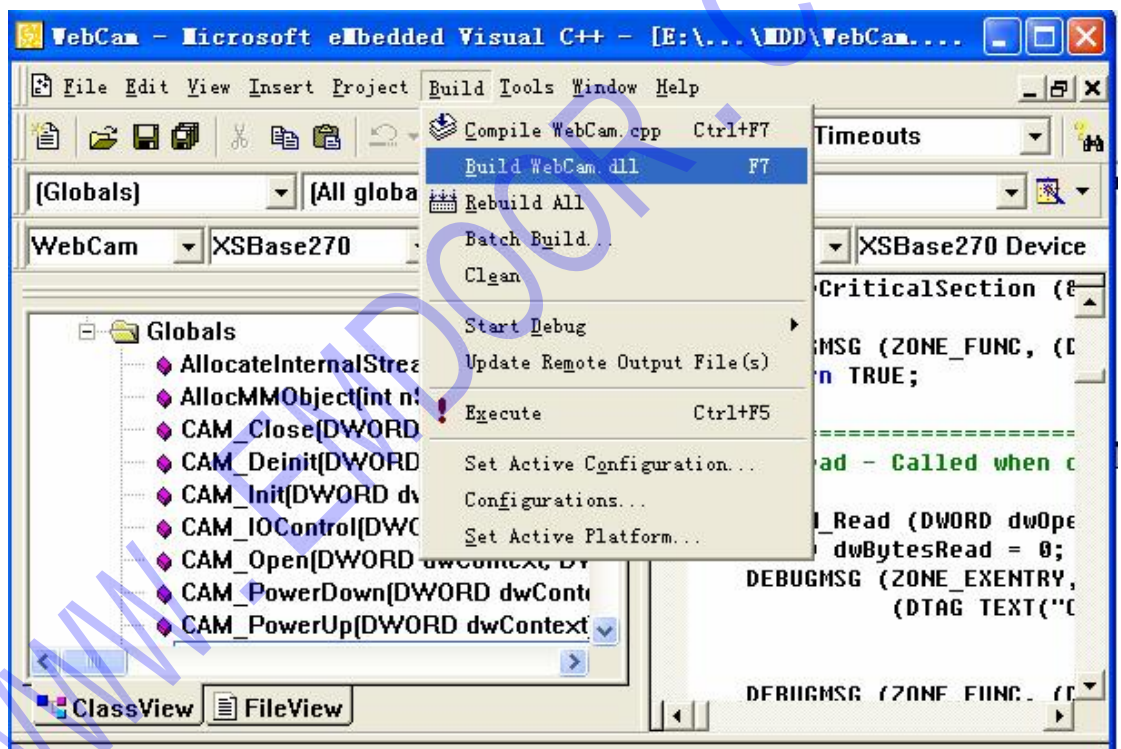


图 15-1 USB 摄像头编译示意图

第三步：将罗技 Pro5000 摄像头插入目标板，系统弹出要求输入驱动程序对话框，如图 15-2 所示，输入 USB 摄像头的驱动程序 webcam.dll。



图 15-2 输入 USB 摄像头驱动程序对话框

USB 摄像头应用程序:

第一步: 利用 Visual Studio 2005.net 打开 USB 摄像头应用测试工程文件 USBCamera.sln, 进行编译: 点击运行按钮, 将 USB 摄像头应用测试程序下载到 XSBase270 目标板运行, 运行界面如图 15-3 所示;



图 15-3 USB 摄像头测试程序运行界面

第二步: 按“打开摄像头”按钮, 测试程序自动进行视频捕获。同时自动获取摄像头所支持的图像尺寸、帧率。按“静态图像”按钮, 测试程序获取一幅图像并将图像进行保存, 所保存的图像名在文本框中输入。

[习题与思考题]

- 1、分析 USB 摄像头源程序，添加设置 USB 摄像头的属性的功能。
- 2、分析 USB 摄像头源程序，添加获取 USB 摄像的各项设置的功能
- 3、如果将获取的静态图像显示在窗体上，该怎样实现？

WWW.EMDOOR.COM

实验十六 应用程序集成实验

[实验目的]

- 1、掌握如何将开发出来的应用程序集成到系统中
- 2、掌握如何修改平台下的注册表文件
- 3、理解映像配置文件参数的作用

[实验仪器]

- 1、装有 Platform Builder、EVC 和 VS.Net 开发平台的 PC 机一台
- 2、XSBase270 实验开发平台一套

[实验原理]

WinCE 利用 MAKEIMG 应用配置文件来创建操作系统运行时映像，常用的配置文件包括二进制映像构建器文件 (.BIB)、注册表文件 (.REG)、文件系统文件 (.DAT) 和数据库文件 (.DB)。下面具体介绍二进制映像构建器文件 (.BIB) 和注册表文件 (.REG) 的主要组成部分的各参数的作用。

1、MODULES 区域

在 MODULES 区域，可执行模块列出了可就地执行 (XIP) 的模块，其语法为：

Name	Path	Memory	Type
explorer.exe	\$(_FLATRELEASEDIR)\explorer.exe	NK	S

该例子告诉 Romimage.exe 包含 explore.exe (浏览器) 模块，并将它加载到标记为 NK 的存储器器区域，且 explore.exe 模块具有系统属性。其中 NK 存储器区域在 Config.bib 文件的 MEMORY 区域定义，位于 MODULES 区域的模块文件可以具有下列类型的任意组合属性。

- S: 系统文件
- H: 隐藏文件
- R: 压缩资源
- C: 压缩全部
- D: 运行是不允许调试
- N: 将模块标记为不可信任
- P: 在每一模块基础上忽略 CPU 类型
- K: 通知 Romeimage 必须修正 DLL 以便正确运行

2、FILES 区域

FILES 区域定义了包含在映像中其它文件，如字符文件（.TTF）、文本文件(.TXT)、位图文件（.BMP）等，FILES 区域也可定义不是就地执行的可执行文件。FILES 区域的语法与 MODULES 区域完全相同。

Name	Path	Memory	Type
windowsce.bmp	\$(_FLATRELEASEDIR)\windowsce_vgal.bmp	NK	S

这个例子表示，windowsce.bmp 文件是来自开发工作% FLATRELEASEDIR%目录下的文件，被包含在 NK 存储区域，并具有系统属性。FILES 区域可用的属性类型可以如下：

- S：系统文件
- H：隐藏文件
- U：未压缩资源
- D：运行是不允许调试
- N：将模块标记为不可信任

3、MEMORY 区域

MEMORY 区域将物理存储器划分如下：

- 数据存储区，ROM 或 RAM 存储区域
- 程序存储器，为内存应用保留的 RAM 区域。

下面为 Config.bib 的 MEMORY 区域

```
MEMORY
IF IMGFLASH!
; Name      Start      Size      Type
; -----
RSVD      80000000  000FF000  RESERVED
ARGS      800FF000  00001000  RESERVED
NK        80100000  03000000  RAMIMAGE
RAM       83100000  00F00000  RAM
ENDIF
```

4、REG 文件

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\LED]
"Dll"="led.dll"
"Prefix"="LED"
"Index"=dword:1
"Order"=dword:2
```

[实验内容]

- 1、添加应用程序实现应用程序集成；
- 2、修改注册表文件添加驱动程序；

[实验步骤]

1、添加应用程序到平台

第一步：先把要添加的文件 ThreadTest.exe 拷贝到平台目录（假设工作平台为 xsbase270），D:\WINCE500\PBWorkspaces\xsbase270\RelDir\XSBase270_ARMV4I_Release

第二步：在 Build OS 菜单中，单击 Open Release Directory，（如图 16-1）打开平台命令行，在命令行中输入：notepad shell.bib，打开 shell.bib。（可采用文本编辑器进行编辑）。在 shell.bib 中的最后一行添加 ThreadTest.exe \$(_FLATRELEASEDIR)\ThreadTest.exe NK S，保存退出（如图 16-2）。



图 16-1 打开 release 目录的示意图

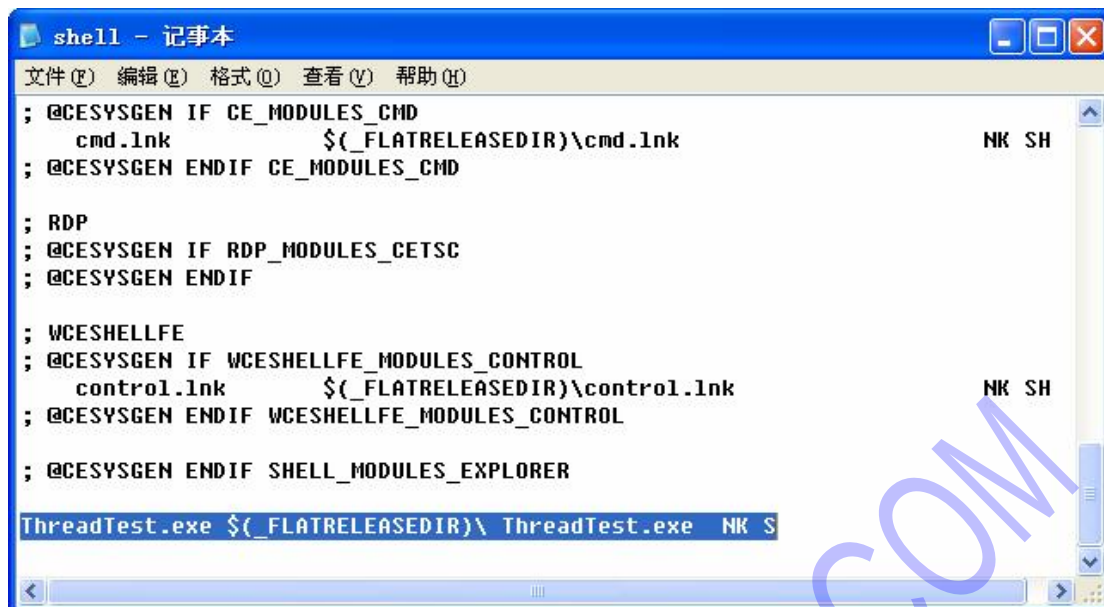


图 16-2 shell 的编辑示意图

2、修改注册表添加驱动

第一步：在 Build OS 菜单中，单击 Open Release Directory，（如图 16-1）打开平台命令行，在命令行中输入：notepad shell.bib，打开 shell.bib。（可采用文本编辑器进行编辑）。在 shell.bib 中的最后一行添加 driver.dll \$(_FLATRELEASEDIR)\ driver.dll NK S，保存退出。

第二步：在命令行中输入：notepad platform.reg，打开 platform.reg。（可采用文本编辑器进行编辑）。在 platform.reg 中的最后一行添加如下内容，保存退出。

```

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\LedDriver]
    "Dll" = "driver.Dll"
    "Prefix" = "DEM"
    "Index" = dword:1
    "Order" = dword:0
    "FriendlyName" = "LED Driver"
    
```

第三步：在 Build OS 菜单中，单击 Make Run-Time Image，（如图 16-3）重新制作平台映像，并把映像下载到目标板中



图 16-3 制作映像文件的示意图

[习题与思考题]

1. 如何修改系统的默认 IP 地址和默认网关？
2. 分析 Config.bib 文件中的 CONFIG 区域中各参数的作用