載具嵌入式系統運用教材

目錄

第 1 章 Eboot的解析

第 2 章 Windows CE 6.0電源管理

第 3 章 PXA310的Interrupt概念及應用

第 4 章 PXA310 的 GPIO 概念及應用

第 5 章 實驗平台驅動程式開發

第 6 章 實驗平臺應用程式開發

第 1章 Eboot的解析

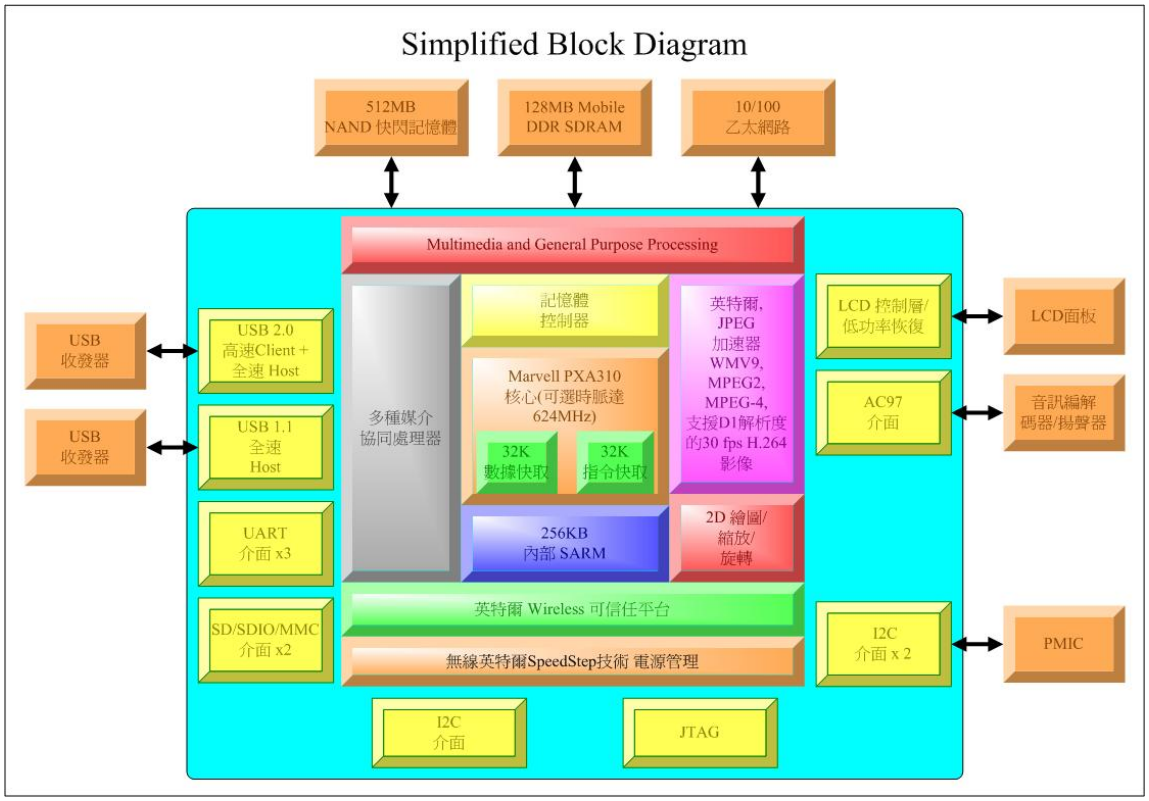
WinFast 310 開發平台其核心晶片技術源於 Intel，後來 Marvell 公司收購了 Intel 的嵌入式產品線，推出了 PXA3xx 系列產品，其中包括 PXA300，PXA310，PXA320 和 Tavor，這些產品核心架構相同，代碼相容，只是定位稍有差別。此系列產品於推 出時即已搭配Windows Embedded CE 6.0 主機板支援套件（BSP）、技術支援及相關文件。

PXA 310 採用經過優化的Xscale 架構，運行頻率624MHz，電源管理則是透過I 2C 匯流排來控制，包括各路電源的開啟，電壓的設定等，具備頻率調整的功能，在待命（Standby）模式時，會降低電源消耗，提升整體節能效率。

下圖為目前Marvell 針對不同應用的處理器的比較：

Function	PXA270	PXA320	PXA300	PXA310
Core Frequency	Up to 624 MHz	Up to 806 MHz	Up to 624 MHz	Up to 624 MHz
System Bus	Single shared internal system bus	Low-latency, fully-connected, memory switch	Low-latency, fully-connected, memory switch	Low-latency, fully-connected, memory switch
L2 Cache Internal SRAM	No L2 cache 256KB SRAM	256KB L2 cache 768KB SRAM	No L2 cache 256KB SRAM	No L2 cache 256KB SRAM
External memory interface	Shared external mem i/f for 104 MHz SDRAM x32 or x16, and NOR 52 MHz	Separate ext mem i/f for 260 MHz DDR x32 and NAND 13 MHz, NOR 52 MHz	Separate ext mem i/f for 260 MHz DDR x16 and NAND 33 MHz, NOR 52 MHz	Separate ext mem i/f for 260 MHz DDR x16 and NAND 33 MHz, NOR 52 MHz
Security	Wireless Trusted Module	None	None	Wireless Trusted Module 2
Power Mgmt	Wireless Intel SpeedStep® Technology	Enhanced Wireless Intel SpeedStep® Technology	Enhanced Wireless Intel SpeedStep® Technology with standby states	Enhanced Wireless Intel SpeedStep® Technology with standby states
Multimedia acceleration	Intel® Wireless MMX™ technology	Intel® Wireless MMX™ 2 technology. 2D graphics, scaling, rotation	Intel® Wireless MMX™ 2 technology w/ 2D	Intel® Wireless MMX™ 2 technology w/ 2D. Video/JPEG h/w acceleration. Chroma-key.
Movie Playback (video + AAC decode)	MPEG-4 up to 30 fps, VGA H.264 up to 15 fps, VGA	MPEG-4 up to 30 fps, VGA H.264 up to 15+ fps, VGA	MPEG-4 up to 30 fps, VGA H.264 up to 15 fps, VGA	MPEG-4 up to 30 fps, VGA H.264 up to 30 fps, VGA
Camcorder (video + AMR encode)	MPEG-4 up to 30 fps, QVGA H.264 up to 15 fps, QVGA	MPEG-4 up to 30 fps, CIF H.264 up to 15 fps, CIF	MPEG-4 up to 30 fps, CIF H.264 up to 15 fps, CIF	MPEG-4 up to 30 fps, VGA H.264 up to 15 fps, VGA
Video Telephony (video+AMR enc-dec)	MPEG-4 up to 30 fps, QVGA H.264 up to 15 fps, QVGA	MPEG-4 up to 30 fps, QVGA H.264 up to 15+ fps, QVGA	MPEG-4 up to 30 fps, QVGA H.264 up to 15+ fps, QVGA	MPEG-4 up to 30 fps, CIF H.264 up to 15+ fps, CIF
Camera interface sensor support	Intel® Quick Capture Technology, 1.3 Mp	Enhanced Intel® Quick Capture Technology, 2Mp	Enhanced Intel® Quick Capture Technology, 2Mp	Enhanced Intel® Quick Capture Technology + faster clocks, color conversion, JPEG sensors; 5Mp
USB	1 FS client, 1 FS OTG + 3 FS host ports	HS client UTMI and FS OTG + 2 FS ports	HS client UTMI and FS OTG + 2 FS ports	HS Client ULPI and FS OTG + 1 FS port
UARTs SSPs	Three, 921 kbps Three, 13 MHz	Three, 921 kbps Four, 13 MHz	Three, 3.6 Mbps Four, 13 MHz	Three, 3.6 Mbps Four, 26 MHz
MMC/SD/SDIO	One, 19.5 MHz	Two, 19.5 MHz	Two, 26 MHz	Three, 26 MHz
Package Size	13x13mm discrete 14x14mm stacked	14x14mm discrete	13x13mm discrete 15x15mm stacked, PoP	13x13mm discrete 15x15mm stacked, PoP

下圖為WinFast 310 開發平台的系統方塊圖：



WinFast 310 開發平台硬體資源：

中央處理器

CPU：Marvell PXA 310 Core Scalable to 624 MHz。

外部記憶體

- DDR SDRAM：128M 位元組。
- NAND Flash：512M 位元組。

串列埠

三個全雙工非同步串列介面，串列傳輸速率高達38400bps。

網路介面

一個10/100M 網路介面，採用LAN91C113。

USB 介面

一個USB1.1 HOST 介面。

一個USB OTG 介面。

音效介面

採用介面晶片WM9713，一個立體聲音效輸出介面可接耳機或喇叭。

支援錄音，一個麥克風輸入介面可接麥克風。

儲存介面

兩個SD 卡介面，支援4G SDHC 卡。

LCD 和觸控螢幕介面

開發平台內建4 線式電阻觸控螢幕介面的相關電路。

支援65536 色TFT 液晶螢幕，尺寸10.2 寸，螢幕解析度可達到800x480 畫素。

標準配置為INNOLUX 65536 色800x480/10.2 英寸TFT 液晶螢幕，內建觸控螢幕。

時鐘

內建即時時鐘。

重置電路

一個重置按鍵，並採用專用重置晶片進行重置，穩定可靠。

除錯及下載介面一個20 PIN Multi－ICE 標準JTAG 介面。

電源介面

5V 電源供電，帶電源開關和指示燈。

作業系統

- Windows Embedded CE 6.0。

1.1 什麼是BootLoader

BootLoader 就是在作業系統內核執行之前執行的一段小程式。透過這段小程式，我們可以 初始化硬體設備、建立內存空間的映射圖，從而將系統的軟硬體環境設定成一個合適的狀 態，以便為最終調用作業系統內核準備好正確的環境。BootLoader 是嚴重地倚賴於硬體而 實現的，特別是在嵌入式系統。因此，在嵌入式系統裡建立一個通用的BootLoader 幾乎 是不可能的。儘管如此，我們仍然可以對 BootLoader 歸納出一些通用的概念來，以指導 用戶進行特定的BootLoader 設計與實現。

嵌入式系統中，BootLoader 的意義與作用與 PC 上的 BIOS 有點類似，它對開發板上的主 要部件如 CPU 、SDRAM 、FLASH 、串口等進行了初始化，可以使用 Bootloader 下載 檔案到開發板，可以瀏覽目錄，可以燒錄 FLASH ，可以啟動系統等，實際上，一個功能 比較強大的BootLoader 已經相當於一個微型的作業系統了。

初始化基礎硬體──CPU 速度、存儲器定時、中斷、檢測 RAM 大小。引導裝載程式通常 是在任何硬體上執行的第一段代碼。台式機這樣的常規系統中，通常將引導裝載程式裝入 主引導記錄（Master Boot Record，MBR）中，在嵌入式設備中，通常將引導程式放置在不 易丟失的存儲器的開始位址或者是系統冷啟動時 PC 暫存器的初始值。通常，在台式機或 其他系統上，BIOS 將控制移交給引導裝載程式。而在嵌入式系統中，通常並沒有像 BIOS 那樣的固件程式（注，有的嵌入式 CPU 也會內嵌一段短小的啟動程式）因此整個系統的 加載啟動任務就完全由 BootLoader 來完成。引導程式完成自己的任務後，也將控制權移 交給作業系統。

總體上BootLoader 需要完成以下工作。

初始化CPU 速度；
初始化內存，包括啟用內存庫，初始化內存配置暫存器等；
初始化中斷控制器，在系統啟動時，關閉中斷，關閉看門狗；
初始化串行端口（如果在目標上有的話）；
啟用指令/數據高速緩存；
設定堆棧指標；
設定參數區域並構造參數架構和標記（這是重要的一步，因為內核在標識根設備、頁面大小、內存大小以及更多內容時要使用引導參數）
執行POST （加電自檢）來標識存在的設備並報告有何問題；
為電源管理提供掛起/恢複支援；
傳輸操作系統內核鏡像文件到目標機。也可以將操作系統內核鏡像文件事先存放在FLASH 中，這樣就不需要BootLoader 和主機傳輸操作系統內核鏡像文件，這通常是在做成產品的情況下使用。而一般在開發過程中，為了調試內核的方便，不將操作系統內核鏡像文件固化在FLASH 中，這就需要主機和目標機進行文件傳輸；
跳轉到內核的開始，在此又分為ROM 啟動和RAM 啟動。所謂ROM 啟動就是用XIP 技術直接在FLASH 中執行作業系統鏡像檔案;所謂RAM 啟動就是指把內核鏡像從FLASH 複製到RAM 中,然後再將PC 指標跳轉到RAM 中的作業系統啟動位址。

嵌入式系統的資源有限，程式通常都是固化在ROM 中執行。ROM 中程式執行前，需要對系統硬體和軟體執行環境進行初始化，這些工作是用組合語言編寫的啟動程式完成。啟 動程式是嵌入式程式的開頭部分，應與應用程式一起固化在 ROM 中，並首先在系統上執 行，它應包含各模塊中可能出現的所有段類，並合理安排它們的次序。

1.2 BootLoader 和主機之間檔案傳輸的通信協議

最常見的情況就是,目標機上的BootLoader 透過串口與主機之間進行檔案傳輸,傳輸協議 通常是xmodem/ymodem/zmodem 協議中的一種。但是，串口傳輸的速度是有限的，因此 透過乙太網連接並借助TFTP 協議來下載檔案是個更好的選擇。

此外,在論及這個話題時,主機方所用的軟體也要考慮。比如,在透過乙太網連接和TFTP 協議來下載檔案時，主機方必須有一個軟體用來的提供TFTP 服務。

.3 BootLoader 選項

程式員可以在自己的BootLoader 中實現不同的啟動選項，在此將對一些常用的選項進行討論。如：BootLoader 通信模式，串口中的啟動功能等。

圖1-1 是最常見的嵌入式系統目標機和主機通信的模型。開發人員可以使用超級終端透過串口向目標機發送命令，由於串口協議是最簡單、可靠的，因此經常被使用在系統未啟動階段。

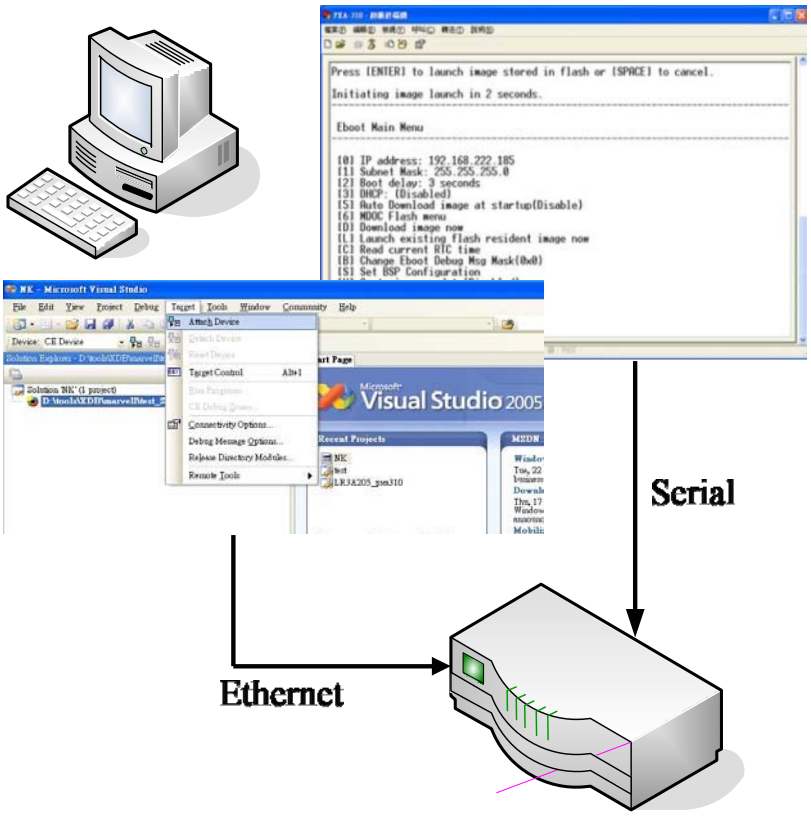


圖1-1　嵌入式系統中目標機和主機通信模型

串口命令行選項

超級終端上使用串口協議，用文本形式來顯示和目標機的通信過程。用戶可以在超級終端的命令行中指定目標機執行某個命令。這些命令包括。

從Ethernet　下載作業系統鏡像檔案；
從FLASH　啟動（這要求作業系統鏡像檔案已經被固化在FLASH　中）；
啟動RAM　測試程式；
對目標機的I/O　端口進行測試；

開發人員可以自己對這些命令進行修改或者擴充。下面就是透過串口通信來控制目標機的選單樣例：

```
Ethernet Boot Loader Configuration:
[0] IP address: 192.168.80.1
[1] Subnet Mask: 255.255.255.0
[2] Boot delay: 3 seconds
[3] DHCP: (Disabled)
[5] Auto Download image at start-up (Disable)
[D] Download image now
[L] Launch existing flash resident image now
[C] Read current RTC time
[B] Change Eboot Debug Msg Mask(0x0)
[S] Set BSP Configuration
[U] Go to image update (Disabled)
[F] Format User Partition (Disabled)
[T] Test Menu
```

1.4 實現一個**BootLoader**

1.4.1 **BootLoader** 的構成組件

BootLoader 主要由以下兩部分組成。

(A) OEM startup code

這部分代碼是在 BootLoader 中最先被執行的。它的主要功能是初始化最小範圍的硬體設備，比如設定CPU 工作頻率、關閉看門狗、設定cache 、設定RAM 的刷新率、填寫內存控制暫存器（通知CPU 有效的數據匯流排引腳數）等。由於系統剛剛啟動，不適合使用複雜的高階語言，因此這部分代碼主要由組合程式完成。在組合程式段設定完堆棧後，就跳轉到C 語言的Main 函數入口（位於 \WINCE600\PLATFORM\PLATFORM_BSP\SRC\BOOTLOADER\EBOOT\main.c）

(B) Main code

這部分代碼由C 語言實現，是BLCOMMON 代碼的一部分，它可以用來執行比較複雜的操作。比如檢測內存和FLASH 的有效性、檢測外部設備界面、檢測串口並且向已經連接的主機發送調試資訊、透過串口等待命令、啟動網路界面、建立內存映射等彙編無法完成的工作。

Main code　包含以下代碼段：

```
void main(void)
{
// Common boot loader (blcommon) main routine.
//
BootloaderMain();
// Should never get here.
//
SpinForever();
}
```

Image　的下載可以透過以下界面。

(1) Ethernet port I/O　界面。這是透過主機和目標機的網路界面之間的數據傳輸來完成下載工作。具體實現代碼,讀者可以參考\WINCE600 \PUBLIC\ COMMON \OAK\DRIVERS\ETHDBG\BLCOMMON檔案夾；

(2) Debug serial I/O　接口。這是通過主機和目標機的串口之間的數據傳輸來完成下載工作。利用串口來傳輸的缺點非常明顯，那就是速度太慢；

通過事先FLASH write代碼將鏡像文件固化入FLASH。只要BootLoader被設計成能從FLASH加載鏡像文件，本選項就可以使用。

1.4.2　BootLoader　控制流函數Control Flow Functions

圖1-2 為Windows CE.NET 的BootLoade的整體架構,其中列出了各種控制流函數。

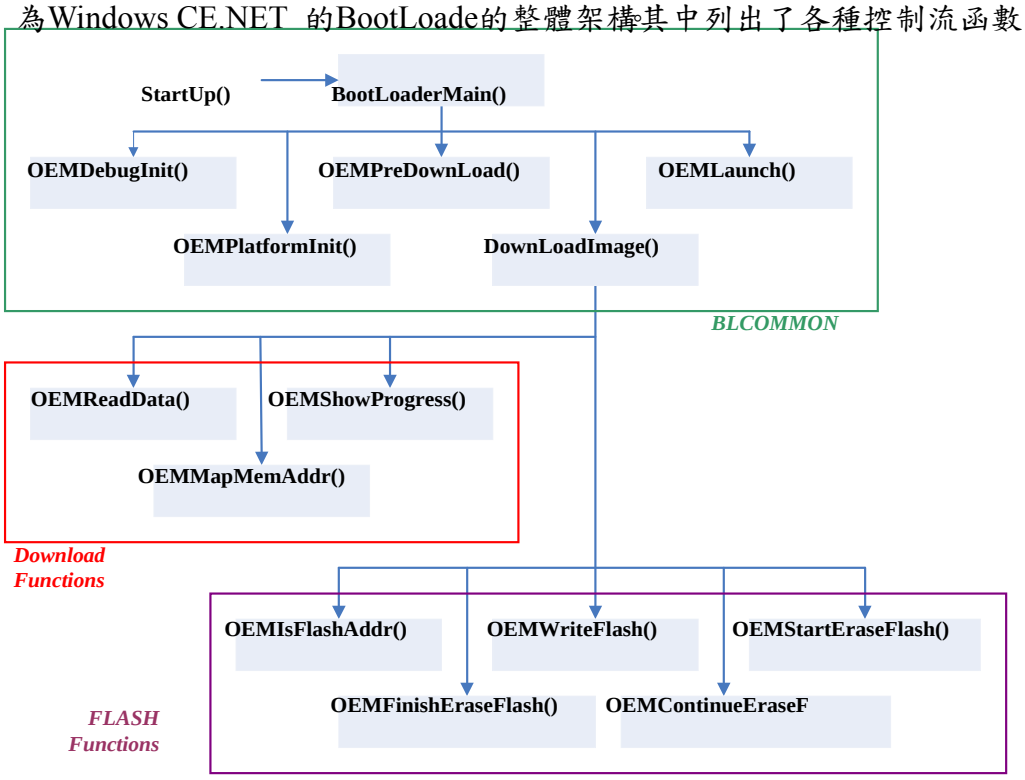


圖1-2 BootLoader 的整體架構

下面，我們來看以下Windows CE.NET 的BootLoader 在系統啟動時所執行的函數。
圖1-2中的所有函數分為3個模塊：BLCOMMON、Download Function、FLASH Function。其中BLCOMMON 模塊是由微軟公司提供的，執行一些邏輯上的功能，因此建議開發人員不要對其進行修改。而Download Function、FLASH Function中的函數與硬件平台息息相關，因此對於每種硬件平台都要將函數的實現進行修改。

BLCOMMON 模塊的功能解釋。

```
void BootloaderMain (void)
{
DWORD dwAction;
DWORD dwpToc = 0;
DWORD dwImageStart = 0, dwImageLength = 0, dwLaunchAddr = 0;
BOOL bDownloaded = FALSE;

// relocate globals to RAM
if (!KernelRelocate (pTOC))
{
// spin forever
HALT (BLERR_KERNELRELOCATE);
}

// (1) Init debug support. We can use OEMWriteDebugString afterward.
if (!OEMDebugInit ())
{
// spin forever
HALT (BLERR_DBGINIT);
}

// output banner
KITLOutputDebugString (NKSignon, CURRENT_VERSION_MAJOR ,
CURRENT_VERSION_MINOR);

// (3) initialize platform (clock, drivers, transports, etc)
if (!OEMPlatformInit ())
{
// spin forever
HALT (BLERR_PLATINIT);
}

// system ready, preparing for download
KITLOutputDebugString ("System ready!\r\nPreparing for download...\r\n");

// (4) call OEM specific pre-download function
switch (dwAction = OEMPreDownload ())
{
case BL_DOWNLOAD:
// (5) download image
if (!DownloadImage (&dwImageStart, &dwImageLength, &dwLaunchAddr))
{
// error already reported in DownloadImage
SPIN_FOREVER;
}
bDownloaded = TRUE;
```

```
// Check for pTOC signature ("CECE") here, after image in place
if (*(LPDWORD) OEMMapMemAddr (dwImageStart, dwImageStart +
ROM_SIGNATURE_OFFSET) == ROM_SIGNATURE)
{
dwpToc = *(LPDWORD) OEMMapMemAddr (dwImageStart, dwImageStart +
ROM_SIGNATURE_OFFSET + sizeof(ULONG));
// need to map the content again since the pointer is going to be in a fixup address
dwpToc = (DWORD) OEMMapMemAddr (dwImageStart, dwpToc +
g_dwROMOffset);

KITLOutputDebugString ("ROMHDR at Address %Xh\r\n", dwImageStart +
ROM_SIGNATURE_OFFSET + sizeof (DWORD)); // right after signature
}

// fall through
case BL_JUMP:
// Before jumping to the image, optionally check the image signature.
// NOTE: if we haven't downloaded the image by now, we assume that it'll be loaded
from local storage in OEMLaunch (or it
// already resides in RAM from an earlier download), and in this case, the image start
address might be 0.       This means
// that the image signature routine will need to find the image in storage or in RAM to
validate it.    Since the OEM"s
// OEMLaunch function will need to do this anyways, we trust that it's within their
abilities to do it here.
//
if (g_bBINDownload && g_pOEMCheckSignature)
{
if (!g_pOEMCheckSignature(dwImageStart, g_dwROMOffset, dwLaunchAddr, bDownloaded))
HALT(BLERR_CAT_SIGNATURE);
}
// (5) final call to launch the image. never returned
OEMLaunch (dwImageStart, dwImageLength, dwLaunchAddr, (const ROMHDR *)dwpToc);
// should never return
// fall through
default:
// ERROR! spin forever
HALT (BLERR_INVALIDCMD);
}
}
```

BLCOMMON 是一個庫，其實現代碼位於\WINCE600 \PUBLIC\ COMMON \OAK\DRIVERS\ETHDBG\BLCOMMON目錄下。
它實現了Windows CE BootLoader的基本框架。這個庫的工作為：將BootLoader 加載到RAM 中執行、解壓縮.bin 檔案、校驗硬體平台的完整性、對加 載 的 進
度進行跟蹤。在BLCOMMON 階段執行的過程中，主要使用OEM 函數集。

BLCOMMON 庫的入口點為BootLoaderMain 函數，它有StartUp 彙編函數完成 後跳轉至該入口。BLCOMMON 庫將被BootLoader 的程式鏈接在一起。
在系統啟動時，CPU 首先執行StartUp 函數，這是個由彙編實現的函數。StartUp 函數主要的功能為：設定CPU 工作頻率、關閉看門狗、設定cache 、設定RAM 的 刷新率、填寫內存控制暫存器通知CPU 有效的數據匯流排引腳數籌。在StartUp 完成任務後，就跳轉到BootLoaderMain 函數中。這個是由C 語言編程實現的函 數入口點。

下面是PXA310 中的BootLoader 中的main 函數實現代碼：

（1）BLCOMMON 模塊函數下面列舉出BLCOMMON 中的控制函數並分析它們，這些函數在 Blcommon.h 中聲明，代碼實現下Blcommon.lib 裡：
OEMDebugInit() / OEMPlatformInit() / OEMPreDownload() / OEMLaunch()

OEMDebugInit() 函數：

在執行BootLoaderMain 程式後，將首先調用OEMDebugInit 函數，它用來初始化調試資訊的 I/O 設備，最常見的是串口設備。由於RS232 協議簡性， 在系統沒有啟動前對串口初始化較適用。在OEMDebugInit 裡，又通常調用OEMInitDebugSerial 函數來初始化串口。

下面是此函數代碼實例：

```
//------------------------------------------------------------------------------
//
//   Function:      OEMDebugInit
//
//   Initialize debug serial UART.
//
BOOL OEMDebugInit(void)
{
// Initialize the debug UART.
// setup MFP for mDOC-H3 device
GPIOSetup();
PlatformPowerUpDbgSerial();
InitDebugSerial((UINT32)DEBUG_SERIAL_ADDR, FALSE);
//todo: check      whether to read from BSPARG or directly set this. - WWB
// Clear the hex LEDs.
//
OEMWriteDebugLED(0, 0);
return(TRUE);
}
```

OEMPlatformInit() 函數：

OEM 層的初始化函數，它主要負責目標機上的硬體初始化。在彙編階段只是初始化了很小 一部分硬體，這是由於BootLoader 要求處理時間短，因此 在彙編階段的硬體初始化是十分簡單的。所以有必要用高階語言完成對目標 機的硬體設定，這包括具體的時鐘設定、驅動和傳輸設備界面的初始化。

下面是此函數代碼實例：

```
//------------------------------------------------------------------------------
//
//   Function:      OEMPlatformInit
//
//   Initialize the platform and show the Menu.
//   It's referenced by blcommon.c, invoked after OEMDebugInit() done.
//
```

```
 BOOL OEMPlatformInit(void)
 {
UINT32 AutoBootDelay = 0;
UINT32 StartTime, CurrTime, PrevTime;
UINT32 Selection;
static BSP_ARGS oldBSPArgs;
KITLOutputDebugString("Microsoft Windows CE Ethernet Bootloader %d.%d for the Marvell %s
 Development Platform Built %s\r\n",
EBOOT_VERSION_MAJOR, EBOOT_VERSION_MINOR, BSP_DEVICE_PREFIX,  ____DATE__);

memset((LPVOID) g_pBSPArgs, 0, sizeof(BSP_ARGS));

//FMD_Init((LPCTSTR)EBOOT_CFG_PARTITION_INDEX, NULL, NULL);
//FMD_Init(L"Eboot BP_Init()", NULL, NULL);

 // BP_Init() will invoke FMD_Init().
if (!BP_Init((LPBYTE)BPbuf, BPBUF_SIZE, (LPCTSTR)EBOOT_CFG_PARTITION_INDEX,
 NULL, NULL))
 {
 // Load default bootloader configuration settings.
 //
 KITLOutputDebugString("FATAL ERROR: Flash or BootPartition initialization failed - loading
 bootloader defaults...\r\n");
 ResetDefaultBSPCFG(g_pBSPArgs);
 }
 else
{
      // Get flash info
      if (FMD_GetInfo(&g_FlashInfo) == FALSE)
      {
      KITLOutputDebugString("FATAL ERROR: OEMDebugInit: FMD_GetInfo call failed!\r\n");
            ResetDefaultBSPCFG(g_pBSPArgs);
      }
      else
      {
      // Load the bootloader configuration from flash (menu settings).
     // LoadEBSPCFG(g_pBSPArgs);
      }
}

if (g_pBSPArgs->header.signature != OAL_ARGS_SIGNATURE ||
      g_pBSPArgs->header.oalVersion != OAL_ARGS_VERSION ||
      g_pBSPArgs->header.bspVersion != BSP_ARGS_VERSION ||
      g_pBSPArgs->argsLength != sizeof(BSP_ARGS) )
{
      ResetDefaultBSPCFG(g_pBSPArgs);
}

if (g_pBSPArgs->header.signature != OAL_ARGS_SIGNATURE ||
      g_pBSPArgs->header.oalVersion != OAL_ARGS_VERSION ||
      g_pBSPArgs->header.bspVersion != BSP_ARGS_VERSION ||
      g_pBSPArgs->argsLength != sizeof(BSP_ARGS) )
{
      ResetDefaultBSPCFG(g_pBSPArgs);
}

g_pBSPArgs->bFormatPartition = FALSE;

// back up the initial config, to be compared later.
```

```
 memcpy(&oldBSPArgs, g_pBSPArgs, sizeof(oldBSPArgs));

PlatformInitPMIC(); // put Voltage change here, before frequency change.
//if (g_pBSPArgs->RunFreq >= BSP_ARGS_RUN_FREQ_2)
DoChangeFreq(BSP_ARGS_RUN_FREQ_5);//default to 624mhz
DoChangeFreq(g_pBSPArgs->RunFreq);//and then switch current setting, this might do nothing

if (g_pBSPArgs->BatteryEnable)
{
      // The battery charger must be enabled as early
      // as possible to insure the battery doesn't run out
      // of juice before we boot.
      // Must pay attention to the productes without Eboot,
      // then this need to be put in some other place.
      //
      // Detect a battery charging source if available
      //
      EnableBatteryCharger(
            OALPAtoVA(MONAHANS_BASE_REG_PA_OST, FALSE),
            OALPAtoVA(MONAHANS_BASE_REG_PA_I2C, FALSE),
            OALPAtoVA(MONAHANS_BASE_REG_PA_MFP, FALSE)
            );
}
else
{
  BootDEBUGMSG(DBG_INFO, (L"Battery charger disabled in EBOOT config. Skipped...\r\n"));
}

// Detect LCD panel
g_pBSPArgs->DefaultLCDPanel = DetectLCDPanel(); //todo: remove this? it will run in the
normal execution path - WWB

ConfigPrint();

// User menu code...
//
AutoBootDelay = g_pBSPArgs->eBootCFG.delay;
if (g_pBSPArgs->eBootCFG.autoDownloadImage)
{
    g_DownloadImage = TRUE;
    KITLOutputDebugString ("\r\nPress [ENTER] to download now or [SPACE] to cancel.\r\n");
    KITLOutputDebugString ("\r\nInitiating image download in %d seconds. ", AutoBootDelay--);
}
else
{
      g_DownloadImage = FALSE;
      KITLOutputDebugString ( "\r\nPress [ENTER] to launch image stored in flash or [SPACE]
      to cancel.\r\n");
      KITLOutputDebugString ( "\r\nInitiating image launch in %d seconds. ", AutoBootDelay--);
}

// Get a snapshot of the RTC count (in seconds).
//
StartTime     = OEMEthGetSecs();
PrevTime      = StartTime;
CurrTime      = StartTime;
Selection     = 0;

// Give the user some time to interrupt the auto boot/download process.
// Count down to 0 before proceeding with default operation.
```

```
while ((CurrTime - StartTime) < g_pBSPArgs->eBootCFG.delay)
{
	UINT8 i=0;
	UINT8 j;
	UINT8 x,y,z;

	Selection = OEMReadDebugByte();

	// Always jump to menu
	//Selection=0x20;
	if ((Selection == 0x20) || (Selection == 0x0d))
	{
		break;
	}
	CurrTime = OEMEthGetSecs();
	if (CurrTime > PrevTime)
	{
		PrevTime = CurrTime;
		if (AutoBootDelay < 9)
			i = 11;
		else if (AutoBootDelay < 99)
			i = 12;
		else if (AutoBootDelay < 999)
			i = 13;

		for (j = 0; j < i; j++)
		{
			OEMWriteDebugByte((BYTE)0x08); // print back space
		}
		x = AutoBootDelay / 100;
		y = (AutoBootDelay % 100) / 10;
		z = ((AutoBootDelay % 100) % 10);
		OEMWriteDebugLED(0, ((x << 8) | (y << 4) | (z)));
		KITLOutputDebugString("%d seconds. ", AutoBootDelay--);
	}

}

switch (Selection)
{
case 0x00: // fall through if nothing typed
case 0x0d: // user canceled wait
	{
		if (g_pBSPArgs->eBootCFG.autoDownloadImage)
		{
			KITLOutputDebugString("\r\nStarting auto download ... \r\n");
		}
		else
		{
			KITLOutputDebugString("\r\nLaunching flash image	... \r\n");
		}
		break;
	}
case 0x20:
	{
		BLMenu();
	}
}

// save the BSP ARGs if there are any changes.
```

```
 if (memcmp(&oldBSPArgs, g_pBSPArgs, sizeof(oldBSPArgs)) != 0)
{
      StoreBSPCFG(g_pBSPArgs);
}

if ((g_DownloadImage) )
{     // If we need to download an image, locate and initialize an download media.

      // First, to mark the boot loader as RESERVED in flash.
      // This should be done after bootimage is burned into Flash, but it's hard to know when.
      // So put the code here before downloading OS image. KITLOutputDebugString("Mark
      the bootimage area in Flash as RESERVED.\r\n");

      if (g_FlashInfo.wDataBytesPerSector == LB_BYTES_PER_PAGE)
            ReservedSpareArea(0, LB_FLASH_BLOCK_EBOOTCFG_START + 1);
      else
            ReservedSpareArea(0, FLASH_BLOCK_EBOOTCFG_START + 1);

      if (InitSpecifiedTransportLayer(&g_pBSPArgs->kitl)!=-1)
      {
            // Make sure MAC address has been programmed.
            // The InitSpecifiedTransportLayer() will update the mac[] by MAC info inside the chip.
            if (!g_pBSPArgs->kitl.mac[0] && !g_pBSPArgs->kitl.mac[1]
               && !g_pBSPArgs->kitl.mac[2])
            {
                  KITLOutputDebugString("ERROR: Invalid Ethernet address read from Ethernet
                  controller.\r\n");
                  return(FALSE);
            }
            else
            {
                  KITLOutputDebugString("INFO:  MAC  address:  %x-%x-%x-%x-%x-%x\r\n",
                        g_pBSPArgs->kitl.mac[0]  & 0x00FF, g_pBSPArgs->kitl.mac[0]  >> 8,
                        g_pBSPArgs->kitl.mac[1]  & 0x00FF, g_pBSPArgs->kitl.mac[1]  >> 8,
                        g_pBSPArgs->kitl.mac[2] & 0x00FF, g_pBSPArgs->kitl.mac[2] >> 8);
            }
      }
      else
      {
            KITLOutputDebugString("ERROR: Cannot initialize specified download transport
            media. \r\nSystem Halted...\r\n");
            return FALSE;      //cannot start download, halt the Eboot.
      }
}
// Platform initialization is done.
// After return, BLCOMMON will continue to do JUMP or DOWNLOAD.
return(TRUE);
 }
```

OEMPreDownload()　函數： 在下載作業系統前執行這個函數，它可以用來設

定如何進行Image　檔案載。

例如,可以設定成從網路下載或者跳過下載直接加載FLASH中的Image檔案。

下面是此函數代碼實例：

```
//------------------------------------------------------------------------------
//
//   Function:       OEMPreDownload
//
//   Pre-download initialization routine.
//   To initialize the transport link.
//
DWORD OEMPreDownload(void)
{
UINT32 SubnetMask;
BOOL       fGotJumpImg = FALSE, fGotIP = FALSE;
UINT32 DHCPLeaseTime = 0;
UINT32 *pDHCPLeaseTime = &DHCPLeaseTime;
UINT32 BootFlags = 0;


BootDEBUGMSG(DBG_FUNC, (L"+%s()\r\n",TEXT(__FUNCTION____)));
// Create device name based on Ethernet address (this is how Platform Builder identifies this
 device).
//
OALKitlCreateName(BSP_DEVICE_PREFIX, g_pBSPArgs->kitl.mac, g_pBSPArgs->deviceId);
OALInitUUID(g_pBSPArgs);
StoreBSPCFG(g_pBSPArgs);
KITLOutputDebugString("INFO: Using device name: '%s'\r\n", g_pBSPArgs->deviceId);

// Set up optional bootloader function pointers.
//
g_pOEMVerifyMemory = OEMVerifyMemory;

g_pOEMMultiBINNotify = OEMMultiBinNotify;

// If user wants to jump to existing image - skip download...
//
if (!g_DownloadImage )
{
      return(BL_JUMP);
}

// if USB SERIAL download just finished
if( (UINT32) (g_pBSPArgs->kitl.devLoc.PhysicalLoc) ==
 USB_SERIAL_DOWNLOAD_FAKE_ADDR)
{
      if(!SerialSendBootRequest(BSP_DEVICE_PREFIX))
      {
            KITLOutputDebugString("Failed to send boot request\r\n");
            return BL_ERROR;
      }
      KITLOutputDebugString("Sending boot request...\r\n");
      // Send boot requests indefinitely
      while(!SerialWaitForBootAck(&fGotJumpImg));
      // Ack block zero to start the download
      SerialSendBlockAck(0);
      if( fGotJumpImg )
      {
            KITLOutputDebugString("Received boot request ack... jumping to image\r\n");
      }
      else
      {
            KITLOutputDebugString("Received boot request ack... starting download\r\n");
      }
```

```
        return fGotJumpImg        ? BL_JUMP : BL_DOWNLOAD;
}

if( (UINT32) (g_pBSPArgs->kitl.devLoc.PhysicalLoc) ==
 SD_MMC_DOWNLOAD_FAKE_ADDR)
{
        // for SDMMC download
        return (BL_DOWNLOAD);
}
if(!(g_pBSPArgs->kitl.flags&OAL_KITL_FLAGS_DHCP))
{
        pDHCPLeaseTime = NULL;


// Initialize the TFTP transport.
memcpy(g_DeviceAddr.wMAC, g_pBSPArgs->kitl.mac, (sizeof(UINT16) * 3));
g_DeviceAddr.dwIP        = g_pBSPArgs->kitl.ipAddress;
g_DeviceAddr.wPort = 0;
SubnetMask               = g_pBSPArgs->kitl.ipMask;

//KITLOutputDebugString("here\r\n");

// todo: do we need to init memory here? - WWB
//KITLOutputDebugString("initializing RAM from 0x%X to 0x%X...", OS_RAM_IMAGE_START,
 RAM_CA_END);
//memset((void*) OALPAtoVA(RAM_LAYOUT_OS_CA_START, FALSE), 0xFF, RAM_CA_END –
 OS_RAM_IMAGE_START);
//KITLOutputDebugString("done\r\n");

if (!EbootInitEtherTransport(&g_DeviceAddr,
                                        &SubnetMask,     // todo: why not save this if DHCP? - WWB
                                        &fGotJumpImg,
                                        pDHCPLeaseTime,
                                        EBOOT_VERSION_MAJOR,
                                        EBOOT_VERSION_MINOR,
                                        BSP_DEVICE_PREFIX,
                                        g_pBSPArgs->deviceId,
                                        EDBG_CPU_ARM720,
                                        BootFlags))
{
        KITLOutputDebugString("Failed to do EbootInitEtherTransport()\r\n");
        return (BL_ERROR);
}
BootDEBUGMSG(DBG_FUNC, (L"-%s()\r\n",TEXT(__FUNCTION____)));
return(fGotJumpImg? BL_JUMP : BL_DOWNLOAD);
}
```

OEMLaunch()　函數： 這個函數將PC指針直接設置到Image文件的開始地址，它是啟動操作系統前 BootLoader的最後一個函數,沒有返回值。在此之後，BootLoader　就消失了。 下面是此函數代碼實例：

```
//------------------------------------------------------------------------------
//
//   Function:       OEMLaunch
//
//   Invoked by BLCOMMON
//   Launch downloaded/stored image.
```

```
//
void OEMLaunch(DWORD dwImageStart,DWORD dwImageLength,DWORD dwLaunchAddr,
const ROMHDR *pRomHdr)
{
EDBG_OS_CONFIG_DATA *pCfgData;
EDBG_ADDR EshellHostAddr;
UINT32 PhysAddress;
//P_XLLP_UDC_REGISTERS_T pUSBReg;

KITLOutputDebugString("+OEMLaunch: Start 0x%x, Length 0x%x, LaunchAddr 0x%x, RomHdr
0x%x\r\n",dwImageStart, dwImageLength, dwLaunchAddr, pRomHdr);

// Wait for PB connection...
//
if (g_DownloadImage)
{
if( (UINT32) (g_pBSPArgs->kitl.devLoc.PhysicalLoc) ==
        USB_SERIAL_DOWNLOAD_FAKE_ADDR)
{
// Wait infinitely for a jump request.
// SerialWaitForJump() returns the debug transport requested by PB.
// However, we only support ethernet for debug so we ignore the
// return value.
SerialWaitForJump();
}
else if( (UINT32) (g_pBSPArgs->kitl.devLoc.PhysicalLoc) ==
        SD_MMC_DOWNLOAD_FAKE_ADDR)
{
// for SDMMC download, do nothing here.
}
else
{
if (!(pCfgData = EbootWaitForHostConnect(&g_DeviceAddr, &EshellHostAddr)))
{
KITLOutputDebugString("ERROR: EbootWaitForHostConnect failed!\r\n");
SpinForever();
}
}

// If the user selected "passive" KITL (i.e., don't connect to the target at boot time), set the
// flag in the args structure so the OS image can honor it when it boots.
//
}

BootDEBUGMSG(DBG_INFO, (L"Disconnect USB\r\n"));
//USB Software disconnect
//pUSBReg=(P_XLLP_UDC_REGISTERS_T)OALPAtoVA(MONAHANS_BASE_REG_PA_UDC,
FALSE);
//pUSBReg->up2ocr=0;

if (dwLaunchAddr==0)
{     // for downloading NB0 file or flash.bin, its dwLaunchAddr is 0, need to fix.
if (g_pBSPArgs->eBootCFG.dwLaunchAddr)        //if there is a last known good address, use it.
{
dwLaunchAddr = g_pBSPArgs->eBootCFG.dwLaunchAddr;
}
else                                                    // otherwise, use the OS RAM start.
{
dwLaunchAddr = RAM_LAYOUT_OS_CA_START;
}
```

```
}
else
{
if (g_DownloadImage && pRomHdr ==0)
{
//This is specially made for downloading XIP.bin...
// looking for pTOC in XIP.bin and fix up the PhysStart/PhysLen inside the eBootCFG.
ROMHDR * pTOC=NULL;
if (*(LPDWORD) (dwLaunchAddr - 0x1000 + ROM_SIGNATURE_OFFSET) ==
        ROM_SIGNATURE)
{
pTOC=    (ROMHDR *) (* (LPDWORD) ( dwLaunchAddr - 0x1000 +
            ROM_TOC_POINTER_OFFSET));
KITLOutputDebugString("pTOC found at:0x%x\r\nFixing the ImageStart to:0x%x\r\nFixing the
ImageLength to:0x%x\r\n", pTOC, pTOC->physfirst, pTOC->physlast - pTOC->physfirst);
dwImageStart=pTOC->physfirst;
dwImageLength=pTOC->physlast - pTOC->physfirst;
}
}

if (g_pBSPArgs->eBootCFG.dwLaunchAddr != dwLaunchAddr ||
   g_pBSPArgs->eBootCFG.dwPhysStart       != dwImageStart ||
   g_pBSPArgs->eBootCFG.dwPhysLen       != dwImageLength)
{
g_pBSPArgs->eBootCFG.dwLaunchAddr = dwLaunchAddr;
g_pBSPArgs->eBootCFG.dwPhysStart       = dwImageStart;
g_pBSPArgs->eBootCFG.dwPhysLen       = dwImageLength;
g_pBSPArgs->eBootCFG.dwStoreAddr       = 0;

StoreBSPCFG(g_pBSPArgs);
}
}
EdbgOutputDebugString("NK Store error write == start=0x%x    length=0x%x
addr=0x%x\r\n",dwImageStart,dwImageLength,dwLaunchAddr);
EdbgOutputDebugString("NK Store error read == start=0x%x    length=0x%x
addr=0x%x\r\n",g_pBSPArgs->eBootCFG.dwPhysStart,g_pBSPArgs->eBootCFG.dwPhysLen,
g_pBSPArgs->eBootCFG.dwLaunchAddr);

if (!g_DownloadImage)
{

if (!BLReadOSImg(g_pBSPArgs->eBootCFG.dwPhysStart,
     g_pBSPArgs->eBootCFG.dwPhysLen))
{
KITLOutputDebugString("ERROR: StoreEBSPCFG: failed to load configuration.\r\n");
}
LoadImageDDR(FALSE);      // reload the Image from Flash to DDR, if it's not a download path.
}
else
{
if (!BLWriteOSImg(dwImageStart, dwImageLength))
{
KITLOutputDebugString("ERROR: StoreEBSPCFG: failed to load configuration.\r\n");
}
if (WriteImageToPartition() == FALSE)
SpinForever();
}

#ifdef DEBUG
{
```

```
ROMHDR* nk_pTOC=FindTOC((DWORD*) RAM_LAYOUT_OS_CA_START);
DumpTOC(nk_pTOC);
}
#endif

BootDEBUGMSG(DBG_INFO, (L"BSPARG: dwLaunchAddr 0x%x, dwPhysStart 0x%x,
dwPhysLen 0x%x\r\n",g_pBSPArgs->eBootCFG.dwLaunchAddr,
g_pBSPArgs->eBootCFG.dwPhysStart, g_pBSPArgs->eBootCFG.dwPhysLen));
// Translate the image start address (virtual address) to a physical address.
//
PhysAddress = (UINT32) OALVAtoPA((VOID *)dwLaunchAddr);
KITLOutputDebugString("Download successful!    Jumping to image at 0x%x (physical
0x%x)...\r\n", dwLaunchAddr, PhysAddress);
// Jump to the image we just downloaded.
//
KITLOutputDebugString("\r\n\r\n\r\nCalling Launch()\r\n");
//KITLOutputDebugString("                                        \r\n\r\n\r\n\r\n"); // flush FIFOs.
Launch(PhysAddress);
KITLOutputDebugString("FATAL ERROR: Launch() unexpected returned!\r\n");
// Should never get here...
//
SpinForever();
}
```

1.5 Windows CE 標準BootLoader 的需求

(A) BSP 必須在加載Windows CE BootLoader 以及作業系統的時候有一個預設的模式。

(1) 在系統啟動時，即使用戶不輸入指令，也能自動地下載作業系統鏡像檔案。即便使用了啟動選單，也是有時間限制了，超過時間限制，BootLoader 應當 能自動地按照預設模式啟動作業系統；

(2) BootLoader 應當被固化在FLASH 中；

(3) BootLoader 應當使用BLCOMMON 的架構；

(4) BootLoader 應當提供FLASH 擦寫和BootLoader 自我更新的功能。這就要求 BootLoader 要分段執行，在FLASH 中執行時，主要把自身剩餘的代碼複製到 RAM 中,然後進入 RAM 中運行時就能更新在 FLASH 中的 BootLoader 鏡 像文件。如果一直在 FLASH 中執行，同時又更新 FLASH 中的數據，那樣會 引起程序的邏輯錯誤；

(5) BootLoader 應當提供向線性位址的FLASH 下載作業系統鏡像檔案，並且跳轉到其鏡像檔案數據首位址的功能；

(6) BootLoader 應當提供向線性位址的RAM 下載作業系統鏡像檔案，並且跳轉到其鏡像檔案數據首位址的功能；

(7) 在Platform Builder 中的功能控制流裡出現的任何分支,BootLoader 都應當提供支援；

(8) BootLoader 應當能使用硬體提供的任何界面來下載作業系統鏡像檔案，如並 口、串口、乙太口以及硬碟。在 Platform Builder 6.0 自帶的樣例 BootLoader 中，都以乙太口為預設的傳輸模式。

(B) 目標機硬體架構的設計應當兼顧軟體性能。硬體架構設計是否合理，這關係到後續軟體開發以及調試的效率。

(1) 目標機的FLASH 應當是可以被替換的；
(2) 目標機應當提供足夠的RAM 和FLASH 來支持debugging；`
(3) 目標機是否有LED 燈之類操作簡單的、可用於調試的設備。

(C) BootLoader 應當提供兩個啟動選項：從主機下載作業系統鏡像檔案以及從FLASH中固化的鏡像檔案啟動。在下載作業系統鏡像檔案執行時，又分為兩個步驟：先將鏡像檔案下載到 RAM 中，然後再寫入到FLASH 裡。

以上便是符合Windows CE 標準要求的BootLoader 功能及其硬件支持。這些要求保證了BootLoader 在各個平台間中邏輯結構上的一致性。

那麼BootLoader 是如何被寫入裸板的呢？實現方法有以下幾種方式。

（1）使用XDB軟件和仿真器。先將BootLoader的鏡像文件通過JTAG下載到目標機的RAM中，然後在XDB中運行FLASH的燒寫軟件，這樣可以把RAM中的數據寫入FLASH；
（2）使用專門的FLASH 編程器，將BootLoader 寫如FLASH（注意，這時FLASH 還沒有插入目標機，不受CPU控制）然後將燒寫完畢的FLASH 插入目標機中。
（3）在BootLoader 已經駐留在FLASH 的情況下，可以通過BootLoader 實現自我更 新的功能。

1.6 編譯**BootLoader** 程序

BootLoader 程式可以透過PB 的集成編譯環境編譯鏈接，控制檔案為.bib 檔案，下面是一個簡單的BootLoader 的eboot.bib 檔案：

```
;*******************************************************************************
; TITLE:            EBOOT.BIB
;
; Ethernet Boot Loader Source Module
;
; Memory Map
; ----------
;*******************************************************************************
;*      RAM Layout: Nand boot; non IU
;*        ---------------------------------------
;*        ---------------------------------------> 8400_0000
;*        |      Display for ULDR UI      |                 ---------------------------------------> 83D0_0000
;*        ---------------------------------------> 83CC_7000 |      EBT RAMIMAGE (1MB)   |
;*        |                                     |                  ---------------------------------------> 83C0_0000
;*        |                                     |                 |      EBT/IPL Stack (1MB)      |
;*        |                                     |                 ---------------------------------------> 83B0_0000
;*        |                                     |                 |      EBT/IPL RAM (1MB-16KB) |
;*        |                                     |                 ---------------------------------------> 83A0_4000
;*        |                                     |                 |      EBT Pagetables (16KB)    |
;*        |                                     |                 ---------------------------------------> 83A0_0000
;*        |                                     |                 |      IPL RAMIMAGE (1MB)    |
;*        |                                     |                 |                                       |
;*        |                                     |                 ---------------------------------------> 8390_0000
;*        |                                     |                 |                                       |
;*        |           OS      RAM         |                  --------------------------------------->
```

```
;*          |                                    |
;*          |                                    |
;*          |                                    |
;*          |                                    |
;*          |                                    |
;*          |                                    |
;*          |                                    |
;*          |                                    |
;*          |                                    |
;*          |                                    |
;*          |------------------------------------| *autosize can alter the start
;*          |             NK RAMIMAGE            |
;*          |                                    |
;*          ------------------------------------> 8000_1000
;*          |                                    |
;*          |             EBT ARGS(4K)           |
;*          ------------------------------------> 8000_0000
;*
;*****************************************************************************
```

MEMORY

```
;     Name         Start(uva)       Size       Type
;     -------      --------         --------   ----
EBOOT        83C00000       00060000
RAMIMAGE STACK                        83B00000
00100000   RESERVED
;RAM         83A04000       000FC000       RAM
RAM          83D00000    00400000    RAM
PT           83A00000    00004000    RESERVED
ARGS         80000000       00001000         RESERVED
```

CONFIG

```
AUTOSIZE=OFF
COMPRESSION=OFF
PROFILE=OFF
KERNELFIXUPS=ON

ROMSTART=83C00000
ROMWIDTH=32
ROMSIZE=00060000

; The bootloader is fixed up to run from RAM, but is stored in flash.       ROMOFFSET
; adjusts the .bin file record headers to that they lie in flash (this tells the
; bootloader already on the device to store the image in flash).
; The flash address we need to put it is 0x00040000 (Block 2 in Large Block NAND)
; 0x83c00000 + 0x7c440000 = 0x00040000
ROMOFFSET=7c440000
```

MODULES

```
;     Name       Path                                                                    Memory Type
;     ------    ---------------------------------------------          -----------
     nk.exe       $(_TARGETPLATROOT)\target\$(_TGTCPU)\$(WINCEDEBUG)\eboot.exe       EBOOT
```

MEMORY 部分，定義了生成的映像檔案的目標位址，以及程式執行可以使用的內存空間。

CONFIG 部分，COMPRESSION 是否對目標代碼進行壓縮；ROMSTART 與 ROMSIZE 、ROMWIDTH 、ROMOFFSET 共同定義了開發平台上

MODULES 部分，定義了BootLoader 所包含的文件，一般就只有一個文件eboot.exe。

BootLoader 文件的下載有很多種方法：可以通過仿真器下載；可以通過其他調試程序下載；還可以直接燒寫到FLASH 中。需要說明的是，這些方法可能會要求不同的映 像格式。在PB環境下，可以生成的.nb0（用於直接燒寫FLASH）以及和Windows CE 映像一樣的.bin格式。

1.7 參考文獻

How to Develop a Boot Loader

Implementing the Boot Loader StartUp Function

Implementing the OEMPlatformInit Function

Implementing the OEMPreDownload Function

Implementing the OEMLaunch Function

Implementing the Serial Debug Functions

Creating the Boot Loader .bib File

第2章 Windows CE 6.0電源管理

2-1 Windows CE 6.0 OAL 電源管理

電源管理可以提高整個系統電源的效率，通常需要為系統的每一個設備都提供電源管理功能。本節分別介紹電源管理器、不同電源狀態之間的切換以及OAL 電源管理函數等。

2-1.1 電源管理器

電源管理器用於管理設備的電源並提高整個Windows CE 作業系統的效率，對於電源管理器，我們應該理解以下幾點：

1. 電源管理可以用來減少目標設備的電源消耗並在系統處於重置、執行、空閒和暫停狀態時維持和保護位於RAM 中的檔案系統。
2. 電源管理器為設備能夠智慧地管理自身的電源提供了框架，並且在設備的電源狀態與整個 系統的電源狀態之間提供了一種隔離機制，它允許當作業系統執行時關閉一些設備的電 源，或者當作業系統的大部分被暫時允許有些設備的電源處於打開狀態。
3. 電源管理器被實現為動態庫Pm.dll，它並不限制平台的電源需求或設計。
4. 在默認情況下，電源管理器被內建到作業系統映射中，但可以透過定義。
5. 開發者可以定制自己的系統電源狀態：如 RunAC、RunDC、暫停等，這些狀態不是預先定義的，也沒有必要是線性排列的，開發者可以在系統配置中將電源狀態名作為註冊鍵，系 統對可以定義多少個系統電源狀態沒有限制。
6. 開發者要負責顯示地定義設備電源狀態與系統電源狀態之間的映射。
7. 設備電源狀態是靜態的、預先定義的，電源管理器將一個設備的狀態傳遞給驅動程式，驅動程式負責將設備狀態映射到設備並在物理設備上完成適當的狀態轉換。

物理設備沒有必要必須支援所有的設備電源狀態。所有設備都必須支援的設備電源狀態是全開（Full On）狀態 D0。當一個驅動程式請求進入一個設備不支援的電源狀態時，進入下一個支援的 可用電源狀態。如果一個設備不能喚醒系統，那麼它應該關閉系統而不是讓他停滯在暫停（Suspend） 狀態。

當一個設備驅動程式被載入時，它應該使這個設備進入全開（Full On）狀態 D0，在一個驅動 程式被卸載時，如果可能則應該使這個設備進入關閉（OFF）狀態 D4。表 6-1 所示為一個設備可能 支援的所有電源狀態。

表6-1 設備電源狀態

設備電源狀態	註冊鍵	描　述
FullOn	D0	設備被打開和執行的狀態,設備接收來自系統的全部電量並為用戶提供全部的功能
LowOn	D1	設備以比D0 更低的電量或性能工作的全功能狀態。D1 可以被應用於正在被使用的設備，這時設備不是工作在最佳的性能狀態
Standby	D2	設備處於被部分供電並在當設備被請求時可以喚醒自己的狀態
Sleep	D3	設備被部分供電並在需要時能以自己的電量去喚醒系統電源的狀態，D3 所消耗的電量必須小於或等於D2 所消耗的電量
Off	D4	設備沒有被供電的狀態。處於D4 狀態的設備不應該消耗明顯的電量

8. 隨電源管理系統存在一組核心IOCTL，電源管理器使用這些IOCTL 來作業系統的電源狀態。這些IOCTL 包含以下幾類：
   - IOCTL HAL ENABLE WAKE：
   - IOCTL HAL DISABLE WAKE：
   - IOCTL POWER SET：
   - IOCTL POWER GET：
   - IOCTL POWER QUERY：
   - IOCTL POWER CAPABILITIES：
   - IOCTL POWER SEQUENCE：
   - IOCTL REGISTER POWER RELATIONSHIP：

2-1.2 系統電源狀態到設備電源狀態的映射

開發者可以在系統配置中將電源狀態名定義為註冊鍵。系統電源狀態到設備電源狀態的映射被 列舉為註冊表裏每一個電源狀態名下的值，註冊表定義如下：

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\Name
Flag：REG_DWORD：xxxx                ; “Flag” = dword：0x00080000
（Default）REG_DWORD：Dx             ; “Default”= dword：0    ；  D0
DeviceName：REG_DWORD：Dx            ; “COM2”= dword：1 ; D1
```

表6-2　描述了上述註冊表中各個職所表示的意義。

表6-2　電源狀態映射的註冊表值

註冊表值	描 述
Name	定義系統的電源狀態名：On、UserIdle 和Suspend
Flags	表示POWER_STATE_XXX 值的遮罩，在Pm.h 中定義；或者是OEM 定義的電源狀態標誌
（Default）	所有設備的默認電源設置。當沒有定義默認設置時，電源管理器假定設備的默認電源狀態為D0
Dx	設備電源狀態，設置為D0、D1、D2、D3 或D4。這是再作業系統電源狀態Name 時設備執行的狀態
DeviceName	可選，定義除（Default）之外有特定設備電源狀態需求的設備，如COM1 = dword：3

如果系統配置沒有定義任何系統電源狀態鍵，那麼所有設備的默認狀態都為D0。

下面是映射系統電源狀態與設備電源狀態的一個例子，它將On 映射到D0，將UserIdle 映射到了D1，將Suspend 映射到D3：

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\On
“Default” = dword：0                    ; D0
“Flag” = dword：0x00080001
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\UserIdle
“Default” = dword：1                    ; D1
“Flag” = dword：0x00080002
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\Susend
“Default” = dword：3                    ; D3
“Flag” = dword：0x00280003
```

下面註冊表程式碼展示了如何定義自己的電源狀態：

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\On
“Default” = dword：0                    ; D0
“COM2:” = dword：1
```

在以上例子中，在系統電源狀態On，除COM2 進入D1 狀態之外，電源管理器迫使其他所有設備進入D0 狀態。

2-1.3 電源狀態間的切換

Windows CE 系統中不同電源狀態之間的切換。

1. No Power On

當打開設備電源時，它清除檔案系統和工作RAM，進入On 狀態。

2. Cold Boot

Cold Boot（冷啟動）是指硬體平台的完全重定，它清除檔案系統和工作RAM。這種情況出現在從系統中移除所有電池的時候。

3. Warm Boot

Warm Boot（熱啟動）清除工作RAM，這種情況出現在透過一個程式呼叫熱啟動或用戶按下重 置（Reset）按鈕的時候。

4. On to Idle

當核心沒有任何執行緒在執行時，它將電源狀態改變為Idle（空閒）然後CPU 等待一個中斷。

5. Idle to On

當出現一個中斷時，電源的狀態由Idle 狀態轉換為On 狀態。在典型情況下，計時器中斷引起這種轉換。

6. On to Suspend

當出現下列事件之一時，電源的狀態由On 狀態轉換為Suspend 狀態：

- 非活動計時器超時；
- 用戶按下Off 按鈕。 另外，開發者也可以添加其他事件，如打開設備的電池蓋板或電池電量太低等中斷事件，使電

源狀態由On 狀態轉換為Suspend 狀態。

7. Suspend to On

當出現下列事件之一時，電源狀態由Suspend 狀態轉換為On 狀態：

- 用戶按下了On 按鈕：
- 出現了一個警報（Alarm）事件；
- 出現了任何喚醒（Wake.up）事件。

8. On to Critical Off

當出現電池電量極低事件時，電源狀態由On 狀態轉為critical Off（緊急關閉）狀態；當更換了 鋅電池時，系統將完成一個熱啟動。

2-1.4 暫停狀態的GWES 控制

開發者可以透過修改註冊表來允許GWES 控制電源由On 狀態轉換為Suspend 狀態，註冊表例程式碼如下：

```
[HKEY\System\CurrentControlSet\Control\Power]
“BattPowerOff”=dword：300
“ExtPowerOff”=dword：0
“WakeupPowerOff”=dword：0
“ScreenPowerOff”=dword：0
```

各個註冊表鍵所表示的意義如表6-3 所示。

註冊表鍵	預設值	描 述
BattPowerOff	300	使用電池作為電源時，在系統暫停之前沒有用戶輸入的秒數
ExtPowerOff	0	使用外部電源時，在系統暫停之前沒有用戶輸入的秒數。0 表示不暫停系統
WakeupPowerOff	60	在一個非用戶事件（如 Alarm 事件）喚醒之後，在系統暫停之前沒有用戶輸入的秒數
ScreePowerOff	0	在GWES 啟動一個螢幕保護之前，沒有用戶輸入的秒數。0 表示不啟動螢幕保護

2-1.5 OAL 中的電源管理函數

當系統改變了它的電源狀態時，會呼叫 OAL 電源管理函數來關閉系統電源或使它處於空閒狀 態，這些系統呼叫既可以由硬體事件也可以由軟體觸發。如透過一個電源開關或一個空閒計時器觸 發等。為此，在OAL 中可以呼叫的電源管理函數有以下兩個：

- OEMIdle：
- OEMPowerOff：

當系統沒有執行緒執行時，核心呼叫OEMIdle 函數，OEM 設備被請求進入休眠或空閒狀態。當 OEMIdle 函數被呼叫時,它首先保存當前狀態,設置記憶體為更新狀態,再停止時脈並暫停執行。然後，當出現一個中斷（包括一個可調度的終端）時，設備結束空閒狀態並恢復到空閒前的狀態，同時調度器被再次呼叫。如果在設置的時間內仍然沒有執行緒執行,那麼OEMIdle 函數會被再次呼叫。

OEMIdle 和OEMPower 函數是完全與CPU 相關的，對於不同的CPU，其實現方法可能完全不同。

除了 OAL 中的電源管理函數外，所有的設備驅動程式都應該實現他自己的電源管理函數，當系統改變電源狀態時，設備的電源管理函數被呼叫。設備的電源管理函數取決於設備驅動程式的類型。流設備驅動程式必須實現XXX_PowerUp 函數和XXX_PowerDown 函數，其中XXX 代表識別設備驅動程式的首碼，在註冊表中定義。同樣，其他類型的設備驅動程式有他們自己的電源管理函數。

2-2 驅動程式的電源管理

電源管理器允許開發者以簡單、獨立於基本的Windows CE 電源管理模型的方式管理設備，電 源管理器介面在不犧牲與基本的電源模型相容性的情況下為 OEM 和驅動程式開發者提供了靈活性。

2-2.1 電源管理器的架構

Windows CE 電源管理器的架構如圖6-1 所示。

使用電源管理器，設備接收作為 I/O 控制程式碼（IOCTL）形式的電源狀態變化的通知
（Notification）由於 IOCTL 線上程的上下文中執行，所以在如何實現電源狀態變化方面驅動程式開發者有更多靈活性。同時，使用 IOCTL 管理電源使設備的電源狀態與整個作業系統的電源狀態 欄分開來，當作業系統正在執行時，一些設備可以關閉自己的電源，而當絕大多數作業系統暫停時， 另外一些設備則可以保持原有狀態。

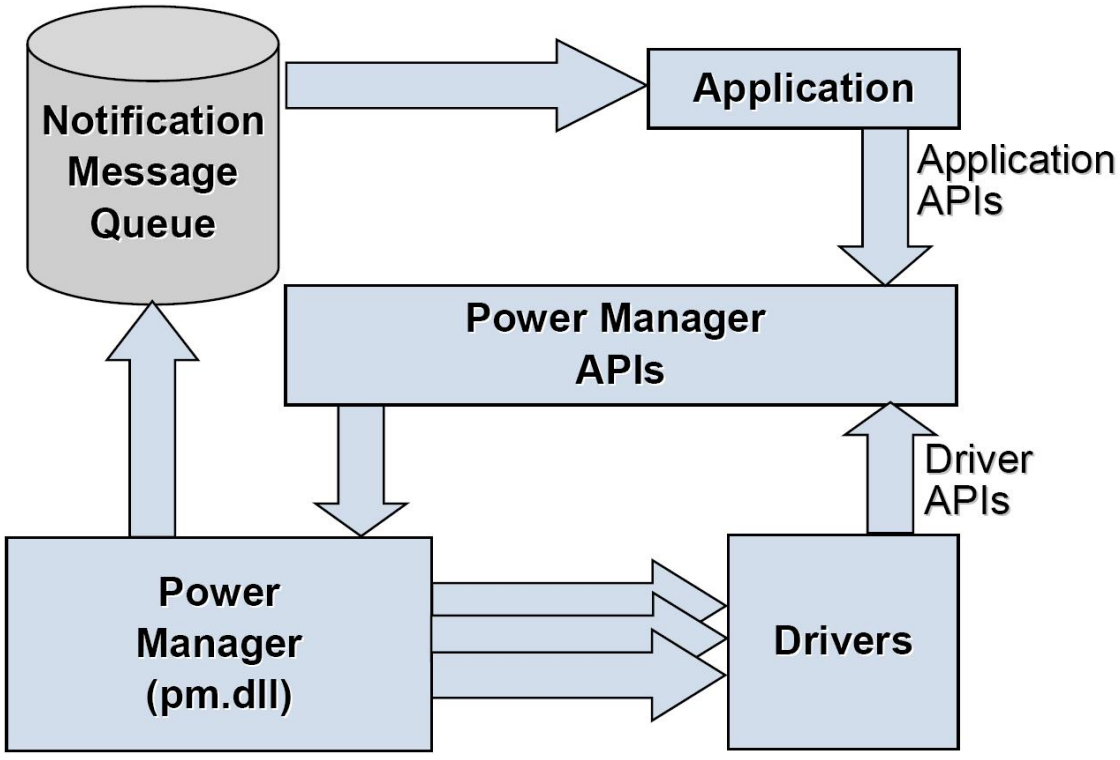


圖6-1 Windows CE 電源管理的架構

除了管理設備電源外，電源管理器還要負責通知（Notify）應用程式關於電源狀態的事件，例如當作業系統從暫時狀態恢復時通知相關的應用程式。

電源管理器被實現為一個名為Pm.dll 的動態連結程式庫而直接與Device.exe 進行鏈結，當電源管理 應用程式編程介面被呼叫時，Device.exe 呼叫位於Pm.dll 中的入口點。

Windows CE 6.0 提供了 Pm.dll 的源程式碼，它位於%_WINCEROOT%\Public\Common\Oak\Drivers\Pm 的資料夾中，開發者可以對它進行定制。

電源管理器作為設備、應用程式和作業系統電源狀態之間的一個仲介，它在這三者之間遵循下 列通信規則：

- 作業系統電源狀態對所有設備施加最大電量消耗的限制。
- 為了獲得最小的性能等級，應用程式對特定的設備施加最小電量消耗的限制。
- 只要設備在它們的最大與最小限制之間保持電源等級，電源管理器允許設備智慧地管理它們自己的電源。
- 如果最小的電量消耗限制設置得比最大值還高，那麼當應用程式需要存取這個設備時，設備的電源將保持最大的電量值。
- 設備可以實現一個或多個設備電源狀態，設備電源狀態被限制為一個有限數。
- 如果作業系統切換到暫停狀態，那麼應用程式將被施加一個最小的電量限制。
- 系統電源狀態對所有設備描述一個最大的設備電源狀態，系統電源狀態是由 OEM 定義 的，並在註冊表中被描述，也可能透過編碼使電源管理器支援它們。OEM 可以定義任意數量 的系統電源狀態。

在電源管理器框架內，OEM 定義作業系統電源狀態以建立最大設備電源框架，設備呼叫 DevicePowerNotify 函數來調整它們自己的電源等級，而應用程式呼叫 SetPowerRequirement 函數來 驗證需要設備執行在一個可接受的電源等級上。

2-2.2 電源狀態

電源管理器期望所有被管理的設備都支援一個或多個設備電源狀態，設備必須像電源管理器報告他們的電源消耗特徵，設備電源狀態通常需要在性能與電量消耗之間進行折衷。電源狀態包括系統電源狀態和設備電源狀態，系統電源狀態是由 OEM 在 OAL 中定義的，而設備電源狀態是由驅動程式開發者在驅動程式中定義的。設備管理器在由 OEM 定義的系統電源狀態的範圍內管理設備電 源狀態，系統電源狀態對設備電源狀態施加了一個上界。

設備的電源狀態是被預先靜態定義的，電源管理器將設備狀態傳遞給一個設備驅動，而驅動程 式負責將這個狀態映射為設備的電源能力，然後對物理設備完成可用的狀態轉換。

電源管理器負責將系統電源狀態映射為相應的設備電源狀態，開發者也可以在註冊表中顯示地 定義設備電源狀態與系統電源狀態之間的映射。

當一個設備驅動程式被載入時應該使設備進入D0 狀態；在一個驅動程式被卸載前，如果可能則應該使設備進入 D4 狀態；如果一個設備在啟動時要進入另一種設備電源狀態，那麼它可以在處理IOCTL POWER CAPABILITIES 時發送一個DevicePowerNotify 請求。

2-2.3 電源管理介面

電源管理器管理三種不同類型的介面。

1. 設備驅動介面 電源管理器使用兩種機制與電源管理的驅動程式進行通信：電源管理器向下呼叫設備驅動來確

定設備的電源能力並修改設備的電源狀態，設備向上呼叫設備管理器來請求設備電源狀態的變化。向下呼叫是透過 IOCTL 實現的(如表 6-4 所示)而向上呼叫則使用 DevicePowerNotify 應用程式編成介面。

表6-4 電源管理的設備控制IOCTL

IOCTL	描 述
IOCTL_POWER_CAPABILITIES	請求設備向電源管理器報告它支援哪些電源狀態及它們的特徵是什麼
IOCTL_POWER_SET	請求設備更新他的設備電源狀態
IOCTL_POWER_QUERY	詢問設備是否已經準備好進入一種新的設備電源狀態
IOCTL_REGISTER_GET	請求設備向電源管理器報告它的當前設備電源狀態
IOCTL_REGISTER_POWER_RELATIONSHIP	通知父設備註冊他控制的所有設備

由於電源管理器使用DeviceIoControl 與電源管理的設備進行通信,所以設備驅動必需導出一個流介面。然而，電源管理器也提供了一種與非流驅動直接進行通信的機制，這種方法是由對打開一 個設備控制碼、發送一個請求等的抽象組成的，絕大多數設備支援流介面。

2. 應用程式介面

電源管理器提供了幾個函數以允許應用程式影響系統和設備的電源管理，表6-5 列出了這些函數。

表6-5 電源管理的應用程式編程介面

函 數	描 述
GetSystemPowerState	返回當前系統電源狀態的名稱
SetSystemPowerState	請求電源管理器改變當前系統電源狀態
SetPowerRequirement	請求電源管理器將給定設備的電源狀態維持在最低
ReleasePowerRequirement	通知電源管理器不再需要將給定設備的電源狀態維
GetDevicePower	返回給定設備的當前電源狀態
SetDevicePower	請求電源管理器改變給定設備的電源狀態

3. 通知（Notification）介面

電源管理器提供了表6-6 所示的函數來允許應用程式接受電源相關的事件通知並決定系統電源狀態的改變。

表6-6 電源管理的通知介面函數

函 數	描 述
RequestPowerNotifications	請求電源管理器發送電源事件的通知
StopPowerNotifications	取消由RequestPowerNotifications 發送的通知請求

電源事件的通知是透過消息佇列提交的，為了使用通知，應用程式必須創建一個消息佇列並將它的控制碼透過 RequestPowerNotifications 傳遞給設備管理器，然後應用程式再創建另一個執行緒等待消息佇列中的消息。

電源管理器定義了 PBT_RESUME 、PBT_POWERSTATUSCHANGE 、PBT_TRANSITION 和

PBT_POWERINFOCHANGE 通知。

2-2.4 在驅動程式中添加電源管理

驅動程式開發者可以在設備的驅動程式中添加電源管理以減少目標設備對電源的消耗。故驅動程式必須導出流介面 XXX_PowerUp 和 XXX_PowerDown，一旦實現了流介面，還必須在設備驅動中添加對電源管理的支援。

下面過程在一個設備驅動程式中添加對電源管理的支援：

1. 創建一個流周邊設備驅動。
2. 在流周邊設備驅動中添加對電源管理I/O 控制程式碼（IOCTL）的支援。

為了創建一個能夠識別目標設備電源狀態的設備驅動，開發者必須首先創建一個能夠非 COM

相關的設備介面的設備驅動，這個非COM 相關的設備介面接著定義設備

是電源管理的。 驅動程式開發者可以使用下列方式通知一個設備介面：

- 在註冊表鍵的IClass 值中定義這個介面，這個鍵值用於啟動設備。
- 定義在設備驅動的Init 函數中使用的Active 註冊鍵裡的IClass 值。
- 使用ActivateDeviceEx 函數的REGINI 參數定義IClass 值。
- 在設備驅動中顯示地呼叫AdvertiseInterface 函數。 典型情況下，一旦一個驅動被通知為一個電源管理的驅動，驅動程式只需要處理來自電源管理器的DeviceIoControl 呼叫。電源管理器使用 IOCTL 程式碼與設備進行通信，設備的電源管理 I/O 如表

6-4 所示。

3. 通知電源管理器驅動程式是電源管理的，這透過電源管理介面函數來實現。

4. 在驅動程式中實現設備電源狀態。 當將一個驅動程式配置為支援電源管理時，必須為這個驅動程式的每一個入口點確定電源狀態值。 如果開發者精確地確定了這些電源狀態值，那麼取決於目標設備的當前電源狀態，驅動程式將以適 當的方式工作。

需要注意的是在對 XXX_PowerUp 呼叫時，驅動程式並不一定是打開（Power up）目標設備，取而 代之的是驅動程式必須將電源狀態恢復到由電源管理器設置的值,這個值很可能是 D3 或 D4 狀態， 而不是想像中的 D0 狀態。同樣，在對 XXX_PowerDown 呼叫時，驅動程式並不一定是關閉（Power down）目標設備。

第3章 PXA310的Interrupt概念及應用

在 Windows CE 6.0 中可以使用 Interrupt 的方式，來減少系統管理負載，進而增加作業系統的運 作效率，本章節將詳細介紹 Interrupt(中斷)的相關概念及在 PXA310 開發平台上如何實現一個以 Interrupt做為觸發依據的驅動程式。

⌘ ISR(Interrupt Service Routine)及IST(Interrupt Service Thread)

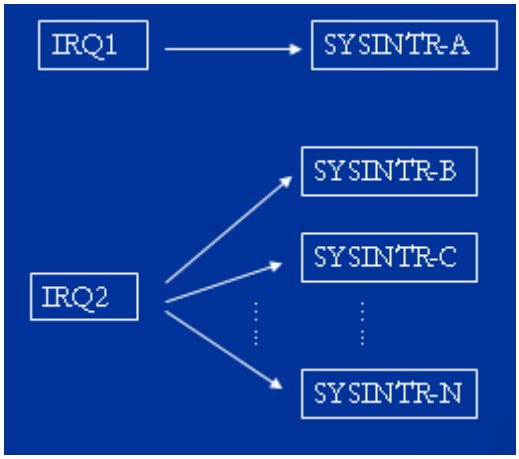
ISR 是執行在 OAL 層的一個服務程序。其目的是在當外部硬體中斷訊號產生後，判別該中斷訊 號是由哪個設備所產生的?並根據預先建立好的中斷代碼表,回傳該中斷識別碼(Interrupt ID) 給作業系統核心。ISR 程序中，大多都只包含了最精簡的、最必要的中斷處理動作，例如：識 別中斷產生源、遮照優先權低的中斷要求。將其他需要較多處理時間或較複雜的處理動作，交 由其他程序(IST)處理，以增加系統中斷處理的效率。

IST 是執行在設備驅動程式端的中斷處理程序。其目的是在 ISR 回傳中斷識別碼之後，繼續處 理中斷處理的後續動作，例如需要大量運算時間的處理程序。系統核心在 ISR 回傳中斷識別碼 之後，會依據不同的識別碼來呼叫不同的IST。

⌘ IRQ及SYSINTR

IRQ(Interrupt Request)中斷要求。可以看成是外部設備所產生的硬體中斷訊號，在硬體 IRQ 產生後，系統中的中斷控制器就會接收並開始做中斷訊號的後續處理。每一個 IRQ 可以在系統 中設定對應到一個 SYSINTR(獨立中斷)或是多個 SYSINTR(共享中斷)。所有有效的 IRQ 都會對 應到一個中斷識別碼(SYSINTR)，用來識別中斷產生源。

SYSINTR,軟體中斷識別碼,是由ISR根據接收的IRQ回傳給核心的一個識別值。每一個 SYSINTR 只會對應到一個 IRQ，其存在目的是為了觸發由InterruptInitialize()函式所設定連接的事 件(EVENT)。



⌘ EVENT

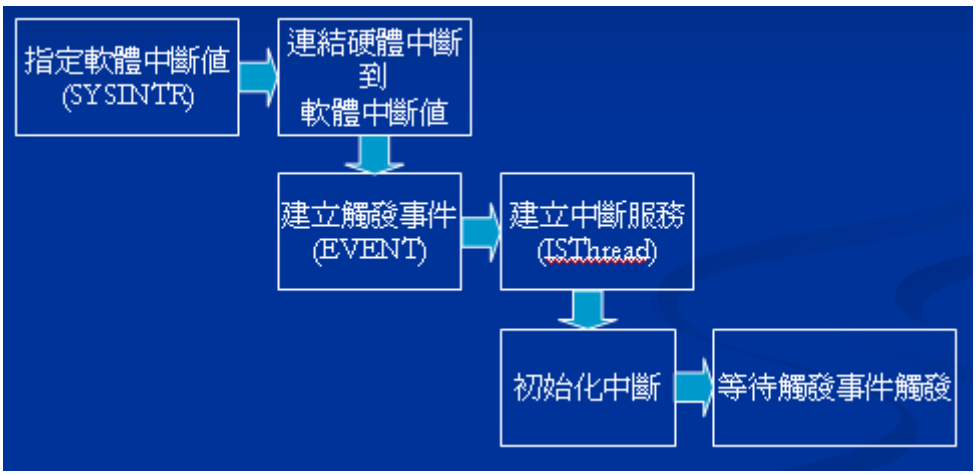
觸發事件，是由呼叫InterruptInitialize()函式來和ISR回傳的SYSINTR作連結。進而呼叫到EVENT所對應到的IST函式，繼續中斷處理動作。

⌘ 中斷處理流程

上圖說明了當一個外部中斷產生後，作業系統對於設備中斷處理的流程。 首先,當系統的中斷控制器接收到外部硬體中斷請求(IRQ)後,會進入系統的ISR處理程序中， 判別當次中斷是由何處產生? 再對照設定的中斷識別碼表，回傳當次的中斷識別碼(SYSINTR)， 並遮罩掉比當次發生中斷優先權低的所有中斷要求。當ISR回傳當次中斷的SYSINTR後，會使
用 SYSINTR 找到一個對應到此 SYSINTR 的事件(EVENT)做觸發，最後在設備驅動程式中，有建立中斷後續處理的 Thread，則會進入 Thread，執行 Thread 中的動作。最後，在 Thread 執行結束後，需通知作業系統該次中斷已完成中斷相關處理動作，要求作業系統清除該次中斷要 求，並解除對低優先權的外部設備中斷請求遮罩，完成整個中斷處理流程。

接下來的章節中將以一個簡單的PXA310按鈕驅動程式,來介紹說明在PXA310開發平台上的驅 動程式中斷處理設定。

⌘ 在PXA310開發平台上建立使用中斷要求的驅動程式



使用者要在 PXA310 開發平台上，建立一個可用的中斷處理程序，需要完成如上圖所示的幾個設定。以下說明前提為，使用者已完成 GPIO 的相關建立設定(設定 MFP 輔助功能、設定 GPIO 方向、激活MFP)，並開啟GPIO的上/下緣偵測功能。

1. 新增軟體中斷識別碼(SYSINTR)

IRQ發生後，第一個相關的設定就是需要有一個中斷識別碼要被ISR回傳給核心，所以使 用者首先要新增一個軟體中斷識別碼給特定的IRQ。

以PXA310開發平台為例，新增識別碼需在\Platform\src\inc\bsp_cfg.h中新增。

```
//------------------------------------------------------------
// Static SYSINTR Mapping for driver.
#define SYSINTR_OHCI            (SYSINTR_FIRMWARE+1)  // 17
#define SYSINTR_TOUCH           (SYSINTR_FIRMWARE+2)  // 18
#define SYSINTR_TOUCH_CHANGED   (SYSINTR_FIRMWARE+3)  // 19
#define SYSINTR_USBFN           (SYSINTR_FIRMWARE+4)  // 20
#define SYSINTR_USB2OFN         (SYSINTR_FIRMWARE+5)  // 21
#define SYSINTR_FFUART      (SYSINTR_FIRMWARE+6)  // 22
#define SYSINTR_SMSC91C111      (SYSINTR_FIRMWARE+7)  // 23
#define SYSINTR_GIR             (SYSINTR_FIRMWARE+8)  // 24
#define SYSINTR_BTN             (SYSINTR_FIRMWARE+9)  // 25
#define SYSINTR_BTNP            (SYSINTR_FIRMWARE+10) // 26
#define SYSINTR_TNR             (SYSINTR_FIRMWARE+11) // 27
#define SYSINTR_RTC             (SYSINTR_FIRMWARE+12) // 28
#define SYSINTR_SDC1            (SYSINTR_FIRMWARE+13) // 29
#define SYSINTR_SDC2            (SYSINTR_FIRMWARE+14) // 30
#define SYSINTR_PMI             (SYSINTR_FIRMWARE+15) // 31
#define SYSINTR_HPD             (SYSINTR_FIRMWARE+16) // 32
```

2. 連結IRQ與SYSINTR

新增完一個新的識別碼後，接下來就需要將此識別碼連結到想連接的IRQ，讓ISR可以在此 IRQ 發生時，傳回所設定的新的 SYSINTR。PXA310 中，要連結 IRQ 與 SYSINTR 需在\SRC\OAL\OALLIB\intr.c中使用OALIntrStaticTranslate()函式來建立連結。

```
//[BTN] Interrupt Mapping

OALIntrStaticTranslate(SYSINTR_BTN,IRQ_GPIO_SHARE(PXA_GPIO_BTN1_ID)); // BTN1
OALIntrStaticTranslate(SYSINTR_BTN,IRQ_GPIO_SHARE(PXA_GPIO_BTN2_ID)); // BTN2
OALIntrStaticTranslate(SYSINTR_BTN,IRQ_GPIO_SHARE(PXA_GPIO_BTN3_ID)); // BTN3
OALIntrStaticTranslate(SYSINTR_BTN,IRQ_GPIO_SHARE(PXA_GPIO_BTN4_ID)); // BTN4
```

其中SYSINTR_BTN是設定的SYSINTR識別碼，PXA_GPIO_BTN1_ID為要連結的GPIO接腳。

3. 設立一個觸發事件(EVENT)

ISR回傳回IRQ連結的SYSINTR後，核心會觸發此SYSINTR有關的EVENT，所以如果此中 斷需要有後續的中斷處理動作，則驅動程式中需再建立EVENT來接收SYSINTR觸發。要建 立一個事件，需透過CreateEvent()函式來建立。

```
// Create a Event
gBTNIntrEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
// Enable the device driver to register an event and enable the interrupt
if (!(InterruptInitialize(g_BTNSysIntr, gBTNIntrEvent, 0, 0)))
{
    ERRORMSG(1, (TEXT("BTN_IntrThread:InterruptInitialize failed.\r\n")));
    CloseHandle(gBTNIntrEvent);
    return 0;
}
```

其中，第一個參數必須為 NULL；第二個參數為設定此事件是手動重置(TRUE)，還是自動重置(FALSE)；第三個參數設定此事件初始狀態為已觸發(TRUE)，還是未觸發(FALSE)；最後一個參數則是此事件名稱的名指標，設定為NULL則為未命名事件。

4. 連結中斷識別碼及觸發事件 成功建立觸發事件後，接下來就是要將中斷識別碼和觸發事件作連結，好讓核心可以得到 SYSINTR後，通知對應的觸發事件來作觸發。為了連結前述兩者，在PXA310開發平台上使
用InterruptInitialize()函式完成。

```
// Enable the device driver to register an event and enable the interrupt
if (!(InterruptInitialize(g_BTNSysIntr, gBTNIntrEvent, 0, 0)))
{
    ERRORMSG(1, (TEXT("BTN_IntrThread:InterruptInitialize failed.\r\n")));
    CloseHandle(gBTNIntrEvent);
    return 0;
}
```

5. 建立中斷後續服務函式(IST)

因為 ISR 只會執行最基本的中斷服務程序，所以如果在中斷發生後，需要執行其他處理，如資料讀寫、資料運算等較複雜的處理，則需要建立一個 IST 來接續 ISR 的工作，來完成剩餘的中斷處理。利用 IST 來執行剩餘的中斷處理的好處是可以利用作業系統的程序多工管理，更有效的利用系統資源完成處理程序，並減低系統的出錯機率。要建立一個 IST，需呼叫CreateThread()函式。

```
gBTNIntrThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)BTN_IntrThread, 0, 0, 0);
if (gBTNIntrThread == NULL){
    ERRORMSG(1,(TEXT("BTN_Init:CreateThread() Fail.\r\n")));
    return 0;
}
```

6. 在IST中，等待觸發事件觸發

完成前述的步驟後，接下來就是要讓IST等待所設立的觸發事件觸發，並完成IST中的所 有中斷處理程序。要讓IST可以等待單一觸發事件，需呼叫WaitForSingleObject()函式。

```
while (1)
{
    ret = WaitForSingleObject(gBTNIntrEvent, INFINITE);
    if ((ret == WAIT_OBJECT_0) && (g_bKillIST == FALSE))
    {
        if (BTN_IsPushed())
        {
```

若要等待多個觸發事件，則需要呼叫WaitForMultipleObjects()函式。

7. 結束並清除當次中斷處理程序

最後，當 IST 完成所有的中斷處理程序後，使用者必須告知系統核心，該次的中斷處理已經處理完畢，要求系統核心將該次的 IRQ 清除，並解除對中斷優先權低於此中斷的外部中斷請求(IRQ)遮罩。使優先權較低的 IRQ 可以再次被系統中斷控制器所接受。要達到上述告知動作，需呼叫InterruptDone()函式。

```
    // Signal to the kernel that interrupt processing has been completed
    InterruptDone(g_BTNSysIntr);
}
return 1;
```

8. 完成建立中斷處理程序。

註1：本章節中的相關詳細資訊內容，請參考PXA310開發平台原廠技術文件及BSP中的BUTTON驅 動程式原始程式碼。

註2：章節中所列出的windows函式詳細資訊，請參考微軟所提供的MSDN說明。

第4章 PXA310的GPIO概念及應用

⌘ 什麼是GPIO（General Purpose Input/Output）

GPIO，通用型輸入輸出（General Purpose I/O）的簡稱，其硬體接腳可以供使用者透過軟體程式自由控制使用，以滿足個別使用者的控制需求。接腳的功能，依實際應用可作為通用輸入（GPI）或通用輸出（GPO）或通用輸入與輸出（GPIO）例如：作外部信號輸入、chip select、 按鈕輸入、LED輸出等。

因為GPIO可以由使用者自由選擇輸出入狀態,因此需要有一個(組)GPIO控制暫存器用來選擇、 設定這些功能。要透過 GPIO 來作為輸入功能，則需要透過讀取暫存器來取得接腳的電壓準位； 作為輸出功能，則需要先將要輸出的資料寫入到暫存器內，再透過暫存器將資料輸出到接腳；若設定為其他特殊功能，則要使用相對應的控制暫存器來控制GPIO。

⌘ PXA310的GPIO配置及功能

PXA310 的接腳可分為兩類，單一功能接腳(SFP)及多功能接腳(MFP)，其中，MFP 可以透過控制單元的設定，來改變接腳的功能。如：一般 GPIO 功能、鍵盤掃描功能、SDIO、RS-232 等。接腳控制單元有以下特色：

- 在RESET或是低電源狀態下，控制接腳的狀態。
- 支援在RESET或是低電源狀態下，接腳最後狀態或暫存器預設接腳狀態的保留。
- 管理GPIO及其他接腳輔助功能。
- 支援軟體控制接腳輸出驅動強度。
- 支援外部中斷及接腳喚醒功能。

PXA310的多功能接腳(MFP)所支援的輔助功能，請參考Marvell技術文件PXA30x_and_PXA31x_Processor_Developer_Manual_Vol_I_System_and_Timer_Config.pdf章節4.3中的Table-12 PXA310 Processor Alternate Function Table。每個MFP接腳都可以經 由控制暫存器的設定開啟，來偵測接腳準位的上昇改變、下降改變或上下改變。有關MFP的控 制暫存器定義，請參照Marvell技術文件PXA30x_and_PXA31x_Processor_Developer_Manual_Vol_I_System_and_Timer_Config.pdf章節4.10.1中的Table-19 MFPR Bits Definitions的說明。

PXA310 提供了 128 個 GPIO 接腳讓使用者依照需求調整功能。若設定為輸入功能，可以作為簡單的輸入，或是開啟正/負緣觸發功能，作為系統外部中斷的中斷輸入訊號；若設定為輸出功 能，可以作為一般的輸出接腳，設定接腳電壓準位為 high、low。每一個 GPIO 接腳都是一個MFP 接腳，當設定為 GPIO 功能後，預設為輸入，使用者可自行更改設定為輸出或雙向接腳。 PXA310共有52個32位元的GPIO控制暫存器來控制128個GPIO。分成6大類，並以4個字母簡寫來代表暫存器名稱。所有的GPIO控制暫存器在系統RESET後將會初始為0x0。

以下簡單說明GPIO相關設定暫存器功能。

- 4個監控暫存器
  - GPLRx(GPIO Pin-Level Register)：唯讀。儲存目前的GPIO準位
- 12個方向控制暫存器
  - 4個GPDRx (GPIO Pin Direction register)：讀寫。設定GPIO輸出入方向(high：output，low：input)。
  - 4個GSDRx(GPIO Pin Set Direction Register)：唯寫。設定GPDR為high。
  - 4個GCDRx(GPIO Pin Clear Direction Register)：唯寫。設定GPDR為low。
- 8個狀態控制暫存器
  - GPSRx(GPIO Pin Output Set Regsiter)：唯寫。設定GPIO輸出high。
  - GPCRx(GPIO Pin Output Clear Regsiter)：唯寫。設定GPIO輸出為low。
- 12個上昇緣偵測暫存器
  - 4個GRERx(GPIO Rising-Edge Detect Enable register)：讀寫。開啟上昇緣偵測。
  - 4個GSRERx(GPIO Set Rising-edge Enable Register)：唯寫。設定GRERx為high。
  - 4個GCRERx(GPIO Clear Rising-edge Enable Register)：唯寫。設定GRERx為low。
- 12個下降緣偵測暫存器
  - 4 個 GFERx(GPIO Falling-Edge Detect Enable register)：讀寫。開啟下降緣偵測。
  - 4 個 GSFERx(GPIO Set Falling-edge Enable Register)：唯寫。設定 GFERx 為high。
  - 4 個 GCFERx(GPIO Clear Falling-edge Enable Register)：唯寫。設定 GFERx為low。
- 4個上下緣偵測暫存器
  - GEDRx(GPIO Edge Detect Enable register)：開啟上下緣偵測。

相關GPIO及控制暫存器詳細資訊，請參考Marvell技術文件PXA30x_and_PXA31x_Processor_Developer_Manual_Vol_I_System_and_Timer_Config.pdf章節5 General-Purpose I/O Unit。

Table 22: GPIO Register Summary

Address	GPIO Pin Assignments	Description	Page
0x40E0_0000	GPIO<31:0>	GPIO Pin-Level Registers (GPL0)	page 116
0x40E0_0004	GPIO<63:32>	GPIO Pin-Level Register 1 (GPLR1)	
0x40E0_0008	GPIO<95:64>	GPIO Pin-Level Register 2 (GPLR2)	
0x40E0_0100	GPIO<127:96>	GPIO Pin-Level Register 3 (GPLR3)	

Table 22: GPIO Register Summary (Continued)

Address	GPIO Pin Assignments	Description	Page
0x40E0_0104– 0x40E0_0108	—	Reserved	
0x40E0_000C	GPIO<31:0>	GPIO Pin Direction Registers (GPDRx)	page 117
0x40E0_0010	GPIO<63:32>	GPIO Pin Direction Register 1 (GPDR1)	
0x40E0_0014	GPIO<95:64>	GPIO Pin Direction Register 2 (GPDR2)	
0x40E0_010C	GPIO<127:96>	GPIO Pin Direction Register 3 (GPDR3)	
0x40E0_0110– 0x40E0_0114	—	Reserved	
0x40E0_0400	GPIO<31:0>	GPIO Pin Bit-Wise Set Direction Register 0 (GSDR0)	page 118
0x40E0_0404	GPIO<63:32>	GPIO Pin Bit-Wise Set Direction Register 1 (GSDR1)	
0x40E0_0408	GPIO<95:64>	GPIO Pin Bit-Wise Set Direction Register 2 (GSDR2)	
0x40E0_040C	GPIO<127:96>	GPIO Pin Bit-Wise Set Direction Register 3 (GSDR3)	
0x40E0_0410– 0x40E0_0414	—	Reserved	
0x40E0_0420	GPIO<31:0>	GPIO Pin Bit-Wise Clear Direction Register 0 (GCDR0)	page 118
0x40E0_0424	GPIO<63:32>	GPIO Pin Bit-Wise Clear Direction Register 1 (GCDR1)	
0x40E0_0428	GPIO<95:64>	GPIO Pin Bit-Wise Clear Direction Register 2 (GCDR2)	
0x40E0_042C	GPIO<127:96>	GPIO Pin Bit-Wise Clear Direction Register 3 (GCDR3)	
0x40E0_0430– 0x40E0_0434	—	Reserved	
0x40E0_0018	GPIO<31:0>	GPIO Pin Output Set Register 0 (GPSR0)	page 119
0x40E0_001C	GPIO<63:32>	GPIO Pin Output Set Register 1 (GPSR1)	
0x40E0_0020	GPIO<95:64>	GPIO Pin Output Set Register 2 (GPSR2)	
0x40E0_0118	GPIO<127:96>	GPIO Pin Output Set Register 3 (GPSR3)	
0x40E0_011C– 0x40E0_0120	—	Reserved	
0x40E0_0024	GPIO<31:0>	GPIO Pin Output Clear Register 0 (GPCR0)	
0x40E0_0028	GPIO<63:32>	GPIO Pin Output Clear Register 1 (GPCR1)	
0x40E0_002C	GPIO<95:64>	GPIO Pin Output Clear Register 2 (GPCR2)	
0x40E0_0124	GPIO<127:96>	GPIO Pin Output Clear Register 3 (GPCR3)	

Table 22: GPIO Register Summary (Continued)

Address	GPIO Pin Assignments	Description	Page
0x40E0_0128–0x40E0_012C	—	Reserved	
0x40E0_0030	GPIO<31:0>	GPIO Rising-Edge Detect-Enable Register 0 (GRER0)	page 120
0x40E0_0034	GPIO<63:32>	GPIO Rising-Edge Detect-Enable Register 1 (GRER1)	
0x40E0_0038	GPIO<95:64>	GPIO Rising-Edge Detect-Enable Register 2 (GRER2)	
0x40E0_0130	GPIO<127:96>	GPIO Rising-Edge Detect-Enable Register 3 (GRER3)	
0x40E0_0134–0x40E0_0138	—	Reserved	
0x40E0_0154–0x40EF 03FC	—	Reserved	
0x40E0_0440	GPIO<31:0>	GPIO Bit-Wise Set Rising-Edge Register 0 (GSRER0)	page 121
0x40E0_0444	GPIO<63:32>	GPIO Bit-Wise Set Rising-Edge Register 1 (GSRER1)	
0x40E0_0448	GPIO<95:64>	GPIO Bit-Wise Set Rising-Edge Register 2 (GSRER2)	
0x40E0_044C	GPIO<127:96>	GPIO Bit-Wise Set Rising-Edge Register 3 (GSRER3)	
0x40E0_0450–0x40E0_0454		Reserved	
0x40E0_0460	GPIO<31:0>	GPIO Bit-wise Clear Rising-Edge Detect-Enable Register 0 (GCRER0)	page 120
0x40E0_0464	GPIO<63:32>	GPIO Bit-wise Clear Rising-Edge Detect-Enable Register 1 (GCRER1)	
0x40E0_0468	GPIO<95:64>	GPIO Bit-wise Clear Rising-Edge Detect-Enable Register 2 (GCRER2)	
0x40E0_046C	GPIO<127:96>	GPIO Bit-wise Clear Rising-Edge Detect-Enable Register 3 (GCRER3)	
0x40E0_0470–0x40E0_00FC	—	Reserved	
0x40E0_003C	GPIO<31:0>	GPIO Falling-Edge Detect-Enable Register 0 (GFER0)	page 122
0x40E0_0040	GPIO<63:32>	GPIO Falling-Edge Detect-Enable Register 1 (GFER1)	
0x40E0_0044	GPIO<95:64>	GPIO Falling-Edge Detect-Enable Register 2 (GFER2)	
0x40E0_013C	GPIO<127:96>	GPIO Falling-Edge Detect-Enable Register3 (GFER3)	
0x40E0_0140–0x40E0_0144	—	Reserved	

⌘

Table 22: GPIO Register Summary (Continued)

Address	GPIO Pin Assignments	Description	Page
0x40E0_0480	GPIO<31:0>	GPIO Bit-Wise Set Falling-Edge Register 0 (GSFER0)	page 123
0x40E0_0484	GPIO<63:32>	GPIO Bit-Wise Set Falling-Edge Register 1 (GSFER1)	
0x40E0_0488	GPIO<95:64>	GPIO Bit-Wise Set Falling-Edge Register 2 (GSFER2)	
0x40E0_048C	GPIO<127:96>	GPIO Bit-Wise Set Falling-Edge Register 3 (GSFER3)	
0x40E0_0490–0x40E0_0494	—	Reserved	
0x40E0_04A0	GPIO<31:0>	GPIO Bit-wise Clear Falling-Edge Detect-Enable Register 0 (GCFER0)	page 122
0x40E0_04A4	GPIO<63:32>	GPIO Bit-wise Clear Falling-Edge Detect-Enable Register 1 (GCFER1)	
0x40E0_04A8	GPIO<95:64>	GPIO Bit-wise Clear Falling-Edge Detect-Enable Register 2 (GCFER2)	
0x40E0_04AC	GPIO<127:96>	GPIO Bit-wise Clear Falling-Edge Detect-Enable Register 3 (GCFER3)	
0x40E0_04B0–0x40E0_04B4	—	Reserved	
0x40E0_0048	GPIO<31:0>	GPIO Edge Detect Status Register 0 (GEDR0)	page 124
0x40E0_004C	GPIO<63:32>	GPIO Edge Detect Status Register 1 (GEDR1)	
0x40E0_0050	GPIO<95:64>	GPIO Edge Detect Status Register 2 (GEDR2)	
0x40E0_0054–0x40E0_00FC	—	Reserved	
0x40E0_0148	GPIO<127:96>	GPIO Edge Detect Status Register 3 (GEDR3)	page 124
0x40E0_014C–0x40E0_0150		Reserved	
0x40E0_04B8–0x40EF_FFFF	—	Reserved	

PXA310的GPIO應用實作

本章節將以PXA310開發平台BSP中的按鈕驅動程式為例，介紹說明如何在PXA310開發平台上 設定MFP的輔助功能為一般GPIO接腳?如何透過設定GPIO控制暫存器，設定GPIO狀態? 及如 何讀取/寫入資料從/到PXA310的GPIO接腳?

④ 使用PXA310 GPIO接腳功能的流程：

為了要使用 PXA310 所提供的 GPIO 功能，必需完成四個主要步驟。首先，使用者需要將要作為GPIO功能的MFP接腳的輔助功能調整為一般GPIO功能,完成輔助功能調整後,該MFP 接腳的控制權將會被切換到GPIO的控制暫存器，使用者即可透過GPIO控制暫存器來控制該MFP接腳。GPIO控制暫存器獲得控制權之後，使用者需要先行設定GPIO控制暫存器中 的輸出入方向(預設為輸入)及其他要求功能(上/下緣偵測)。在完成控制暫存器的相關設定後，即可激活MFP的GPIO功能，激活所設立的GPIO功能。在還沒激活GPIO功能之前， 所有的GPIO控制設定將不會真正的影響GPIO。設定完成並激活後的GPIO就可以依照使用 者的設定來讀取/寫入資料了。

④ 定義MFP接腳的輔助功能屬性

在預設路徑 Platform\zylonite_mhlv\src\drivers\mfp\MFP_platform.c 檔案中可以增加、更改設定所有MFP的輔助功能屬性及相關MFP特性。以下以開發平台上的按鈕來說明。

在 gaulNonActiveModeMFPTable 中設立了所有 MFP 接腳的名稱索引 ， 其 中 PXA_MFP_GPIO_BTNx_ID為對應到PXA310開發平台的按鈕MFP接腳GPIO 116-119腳。

```
    {PXA_MFP_GPIO_UART2_RTS_ID,     PXA_MFP_PIN_GPIO111_OFFSET,    PXA_MFP_NOT_DEFINED_VALUE},
    {PXA_MFP_GPIO_UART2_RXD_ID,     PXA_MFP_PIN_GPIO112_OFFSET,    PXA_MFP_NOT_DEFINED_VALUE},
    {PXA_MFP_GPIO_UART2_TXD_ID,     PXA_MFP_PIN_GPIO113_OFFSET,    PXA_MFP_NOT_DEFINED_VALUE},
    {PXA_MFP_GPIO_UART2_CTS_ID,     PXA_MFP_PIN_GPIO114_OFFSET,    PXA_MFP_NOT_DEFINED_VALUE},
    {PXA_MFP_GPIO_CIF_RESET_ID,     PXA_MFP_PIN_GPIO115_OFFSET,    PXA_MFP_NOT_DEFINED_VALUE},
    {PXA_MFP_GPIO_BTN1_ID,          PXA_MFP_PIN_GPIO116_OFFSET,    PXA_MFP_NOT_DEFINED_VALUE},
    {PXA_MFP_GPIO_BTN2_ID,          PXA_MFP_PIN_GPIO117_OFFSET,    PXA_MFP_NOT_DEFINED_VALUE},
    {PXA_MFP_GPIO_BTN3_ID,          PXA_MFP_PIN_GPIO118_OFFSET,    PXA_MFP_NOT_DEFINED_VALUE},
    {PXA_MFP_GPIO_BTN4_ID,          PXA_MFP_PIN_GPIO119_OFFSET,    PXA_MFP_NOT_DEFINED_VALUE},
    {PXA_MFP_GPIO_LR_GPIO_07_ID,    PXA_MFP_PIN_GPIO120_OFFSET,    PXA_MFP_NOT_DEFINED_VALUE},
    {PXA_MFP_GPIO_LR_GPIO_08_ID,    PXA_MFP_PIN_GPIO121_OFFSET,    PXA_MFP_NOT_DEFINED_VALUE},
    {PXA_MFP_GPIO_LR_GPIO_09_ID,    PXA_MFP_PIN_GPIO122_OFFSET,    PXA_MFP_NOT_DEFINED_VALUE},
    {PXA_MFP_GPIO_LR_GPIO_10_ID,    PXA_MFP_PIN_GPIO123_OFFSET,    PXA_MFP_NOT_DEFINED_VALUE},
```

根據所定義的索引名稱，建立一個想要設定MFP輔助功能群組陣列gaulBtnMFPTable，在這個群組陣列中加入要調整的MFP名稱並更改Af.Code欄位值為PXA_MFP_ALT_FN_0（一般GPIO功能），並依據需求，更改其他相關欄位值。

```
PXA_MFP_ACTIVE_MODE_SETTING_T gaulBtnMFPTable[] =
{
 /* MFP Pin ID */                  /* Af Code */     /* Drive Str*/     /* LPM or PULL*/   /* LPM
 {PXA_MFP_GPIO_BTN1_ID,          PXA_MFP_ALT_FN_0, PXA_MFP_DEFAULT_DS, PXA_MFP_CFG_NONE, PXA_MFP_L
 {PXA_MFP_GPIO_BTN2_ID,          PXA_MFP_ALT_FN_0, PXA_MFP_DEFAULT_DS, PXA_MFP_CFG_NONE, PXA_MFP_L
 {PXA_MFP_GPIO_BTN3_ID,          PXA_MFP_ALT_FN_0, PXA_MFP_DEFAULT_DS, PXA_MFP_CFG_NONE, PXA_MFP_L
 {PXA_MFP_GPIO_BTN4_ID,          PXA_MFP_ALT_FN_0, PXA_MFP_DEFAULT_DS, PXA_MFP_CFG_NONE, PXA_MFP_L
};
```

再在MFP_GetMFPTableAddrByComponentID函式中，加入新增一個群組選項PXA_COMPONENT_BTN_ID，並在選項內加入gaulBtnMFPTable群組陣列的起始位置及大小。

```
case PXA_COMPONENT_BTN_ID:
{
    *pulMFPTableAddr = (UINT32)(&gaulBtnMFPTable[0]);
    *pulMFPTableSize = sizeof(gaulBtnMFPTable)/sizeof(PXA_MFP_ACTIVE_MODE_SETTING_T);
    break;
}
```

完成新增後，在gaulBtnMFPTable群組陣列中的所有MFP接腳，將會在激活MFP時，依照所設定的參數激活成為一般GPIO的功能。

④ 設定GPIO接腳的方向

一般而言，GPIO的控制暫存器的參數設定，多是在驅動程式中的XXX_INIT中設定，但也 可以依照使用環境而在其他驅動程式入口函式中，加入 GPIO 控制設定。使用者要自行設定GPIO的輸出入方向，首先需先取得GPIO控制暫存器在記憶體中的起啟位置。在PXA310 開發平台上，提供了一個PXA_CTX_GetRegAddr()函式，來讓開發者可以呼叫取得想要的暫 存器記憶位置，以PXA310 中按鈕驅動程式為例，在驅動程式原始碼 Button_drv.c 的 BTN_INIT()中，可以看到呼叫PXA_CTX_GetRegAddr(PXA_PERIPHERAL_REGIDX_GPIO)函式， 其中的PXA_PERIPHERAL_REGIDX_GPIO表示GPIO控制暫存器的代碼。函式傳回暫存器的記 憶體位置，用來當作之後控制暫存器設定的記憶體起始位置。

```
BTN_InitializeAddresses(void)
{
    BOOL RetValue = TRUE;
    RETAILMSG(1, (TEXT("BTN_InitalizeAddresses.\r\n")));
    // GET Allocate Resources
    g_pvGpioRegs = (PVOID)PXA_CTX_GetRegAddr(PXA_PERIPHERAL_REGIDX_GPIO);
    if (g_pvGpioRegs == NULL)
    {
        ERRORMSG(1, (TEXT("BTN_InitializeAddresses:Failed to map GPIO registers.\r\n")));
        RetValue = FALSE;
    }
    if (MFP_SetActiveMode(PXA_COMPONENT_BTN_ID) != PXA_STATUS_SUCCESS)
    {
        ERRORMSG(1, (TEXT("BTN_InitializeAddresses:Can't activate BTNs pin.\r\n")));
        RetValue = FALSE;
    }
    return RetValue;
}
```

在取得記憶體位置之後接下來就可以透過 PXA310 平台所提供的 PXA_GPIOSetDirection() 函式來設定 GPIO 輸出入方向。若要設定為輸出方向，則設定方向 參 數 為 PXA_GPIO_DIRECTION_OUT

例如：

PXA_GPIOSetDirection (g_pvGpioRegs,PXA_GPIO_KP_MKOUT_0_ID,PXA_GPIO_DIRECTION_OUT);

若要設定為輸入方向則設定參數為PXA_GPIO_DIRECTION_IN

例如：

PXA_GPIOSetDirection (g_pvGpioRegs,PXA_GPIO_KP_MKOUT_0_ID,PXA_GPIO_DIRECTION_IN);

函式成功完成後，GPIO 的方向將會在激活後，設定為使用者所要求的方向(若沒有特別設定方向，則激活後的GPIO方向預設為輸入)。

④ 激活MFP功能設定

PXA310中的MFP激活程序，其功能簡單來說，就是執行MFP的控制權轉移。激活程序根據 MFP 的輔助功能參數來切換可控制該 MFP 的暫存器。讓 MFP 由正確的功能控制暫存器來控制。以 GPIO 應用實作中，激活程序將使用者設定的 MFP 接腳的控制權交給 GPIO 控制暫存器，讓使用者可以利用 GPIO 控制暫存器來控制相對應的 MFP。所以，激活程序可以在任何時間點執行，但在還沒有執行激活之前，所有 GPIO 控制暫存器內的設定，都將不會影響 MFP狀態。

PXA310 提供了一個 MFP 激活函式 MFP_SetActiveMode()，激活程序將根據在Platform\zylonite_mhlv\src\drivers\mfp\MFP_platform.c檔案中所建立的MFP設定來激活相對應的MFP。以先前的PXA310按鈕為例，在驅動程式中加入MFP_SetActiveMode(PXA_COMPONENT_BTN_ID)來執行激活 MFP。激活程序將根據輸入參數 PXA_COMPONENT_BTN_ID，找到 MFP_GetMFPTableAddrByComponentID 中相對應的 MFP 群組 陣列 gaulBtnMFPTable，並以陣列中所設定的 MFP 參數來激活 MFP。並將陣列中的MFP控制權切換到GPIO控制暫存器。

④ 設定開啟、關閉GPIO上/下緣偵測

當PXA310中的MFP設定為一般GPIO功能後，可以讓使用者依需要，設定作為外部中斷的 訊號輸入。為了達到中斷偵測功能，則需要先開啟GPIO的上/下緣偵測，好讓系統可以在 外部中斷訊號變化時，即時偵測到改變，進而產生系統中斷訊號。PXA310提供了兩個函式來開啟/關閉 GPIO 的上/下緣觸發偵測。PXA_GPIOSetRisingEdgeDetectEnable()、PXA_GPIOSetFallingEdgeDetectEnable()。使用者需在呼叫上述函式時，傳入 GPIO 控制暫存器起始位置、欲設定的 GPIO 名稱、設定值，完成函式呼叫。以 PXA310 的按鈕驅動程式為例(\PLATFORM\BSP2008W41_LR3A205\SRC\DRIVERS\BUTTON\Button_drv.c)：

```
PXA_GPIOSetRisingEdgeDetectEnable(g_pvGpioRegs, PXA_GPIO_BTN1_ID, PXA_ON);
```

其中,g_pvGpioRegs是GPIO控制暫存器的起始位置;PXA_GPIO_BTN1_ID是要設定的GPIO名稱；PXA_ON(PXA_OFF)是設定偵測功能為開啟(關閉)。在 PXA310 中使用 GPIO 作為Interrupt訊號輸入的詳細設定，將在另外的章節中說明。

④ 讀取、寫入GPIO

GPIO 接腳在相關參數設定完成，並成功激活的 GPIO 接腳，就可以透過 PXA310 所提供的 GPIO 讀寫函式來讀取/寫入GPIO。若GPIO方向為輸入，想要讀取GPIO目前狀態，可以呼叫PXA_GPIOGetLevel()函式來達成。例如：

```
PXA_GPIOGetLevel(g_pvGpioRegs,PXA_GPIO_SMEM_nCSXR3_ID,&Level);
```

其中g_pvGpioRegs是GPIO控制暫存器的記憶體起始位置，PXA_GPIO_SMEM_nCSXR3_ID是 想要讀取狀態的GPIO名稱，&Level為GPIO所回傳的目前狀態。若GPIO方向為輸出，想要GPIO輸出某值，可以呼叫PXA_GPIOSetLevel()來達成。例如：

```
PXA_GPIOSetLevel(g_pvGpioRegs,PXA_GPIO_SMEM_nCSXR3_ID,&Level);
```

其中g_pvGpioRegs是GPIO控制暫存器的記憶體起始位置，PXA_GPIO_SMEM_nCSXR3_ID是 想要讀取狀態的GPIO名稱，&Level為想要GPIO輸出的值。

第5章 實驗平台驅動程式開發

- 實驗名稱

OV7670 Camera module

Video Preview

Version 1.0 -

1. Camera Driver 架構

④ Camera MDD Layer
由微軟所提供，符合微軟DirectShow介面規範。

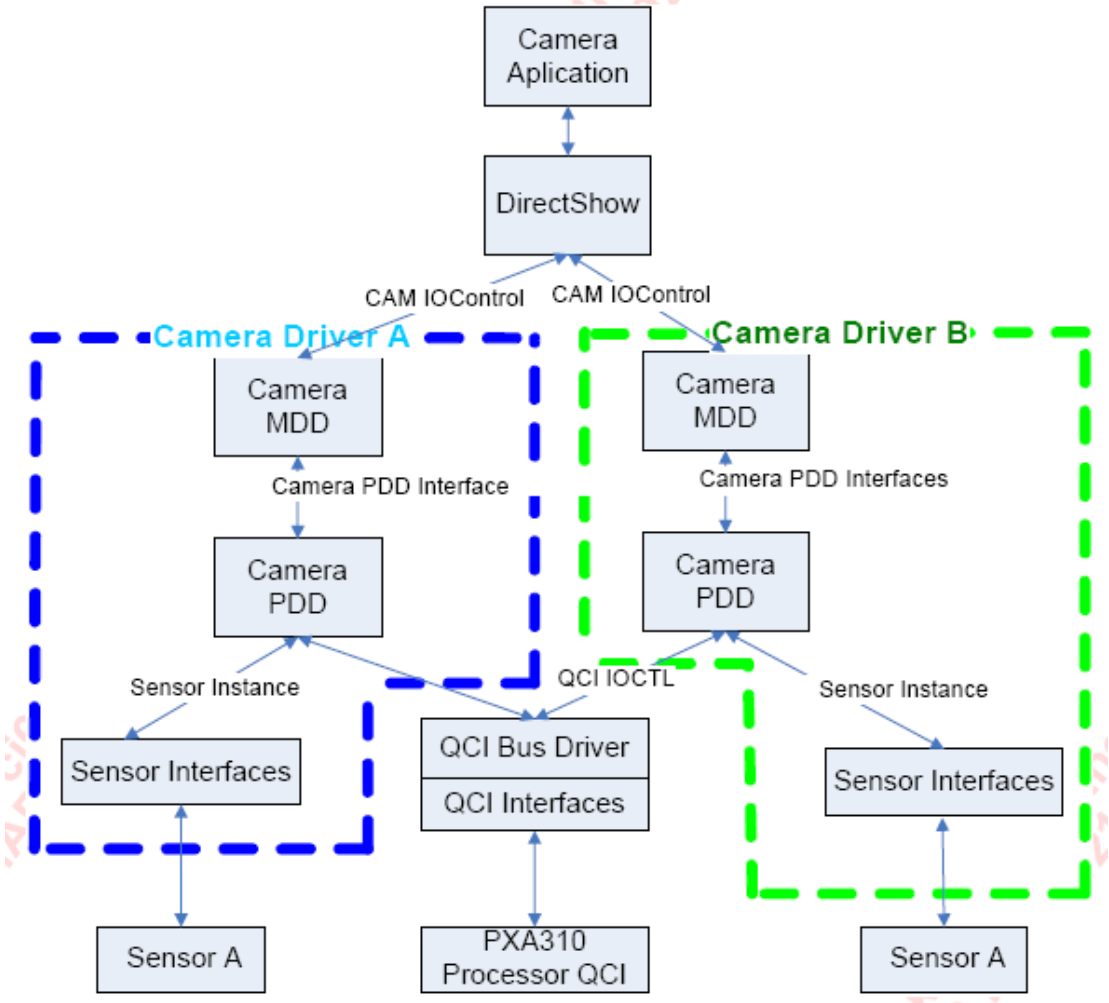
④ Camera PDD Layer
由微軟所提供，符合微軟DirectShow介面規範。

④ Sensor Interface
提供設定sensor組態的方法，例如：format、size、timing等。

④ Quick Capture Interface(QCI) Bus Driver 主要功能是用來支援雙攝影頭。最重要的功能是用來例舉、載入兩個camera的驅動程式， 並且管理QCI硬體。

Marvell PXA310的camera driver架構圖，如下圖1-1所示：

圖1-1 Marvell PXA310 Camera Driver 架構圖 (MV-S301562-00.pdf)



2. Camera Driver 功能

④ 預覽擷取、Video擷取

- 攝影頭輸入color space： YCbCr4:2:2、YCbCr4:2:0、RGB、RAW10 RGB
- 攝影頭輸入size：QCIF、CIF、QVGA、VGA
- DirectShow輸出color space：YCbCr4:2:0、 RGB565
- DirectShow輸出size：QCIF、CIF、QVGA、VGA

④ 靜態影像擷取

- 攝影頭輸入至驅動程式： YCbCr4:2:2、YCbCr4:2:0、RGB、RAW10 RGB
- 驅動程式輸出至DirectShow：VGA、QCIF、YCbCr4:2:0

④ 雙攝影頭

僅支援同時間使用單一攝影頭，支援靜態選擇攝影頭(in Eboot)，不支援動態攝影頭選擇(此版本尚未支援雙攝影頭功能)。

3. Camera Driver 限制

④ 未支援的Video Processing功能：

- CSPROPERTY_VIDEOPROCAMP_BRIGHTNESS
- CSPROPERTY_VIDEOPROCAMP_CONTRAST
- CSPROPERTY_VIDEOPROCAMP_HUE
- CSPROPERTY_VIDEOPROCAMP_SATURATION
- CSPROPERTY_VIDEOPROCAMP_SHARPNESS
- CSPROPERTY_VIDEOPROCAMP_GAMMA
- CSPROPERTY_VIDEOPROCAMP_COLORENABLE
- CSPROPERTY_VIDEOPROCAMP_WHITEBALANCE
- CSPROPERTY_VIDEOPROCAMP_BACKLIGHT_COMPENSATION
- CSPROPERTY_VIDEOPROCAMP_GAIN

④ 未支援的camera控制功能：

- CSPROPERTY_CAMERACONTROL_PAN
- CSPROPERTY_CAMERACONTROL_TILT

- CSPROPERTY_CAMERACONTROL_ROLL
- CSPROPERTY_CAMERACONTROL_ZOOM
- CSPROPERTY_CAMERACONTROL_EXPOSURE
- CSPROPERTY_CAMERACONTROL_IRIS
- CSPROPERTY_CAMERACONTROL_FOCUS
- CSPROPERTY_CAMERACONTROL_FLASH

4. Marvell PXA310 實現 OV7670 預覽範例程式

由於Marvell所提供的camera驅動程式，是針對OV7660及OV2630模組所開發的camera驅動程式。而OV7660和OV7670兩個模組的相關控制暫存器的設定，並不完全相同。因此，為 了能夠正常使用OV7670 模組，必須更改部份OV7660驅動程式內容，才能正確使用OV7670。 以下的OV7670 預覽範例說明，將分為三大部份來說明：第一部份為如何調整 Marvell 所提供的 OV7660 模組驅動程式內容，使之可以使用 OV7670 模組。第二部份為微軟 DirectShow 的基礎介紹。第三部份為如何在wince 6上利用DirectShow來實現camera模組的預覽程式。

4.1修改OV7660模組驅動程式內容：

在進行驅動程式修改之前，因為OV7670的接腳中的PWDN和RESET在1A01V02電路板上接 腳有所更動，所以需要先在 MFP 組態設定的列表中，加入此兩腳的硬體接腳。修正資訊如下：

修改檔案：

\PLATFORM\BSP_ROOT\SRC\DRIVERS\MFP\MFP_Platform.c

修改位置：

gaulCIFMFPTable[ ]

修改內容：

加入

```
{PXA_MFP_GPIO_CIF_PWDN_ID, PXA_MFP_ALT_FN_0, PXA_MFP_DS_04X,
    PXA_MFP_CFG_NONE, PXA_MFP_NOT_DEFINED_VALUE, PXA_MFP_PULL_NEITHER,
    PXA_OFF, PXA_MFP_NEITHER_EDGE, PXA_OFF},
{PXA_MFP_GPIO_CIF_RESET_ID, PXA_MFP_ALT_FN_0, PXA_MFP_DS_04X,
    PXA_MFP_CFG_NONE, PXA_MFP_NOT_DEFINED_VALUE, PXA_MFP_PULL_NEITHER,
    PXA_OFF, PXA_MFP_NEITHER_EDGE, PXA_OFF},
```

4.1.1 修改模組reset流程，以符合OV7670 RESET程序

因OV7670的reset為LOW，調整reset函式中的reset順序為HI、LOW、HI，完成模組的reset動作。

修改檔案：

\PLATFORM\BSP_ROOT\SRC\DRIVERS\camera\Sensor\ov7660_hw.c

修改位置：

OV7660Reset( )

修改內容：

```
PXA_GPIOSetLevel(gpio_regs, PXA_MFP_GPIO_CIF_RESET_ID, PXA_HI);
PXA_GPIOSetLevel(gpio_regs,PXA_MFP_GPIO_CIF_RESET_ID,  PXA_LO);
PXA_OST_DelayMilliSeconds(10);
PXA_GPIOSetLevel(gpio_regs, PXA_MFP_GPIO_CIF_RESET_ID, PXA_HI);
```

4.1.2增加Reset OV7670模組函式至初始化過程

原始驅動程式在模組初始化過程中,並沒有先呼叫reset函式來重置OV7670模組,將reset硬體接腳電壓準位設定至Normal的準位，造成OV7670模組reset準位錯誤，模組一直處於reset狀態，驅動程式無法偵側到設備。因此我們需要在系統執行模組初始化過程中， 多加入呼叫OV7670模組reset函式，讓模組可以先行執行reset流程再完成初始化。

修改檔案：

\PLATFORM\BSP_ROOT\SRC\DRIVERS\camera\Sensor\ov7660.cpp

修改位置：

Ov7660::Initialize()

修改內容：

加入OV7660Reset();

4.1.3 Replace camera模組暫存器最大位置

OV7660 模組所擁有的最大有效暫存器位置為 0XA3，而 OV7670 模組所擁有的最大有效暫存器位置為 0XC9。在驅動程式中，會根據模組最大暫存器的位置+1 來作為讀寫暫存器及其他相關驅動流程的結束判斷條件，所以為了可以讓 OV7670 後段的暫存器(0XA3~0XC9)可以正確被讀寫、驅動流程可以正常判斷，我們必須將原有的暫存器最大值，調整為 OV7670 暫存器的最大位置0XC9。

修改檔案：

A: \PLATFORM\BSP_ROOT\SRC\DRIVERS\camera\Sensor\ov7660_hw.h

B: \PLATFORM\BSP_ROOT\SRC\DRIVERS\camera\Sensor\ov7660_hw.c

修改位置：

A: 參數定義

B: 全文件

修改內容：

A: 新增參數 #define OV7670_REGEND (0xc9 + 1)

B: Replace 所有的OV7660_REGEND成為OV7670_REGEND

4.1.4 修改camera模組載入暫存器參數

由於Marvell所提供的BSP中的驅動程式，是針對OV7660及OV2630所撰寫的驅動程式， 所以驅動程式中的模組暫存器載入值都是以 OV7660 或 OV2630 所設立的。換成 OV7670 模組後，部份暫存器的位置、功能及數值和 OV7660 不同，所以我們需要進一步調整所載入的參數值，以達到正確的模組控制。在開啟、使用 camera 模組時，驅動程式會呼叫一連串的函式來建立應用程式及模組間的通道，並根據使用者所選擇的功能來載入預先建立的模組控制暫存器參數，以影像預覽功能為例，驅動程式將會載入 QCIF、YUV 的相關暫存器參數,gYUV_QCIF_15fps_INITIALIZE[],所以我們就需要修改此組參數,使之符合 OV7670。 模組暫存器的詳細資訊，請參考OV7670相關DATASHEET。

修改檔案：

B: \PLATFORM\BSP_ROOT\SRC\DRIVERS\camera\Sensor\ov7660_hw.c

修改位置：

gYUV_QCIF_15fps_INITIALIZE[]

修改內容：

```
0x11, 0x40, // PCLK=XCLK old
0x3A, 0x0c, // TSLB disable Auto output window
0x3D, 0xc1, // Gamma enable, UV auto adjustment, VYUY
0x12, 0x08, // 00:YUV VGA  02:colorbar
0x09, 0x03, // output drive capability
0x17, 0x13, // HSTART
0x18, 0x90, // HSTOP 0x01
0x03, 0x00, // VREF 0x0f
0x0C, 0x00, // COM3
0x3E, 0x00, // COM14
```

```
0x70, 0x3A, // Scaling_XSC default
0x71, 0x35, // Scaling_YSC default
0x72, 0x11, // default
0x73, 0xF0, // Scaling_PCLK_DIV
0xA2, 0x3B, // Scaling_PCLK_delay
0x1E, 0x17, //flip, Black sun enable
// Gamma curve for Capture mode
0x7a, 0x20, // Gamma Curve Highest Segment Slop
0x7b, 0x03, // GAM
0x7c, 0x0a, // GAM
0x7d, 0x1a, // GAM
0x7e, 0x3f, // GAM
0x7f, 0x4e, // GAM
0x80, 0x5b, // GAM
0x81, 0x68, // GAM
0x82, 0x75, // GAM
0x83, 0x7f, // GAM
0x84, 0x89, // GAM
0x85, 0x9a, // GAM
0x86, 0xa6, // GAM
0x87, 0xbd, // GAM
0x88, 0xd3, // GAM
0x89, 0xe8, // GAM
0x55, 0x80, //brightness offset
//AEC/AGC range
0x13, 0xE0,
0x00, 0x00, // GAIN
0x10, 0x00, // AECH
0x0D, 0x50,// COM4 1/2 window for AEC 1/22/2007
0x42, 0x40,// COM17 1/2 window for AEC 1/22/2007
0x14, 0x28,// max 8x gain
0xA5, 0x03, // 50Hz Banding Step Limit
0xAB, 0x03, // 60Hz Banding Step Limit
0x24, 0x50, // AEW decrease AE target  1/22/2007
0x25, 0x43, // AEB decrease AE target  1/22/2007
0x26, 0xa3, // VPT AGC/AEC Fast Mode Operationg Region
```

```
0x9F, 0x78, // HAECC1
0xA0, 0x68, // HAECC2
0xA1, 0x03, // Reserved
0xA6, 0xd2, // HAECC3
0xA7, 0xd2, // HAECC4
0xA8, 0xF0, // HAECC5
0xA9, 0x80, // HAECC6
0xAA, 0x14, // HAECC7, average AEC/AGC mode
0x13, 0xE5, // fast AGC/AEC, Unlimit step size, AGC,AEC
0x0E, 0x61, // COM5, Reserved
0x0F, 0x4B, // COM6
0x16, 0x02, // Reserved
0x21, 0x02, // Reserved
0x22, 0x91, // Reserved
0x29, 0x07, // Reserved
0x33, 0x0B, // Array Current control
0x35, 0x0B, // Reserved
0x37, 0x1D, // ADC control
0x38, 0x71, // ADC analog common mode control
0x39, 0x2A, // ADC offset control
0x3C, 0x78, // COM12, Always has HREF
0x4D, 0x40, // Reserved
0x4E, 0x20, // Reserved
0x69, 0x00, // Fix Gain Control, default
0x6B, 0x0A, // PLL control , default
0x74, 0x10, // Bypass Digital gain manual control
0x8D, 0x4F, // Reserved
0x8E, 0x00, // Reserved
0x8F, 0x00, // Reserved
0x90, 0x00, // Reserved
0x91, 0x00, // Reserved
0x96, 0x00, // Reserved
0x9A, 0x80, // Reserved
0xB0, 0x84, // Reserved
0xB1, 0x0C, // Enable ABLC function
0xB2, 0x0E, // Reserved
```

```
0xB3, 0x82, // ABLC Target
0xB8, 0x0A, // Reserved
//AWB
0x43, 0x2 , // Reserved
0x44, 0xf2, // Reserved
0x45, 0x46, // Reserved
0x46, 0x63, // Reserved
0x47, 0x32, // Reserved
0x48, 0x3b, // Reserved
0x59, 0x92, // AWB Control
0x5a, 0x9b, // AWB Control
0x5b, 0xa5, // AWB Control
0x5c, 0x7a, // AWB Control
0x5d, 0x4a, // AWB Control
0x5e, 0xa , // AWB Control
0x6c, 0xa , // AWB Control3
0x6d, 0x55, // AWB Control2
0x6e, 0x11, // AWB Control1
0x6f, 0x9e, // AWB Control0,Advance AWB mode
0x6A, 0x40, // G channel AWB Gain
0x01, 0x40, // AWB-Blue channel gain setting
0x02, 0x40, // AWB-Red channel gain setting
0x13, 0xf7, // fast AGC/AEC
//color matrix
0x4f, 0x9c, // Matrix coef.
0x50, 0x99, // Matrix coef.
0x51, 0x2 , // Matrix coef.
0x52, 0x29, // Matrix coef.
0x53, 0x8b, // Matrix coef.
0x54, 0xb5, // Matrix coef.
0x58, 0x1e, // Matrix coef. sign +
//Lens shading correction
0x62, 0x08, // LCC1
0x63, 0x10, // LCC2
0x64, 0x04, // LCC3
0x65, 0x00, // LCC4
```

```
0x66, 0x05, // LCC5
0x94, 0x04, // LCC6
0x95, 0x06, // LCC7
0x41, 0x08, // COM16, AWB gain enable, De-noise auto-adj
0x3F, 0x00, // Edge enhancement factor
0x75, 0x44, //Edge enhancement lower limit
0x76, 0xe1,
0x4C, 0x00, // De-noise Strength
0x77, 0x01, // De-noise offset
0x3D, 0xC3, //c3:VYUY format C0:UYVY
0x4B, 0x09, // UV average enable
0xC9, 0x60, // saturation control
0x41, 0x38,
0x56, 0x40, // contrast control
0x34, 0x11, // array reference control
0x3b, 0x02, // exposure timing can be less than limit
0x92, 0x00, // dummy line low 8bit
//0xa4, 0x88, // reduce frame rate by half
0x96, 0x00, // Reserved
0x97, 0x30, // Reserved
0x98, 0x20, // Reserved
0x99, 0x20, // Reserved
0x9A, 0x84, // Reserved
0x9B, 0x29, // Reserved
0x9C, 0x03, // Reserved
0x9D, 0x99, // 50Hz value
0x9E, 0x7F, // 60Hz value
0x78, 0x00 , // Reserved
0x79, 0x01 , // Reserved
0xc8, 0xf0 , // Reserved
0x79, 0x0f , // Reserved
0xc8, 0x00 , // Reserved
0x79, 0x10 , // Reserved
0xc8, 0x7e , // Reserved
0x79, 0x0a , // Reserved
0xc8, 0x80 , // Reserved
```

```
0x79, 0x0b , // Reserved
0xc8, 0x01 , // Reserved
0x79, 0x0c , // Reserved
0xc8, 0x0f , // Reserved
0x79, 0x0d , // Reserved
0xc8, 0x20 , // Reserved
0x79, 0x09 , // Reserved
0xc8, 0x80 , // Reserved
0x79, 0x02 , // Reserved
0xc8, 0xc0 , // Reserved
0x79, 0x03 , // Reserved
0xc8, 0x40 , // Reserved
0x79, 0x05 , // Reserved
0xc8, 0x30 , // Reserved
0x79, 0x26 , // Reserved
0x2b, 0x62, // Dummy pixel insert LSB, for 27MHz
0x40, 0x00, // output range 0x10~0xf0
OV7670_REGEND, 0x00
```

4.2 DirectShow簡介

DirectShow（有時縮寫如 DS 或 DShow）開發代號 Quartz ，是一種由微軟公司開發的能夠讓軟體開發者對媒體文件執行各種不同處理的應用程式設計接口(API)。DirectShow 是以微軟 Windows 構成物件模型(COM)框架為基礎架構，DirectShow 為大部份微軟程式 設計語言提供了一個媒體的標準介面，而且是一個可擴展的介面，能在使用者或開發者的 命令下播放或記錄媒體文件，DirectShow 的基本組成單位是 Filter。DirectShow 開發工 具及憑證被加入到微軟 SDK 平台的一部份。例如 Windows Media Player 就是運用了 DirectShow 及各種衍生出來的工具來播放來自儲存設備中文件或是網路上的多媒體內容。 DirectShow 運作的方式通常是由程式開發者創建一個 Filter Graph，再把一些 Filter 加入至 Filter Graph，然後播放文件，或者播放來自網路或照相機的數據。當播放程式執 行時，Filter Graph 會在 Windows 註冊表中尋找已註冊的 Filters 並且為這些 Filter 創建本地提供的 Graph 。接著，再將所有的 Filter 連接在一起，並且在開發者的請求下， 播放／中止創建的Graph。

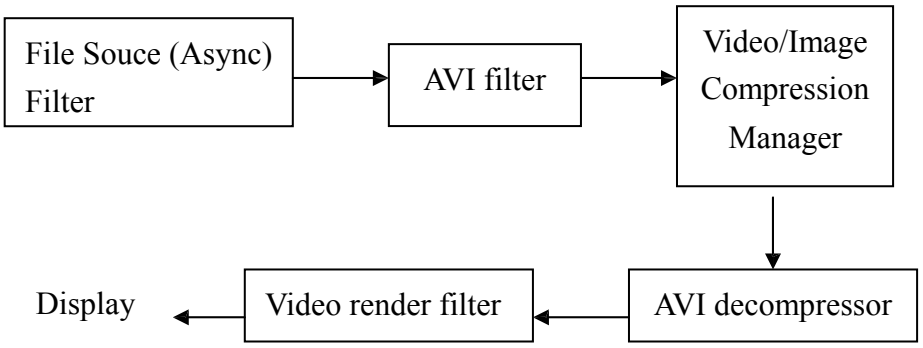
DirectShow 預先設置支持常見的媒體格式，如 MP3、ASF、MPEG、AVI、WAV、WMA及WMV。

4.2.1 DirectShow Filters

Filter 是 DirectShow 的組成基本元件。DirectShow 預設一些基本的 filter 提供給開發者使用。DirectShow 利用不同的 filter 來執行不同的動作，如解碼、格式化、渲染媒體串流資料，而每一個 filter 的輸出端都可以直接連結到另一個或多個 filter 的輸入端。開發者可以自由的組合各個 filter，來完成所需的媒體應用。由多個 filter 所組成的 filter 群組，稱為一個 Filter Graph。不同種類的 Filter 就負責不同步驟的處理。例如，File Source (Async) Filter 是可以從磁碟中讀取檔案。DirectShow 中的 Filter 可以分為三大類，Source Filters、Transform Filters及Renderer Filters。所有DirectShow SDK所提供的filter請參考附錄A。

4.2.2 DirectShow Filter Graph

Filter Graph的建立，可以透過應用程式的手動建立，或是透過Graph Builder或Filter Graph Manager自動建立。例如一個簡單的avi影音檔撥放graph如下圖所示。



4.2.3 Capture Graph Builder

Capture Graph Builder 不只可以用來建立 capture graph，也可以用來建立多種客製化的filter graph。要使用Graph Builder前需先完成幾項建立builder的動作。首先，呼叫 CoCreateInstance() 函式來建立一個 Graph Builder 和 Filter Graph Manager，成功建立後，接著就可以透過呼叫 ICaptureGraphBuilder2::SetFiltergraph 函式來初始化所建立的 Capture Graph Builder。使用 Capture Graph Builder 建立 filter graph 的範例程式碼如下所示(MSDN提供)：

```
IGraphBuilder *pGraph = NULL;
ICaptureGraphBuilder2 *pBuilder = NULL;
// Create the Filter Graph Manager.
```

```
HRESULT hr =    CoCreateInstance(CLSID_FilterGraph, NULL,
      CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&pGraph);
if (SUCCEEDED(hr))
{
      // Create the Capture Graph Builder.
      hr = CoCreateInstance(CLSID_CaptureGraphBuilder2, NULL,
            CLSCTX_INPROC_SERVER, IID_ICaptureGraphBuilder2,
            (void **)&pBuilder);
     if (SUCCEEDED(hr))
      {
          pBuilder->SetFiltergraph(pGraph);
      }
};
```

ICaptureGraphBuilder2是一個用來建立capture graph或其他客製化filter graph的功 能介面，使用者可以利用這個介面來定義 filter graph (SetFiltergraph)、尋找特定的graph介面(FindInterface)、連接輸出至渲染filter(RenderStream)等graph相關功能。

4.2.4 如何使用DirectShow撥放影音檔案

要使用DirectShow來完成一個簡單的撥放影音檔案,只需要少數幾行程式碼就可以完成。DirectShow的應用程式一般可以分為以下幾個步驟來完成：

1、建立一個Filter Graph Manager的instance。

2、利用Filter Graph Manager來建立一個Filter Graph。

3、運行Graph，將媒體資料導入filter處理。

以下為簡單的撥放檔案範例程式(MSDN提供)

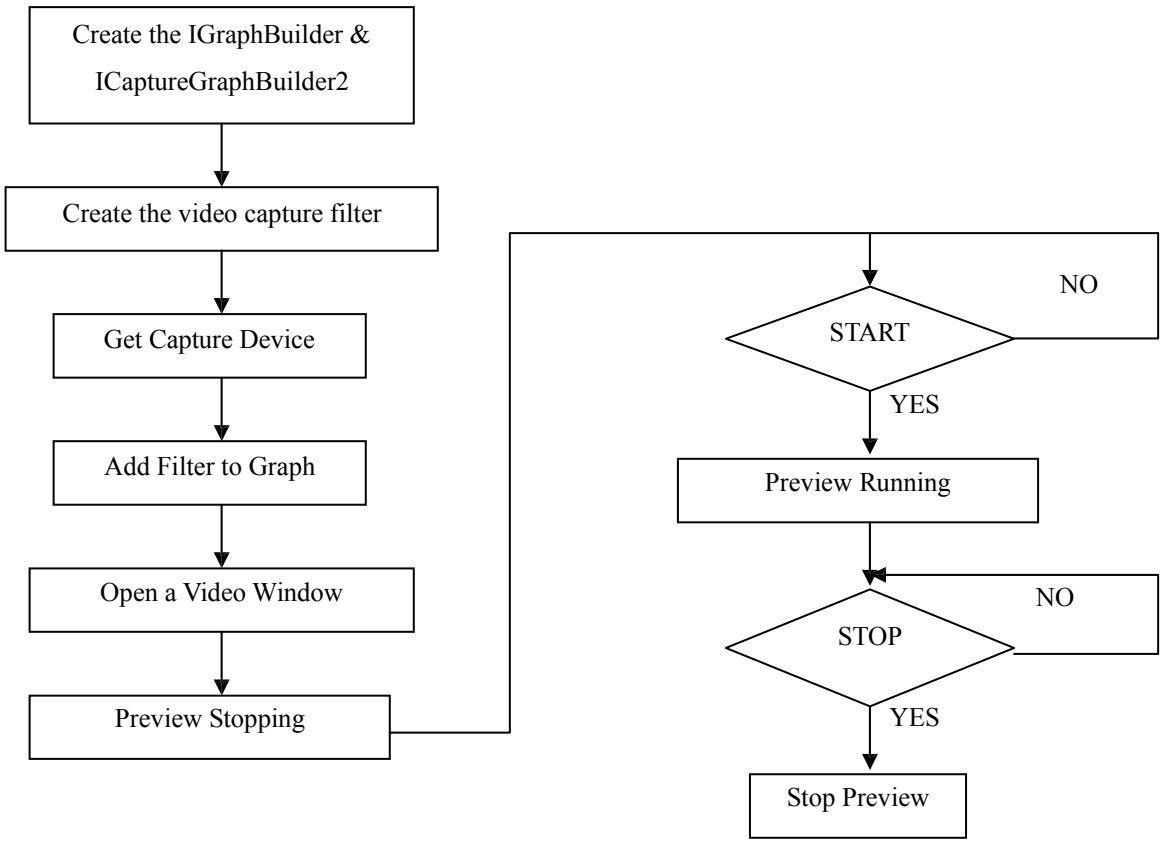
```
#include <dshow.h>
void main(void)
{
    IGraphBuilder *pGraph = NULL;
    IMediaControl *pControl = NULL;
    IMediaEvent     *pEvent = NULL;
    // Initialize the COM library.
    HRESULT hr = CoInitialize(NULL);
    // Create the filter graph manager and query for interfaces.
    hr = CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
```

```
            IID_IGraphBuilder, (void **)&pGraph);
    hr = pGraph->QueryInterface(IID_IMediaControl, (void **)&pControl);
    hr = pGraph->QueryInterface(IID_IMediaEvent, (void **)&pEvent);
    // Build the graph. IMPORTANT: Change this string to a file on your system.
    hr = pGraph->RenderFile(L"C:\\Example.avi", NULL);
    if (SUCCEEDED(hr))
    {
        // Run the graph.
        hr = pControl->Run();
        if (SUCCEEDED(hr))
        {
            // Wait for completion.
            long evCode;
            pEvent->WaitForCompletion(INFINITE, &evCode);
            // Note: Do not use INFINITE in a real application, because it
            // can block indefinitely.
        }
    }
    pControl->Release();
    pEvent->Release();
    pGraph->Release();
    CoUninitialize();
}
```

其中IMediaControl是filter graph所提供的一個媒體控制介面，IMediaControl介面可以允許應用程式控制通過filter的串流媒體資料。例如running、pausing、stopping等。

4.3 DirectShow實現Marvell PXA310的camera即時預覽功能

流程圖如下所示：



4.3.1 建立IGraphBuilder及ICaptureGraphBuilder2

```
HRESULT CreateBuilder()
{
HRESULT dwErr = S_OK;
//--------------------------------------------------------------------
// (1)Create the IGraphBuilder & ICaptureGraphBuilder2
//--------------------------------------------------------------------
dwErr = CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
IID_IGraphBuilder, (void **)&pGraph);
if (dwErr != NOERROR)
    RETAILMSG(1, (TEXT("Error---Cannot instantiate filtergraph!")));
dwErr = CoCreateInstance(CLSID_CaptureGraphBuilder , NULL,
CLSCTX_INPROC_SERVER,IID_ICaptureGraphBuilder2, (void **)&pBuild);
if (dwErr != NOERROR)
    RETAILMSG(1, (TEXT("Error---Cannot instantiate filtergraph2!")));
//--------------------------------------------------------------------
// (2) specifies which filter graph the capture graph builder will use.
//--------------------------------------------------------------------
dwErr = pBuild->SetFiltergraph(pGraph);
if(dwErr != S_OK)
    RETAILMSG(1, (TEXT("Error---SetFiltergraph Faild !")));
// Create the video window interface
pGraph->QueryInterface(IID_IVideoWindow, (void **)&pVidWin);
// Create a Media Control interface
pGraph->QueryInterface(IID_IMediaControl, (void **)&pMediaControl);
return dwErr;
}
```

4.3.2 建立video capture filter及初始化filter

```
HRESULT InitFilter()
{
HRESULT dwErr = S_OK;
dwErr = CoCreateInstance(CLSID_VideoCapture, NULL, CLSCTX_INPROC_SERVER,
IID_IBaseFilter, (void**)&pVCap); // Create a new video capture filter
```

```
if (dwErr != NOERROR)
    RETAILMSG(1, (TEXT("Error---Cannot CLSID_VideoCapture!")));
dwErr = pVCap->QueryInterface(&m_pVideoControl);
if (dwErr != NOERROR)
    RETAILMSG(1, (TEXT("Error---pVCap->QueryInterface Fail ==> m_pVideoControl!")));
dwErr = pVCap->QueryInterface(&m_pCameraControl);
if (dwErr != NOERROR)
    RETAILMSG(1, (TEXT("Error---pVCap->QueryInterface Fail ==>
    m_pCameraControl!")));
return dwErr;
}
```

4.3.3 取得擷取設備資訊

```
HRESULT GetCaptureDevice()
{
HRESULT dwErr = S_OK;
HANDLE handle = NULL;
DEVMGR_DEVICE_INFORMATION di;
VARIANT varCamName;
IPropertyBag *PropBag;
IPersistPropertyBag *pPropertyBag;
dwErr = pVCap->QueryInterface( &pPropertyBag );
if(dwErr != S_OK)
    RETAILMSG(1, (TEXT("Error QueryInterface-pPropertyBag Fail!")));
GUID guidCamera = { 0xCB998A05, 0x122C, 0x4166, 0x84, 0x6A, 0x93, 0x3E, 0x4D, 0x7E,
0x3C, 0x86 };
di.dwSize = sizeof(di);
// Find the first Camera Device
handle = FindFirstDevice( DeviceSearchByGuid, &guidCamera, &di);
if(( handle == NULL ) || ( di.hDevice == NULL ))
    RETAILMSG(1, (TEXT("Error---Cannot FindFirstDevice!")));
FindClose(handle);
//----------------------------------------
varCamName.bstrVal = di.szLegacyName; // the Camera device id, ex: CAM1:
dwErr = PropBag->Write( L"VCapName", &varCamName);
```

```
if(dwErr != S_OK)
    RETAILMSG(1, (TEXT("Error PropBag->Write Fail !")));
dwErr = pPropertyBag->Load(NULL, NULL);
if(dwErr !=S_OK)
    RETAILMSG(1, (TEXT("Error pPropertyBag->Load Fail!")));
return dwErr;
}
```

4.3.4 將Filter加入至Graph

```
HRESULT AddFilterToGraph()
{
HRESULT dwErr = S_OK;
if(pVCap == NULL)
    RETAILMSG(1, (TEXT("Error pVCap == NULL----------!")));
// Add filter to Graph, named Video Capture Filter
dwErr = pGraph->AddFilter(pVCap, L"Video Capture Filter");
if (dwErr != NOERROR)
    RETAILMSG(1, (TEXT("Error---Cannot AddFilter to Graph Builder!")));
//Link to Capture Filter's Preview Pin
dwErr = pBuild->RenderStream(&PIN_CATEGORY_PREVIEW, &MEDIATYPE_Video,
pVCap, NULL, NULL);
if(dwErr != S_OK)
        RETAILMSG(1, (TEXT("Error --RenderStream Fail")));
return dwErr;
}
```

4.3.5 開啟一個影像視窗

```
RECT grc;
pVidWin->put_Owner((OAHWND)g_hwnd);       // We own the window now
// be a child window
GetClientRect(g_hwnd, &grc);
pVidWin->put_WindowStyle(WS_CHILD | WS_CLIPSIBLINGS);
pVidWin->SetWindowPosition(20, 50, 176, 144); // Set window size and position
```

4.3.6 啟動及停止影像預覽功能

```
HRESULT StartPreview()
{
HRESULT dwErr = S_OK;
dwErr = pMediaControl->Run(); // starting video preview
return dwErr;
}

HRESULT StopPreview()
{
HRESULT dwErr = S_OK;
dwErr = pMediaControl->Stop(); // stopping video preview
return dwErr;
}
```

附錄A

DirectShow Filters

Filter	Description
ACM Wrapper	Enables Audio Compression Manager (ACM) codecs to join a filter graph.
Analog Video Crossbar	Represents a video crossbar on a video capture device that supports the Windows Driver Model (WDM).
Audio Capture	Represents an audio capture device.
Audio Renderer (WaveOut)	Uses the waveOut* APIs to render waveform audio.
AVI Compressor	Enables Video Compression Manager (VCM) compressors to join a filter graph.
AVI Decompressor	Enables Video Compression Manager (VCM) decompressors to join a filter graph.
AVI Draw	Pulled automatically into a playback graph instead of the AVI Decompressor when video is being output to an external NTSC television monitor.
AVI Mux	Accepts multiple input streams and interleaves them into AVI format.
AVI Splitter	Splits audio and video streams in playback of AVI files.
AVI/WAV File Source	(Deprecated.) Reads AVI and WAV source files and generates the appropriate output pins for the file type.
CC Decoder	Accepts sample waveforms delivered by a capture filter and delivers decoded closed-captioning data.
Color Space Converter	Converts from one RGB color type to another RGB type.
DirectSound Renderer	Renders audio using the Microsoft® DirectSound® API.
DMO Wrapper	Enables a DirectShow application to use a Microsoft® DirectX® Media Object (DMO) within a filter graph.

DV Muxer	Combines a digital video (DV)-encoded video stream with one or two audio streams to produce an interleaved DV stream.
DV Splitter	Splits an interleaved digital video (DV) stream into its component video and audio streams.
DV Video Decoder	Decodes a digital video (DV) stream into uncompressed video.
DV Video Encoder	Encodes an uncompressed video stream into digital video (DV).
DVD Navigator	Opens all necessary files in a DVD-Video volume, navigates through the linear DVD-Video .vob files, and parses the resulting MPEG-2 program stream.
Enhanced Video Renderer	Video renderer with the same core functionality and plug-in model as the Media Foundation EVR media sink.
File Source (Async)	Opens and reads local files of many different data formats and passes the data to a parser filter.
File Source (URL)	Works with any source file that can be identified by a Uniform Resource Locator (URL) and whose media major type is stream.
File Stream Renderer	Renders file names that are parsed by the Multi-File Parser filter.
File Writer	Used to write files to disc regardless of format.
Full Screen Renderer	Uses Microsoft® DirectDraw® to render full-screen video on older graphics cards.
Infinite Pin Tee	Delivers samples delivered to its input pin to a variable number of output pins.
Internal Script Command Renderer	Receives script commands and dispatches them to the application.
Line 21 Decoder	Converts line-21 closed caption information to bitmaps with caption text.
Microsoft MPEG-1/DD Audio Decoder	Decodes MPEG-1, MPEG-2, and Dolby Digital audio.
Microsoft MPEG-2 Audio Encoder	Encodes MPEG-2 audio.
Microsoft MPEG-2 Encoder	Encodes MPEG-2 audio and video.
Microsoft MPEG-2 Video Decoder	Decodes MPEG-2 video.
Microsoft MPEG-2 Video Encoder	Encodes MPEG-2 video.

MIDI Parser	Reads MIDI data that is found in .MID and .RMI files.
MIDI Renderer	Renders MIDI data from the MIDI Parser filter.
MJPEG Compressor	Compresses an uncompressed video stream, using motion JPEG compression.
MJPEG Decompressor	Decodes a video stream from motion JPEG to uncompressed video.
MPEG-1 Audio Decoder	Decodes MPEG-1 Layer I and Layer II audio to PCM.
MPEG-1 Stream Splitter	Splits an MPEG-1 system stream into its component audio and video streams.
MPEG-1 Video Decoder	Decodes MPEG-1 video.
MPEG-2 Demultiplexer	Demultiplexes MPEG-2 transport streams that are delivered in push mode, and program streams that are delivered in push or pull mode.
MPEG-2 Splitter	Parses MPEG-2 program streams, creates an output pin for each stream, and outputs the compressed audio and/or video MPEG packets to an MPEG-2 decoder filter.
MSDV Driver	The Microsoft® Windows® Driver Model (WDM) driver for DV camcorders.
MSTape Driver	Supports D-VHS and MPEG camcorder devices.
MSYUV Color Space Converter Codec	Enables playback of video source data in YUV formats on clients whose video display adapter cannot be used for YUV-to-RGB conversions in hardware.
Multi-File Parser	Parses a simple file format that enables multiple file names to be specified as though they were one file.
Null Renderer	Discards every sample it receives, without displaying or rendering the sample data.
Overlay Mixer 2	Like the Overlay Mixer, but can be added to a filter graph automatically.
Overlay Mixer	Designed specifically for DVD playback and broadcast video streams with line-21 closed captioning. (Superseded by Video Mixing Renderer on Windows XP.)

QT Decompressor	Decompresses Apple® QuickTime® 2.0 video.
QuickTime Movie Parser	Splits Apple® QuickTime® data into audio and video streams.
SAMI (CC) Parser	Parses captioning data from Synchronized Accessible Media Interchange (SAMI) files.
Sample Grabber	Provides a way to retrieve samples as they pass through the filter graph.
Smart Tee	Used in video capture graphs to split the video stream into a preview stream and a capture stream.
Tee/Sink-to-Sink Converter	Provides an efficient means to duplicate streams of data within kernel mode without the expensive transitions between kernel and user mode.
TV Audio	Provides control of television audio decoding, stereo or monoaural selection, and secondary audio program (SAP) selection.
TV Tuner	Selects an analog broadcast or cable channel to be viewed.
VBI Surface Allocator	Controls the allocation of VBI buffers in analog television graphs with hardware video port capture scenarios.
VFW Capture Filter	Works with older video capture hardware that uses Video For Windows.
VGA 16 Color Ditherer	Converts from an RGB color type to a 4-bit color display so that AVI and MPEG video streams may be displayed on older 16-color monitors.
Video Mixing Renderer Filter 7 (VMR-7)	The default video renderer in Windows XP. Offers advanced rendering and video mixing capabilities.
Video Mixing Renderer Filter 9 (VMR-9)	Similar to VMR-7 but available on all platforms supported by DirectX.
Video Port Manager	Enables the Video Mixing Renderer to work seamlessly on systems where video data is transferred directly from a video capture device or hardware decoder to the graphics chip.
Video Renderer	Default video renderer on Windows 98SE, Windows 2000, and Windows Millennium Edition. Connects to any video transform filter that produces

	decompressed video data.
WAVE Parser	Parses WAV-format audio data from .wav, .au, or .aif files.
WDM Video Capture	Controls analog capture devices that use Windows Driver Model (WDM) drivers.
Windows Media Source Filter	Default source filter for playback of Windows Media and MPEG-4 content created using the Microsoft MPEG-4 Encoder. This is the source filter used by Windows Media™ Player 6.4.
WM ASF Reader	Source filter for file playback of Windows Media-based content and content created with any of the Microsoft MPEG-4 Encoder DMOs. Must be explicitly added to a filter graph. This filter is based on the Windows Media Format SDK.
WM ASF Writer	Accepts uncompressed input streams and creates ASF files containing either Windows Media streams or MPEG-4 streams using the Microsoft MPEG-4 Encoder DMO. This filter is based on the Windows Media Format SDK.
WST Codec	Decodes and/or duplicates the decoded and forward-error-corrected Teletext data for the WST Decoder filter.
WST Decoder	Accepts decoded World Standard Teletext data from the WST Codec and delivers the bitmaps to Pin 2 on the Overlay Mixer using fonts supplied by Microsoft.

第6章 實驗平臺應用程式開發

應用程式開發實驗

應用程式開發實驗

實驗一

動畫特效

Rev. 1.0

一. 實驗目的：

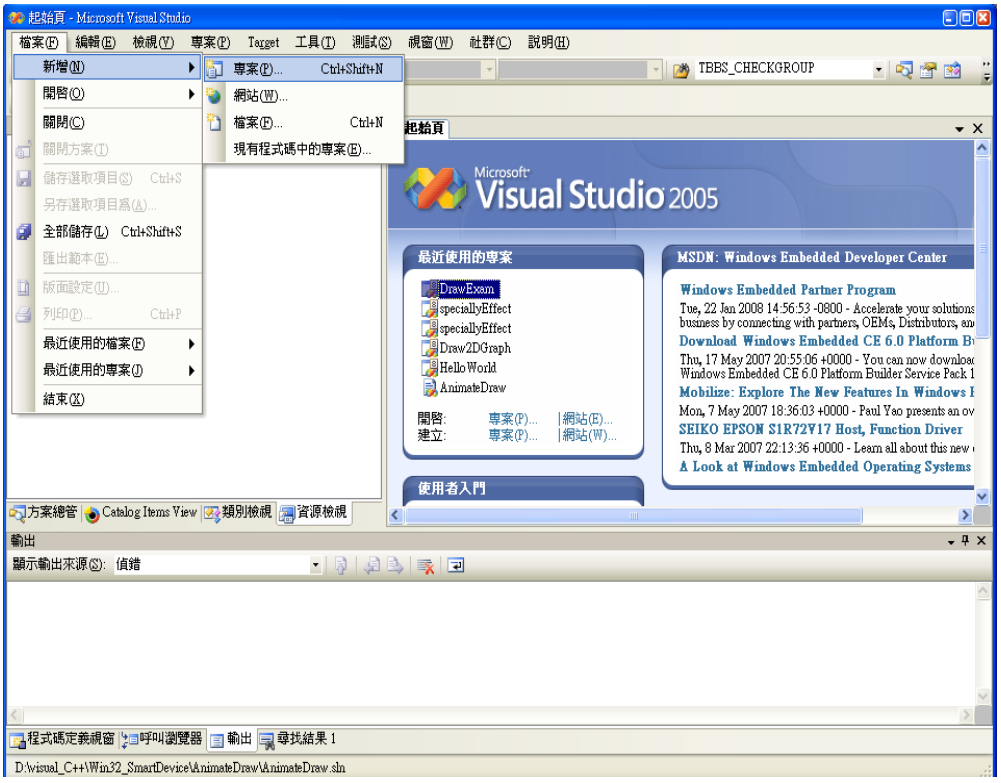
了解程式取得的視窗區域**DC**（**CPaintDC**）物件提供的一些繪圖函數，來 實現動畫效果。 實驗前先準備一張位元圖（**bitmap**）此張位元圖為一連串動作的分解。以 此範例來說：此位元圖是由**4** 個人所組成。

能夠形成動畫的效果，主要是將一張圖片分解成四幅圖片；然後再利用提高顯示效率的方法將圖片繪出，而提高顯示效率的方法：主要是先將所要繪製的圖形繪製到一個臨時的繪圖環境中，然後再將臨時的繪圖環境繪製到螢幕上。如此可以確保顯示時不會出現屏幕閃爍的繪圖。

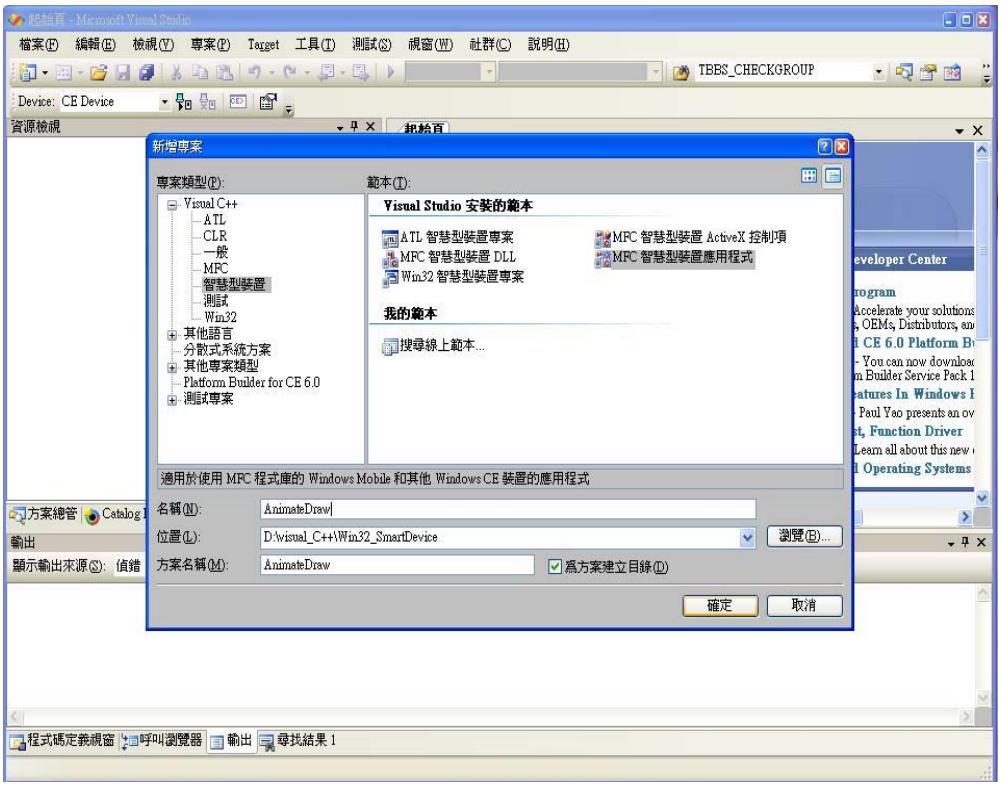
二. 實驗步驟：

1. 建立一個 **MFC** 對話方塊為基礎的專案

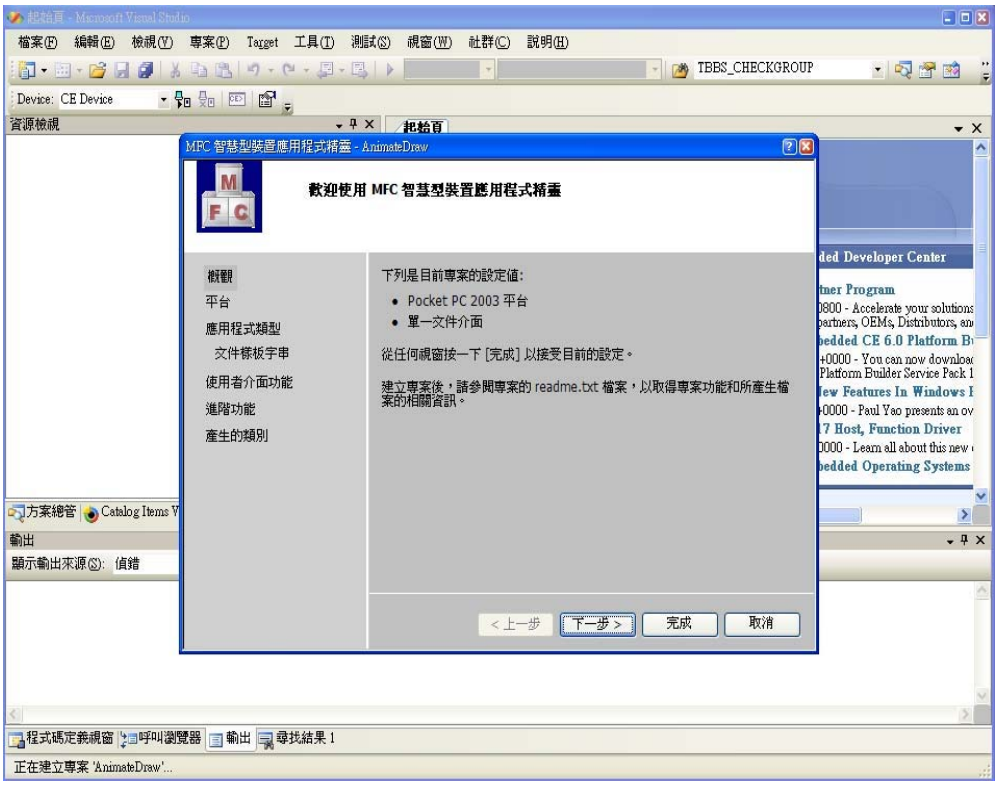
A. 在**Visual Studio 2005** 整合式開發環境中，點選左上角[檔案]->[新增]->[專案]後便會開啟新增專案視窗。



B. 在新增專案視窗內，左邊專案類型欄中選擇[智慧型裝置]；右邊 **Visual Studio** 安裝的範本欄中選擇[**MFC** 智慧型裝置應用程式]。

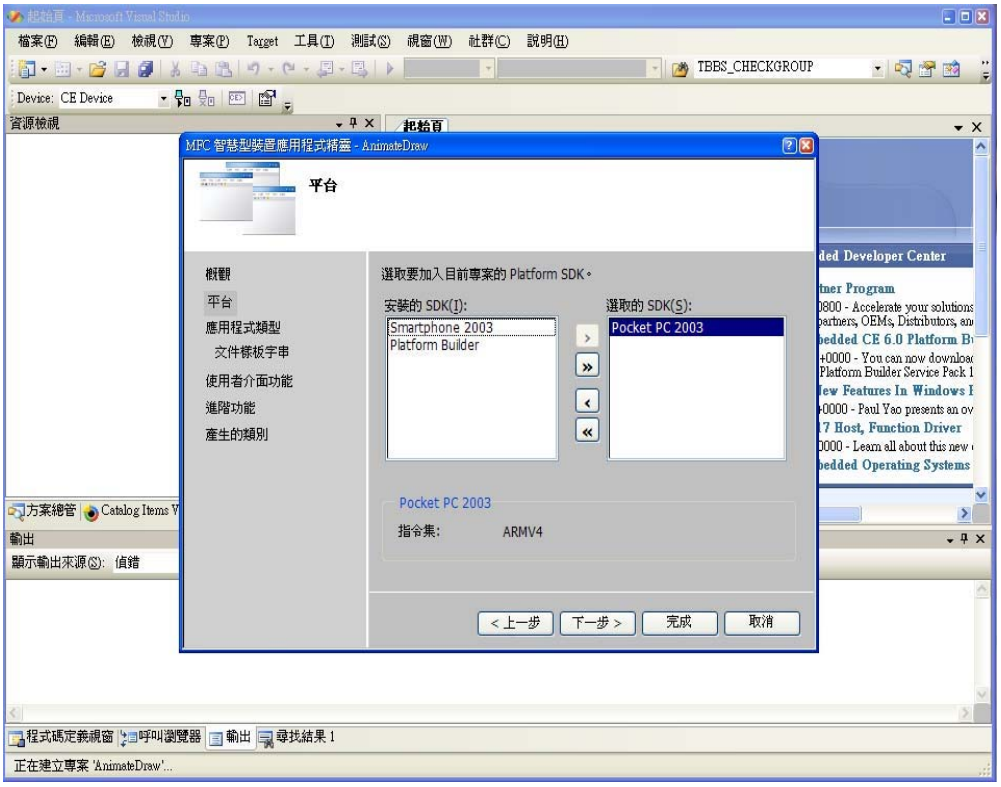


C. 在 **MFC** 智慧型裝置應用程式精靈對話窗中選擇［下一步］按鈕。

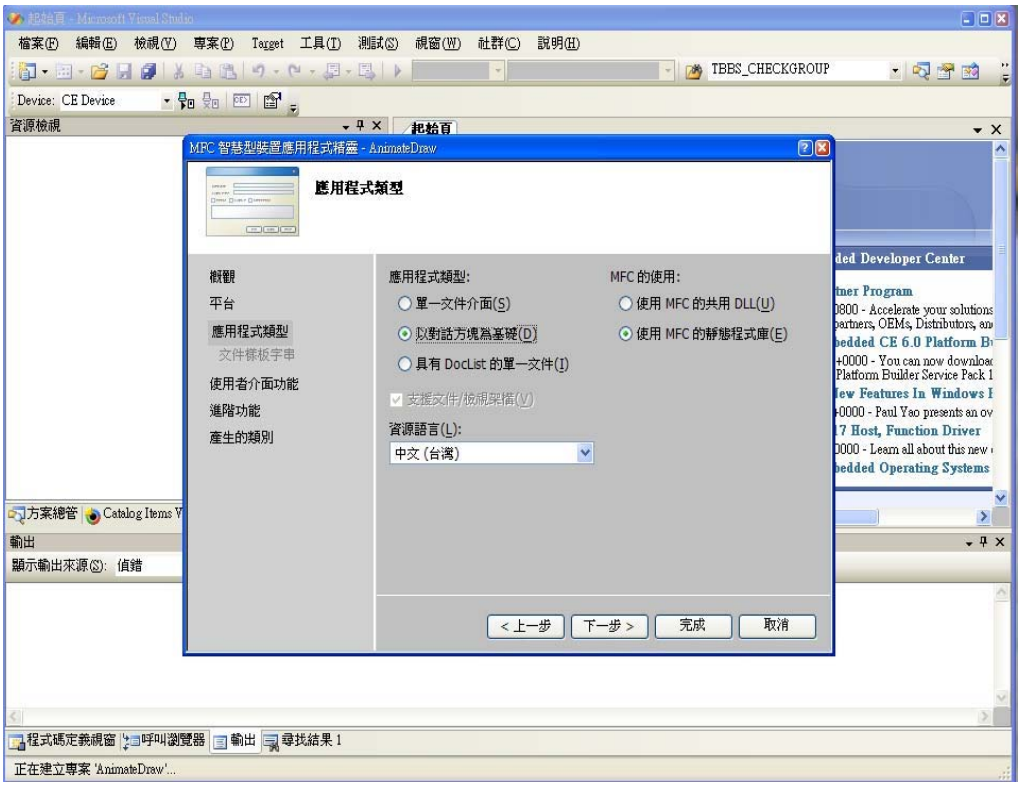


D. 選取要加入目前專案的平台軟體開發包（**Platform SDK**）在此實驗中

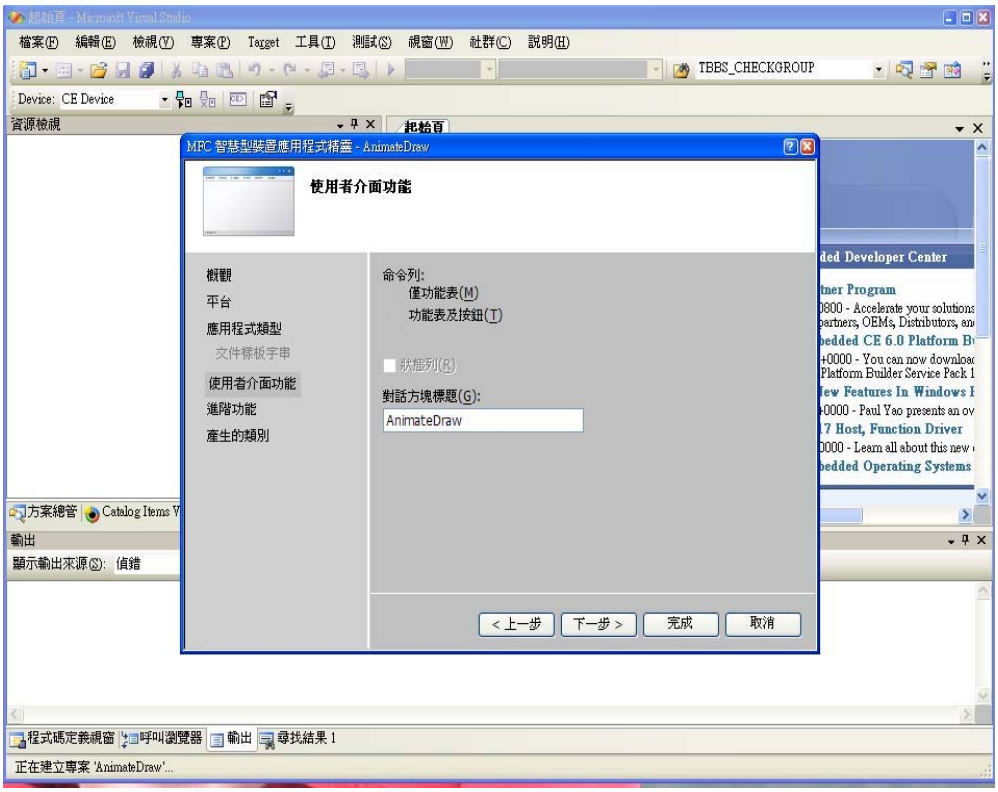
我們選擇[**Pocket PC 2003**]，之後按下[下一步] 按鈕。



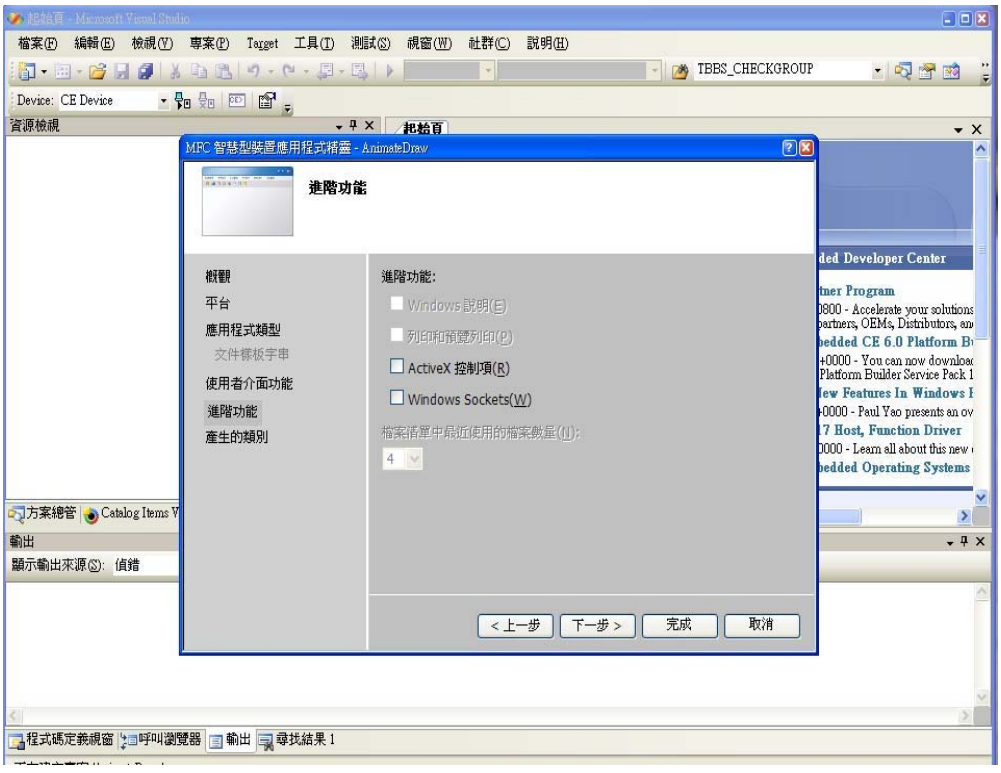
E. 應用程式類型中選取 [以對話方塊為基礎]，**MFC** 的使用中選取[使用 **MFC** 的靜態程式庫] ，之後按下[下一步] 按鈕。



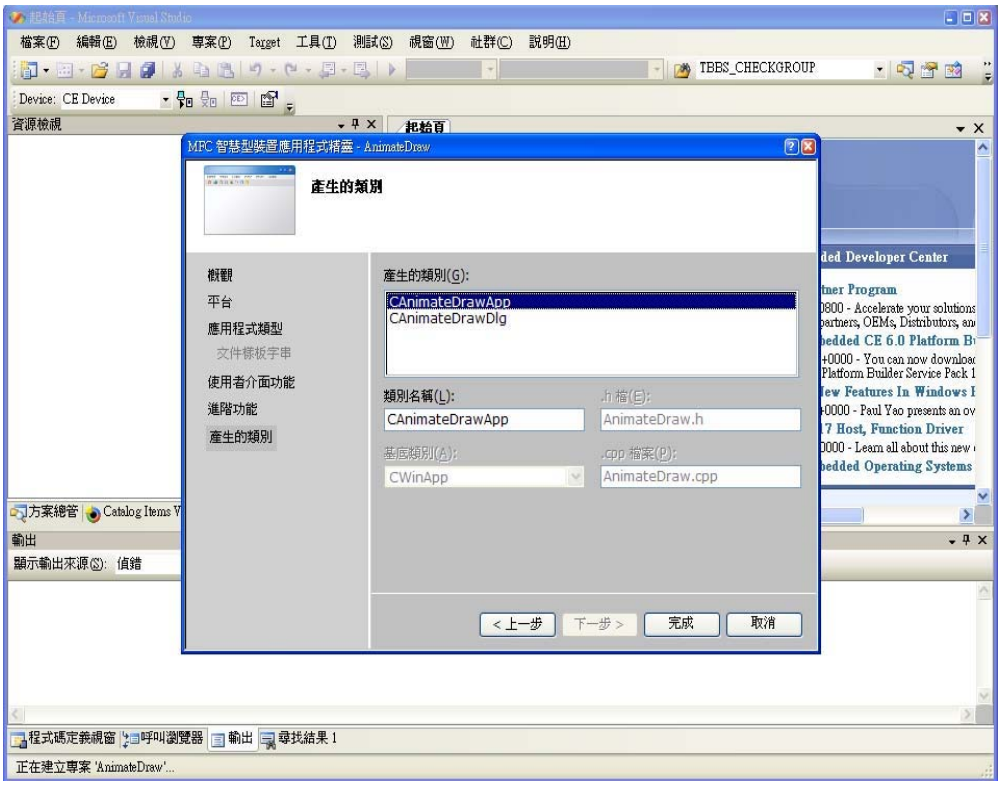
F. 按下[下一步]按鈕。



G. 按下[下一步]按鈕。

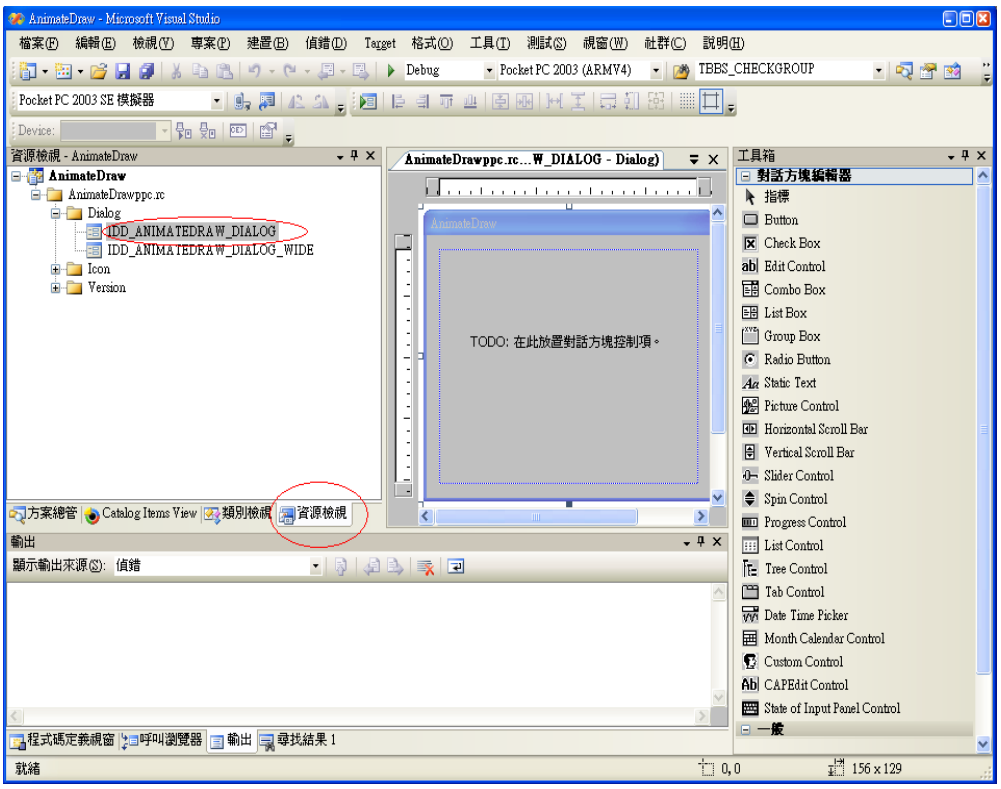


H. 按下[完成] 按鈕。

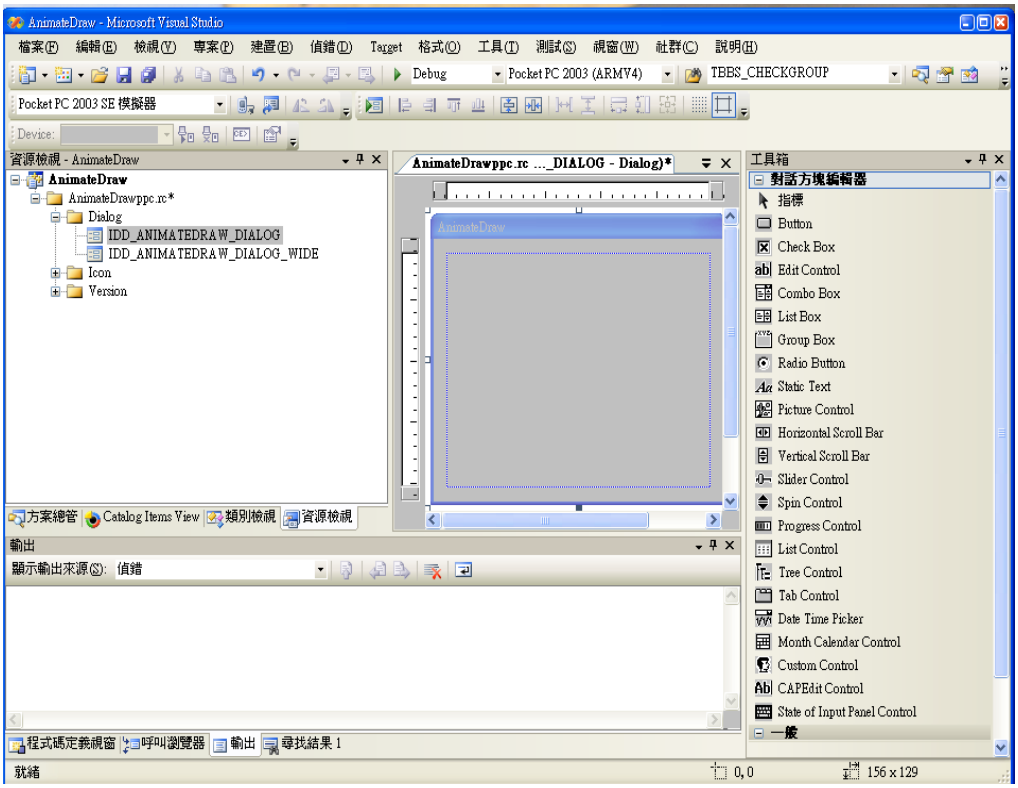


2. 設計使用者界面

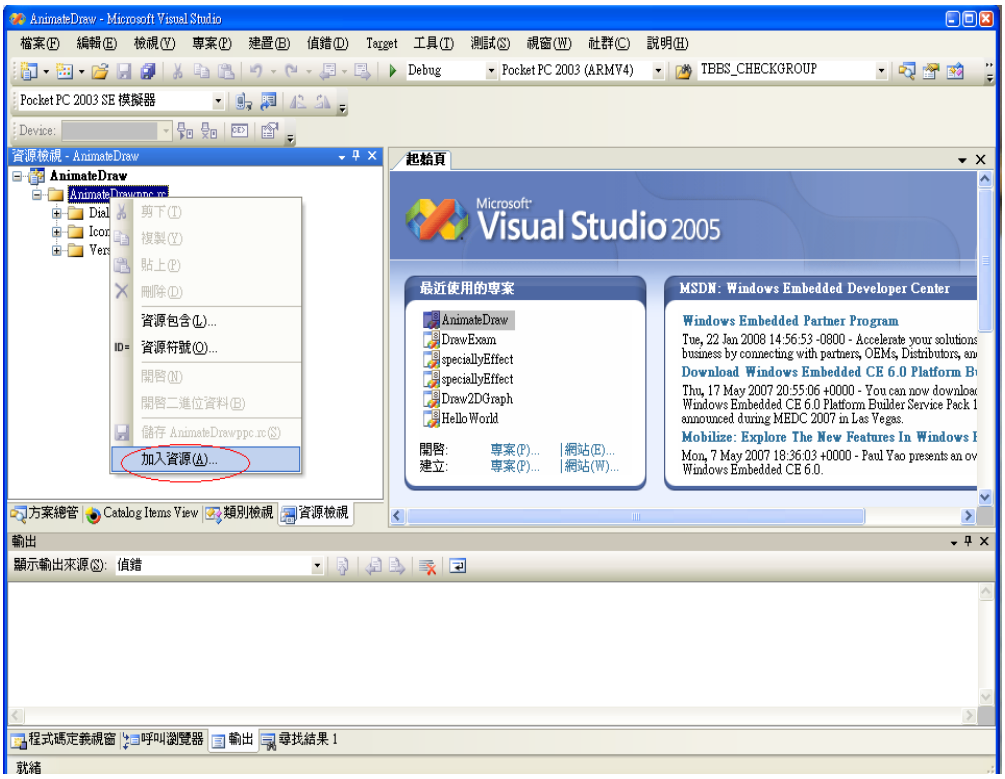
A. 切換左方視窗到資源檢視，準備開始設計使用者對話方塊介面（**UI**）



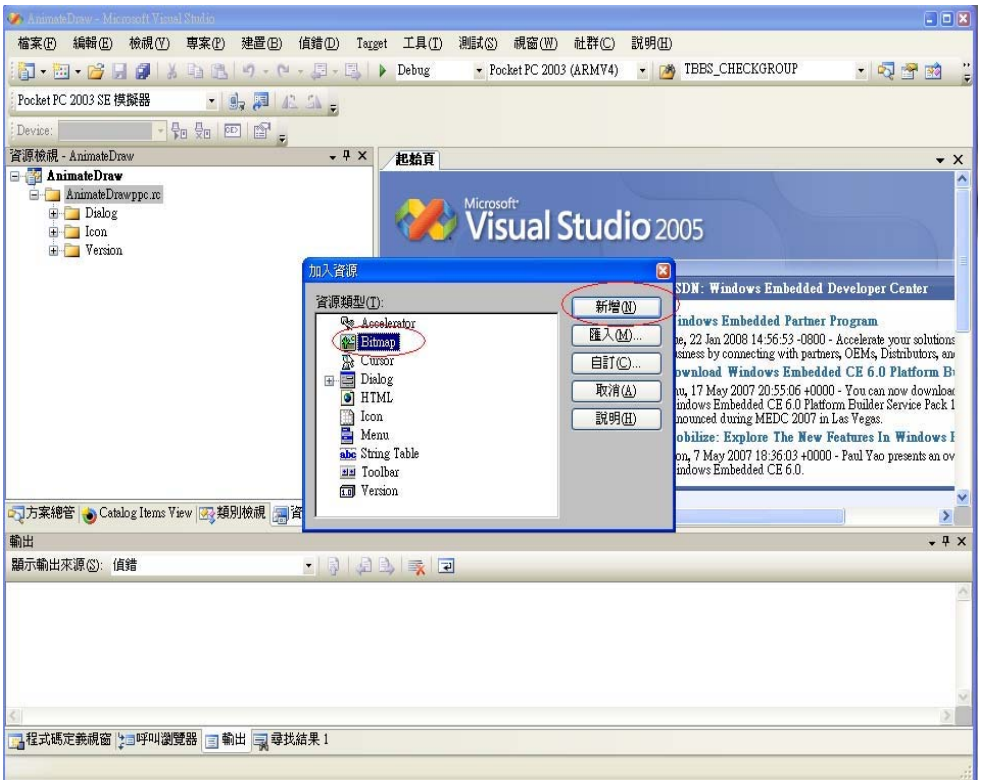
B. 將對話方塊中存在的文字對話方塊 （**TODO**：在此放置對話方塊控制項）刪除。



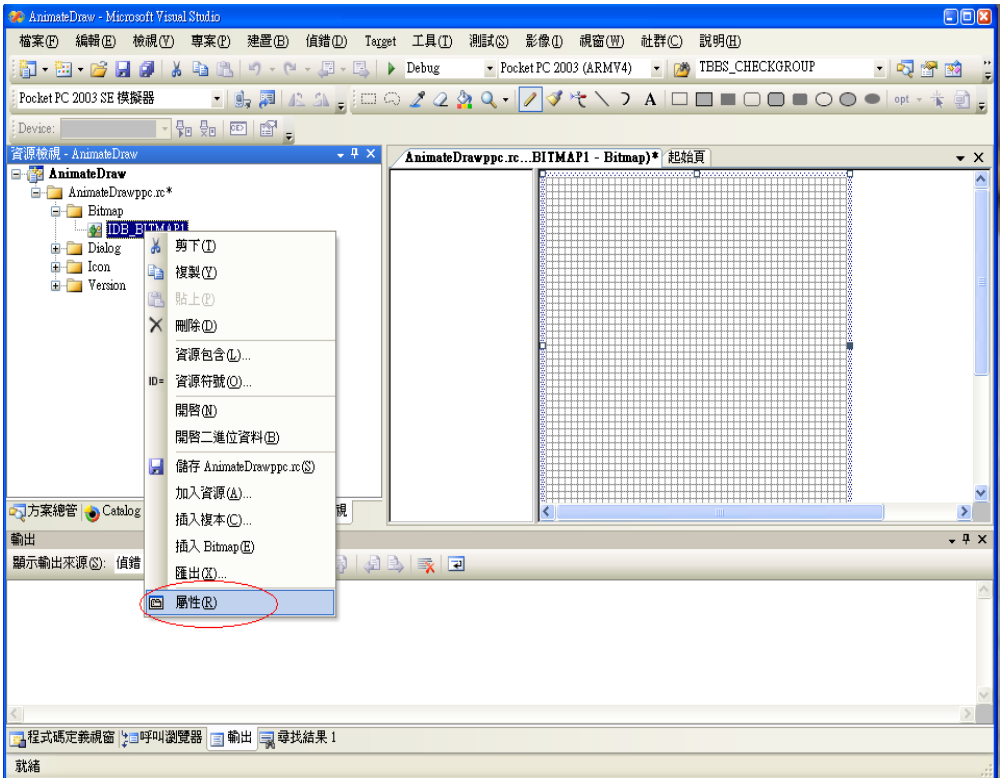
C. 在資源檢視視窗 **AnimateDraw.rc** 上按下滑鼠右鍵，選擇〔加入資源〕



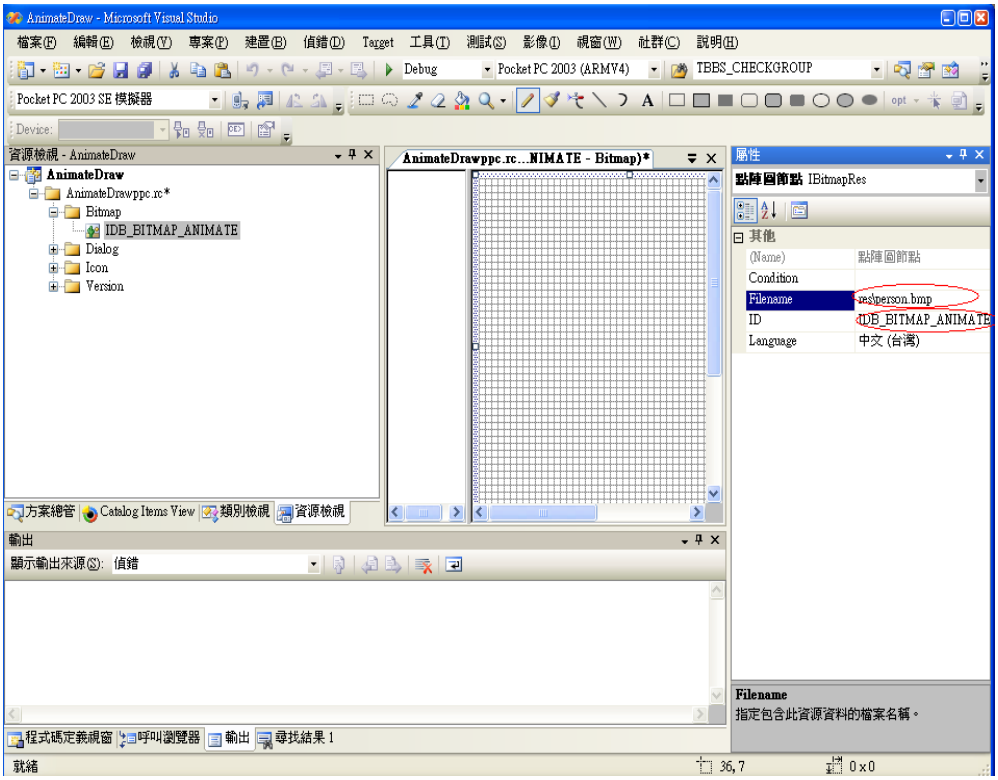
D. 在加入資源對話窗的資源類型，選擇[**Bitmap**]，按下新增按鈕。



E. 在資源檢視視窗 **Bitmap** 上按下滑鼠右鍵，選擇〔屬性〕

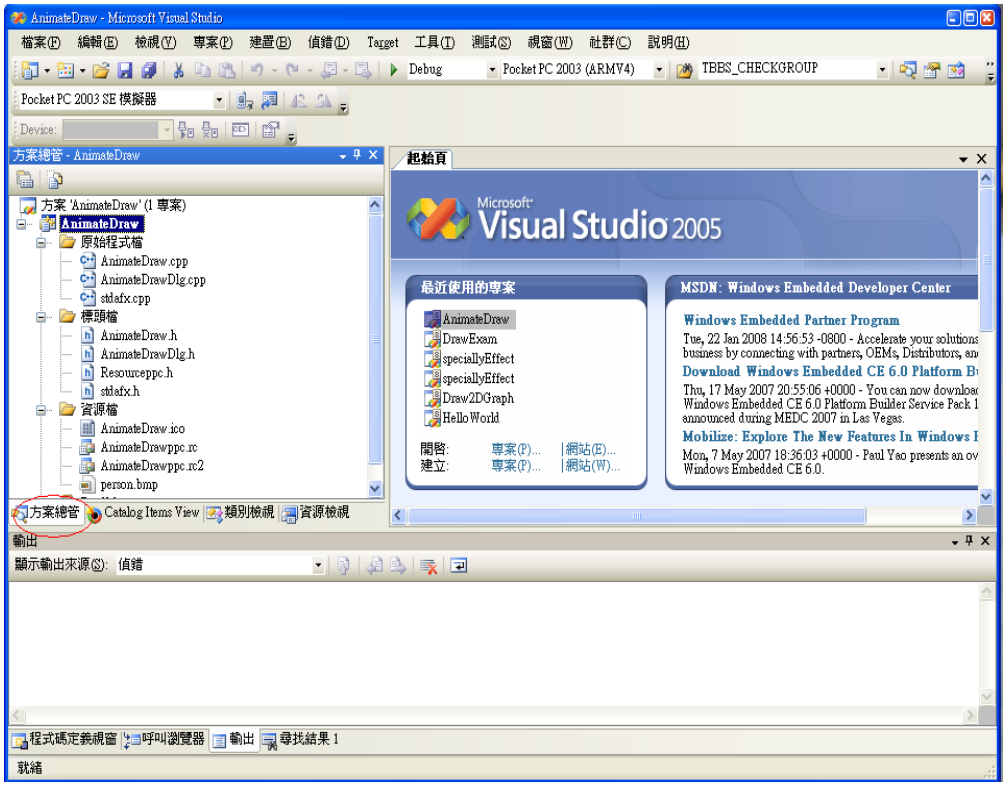


F. 在右方的屬性視窗中，變更［Filename］為圖片存放的相對路徑。

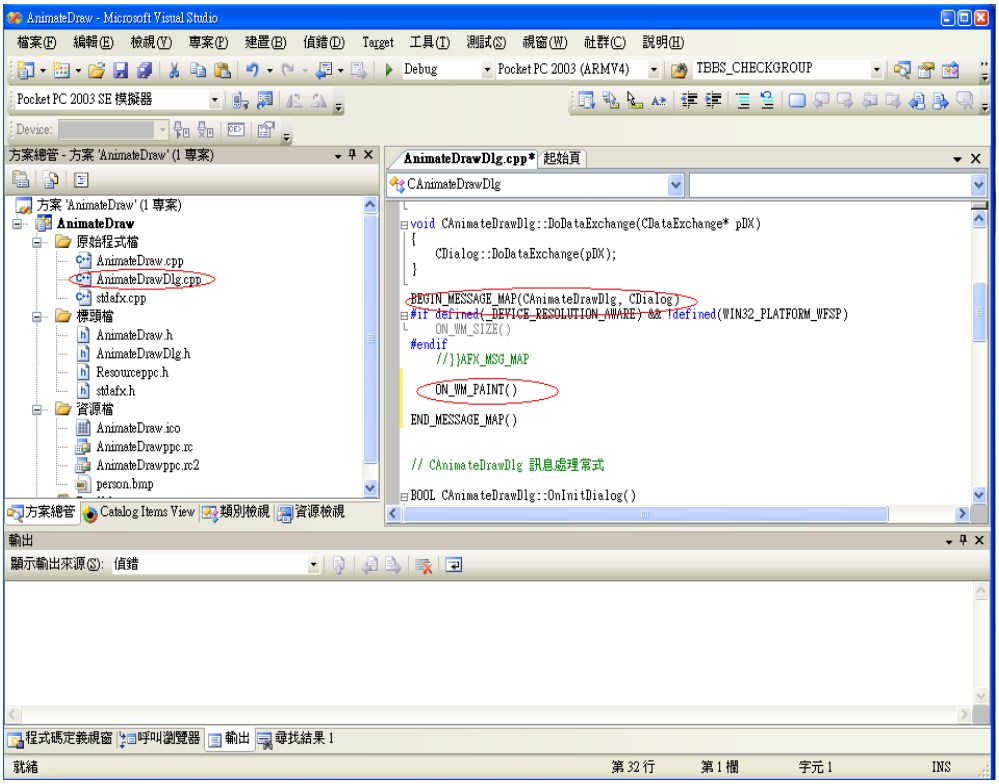


3. 程式設計

A. 切換左方視窗到方案總管，準備開始設計使用者程式。



B. 在方案總管視窗中的原始程式檔內，開啟 **AnimateDrawDlg.cpp** ，在 **BEGIN MESSAGE_MAP(CAnimateDrawDlg,CDialog)** 後，加入一段程式碼。



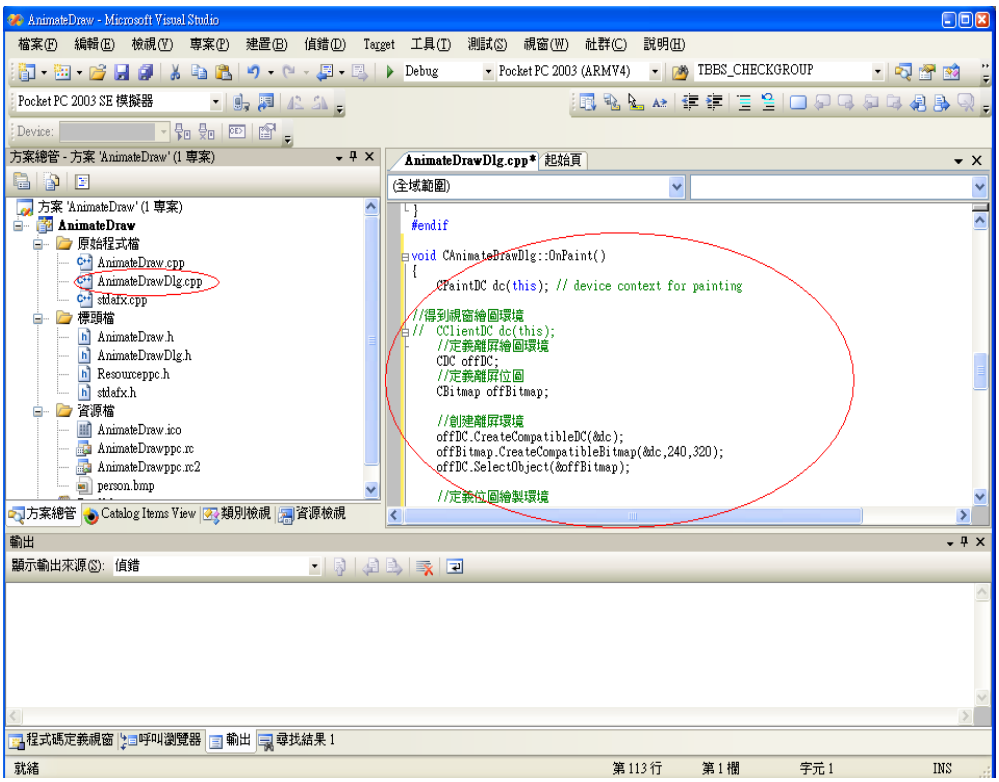
加入的程式碼 ：

//ON_WM_PAINT：對視窗進行繪製。

ON_WM_PAINT()

C. 在**AnimateDrawDlg.cpp** 中的

Void CAnimateDrawDlg::OnPaint() 後，加入一段程式碼。



加入的程式碼 ：

void CAnimateDrawDlg::OnPaint()

{

// 用來管理描繪**DC**，裝置內容只會在反應**WM_PAINT** 訊息的時候被使用，應用在**OnPaint** 函數。

// **CPaintDC** 類別的物件不只是用來建立一個裝置內容，還會在類 別建構式裡呼叫視窗 **API** 函式 **BeginPaint()**，並在解構式中呼叫 **EndPaint()**。

// **CPaintDC** 只需要一個指標引數，用來指向建立好的**DC** 視窗。

// **CPaintDC** 可以使用**CDC** 中的所有函數。

CPaintDC dc(this);

// 定義離屏繪圖環境

CDC offDC;

// 定義離屏位圖

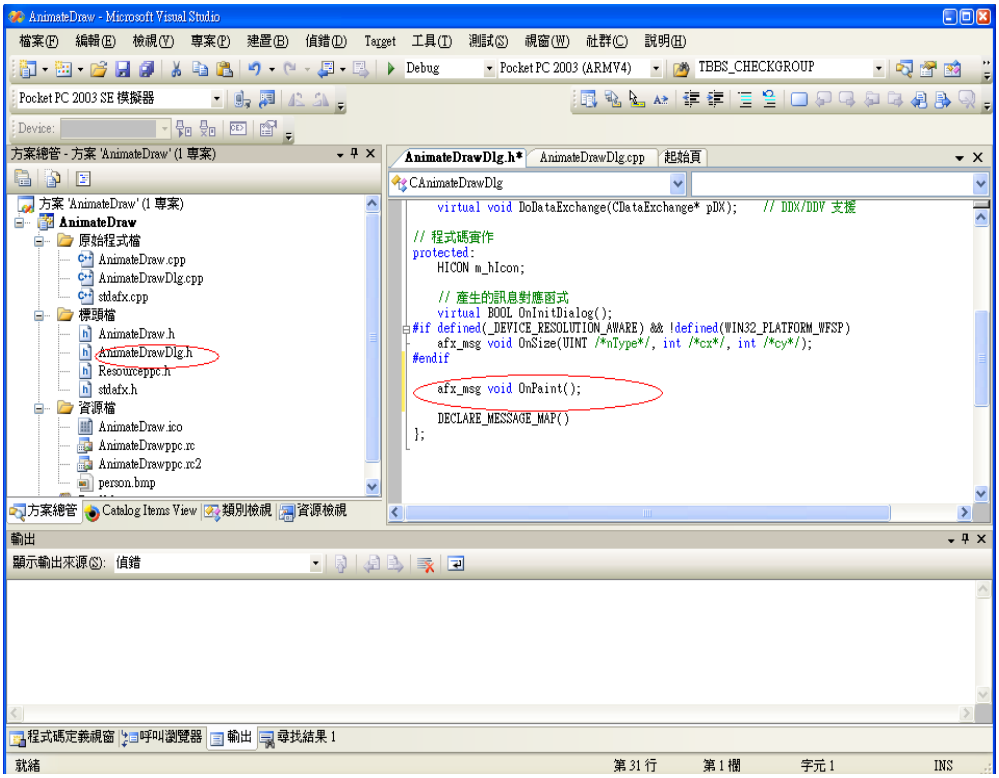
// **MFC** 中的**CBitmap** 類別封裝了一個用於**Windows** 的**Bitmap**，提供創建和載入**Bitmap** 的方法

CBitmap offBitmap;

```
//  創建離屏環境

//  建立一個相容的記憶體裝置
offDC.CreateCompatibleDC(&dc);

//  建立一個和設備相容的記憶體位元映射圖
offBitmap.CreateCompatibleBitmap(&dc,240,320);

//  選擇一個物件到一個裝置環境
offDC.SelectObject(&offBitmap);

//  定義位圖繪製環境
CDC bmpDC;
CBitmap aniBmp;

//  加載位圖
//  將位元映射式資源載入記憶體中
aniBmp.LoadBitmap(IDB_BITMAP_ANIMATE);

//  建立一個和設備相容的記憶體位元映射圖
bmpDC.CreateCompatibleDC(&offDC);

//  選擇一個物件到一個裝置環境
bmpDC.SelectObject(aniBmp);

//  循環顯示位圖，形成動畫效果
for(int nCnt=0; nCnt<4; nCnt++)
{
    //  設定離屏環境為純白色
    //  將刷子圖樣輸出到設備中
    offDC.PatBlt(0, 0, 240, 320, WHITENESS);

    //  繪製位圖到離屏繪圖環境中
    //  複製位元圖，省略透明顏色繪製的部分
    TransparentImage(offDC.m_hDC,nCnt*4,0,46,57,bmpDC.m_hD
                     C,nCnt*46,0,46,57, RGB(255,0,255));

    //  將離屏繪圖環境繪製到實際視窗繪圖環境
```

```
        //  從記憶體將一個位元映射圖複製到另一個設備中
        dc.BitBlt(0,0,240,320,&offDC,0,0,SRCCOPY);

        //  延遲時間
        Sleep(100);
    }

    //  刪除位圖GDI對象
    aniBmp.DeleteObject();

    //  刪除位圖繪製環境
    bmpDC.DeleteDC();

    //  刪除離屏位圖GDI對象
    offBitmap.DeleteObject();

    //  刪除離屏繪圖環境
    offDC.DeleteDC();

    //  設為下次WM_PAINT  發出後需要重繪的區域
    Invalidate();
}
```

D. 在方案總管視窗中的標頭檔內，開啟 **AnimateDrawDlg.h** ，在 **AnimateDrawDlg.h** 中加入一段程式碼



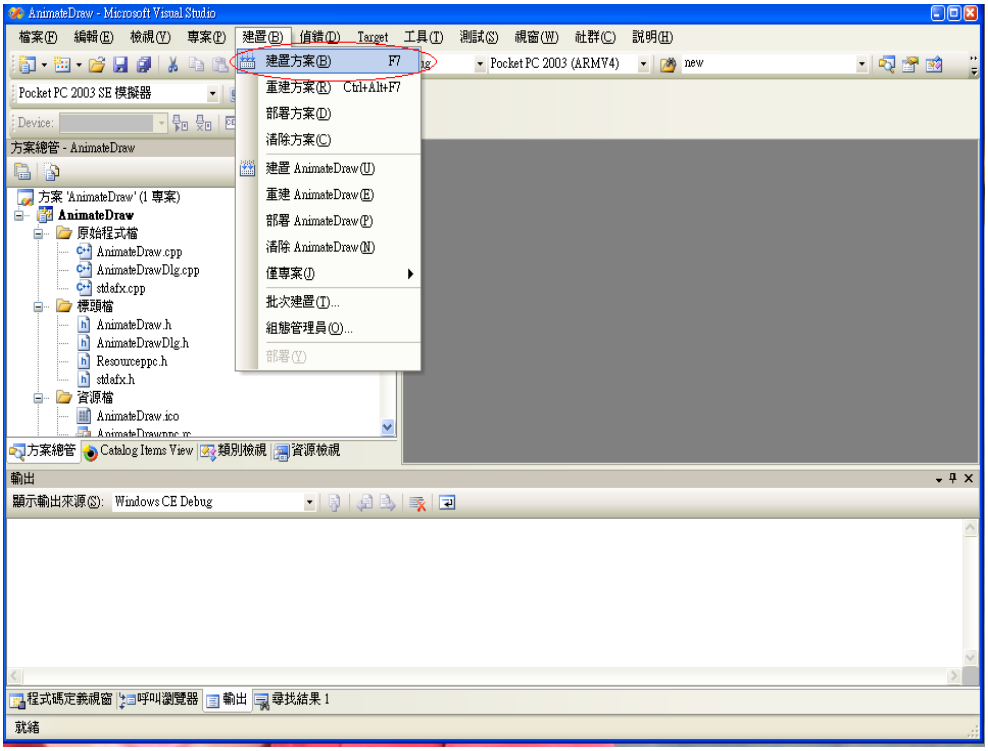
加入程式碼 ：

// **afx_msg** 表示後面的函數是訊息處理函數，編譯時不會被編譯

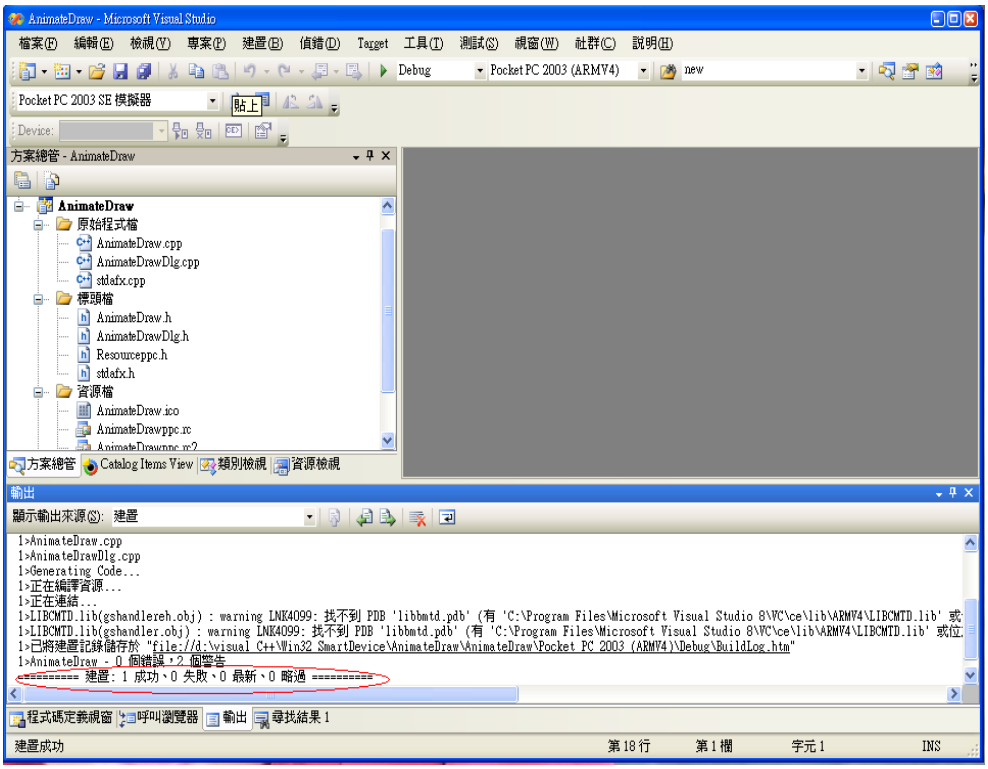
afx_msg void OnPaint();

4. 編譯和執行

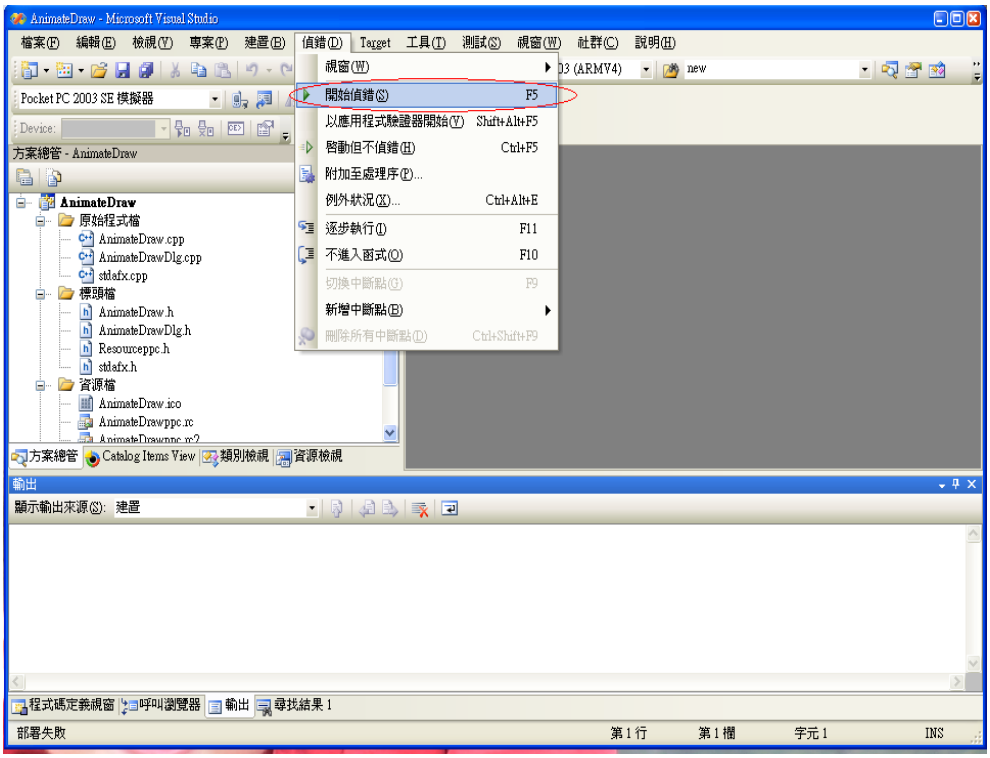
A. 在**Visual Studio 2005** 整合式開發環境中，點選[建置] -> [建置專案]後便會開始編譯整個專案。



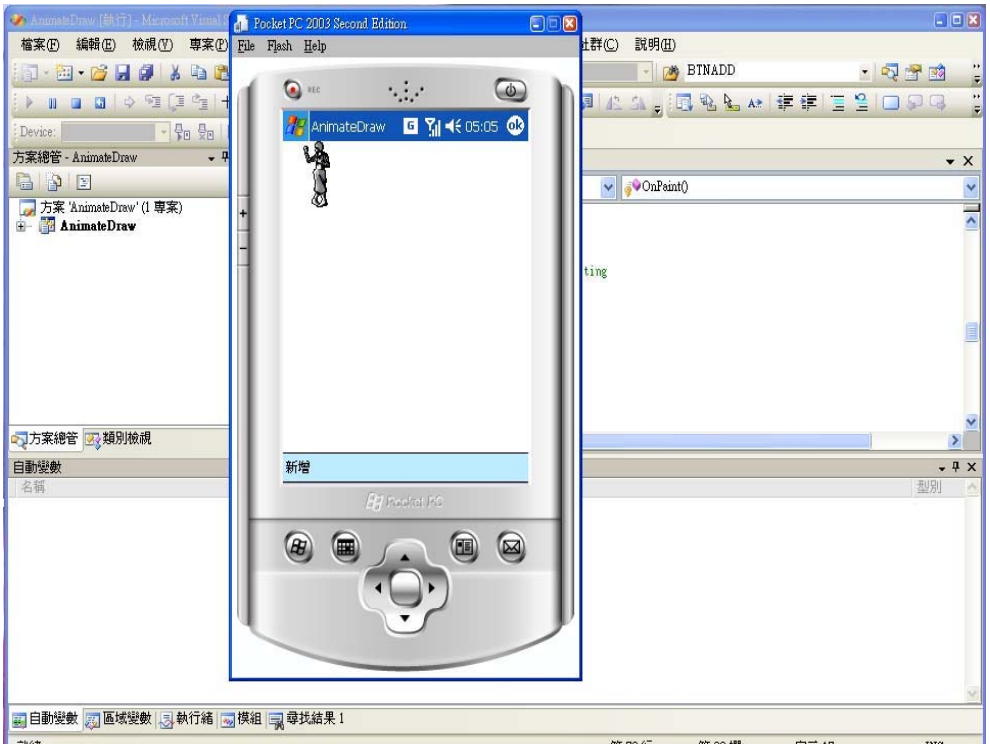
B. 編譯成功後，在**Visual Studio 2005** 整合式開發環境下方的輸出視窗中顯示建置成功。



C. 在**Visual Studio 2005** 整合式開發環境中，點選[偵錯] -> [開始偵錯]後便會開始對此專案執行並偵錯。


AnimateDraw - Microsoft Visual Studio
檔案(F) 編輯(E) 檢視(V) 專案(P) 建置(B) 偵錯(D) Target 工具(T) 測試(S) 視窗(W) 社群(C) 說明(H)
視窗(W)
開始偵錯(S) F5
以應用程式驗證器開始(Y) Shift+Alt+F5
啓動但不偵錯(H) Ctrl+F5
附加至處理序(P)...
例外狀況(X)... Ctrl+Alt+E
逐步執行(I) F11
不進入函式(O) F10
切換中斷點(G) F9
新增中斷點(B)
刪除所有中斷點(D) Ctrl+Shift+F9
Pocket PC 2003 SE 模擬器
方案總管 - AnimateDraw
AnimateDraw
原始程式檔
AnimateDraw.cpp
AnimateDrawDlg.cpp
stdafx.cpp
標頭檔
AnimateDraw.h
AnimateDrawDlg.h
Resourceppc.h
stdafx.h
資源檔
AnimateDraw.ico
AnimateDrawppc.rc
方案總管 Catalog Items View 類別檢視 資源檢視
輸出
顯示輸出來源(S): 建置
程式碼定義視窗 呼叫瀏覽器 輸出 尋找結果 1
部署失敗
第 1 行 第 1 欄 字元 1 INS

D. 執行結果如下圖所示。


Pocket PC 2003 Second Edition
File Flash Help
AnimateDraw 05:05 ok
新增
方案總管 類別檢視
自動變數
名稱
型別
自動變數 區域變數 執行緒 模組 尋找結果 1

應用程式開發實驗

實驗二

圖形顯示特效

Rev. 1.0

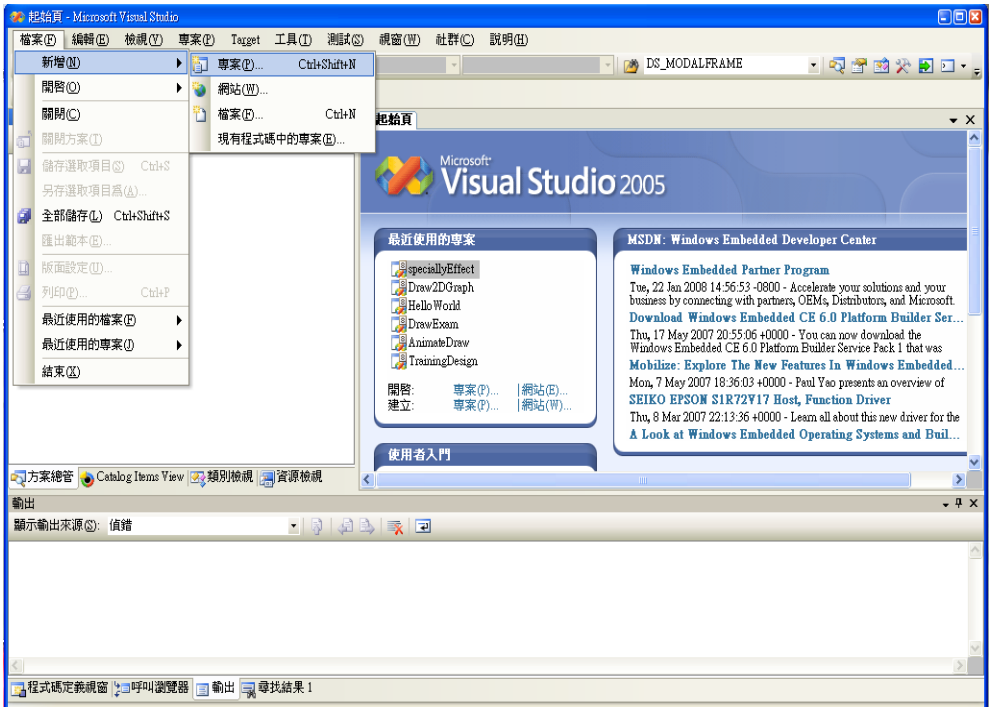
一. 實驗目的： 在日常生活中，我們經常會在某些廣告看板上看到一幅圖畫以水平向右掃 瞄、水平向左掃瞄、水平百葉窗、垂直百葉窗和雨滴效果等方式進行顯示，

因為如此會給人更加醒目和更加深刻的印象。此實驗將介紹如何在**WinCE**中實現這些特殊方式顯示的方法。

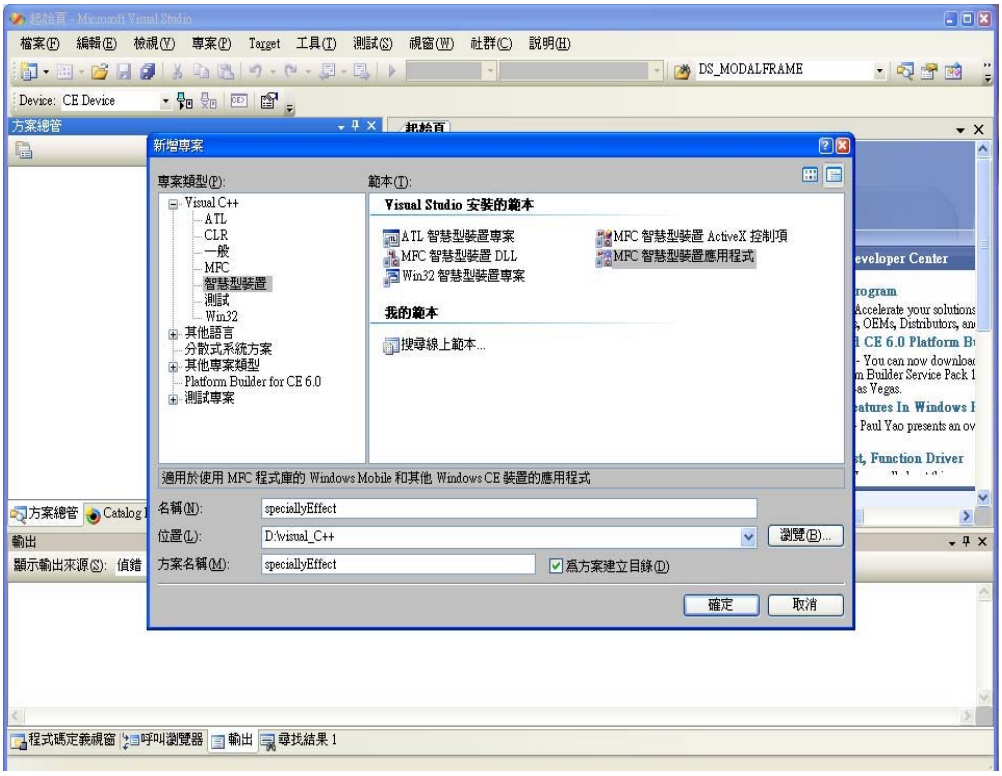
二. 實驗步驟：

1. 建立一個 **MFC** 對話方塊為基礎的專案

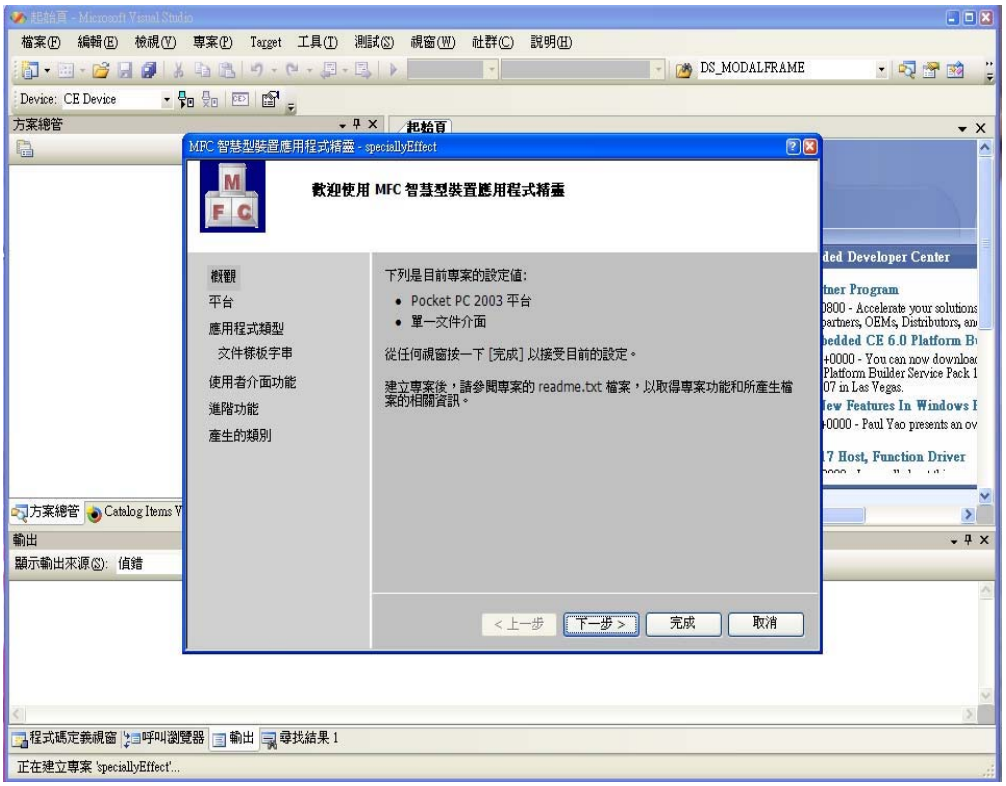
A. 在**Visual Studio 2005** 整合式開發環境中，點選左上角[檔案]->[新增]->[專案]後便會開啟新增專案視窗。



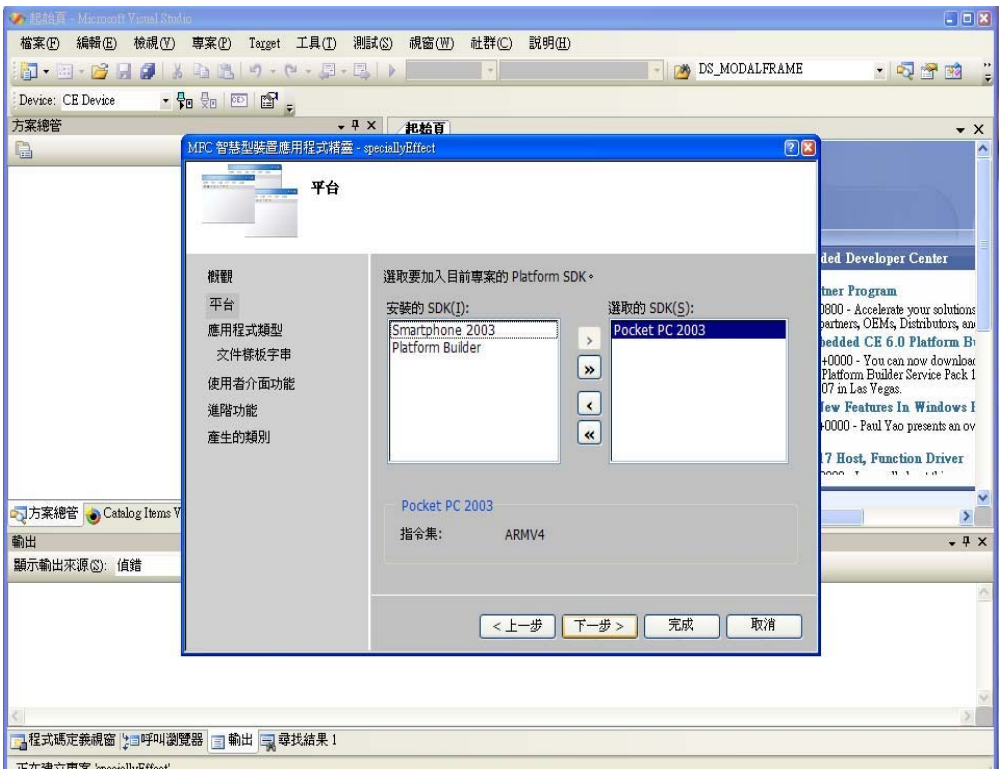
B. 在新增專案視窗內,左邊專案類型欄中選擇[智慧型裝置];右邊 **Visual Studio** 安裝的範本欄中選擇[**MFC** 智慧型裝置應用程式]。



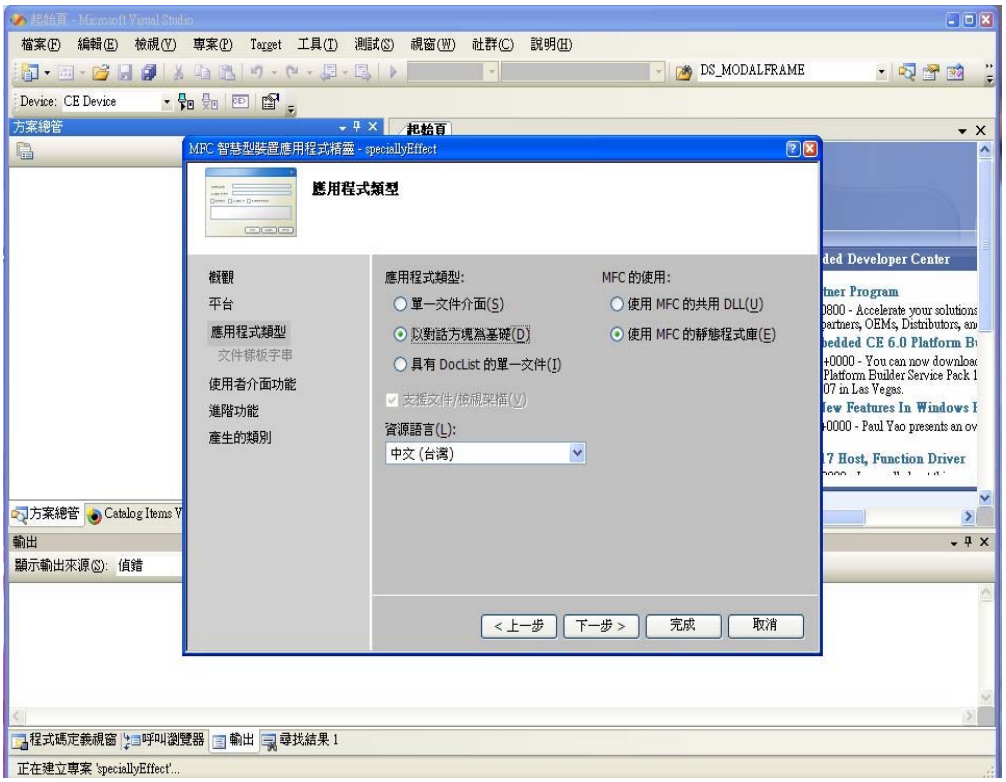
C. 在 **MFC** 智慧型裝置應用程式精靈對話窗中選擇［下一步］按鈕。



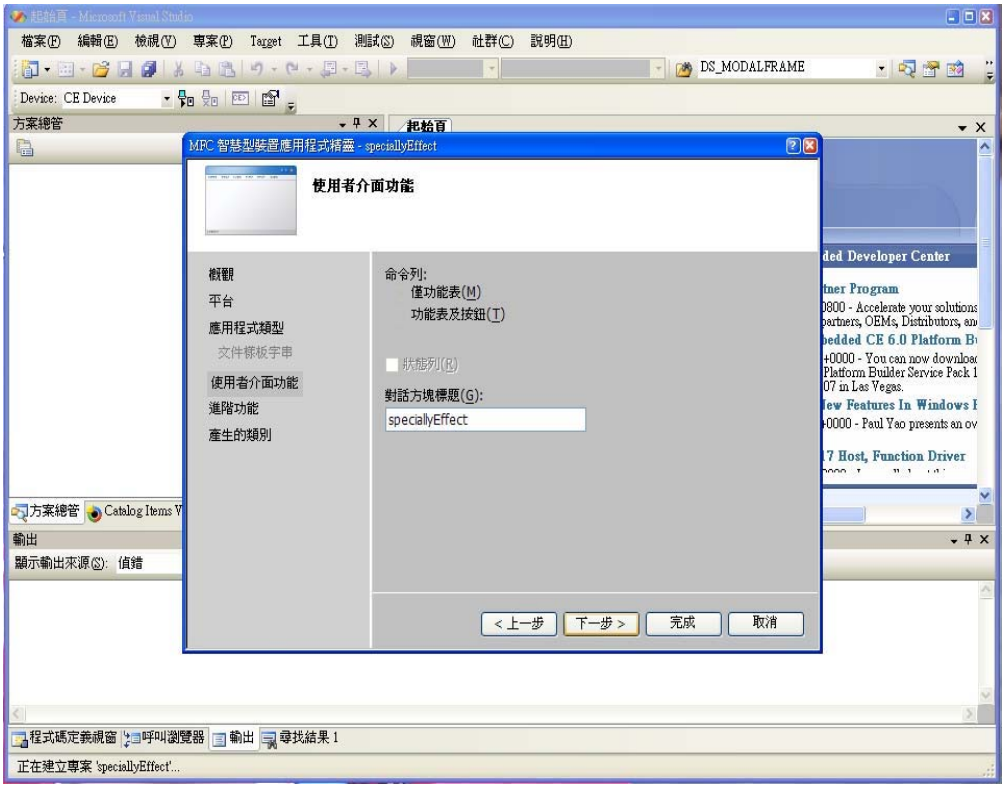
D. 選取要加入目前專案的平台軟體開發包（**Platform SDK**）在此實驗中 我們選擇[**Pocket PC 2003**]，之後按下[下一步］按鈕。



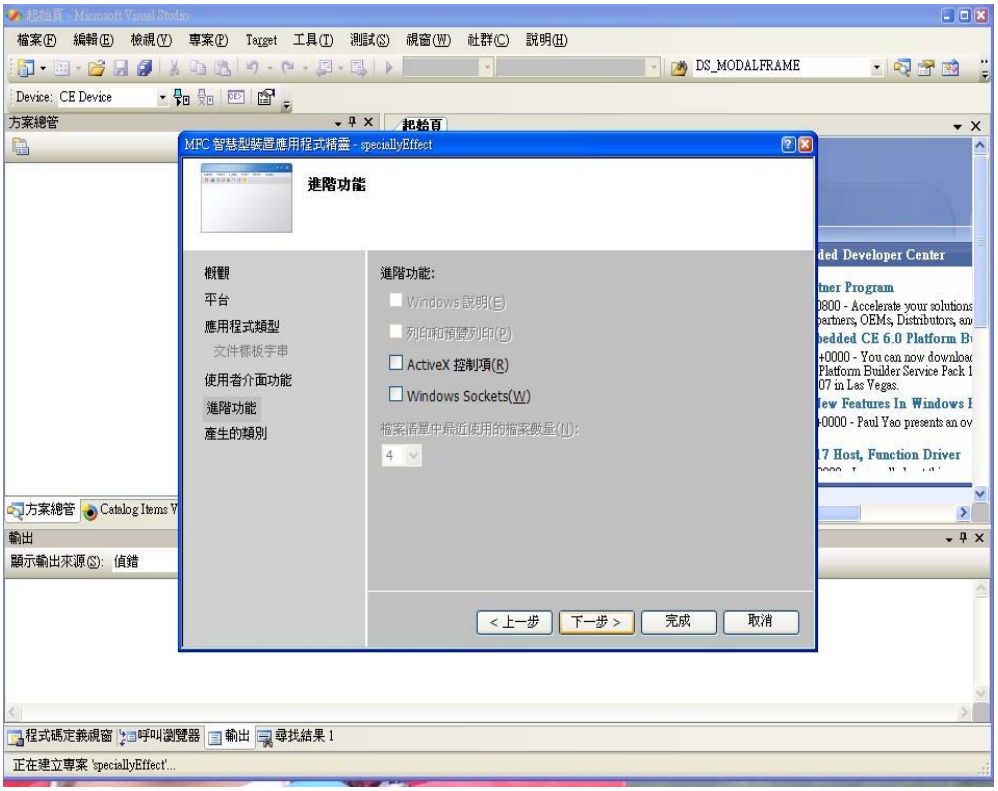
E. 應用程式類型中選取［以對話方塊為基礎］，**MFC** 的使用中選取[使用 **MFC** 的靜態程式庫］，之後按下[下一步］按鈕。



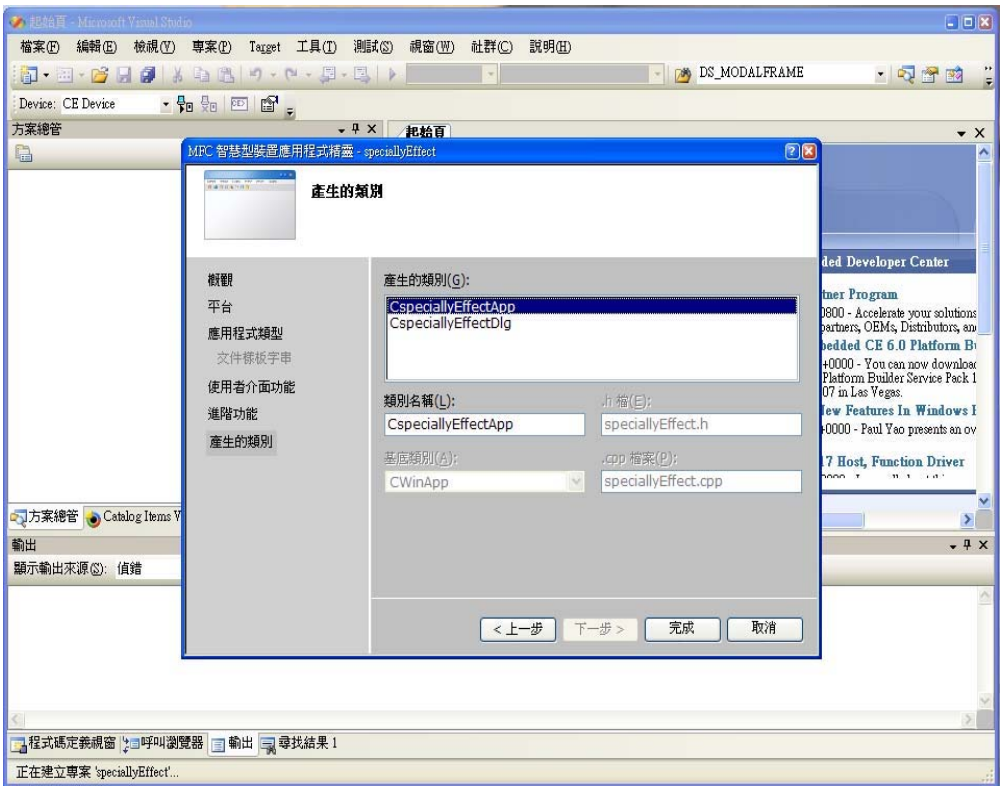
F. 按下[下一步］按鈕。



G. 按下[下一步] 按鈕。

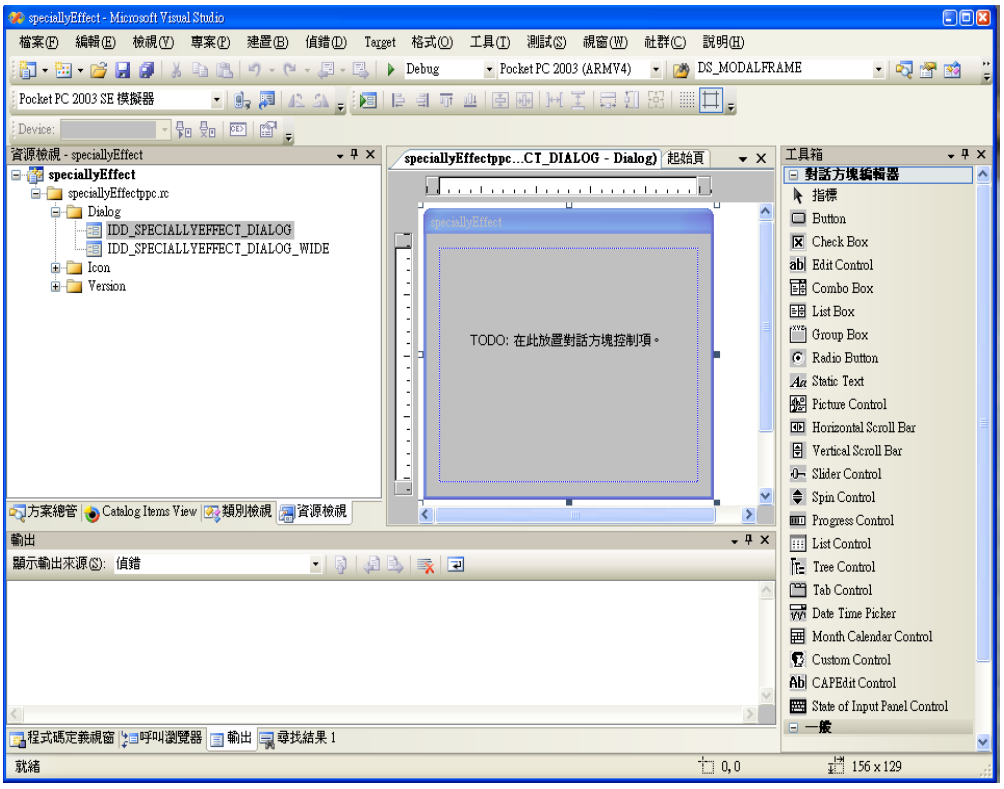


H. 按下[完成] 按鈕。

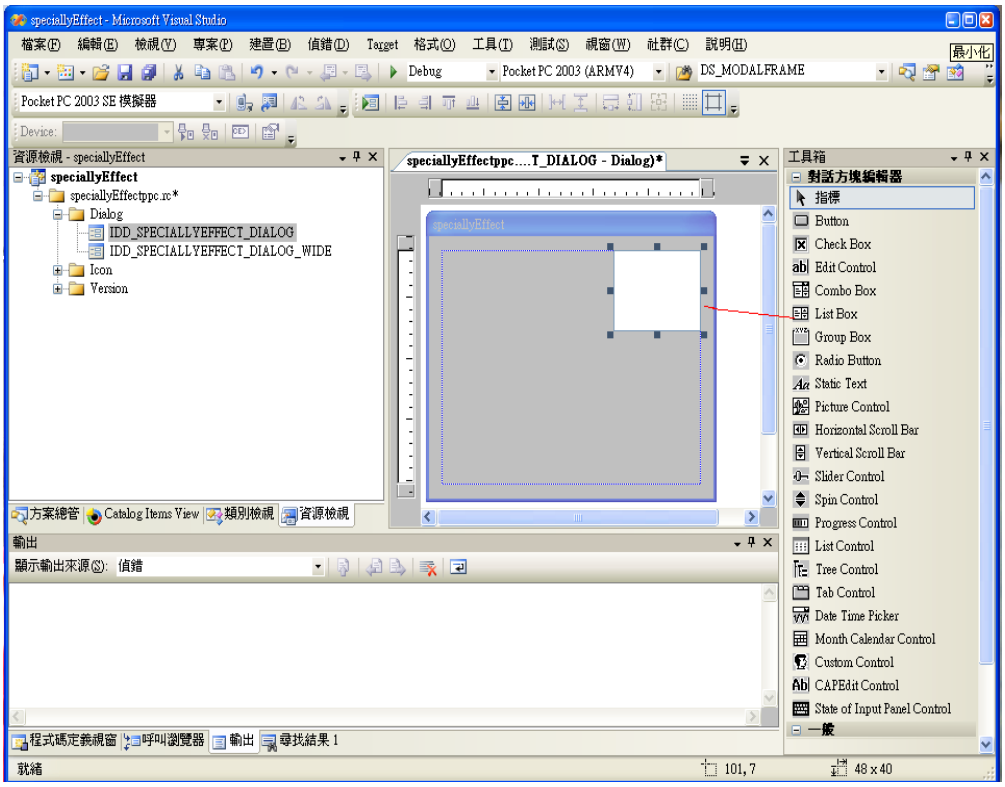


2. 設計使用者界面

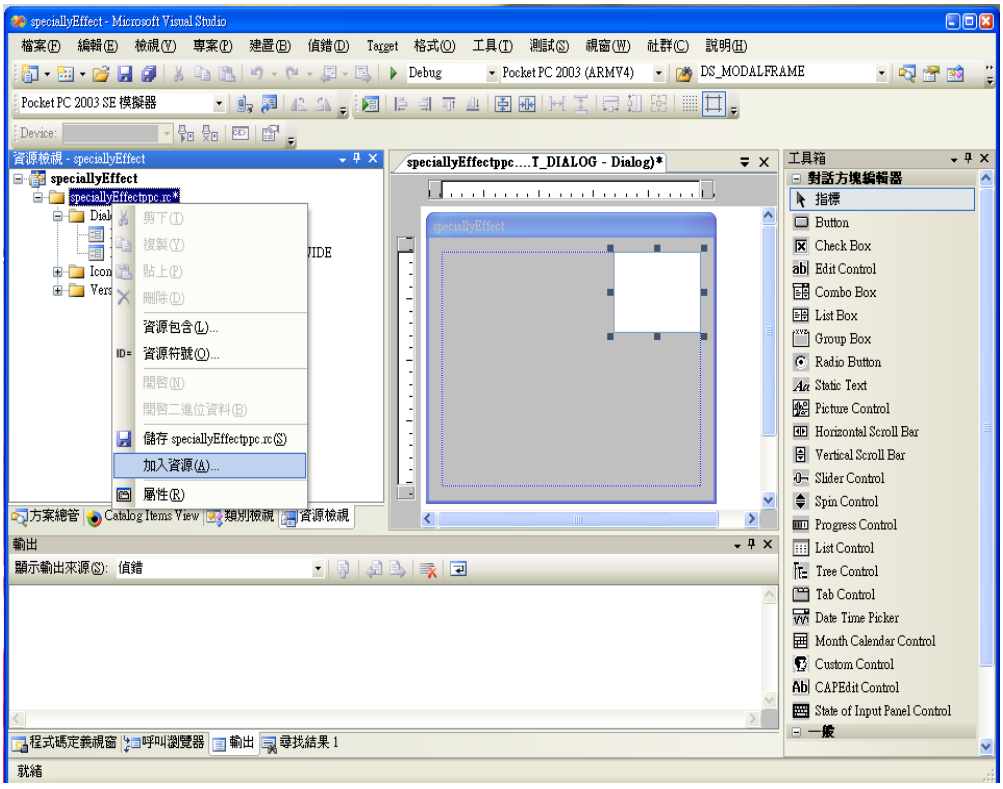
A. 切換左方視窗到資源檢視，準備開始設計使用者對話方塊介面（**UI**）



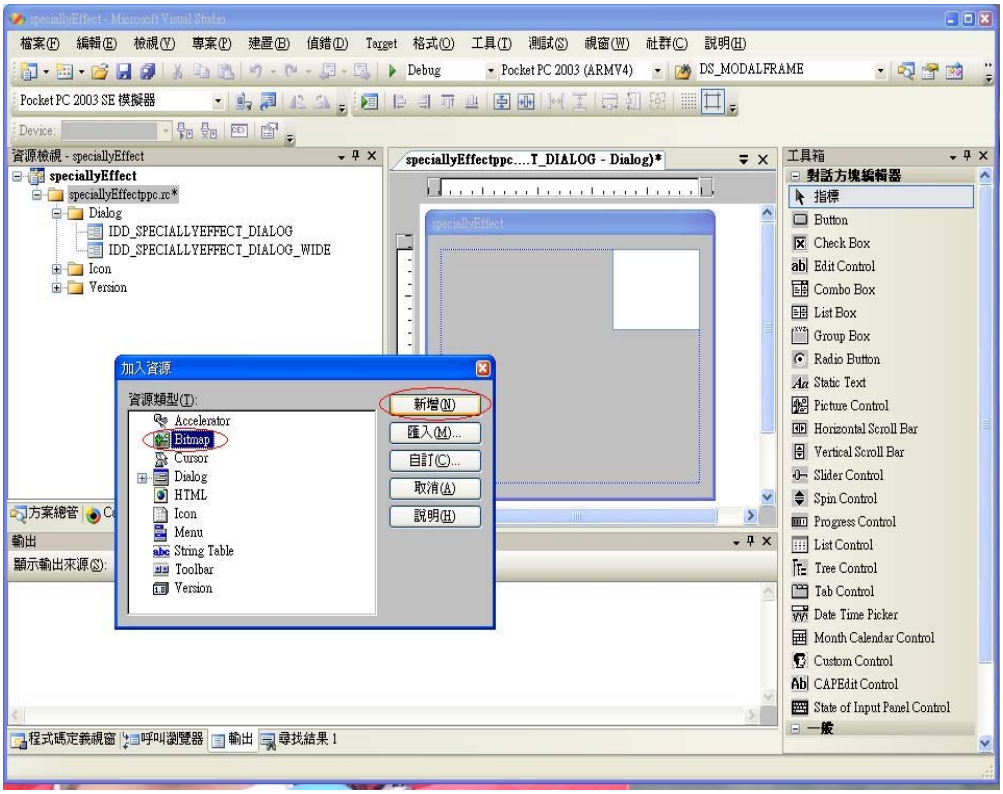
B. 將對話方塊中存在的文字對話方塊 （**TODO**：在此放置對話方塊控制項）刪除，然後將新選取［**ListBox**］對話方塊拖曳到適當位置。



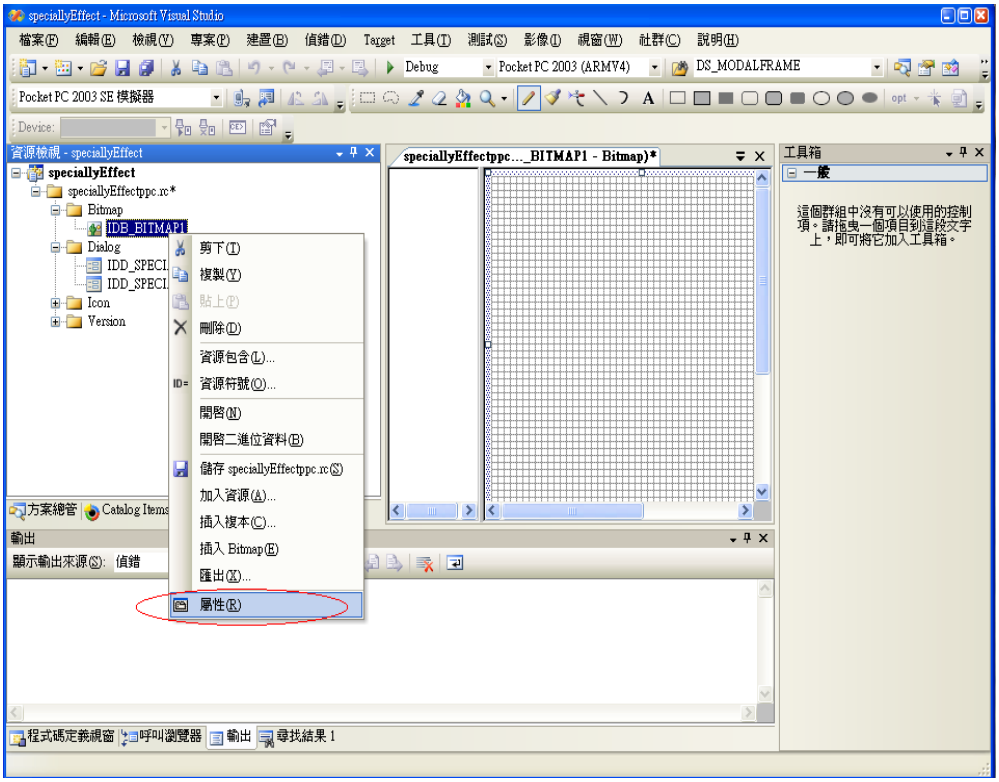
C. 在資源檢視視窗 **AnimateDraw.rc** 上按下滑鼠右鍵，選擇〔加入資源〕



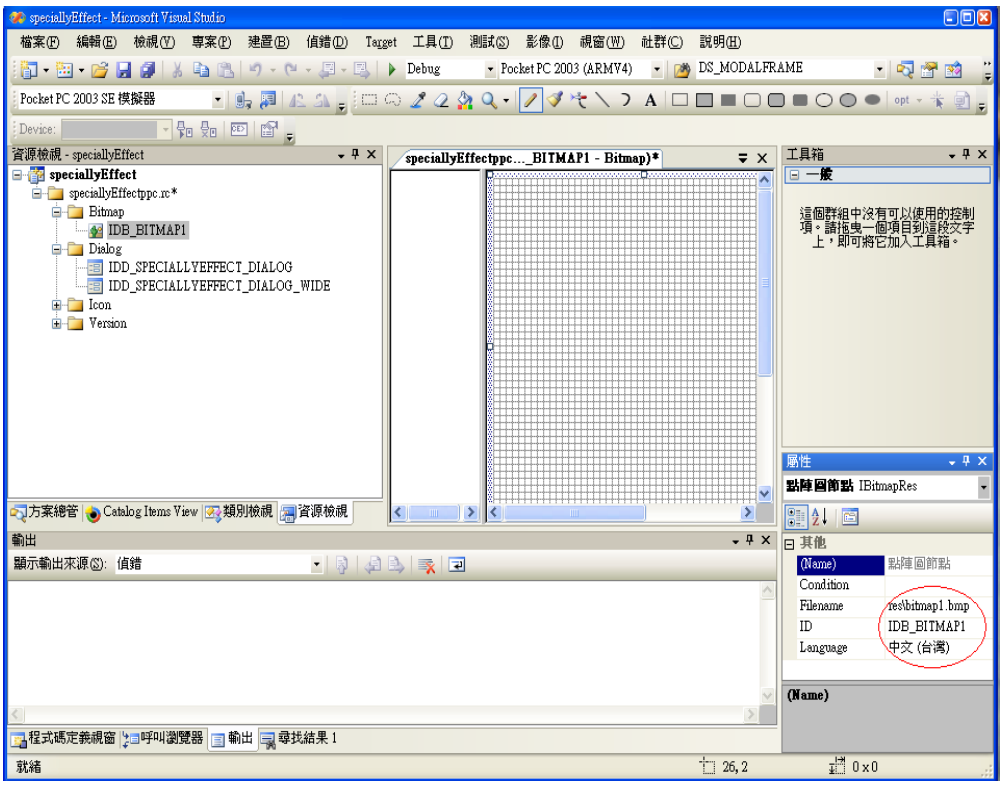
D. 在加入資源對話窗的資源類型，選擇[**Bitmap**]，按下新增按鈕。



E. 在資源檢視視窗 **Bitmap** 上按下滑鼠右鍵，選擇〔屬性〕

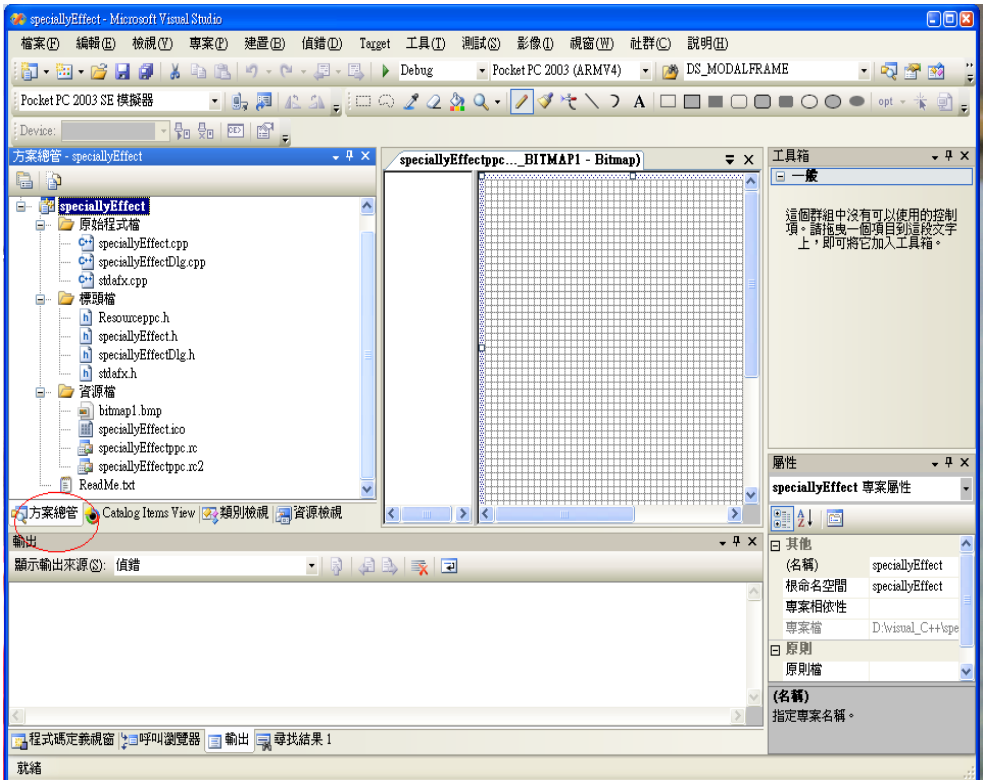


F. 在右方的屬性視窗中，變更［Filename］為圖片存放的相對路徑。

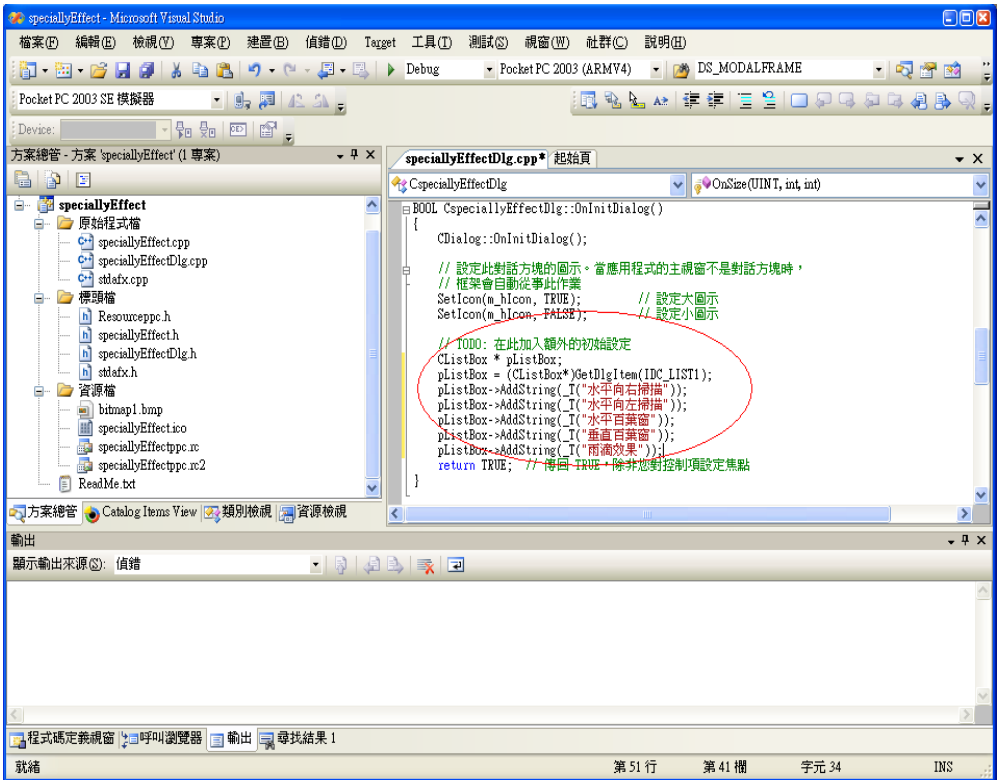


3. 程式設計

A. 切換左方視窗到方案總管，準備開始設計使用者程式。



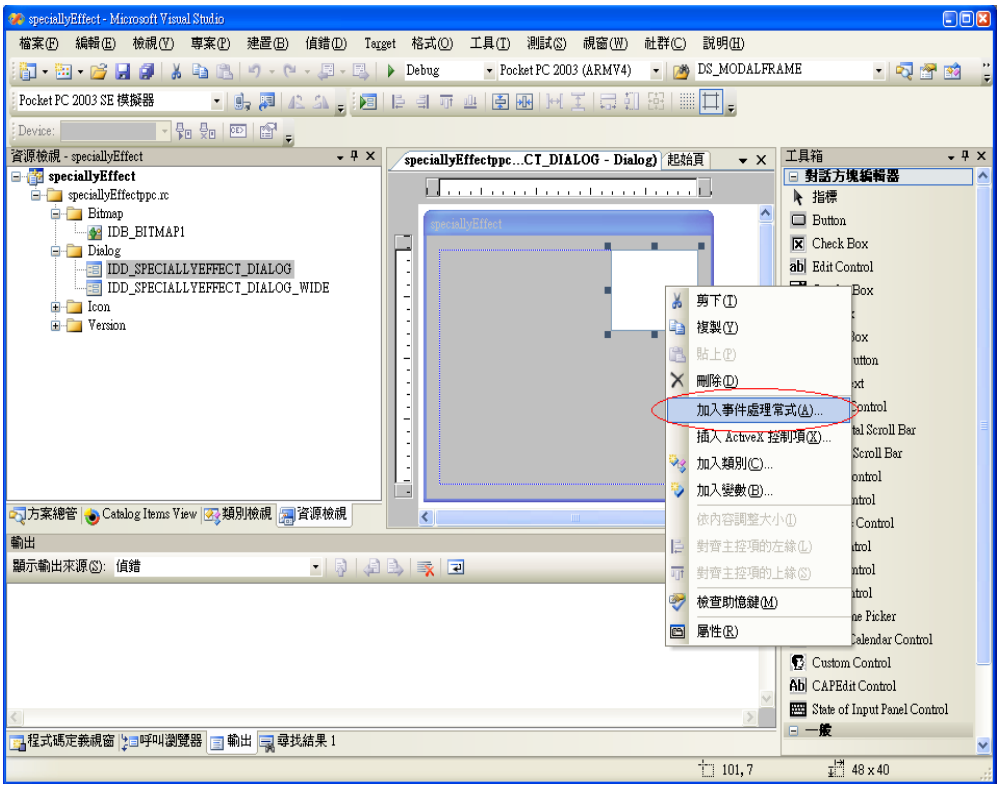
B. 在方案總管視窗中的原始程式檔內，開啟 **speciallyEffectDlg.cpp** ，在 **BOOL CspeciallyEffectDlg::OnInitDialog()** 後，加入一段程式碼。



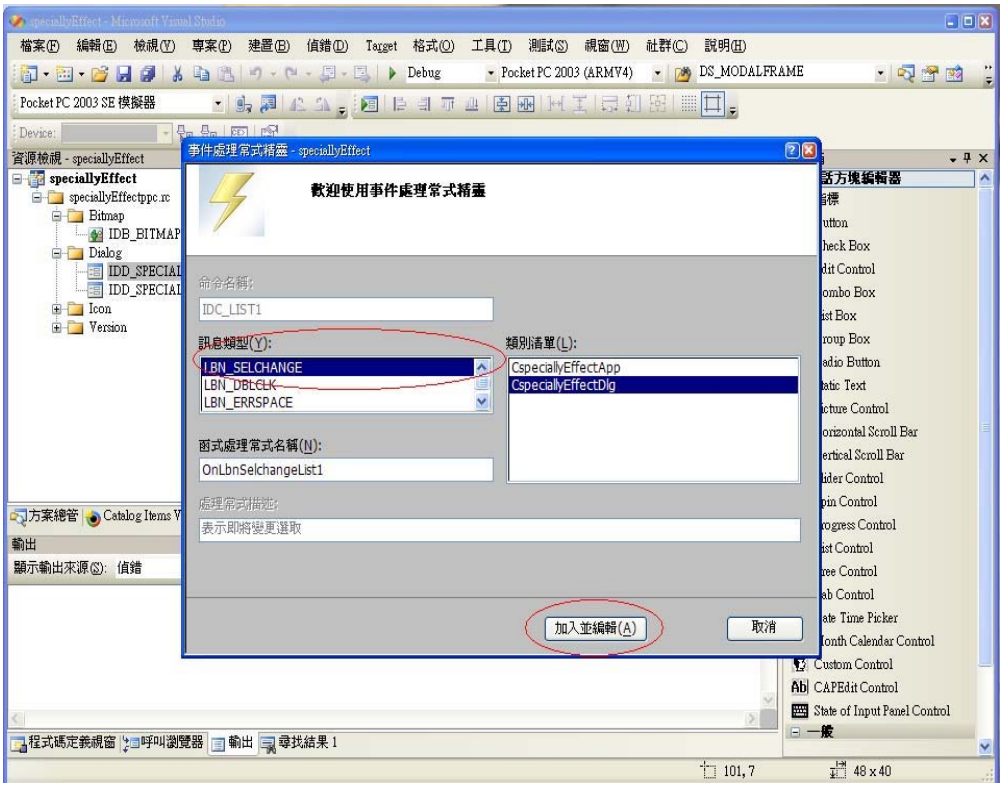
加入的程式碼 ：

```
//  將列表框添加所需要的項目

CListBox * pListBox;

//  給定一個控制元的識別代碼以取得該對話盒控制元的視窗代碼
pListBox = (CListBox*)GetDlgItem(IDC_LIST1);

//  在ListBox 中添加項目
pListBox->AddString(_T("水平向右掃描"));
pListBox->AddString(_T("水平向左掃描"));
pListBox->AddString(_T("水平百葉窗"));
pListBox->AddString(_T("垂直百葉窗"));
pListBox->AddString(_T("雨滴效果"));
```

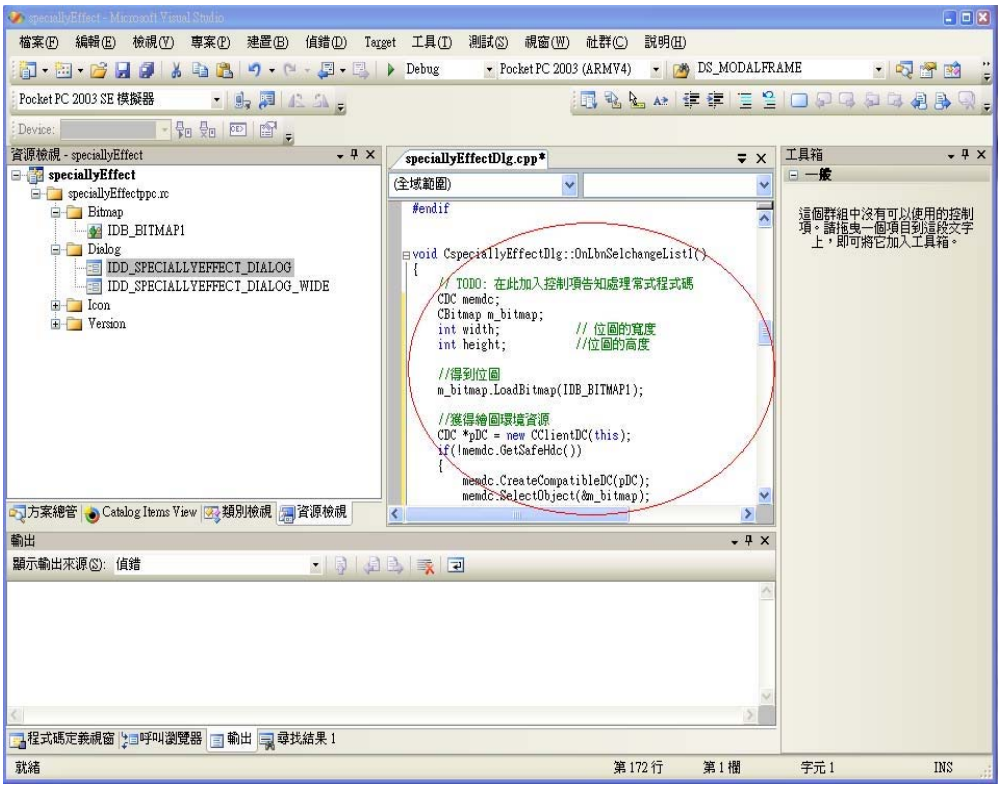
C. 切換左方視窗到資源檢視，準備開始加入事件處理常式。在要加入事件 處理常式的對話方塊上按下滑鼠右鍵，選擇［加入事件處理常式］。



D. 選取事件訊息類型後，按下［加入並編輯］按鈕。



E. 程式編輯視窗開啟在**speciallyEffectDlg.cpp**，在**void CspeciallyEffectDlg::OnLbnSelchangeList1()** 後，加入事件處理常式程式碼。



加入的程式碼 ：

```
//  定義位元圖繪製環境
CDC memdc;
CBitmap m_bitmap;

//  定義位元圖的寬度
int width;

//  定義位元圖的高度
int height;

//  將位元映射式資源載入記憶體中
m_bitmap.LoadBitmap(IDB_BITMAP1);

//  獲得繪圖環境資源

// CClientDC  使用只代表視窗工作區的裝置內容
// new CClientDC  取得視窗工作區的裝置內容
CDC *pDC = new CClientDC(this);

//   判斷是否得到對應窗口的設備描述表
if(!memdc.GetSafeHdc())
{
    //  建立一個相容的記憶體裝置
    memdc.CreateCompatibleDC(pDC);

    //  選擇一個物件到一個裝置環境
    memdc.SelectObject(&m_bitmap);
}

//  獲取位圖大小訊息

BITMAP bm;

// GetBitMap  來查詢DDB （Device Dependent Bitmap）的各種屬性
m_bitmap.GetBitmap(&bm);
width=bm.bmWidth;
```

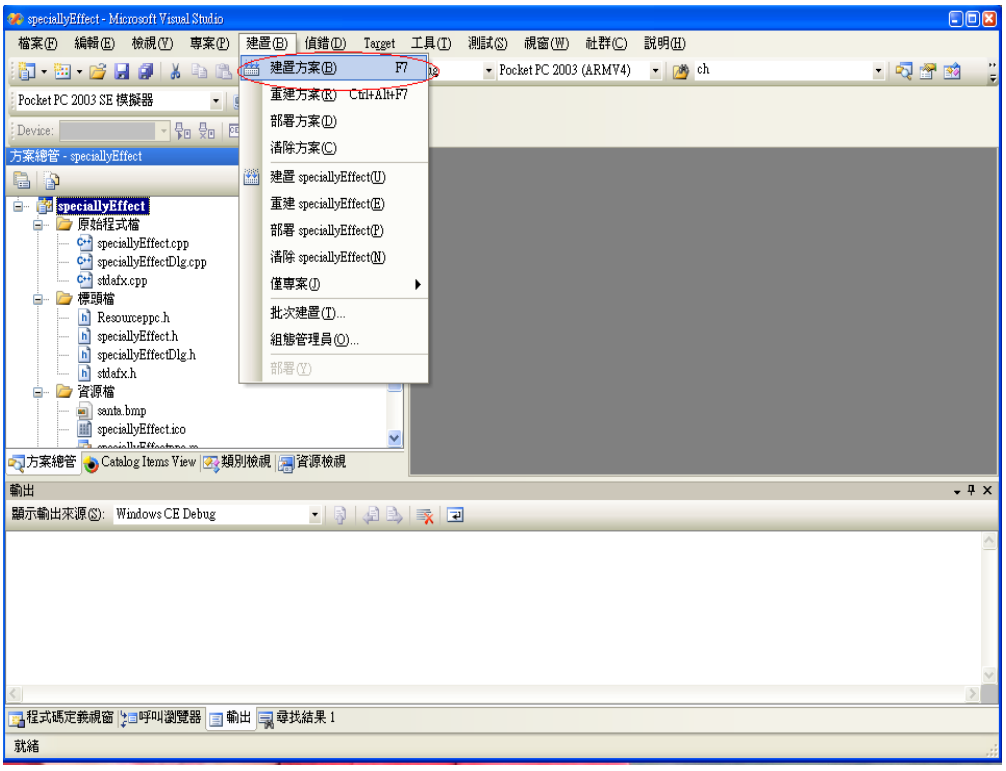
```
height=bm.bmHeight;

//  在選擇視窗工作區的裝置內容上繪製矩形
pDC->Rectangle(0,0,width,height);

CListBox * pListBox;

//  給定一個控制元的識別代碼以取得該對話盒控制元的視窗代碼
pListBox = (CListBox*)GetDlgItem(IDC_LIST1);

//  用以得到用戶選中下拉列表框中數據的索引值
int nIndex = pListBox->GetCurSel();

//  繪圖事件處理
switch(nIndex)
{
case 0: //  水平向左掃描
    {
        for(int i=0;i<width;i++)
        {
            //  把螢幕設備描述表拷貝到內存設備描述表中
            pDC->BitBlt(i,0,1,height,&memdc,i,0,SRCCOPY);

            Sleep(10);
        }
        break;
    }
case 1: //  水平向右掃描
    {

        for(int i=width-1;i>=0;i--)
        {
            //  把螢幕設備描述表拷貝到內存設備描述表中
            pDC->BitBlt(i,0,1,height,&memdc,i,0,SRCCOPY);

            Sleep(10);
        }
        break;
```

```
    }
case 2: //  水平百葉窗
    {
        //每條20 像素寬
        int num=width/20;
        for(int i=0;i<20;i++)
        {
            //  分別掃描每條
            for(int j=0;j<num+1;j++)
            {
                //  把螢幕設備描述表拷貝到內存設備描述表中
                pDC->BitBlt(j*20+i,0,1,height,&memdc,j*20+i,0,SRC
                COPY);

                Sleep(10);
            }
            Sleep(1);
        }
        break;
    }
case 3: //  垂直百葉窗
    {
        int num=height/20;
        for(int i=0;i<20;i++)
        {
            //  分別掃描每條
            for(int j=0;j<num+1;j++)
            {
                //  把螢幕設備描述表拷貝到內存設備描述表中
                pDC->BitBlt(0,j*20+i,width,1,&memdc,0,j*20+i,SRCC
                OPY);

                Sleep(10);
            }
        Sleep(1);
      }
    break;
  }
```

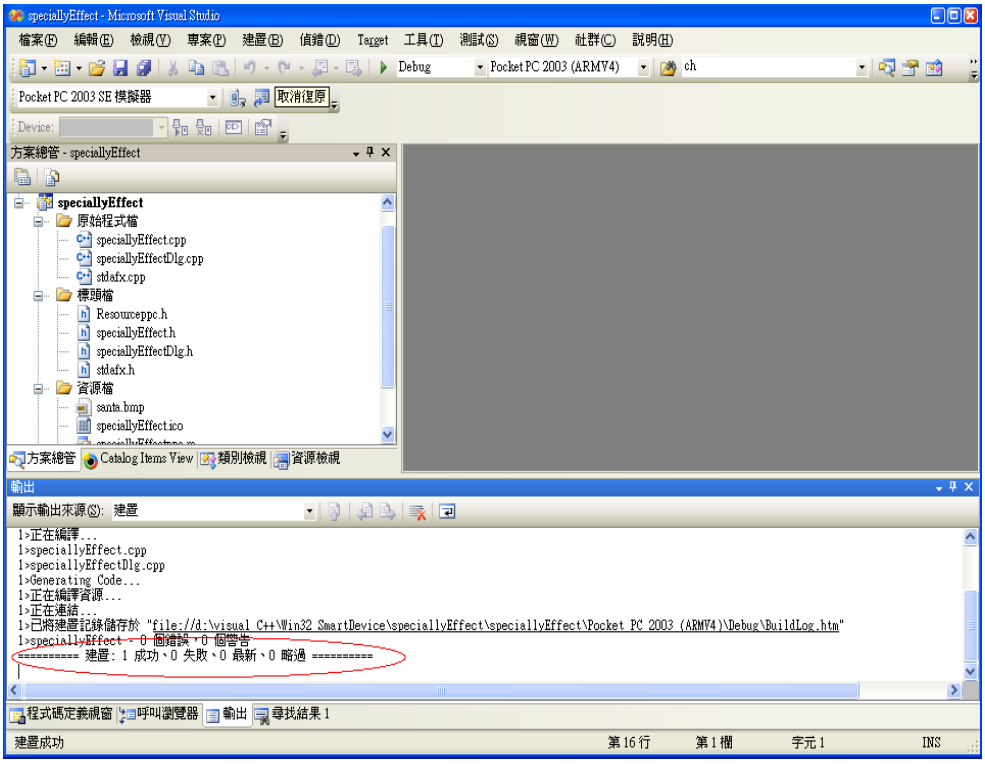
```
case 4: //  雨滴效果
    {
        for(int i=height-1;i>=0;i--)
        {
            for(int j=0;j<i;j++)
            {
                //  把螢幕設備描述表拷貝到內存設備描述表中
                pDC->BitBlt(0,j,width,1,&memdc,0,i,SRCCOPY);
                Sleep(1);
            }
            Sleep(1);
        }
        break;
    }
}

//  從記憶體中刪除筆、畫筆、字型和調色盤 …
m_bitmap.DeleteObject();

//  刪除記憶體中繪圖的裝置環境
memdc.DeleteDC();

//  刪除記憶體中pDC  空間
delete pDC;
```

4. 編譯和執行

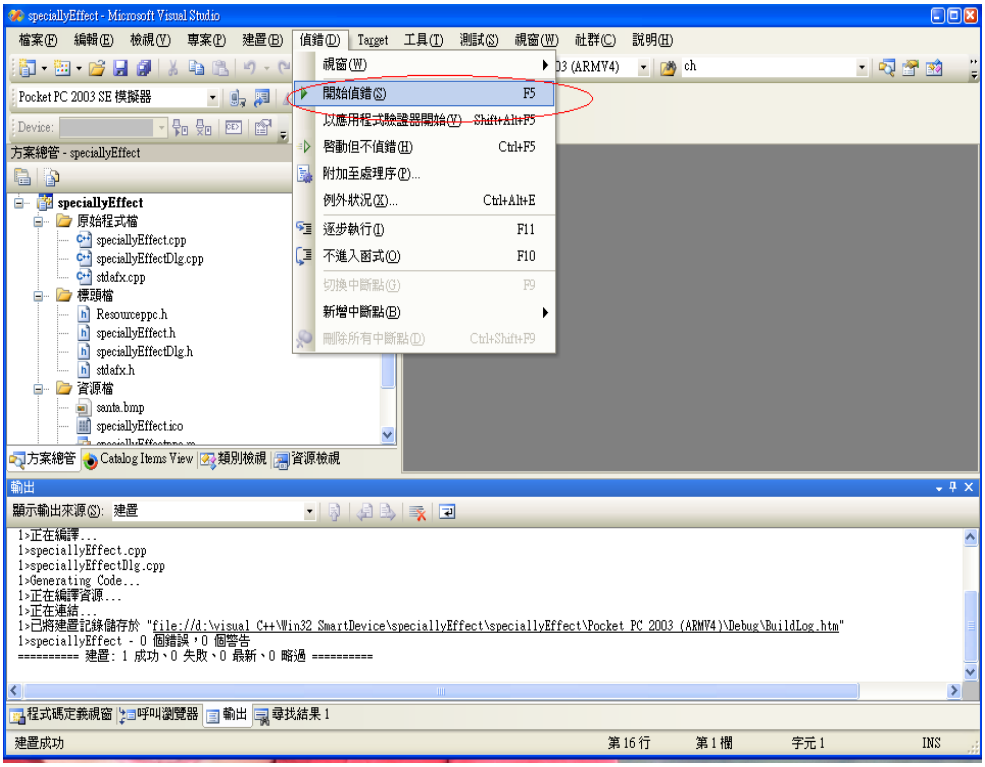
A. 在**Visual Studio 2005** 整合式開發環境中，點選[建置] -> [建置專案]後便會開始編譯整個專案。



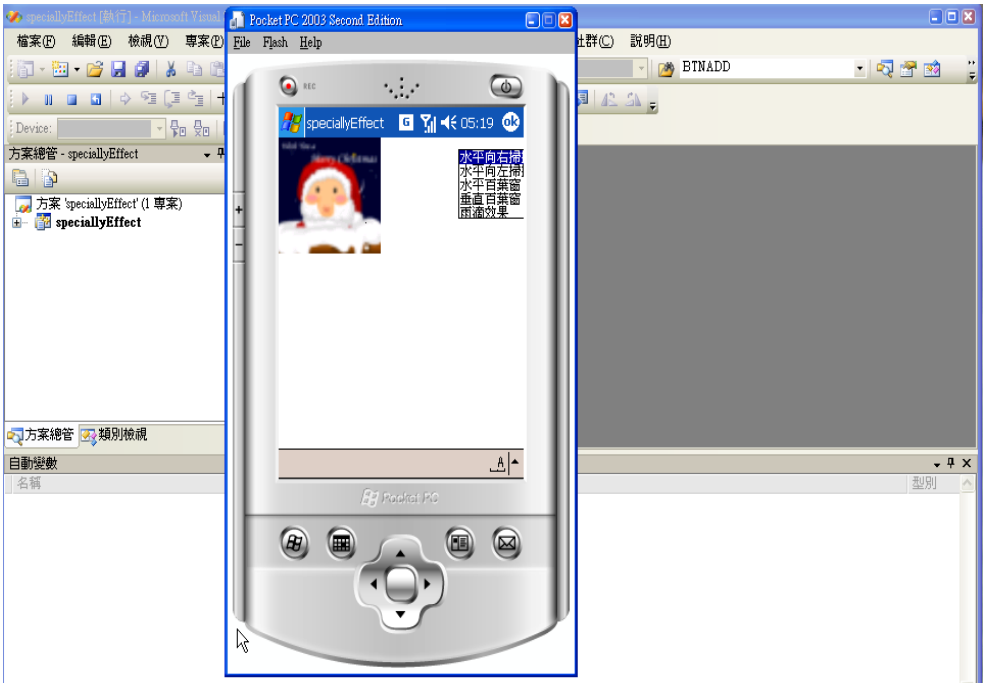
B. 編譯成功後，在**Visual Studio 2005** 整合式開發環境下方的輸出視窗中顯示建置成功。



C. 在**Visual Studio 2005** 整合式開發環境中，點選[偵錯] -> [開始偵錯]後便會開始對此專案執行並偵錯。


speciallyEffect - Microsoft Visual Studio
開始偵錯(S) F5
以應用程式驗證器開始(Y) Shift+Alt+F5
啓動但不偵錯(H) Ctrl+F5
附加至處理序(P)...
例外狀況(X)... Ctrl+Alt+E
逐步執行(I) F11
不進入函式(O) F10
新增中斷點(B)
speciallyEffect.cpp
speciallyEffectDlg.cpp
stdafx.cpp
Resourceppc.h
speciallyEffect.h
speciallyEffectDlg.h
stdafx.h
santa.bmp
speciallyEffect.ico
1>正在編譯...
1>speciallyEffect.cpp
1>speciallyEffectDlg.cpp
1>Generating Code...
1>正在編譯資源...
1>正在連結...
1>已將建置記錄儲存於 "file://d:\visual C++\Win32 SmartDevice\speciallyEffect\speciallyEffect\Pocket PC 2003 (ARMV4)\Debug\BuildLog.htm"
1>speciallyEffect - 0 個錯誤，0 個警告
========== 建置: 1 成功、0 失敗、0 最新、0 略過 ==========
建置成功

D. 執行結果如下圖所示。


Pocket PC 2003 Second Edition
File Flash Help
speciallyEffect
05:19
水平向右捲
水平向左捲
水平百葉窗
垂直百葉窗
雨滴效果
方案 'speciallyEffect' (1 專案)
speciallyEffect

應用程式開發實驗

實驗三

處理程序建立

Rev. 1.0

一. 實驗目的： 每個應用程序自動啟動後，就會變成一個單獨的處理序，並且每個處理序都 有自己的虛擬內存空間。作業系統可以列舉系統進行的處理序，並且可以根 據處理序的狀態來「終止處理序」和「啟動處理序」等操作。

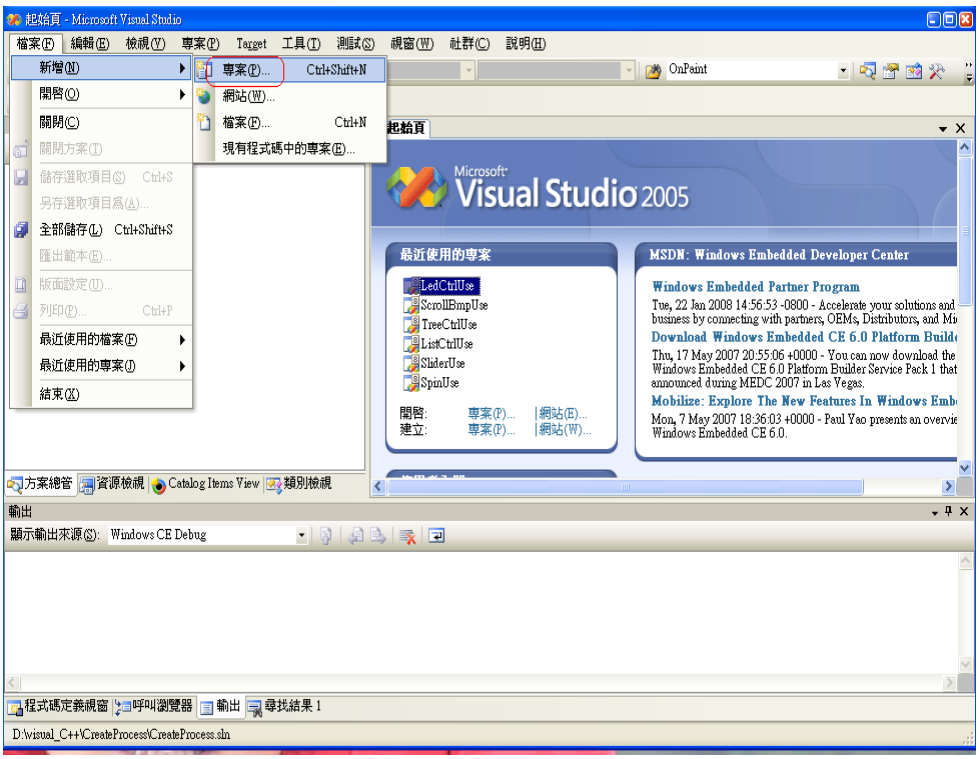
處理序（**Process**）是指當前所加載程序的專業 **Win32** 術語，例如：磁碟上 的任何一個可執行檔案，僅僅是一個檔案，它只有在被啟動後才是一個處理序。處理序僅僅是存在的，它不做認何事。一個處理序可以有多個執行緒（**Thread**）但至少應包含一個執行緒。處理序中所有的操作都是由執行緒來完成的。同時每一個處理序有且僅有一個主執行緒，由主執行緒來進行所有初始化操作。

WinCE 的處理序中任一時刻最多可以有**32** 個處理序同時運行。當**WinCE** 系統啟動時，至少會啟動**4** 個處理序：**NK.EXE** 提供內核服務；**FILESYS.EXE** 提供檔案系統服務；**GWES.EXE** 提供**GUI** 服務；**DEVICE.EXE** 提供加載和維護設備驅動程序。

二. 實驗步驟：

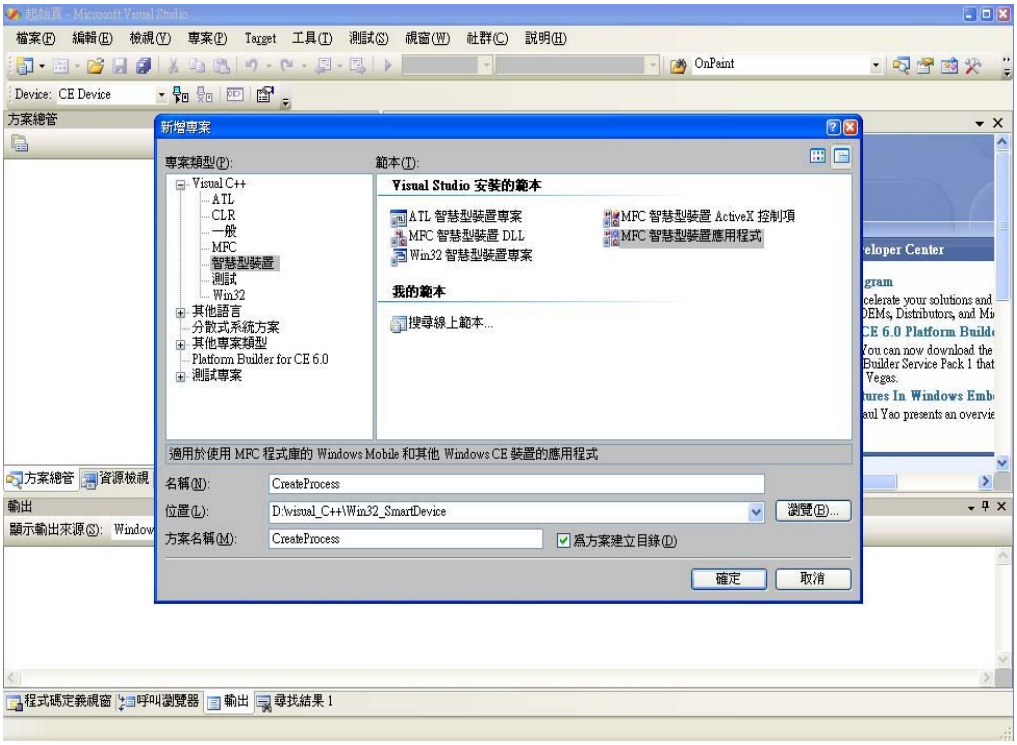
1. 建立一個 **MFC** 對話方塊為基礎的專案

A. 在**Visual Studio 2005** 整合式開發環境中，點選左上角[檔案]->[新增]->[專案]後便會開啟新增專案視窗。

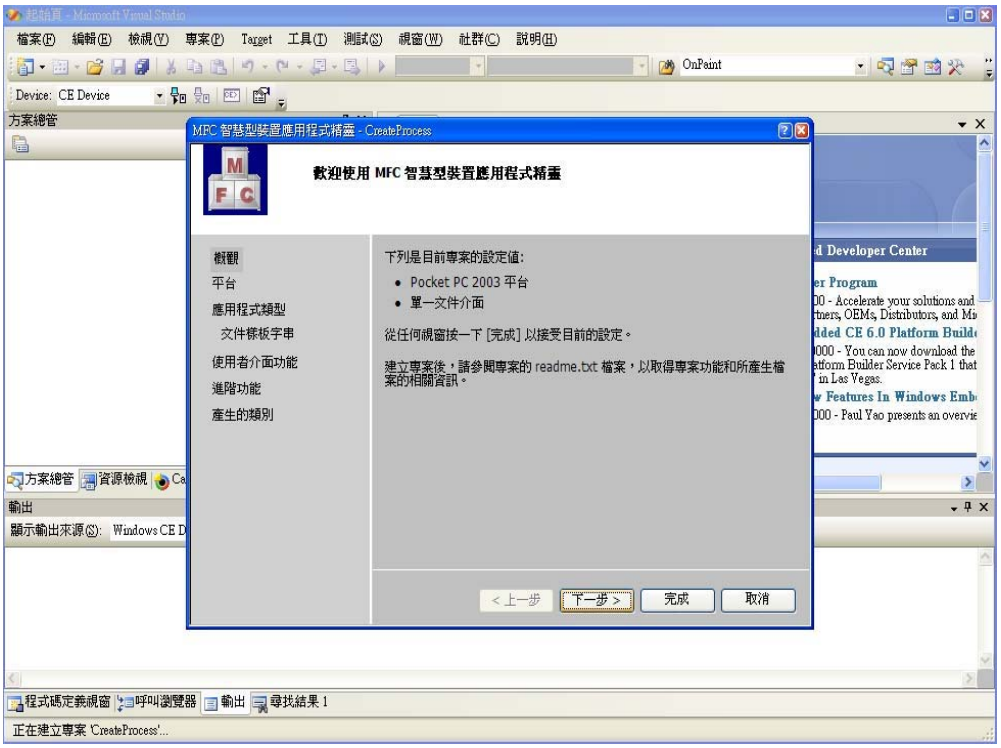


B. 在新增專案視窗內，左邊專案類型欄中選擇[智慧型裝置]；右邊 **Visual**

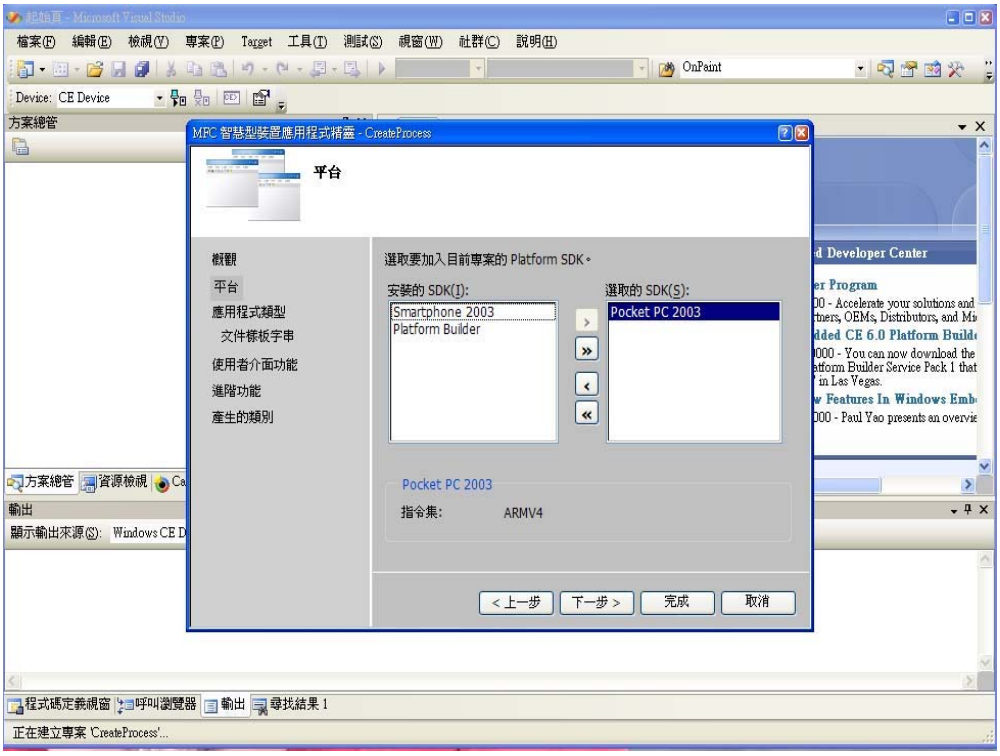
Studio　安裝的範本欄中選擇[**MFC**　智慧型裝置應用程式]。



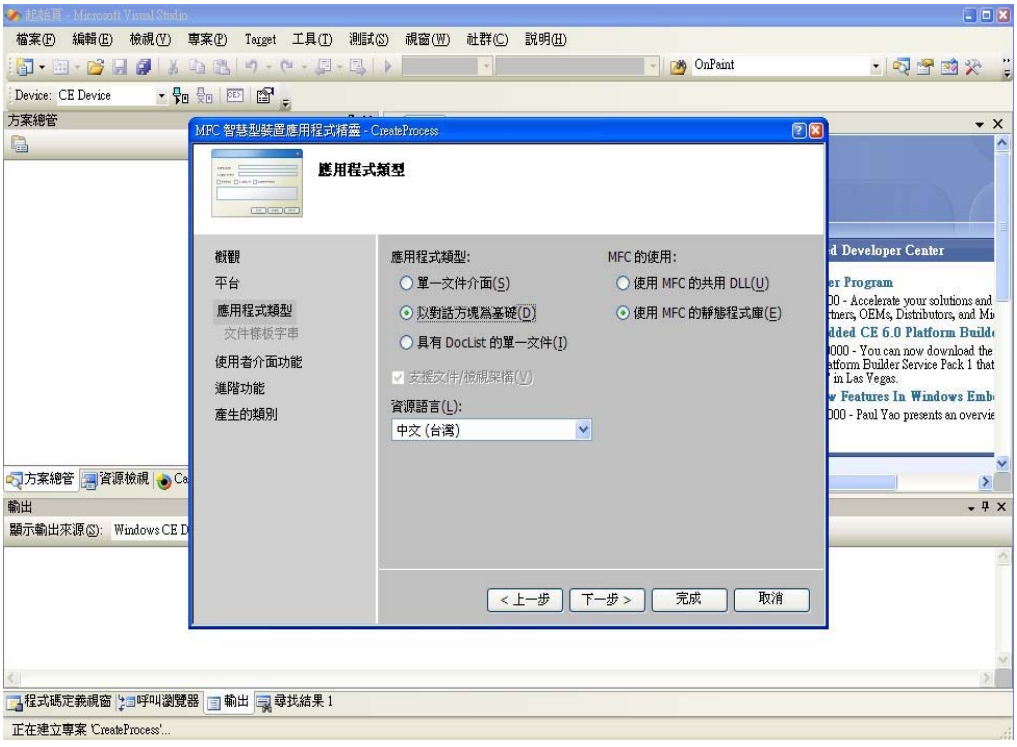
C. 在 **MFC**　智慧型裝置應用程式精靈對話窗中選擇［下一步］按鈕。



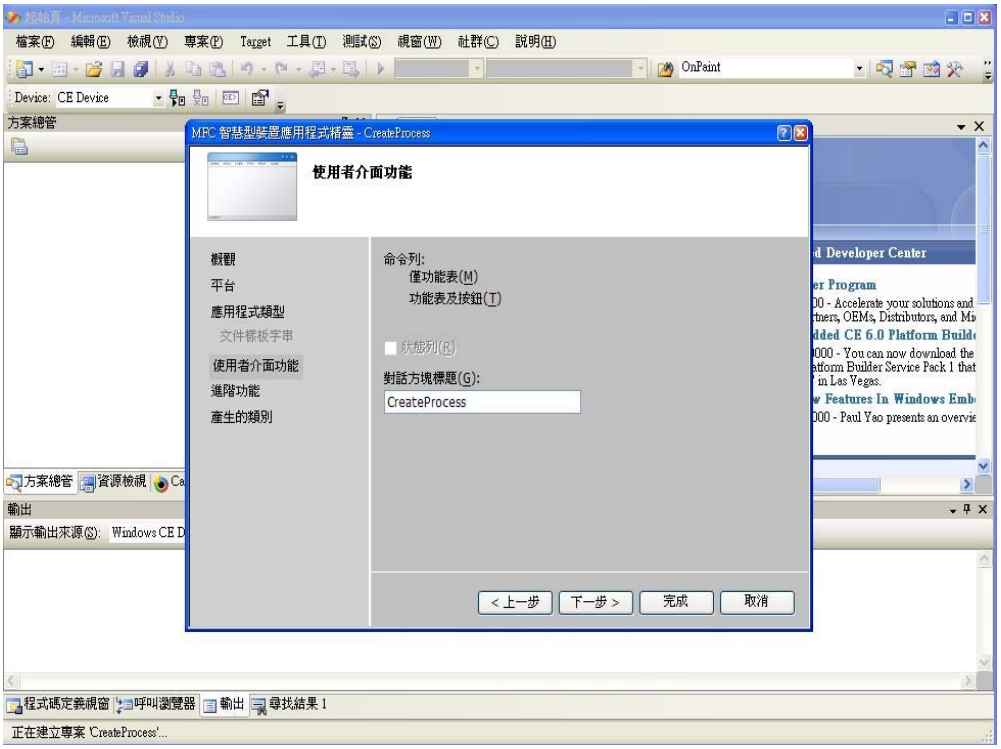
D. 選取要加入目前專案的平台軟體開發包（**Platform SDK**）在此實驗中 我們選擇[**Pocket PC 2003**]，之後按下[下一步］按鈕。



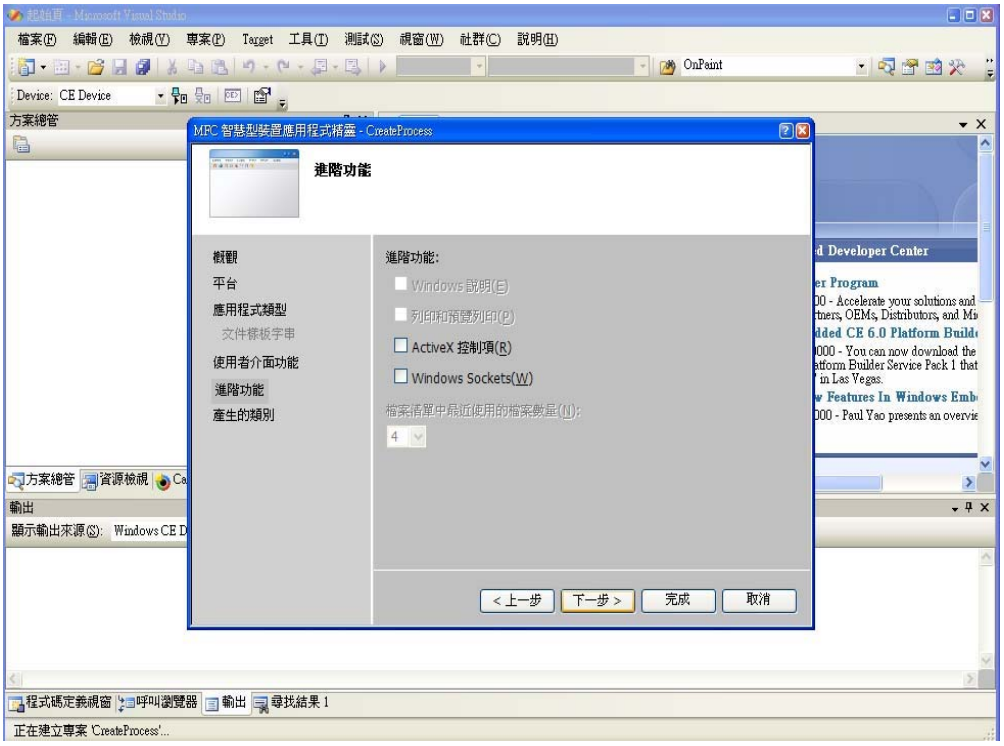
E. 應用程式類型中選取［以對話方塊為基礎］，**MFC** 的使用中選取[使用 **MFC** 的靜態程式庫］，之後按下[下一步］按鈕。



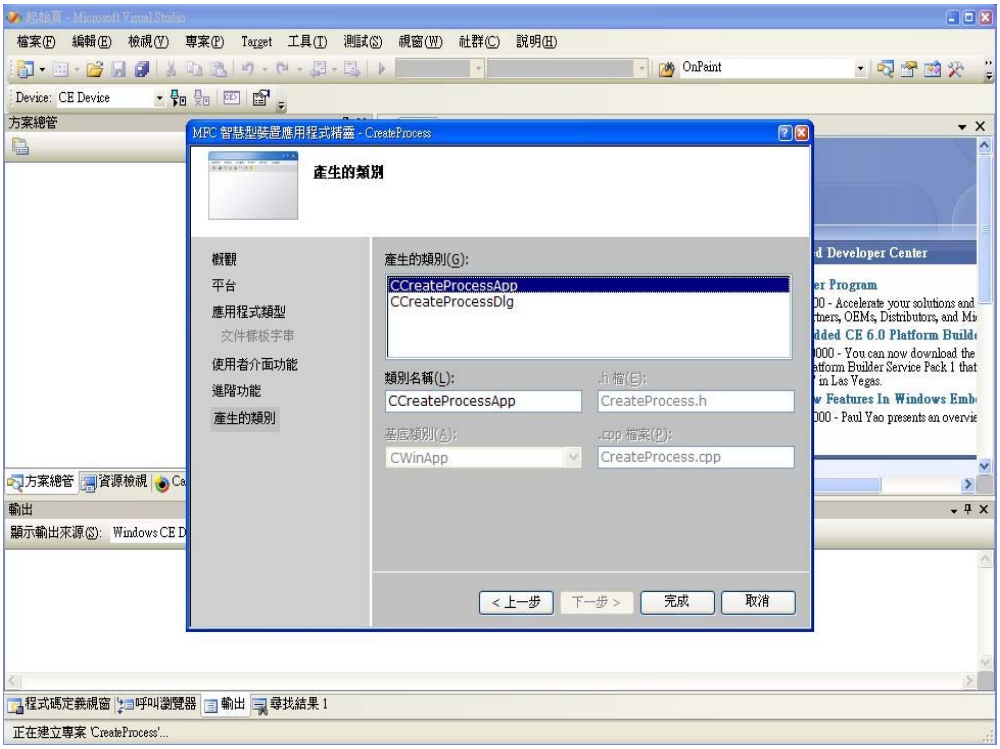
F. 按下[下一步］按鈕。



G. 按下[下一步] 按鈕。

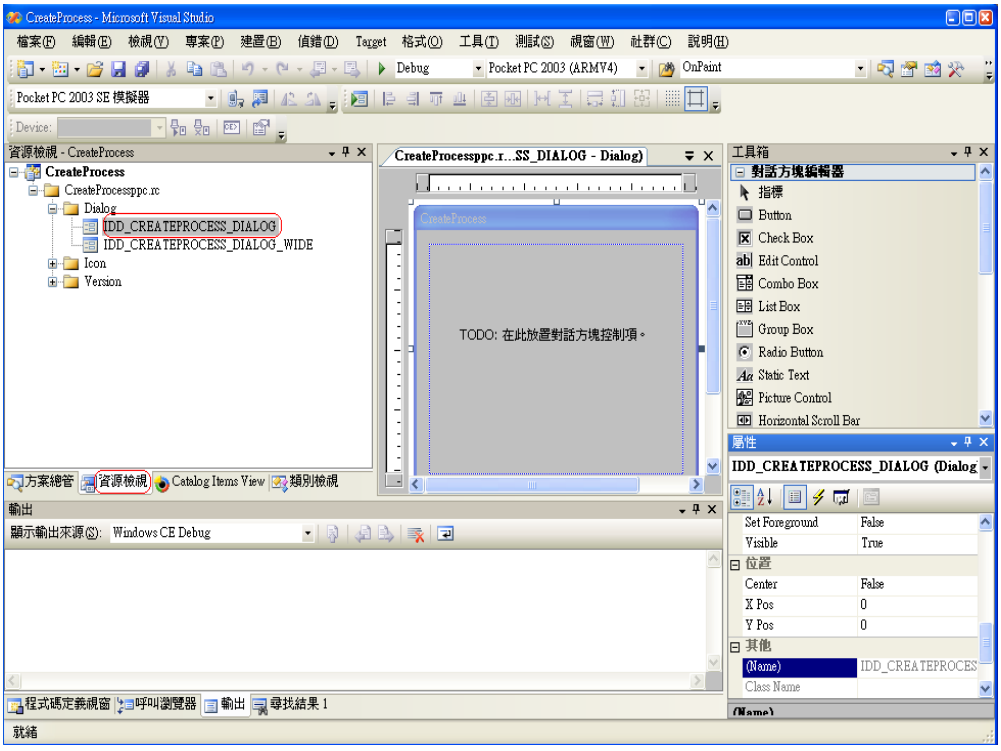


H. 按下[完成] 按鈕。



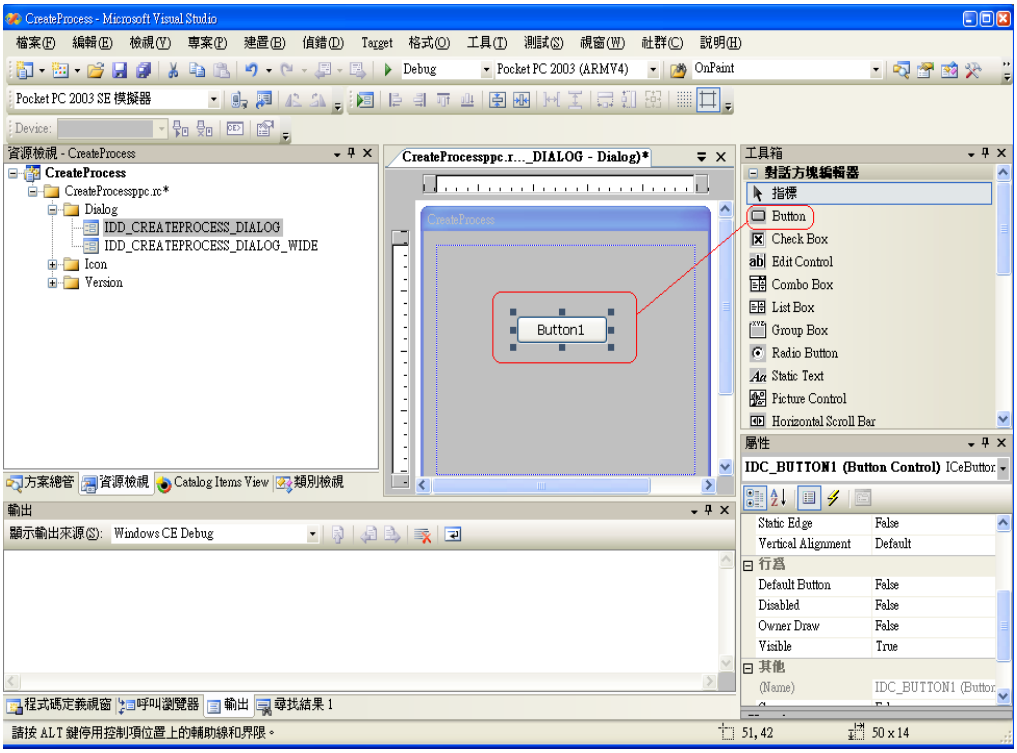
2. 設計使用者界面

A. 切換左方視窗到資源檢視，準備開始設計使用者對話方塊介面（**UI**）

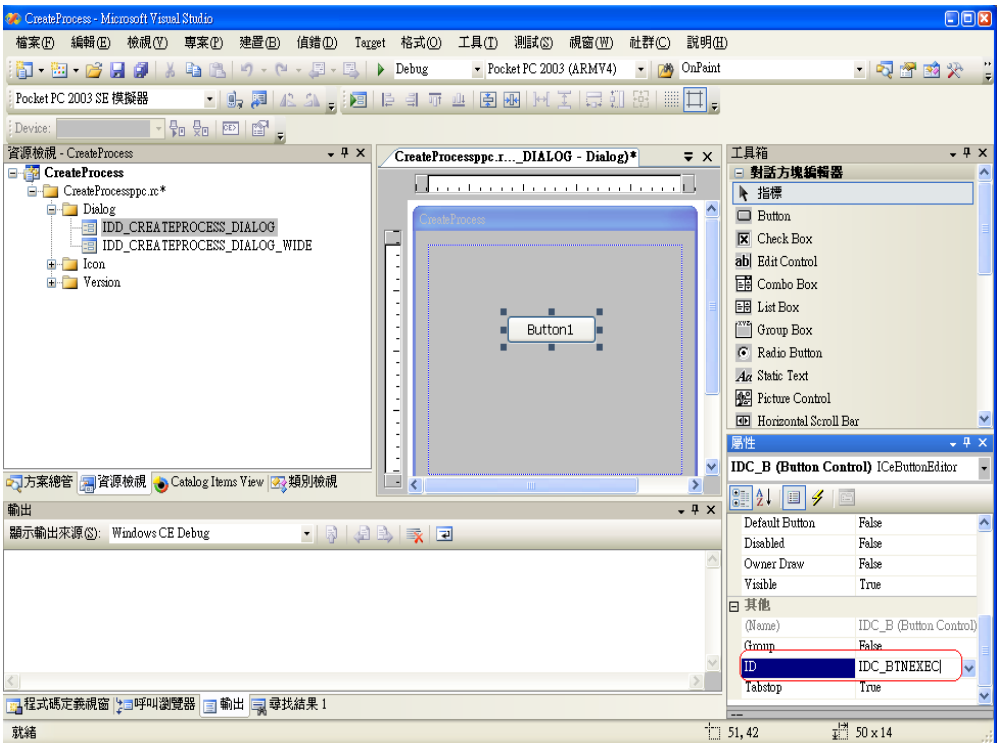


B. 將對話方塊中存在的文字對話方塊 （**TODO**：在此放置對話方塊控制

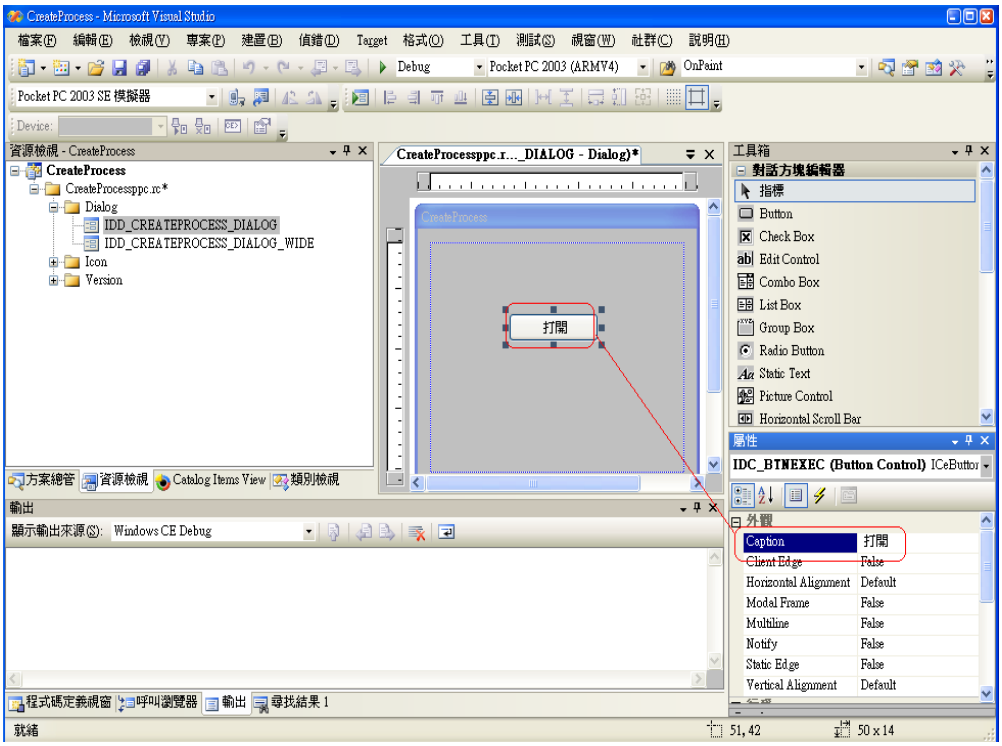
項）刪除，然後將新選取［**Button**］對話方塊拖曳到適當位置。



C. 在 **Button** 上按下滑鼠右鍵，選擇〔屬性〕修改屬性對話窗中 **ID** 項為 **IDC_BTNEXEC**。

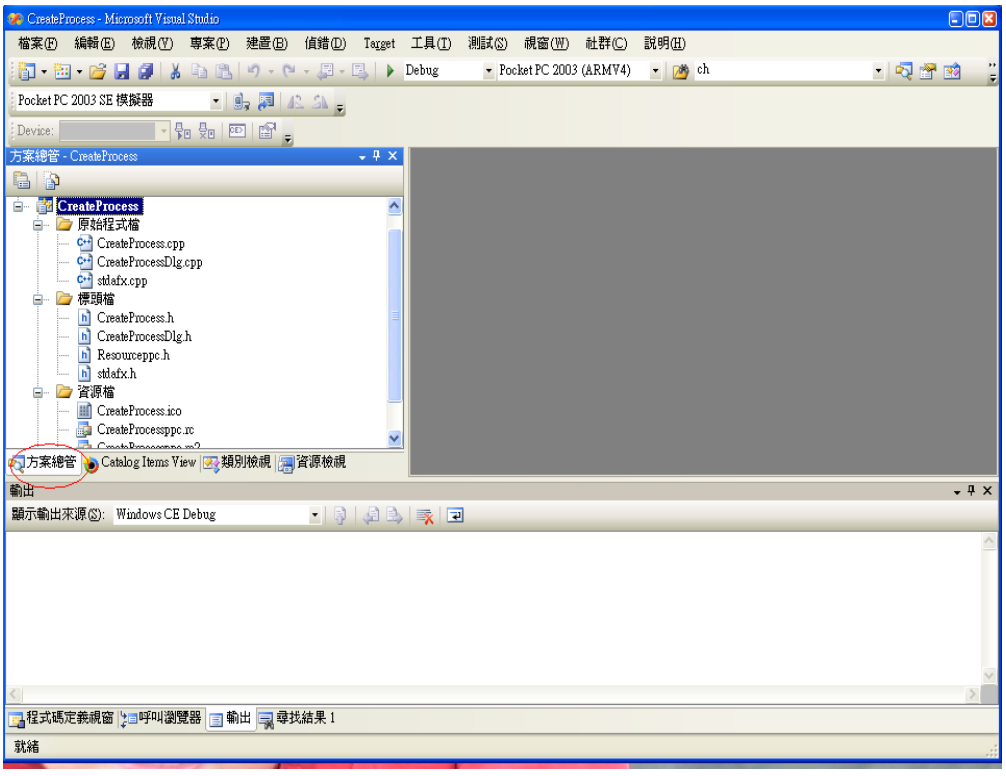


D. 修改屬性對話窗中 **Caption** 項為打開。

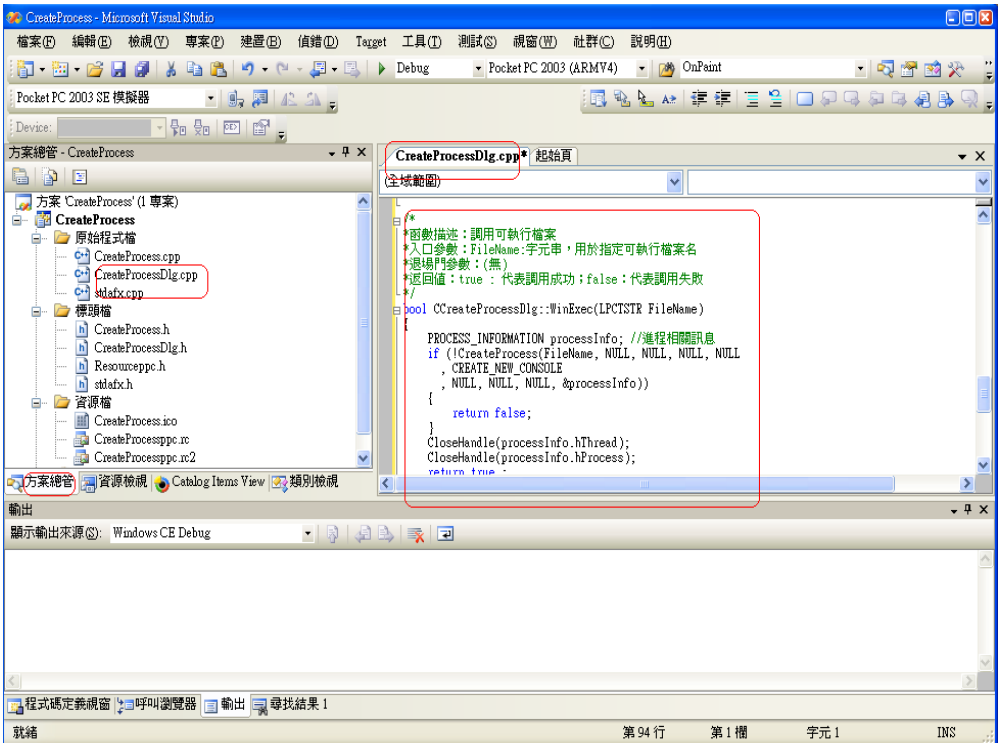


3. 程式設計

A. 切換左方視窗到方案總管，準備開始設計使用者程式。



B. 在方案總管視窗中的原始程式檔內，開啟 **CreateProcessDlg.cpp** ，在原始程式碼最後面加入一段調用可執行檔案函數程式碼。

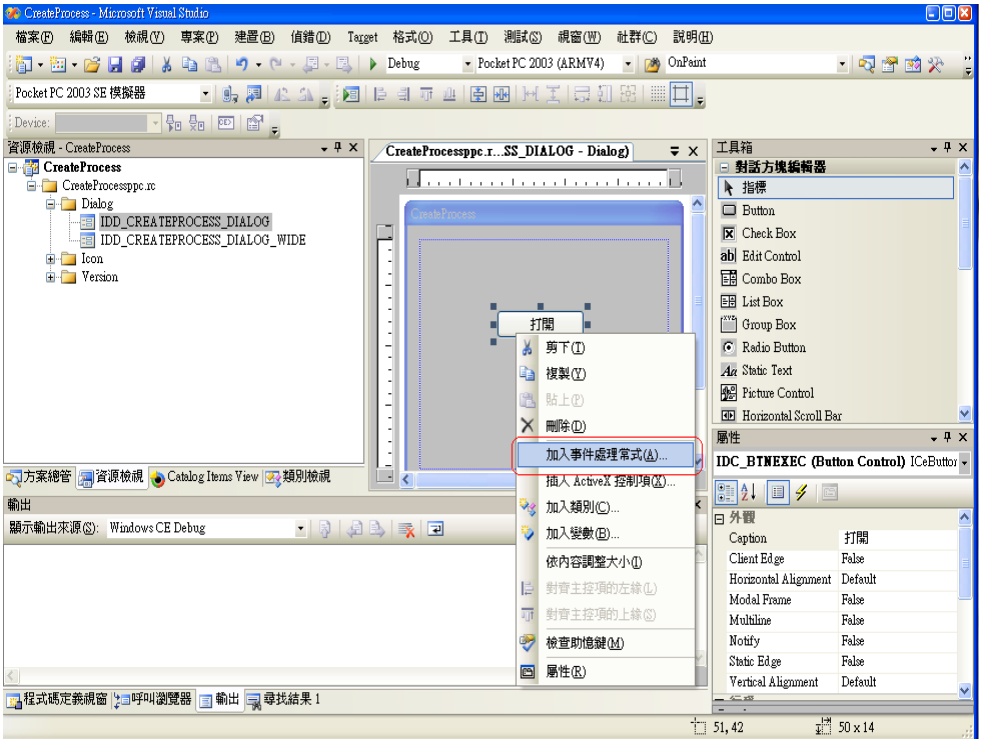


加入的程式碼 ：

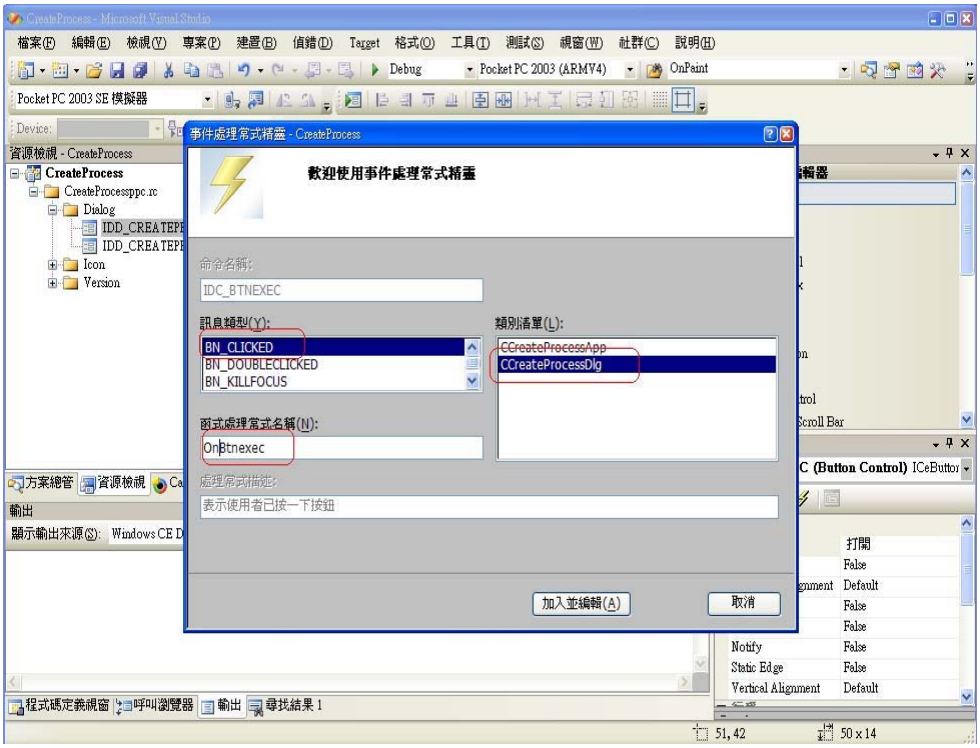
```
bool CCreateProcessDlg::WinExec(LPCTSTR FileName)
{
    //  進程相關訊息
    PROCESS_INFORMATION processInfo;

    // CreateProcess  處理緒的建立
    // CreateProcess  原型
    // BOOL CreateProcess（
    // LPCWSTR pszImageName,  「指向可執行檔案名的指標」
    // LPCWSTR pszCmdLine,  「指向執行命令字串的指標」
    // LPSECURITY_ATTRIBUTES psaProcess,  「WinCE  不支援」
    // LPSECURITY_ATTRIBUTES psaThread,  「WinCE  不支援」
    // BOOL fInheritHandles,  「WinCE  不支援」
    // DWORD fdwCreate,  「指定處理序被創建後的初始值狀態」
    // PVOID pvEnvironment,  「WinCE  不支援」
    // LPWSTR pszCurDir,  「WinCE  不支援」
    // LPSTARTUPINFOW psiStartInfo,  「WinCE  不支援」
    // LPPROCESS_INFORMATION pProcInfo,  「指向處理序訊息結構的指標」
```

```
//
);
//   WinCE   支援的「指定處理序被創建後的初始值狀態」
// CREATE_NEW_CONSOLE：強制建立一個新的控制
// CREATE_SUSPENDED：建立處理序，然後掛起最初的執行緒，
                              直到使用Resume Thread   函數喚醒它
// DEBUG_PROCESS：已建立的處理序被看作由調用者進行調試
                              的處理序，調用的處理序接收自己運行處
                              理序的調試訊息
//  DEBUG_ONLYTHIS_PROCESS：當與DEBUG_PROCESS
                                        組 合時，將調試某個處理
                                        序， 但不測試由被調試處
                                        理序啟 動的任何子處理序
// PROCESS_INFORMATION   結構定義
// typedef struct _PROCESS_INFORMATION{
// HANDLE hProcess;
// HANDLE hThread;
// DWORD dwProcessId;
// DWORD dwThreadId;
// } PROCESS_INFORMATION;

if (!CreateProcess(FileName, NULL, NULL, NULL, NULL
   , CREATE_NEW_CONSOLE , NULL, NULL, NULL, &processInfo))
{
     return false;
}

//   關閉一個打開的執行緒或處理序等內核對象
CloseHandle(processInfo.hThread);
CloseHandle(processInfo.hProcess);
return true ;
}
```

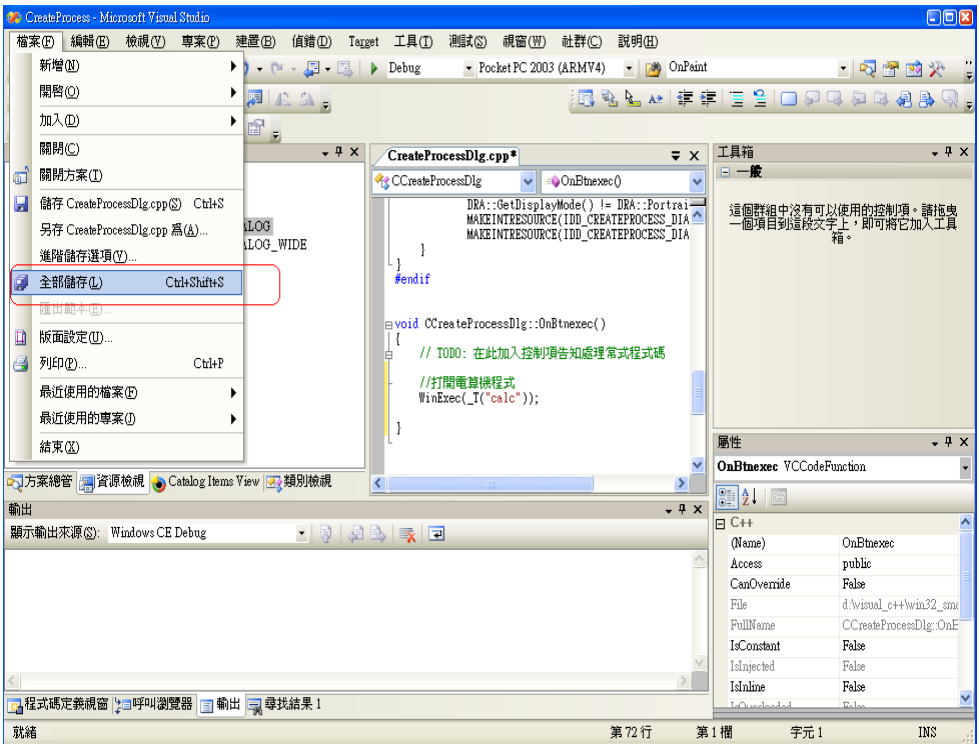
C. 切換左方視窗到資源檢視，準備開始加入事件處理常式。在要加入事件處理常式的**Button** 對話方塊上按下滑鼠右鍵，選擇［加入事件處理常式］。



D. 選取事件訊息類型後，按下［加入並編輯］按鈕。



E. 程式編輯視窗開啟在**CreateProcessDlg.cpp**，在**void CCreateProcessDlg::OnBtnexec()**後，加入事件處理常式程式碼。



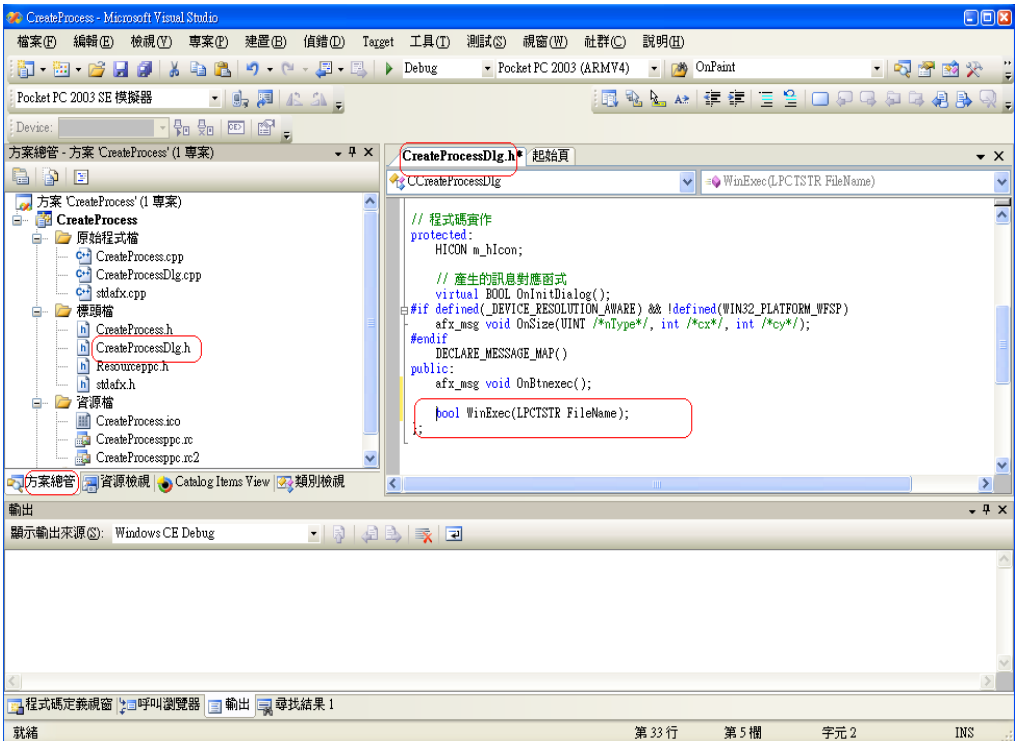
加入的程式碼 ：

// 呼叫小算盤應用程序

// **WinExec**：執行並載入一個 **Window** 的程式

WinExec(_T(“calc”));

F. 在方案總管視窗中的標頭檔內，開啟 **CreateProcessDlg.h** ，在 **CreateProcessDlg.h** 中加入一段程式碼

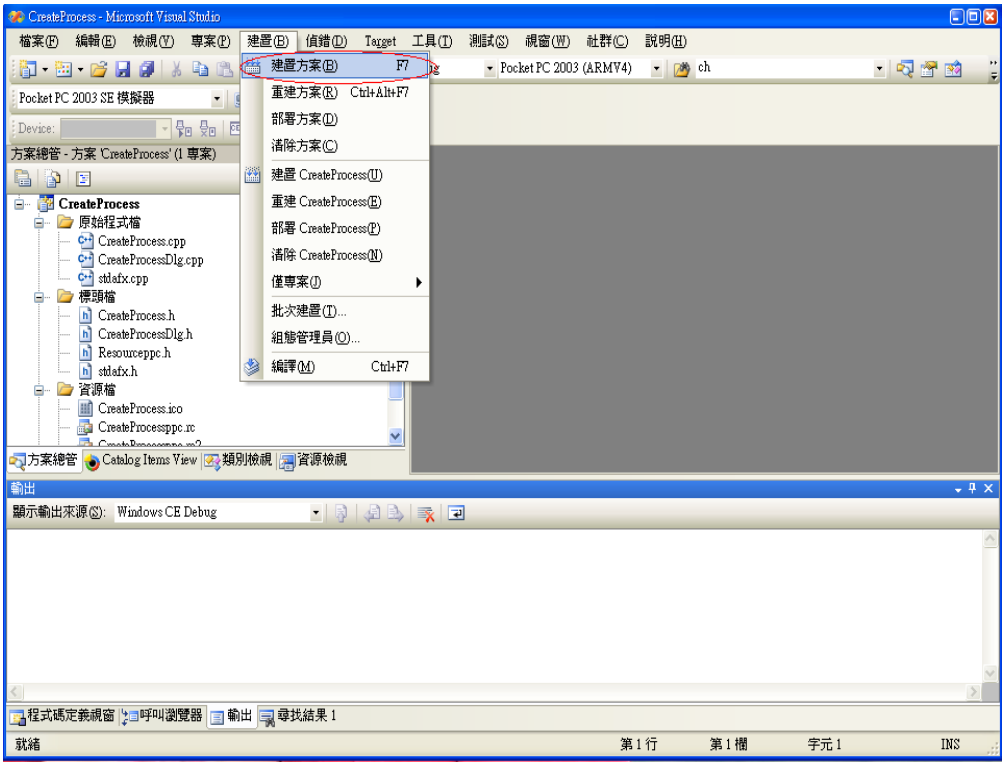


加入的程式碼 ：

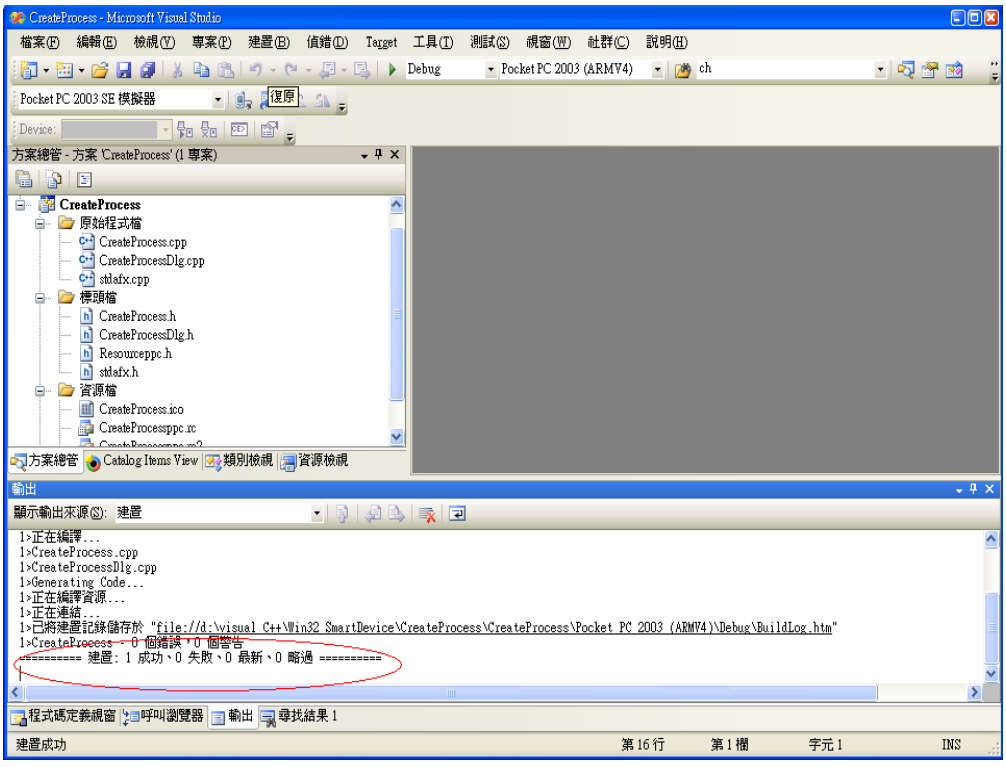
bool WinExec(LPCTSTR FileName);

4. 編譯和執行

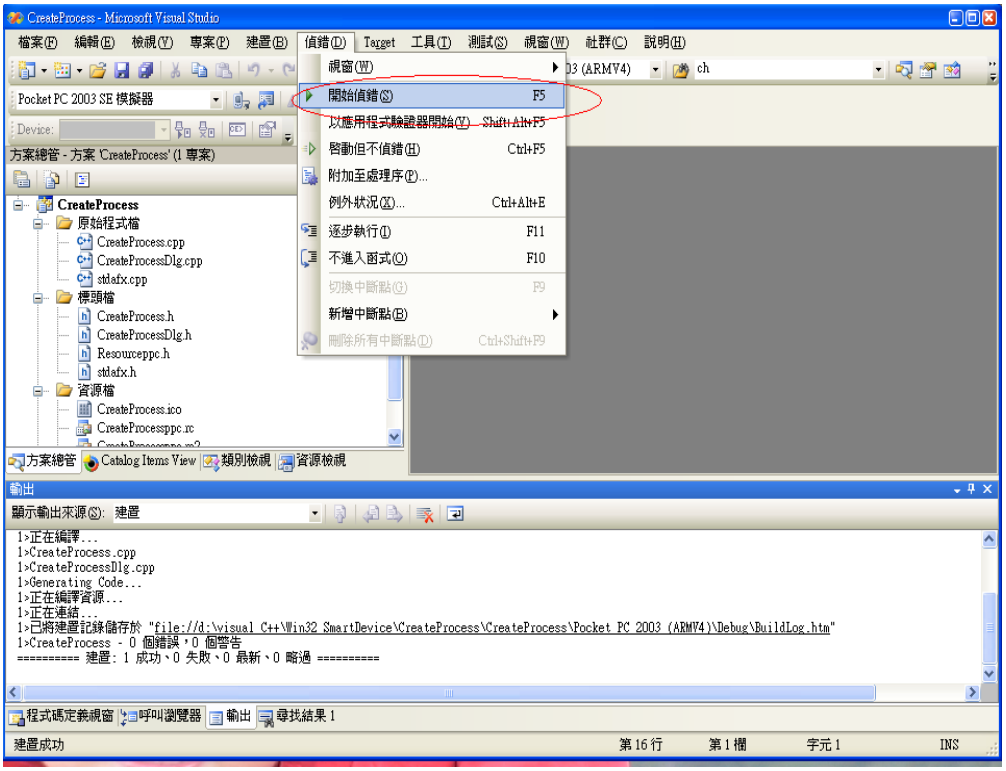
A. 在**Visual Studio 2005** 整合式開發環境中，點選[建置] -> [建置專案]後便會開始編譯整個專案。



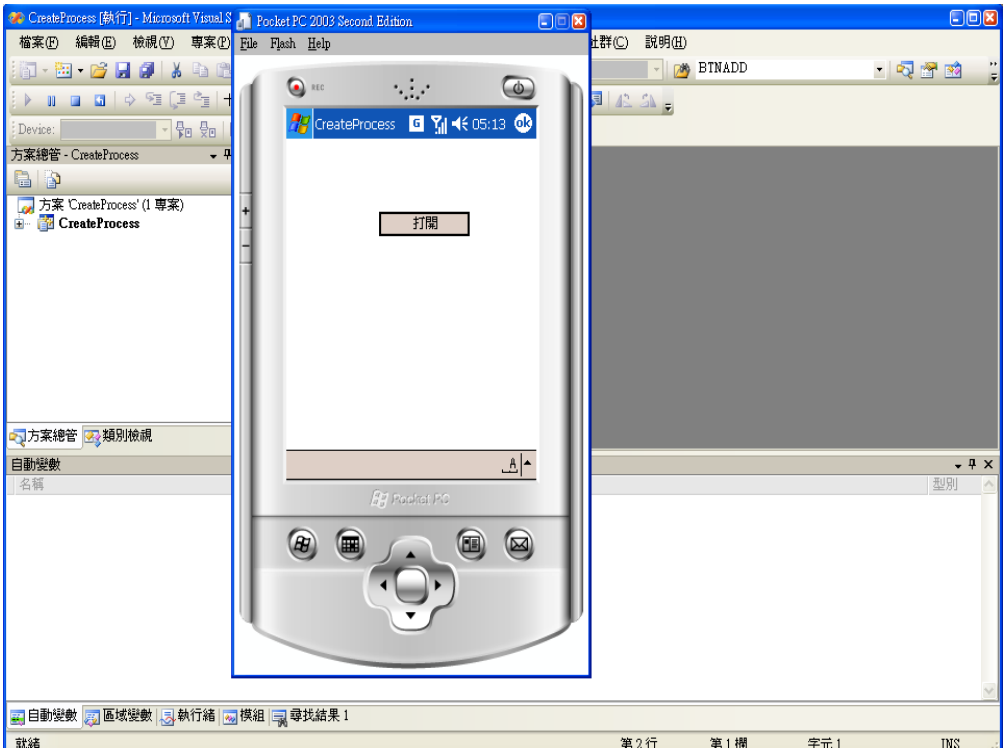
B. 編譯成功後，在**Visual Studio 2005** 整合式開發環境下方的輸出視窗中顯示建置成功。

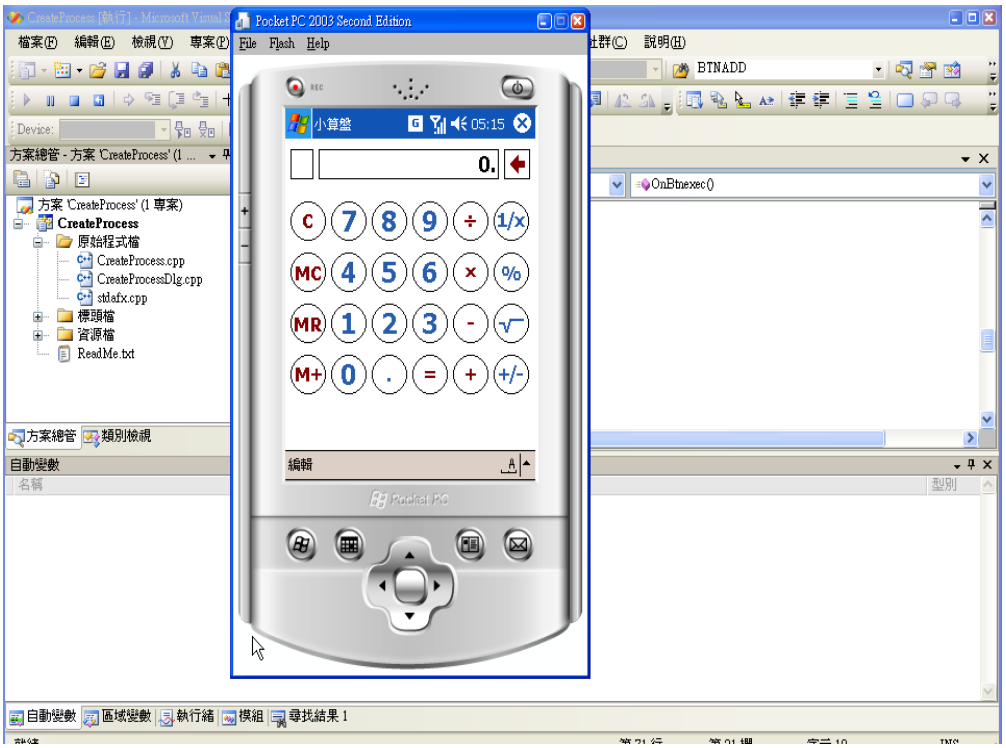


C. 在**Visual Studio 2005** 整合式開發環境中，點選[偵錯] -> [開始偵錯]後便會開始對此專案執行並偵錯。


CreateProcess - Microsoft Visual Studio
檔案(F) 編輯(E) 檢視(V) 專案(P) 建置(B) 偵錯(D) Target 工具(T) 測試(S) 視窗(W) 社群(C) 說明(H)
開始偵錯(S) F5
啓動但不偵錯(H) Ctrl+F5
附加至處理程序(P)...
例外狀況(X)... Ctrl+Alt+E
逐步執行(I) F11
不進入函式(O) F10
新增中斷點(B)
Pocket PC 2003 SE 模擬器
方案總管 - 方案 'CreateProcess' (1 專案)
CreateProcess
原始程式檔
CreateProcess.cpp
CreateProcessDlg.cpp
stdafx.cpp
標頭檔
CreateProcess.h
CreateProcessDlg.h
Resourceppc.h
stdafx.h
資源檔
CreateProcess.ico
CreateProcessppc.rc
輸出
顯示輸出來源(S): 建置
1>正在編譯...
1>CreateProcess.cpp
1>CreateProcessDlg.cpp
1>Generating Code...
1>正在編譯資源...
1>正在連結...
1>已將建置記錄儲存於 "file://d:\visual C++\Win32 SmartDevice\CreateProcess\CreateProcess\Pocket PC 2003 (ARMV4)\Debug\BuildLog.htm"
1>CreateProcess - 0 個錯誤，0 個警告
========== 建置: 1 成功、0 失敗、0 最新、0 略過 ==========
建置成功
第 16 行
第 1 欄
字元 1
INS

D. 執行結果如下圖所示。


Pocket PC 2003 Second Edition
File Flash Help
CreateProcess 05:13
打開
方案總管 - CreateProcess
方案 'CreateProcess' (1 專案)
CreateProcess
自動變數
名稱
型別
BTNADD


Pocket PC 2003 Second Edition
File Flash Help
小算盤
05:15
0.
C 7 8 9 ÷ 1/x
MC 4 5 6 × %
MR 1 2 3 - √
M+ 0 . = + +/-
編輯
Pocket PC
檔案(F) 編輯(E) 檢視(V) 專案(P)
BTNADD
OnBtnexec()
方案總管 - 方案 'CreateProcess' (1 ...
方案 'CreateProcess' (1 專案)
CreateProcess
原始程式檔
CreateProcess.cpp
CreateProcessDlg.cpp
stdafx.cpp
標頭檔
資源檔
ReadMe.txt
方案總管
類別檢視
自動變數
名稱
型別
自動變數
區域變數
執行緒
模組
尋找結果 1

應用程式開發實驗

實驗四

螢幕複製

Rev. 1.0

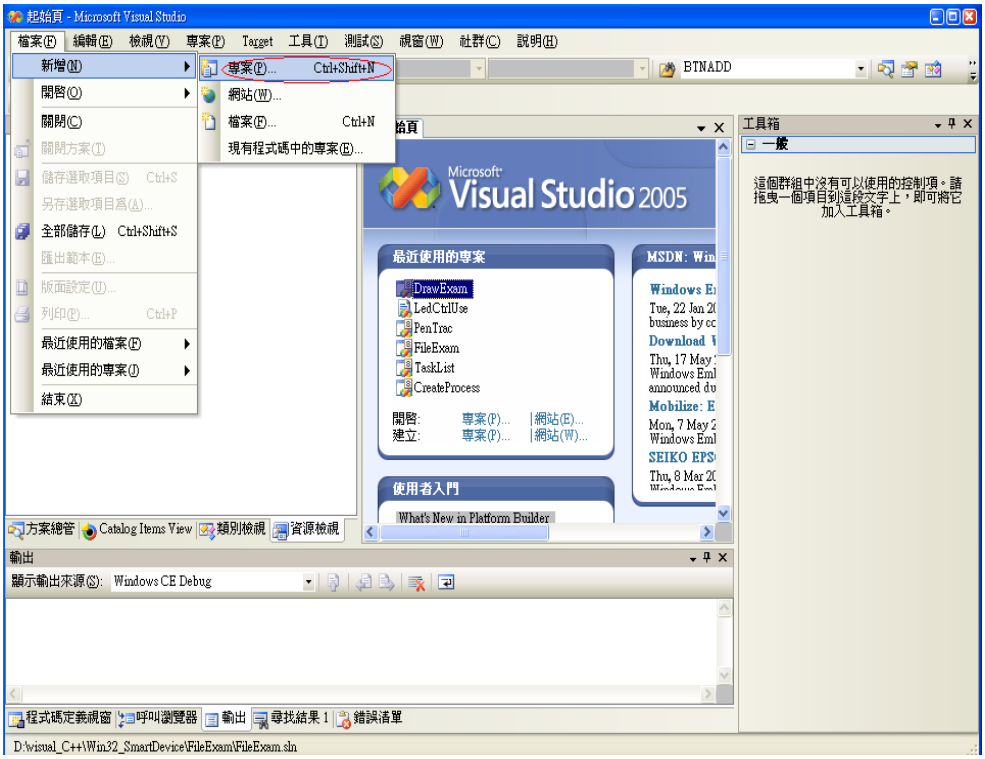
2008/11

一. 實驗目的： 使用螢幕複製函數將抓到的螢幕縮小顯示在使用者自行建立的一個對話框 的窗體上。 首先我們先封裝一個螢幕複製函數,這個函數主要用於抓取螢幕並且把它保 存為一個位元圖;然後將此位元圖縮小成使用者自行建立的一個對話框的窗 體的大小後顯示。

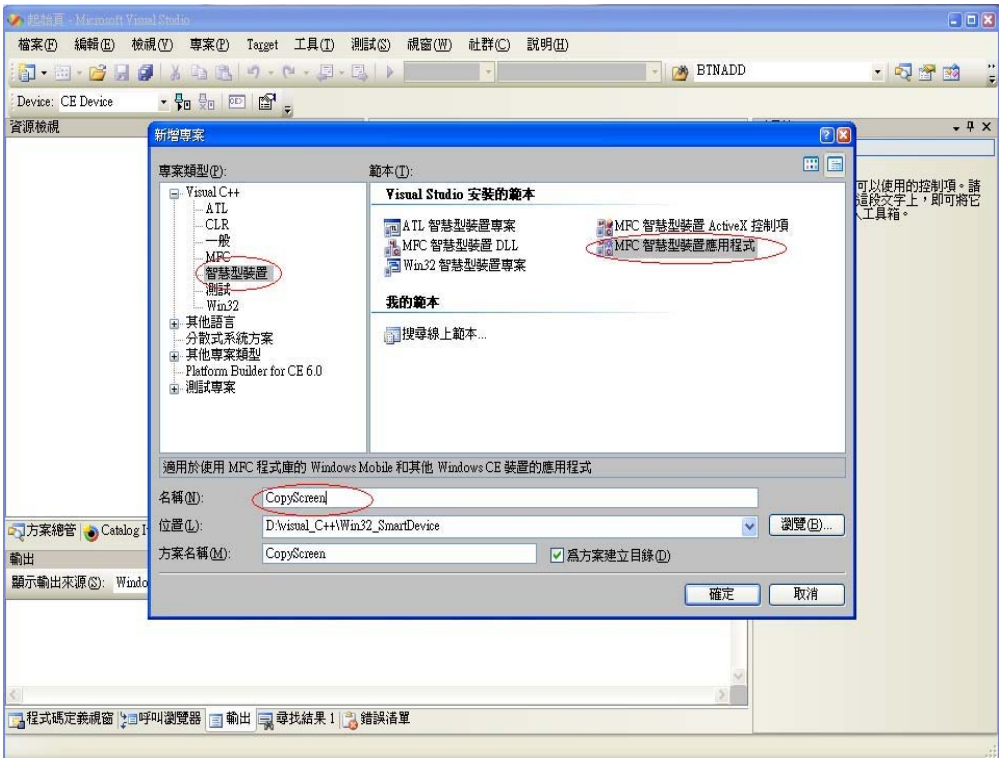
二. 實驗步驟：

1. 建立一個 **MFC** 對話方塊為基礎的專案

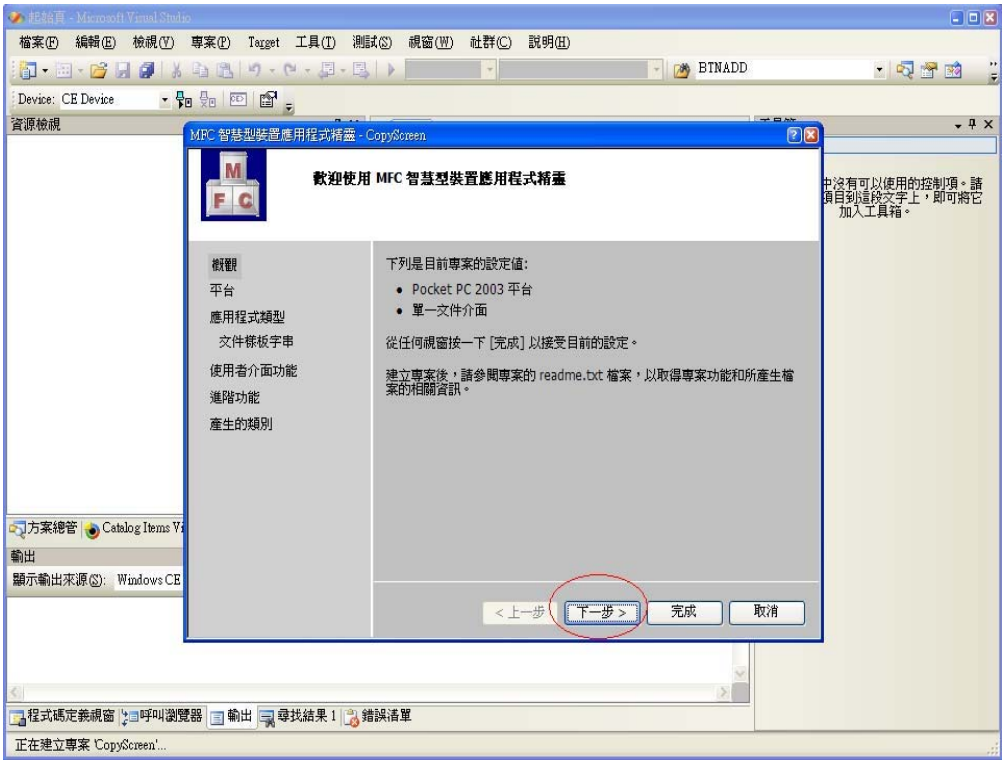
A. 在**Visual Studio 2005** 整合式開發環境中，點選左上角[檔案]->[新增]->[專案]後便會開啟新增專案視窗。



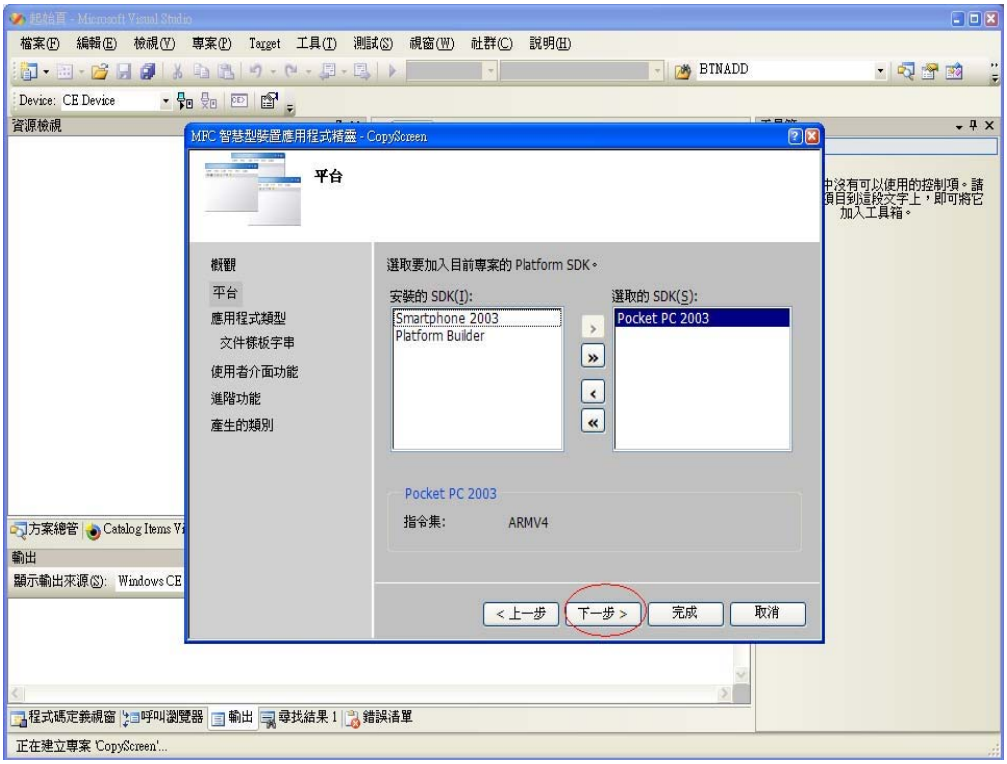
B. 在新增專案視窗內,左邊專案類型欄中選擇[智慧型裝置];右邊 **Visual Studio** 安裝的範本欄中選擇[**MFC** 智慧型裝置應用程式]。



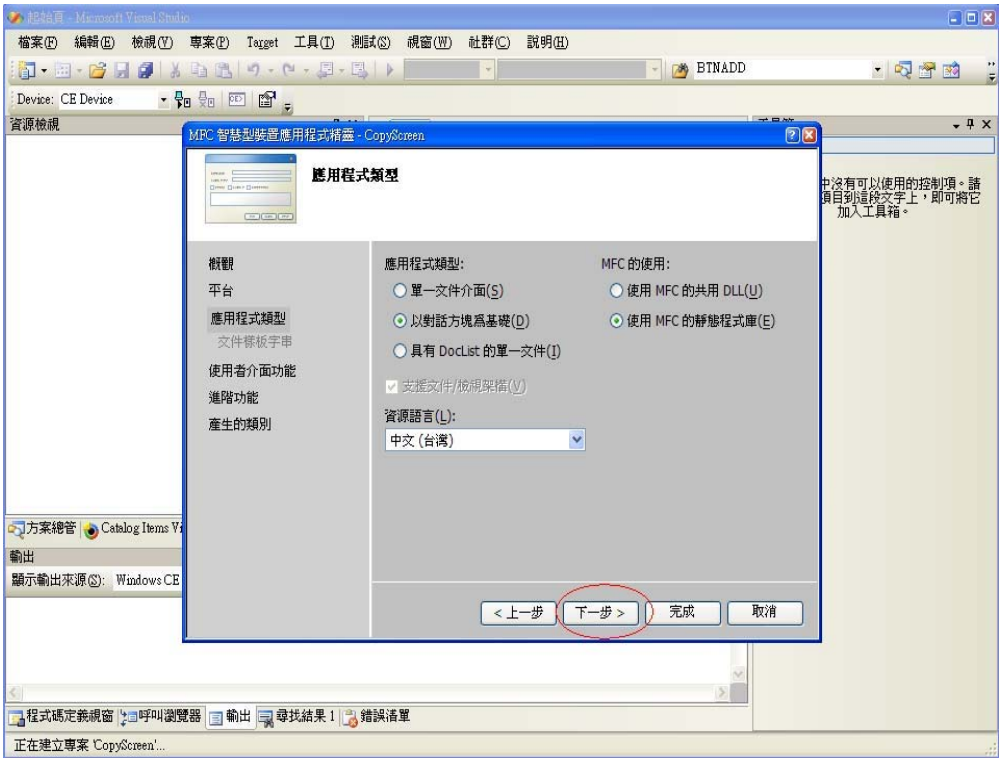
C. 在 **MFC** 智慧型裝置應用程式精靈對話窗中選擇［下一步］按鈕。



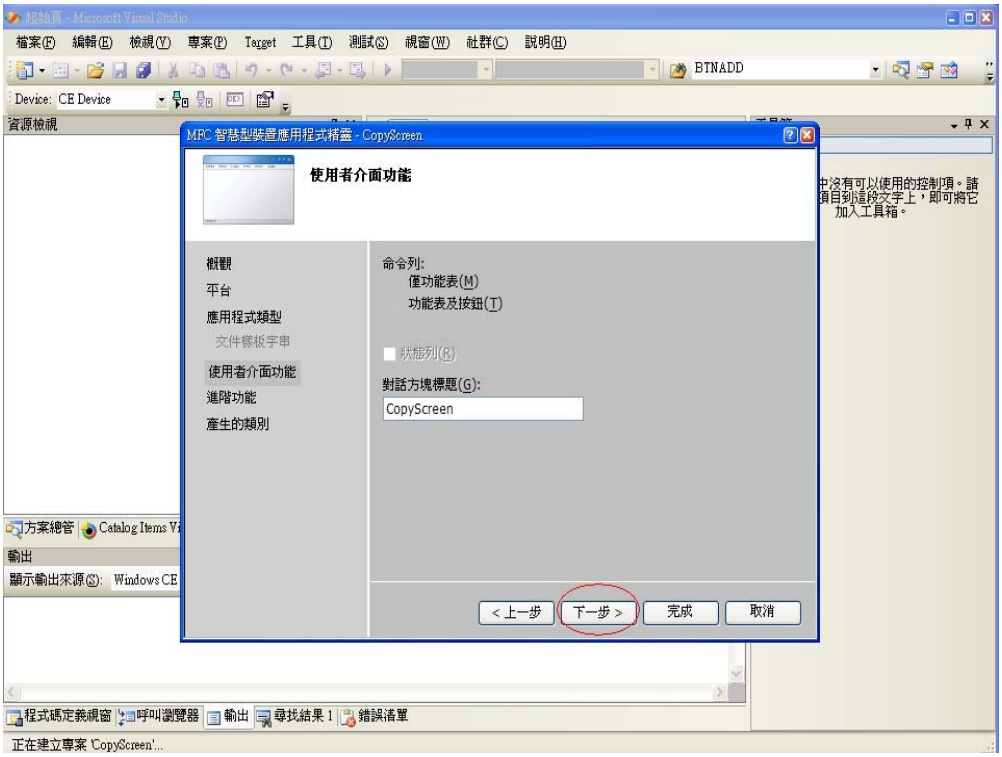
D. 選取要加入目前專案的平台軟體開發包（**Platform SDK**）在此實驗中 我們選擇[**Pocket PC 2003**]，之後按下[下一步］按鈕。



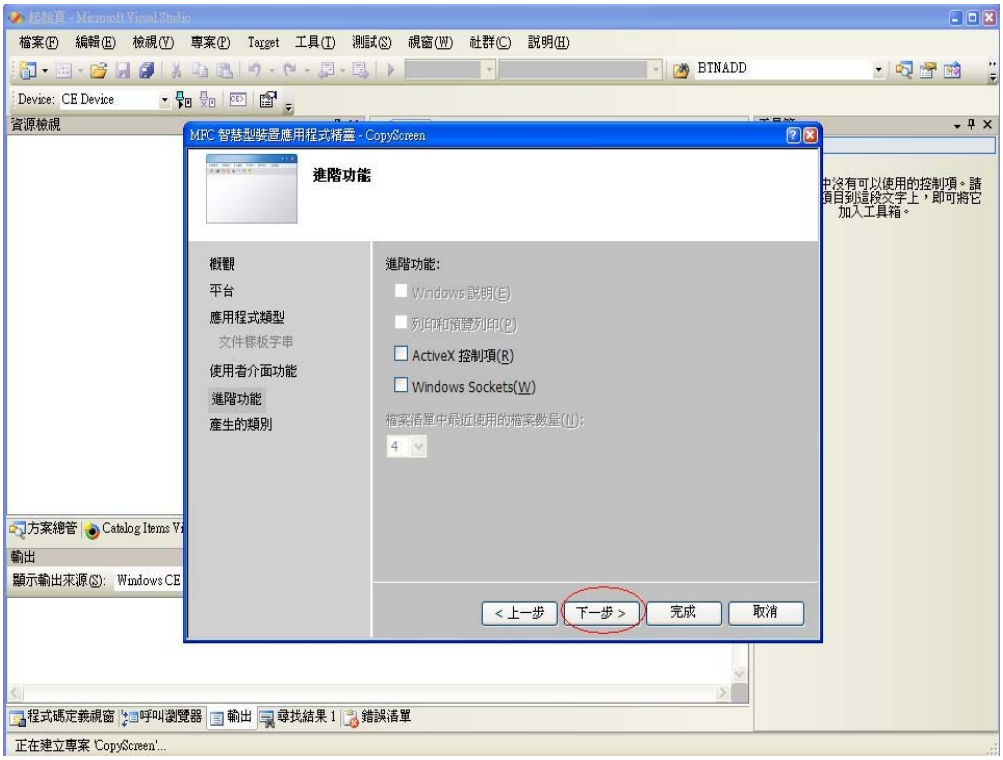
E. 應用程式類型中選取［以對話方塊為基礎］，**MFC** 的使用中選取[使用 **MFC** 的靜態程式庫］，之後按下[下一步］按鈕。



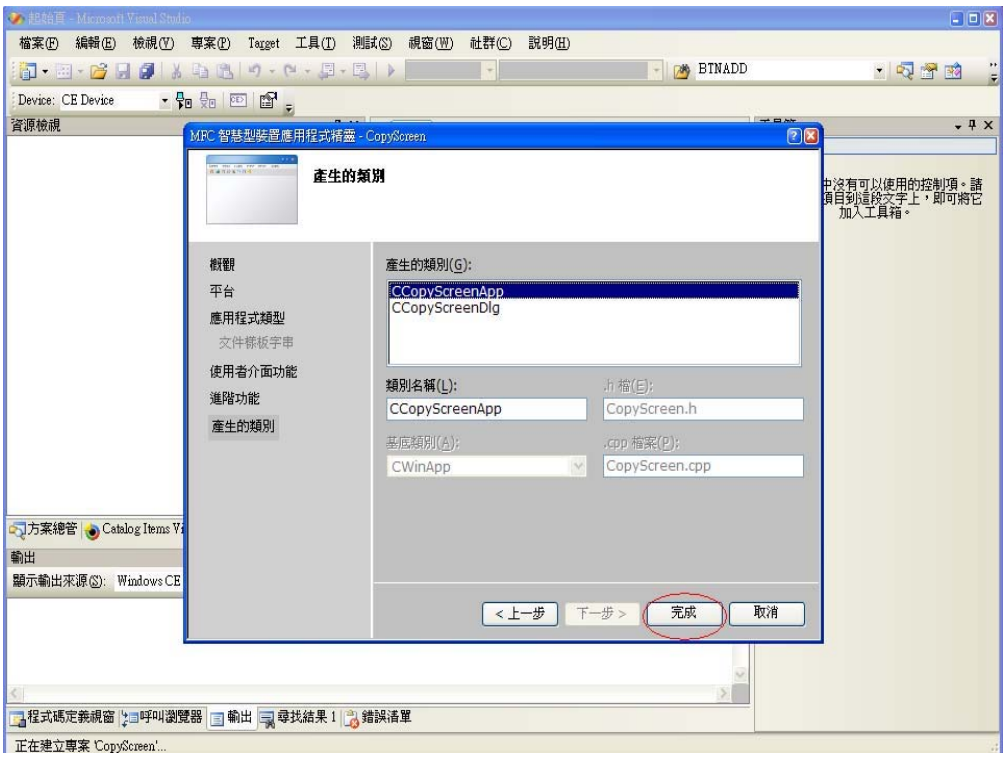
F. 按下[下一步］按鈕。



G. 按下[下一步] 按鈕。

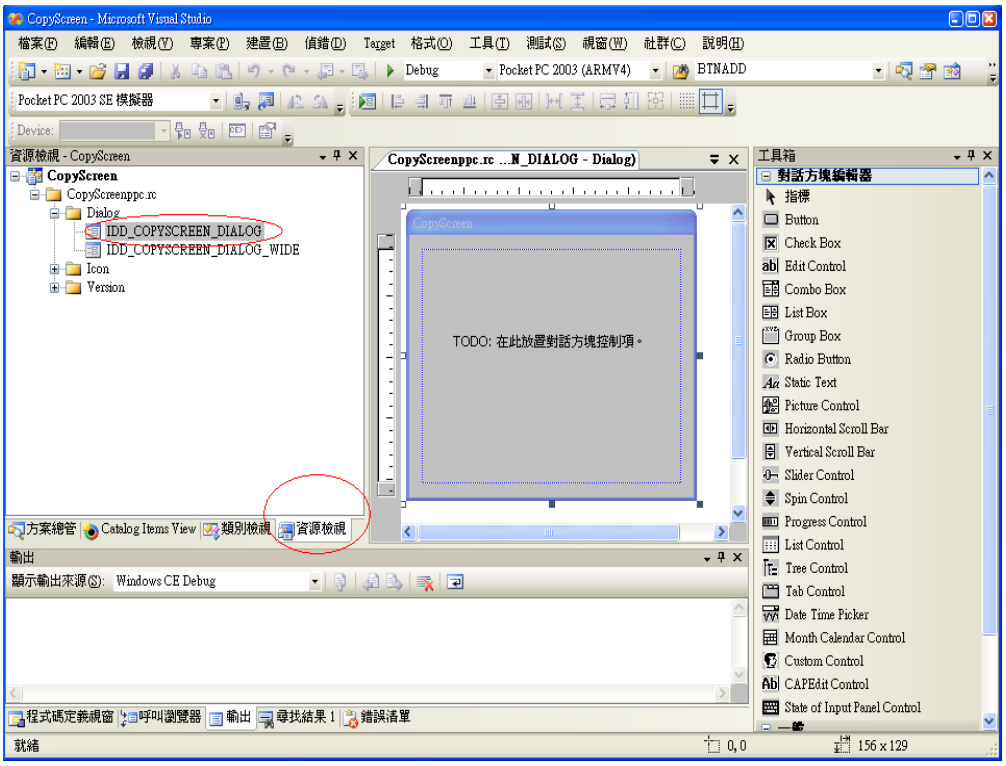


H. 按下[完成] 按鈕。



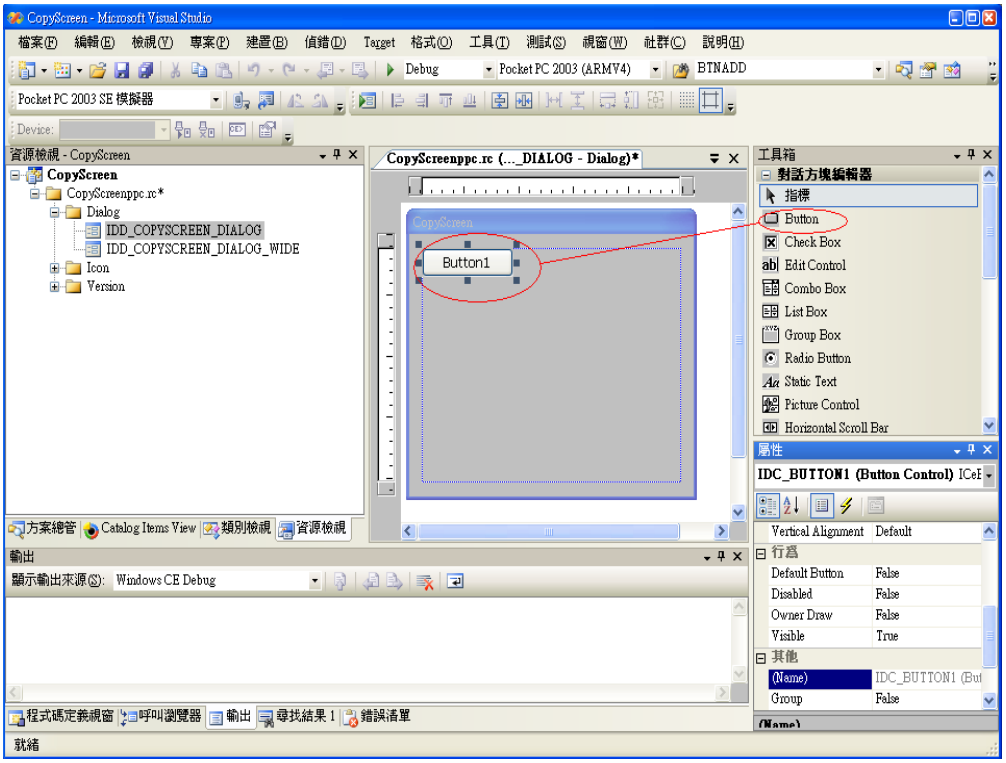
2. 設計使用者界面

A. 切換左方視窗到資源檢視，準備開始設計使用者對話方塊介面（**UI**）

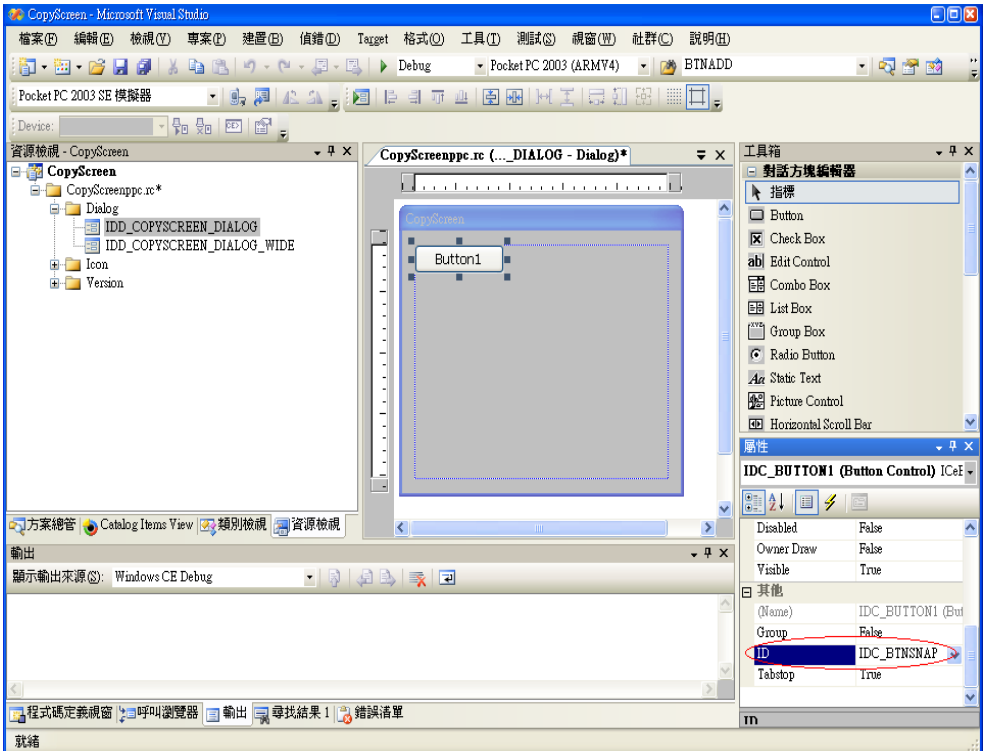


B. 將對話方塊中存在的文字對話方塊 （**TODO**：在此放置對話方塊控制

項）刪除，然後將新選取［**Button**］對話方塊拖曳到適當位置。

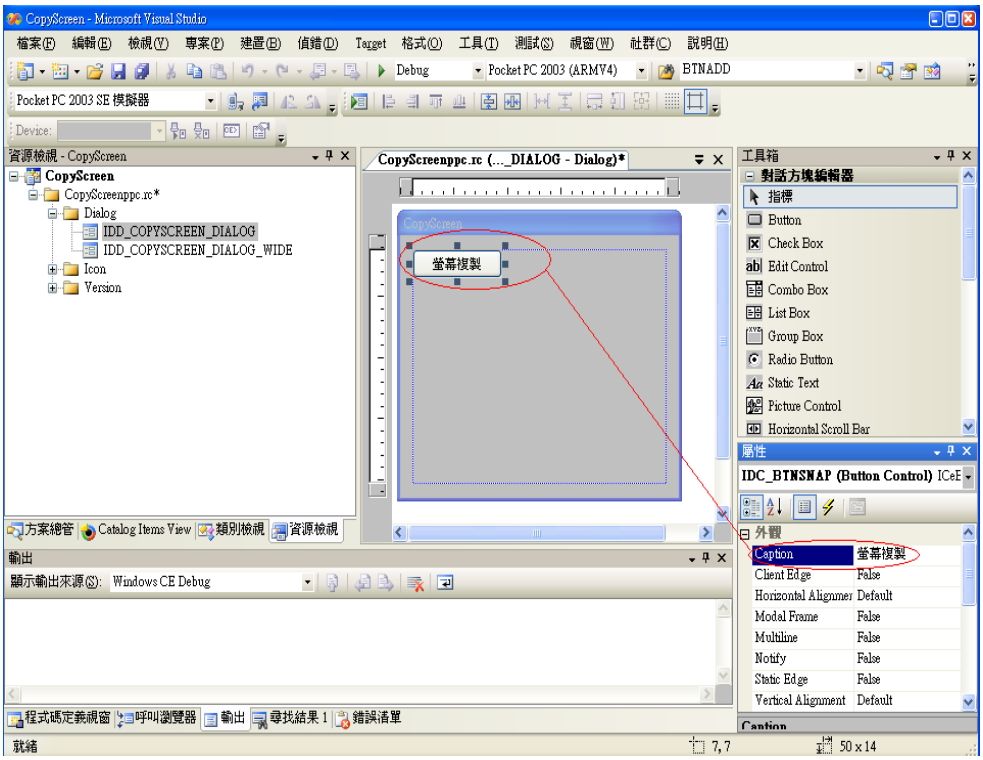


C. 在 「**Button**」 上按下滑鼠右鍵，選擇〔屬性〕修改屬性對話窗中 **ID** 項為「**IDC_BTNSNAP**」。



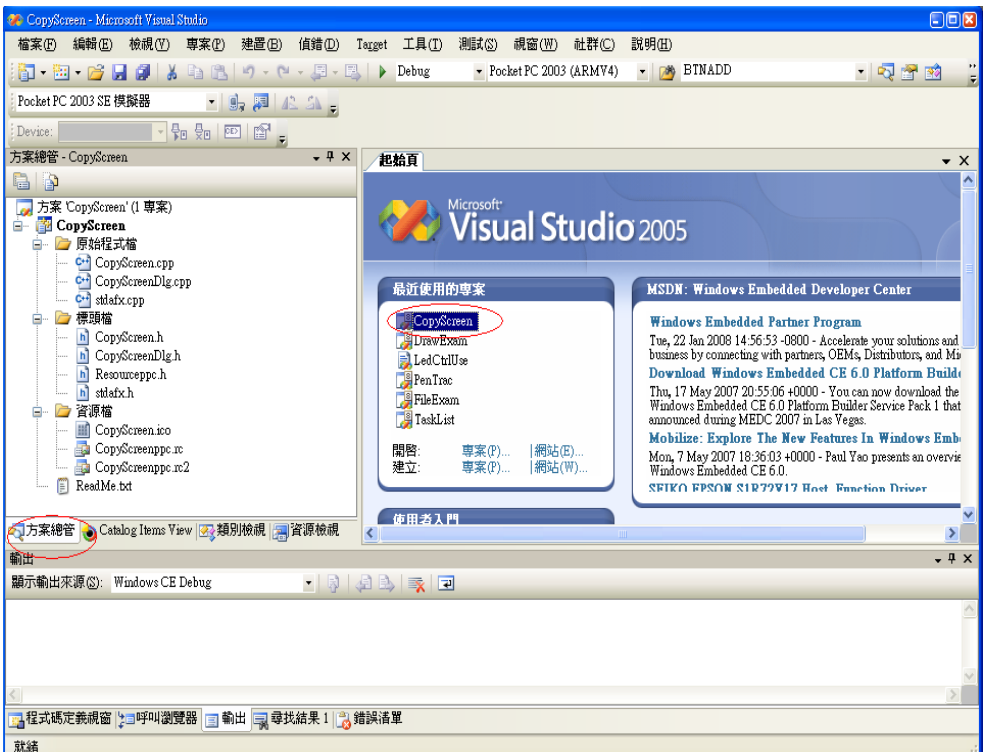
D. 在 「**Button**」 上按下滑鼠右鍵，選擇〔屬性〕修改屬性對話窗中

Caption 項為「螢幕複製」

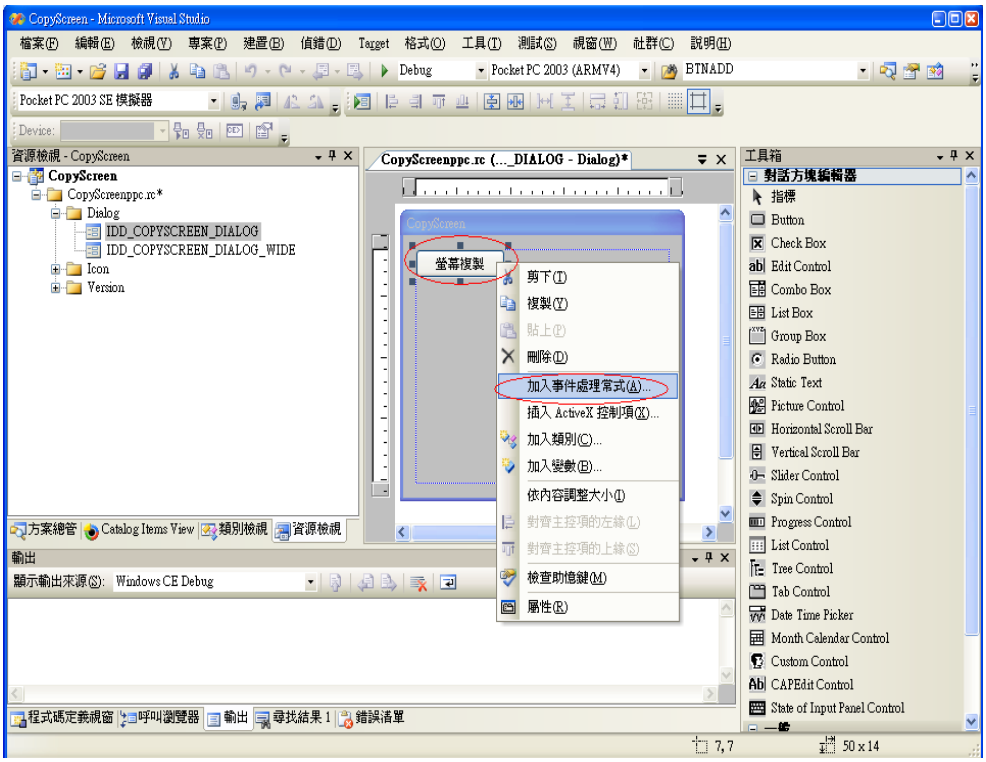


3. 程式設計

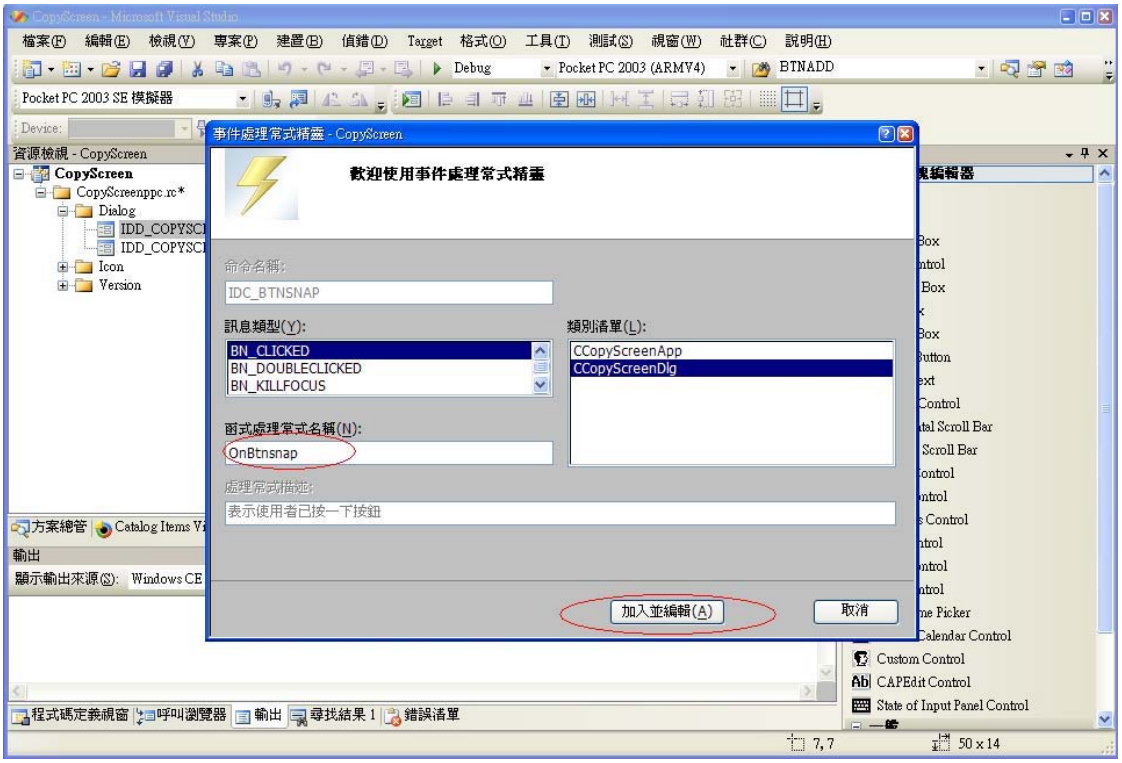
A. 切換左方視窗到方案總管，準備開始設計使用者程式。



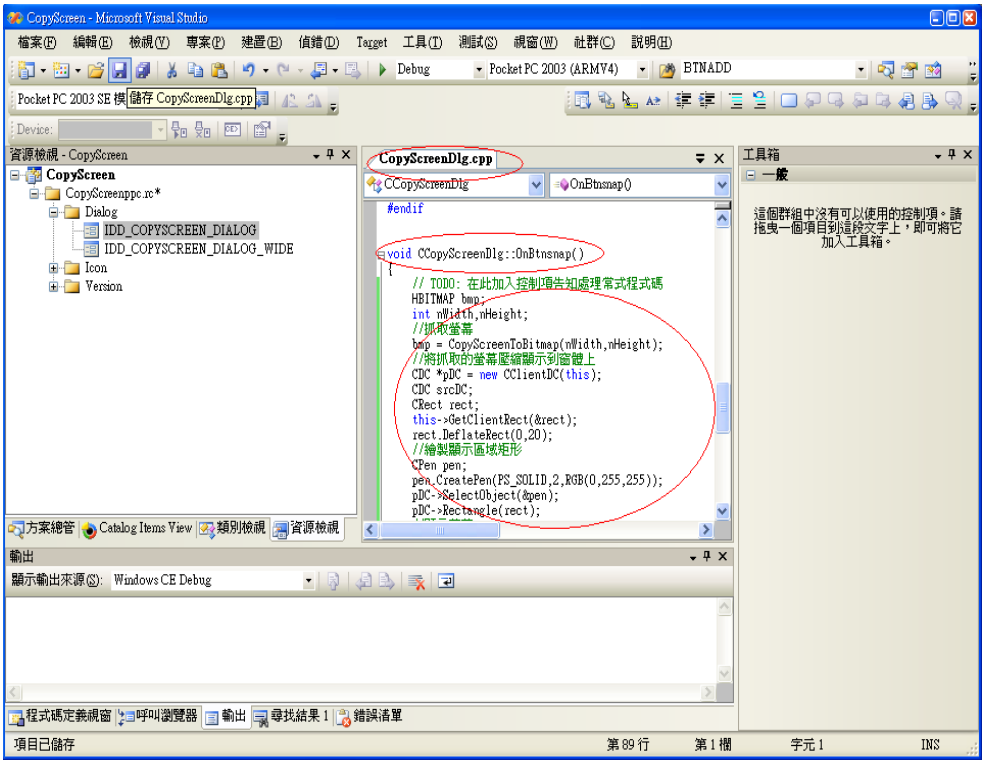
B. 切換左方視窗到資源檢視，準備開始加入事件處理常式。在要加入事件 處理常式（螢幕複製）的**Button** 對話方塊上按下滑鼠右鍵，選擇［加入事件處理常式］。



C. 選取「加入」事件訊息類型後，更改函式處理常式名稱為 **OnBtnsnap**，接下來按下［加入並編輯］按鈕。



D. 程式編輯視窗開啟在**CopyScreenDlg.cpp**，在**void CCopyScreenDlg::OnBtnsnap()**後，加入事件處理常式程式碼。



加入的程式碼 ：

HBITMAP bmp;

int nWidth,nHeight;

// 呼叫自訂的螢幕複製函數

// 將螢幕指定區域存成位元圖

bmp = CopyScreenToBitmap(nWidth,nHeight);

// 將抓取的螢幕壓縮顯示到窗體上

*// **CClientDC** 使用只代表視窗工作區的裝置內容*

*// **new CClientDC** 取得視窗工作區的裝置內容*

*CDC *pDC = new CClientDC(this);*

*// **CDC** 封裝了所有圖形輸出函數，繼承於**CObject***

CDC srcDC;

*// **CRect** 一個指向**RECT** 的**32** 位元指標*

*// **RECT** 是由宣告為**long** 的**Left**、**Top**、**Right** 和 **Bottom** 的變*

數組成的結構體

CRect rect;

// 取得指定視窗代碼的工作區大小

// 取得的**Left** 及**Top** 值會是**0** ；**Right** 及**Bottom** 會是**Width** 和

Height

this->GetClientRect(&rect);

// 朝中心移動邊以縮小**CRECT**

rect.DeflateRect(0,20);

// 繪製顯示區域矩形

// **CPen** 是 **CGdiObject** 的衍生類

// 實現畫筆功能，用於繪製邊框、直線和曲線等

CPen pen;

// 產生一枝使用者自定的筆

pen.CreatePen(PS_SOLID,2,RGB(0,255,255));

// 選擇視窗工作區的裝置內容使用**pen** 物件

pDC->SelectObject(&pen);

// 在選擇視窗工作區的裝置內容上繪製矩形

pDC->Rectangle(rect);

// 顯示螢幕

// 建立一個相容的記憶體裝置

srcDC.CreateCompatibleDC(pDC);

// 選擇將螢幕指定區域存成位元圖到一個裝置環境

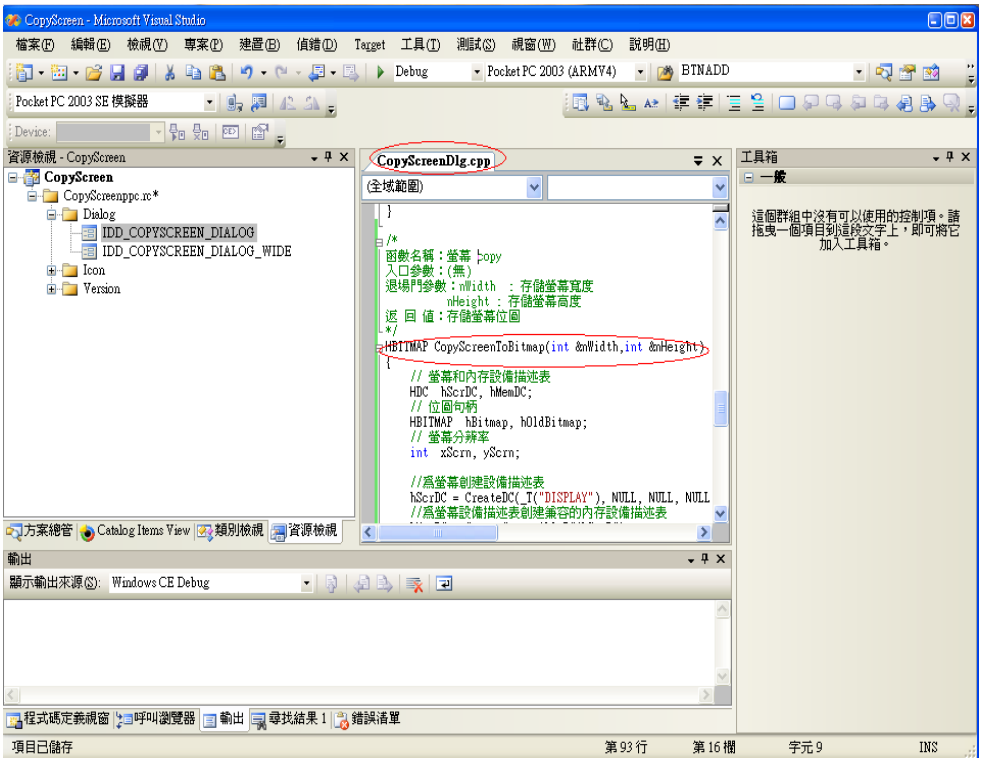
srcDC.SelectObject(bmp);

// 將一幅位元圖從一個設備場景複製到另一個

```
pDC->StretchBlt(rect.left+2,rect.top+2,rect.right - rect.left –
                4 ,rect.bottom - rect.top –
                4,&srcDC,0,0,nWidth,nHeight,SRCCOPY);
```

```
// 刪除位元圖
DeleteObject(bmp);
```

E. 程式編輯視窗開啟在**CopyScreenDlg.cpp**，在原始程式碼最後，加入一段螢幕複製的處理函數程式碼。



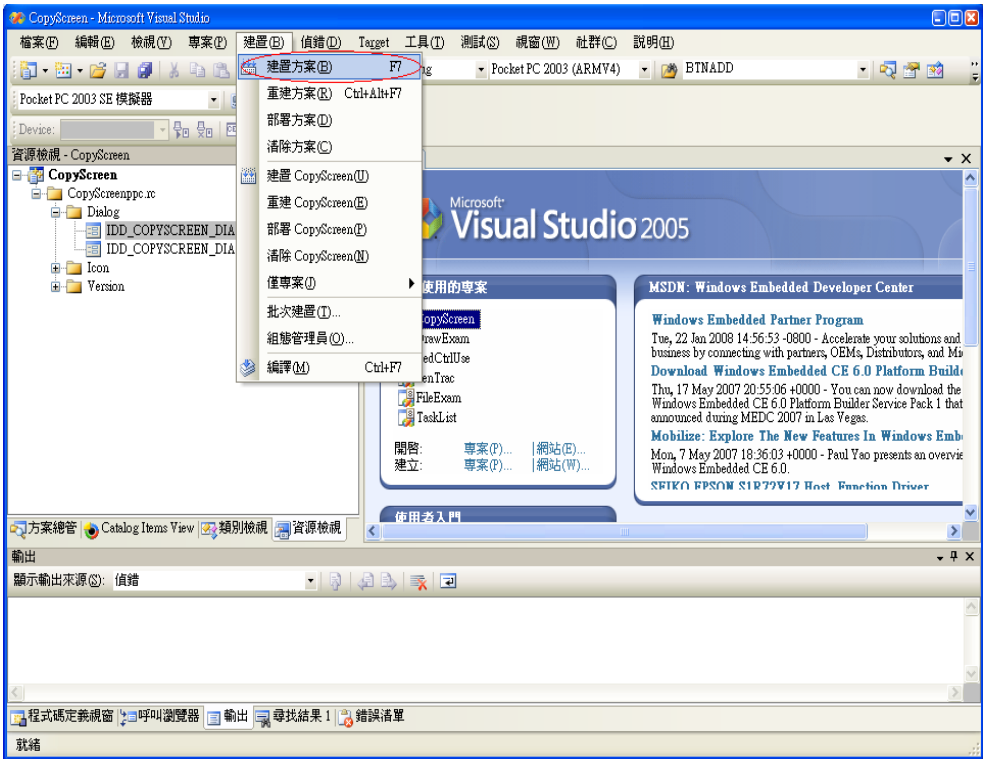
加入的程式碼 ：

```
// 自訂的螢幕複製函數
HBITMAP CopyScreenToBitmap(int &nWidth,int &nHeight)
{
    // 螢幕和內存設備描述表宣告
    HDC  hScrDC, hMemDC;

    // 位元圖宣告
    HBITMAP  hBitmap, hOldBitmap
;
    // 螢幕分辨率宣告
    int  xScrn, yScrn;

    // 取得螢幕設備場景
    hScrDC = CreateDC(_T("DISPLAY"), NULL, NULL, NULL);
```

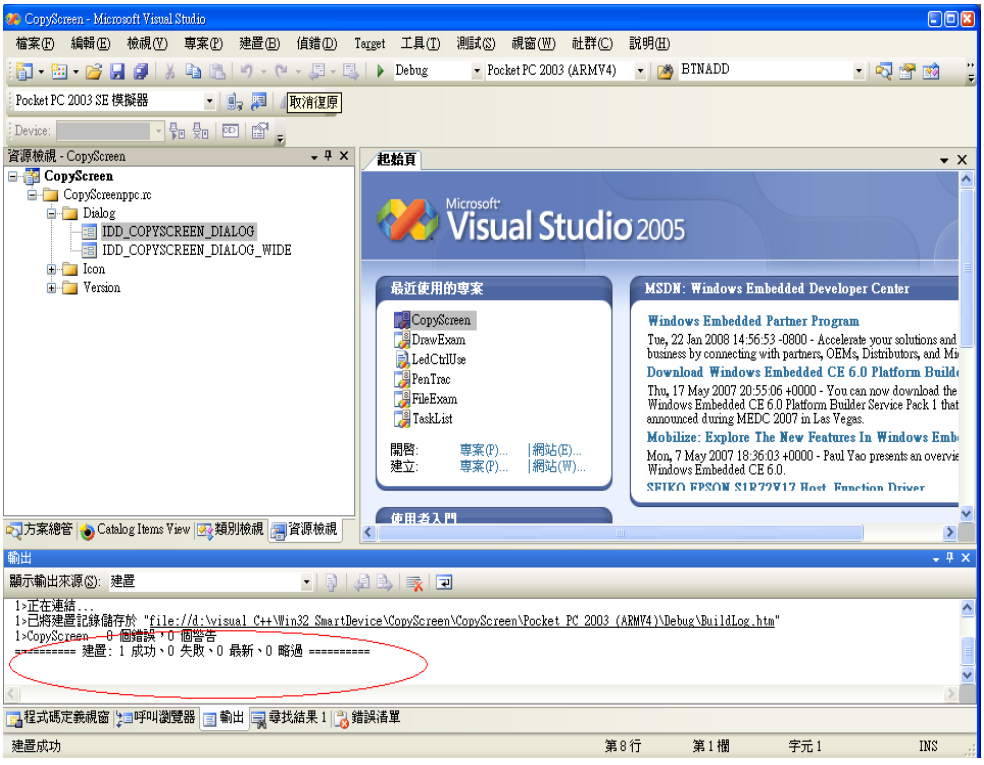
```
    //  為螢幕設備描述表創建兼容的內存設備描述表
    hMemDC = CreateCompatibleDC(hScrDC);

    //  獲得螢幕分辨率
    xScrn = GetDeviceCaps(hScrDC, HORZRES);
    yScrn = GetDeviceCaps(hScrDC, VERTRES);

    //  存儲螢幕的長
    寬 nWidth = xScrn;
    nHeight = yScrn;

    //  創建一個與螢幕設備描述表兼容的位圖
    hBitmap = CreateCompatibleBitmap(hScrDC, xScrn, yScrn);

    //  把新位圖選到內存設備描述表中
    hOldBitmap = (HBITMAP)SelectObject(hMemDC, hBitmap);

    //  把螢幕設備描述表拷貝到內存設備描述表中
    BitBlt(hMemDC, 0, 0, xScrn,yScrn,hScrDC, 0, 0, SRCCOPY);

    //  得到螢幕位圖的句柄
    hBitmap = (HBITMAP)SelectObject(hMemDC, hOldBitmap);

    //  清除
    DeleteDC(hScrDC);
    DeleteDC(hMemDC);

    //  返回位圖句柄
    return hBitmap;
}
```

4. 編譯和執行

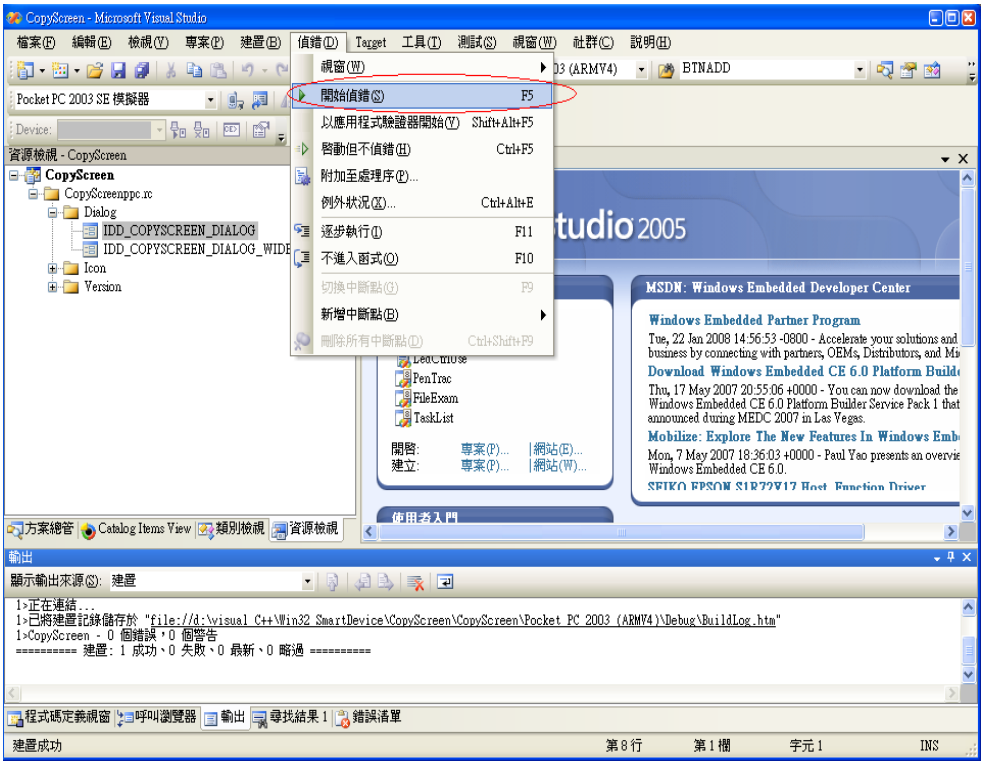
A. 在**Visual Studio 2005** 整合式開發環境中，點選[建置] -> [建置專案]後便會開始編譯整個專案。



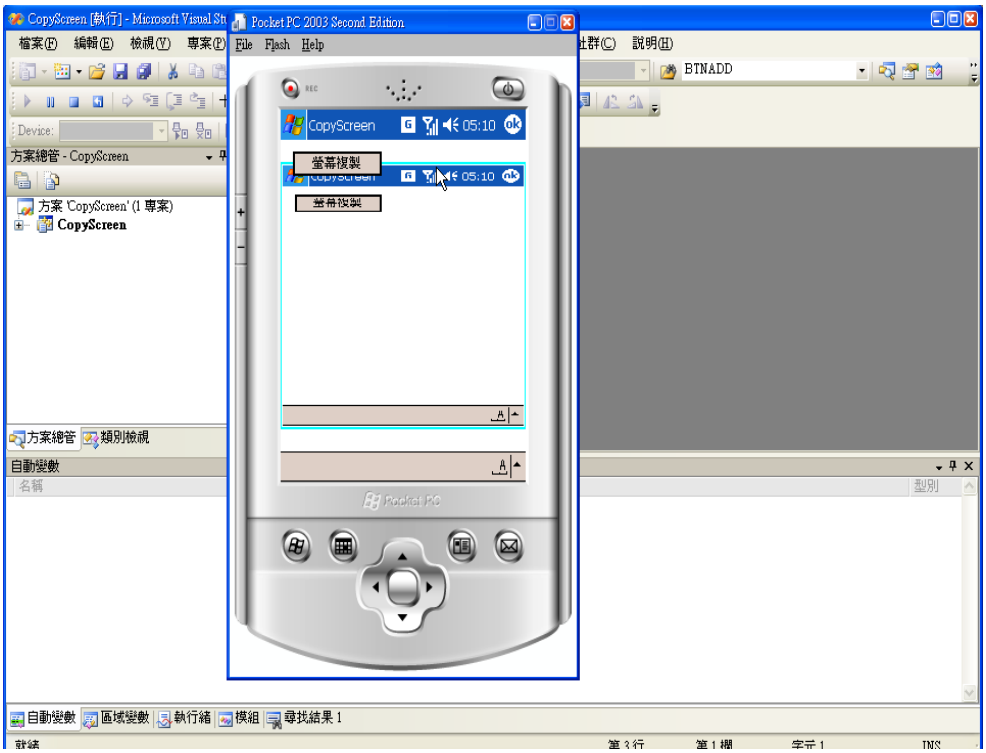
B. 編譯成功後，在**Visual Studio 2005** 整合式開發環境下方的輸出視窗中顯示建置成功。



C. 在**Visual Studio 2005** 整合式開發環境中，點選[偵錯] -> [開始偵錯]後便會開始對此專案執行並偵錯。


CopyScreen - Microsoft Visual Studio
開始偵錯(S) F5
以應用程式驗證器開始(Y) Shift+Alt+F5
啓動但不偵錯(H) Ctrl+F5
附加至處理序(P)...
例外狀況(X)... Ctrl+Alt+E
逐步執行(I) F11
不進入函式(O) F10
新增中斷點(B)
CopyScreen
CopyScreenppc.rc
Dialog
IDD_COPYSCREEN_DIALOG
IDD_COPYSCREEN_DIALOG_WIDE
Icon
Version
MSDN: Windows Embedded Developer Center
Windows Embedded Partner Program
Download Windows Embedded CE 6.0 Platform Build
Mobilize: Explore The New Features In Windows Emb
輸出
1>正在連結...
1>已將建置記錄儲存於 "file://d:\visual C++\Win32 SmartDevice\CopyScreen\CopyScreen\Pocket PC 2003 (ARMV4)\Debug\BuildLog.htm"
1>CopyScreen - 0 個錯誤，0 個警告
========== 建置: 1 成功，0 失敗，0 最新，0 略過 ==========
建置成功

D. 執行結果如下圖所示。


CopyScreen [執行] - Microsoft Visual Stu
Pocket PC 2003 Second Edition
File Flash Help
CopyScreen 05:10
螢幕複製
方案總管 - CopyScreen
方案 'CopyScreen' (1 專案)
CopyScreen
自動變數
名稱
型別

應用程式開發實驗

實驗五

工作列表

Rev. 1.0

一. 實驗目的： 每個應用程序自動啟動後,就會變成一個單獨的處理序,並且每個處理序都 有自己的虛擬內存空間。作業系統可以列舉系統進行的處理序,並且可以根 據處理序的狀態來「終止處理序」和「啟動處理序」等操作。

處理序（**Process**）是指當前所加載程序的專業 **Win32** 術語,例如：磁碟上 的任何一個可執行檔案,僅僅是一個檔案,它只有在被啟動後才是一個處理序。處理序僅僅是存在的，它不做認何事。一個處理序可以有多個執行緒（**Thread**）但至少應包含一個執行緒。處理序中所有的操作都是由執行緒來完成的。同時每一個處理序有且僅有一個主執行緒,由主執行緒來進行所有初始化操作。

WinCE 的處理序中任一時刻最多可以有**32** 個處理序同時運行。當**WinCE** 系統啟動時，至少會啟動**4** 個處理序：**NK.EXE** 提供內核服務；**FILESYS.EXE** 提供檔案系統服務；**GWES.EXE** 提供**GUI** 服務；**DEVICE.EXE** 提供加載和維護設備驅動程序。

WinCE 的處理序中止方法是採用透過處理序的主執行緒調用**ExitThread** 來中止處理序。一旦主執行緒終止了，處理序也將終止。另外也可以調用 **GetExitCodeProcess** 函數來確定處理序是否終止。

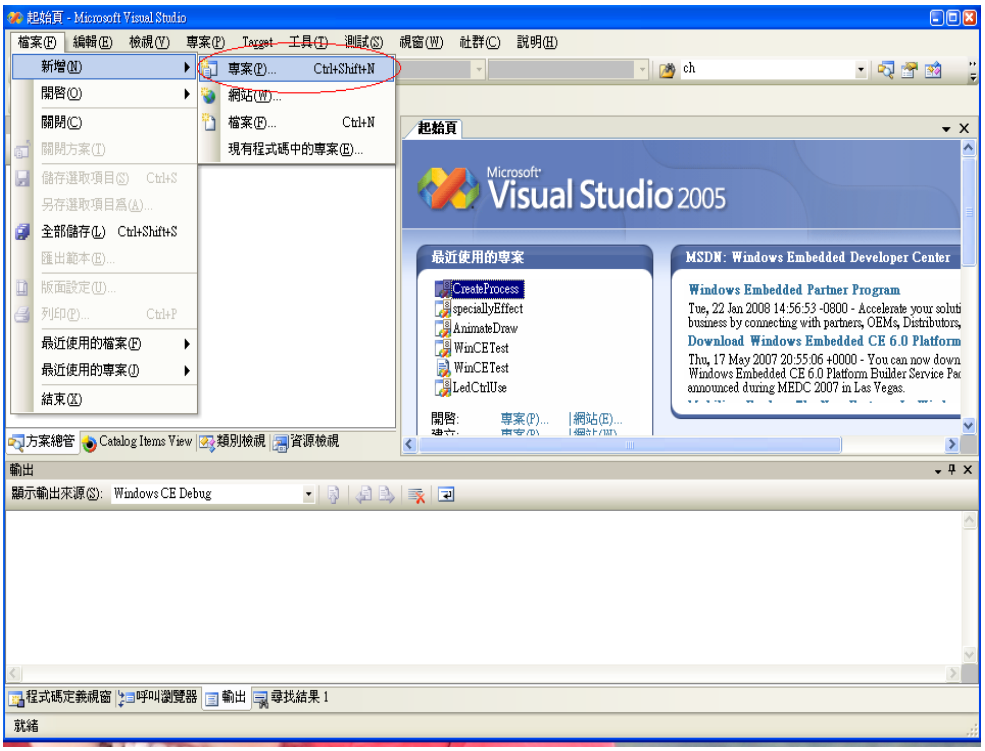
此實驗將透過**WinCE** 中 **API** 的 **ToolHelp** 函數，來獲取作業系統的某些底層訊息,尤其是關於當前正在運行的處理序以及每個處理序下的執行緒、模組和堆積訊息。**ToolHelp API** 都存儲在一個叫 **TOOLHELP.DLL** 的動態 連結庫中，因此我們在使用時，必須先加載此動態庫，才能使用此 **API**。 **WinCE** 是多任務的，經常需要建立、修改或刪除像處理序、執行緒、模組

或堆積等對象。由於系統的各種狀態經常處於變動當中,系統訊息現在有意義，不代表過一會還有意義。例如：想要列舉操作系統中所加載的模組，由於操作系統有可能正在動態地調度執行緒的執行,因此即使已經列舉出許多模組,但是可能還有些模組正在建立和刪除。這樣一種動態環境中，為了獲取諸如此類的訊息，最好是能把系統凍結一段時間。儘管**ToolHelp** 無法凍 結系統,但它可以給某一時刻的系統拍一個快照,我們可以在快照裡來慢慢查詢我們要的訊息。建立系統快照需要調用**CreateToolHelp32Snapshot** 函數。

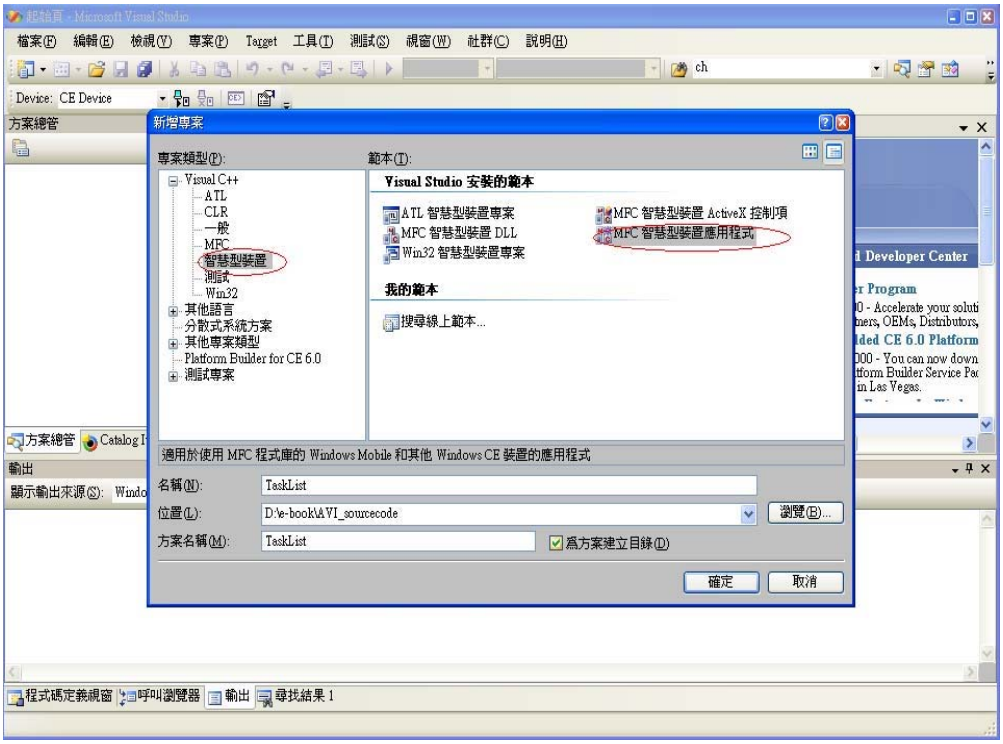
二. 實驗步驟：

1. 建立一個 **MFC** 對話方塊為基礎的專案

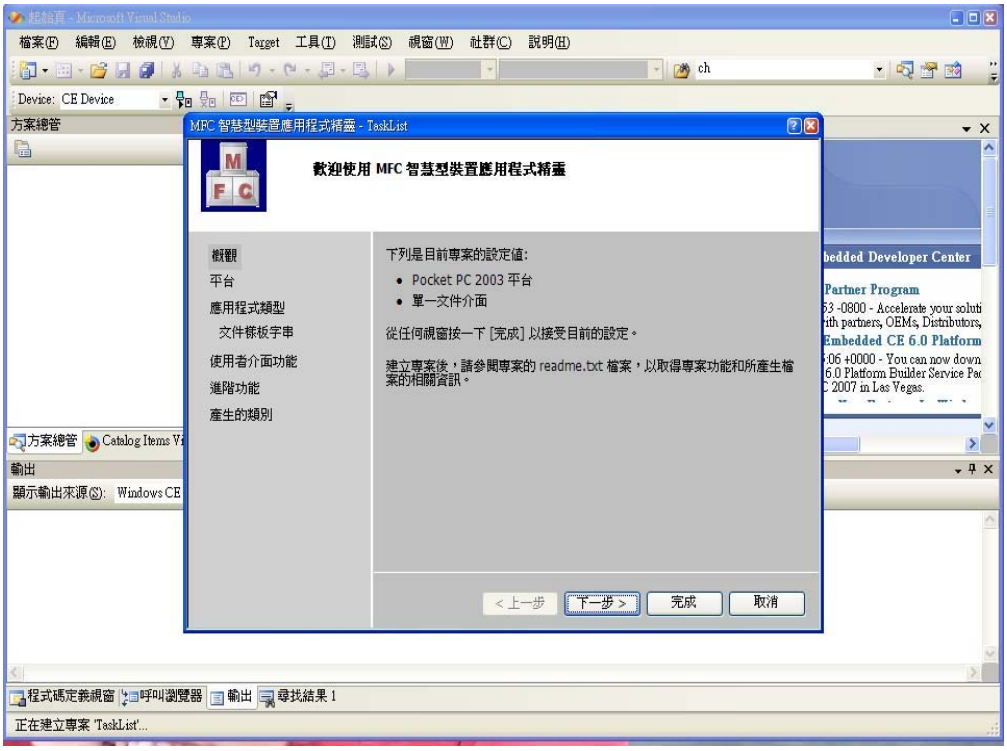
A. 在**Visual Studio 2005** 整合式開發環境中，點選左上角[檔案]->[新增]->[專案] 後便會開啟新增專案視窗。



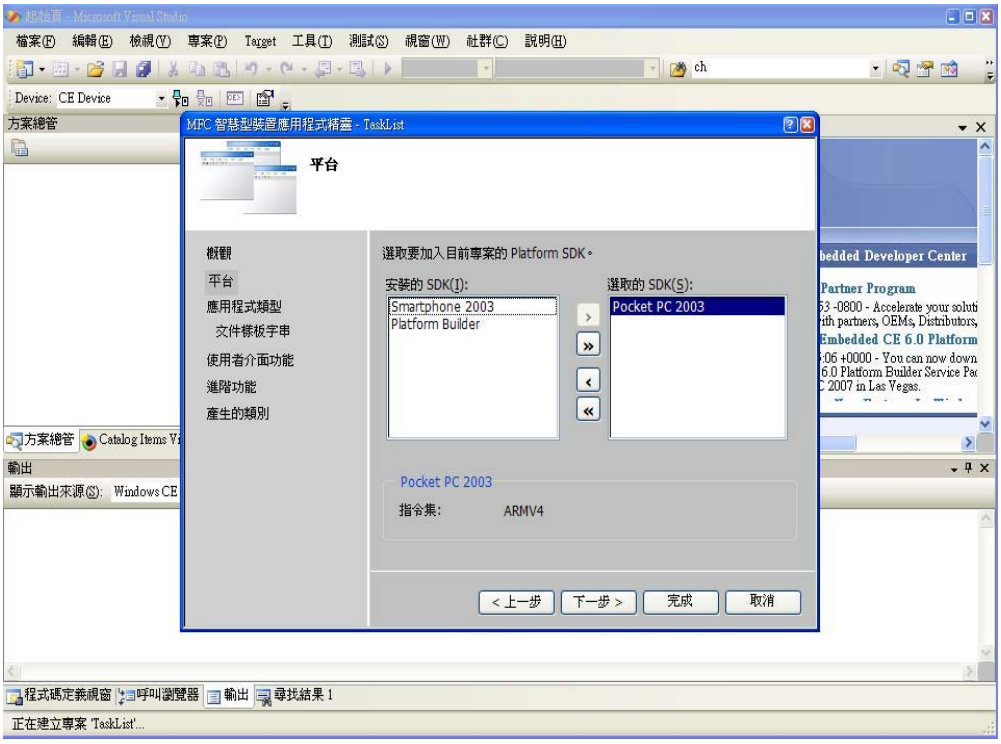
B. 在新增專案視窗內,左邊專案類型欄中選擇[智慧型裝置];右邊 **Visual Studio** 安裝的範本欄中選擇[**MFC** 智慧型裝置應用程式]。



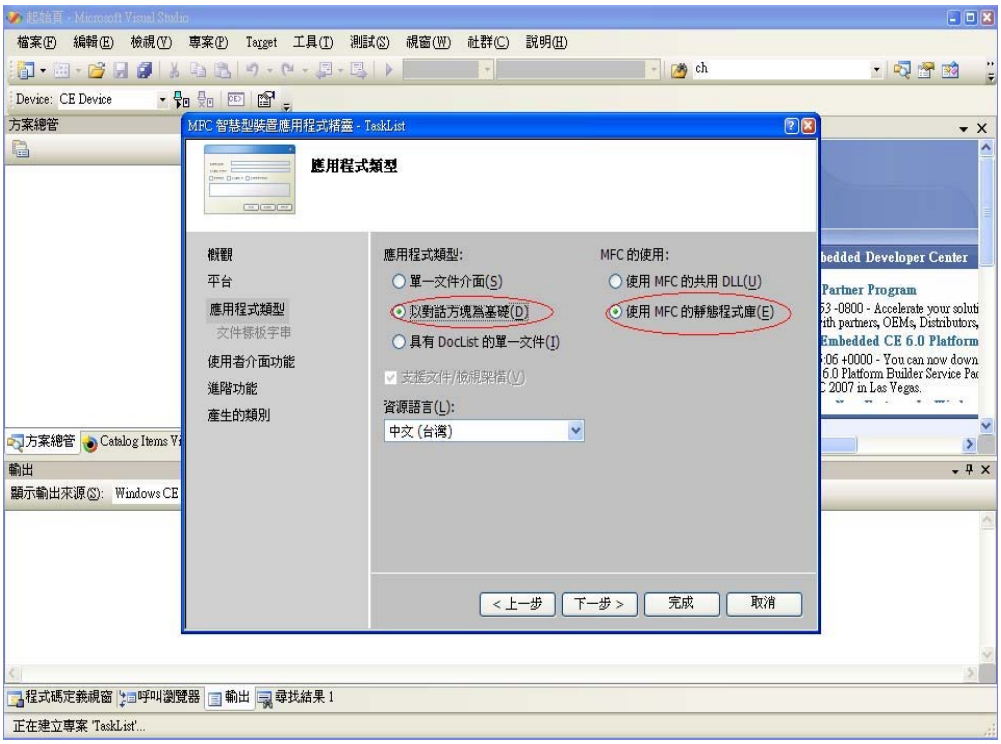
C. 在 **MFC** 智慧型裝置應用程式精靈對話窗中選擇［下一步］按鈕。



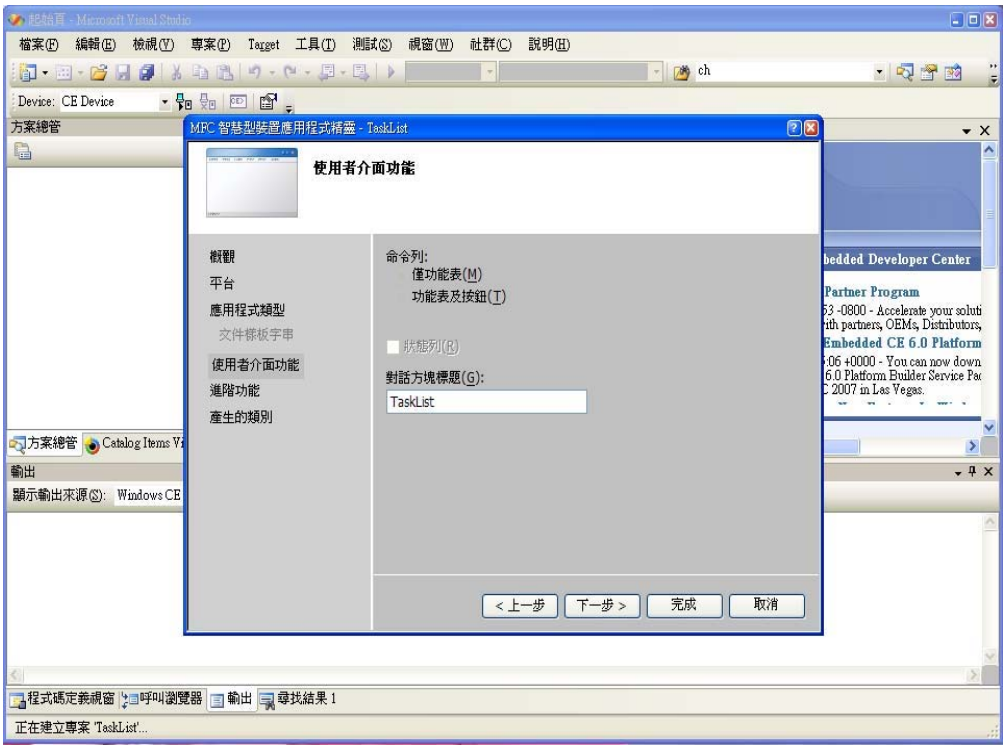
D. 選取要加入目前專案的平台軟體開發包（**Platform SDK**）在此實驗中我們選擇[**Pocket PC 2003**]，之後按下[下一步] 按鈕。



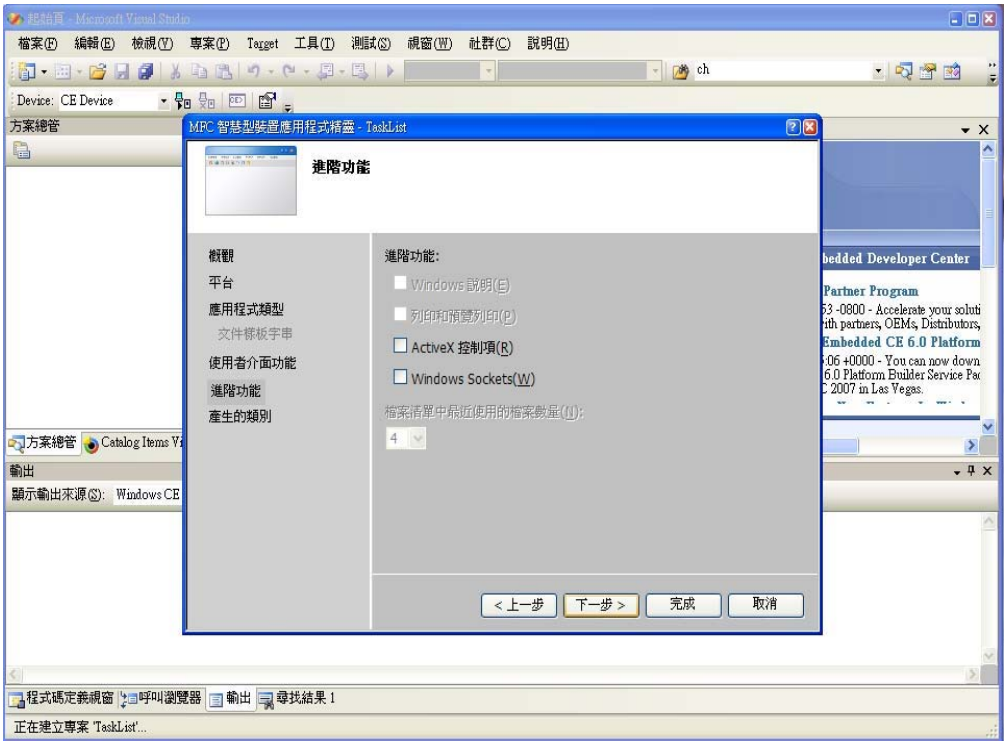
E. 應用程式類型中選取 [以對話方塊為基礎]，**MFC** 的使用中選取[使用 **MFC** 的靜態程式庫] ，之後按下[下一步] 按鈕。



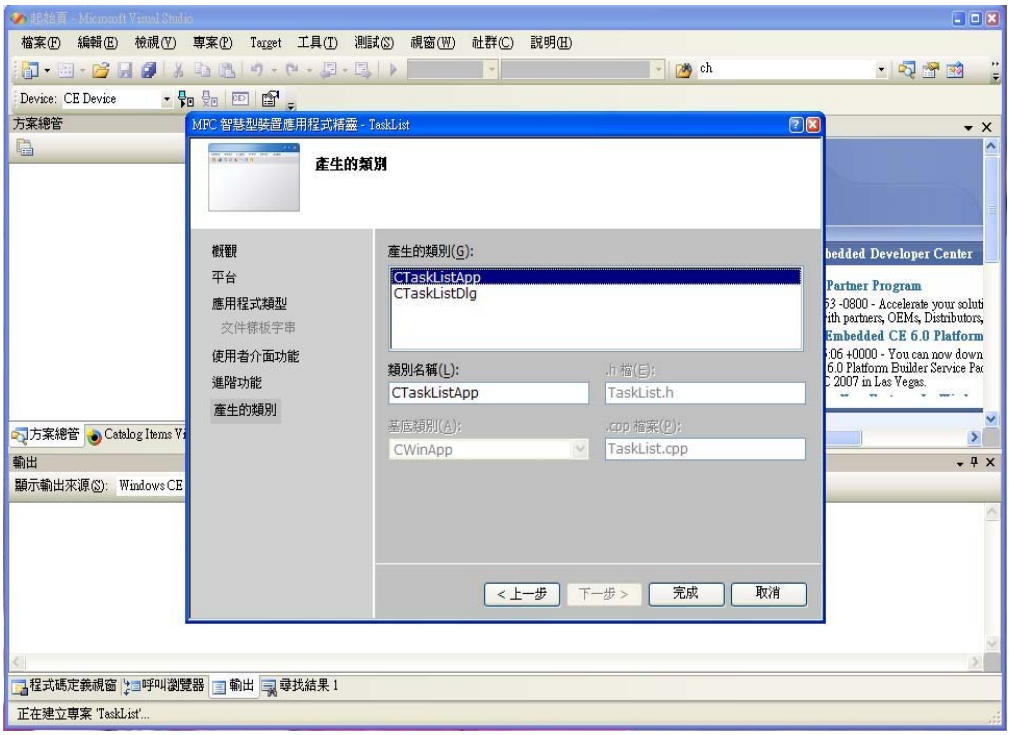
F. 按下[下一步]按鈕。



G. 按下[下一步]按鈕。

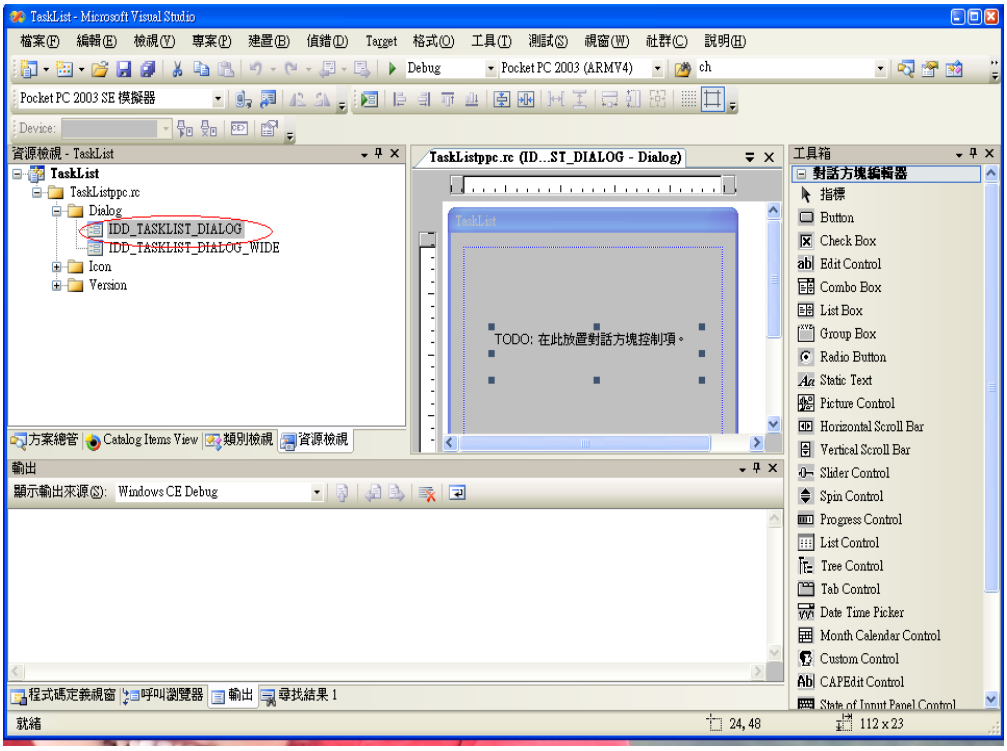


H. 按下[完成] 按鈕。

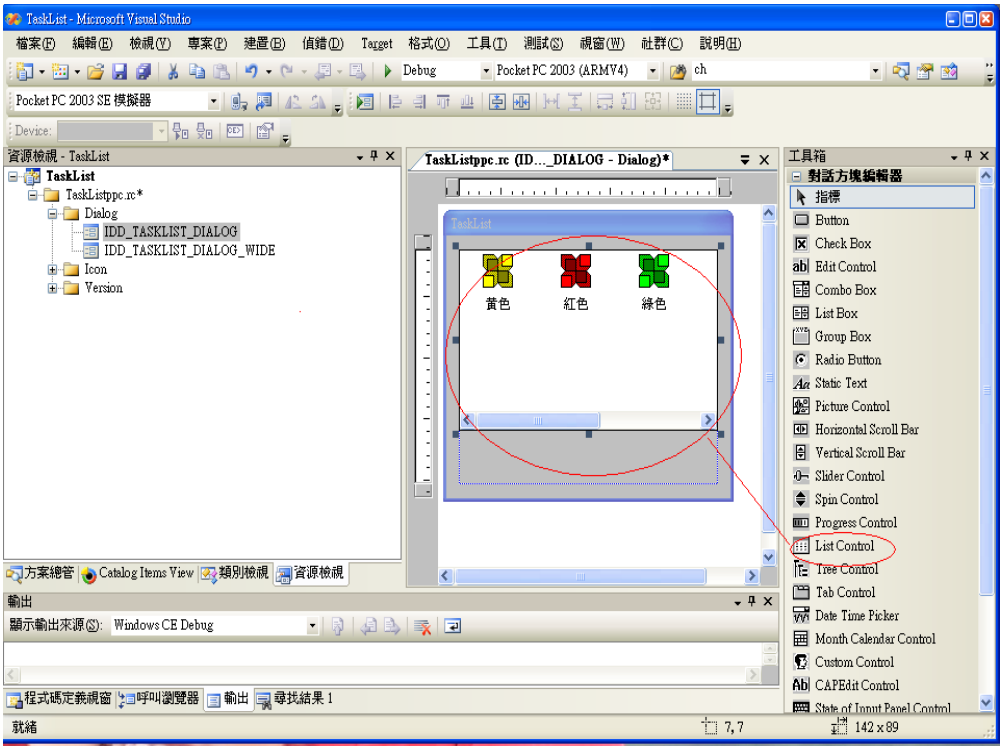


2. 設計使用者界面

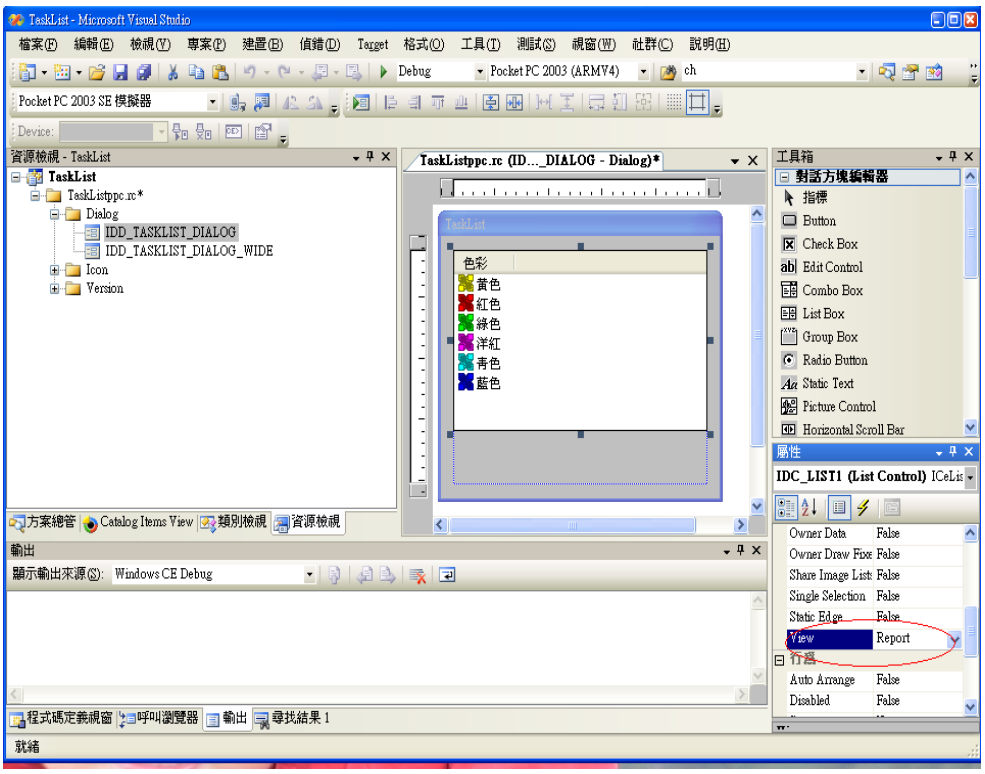
A. 切換左方視窗到資源檢視，準備開始設計使用者對話方塊介面（**UI**）



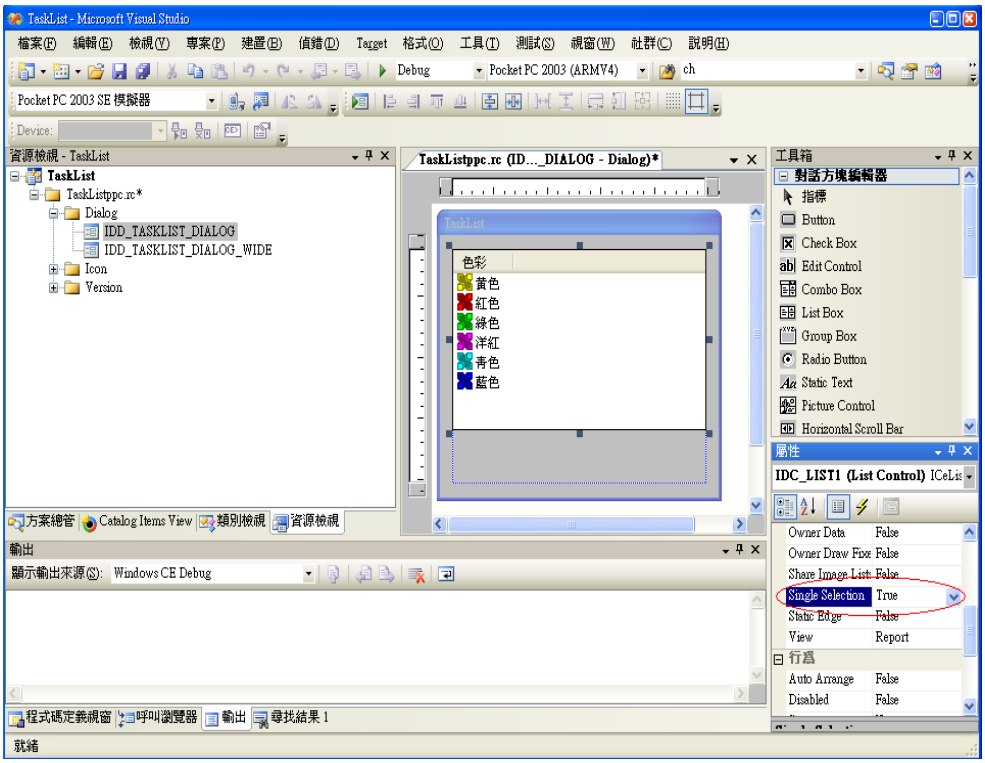
B. 將對話方塊中存在的文字對話方塊 （**TODO**：在此放置對話方塊控制項）刪除，然後將新選取［**List Control**］對話方塊拖曳到適當位置。



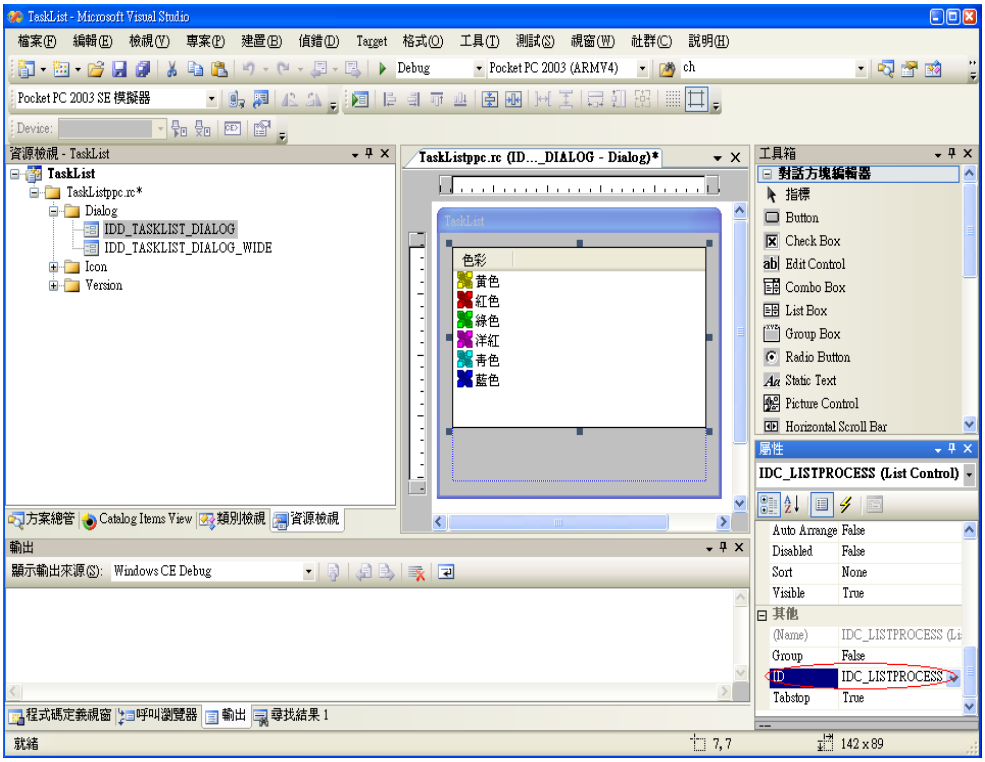
C. 在 **List Control** 上按下滑鼠右鍵，選擇〔屬性〕修改屬性對話窗中 **View** 項為**Report**。



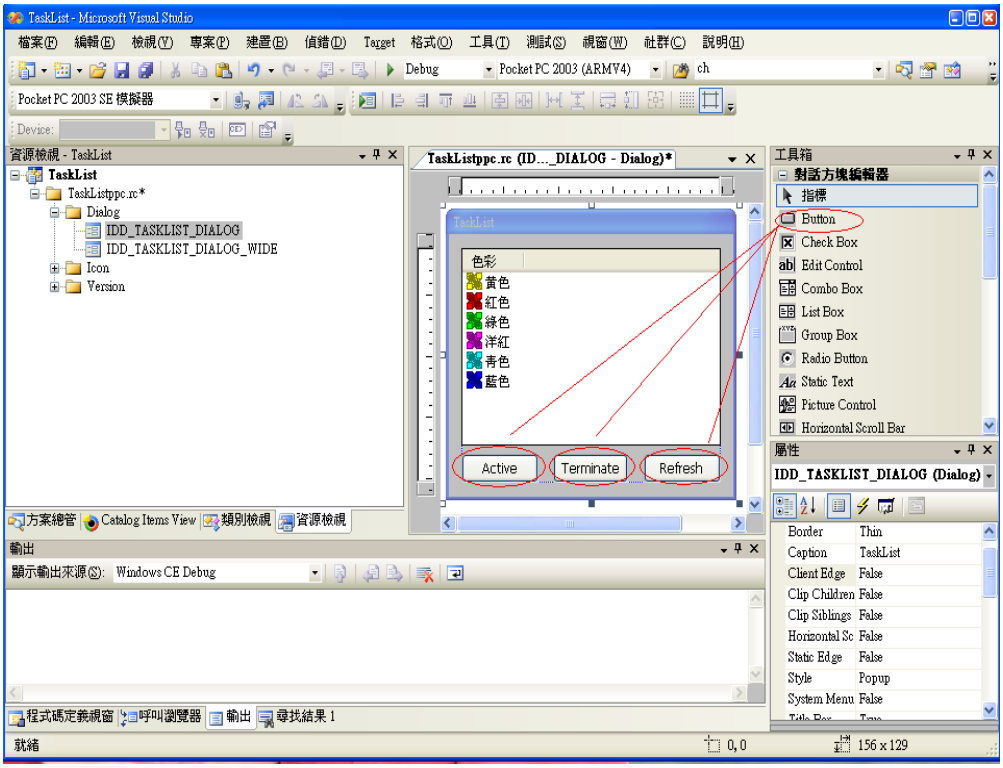
D. 修改屬性對話窗中 **Single Selection** 項為**True**。



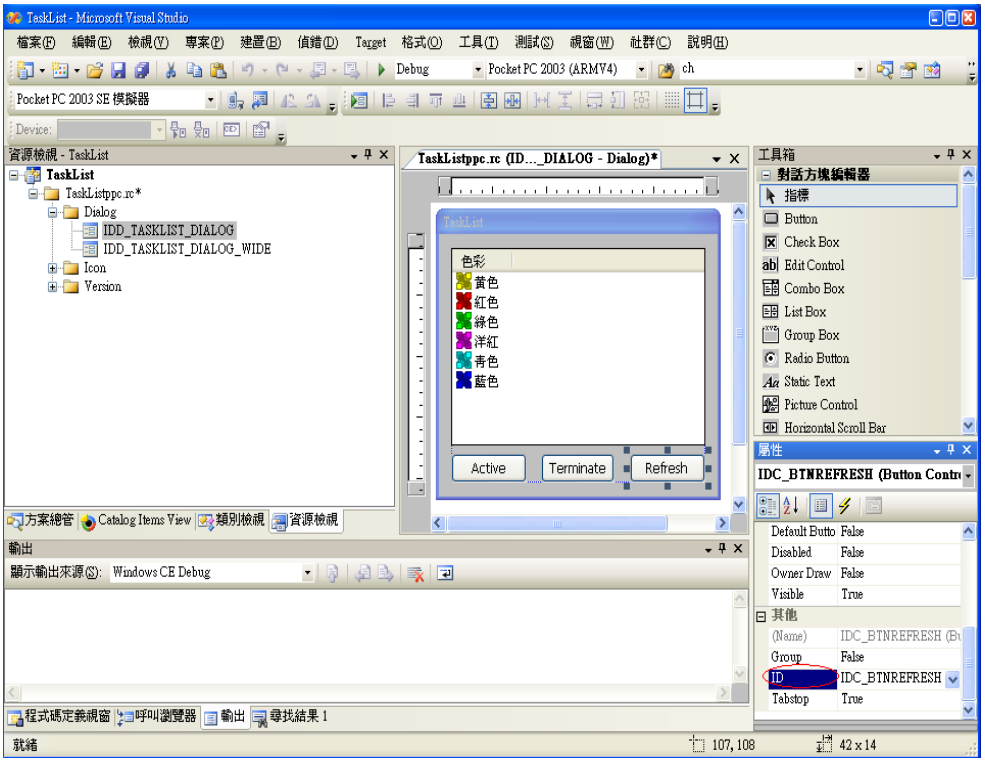
E. 修改屬性對話窗中**ID** 項為**IDC_LISTPROCESS**。



F. 然後將新選取［**Button**］對話方塊三個拖曳到適當位置，分別修改屬性對話窗中 **Caption** 項為**Active**、**Terminate** 和**Refresh**。

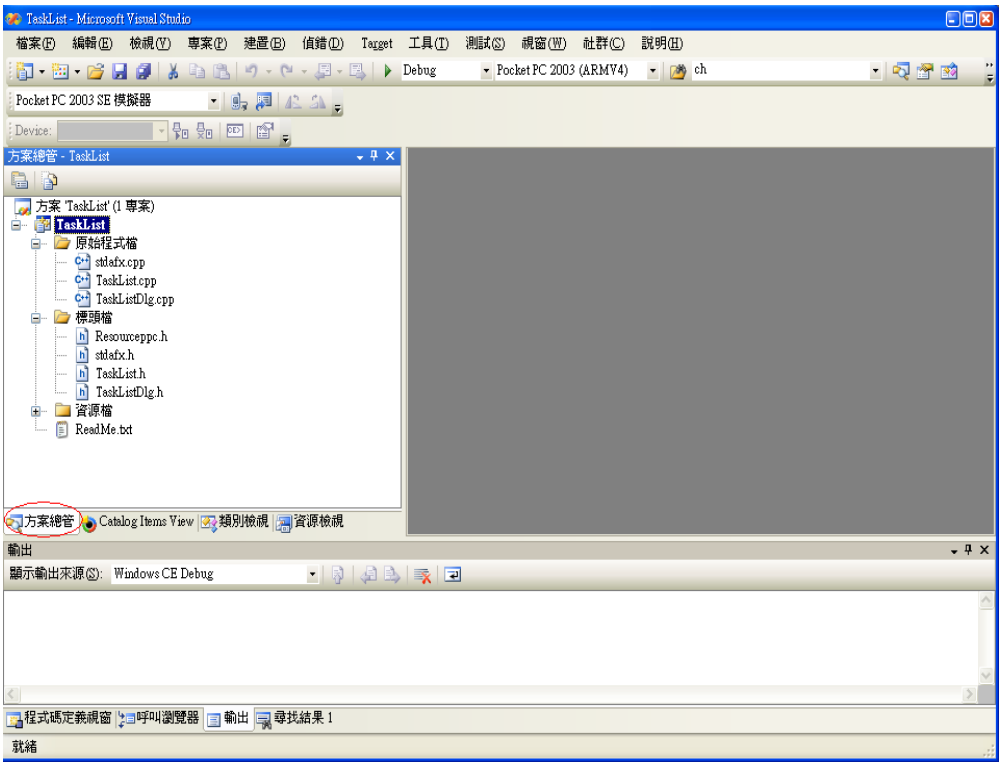


G. 再分別修改屬性對話窗中 **ID** 項為**IDC_BTNACTIVE**、**IDC_BTNTERMINATE** 和**IDC_BTNREFRESH**。



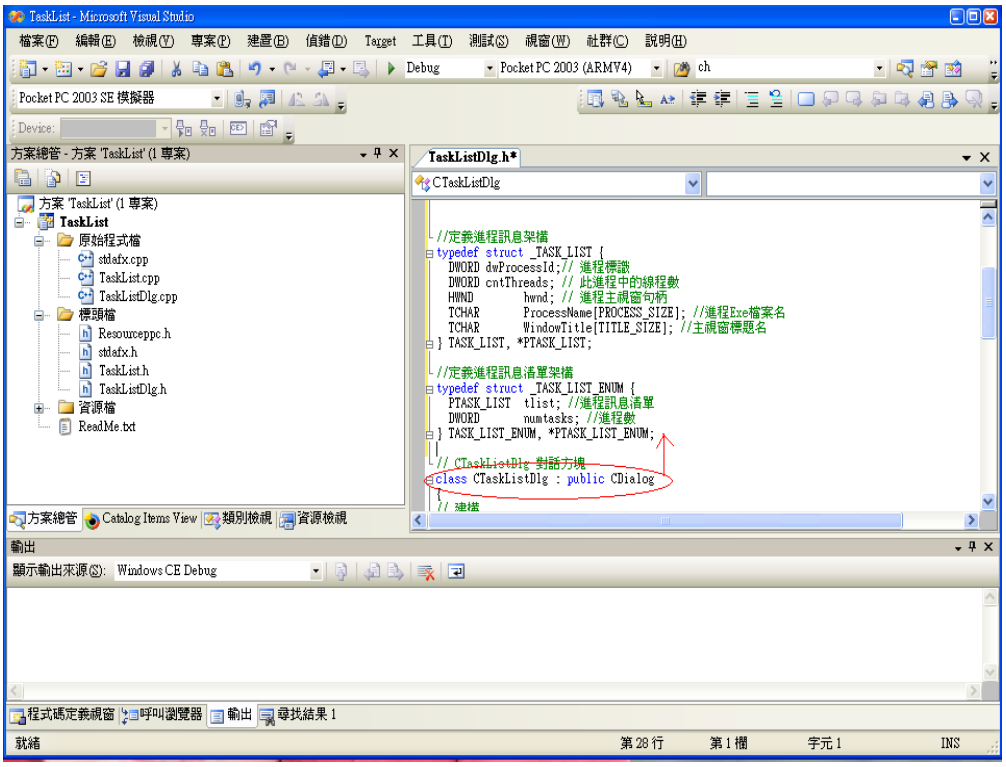
3. 程式設計

A. 切換左方視窗到方案總管，準備開始設計使用者程式。



B. 在方案總管視窗中的原始程式檔內，開啟 **TaskListDlg.h** ，在原始程式

碼**class CTaskListDlg : public CDialog** 前面加入一段使用到的常數、結構及變數定義。

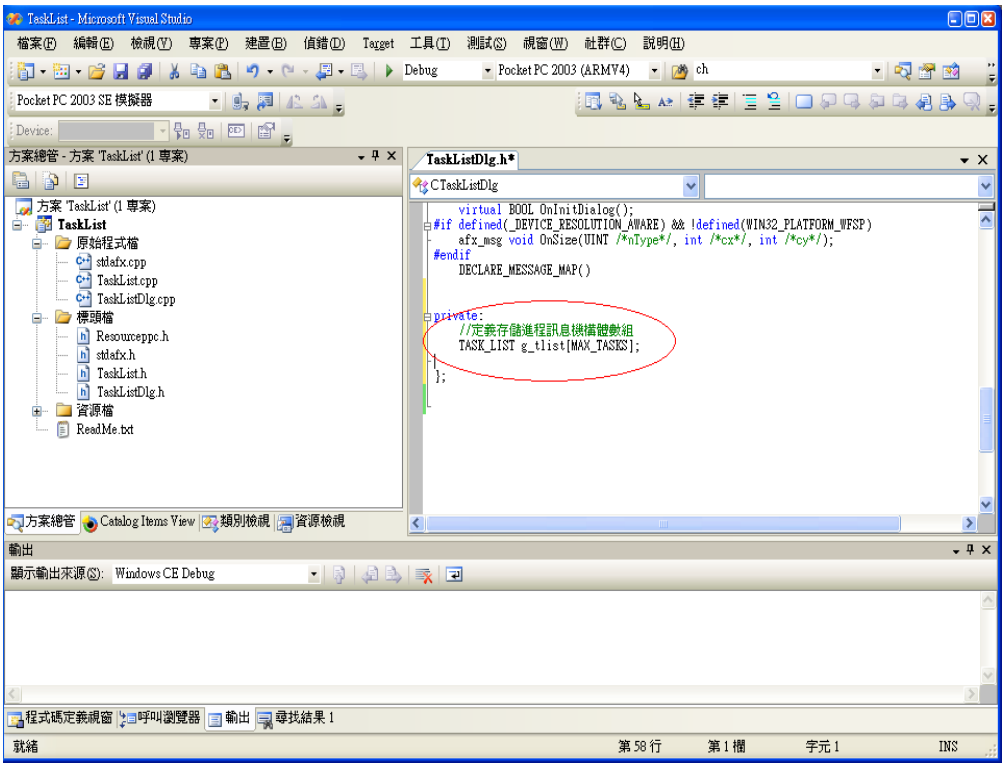


加入的程式碼 ：

```
//  定義用到的常數
#define TITLE_SIZE          64          //  標題字元串的大小
#define PROCESS_SIZE        MAX_PATH    //  處理序名字元串的大
#define MAX_TASKS           25          //  最大處理序數

//  定義進程訊息結構
typedef struct _TASK_LIST {

DWORD   cntThreads;                     //  此處理序中的執行緒
HWND    hwnd;                           //  處理序主視窗代號
TCHAR   ProcessName[PROCESS_SIZE];      //  處理序Exe檔案名
TCHAR   WindowTitle[TITLE_SIZE];        //  主視窗標題名
} TASK_LIST, *PTASK_LIST;

//  定義處理序訊息清單結構
typedef struct _TASK_LIST_ENUM {
PTASK_LIST  tlist;                      //  處理序訊息清單
DWORD       numtasks;                   //  處理序數
```

*} TASK_LIST_ENUM, *PTASK_LIST_ENUM;*

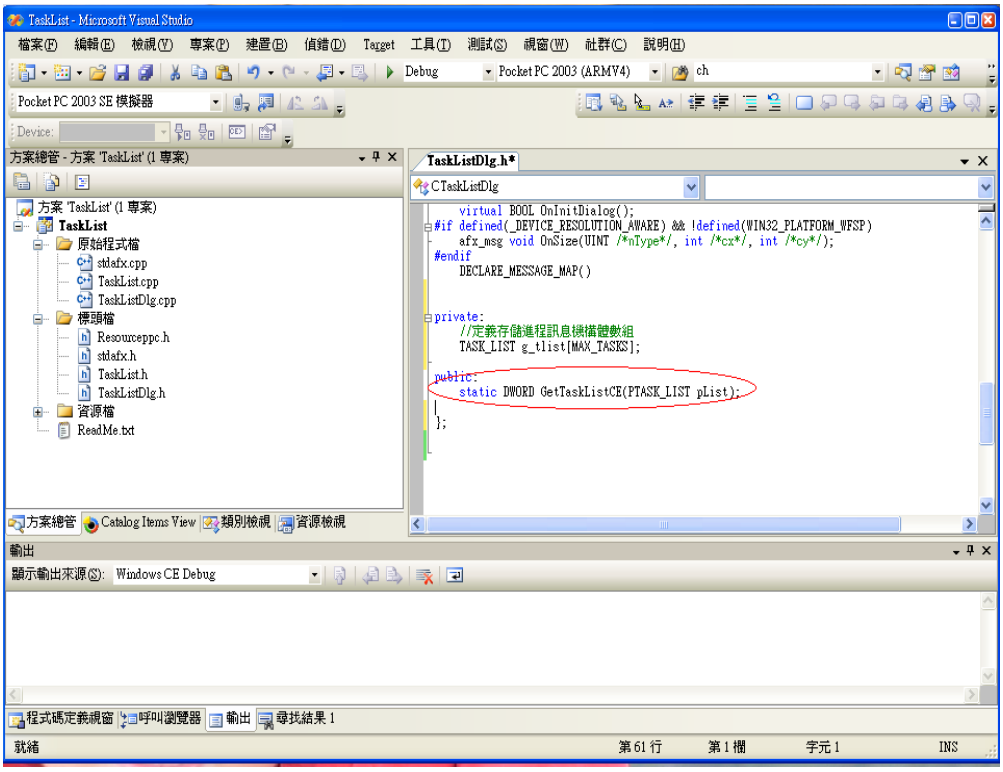
C. 在開啟的 **TaskListDlg.h** 原始程式碼**Class CtaskListDlg** 中，加入定義的**private** 變數 **g_tlist**。



加入的程式碼 ：

private:
//定義存儲處理序訊息機構體數組
TASK_LIST g_tlist[MAX_TASKS];

D. 在開啟的 **TaskListDlg.h** 原始程式碼**Class CtaskListDlg** 中，加入定義的**public** 靜態函數 **GetTaskListCE**。

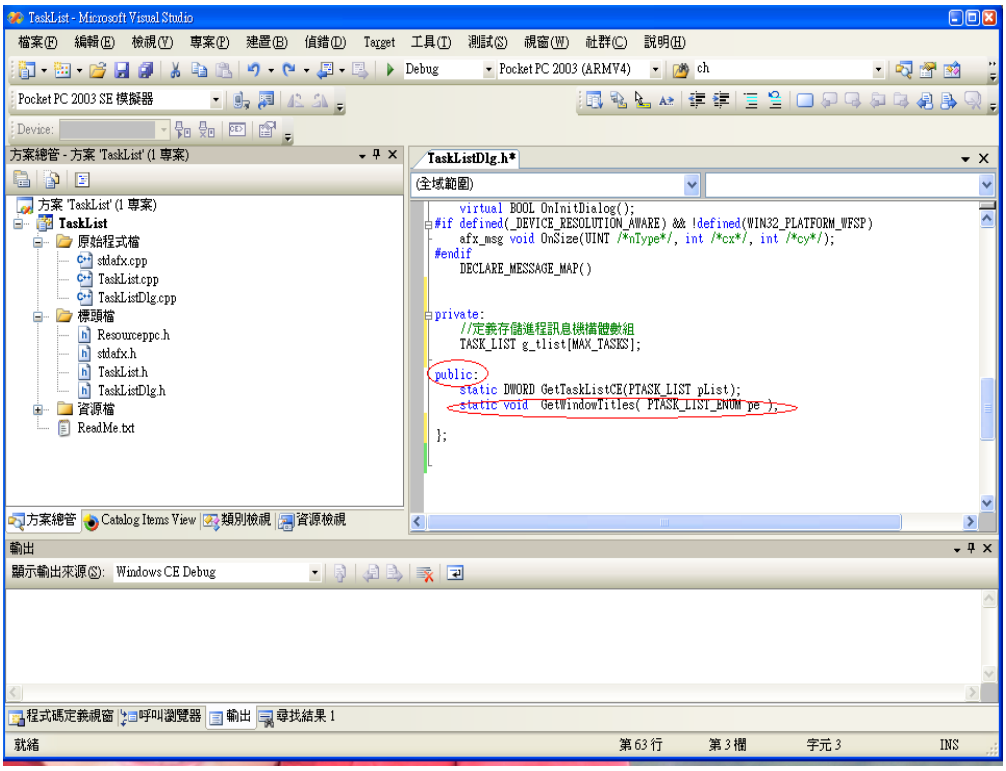


加入的程式碼 ：

public:

static DWORD GetTaskListCE(PTASK_LIST pList);

E. 在開啟的 **TaskListDlg.h** 原始程式碼**Class CtaskListDlg** 中，加入定義的**public** 靜態函數 **GetWindowTitles**。

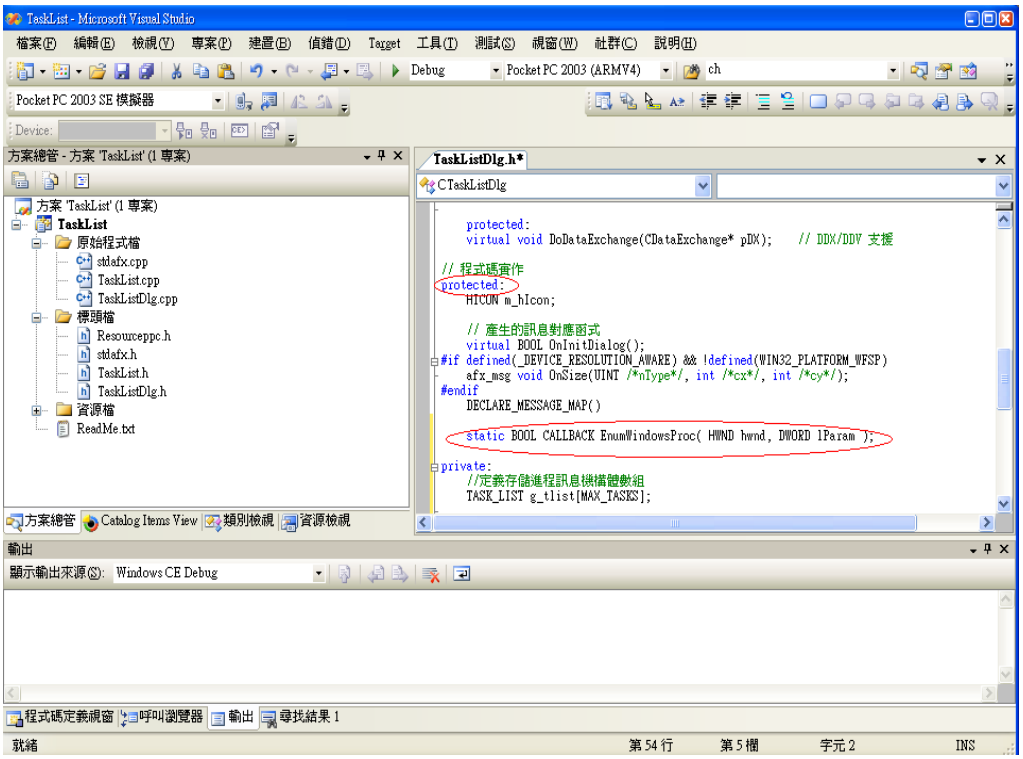


加入的程式碼 ：

public:

static void GetWindowTitles(PTASK_LIST_ENUM pe);

F. 在開啟的 **TaskListDlg.h** 原始程式碼**Class CtaskListDlg** 中，加入定義的**protected** 靜態函數 **EnumWindowsProc**。

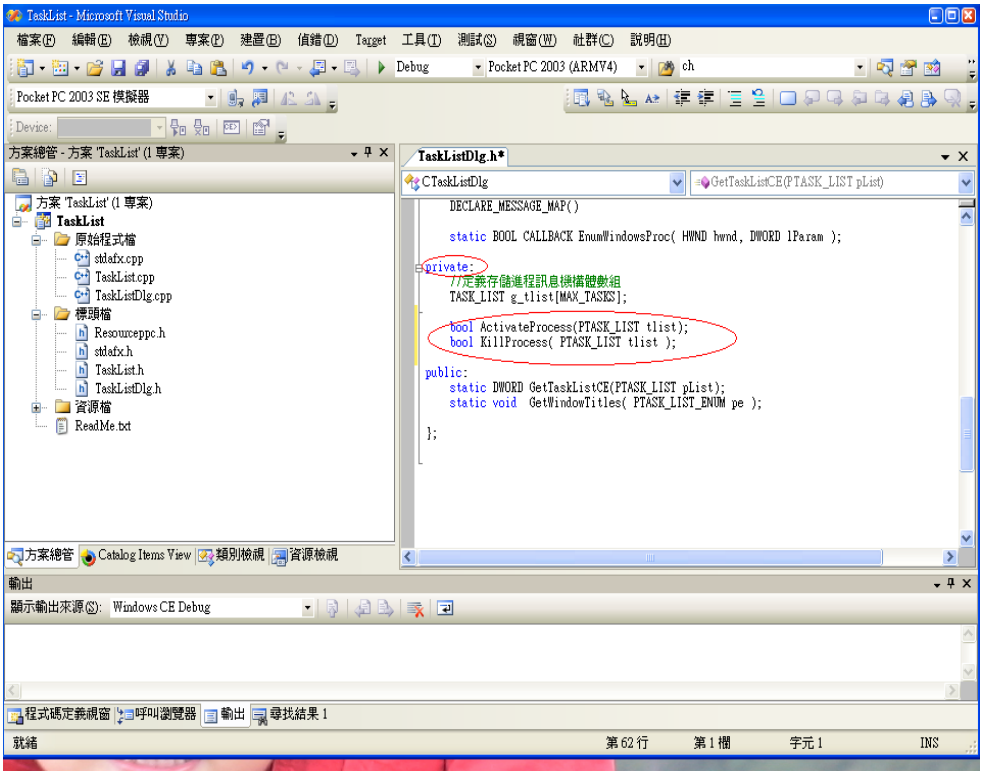


加入的程式碼 ：

protected:

static BOOL CALLBACK EnumWindowsProc(HWND hwnd, DWORD lParam);

G. 在開啟的 **TaskListDlg.h** 原始程式碼**Class CtaskListDlg** 中，加入定義的**private** 方法 **ActivateProcess** 和 **KillProcess**。

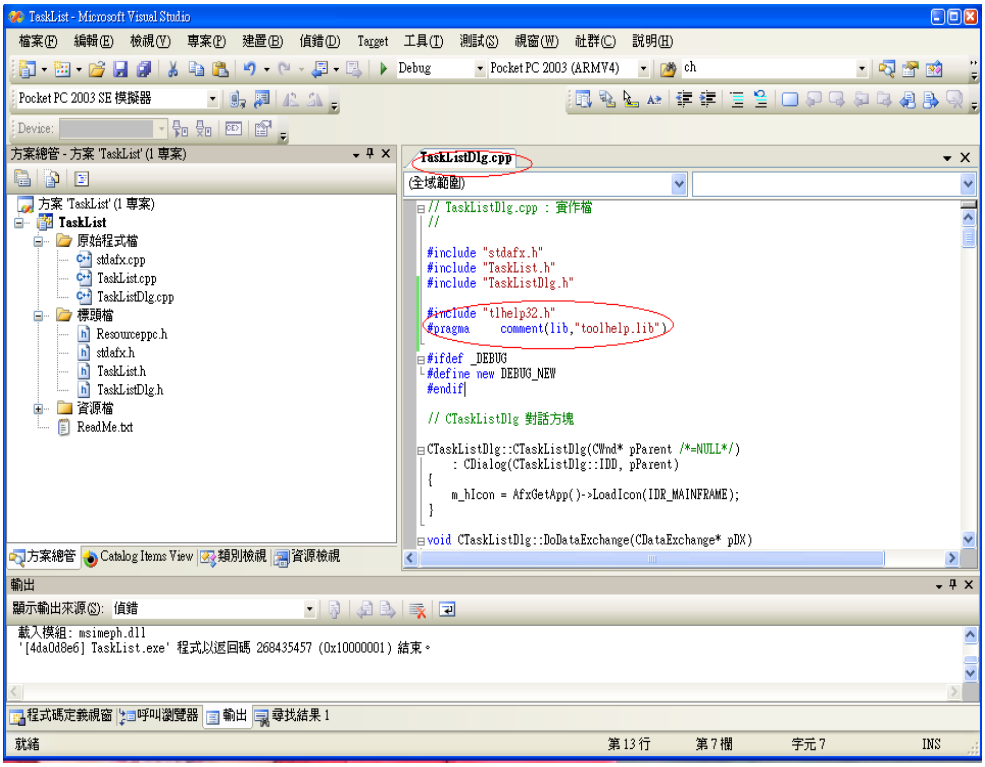


加入的程式碼 ：

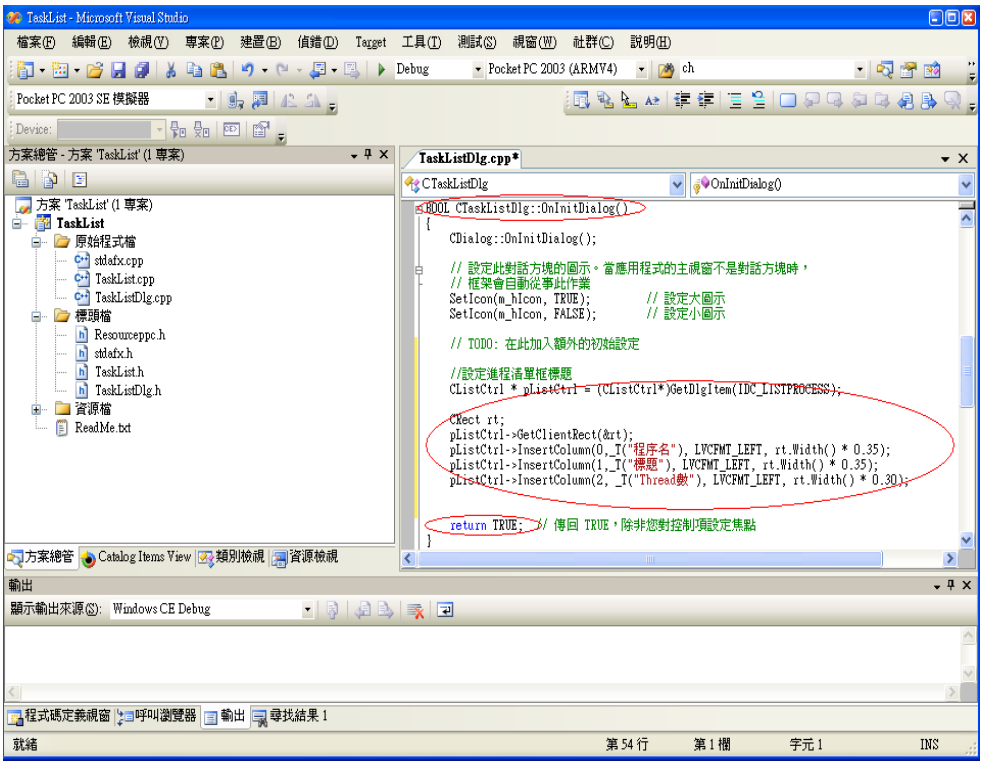
bool ActivateProcess(PTASK_LIST tlist);

bool KillProcess(PTASK_LIST tlist);

H. 在方案總管視窗中的原始程式檔內，開啟 **TaskListDlg.cpp** ，在原始程式碼前置處理器宣告後面加入需要用到的**header** 檔 **thelp32.h** 及 **Library** 檔 **toolhelp.li**b。



I. 在方案總管視窗中的原始程式檔內，開啟 **TaskListDlg.cpp** ，在原始程式碼**BOOL CTaskListDlg::OnInitDialog()** 內的**return TRUE** 前加入一段處理 **List Control** 標題的程式碼。



加入的程式碼 ：

// 設定進程清單框標題

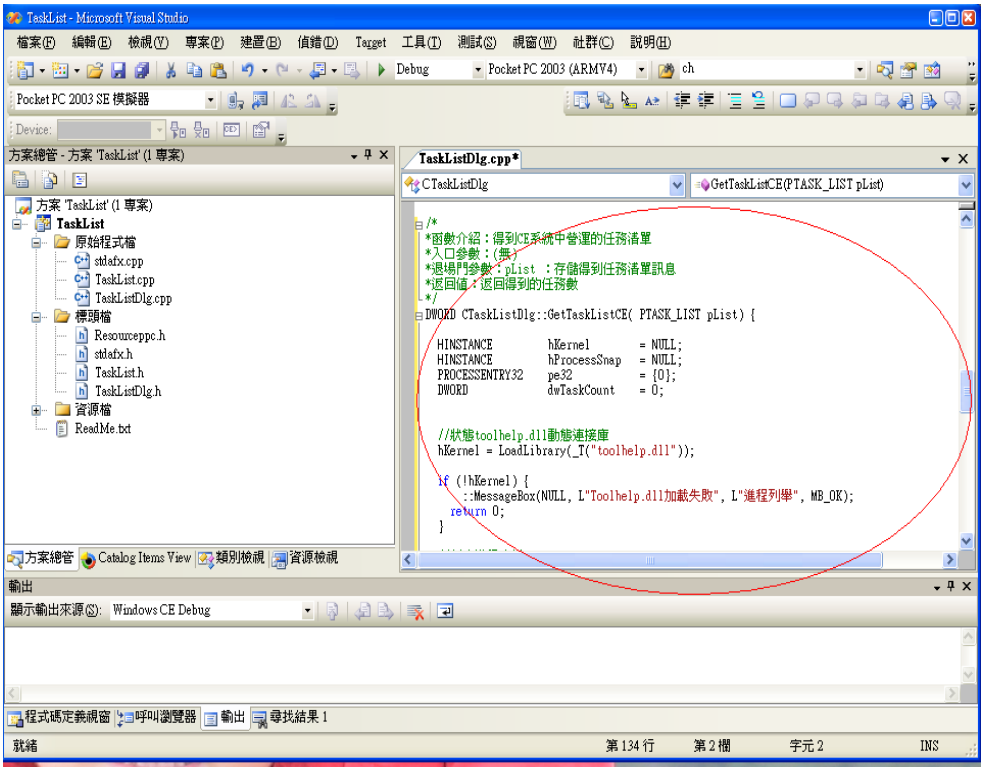
// 給定一個控制元的識別代碼以取得該對話盒控制元的視窗代碼
*CListCtrl * pListCtrl = (CListCtrl*)GetDlgItem(IDC_LISTPROCESS);*

// **CRect** 一個指向**RECT** 的**32** 位元指標
// **RECT** 是由宣告為**long** 的**Lef**t、**Top**、**Right** 和 **Bottom** 的變 數組成的結構體
CRect rt;

// 取得指定視窗代碼的工作區大小
// 取得的**Left** 及**Top** 值會是**0** ；**Right** 及**Bottom** 會是**Width** 和 **Height**
pListCtrl->GetClientRect(&rt);

// 在**ListCtrl** 中添加一列
*pListCtrl->InsertColumn(0,_T("程序名"), LVCFMT_LEFT, rt.Width() * 0.35);*
*pListCtrl->InsertColumn(1,_T("標題"), LVCFMT_LEFT, rt.Width() * 0.35);*
*pListCtrl->InsertColumn(2, _T("Thread 數"), LVCFMT_LEFT, rt.Width() * 0.30);*

J. 在開啟的 **TaskListDlg.cpp** 原始程式碼最後，加入「得到CE系統中營運 的任務清單」的靜態函數 **GetTaskListCE** 程式碼。



加入的程式碼 ：

```
//  得到CE 系統中營運的任務清單
DWORD CTaskListDlg::GetTaskListCE( PTASK_LIST pList) {

   HINSTANCE          hKernel          = NULL;
   HINSTANCE          hProcessSnap     = NULL;
   PROCESSENTRY32     pe32                = {0};
   DWORD                 dwTaskCount      = 0;


   //  載入toolhelp.dll 動態連接庫
   hKernel = LoadLibrary(_T("toolhelp.dll"));

   if (!hKernel) {
         ::MessageBox(NULL, L"Toolhelp.dll 加載失敗", L"進程列舉
                              ", MB_OK);
      return 0;
   }

   //  建立系統快照
   // CreateToolhelp32Snapshot  原型
```

// **HANDLE WINAPI CreateToolhelp32Snapshoot(**
// **DWORD dwflag,** 「指定快照中要包含的訊息類型」
// **DWORD th32ProcessID** 「用於定義一個處理序」
// **);**
// 指定快照中要包含的訊息類型：
// **TH32CS_SNAPALL**：同時指定**TH32CS_SNAPHEAPLIST**、
TH32CS_SNAPMODULE、
TH32CS_SNAPPROCESS 和
TH32CS_SNAPTHREAD 四個參數
// **TH32CS_SNAPHEAPLIST**：指定處理序中的堆積的列表
// **TH32CS_SNAPMODULE**：指定處理序中的模組的列表
// **TH32CS_SNAPPROCESS**：系統處理序訊息
// **TH32CS_SNAPTHREAD**：系統執行緒訊息
// **th32ProcessID** 如果設為**0** 則表示當前的處理序。此參數只會影響模組與堆積訊息，因為他們是與處理序相關

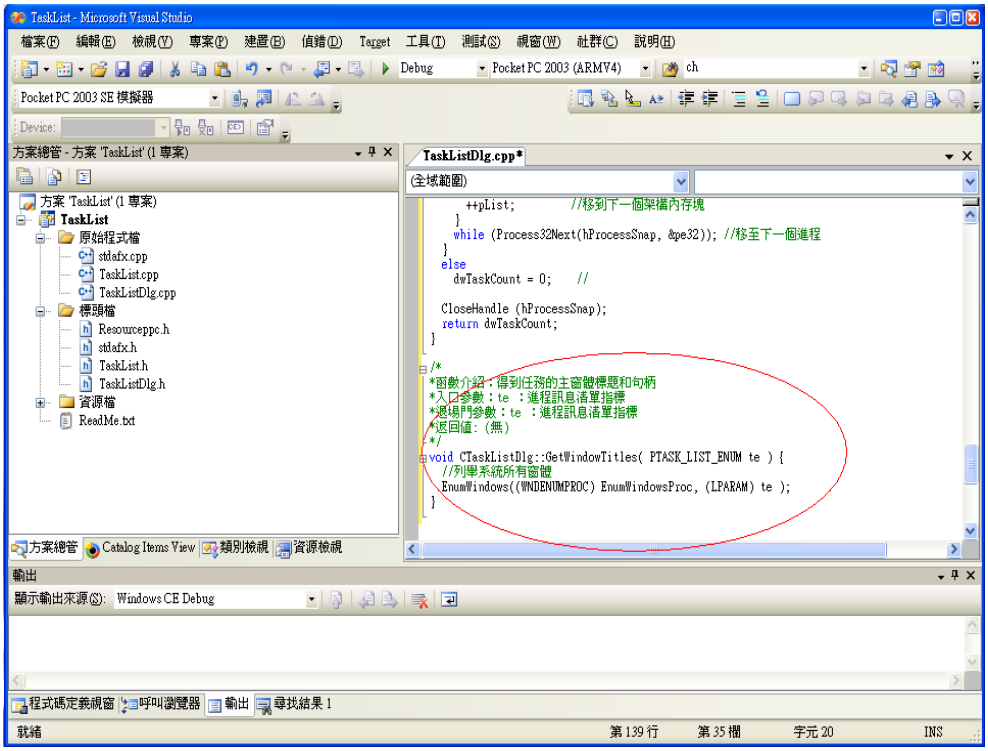
```
hProcessSnap =
(HINSTANCE)CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);

//  如果失敗，就退出
if (hProcessSnap == (HANDLE)-1)
   return 0;

dwTaskCount = 0;

//  設定儲存處理序相關內容的變量的架構大小
```

// **typedef struct tagPROCESSENTRY32{**
// **DWORD dwSize;**「**PROCESSENTRY32** 結構的實際長度」
// **DWORD cntUsage;**「處理序的引用計數，當引用計數減到0時， 作業系統就會卸載此處理序」
// **DWORD th32ProcessID;**「處理序標識」
// **DWORD th32DefaultHeapID;**「處理序中默認堆積的標識，僅在**ToolHelp** 中有意義」
// **DWORD th32ModuleID;**「模組的標識，僅在**ToolHelp** 中有意義」
// **DWORD cntThreads;**「處理序中啟動了執行緒的數目」
// **DWORD th32ParentProcessID;**「處理序的父處理序」
// **DWORD pcPriClassBase;**「處理序的優先級，**WinCE** 中，它的值將一直都是**THREAD_PRIORITY_NORMAL**」

```
// DWORD dwFlags;「保留字，未使用」
// TCHAR SzExeFile[MAX_PATH];「一個以NULL 值為結束的字
                                串，包含處理序的EXE 檔案的
                                路徑和檔名」
// DWORD th32MemoryBase;「加載可執行檔案的內存地址」
// DWORD th32AccessKey;「參考此處理序地址空間的權限」
// } PROCESSENTRY32;
pe32.dwSize = sizeof(PROCESSENTRY32);

//  獲取第一個進程，並將此進程訊息寫入進程架構的變量pe32 中

// Process32First  原型
// BOOL WINAPI Process32First (
// HANDLE hSnapshot,「由CreateToolhelp32Snapshot 返回的代碼」
// LPROCESSENTRY32 lppe「指向PROCESSENTRY32 結構指標」
// );
if (Process32First(hProcessSnap, &pe32)) {
   do {
      LPTSTR pCurChar;

      // _tcsstr（TCHAR.H）找出字串第一次出現的地方
      if(_tcsstr(pe32.szExeFile, L"\\"))

         // _tcsrchr (TCHAR.H)
         //  取得字元最後一次出現處到結尾的字串
         pCurChar = _tcsrchr(pe32.szExeFile, '\\');
      else
         pCurChar = pe32.szExeFile;

      //  複製一個字元字串至一記憶體緩衝區
      lstrcpy(pList-> ProcessName, pCurChar);



      pList ->dwProcessId = pe32.th32ProcessID;
      pList ->cntThreads = pe32.cntThreads;

      //  處理序數目加1
      ++dwTaskCount;
```

```
        //  移到下一個架構內存塊
        ++pList;
      }

      //  移至下一個處理序
      // Process32Next  原型
      // BOOL WINAPI Process32Next (
      // HANDLE hSnapshot,  「由CreateToolhelp32Snapshot 返回的代
                                碼」
      // LPROCESSENTRY32 lppe  「指向PROCESSENTRY32 結構指
                                標」
      // );
      while (Process32Next(hProcessSnap, &pe32));
    }
    else
      dwTaskCount = 0;

    //  使用完快照後，需要用CloseHandle 關閉
    CloseHandle (hProcessSnap);
    return dwTaskCount;
  }
```

K. 在開啟的 **TaskListDlg.cpp** 原始程式碼最後，加入「得到任務的視窗標題」的靜態函數**GetWindowTitles** 程式碼。



加入的程式碼 ：

// 得到任務的視窗標題

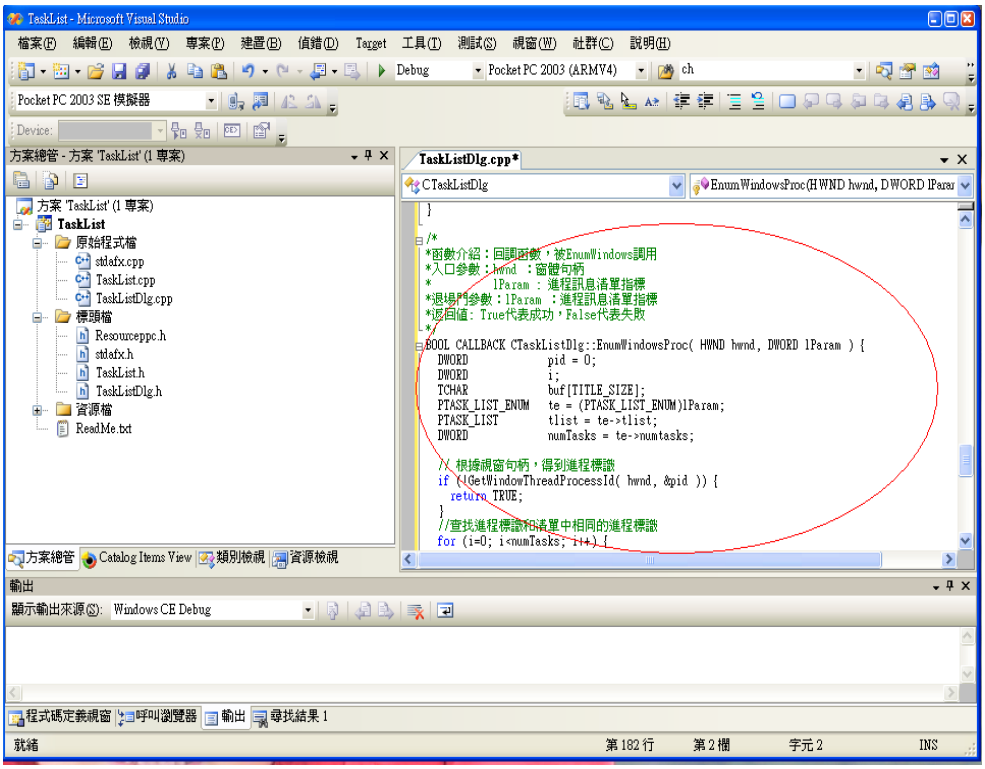
void CTaskListDlg::GetWindowTitles(PTASK_LIST_ENUM te) {

// 列舉系統所有視窗

EnumWindows((WNDENUMPROC) EnumWindowsProc, (LPARAM) te);

}

L. 在開啟的 **TaskListDlg.cpp** 原始程式碼最後，加入「被**EnumWindows**調用的回調函數，」的靜態函數**EnumWindowsProc** 程式碼。



加入的程式碼 ：

// 被**EnumWindows** 調用的回調函數，

BOOL CALLBACK CTaskListDlg::EnumWindowsProc(HWND hwnd, DWORD lParam) {

DWORD pid = 0;

DWORD i;

TCHAR buf[TITLE_SIZE];

PTASK_LIST_ENUM te = (PTASK_LIST_ENUM)lParam;

PTASK_LIST tlist = te->tlist;

DWORD numTasks = te->numtasks;

// 根據視窗代號，得到建立視窗的處理序**ID** 值和執行緒的**ID** 值

// **GetWindowThreadProcessID** 原型

// **DWORD GetWindowThreadProcessID (**

// **HWND hWnd,** 「建立視窗的代號」

// **LPDWORD lpdwProcessId** 「接收建立視窗的處理序**ID** 值」

// **);**

// 函數的返回值代表建立視窗的執行緒的**ID** 值

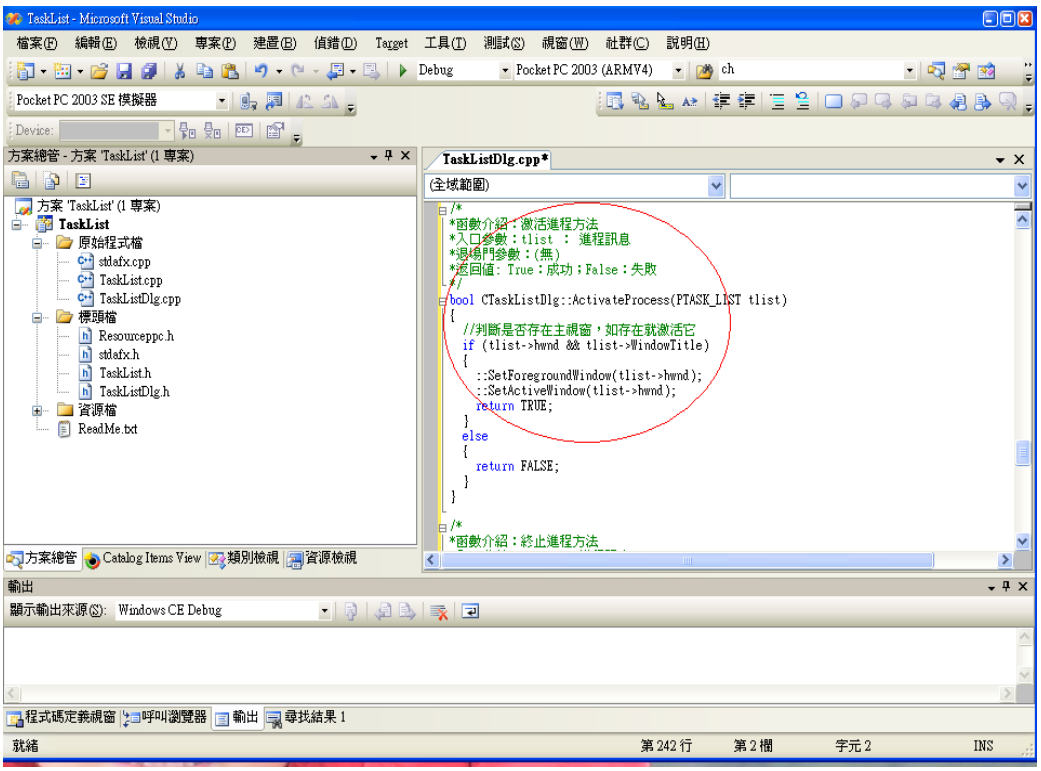
if (!GetWindowThreadProcessId(hwnd, &pid)) {

return TRUE;

}

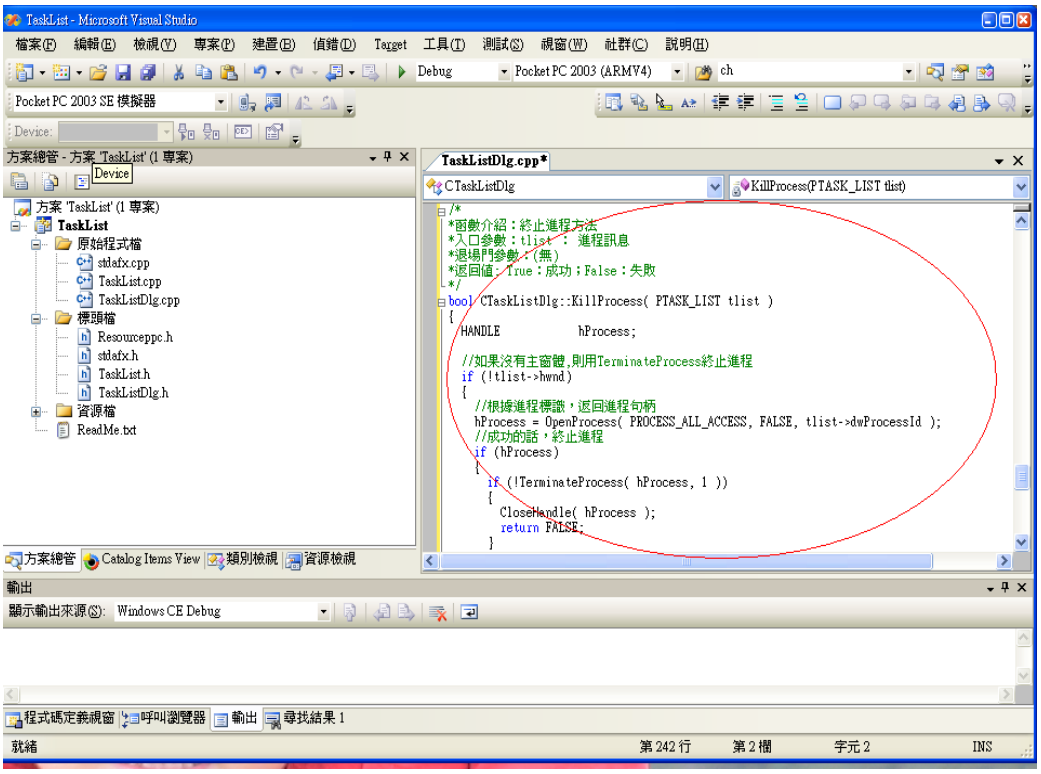
```
    //  查詢和尋找處理序ID 值和清單中相同的處理序ID 值
    for (i=0; i<numTasks; i++) {
      if (tlist[i].dwProcessId == pid) {

      // IsWindowVisible 函式可用來判斷指定的視窗代號的狀態是顯示
        或隱藏
      //  若指定的視窗為顯示狀態時，會回傳非零的數值；反之則回傳0
      if (::IsWindowVisible(hwnd)) {
          tlist[i].hwnd = hwnd;
          int nCnt = ::GetWindowText( hwnd, buf, TITLE_SIZE );
          buf[nCnt] = '\0';
          if (nCnt) {
             lstrcpy( tlist[i].WindowTitle, buf );
          }
        }
          break;
      }
    }
    //  繼續列舉窗體
    return TRUE;
  }
```

M. 在開啟的 **TaskListDlg.cpp** 原始程式碼最後，加入「Active」 和「Terminal」程序的 **ActivateProcess** 和 **KillProcess** 程式碼。



加入的程式碼 ：

```
//  Active  程序方法
bool CTaskListDlg::ActivateProcess(PTASK_LIST tlist)
{
   //  判斷是否存在主視窗，如存在就 Active 它
   if (tlist->hwnd && tlist->WindowTitle)
   {
      //  將視窗置於前臺
      ::SetForegroundWindow(tlist->hwnd);

      //  使視窗可見
      ::SetActiveWindow(tlist->hwnd);

      return TRUE;
   }
   else
   {
      return FALSE;
   }
}
```



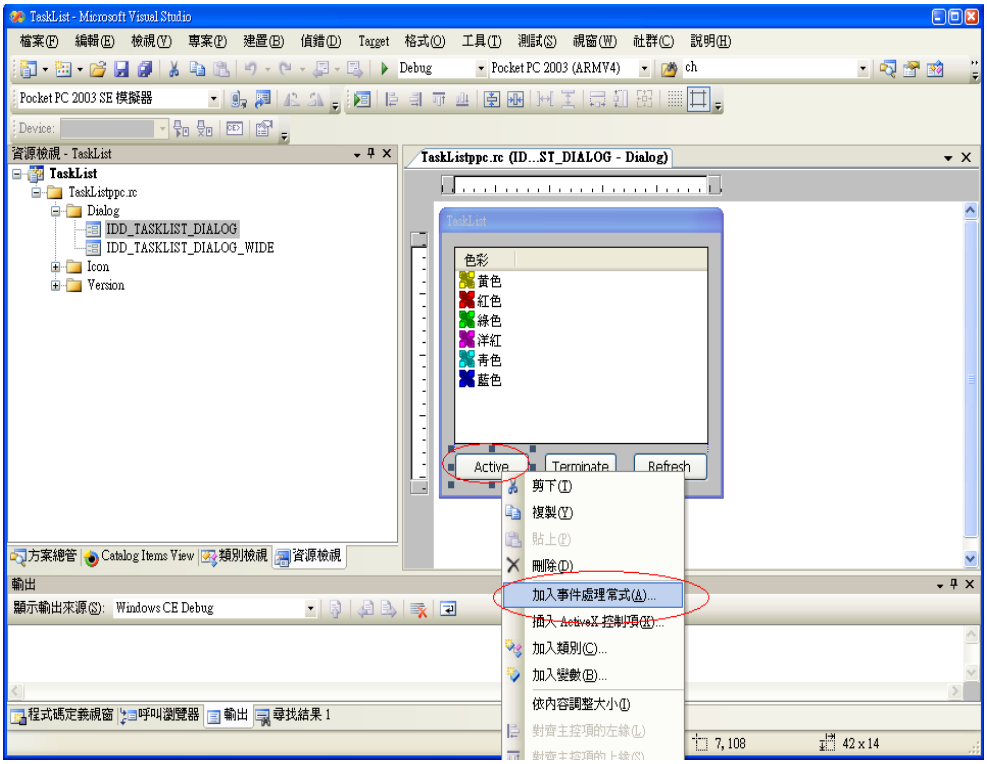
加入的程式碼 ：

// **Terminal**　程序方法

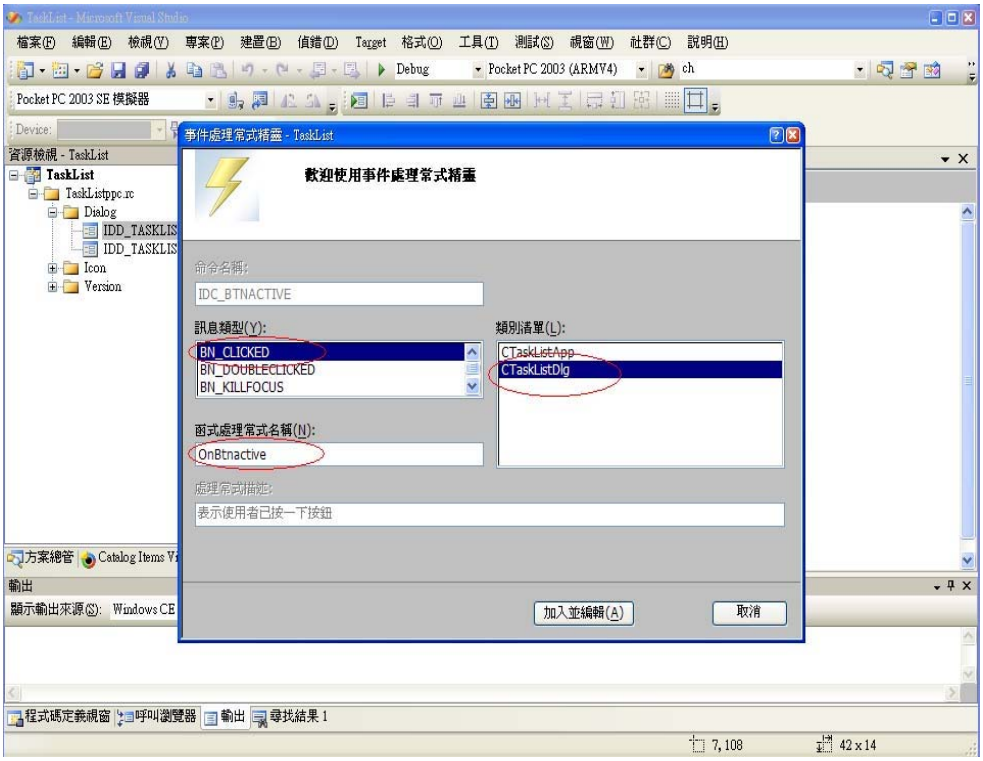
```
bool CTaskListDlg::KillProcess( PTASK_LIST tlist )
{
    HANDLE              hProcess;

    //  如果沒有主窗體,則用TerminateProcess終止處理序
    if (!tlist->hwnd)
    {
        // 根據處理序ID 值，返回處理序代號
        hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
                                tlist->dwProcessId );

        //  成功的話，終止進程
        if (hProcess)
        {
            //  中止處理序
            // TerminateProcess　原型
            // BOOL TerminateProcess (
            // HANDLE hProcess,
            // UINT uExitCode
```

```
                // );
                if (!TerminateProcess( hProcess, 1 ))
                {
                    CloseHandle( hProcess );
                    return FALSE;
                }
                CloseHandle( hProcess );
                return TRUE;
            }
            else
            {
                return FALSE;
            }
        }
        //  有主窗體，發送主視窗關閉消息，終止執行緒
        else
        {
            // PostMessage  將message 送出去之後，便馬上return
            // BOOL PostMessage (
            // HWND hWnd,  「視窗代碼」
            // UINT Msag,  「Post  訊息」
            // WPARAM wParam,  「第一個訊息參數」
            // LPARAM lParam「第二個訊息參數」
            // );
            ::PostMessage( tlist->hwnd, WM_CLOSE, 0, 0 );

            return TRUE;
        }
    }
```

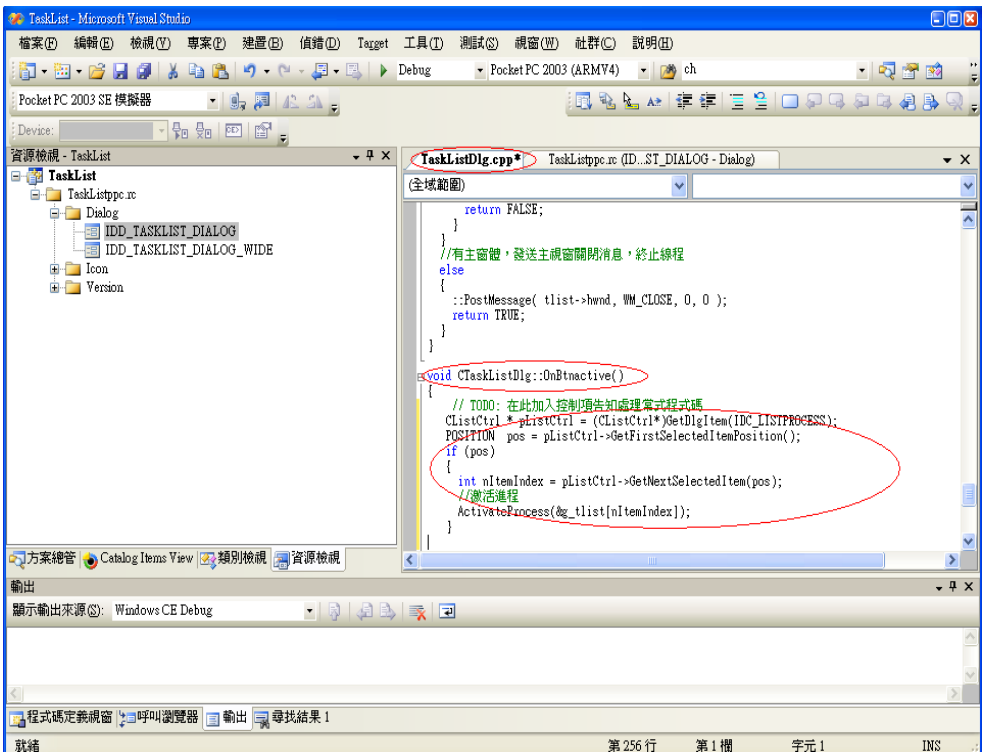
N. 切換左方視窗到資源檢視，準備開始加入事件處理常式。在要加入事件處理常式（分別為**Active**、**Terminate** 和 **Refresh**）的**Button** 對話方塊上按下滑鼠右鍵，選擇［加入事件處理常式］。



0. 選取**Active** 事件訊息類型後，更改函式處理常式名稱為 **OnBtnActive**，接下來按下［加入並編輯］按鈕。



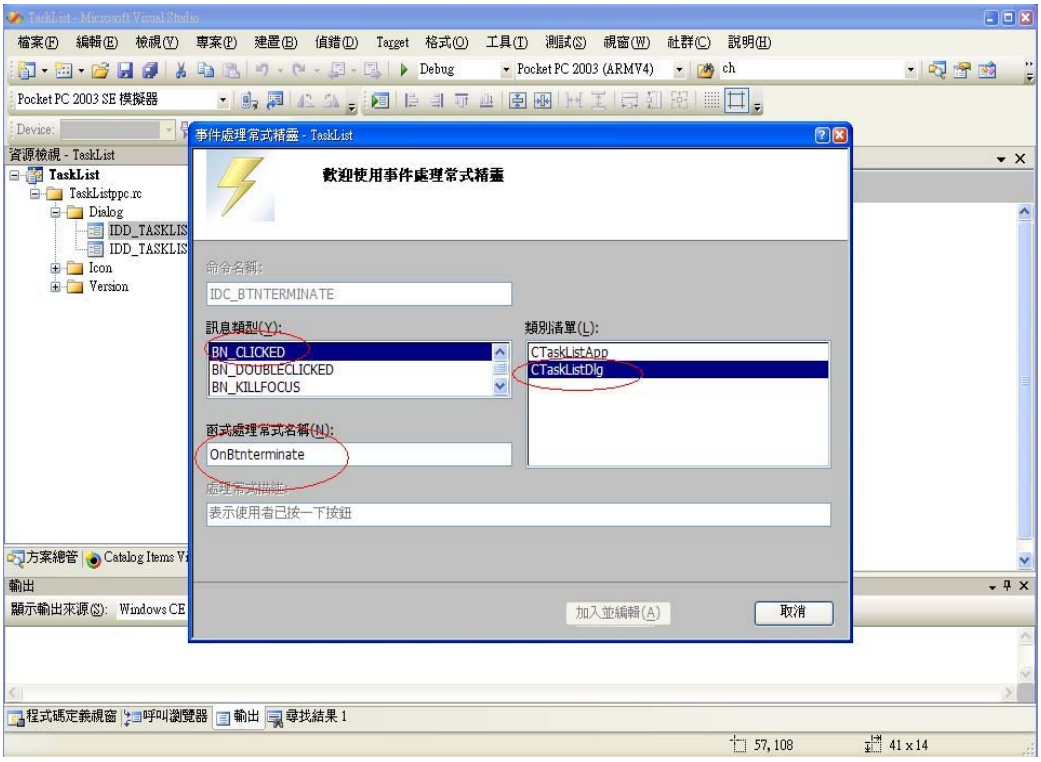
P. 程式編輯視窗開啟在**TaskListDlg.cpp**，在**void CTaskListDlg::OnBtnactive()**後，加入事件處理常式程式碼。



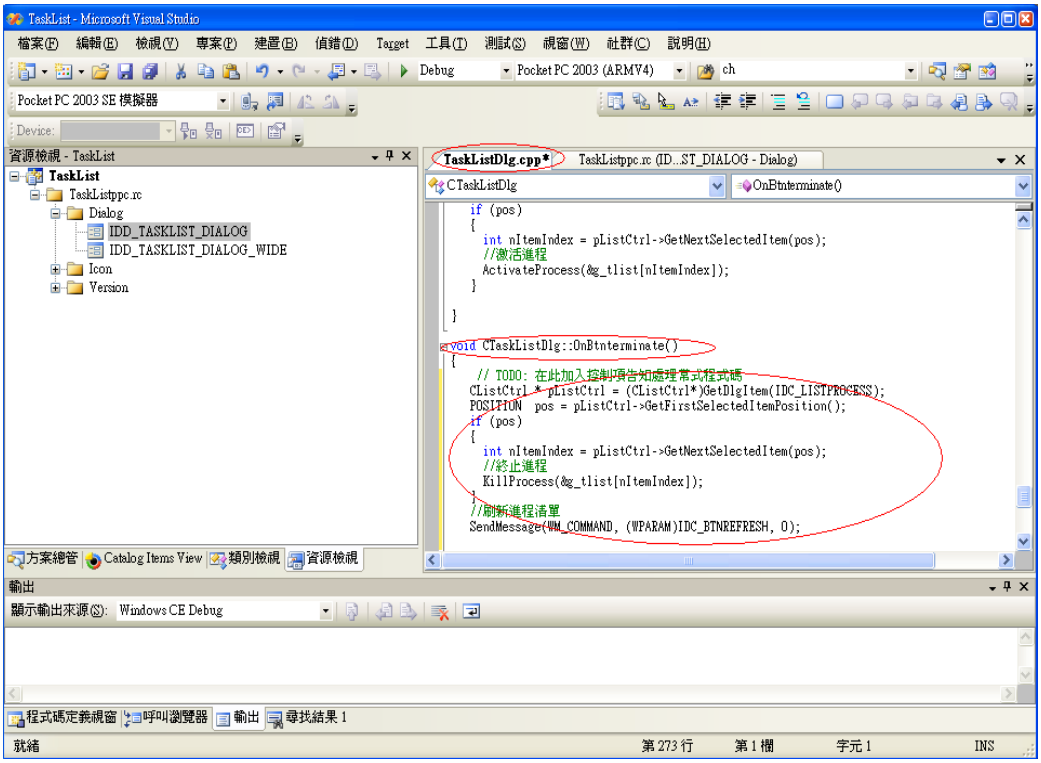
加入的程式碼 ：

```
CListCtrl * pListCtrl = (CListCtrl*)GetDlgItem(IDC_LISTPROCESS);
POSITION    pos = pListCtrl->GetFirstSelectedItemPosition();
if (pos)
{
   int nItemIndex = pListCtrl->GetNextSelectedItem(pos);

   //   激活處理序
   ActivateProcess(&g_tlist[nItemIndex]);

}
```

Q. 選取**Terminate** 事件訊息類型後，更改函式處理常式名稱為 **OnBtnterminate**，接下來按下［加入並編輯］按鈕。



R. 程式編輯視窗開啟在**TaskListDlg.cpp**，在**void CTaskListDlg::OnBtnterminate()**後，加入事件處理常式程式碼。

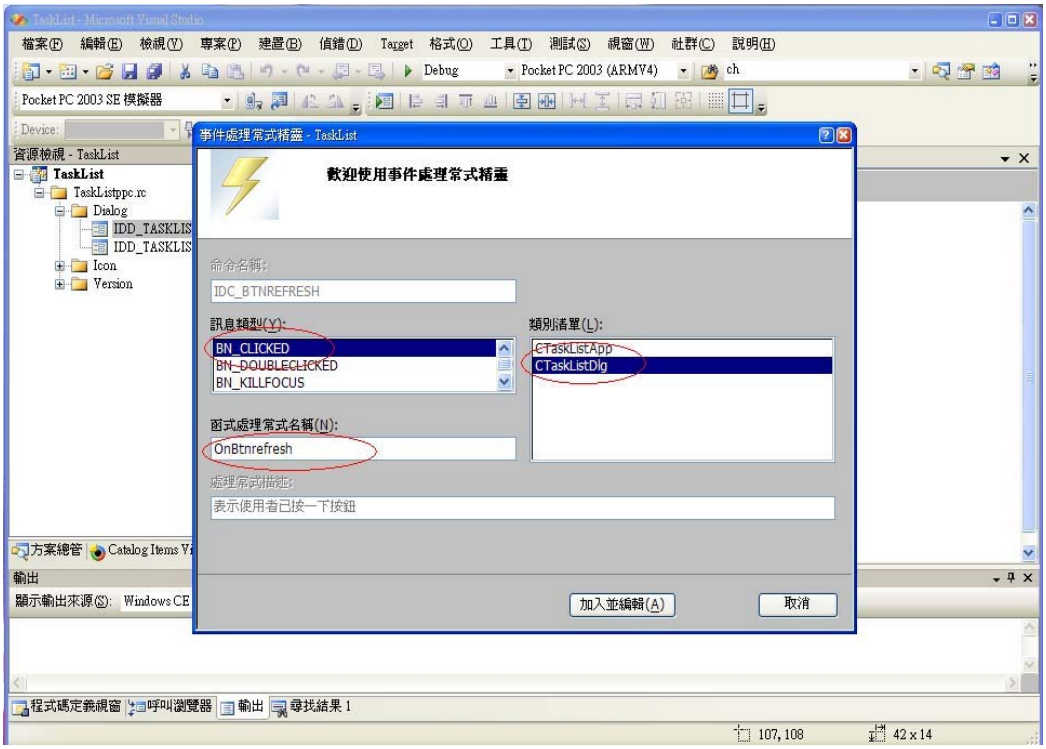


加入的程式碼 ：

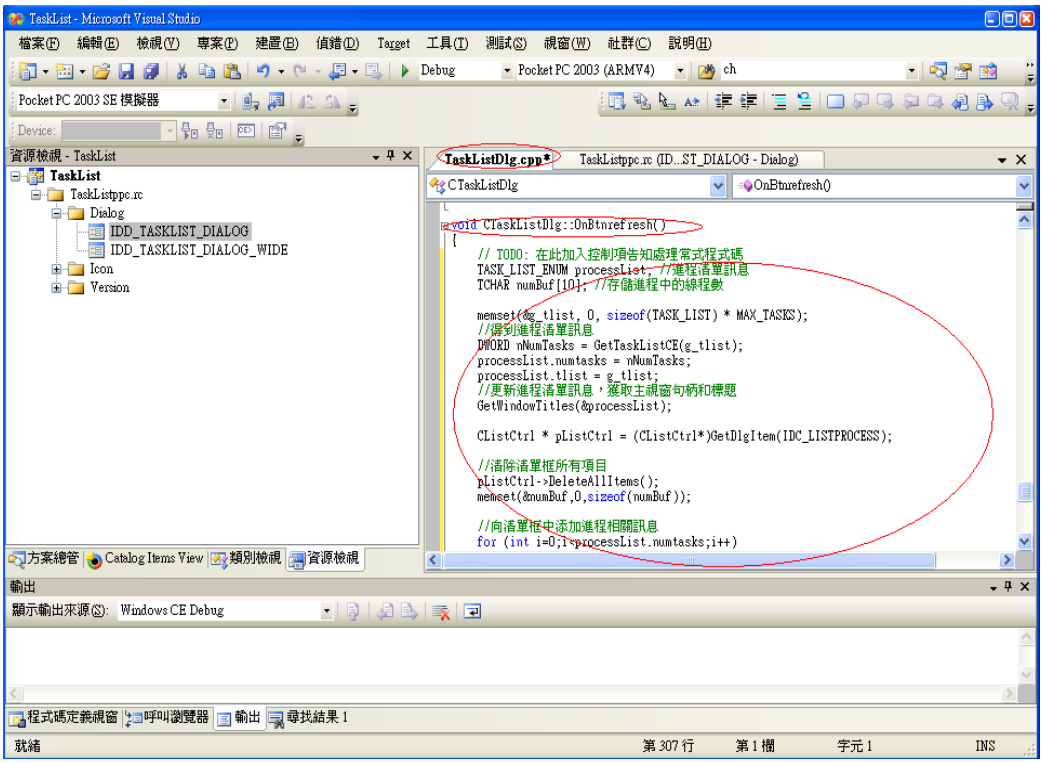
*CListCtrl * pListCtrl = (CListCtrl*)GetDlgItem(IDC_LISTPROCESS);*

```
POSITION        pos = pListCtrl->GetFirstSelectedItemPosition();
if (pos)
{
   int nItemIndex = pListCtrl->GetNextSelectedItem(pos);

  //  終止處理序
  KillProcess(&g_tlist[nItemIndex]);

}

 //  刷新處理序清單
 // SendMessage  將message 送出去之後，便會停住，等待該
    message 處理完之後，才會return
 // LRESULT SendMessage (
 // HWND hWnd,  「視窗代碼」
 // UINT Msag,  「Post  訊息」
 // WPARAM wParam,  「第一個訊息參數」
 // LPARAM lParam「第二個訊息參數」
 // );

SendMessage(WM_COMMAND, (WPARAM)IDC_BTNREFRESH, 0);
```

S. 選取**Refresh** 事件訊息類型後，更改函式處理常式名稱為 **OnBtnrefresh**，接下來按下［加入並編輯］按鈕。



T. 程式編輯視窗開啟在**TaskListDlg.cpp**，在**void CTaskListDlg::OnBtnrefresh()**後，加入事件處理常式程式碼。



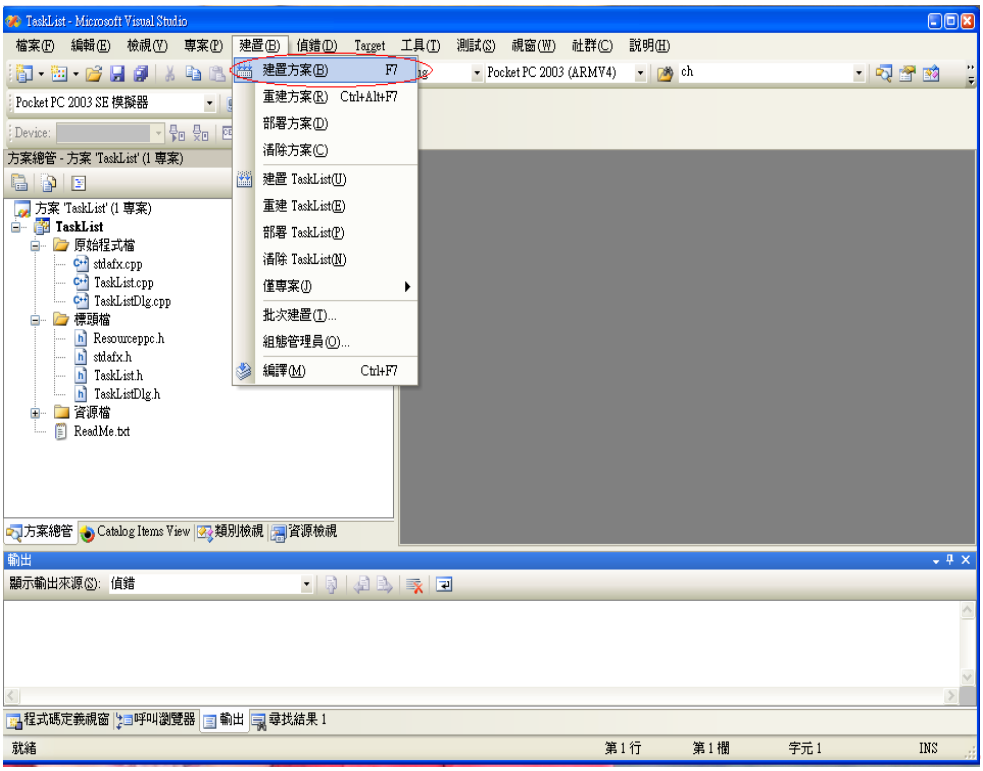
加入的程式碼 ：

// 處理序清單訊息

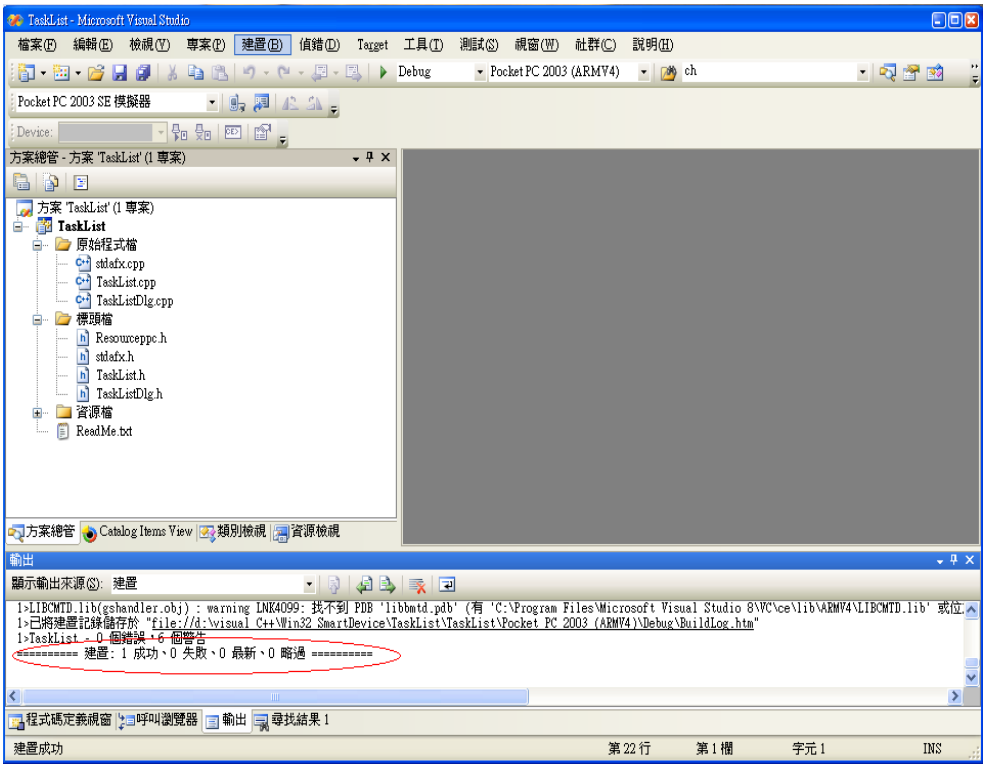
```
TASK_LIST_ENUM processList;

//  存儲處理序中的執行緒數目
TCHAR numBuf[10];

// memset  用來對一段內存空間全部設置為某個字元
// memset  原型
// extern void *memset(void *buffer, int c, int count);
//  把buffer 所指內存區域的前count 個字節設置成字元c
memset(&g_tlist, 0, sizeof(TASK_LIST) * MAX_TASKS);

//  得到處理序清單訊息
DWORD nNumTasks = GetTaskListCE(g_tlist);
processList.numtasks = nNumTasks;
processList.tlist = g_tlist;

//  更新處理序清單訊息，獲取主視窗代號和標題
GetWindowTitles(&processList);

CListCtrl * pListCtrl = (CListCtrl*)GetDlgItem(IDC_LISTPROCESS);

//  清除清單框所有項目
pListCtrl->DeleteAllItems();
memset(&numBuf,0,sizeof(numBuf));

//  向清單框中添加處理序相關訊息
for (int i=0;i<processList.numtasks;i++)
{
      //  進程exe 名，標題名，執行緒數目
     pListCtrl->InsertItem(i,_T("Test"));
     pListCtrl->SetItemText(i,0,processList.tlist[i].ProcessName);
     pListCtrl->SetItemText(i,1,processList.tlist[i].WindowTitle);
    _itow(processList.tlist[i].cntThreads,numBuf,10);
    pListCtrl->SetItemText(i,2,numBuf);
}
```

4. 編譯和執行

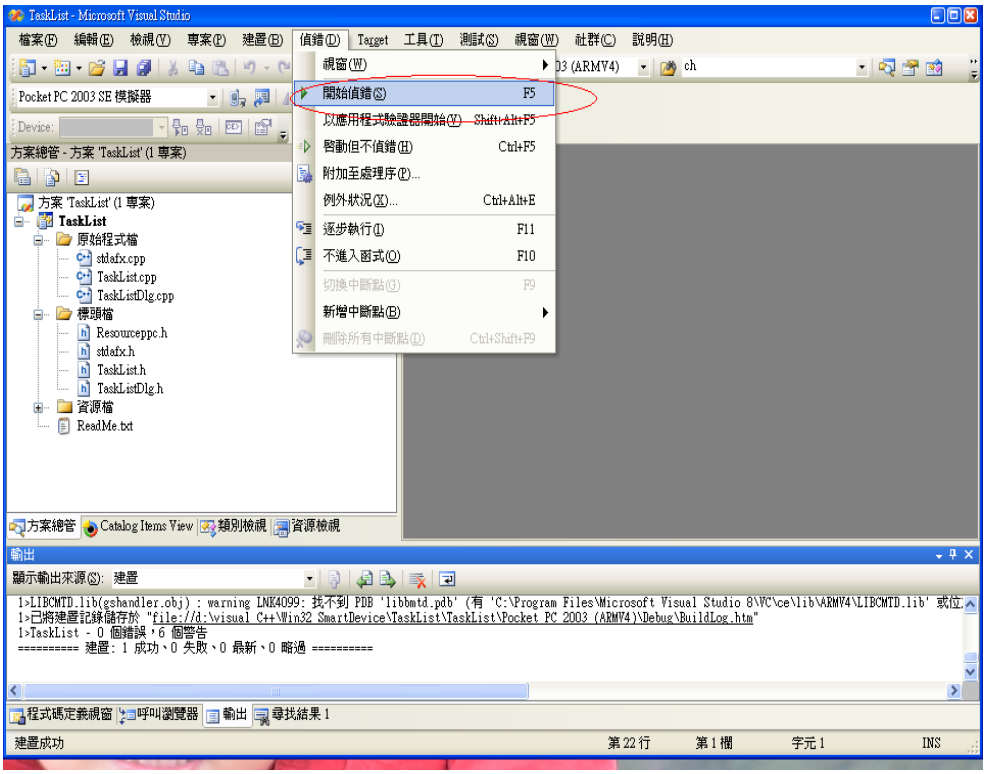
A. 在**Visual Studio 2005** 整合式開發環境中，點選[建置] -> [建置專案]後便會開始編譯整個專案。



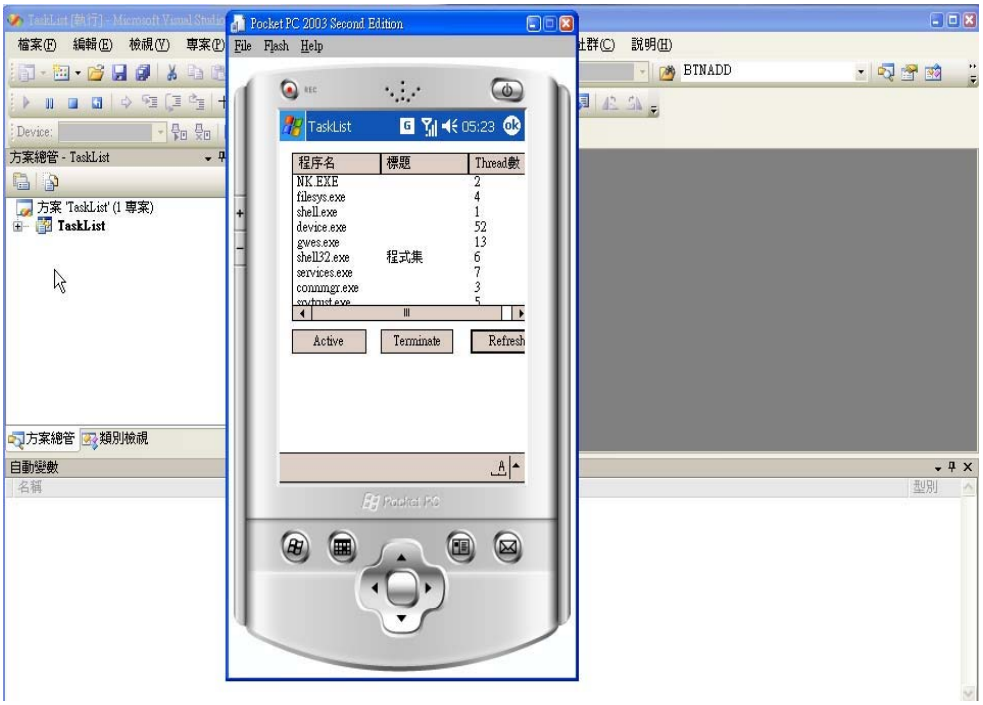
B. 編譯成功後，在**Visual Studio 2005** 整合式開發環境下方的輸出視窗中顯示建置成功。



C. 在**Visual Studio 2005** 整合式開發環境中，點選[偵錯] -> [開始偵錯]後便會開始對此專案執行並偵錯。



D. 執行結果如下圖所示。



f

應用程式開發實驗

實驗六

檔案操作

Rev. 1.0

一. 實驗目的： 在當今資訊爆炸的社會，檔案資料對於很多人來說，都是再熟悉不過的。我 們可以透過檔案的格式來存儲特定的資料。在嵌入式系統中，我們更加迫切 地需要透過檔案來存儲應用程序配置訊息或存儲應用程序所需資源數據。 檔案的基本操作主要包括新建檔案、對已經建立的檔案進行讀取或寫入等操

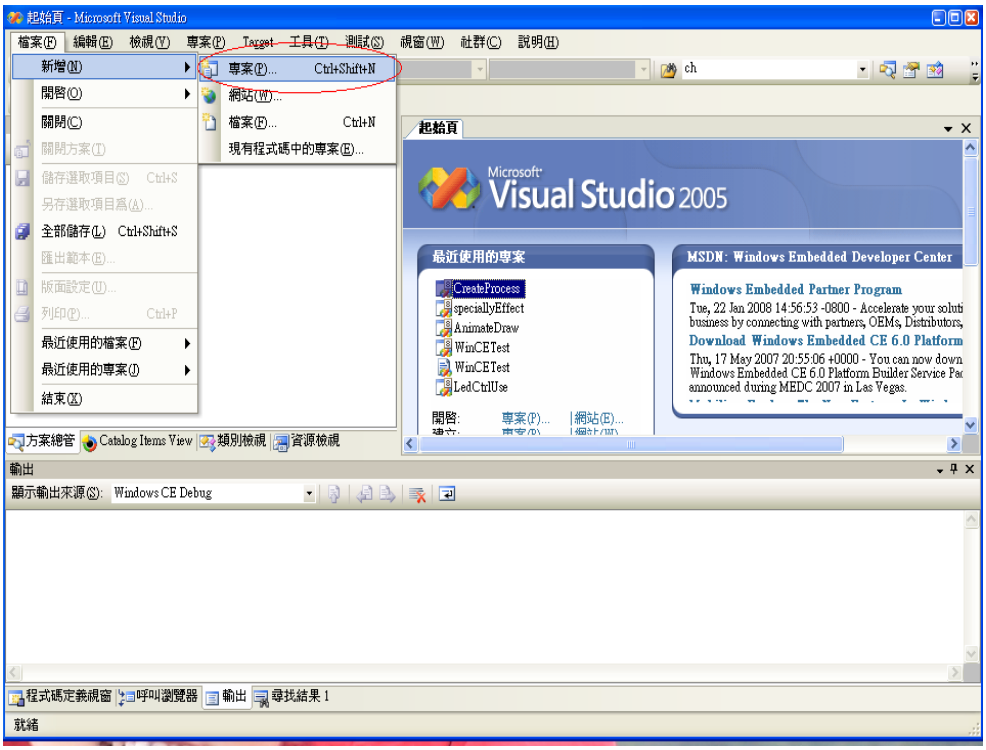
作和檔案關閉。檔案的處理方式可以分成兩種：一種是透過使用 **WINCE** 提供的 **API** 函數；另一種則是使用 **MFC** 類別庫中的 **CFile** 類別。但是實際 上兩種操作的實質是相同的，因為 **CFile** 類別只是對原始檔案操作的 **API**

函數進行封裝。

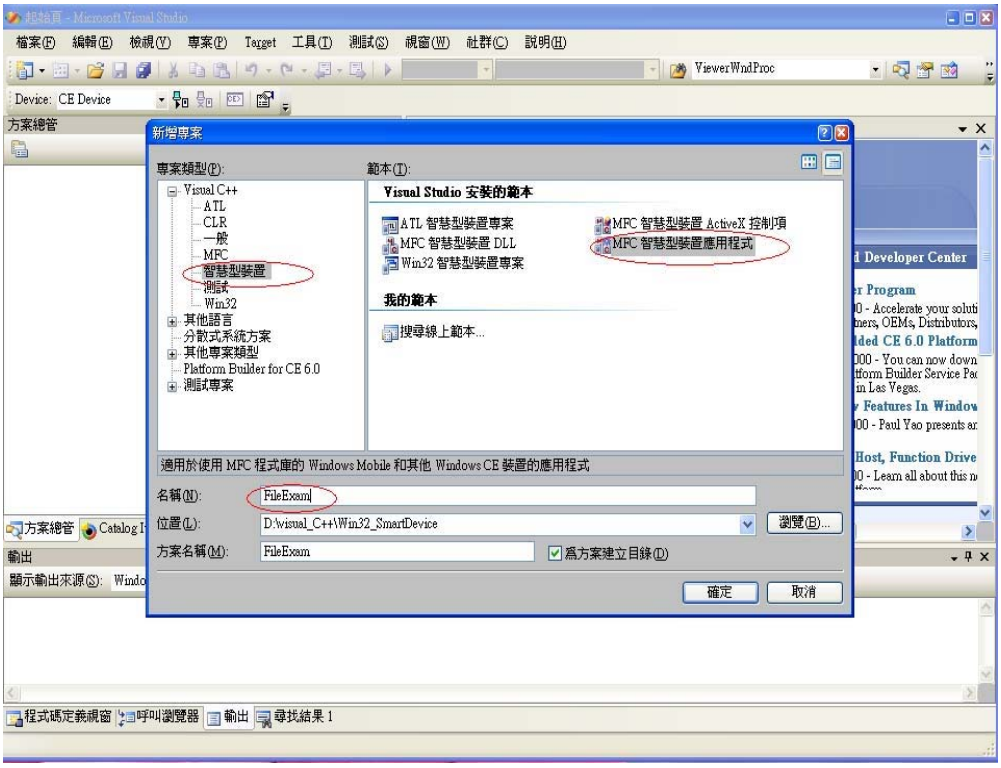
二. 實驗步驟：

1. 建立一個 **MFC** 對話方塊為基礎的專案

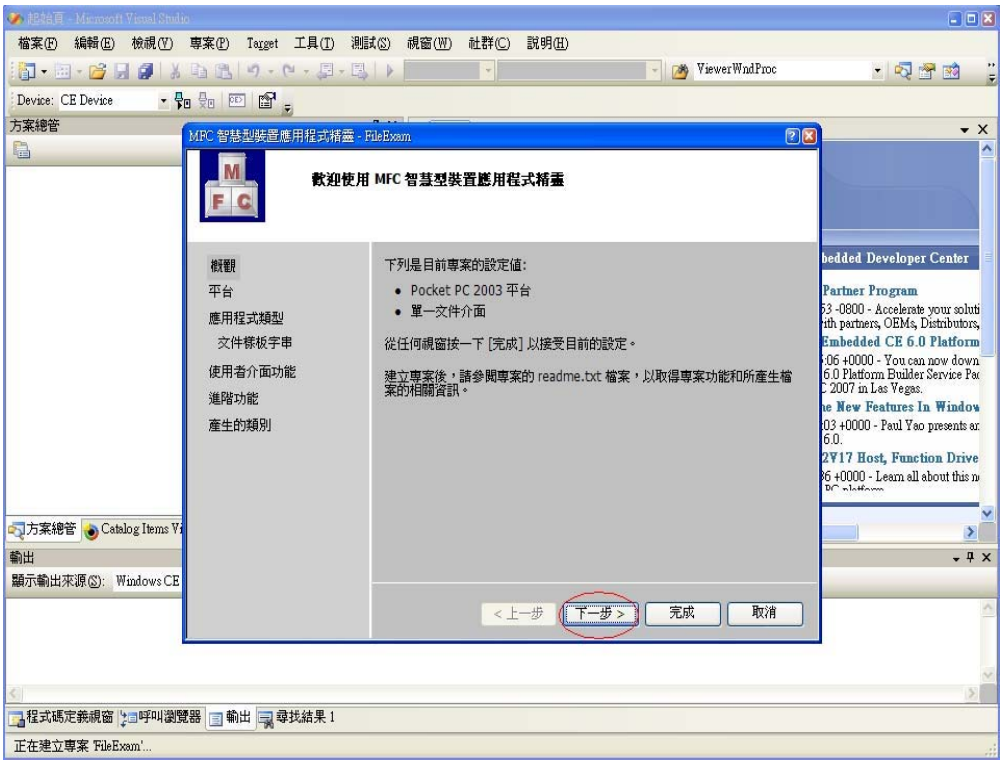
A. 在**Visual Studio 2005** 整合式開發環境中，點選左上角[檔案]->[新增]->[專案]後便會開啟新增專案視窗。



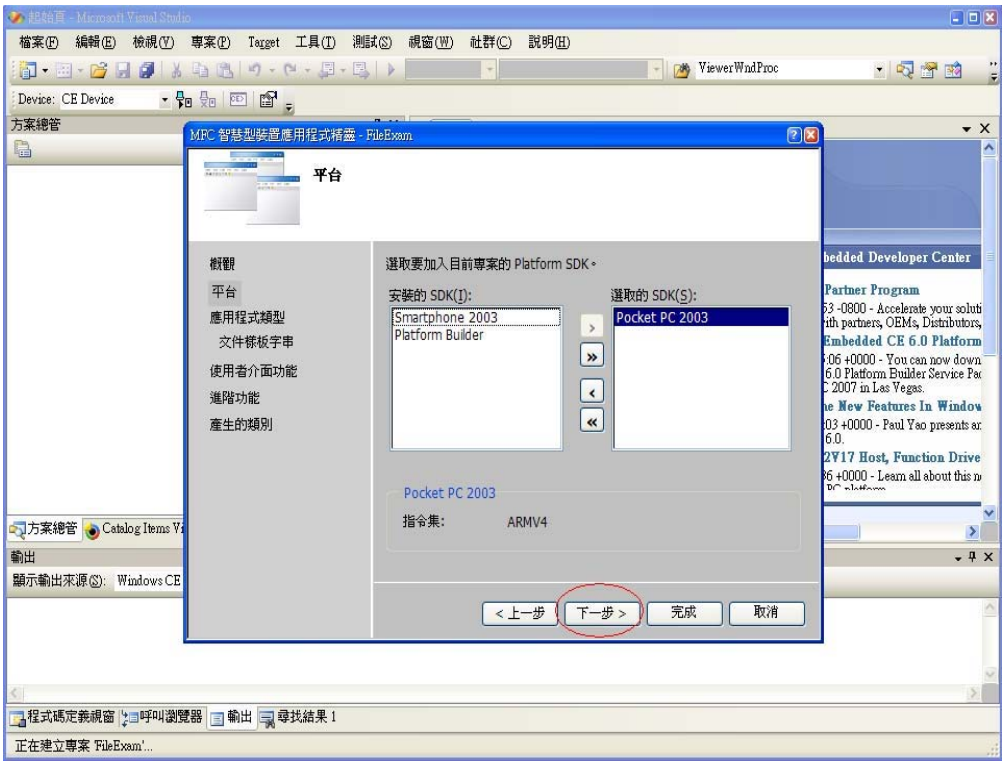
B. 在新增專案視窗內，左邊專案類型欄中選擇[智慧型裝置]；右邊 **Visual Studio** 安裝的範本欄中選擇[**MFC** 智慧型裝置應用程式]。



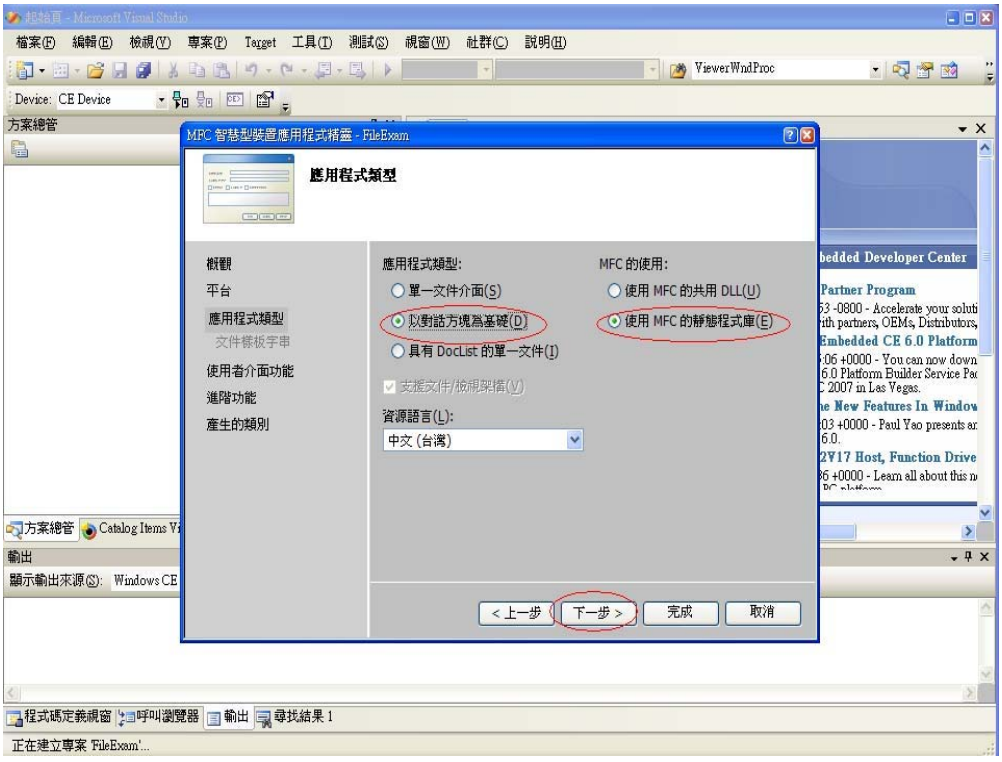
C. 在 **MFC** 智慧型裝置應用程式精靈對話窗中選擇［下一步］按鈕。



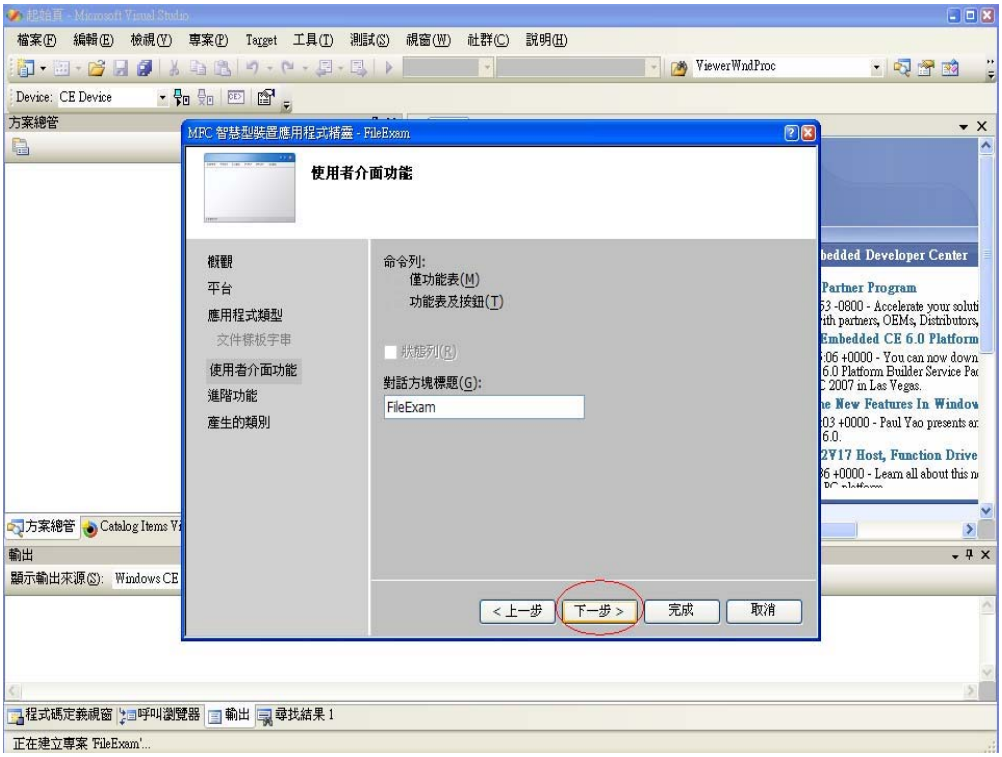
D. 選取要加入目前專案的平台軟體開發包（**Platform SDK**）在此實驗中 我們選擇[**Pocket PC 2003**]，之後按下[下一步］按鈕。



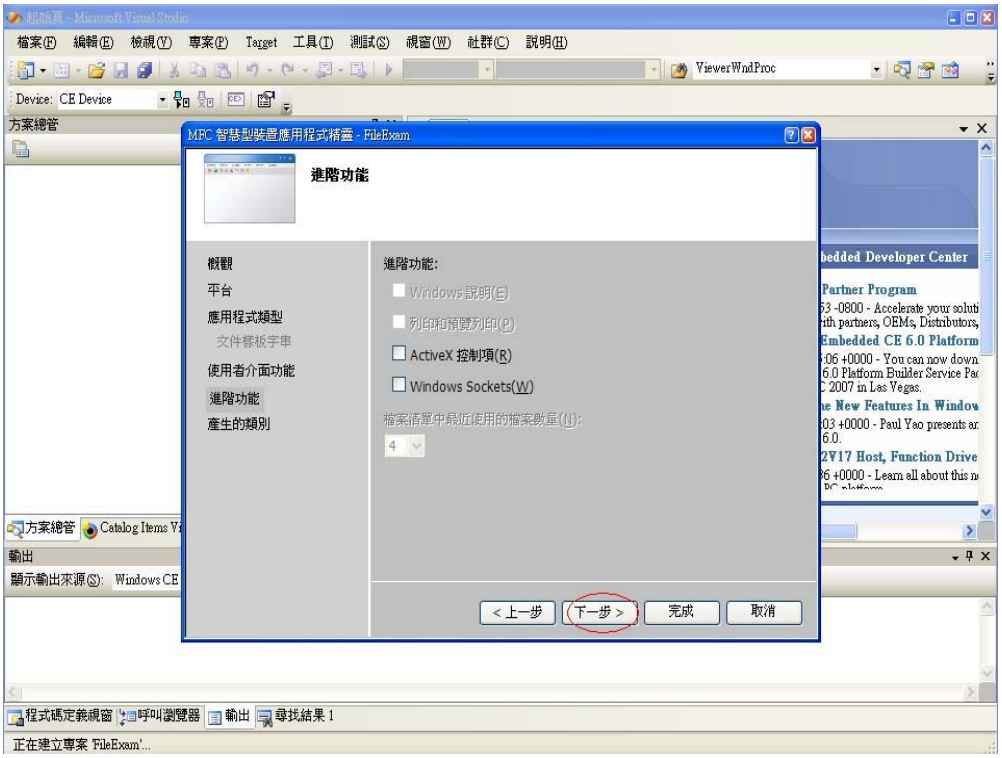
E. 應用程式類型中選取［以對話方塊為基礎］，**MFC** 的使用中選取[使用 **MFC** 的靜態程式庫］，之後按下[下一步］按鈕。



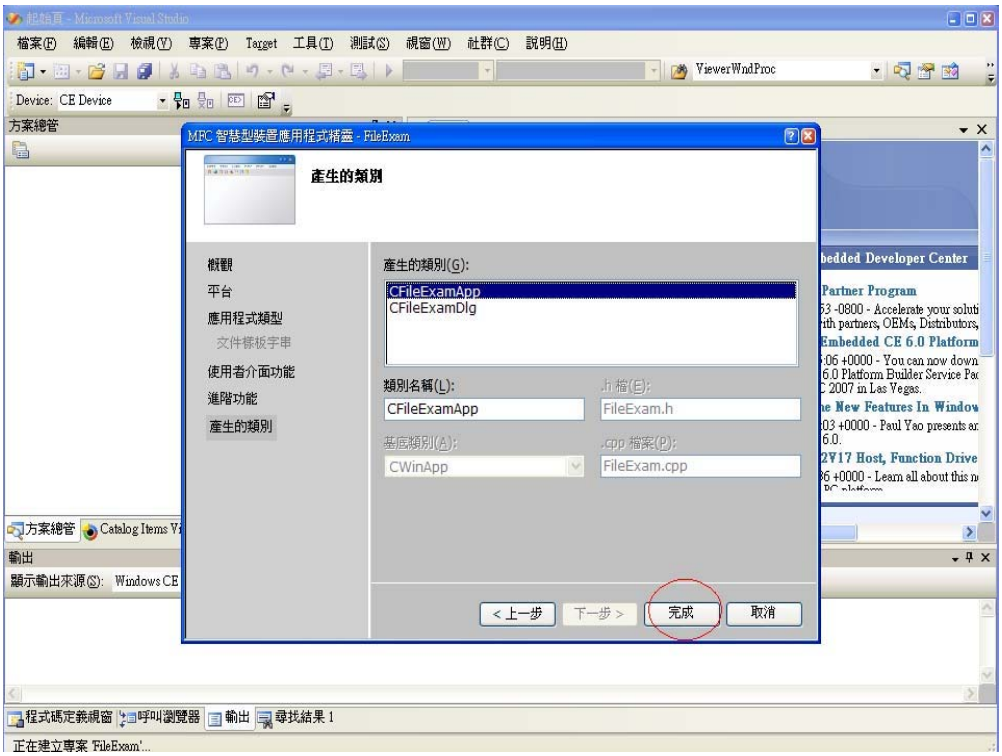
F. 按下[下一步］按鈕。



G. 按下[下一步]按鈕。

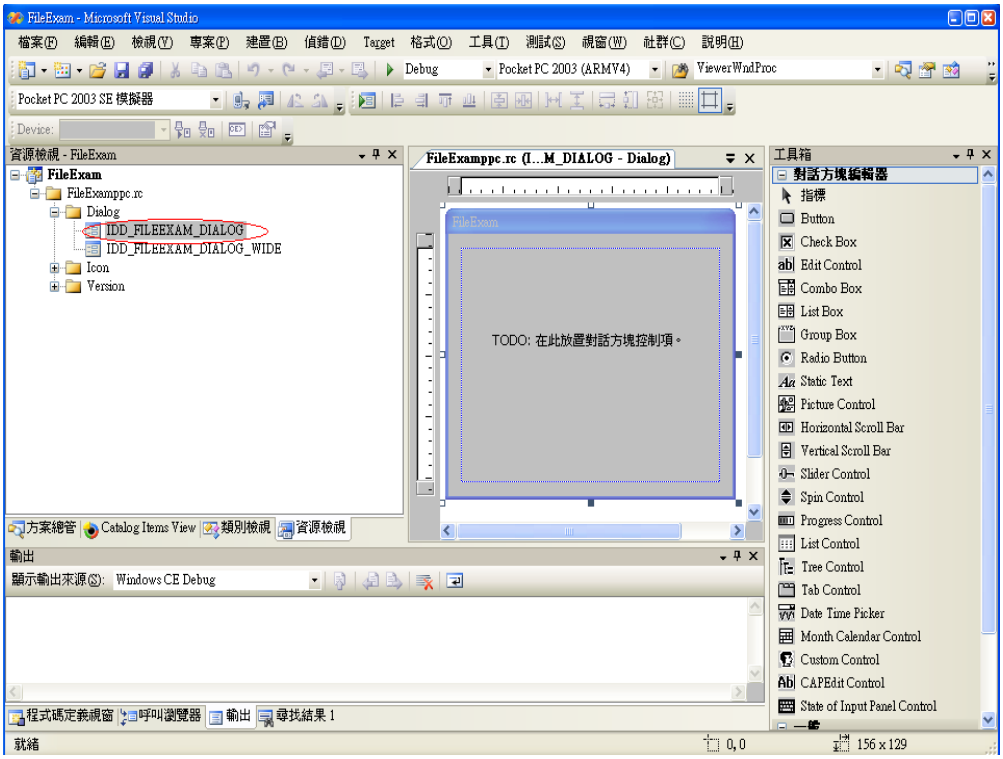


H. 按下[完成]按鈕。



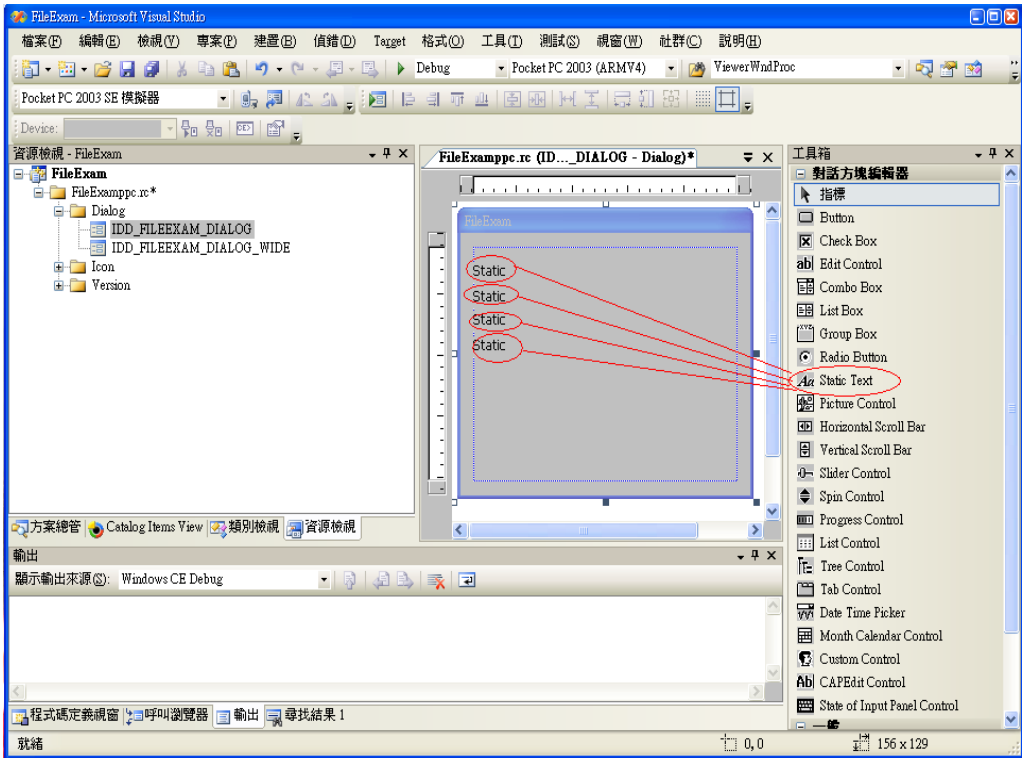
2. 設計使用者界面

A. 切換左方視窗到資源檢視，準備開始設計使用者對話方塊介面（**UI**）

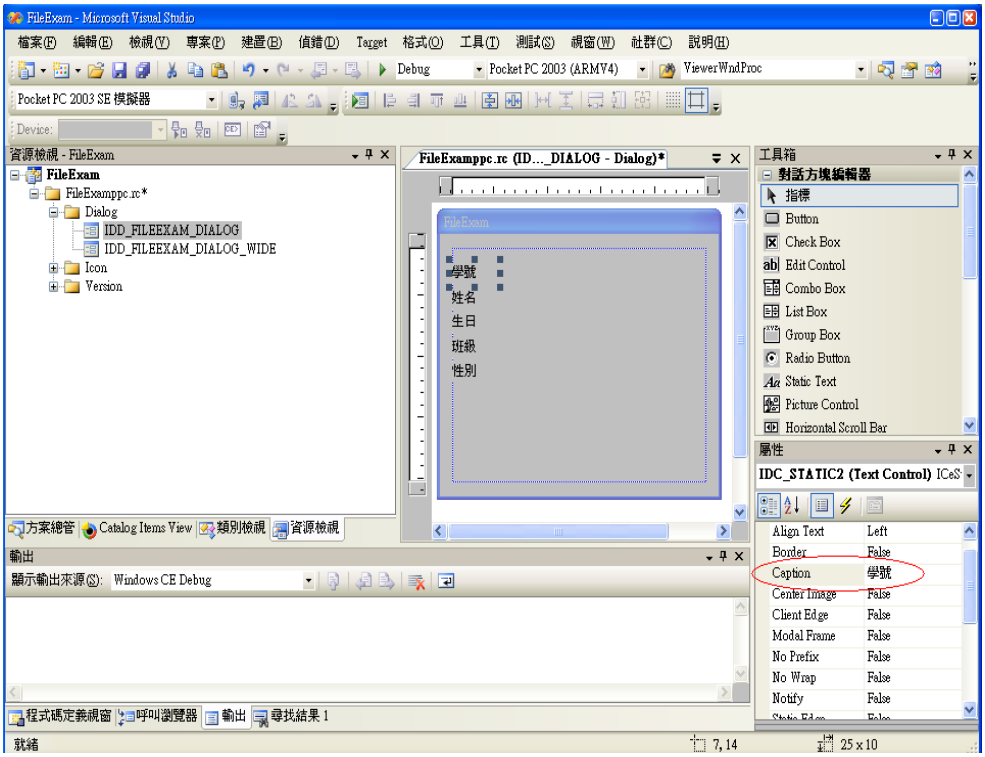


B. 將對話方塊中存在的文字對話方塊 （**TODO**：在此放置對話方塊控制

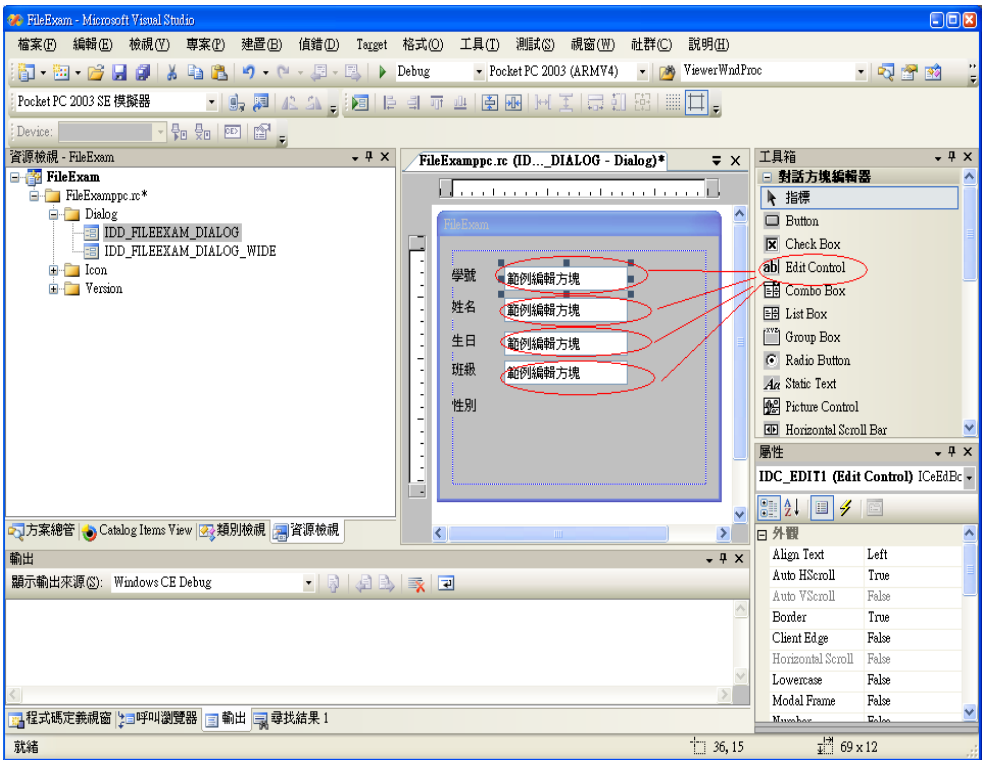
項）刪除，然後將新選取［**Static Text**］對話方塊拖曳到適當位置，總共產生五個。



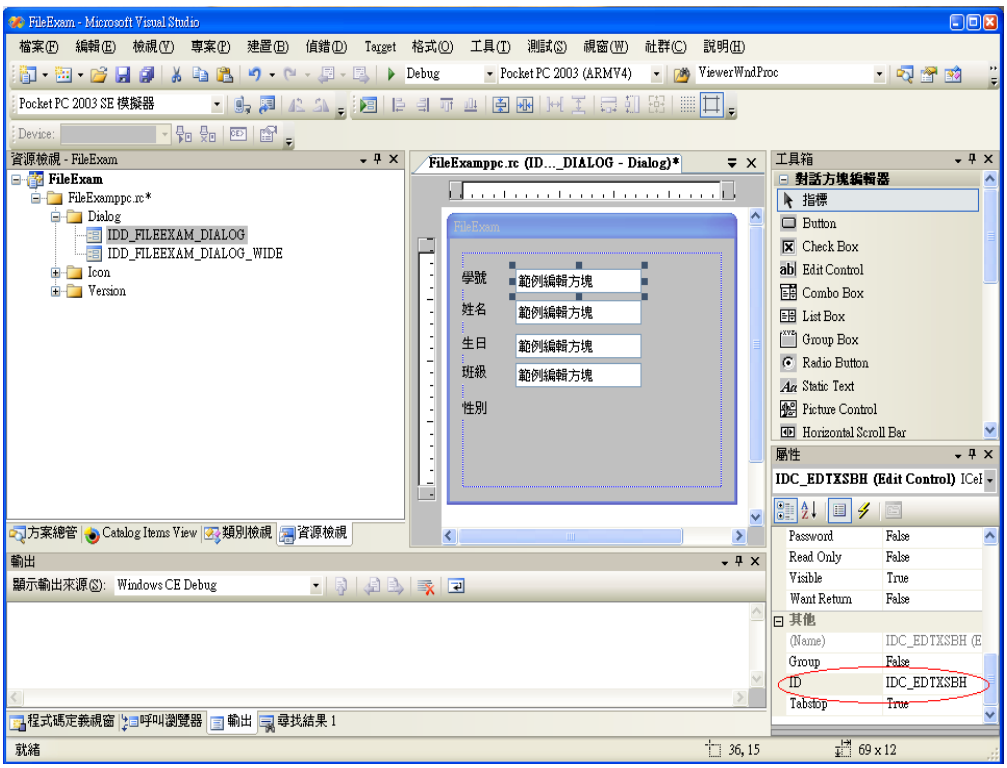
C. 分別在五個 「**Static Text**」 上按下滑鼠右鍵，選擇〔屬性〕修改屬性對話窗中 **Caption** 項分別為「學號」姓名」生日」班級」和「性 別」



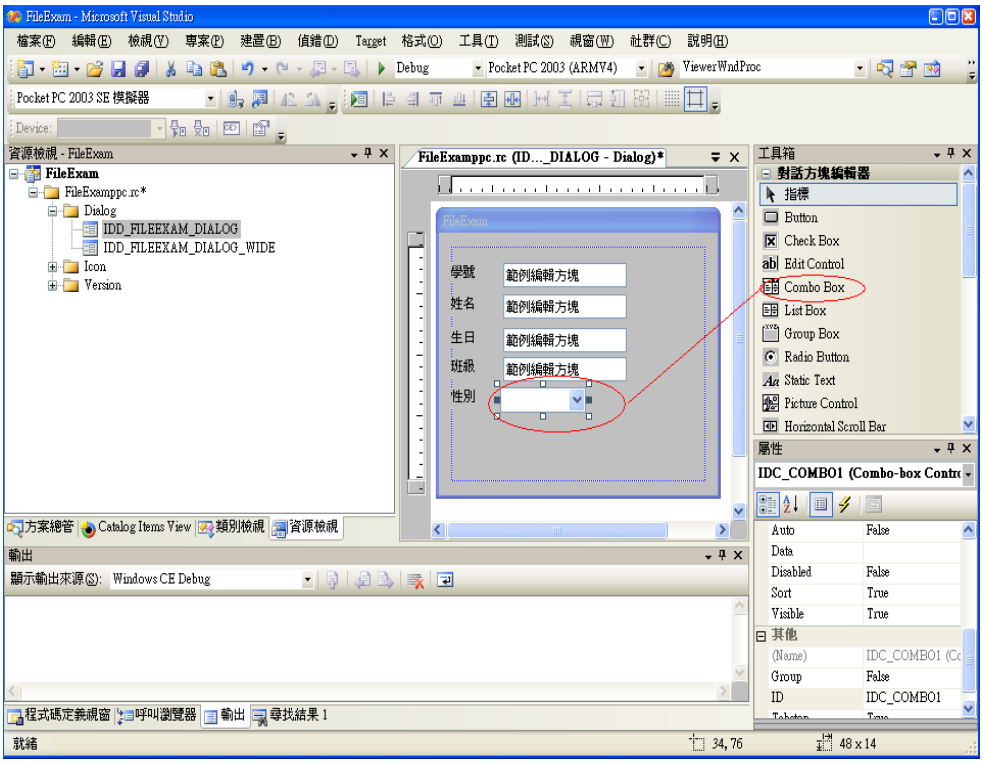
D. 新選取［**Edit Control**］對話方塊拖曳到適當位置，總共產生四個。



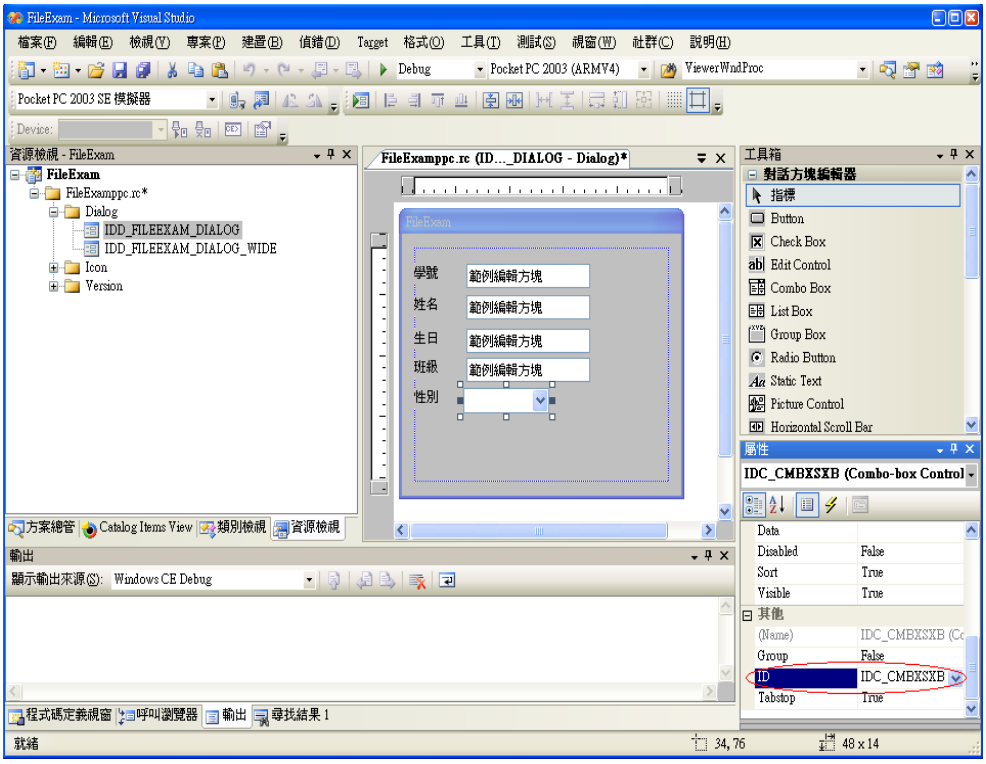
E. 分別在四個 「**Edit Control**」 上按下滑鼠右鍵，選擇〔屬性〕修改屬 性對話窗中 **ID** 項分別為「**IDC_EDTXSBH**」**IDC_EDTXSXM**」「**IDC_EDTCSRQ**」和「**IDC_EDTBJMC**」。



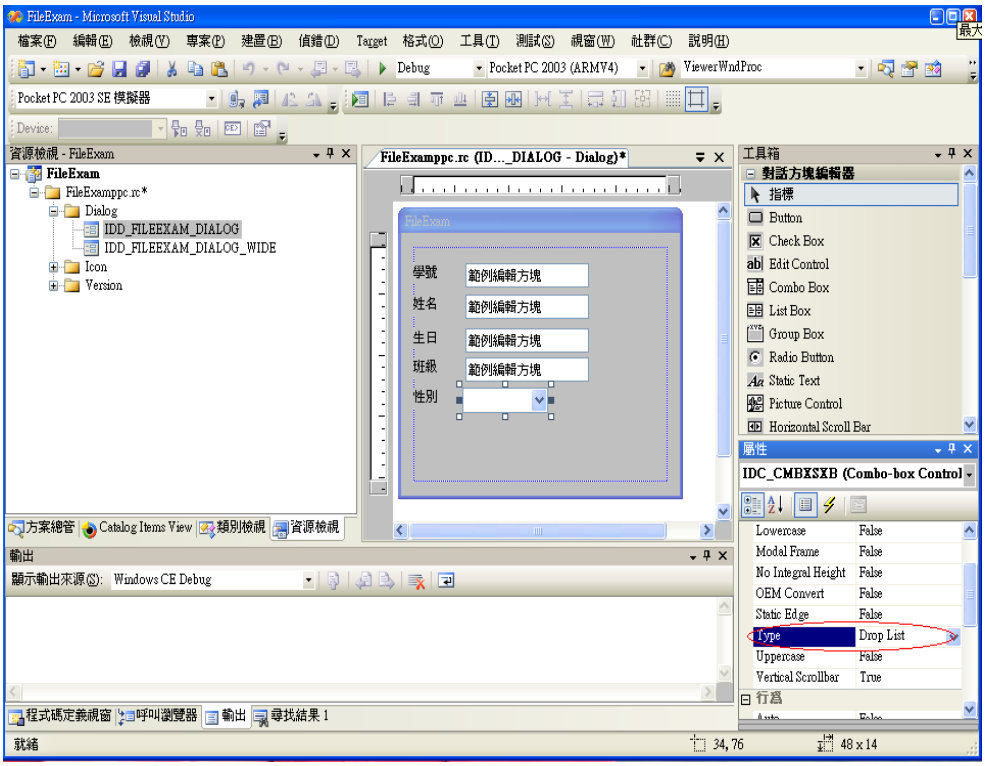
F. 新選取［**Combo Box**］對話方塊拖曳到適當位置。



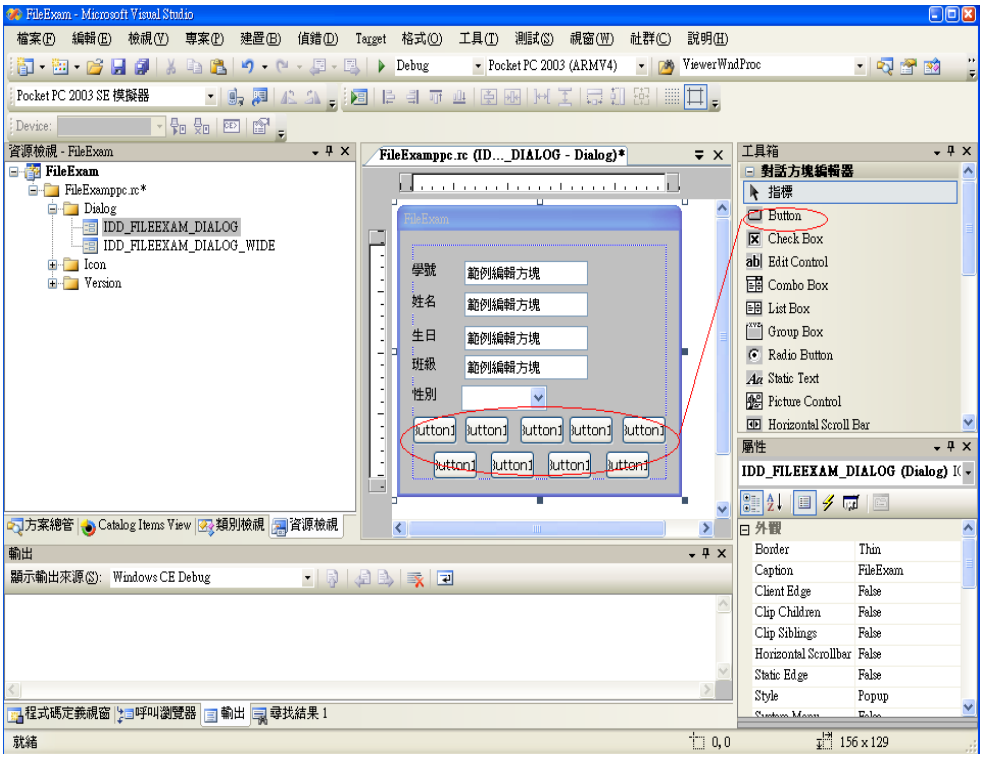
G. 在「**Combo Box**」上按下滑鼠右鍵，選擇〔屬性〕修改屬性對話窗中 **ID** 項為「**IDC_CMBXSXB**」。



H. 再修改屬性對話窗中 **Type** 項為**Drop List**。

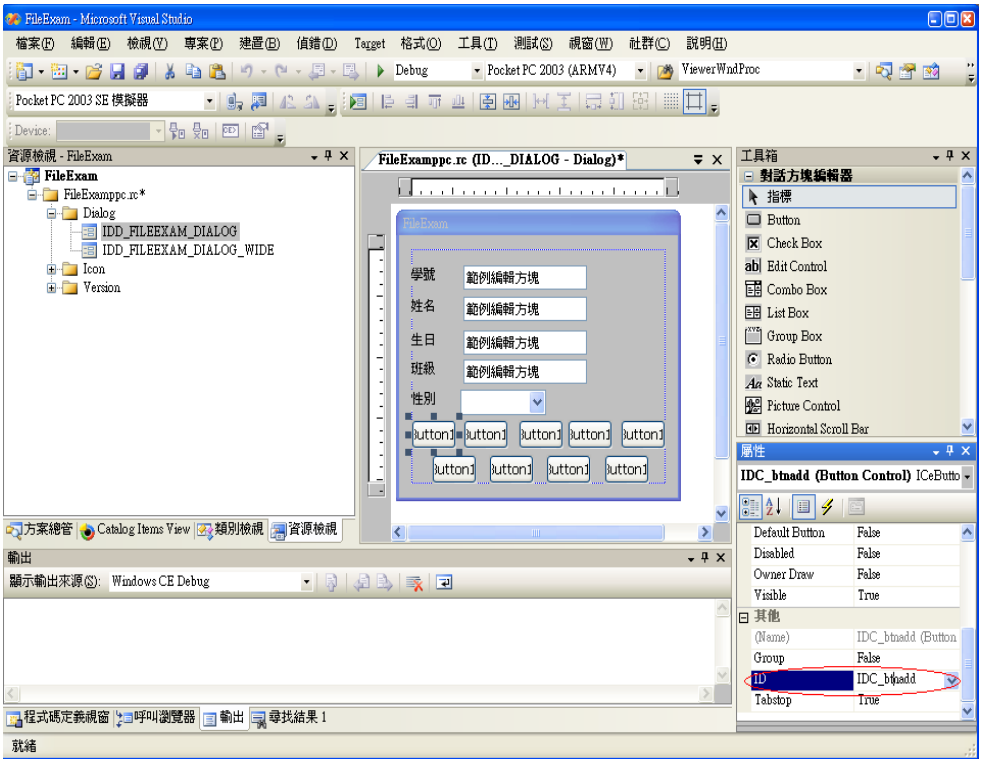


I. 新選取［**Button**］對話方塊拖曳到適當位置，總共產生九個。

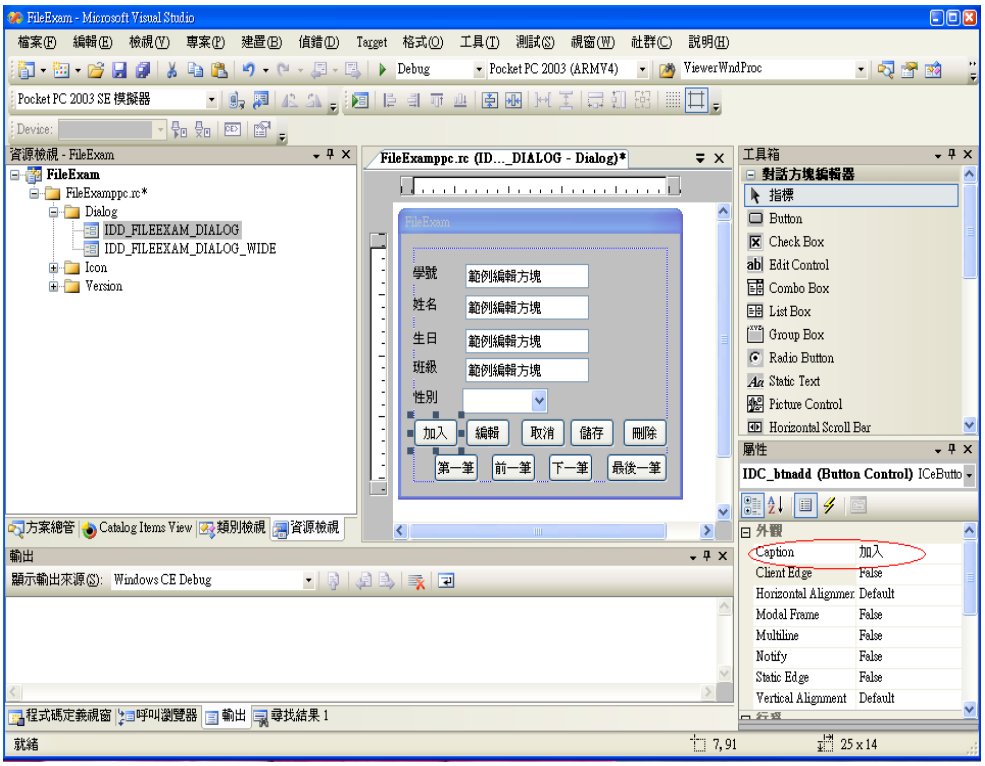


J. 分別在九個 「**Button**」 上按下滑鼠右鍵，選擇〔屬性〕修改屬性對 話窗中 **ID** 項分別為「**IDC_BTNADD**」**IDC_BTNEDIT**」

「**IDC_BTNCANCEL**」、「**IDC_BTNSAVE**」、「**IDC_BTNDELETE**」、「**IDC_BTNFIRST**」、「**IDC_BTNPRIOR**」、「**IDC_BTNNEXT**」和「**IDC_BTNLAST**」。

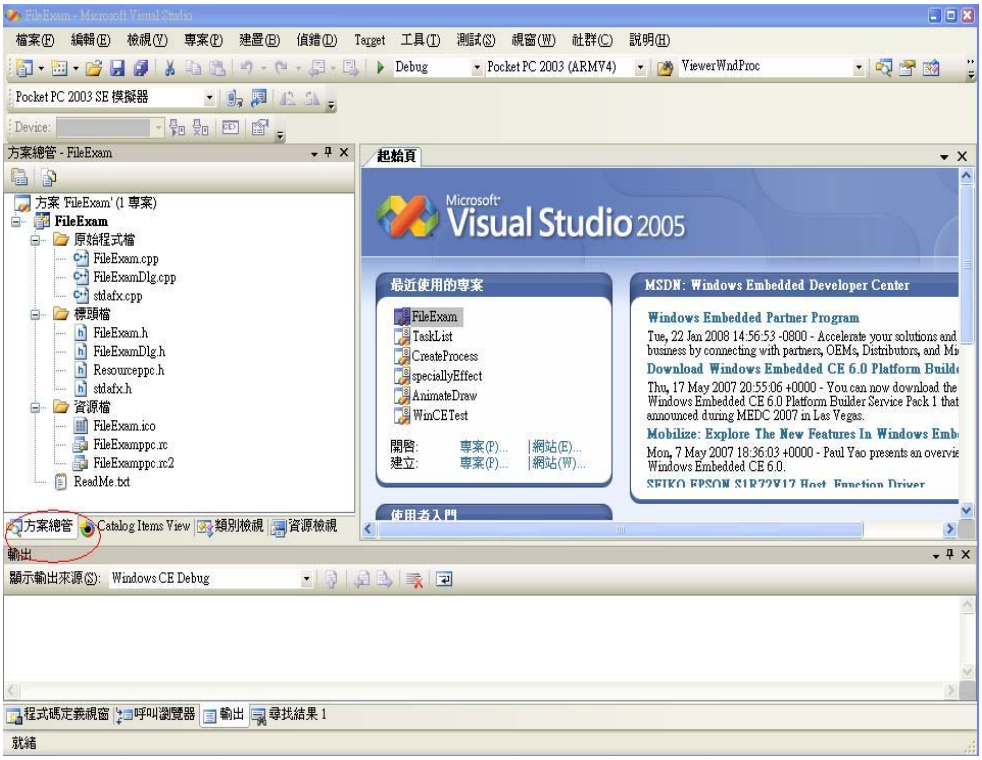


K. 再修改屬性對話窗中 **Caption** 項分別為「加入」、「編輯」、「取消」、「儲 存」、「刪除」、「第一筆」、「前一筆」、「下一筆」和「最後一筆」。

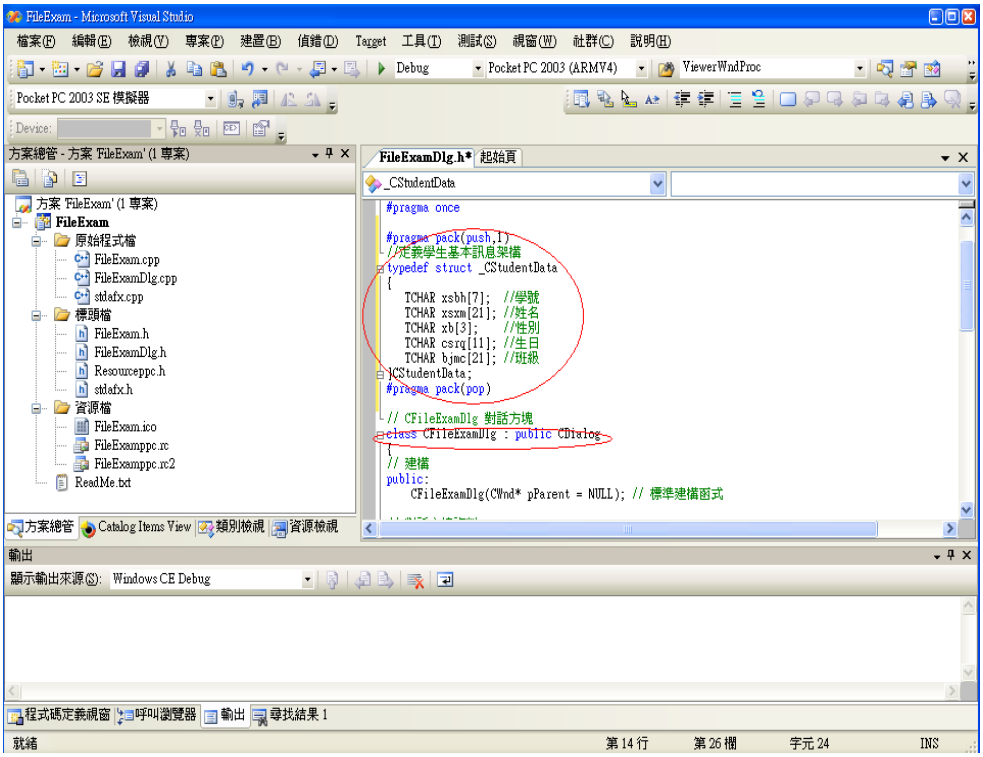


3. 程式設計

A. 切換左方視窗到方案總管，準備開始設計使用者程式。



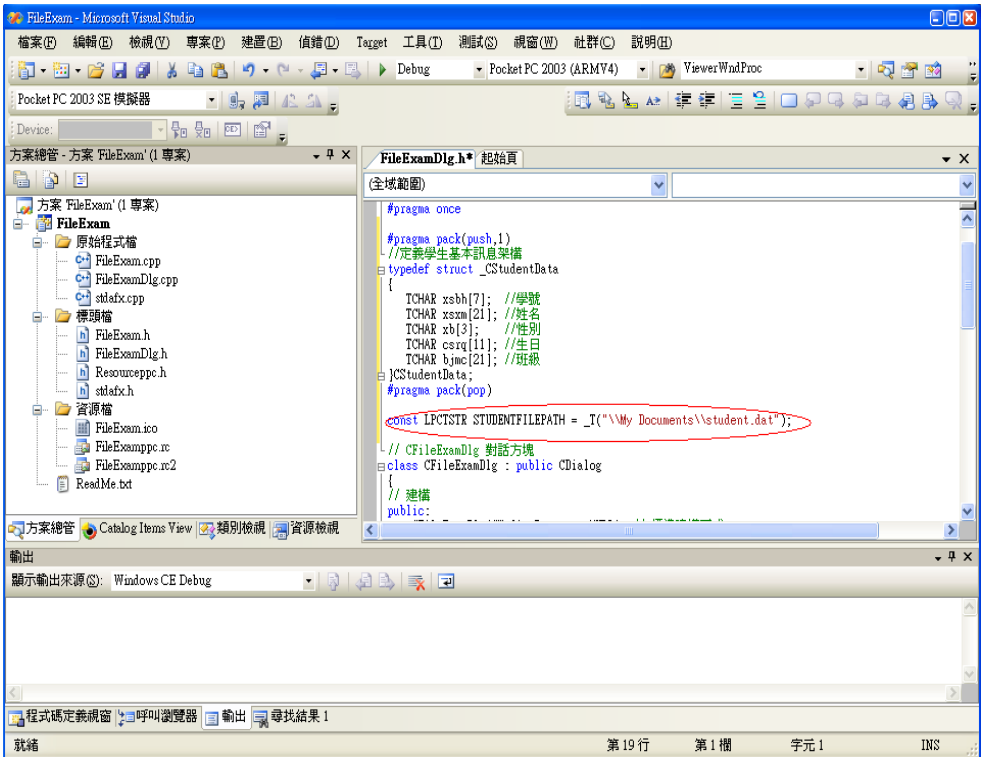
B. 在方案總管視窗中的原始程式檔內，開啟 **FileExamDlg.h** ，在原始程 式碼**class CFileExamDlg : public CDialog** 前面加入一段定義學生基本 訊息的結構。



加入的程式碼 ：

```
#pragma pack(push,1)
//定義學生基本訊息架構
typedef struct _CStudentData
{
    TCHAR xsbh[7];      //  學號
    TCHAR xsxm[21];     //  姓名
    TCHAR xb[3];        //  性別
    TCHAR csrq[11];     //  生日
    TCHAR bjmc[21];     //  班級
}CStudentData;
#pragma pack(pop)
```

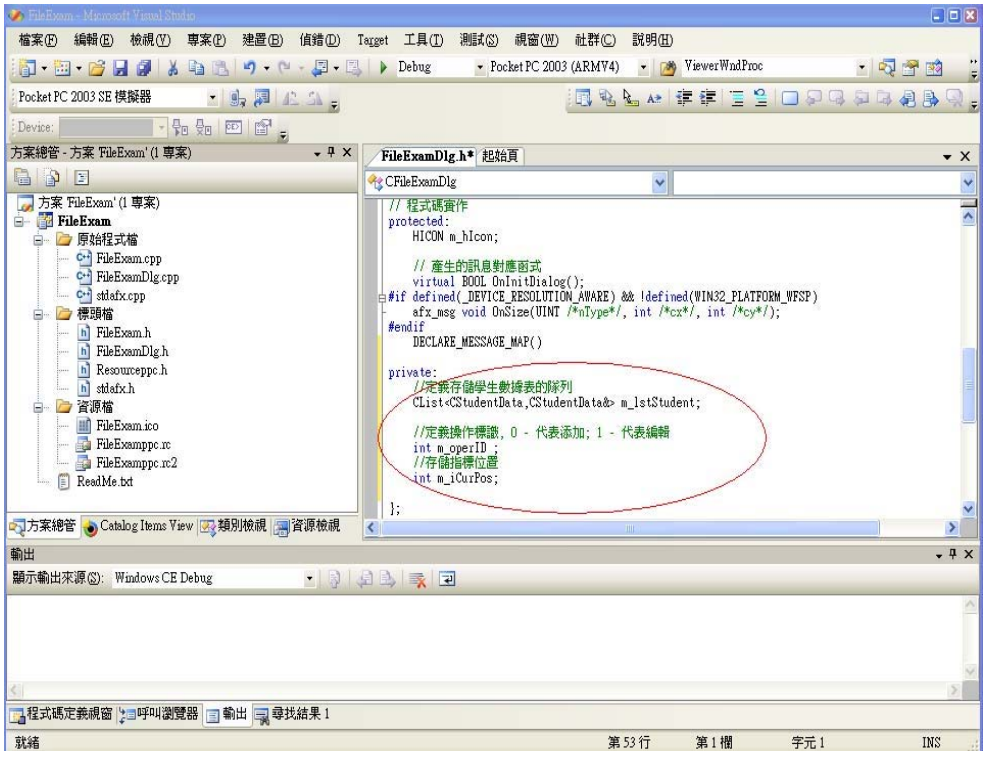
C. 在開啟的**FileExamDlg.h** 原始程式碼中，加入定義一個全域常數的標示來儲存學生訊息的檔案名。



加入的程式碼 ：

const LPCTSTR STUDENTFILEPATH = _T("\\My Documents\\student.dat");

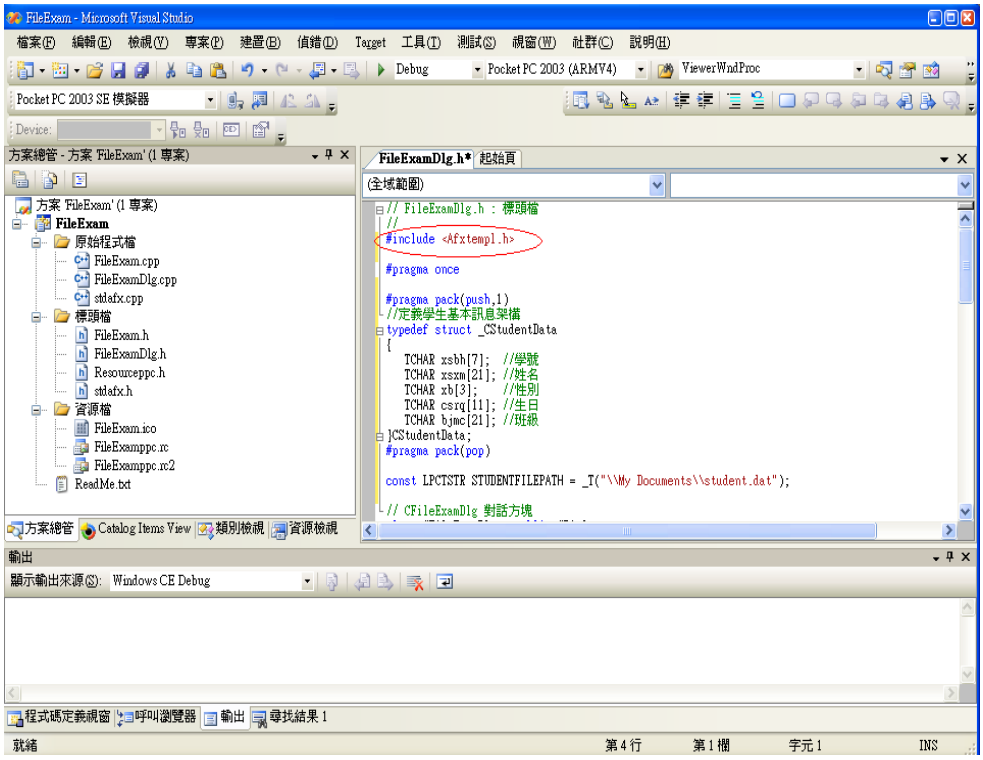
D. 在開啟的 **FileExamDlg.h** 原始程式碼**Class CFileExamDlg** 中，加入定義 **private** 的變數，用以定義一個列表儲存學生基本訊息。



加入的程式碼 ：

```
private:
//  定義存儲學生數據表的列
CList<CStudentData,CStudentData&> m_lstStudent;

//  定義操作標識, 0 -  代表添加; 1 -  代表編輯
int m_operID ;

//  存儲指標位置
int m_iCurPos;
```

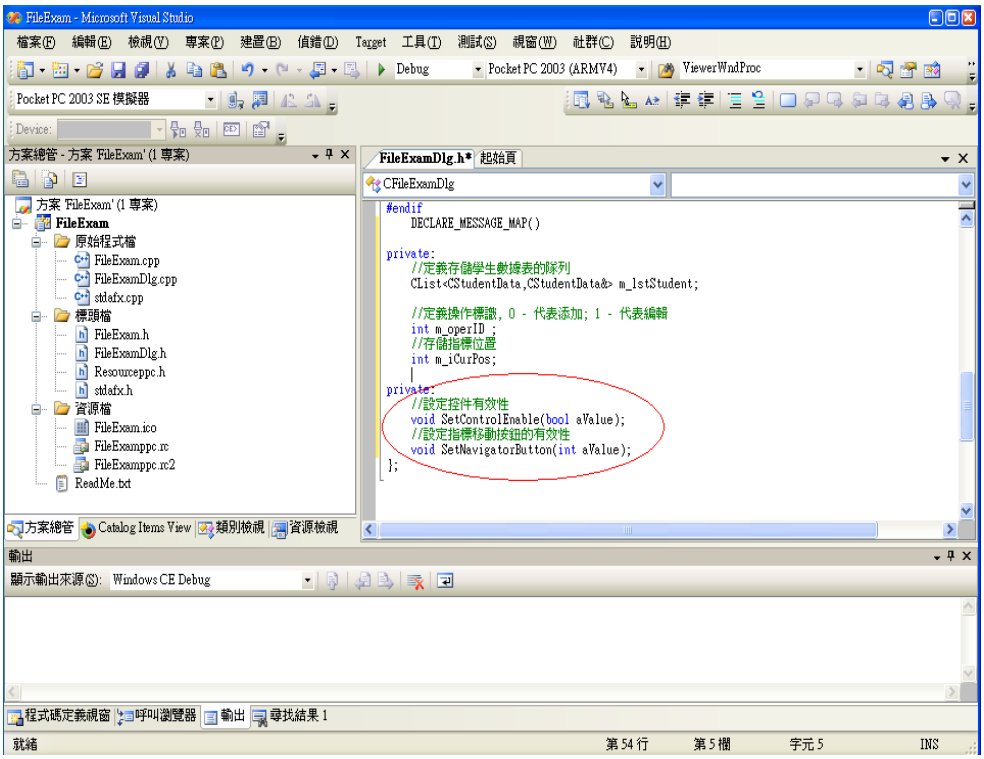
E. 在開啟的 **FileExamDlg.h** 原始程式碼中,加入宣告列表變數所引用標準模板庫的 **Afxtempl.h**。



加入的程式碼 ：

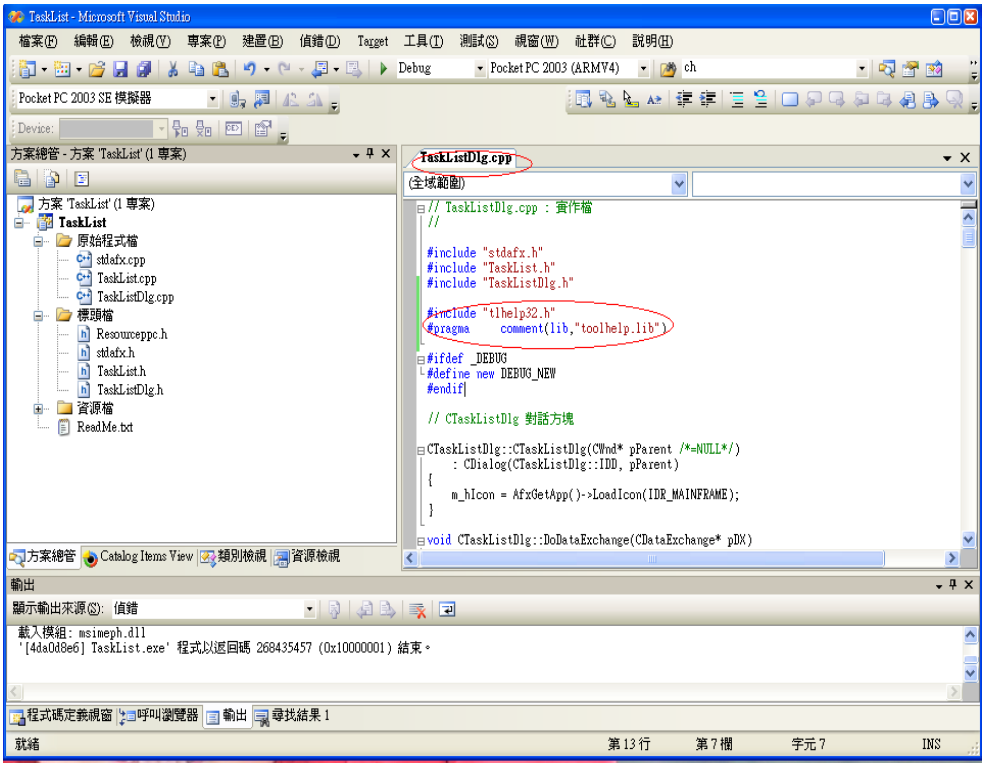
```
#include <Afxtempl.h>
```

F. 在開啟的 **FileExamDlg.h** 原始程式碼**Class CFileExamDlg** 中，加入分別設定控件有效性和設定指標移動按鈕的有效性 **private** 方法定義。

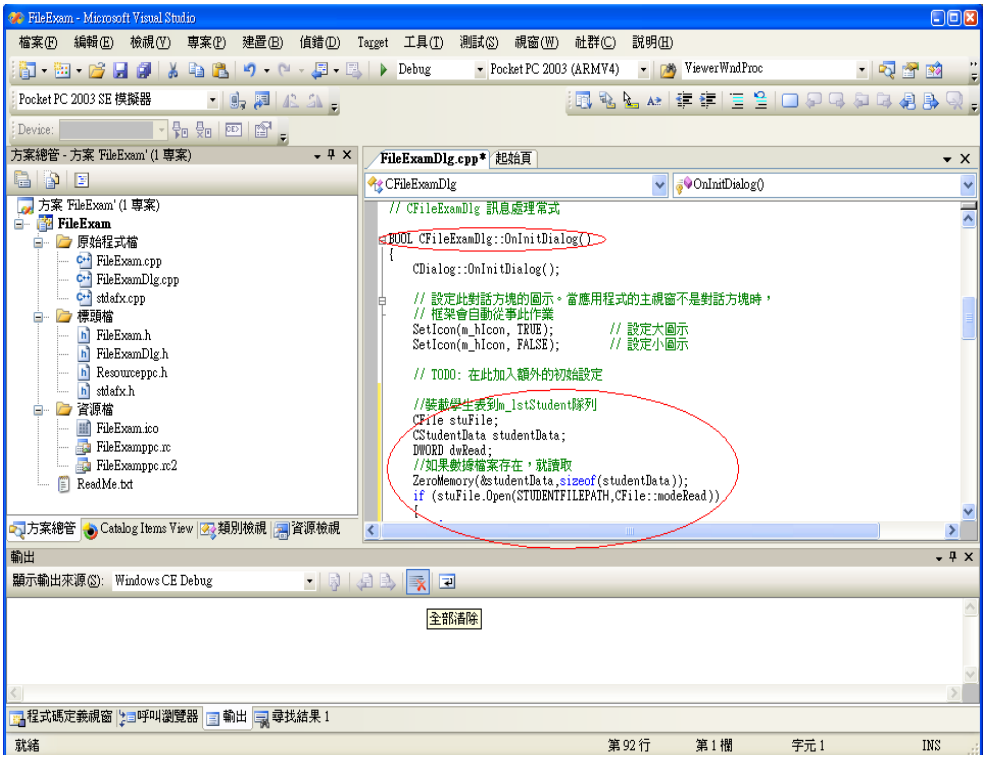


加入的程式碼 ：

```
private:
    //  設定控件有效性
    void SetControlEnable(bool aValue);

    //  設定指標移動按鈕的有效性
    void SetNavigatorButton(int aValue);
```



G. 在方案總管視窗中的原始程式檔內，開啟 **FileExamDlg.cpp** ，在原始程式碼**BOOL CFileExamDlg::OnInitDialog()** 內的**return TRUE** 前加入一段處理初始化學生數據檔案的程式碼。

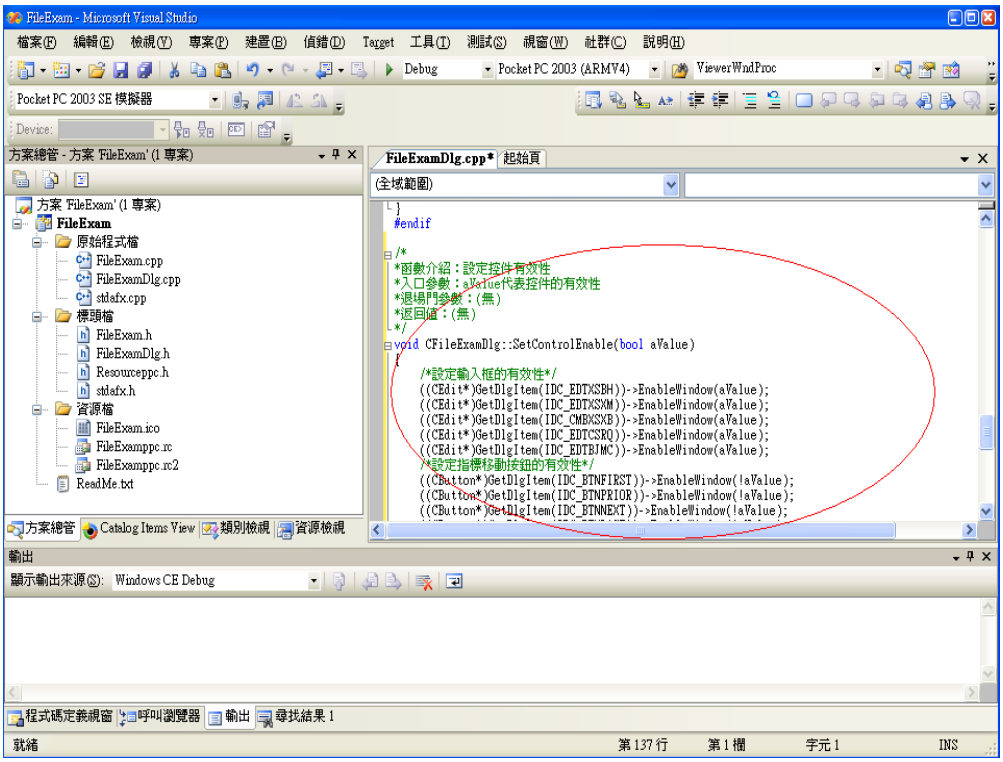


加入的程式碼 ：

```
//  裝載學生表到m_lstStudent
CFile stuFile;
CStudentData studentData;
DWORD dwRead;

//  如果數據檔案存在，就讀取

// ZeroMemory  將一塊內存記憶體置於零
ZeroMemory(&studentData,sizeof(studentData));

if (stuFile.Open(STUDENTFILEPATH,CFile::modeRead))
{
    do
    {
        dwRead = stuFile.Read(&studentData,sizeof(studentData));
        if (dwRead != 0 )
        {
            m_lstStudent.AddTail(studentData);
        }
    }while(dwRead > 0);

    //  關閉數據檔案
    stuFile.Close();
}
else  //  創建檔案
{
     if
        (!stuFile.Open(STUDENTFILEPATH,CFile::modeCreat
        e|CFile::modeWrite))
    {
        AfxMessageBox(_T("創建數據庫失敗"));
        return FALSE;
    }
    stuFile.Close();
}

//  初始化界面顯示
if (m_lstStudent.GetCount() > 0)
```

```
{
    studentData = m_lstStudent.GetHead();
    m_iCurPos = 0;
    //  初始劃第一條顯示
    m_xsbh = studentData.xsbh;
    m_xsxm = studentData.xsxm;
    m_xsxb = studentData.xb;
    m_csrq = studentData.csrq;
    m_bjmc = studentData.bjmc;

    UpdateData(false);
}
SetControlEnable(FALSE);

//  設定指標移動按鈕有效性
SetNavigatorButton(m_iCurPos);
```

H. 在開啟的 **FileExamDlg.cpp** 原始程式碼最後，加入「設定控件有效性」的函數 **SetControlEnable** 程式碼。



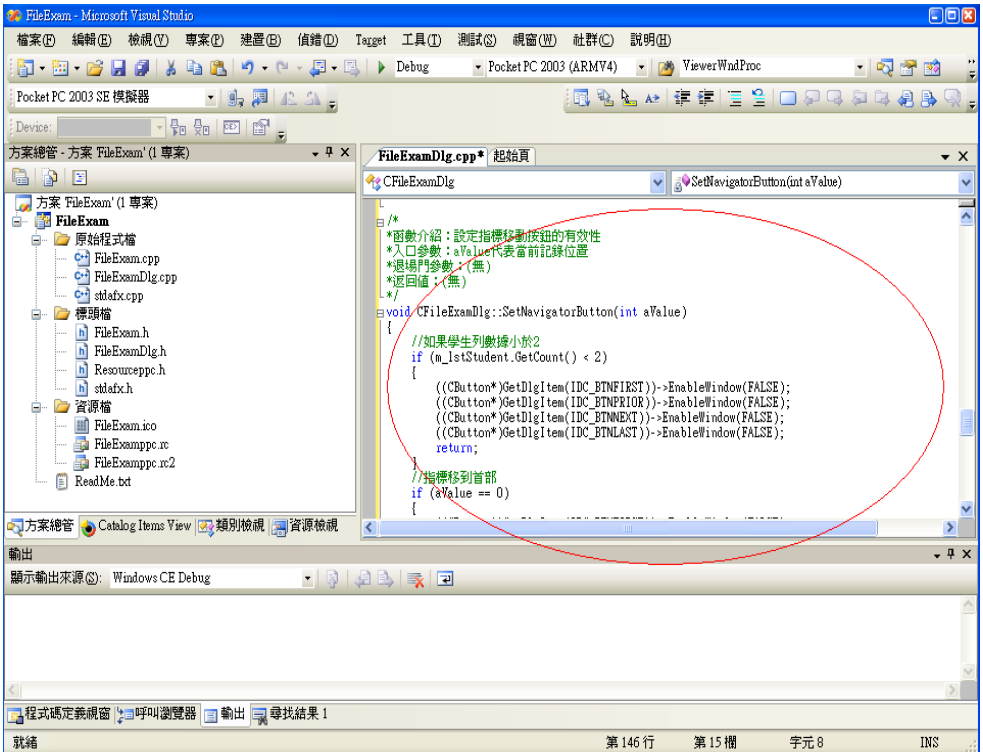
加入的程式碼 ：

```
//  設定控件有效性
```

```
void CFileExamDlg::SetControlEnable(bool aValue)
{
    /*設定輸入框的有效性*/
((CEdit*)GetDlgItem(IDC_EDTXSBH))->EnableWindow(aValue);
((CEdit*)GetDlgItem(IDC_EDTXSXM))->EnableWindow(aValue);
((CEdit*)GetDlgItem(IDC_CMBXSXB))->EnableWindow(aValue);
((CEdit*)GetDlgItem(IDC_EDTCSRQ))->EnableWindow(aValue);
((CEdit*)GetDlgItem(IDC_EDTBJMC))->EnableWindow(aValue);
    /*設定指標移動按鈕的有效性*/
((CButton*)GetDlgItem(IDC_BTNFIRST))->EnableWindow(!aValue);
((CButton*)GetDlgItem(IDC_BTNPRIOR))->EnableWindow(!aValue);
((CButton*)GetDlgItem(IDC_BTNNEXT))->EnableWindow(!aValue);
((CButton*)GetDlgItem(IDC_BTNLAST))->EnableWindow(!aValue);
    /*設定操作按鈕的有效性*/
((CButton*)GetDlgItem(IDC_BTNADD))->EnableWindow(!aValue);
((CButton*)GetDlgItem(IDC_BTNEDIT))->EnableWindow(!aValue);
((CButton*)GetDlgItem(IDC_BTNCANCEL))->EnableWindow(aValue);
((CButton*)GetDlgItem(IDC_BTNSAVE))->EnableWindow(aValue);
((CButton*)GetDlgItem(IDC_BTNDELETE))->EnableWindow(!aValue);
}
```

Ⅰ. 在開啟的 **FileExamDlg.cpp** 原始程式碼最後，加入「設定指標移動按鈕

的有效性」的函數**SetNavigatorButton**　程式碼。



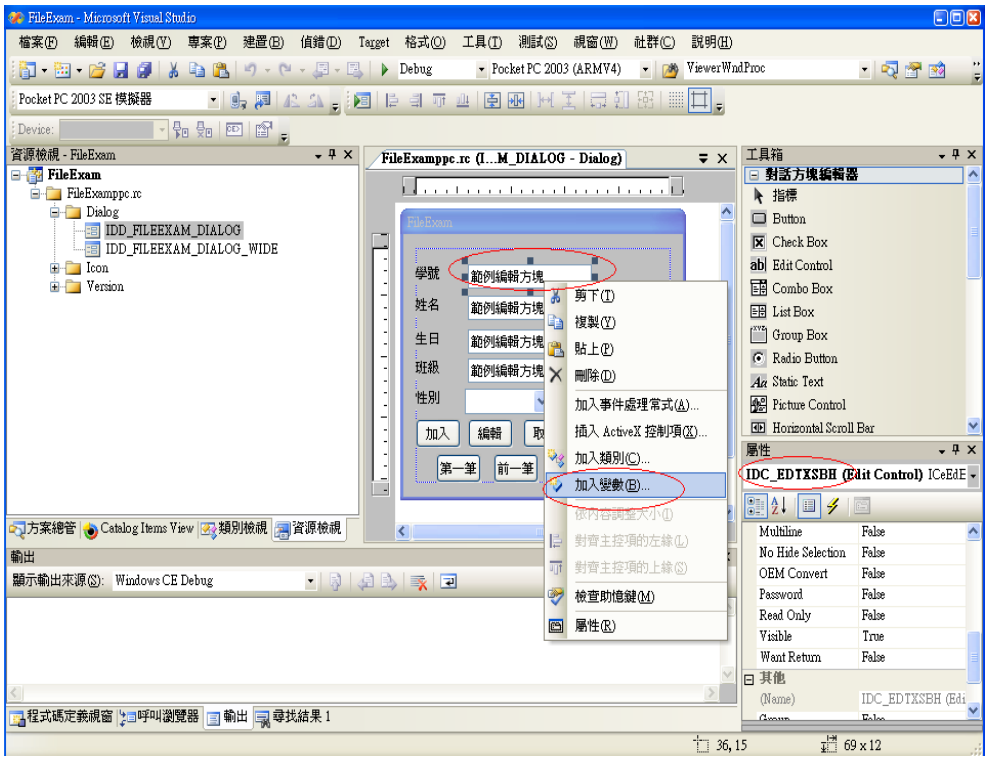
加入的程式碼 ：

```
//  設定指標移動按鈕的有效性
void CFileExamDlg::SetNavigatorButton(int aValue)
{
    //  如果學生列數據小於2
    if (m_lstStudent.GetCount() < 2)
    {

        ((CButton*)GetDlgItem(IDC_BTNFIRST))->EnableWindow
        (FALSE);

        ((CButton*)GetDlgItem(IDC_BTNPRIOR))->EnableWindow
        (FALSE);

        ((CButton*)GetDlgItem(IDC_BTNNEXT))-
        >EnableWindow( FALSE);

        ((CButton*)GetDlgItem(IDC_BTNLAST))-
        >EnableWindow( FALSE);
        return;
```

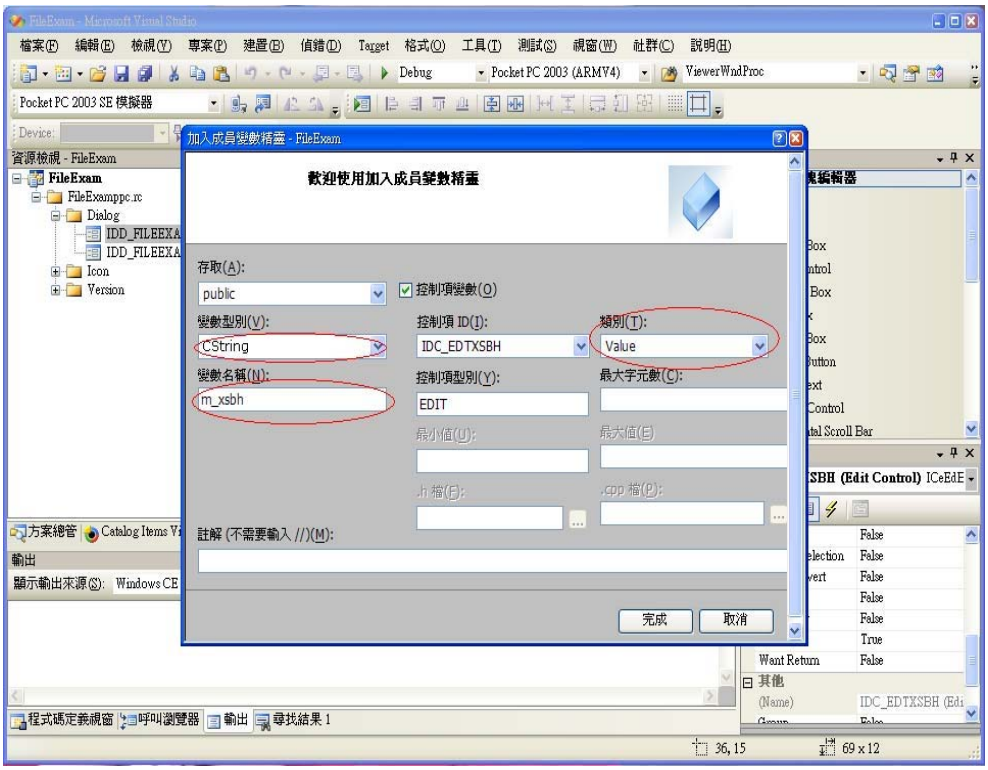
```
}
//  指標移到首部
if (aValue == 0)
{

    ((CButton*)GetDlgItem(IDC_BTNFIRST))->EnableWindow
    (FALSE);

    ((CButton*)GetDlgItem(IDC_BTNPRIOR))->EnableWindow
    (FALSE);
    if (m_lstStudent.GetCount() >=2)
    {

        ((CButton*)GetDlgItem(IDC_BTNNEXT))->EnableWin
        dow(TRUE);

        ((CButton*)GetDlgItem(IDC_BTNLAST))->EnableWin
        dow(TRUE);
    }
}
//  如果指標移到末尾
else if (aValue == m_lstStudent.GetCount() - 1)
{

    ((CButton*)GetDlgItem(IDC_BTNNEXT))-
    >EnableWindow( FALSE);

    ((CButton*)GetDlgItem(IDC_BTNLAST))-
    >EnableWindow( FALSE);
    if (m_lstStudent.GetCount() >=2)
    {

        ((CButton*)GetDlgItem(IDC_BTNFIRST))->EnableWi
        ndow(TRUE);

        ((CButton*)GetDlgItem(IDC_BTNPRIOR))->EnableWi
        ndow(TRUE);
    }
```

```
        }
        else
        {

            ((CButton*)GetDlgItem(IDC_BTNFIRST))->EnableWindow
            (TRUE);

            ((CButton*)GetDlgItem(IDC_BTNPRIOR))->EnableWindow
            (TRUE);

            ((CButton*)GetDlgItem(IDC_BTNNEXT))-
            >EnableWindow( TRUE);

            ((CButton*)GetDlgItem(IDC_BTNLAST))->EnableWindow(
            TRUE);
        }
    }
```

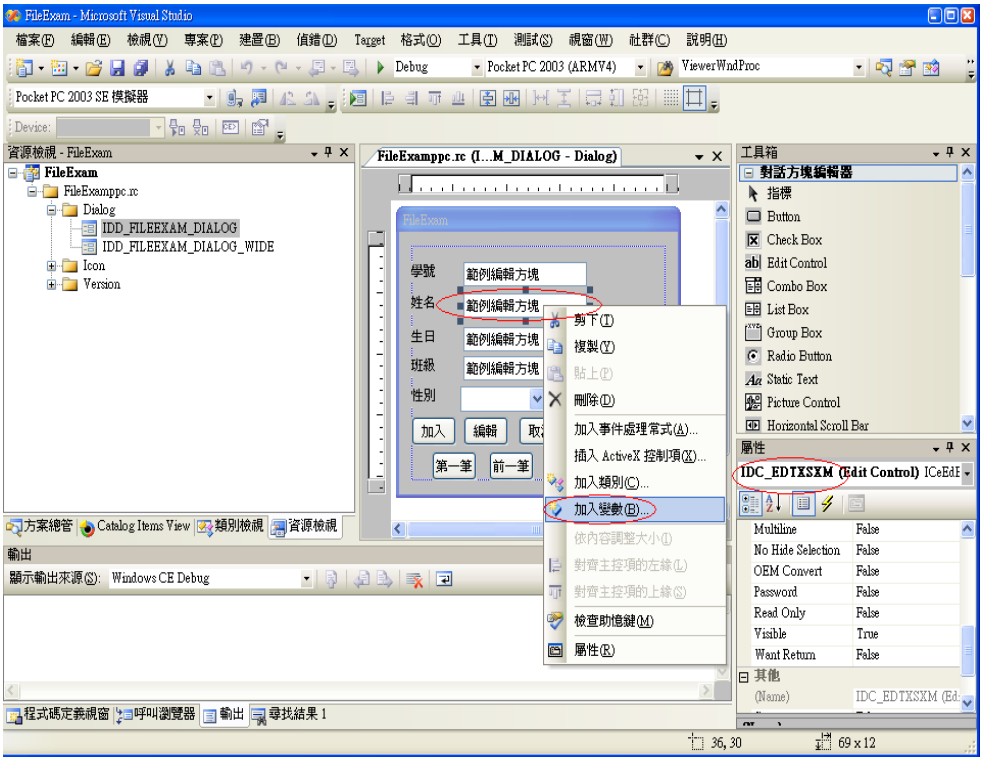
J. 切換左方視窗到資源檢視，準備開始加入變數。在要加入變數的「**Edit Control**」對話方塊上（**IDC_EDTXSBH**）按下滑鼠右鍵，選擇［加入變數］。



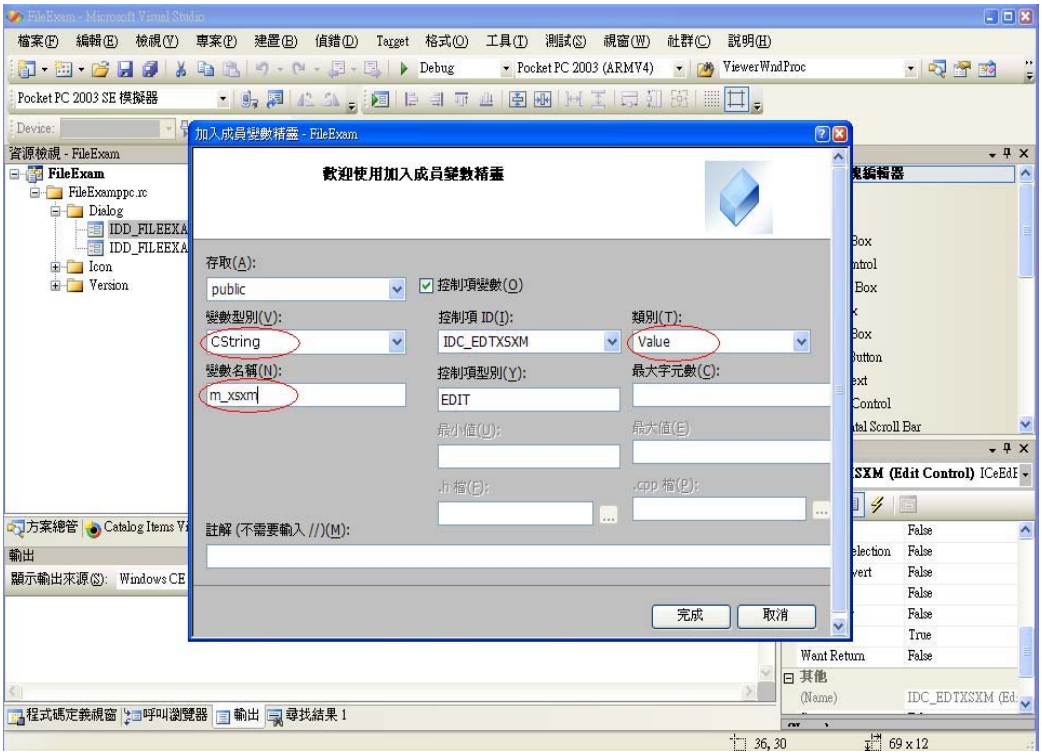
K. 在加入成員變數的對話框上，將類別選單改為「**Value**」變數型別改為「**CString**」變數名稱為「**m_xsbh**」



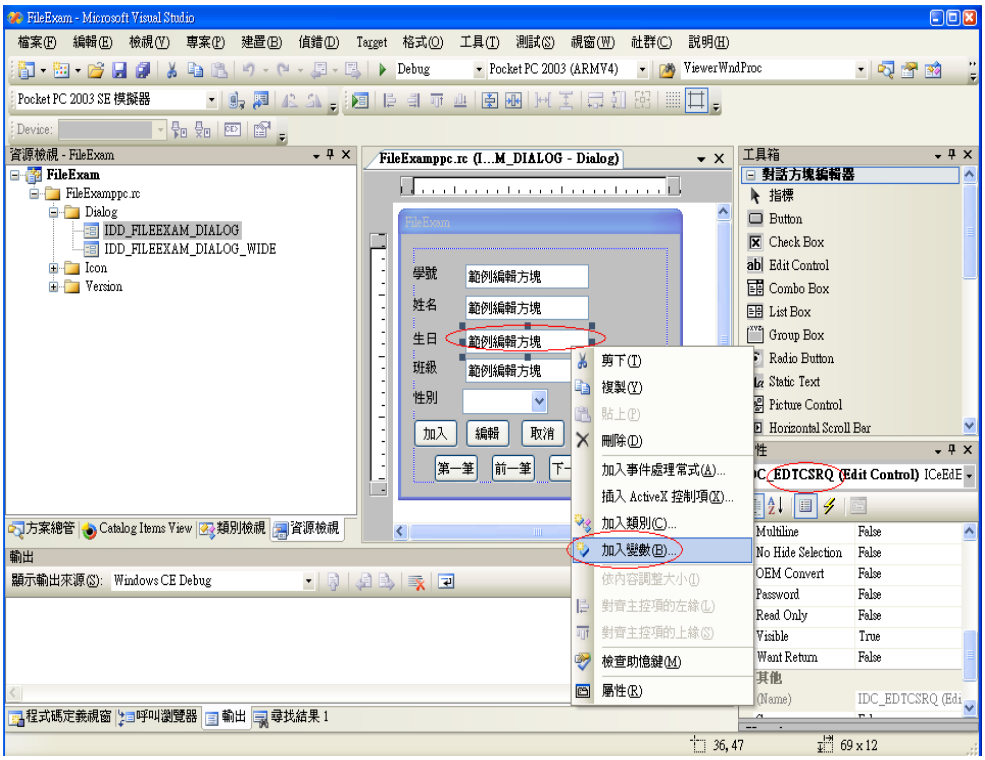
L. 切換左方視窗到資源檢視，準備開始加入變數。在要加入變數的「**Edit Control**」對話方塊上（**IDC_EDTXSXM**）按下滑鼠右鍵，選擇［加入變數］。



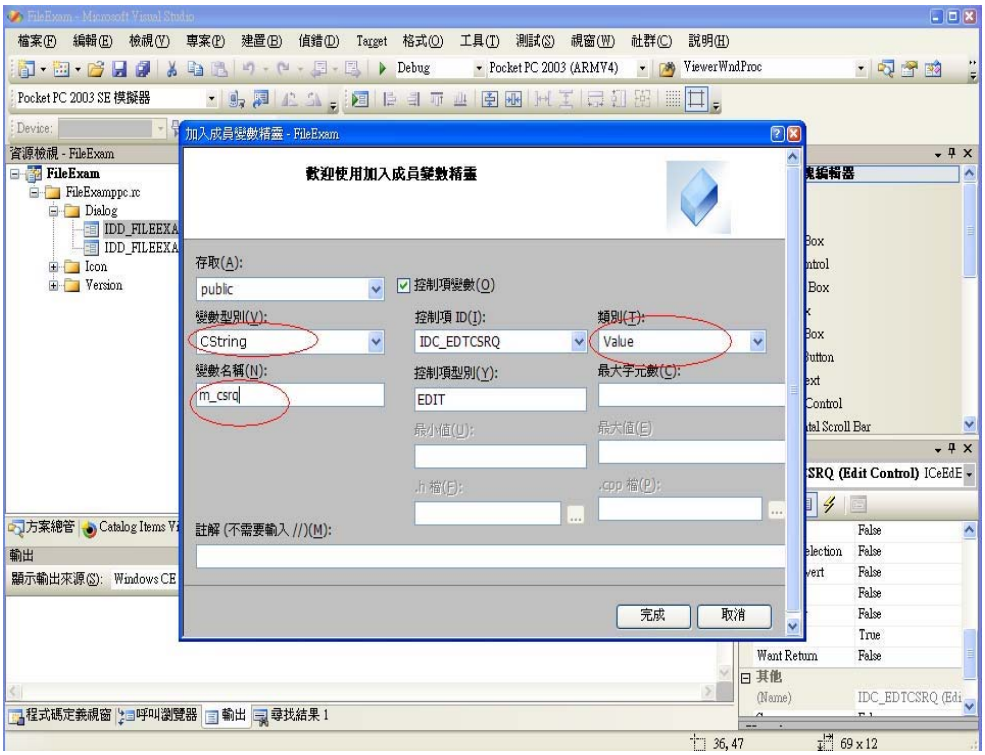
M. 在加入成員變數的對話框上，將類別選單改為「**Value**」變數型別改為「**CString**」變數名稱為「**m_xsxm**」。



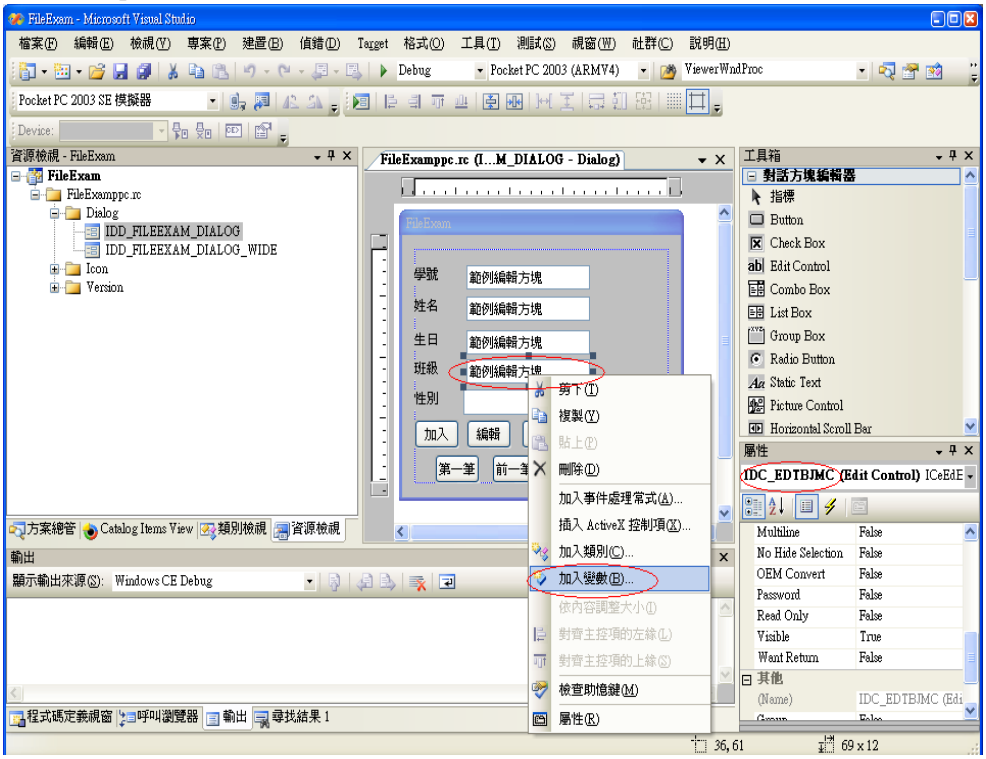
N. 切換左方視窗到資源檢視，準備開始加入變數。在要加入變數的「**Edit Control**」對話方塊上（**IDC_EDTCSRQ**）按下滑鼠右鍵，選擇［加入變數］。



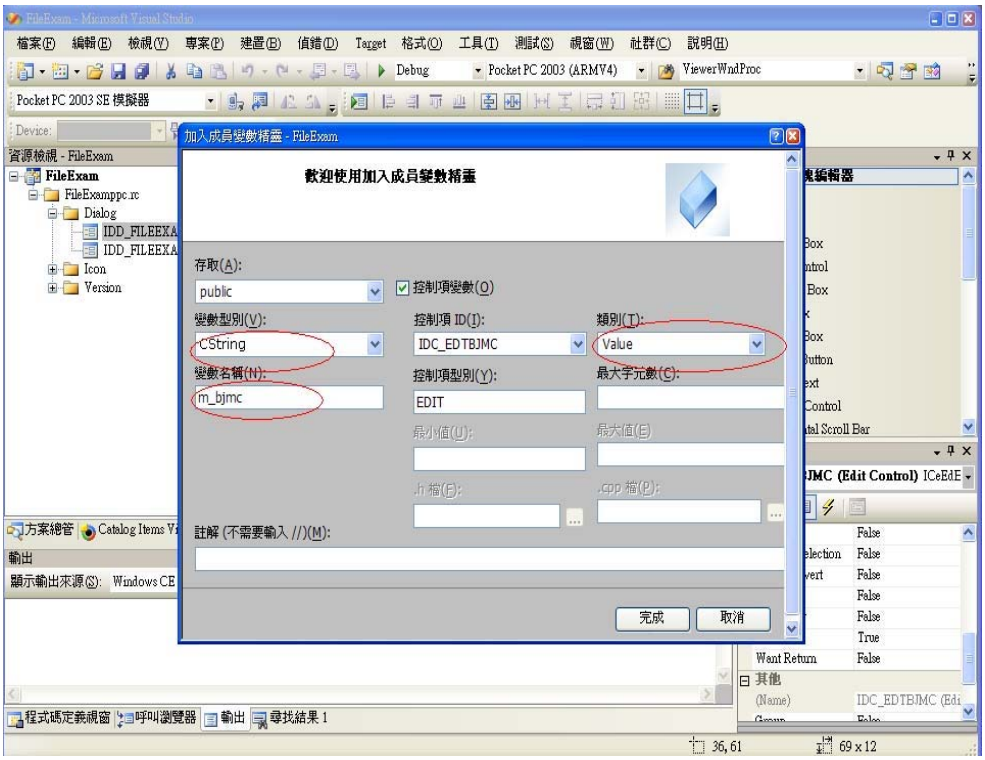
0. 在加入成員變數的對話框上，將類別選單改為「**Value**」變數型別改為「**CString**」變數名稱為「**m_csrq**」。



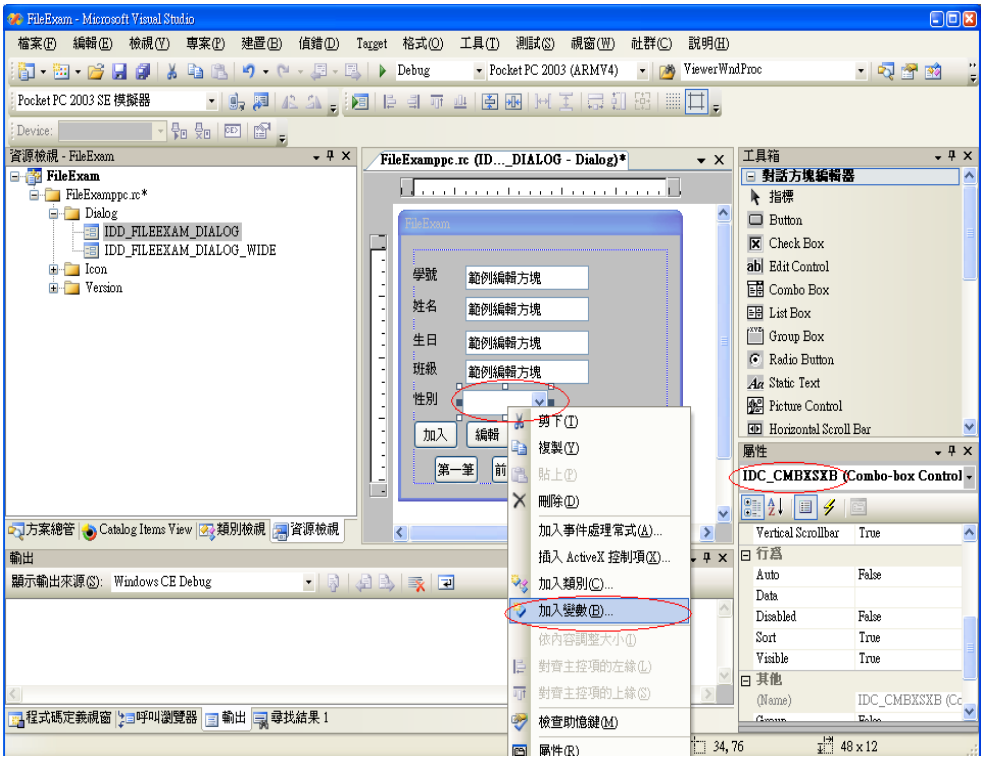
P. 切換左方視窗到資源檢視，準備開始加入變數。在要加入變數的「**Edit Control**」對話方塊上（**IDC_EDTBJMC**）按下滑鼠右鍵，選擇［加入變數］。



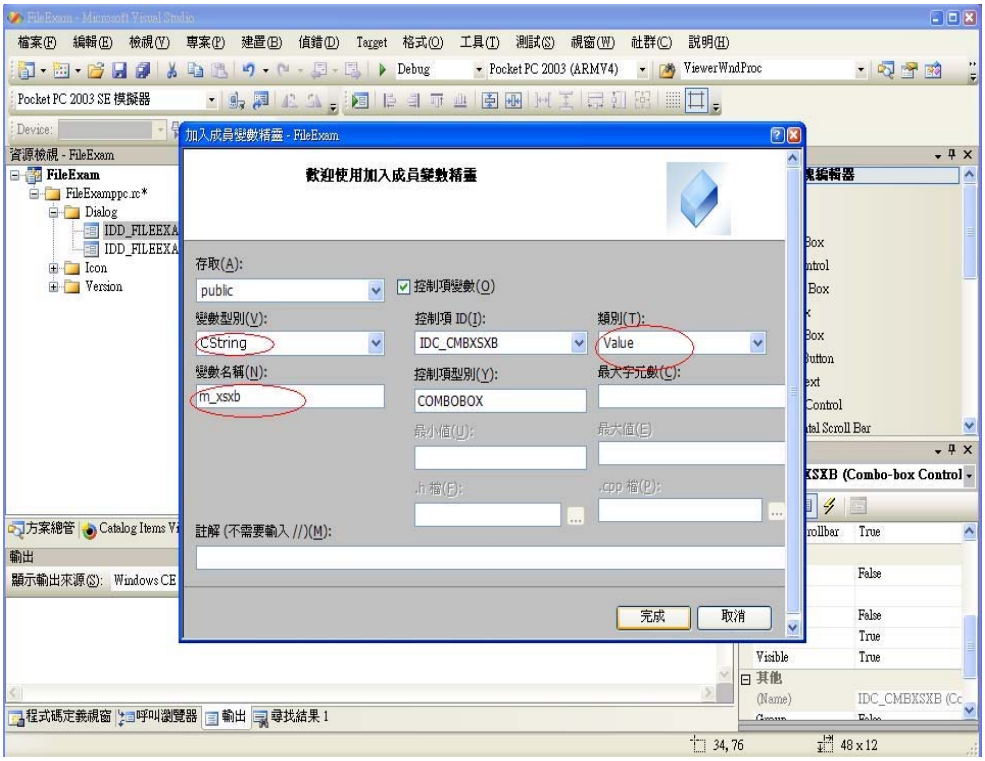
Q. 在加入成員變數的對話框上，將類別選單改為「**Value**」變數型別改為「**CString**」變數名稱為「**m_bjmc**」。



R. 切換左方視窗到資源檢視，準備開始加入變數。在要加入變數的「**Combo Box**」對話方塊上（**IDC_CMBXSXB**）按下滑鼠右鍵，選擇［加入變數］。

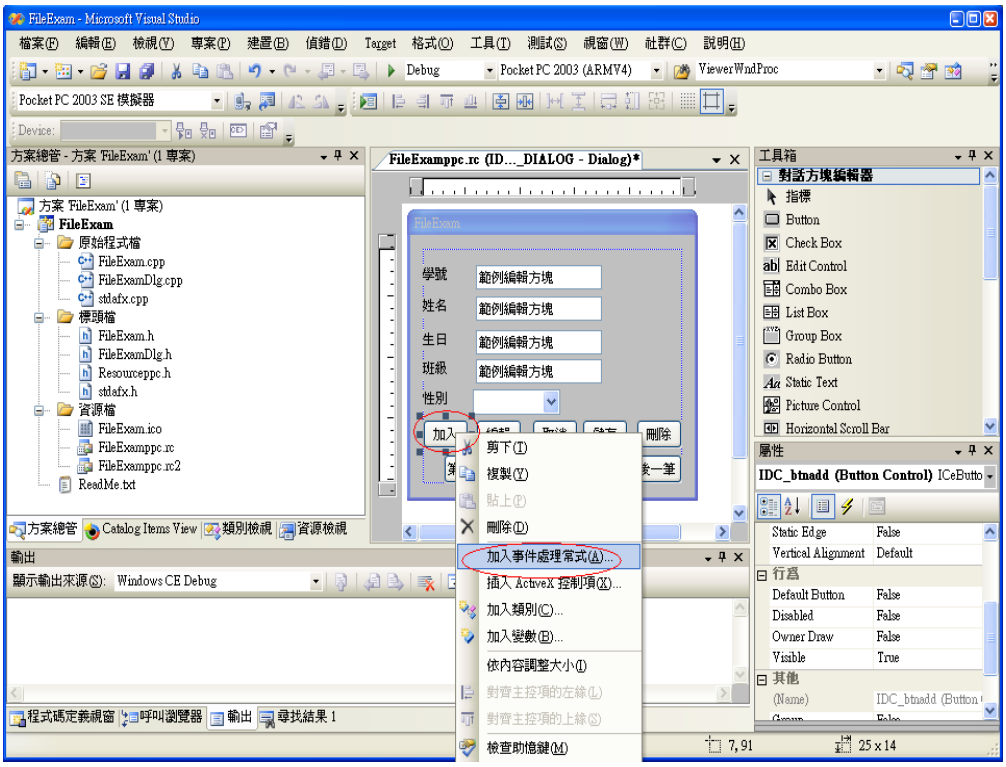


S. 在加入成員變數的對話框上，將類別選單改為「**Value**」變數型別改為「**CString**」變數名稱為「**m_xsxb**」

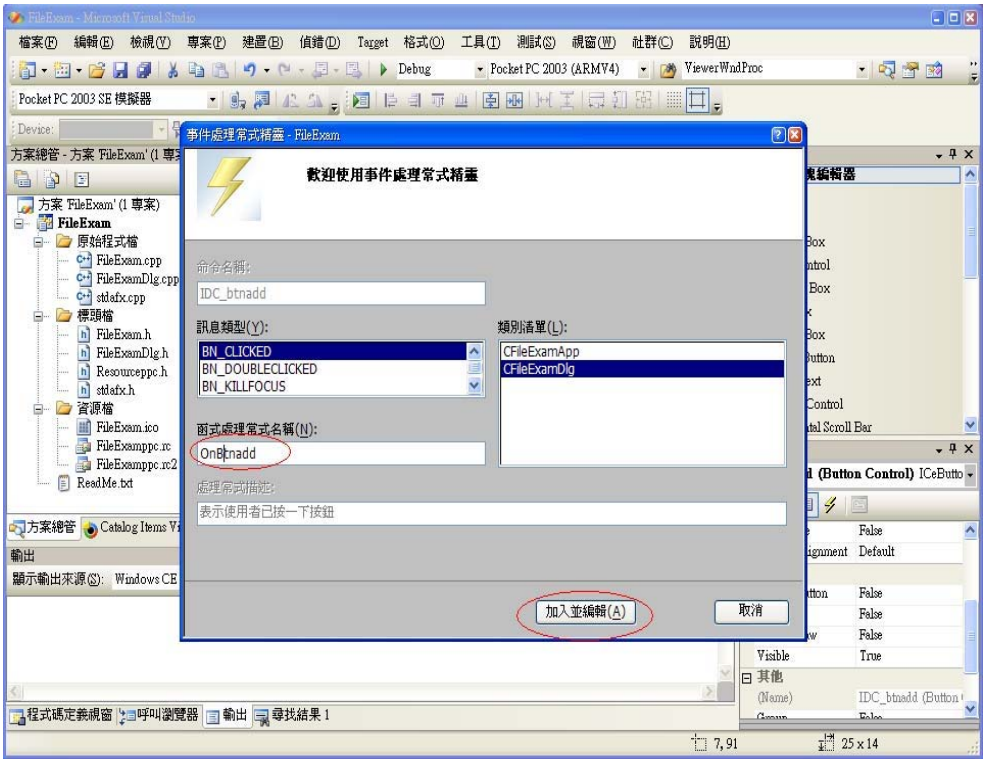


T. 切換左方視窗到資源檢視，準備開始加入事件處理常式。在要加入事件 處理常式（分別為「加入」編輯」取消」儲存」刪除」第一

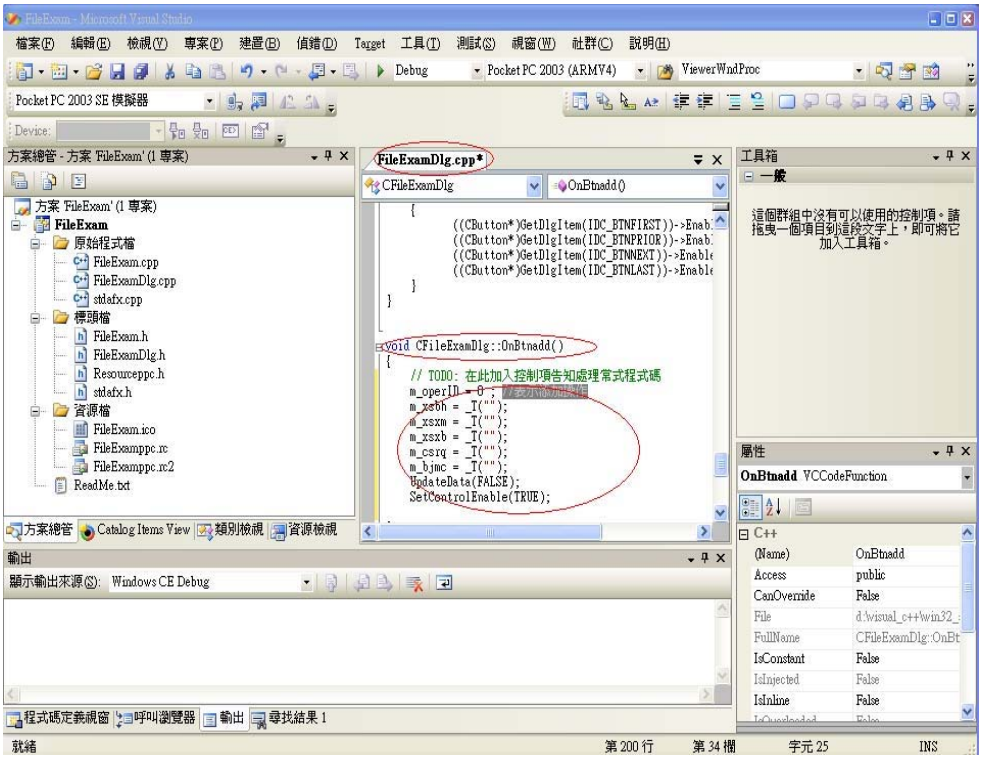
筆」、「前一筆」、「下一筆」和「最後一筆」的**Button** 對話方塊上按下 滑鼠右鍵，選擇［加入事件處理常式］。



U. 選取「加入」事件訊息類型後，更改函式處理常式名稱為 **OnBtnadd**，接下來按下［加入並編輯］按鈕。



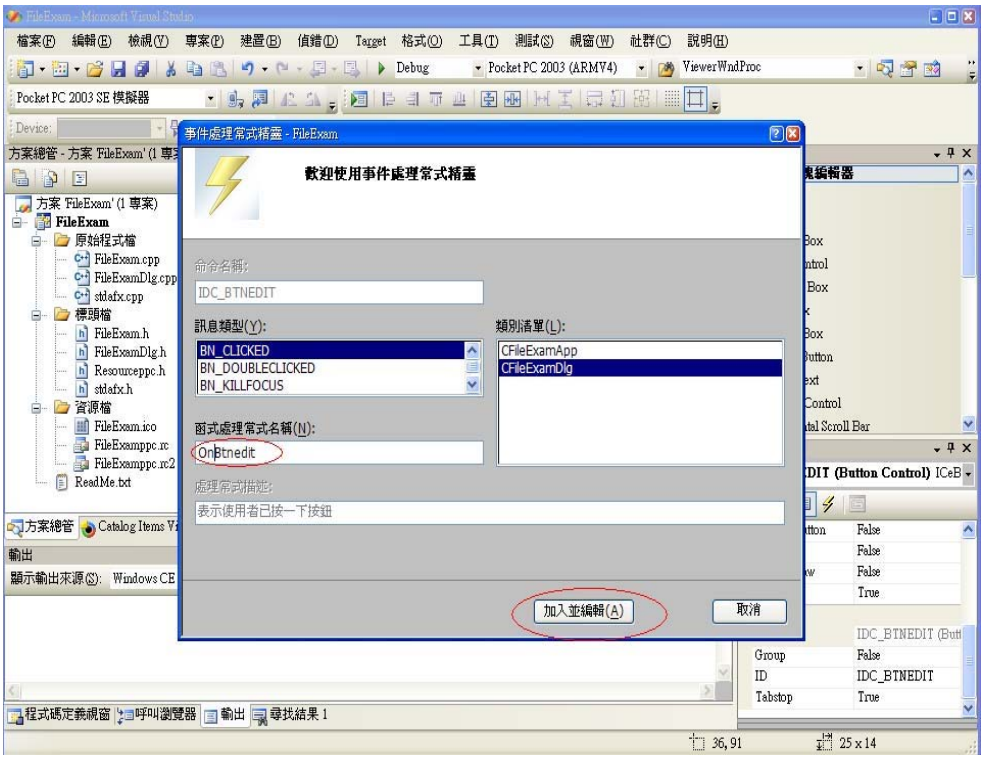
Ⅴ. 程式編輯視窗開啟在**FileExamDlg.cpp**，在**void CFileExamDlg::OnBtnadd()**後，加入事件處理常式程式碼。



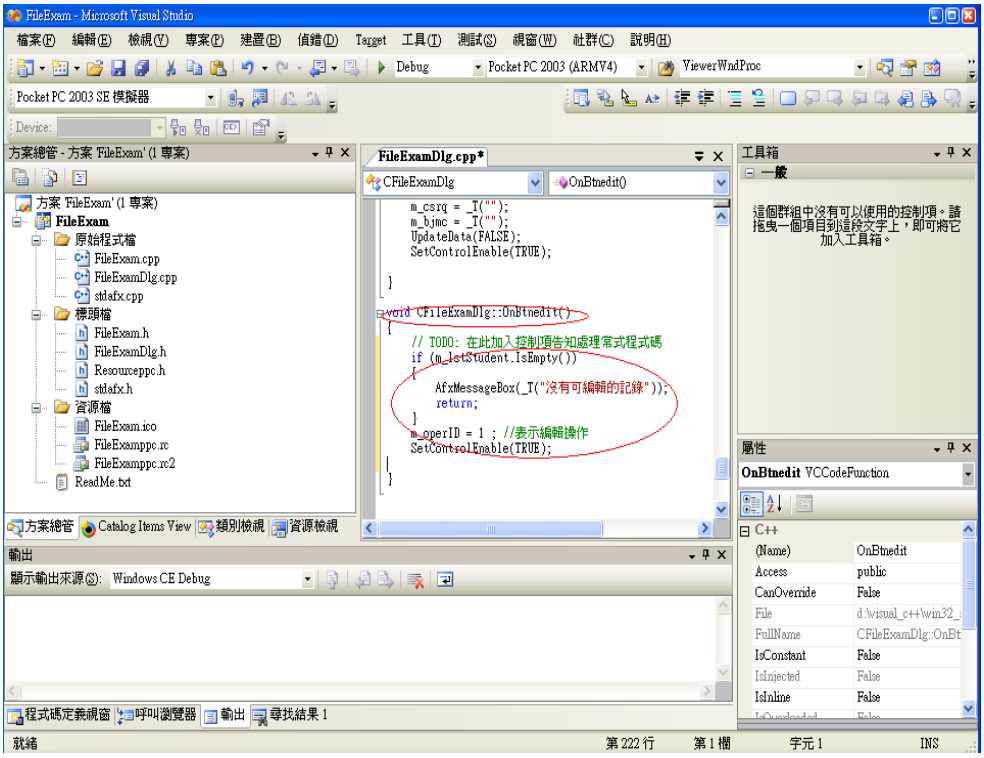
加入的程式碼 ：

```
m_operID = 0 ;     //  表示添加操作
m_xsbh = _T("");
m_xsxm = _T("");
m_xsxb = _T("");
m_csrq = _T("");
m_bjmc = _T("");
UpdateData(FALSE);
SetControlEnable(TRUE);
```

Ⅵ. 選取「編輯」事件訊息類型後，更改函式處理常式名稱為 **OnBtnedit**，接下來按下［加入並編輯］按鈕。



X. 程式編輯視窗開啟在**FileExamDlg.cpp**，在**void CFileexamDlg::OnBtnedit()**後，加入事件處理常式程式碼。

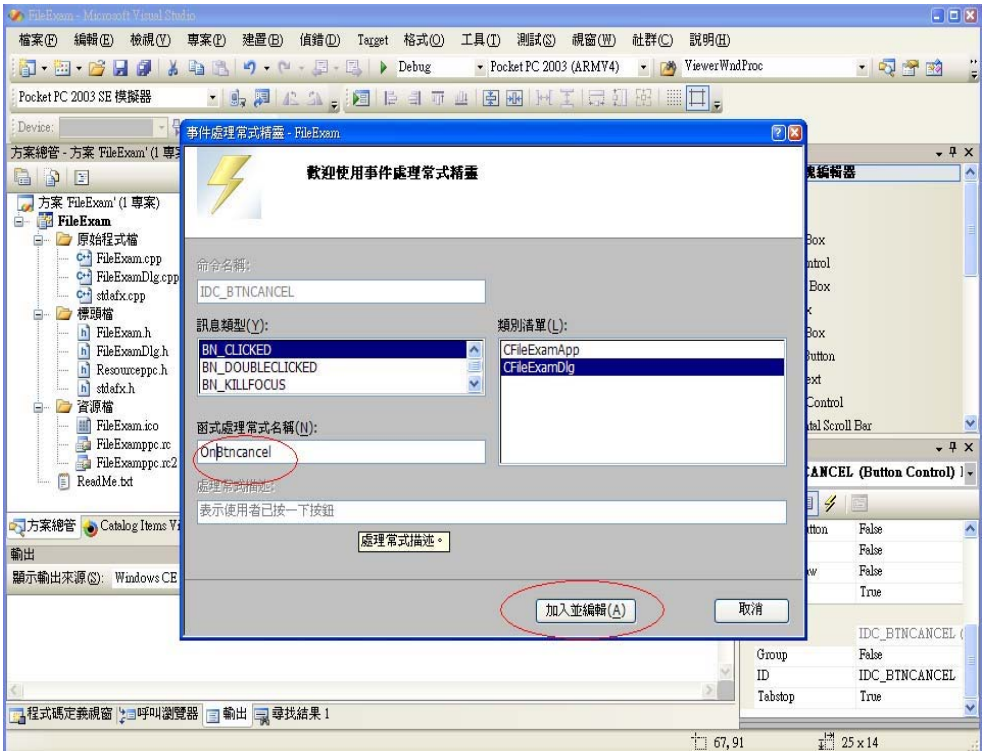


加入的程式碼 ：

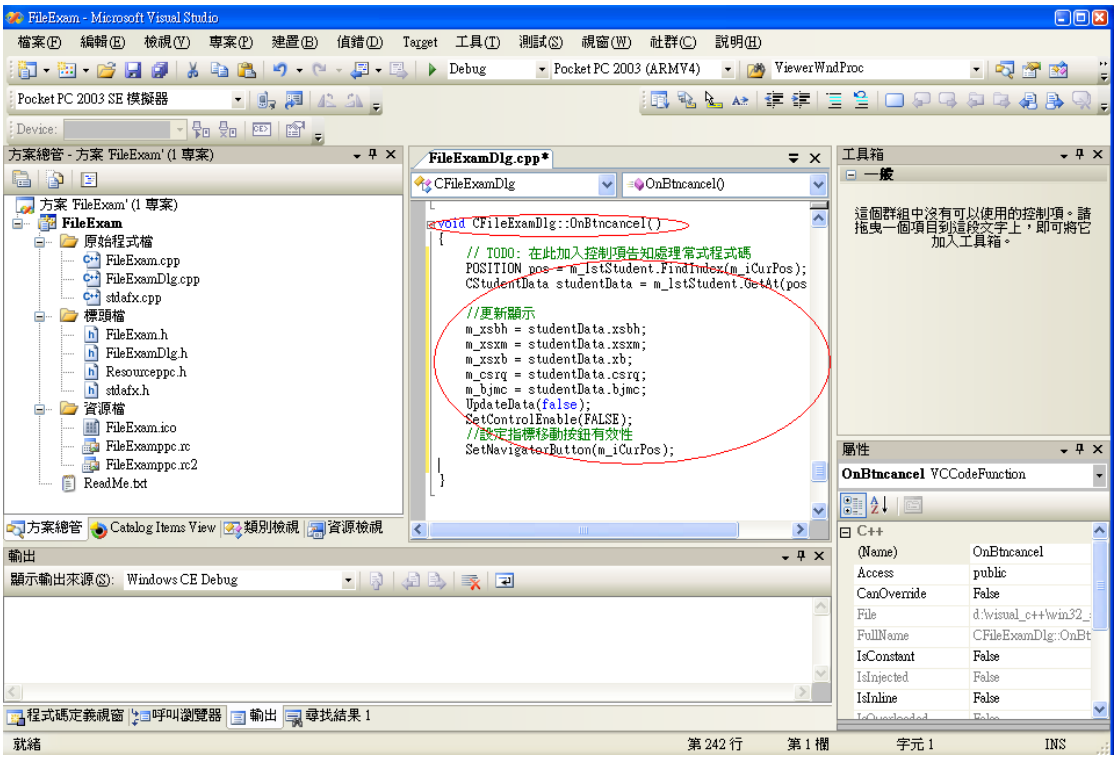
if (m_lstStudent.IsEmpty())

```
{
    AfxMessageBox(_T("沒有可編輯的記錄"));
    return;
}
m_operID = 1 ;    //  表示編輯操作
SetControlEnable(TRUE);
```

Y. 選取「取消」事件訊息類型後，更改函式處理常式名稱為 **OnBtncancel**， 接下來按下［加入並編輯］按鈕。



Z. 程式編輯視窗開啟在**FileExamDlg.cpp**，在**void CFileExamDlg::OnBtncancel()**後，加入事件處理常式程式碼。



加入的程式碼 ：

POSITION pos = m_lstStudent.FindIndex(m_iCurPos);

CStudentData studentData = m_lstStudent.GetAt(pos);

// 更新顯示

m_xsbh = studentData.xsbh;

m_xsxm = studentData.xsxm;

m_xsxb = studentData.xb;

m_csrq = studentData.csrq;

m_bjmc = studentData.bjmc;

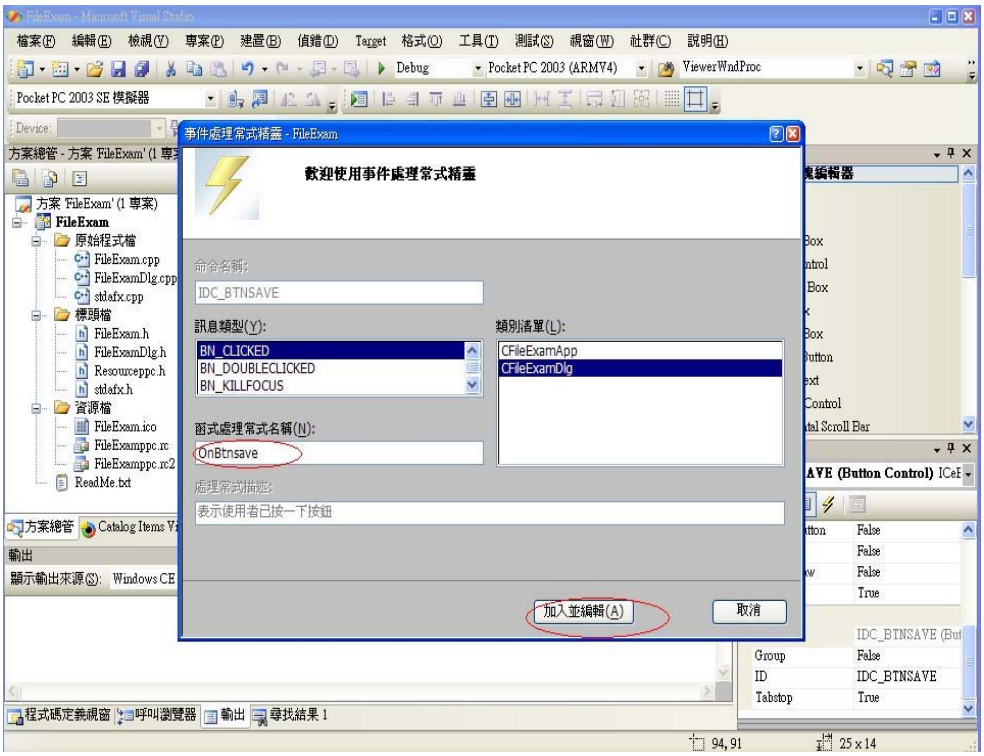
UpdateData(false);

SetControlEnable(FALSE);

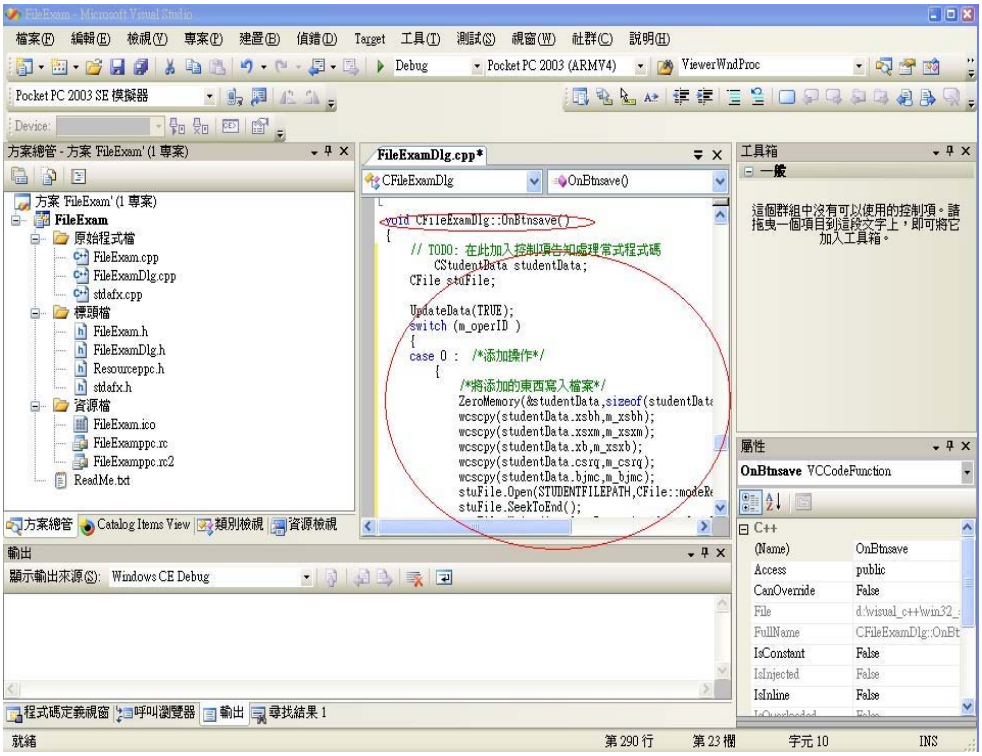
// 設定指標移動按鈕有效性

SetNavigatorButton(m_iCurPos);

AA. 選取「儲存」事件訊息類型後，更改函式處理常式名稱為 **OnBtnsave**， 接下來按下［加入並編輯］按鈕。



BB. 程式編輯視窗開啟在**FileExamDlg.cpp**，在**void CFileExamDlg::OnBtnsave()**後，加入事件處理常式程式碼。



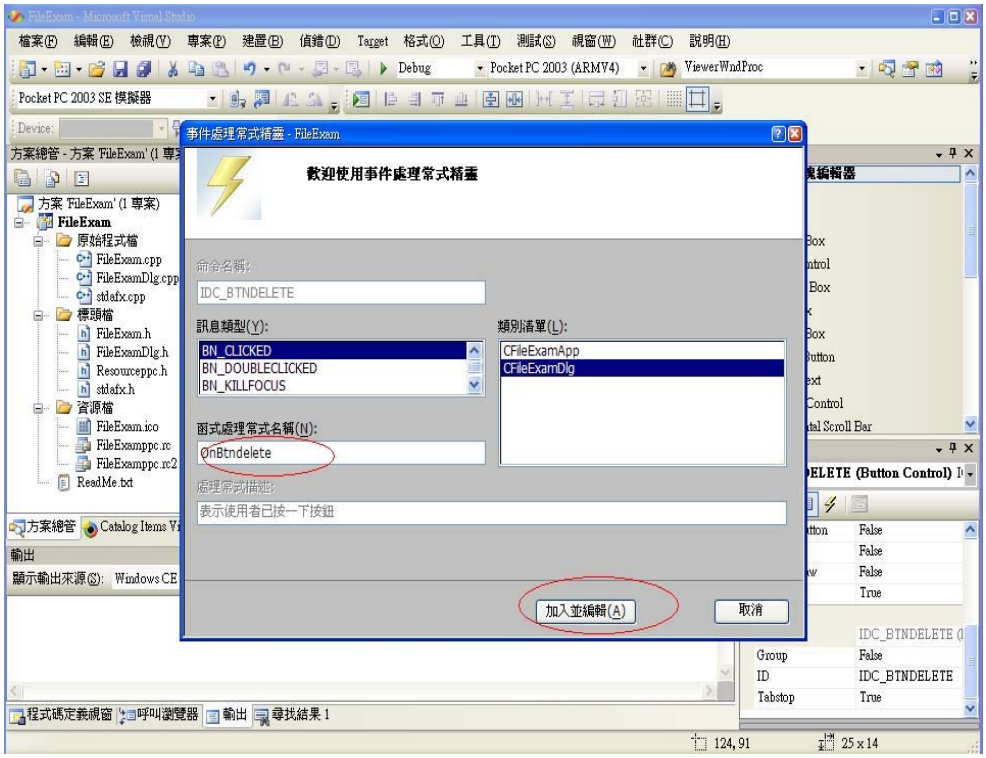
加入的程式碼 ：

CStudentData studentData;

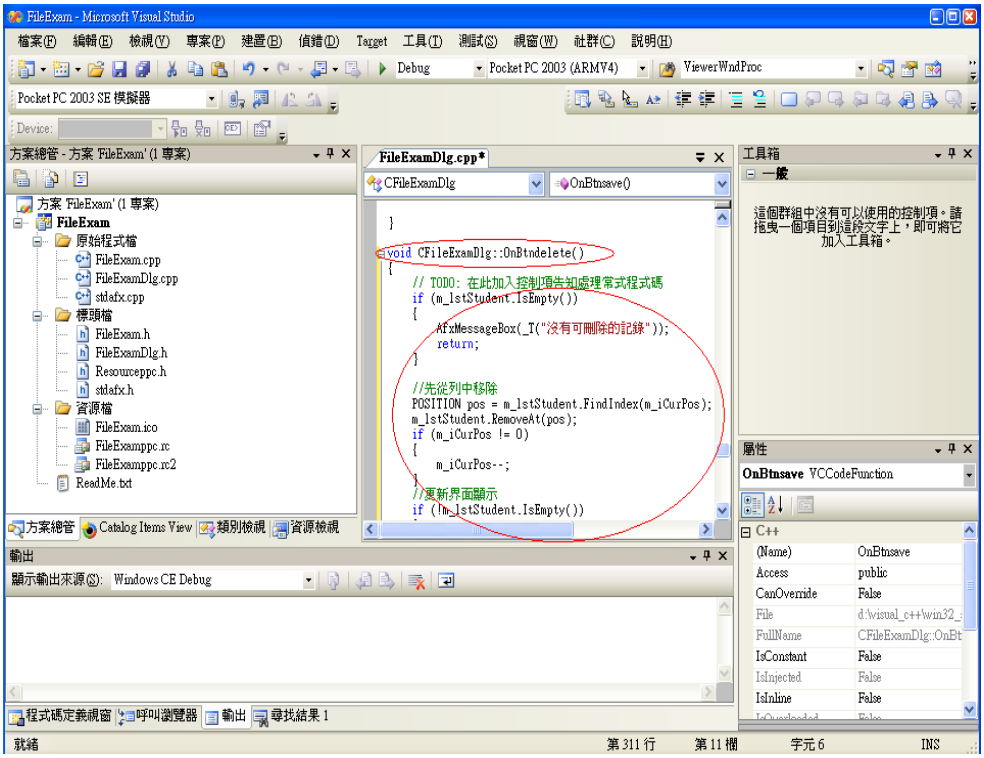
```
CFile stuFile;

UpdateData(TRUE);
switch (m_operID )
{
    case 0 :  //  添加操作
    {
        //  將添加的東西寫入檔案
        ZeroMemory(&studentData,sizeof(studentData));
        wcscpy(studentData.xsbh,m_xsbh);
        wcscpy(studentData.xsxm,m_xsxm);
        wcscpy(studentData.xb,m_xsxb);
        wcscpy(studentData.csrq,m_csrq);
        wcscpy(studentData.bjmc,m_bjmc);
        stuFile.Open(STUDENTFILEPATH,CFile::modeRead |
                        CFile::modeWrite);
        stuFile.SeekToEnd();
        stuFile.Write(&studentData,sizeof(studentData));
        stuFile.Close();

        //  更新內存列
        m_lstStudent.AddTail(studentData);

        SetControlEnable(FALSE);

        //  設定指標移動按鈕有效性
        m_iCurPos = m_lstStudent.GetCount()-1;
        SetNavigatorButton(m_iCurPos);
        break;
    }
    case 1:  //  編輯操作
    {
        //  將添加的東西寫入檔案
        ZeroMemory(&studentData,sizeof(studentData));
        wcscpy(studentData.xsbh,m_xsbh);
        wcscpy(studentData.xsxm,m_xsxm);
        wcscpy(studentData.xb,m_xsxb);
        wcscpy(studentData.csrq,m_csrq);
```

```
wcscpy(studentData.bjmc,m_bjmc);
stuFile.Open(STUDENTFILEPATH,CFile::modeRead |
                    CFile::modeWrite);

 stuFile.Seek(sizeof(studentData)*(m_iCurPos) ,CF
                    ile::begin);
stuFile.Write(&studentData,sizeof(studentData));
stuFile.Close();

//   更新內存列

 m_lstStudent.SetAt(m_lstStudent.FindIndex(m_iCurPos),stu
 dentData);

SetControlEnable(FALSE);

//   設定指標移動按鈕有效性
SetNavigatorButton(m_iCurPos);
break;
}
```

CC. 選取「刪除」事件訊息類型後，更改函式處理常式名稱為 **OnBtndelete**，接下來按下［加入並編輯］按鈕。



DD. 程式編輯視窗開啟在**FileExamDlg.cpp**，在**void CFileExamDlg::OnBtndelete()**後，加入事件處理常式程式碼。



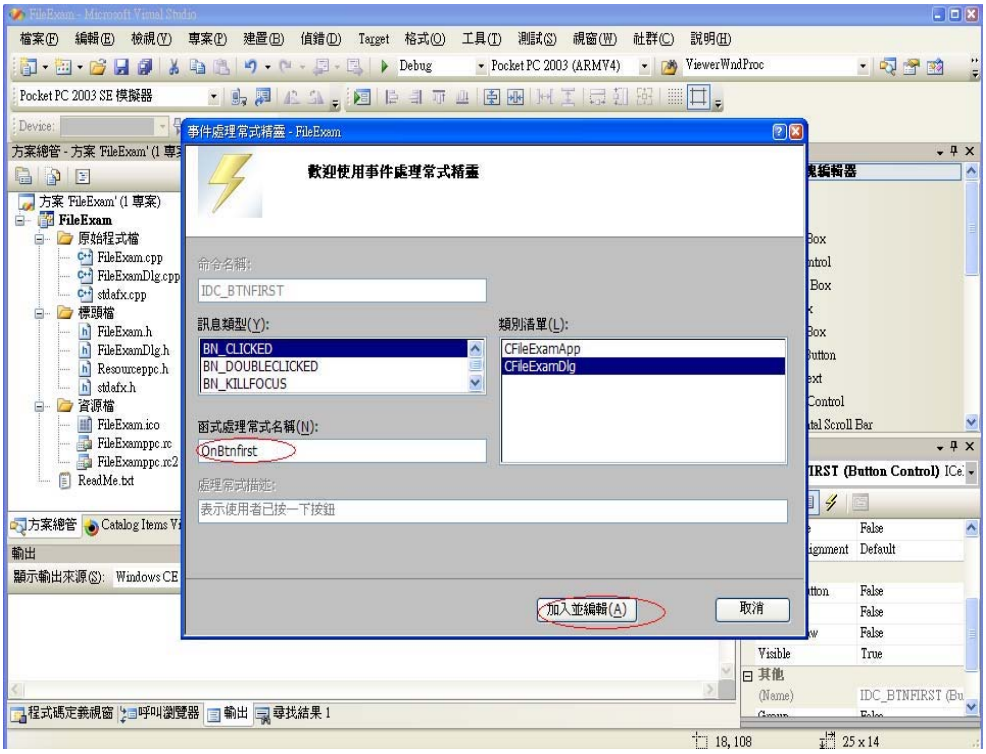
加入的程式碼 ：

```
if (m_lstStudent.IsEmpty())
{
    AfxMessageBox(_T("沒有可刪除的記錄"));
    return;
}

//  先從列中移除
POSITION pos = m_lstStudent.FindIndex(m_iCurPos);
m_lstStudent.RemoveAt(pos);
if (m_iCurPos != 0)
{
    m_iCurPos--;
}

//更新界面顯示
if (!m_lstStudent.IsEmpty())
{
    pos = m_lstStudent.FindIndex(m_iCurPos);
```

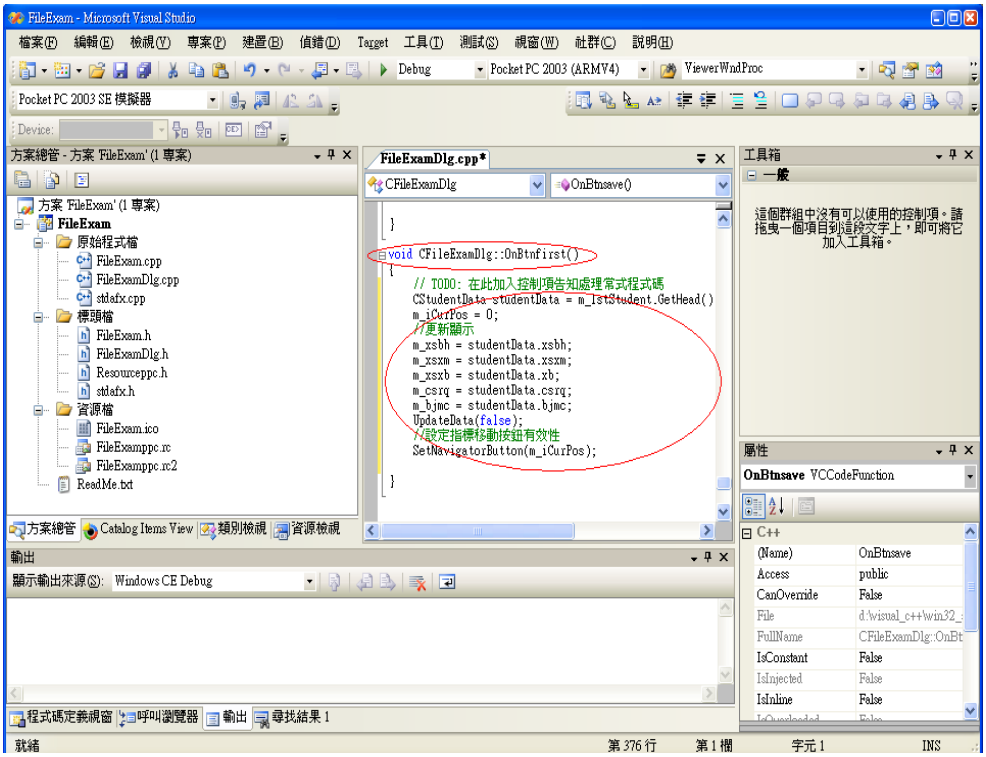
```
        CStudentData studentData = m_lstStudent.GetAt(pos);

        //  更新顯示
        m_xsbh = studentData.xsbh;
        m_xsxm = studentData.xsxm;
        m_xsxb = studentData.xb;
        m_csrq = studentData.csrq;
        m_bjmc = studentData.bjmc;
        UpdateData(false);
}
else
{
        //  更新顯示
        m_xsbh = _T("");
        m_xsxm = _T("");
        m_xsxb = _T("");
        m_csrq = _T("");
        m_bjmc = _T("");
        UpdateData(false);
}
SetControlEnable(FALSE);

//  設定指標移動按鈕有效性
SetNavigatorButton(m_iCurPos);

//  將檔案重寫
CStudentData studentData;
CFile stuFile;
stuFile.Open(STUDENTFILEPATH,CFile::modeCreate |
                CFile::modeWrite);

pos = m_lstStudent.GetHeadPosition();
for (int i=0;i<m_lstStudent.GetCount();i++)
{
        ZeroMemory(&studentData,sizeof(studentData));
        studentData = m_lstStudent.GetNext(pos);
        stuFile.Write(&studentData,sizeof(studentData));
}
```

stuFile.Close();

EE. 選取「第一筆」事件訊息類型後，更改函式處理常式名稱為 **OnBtnfirst**，接下來按下［加入並編輯］按鈕。



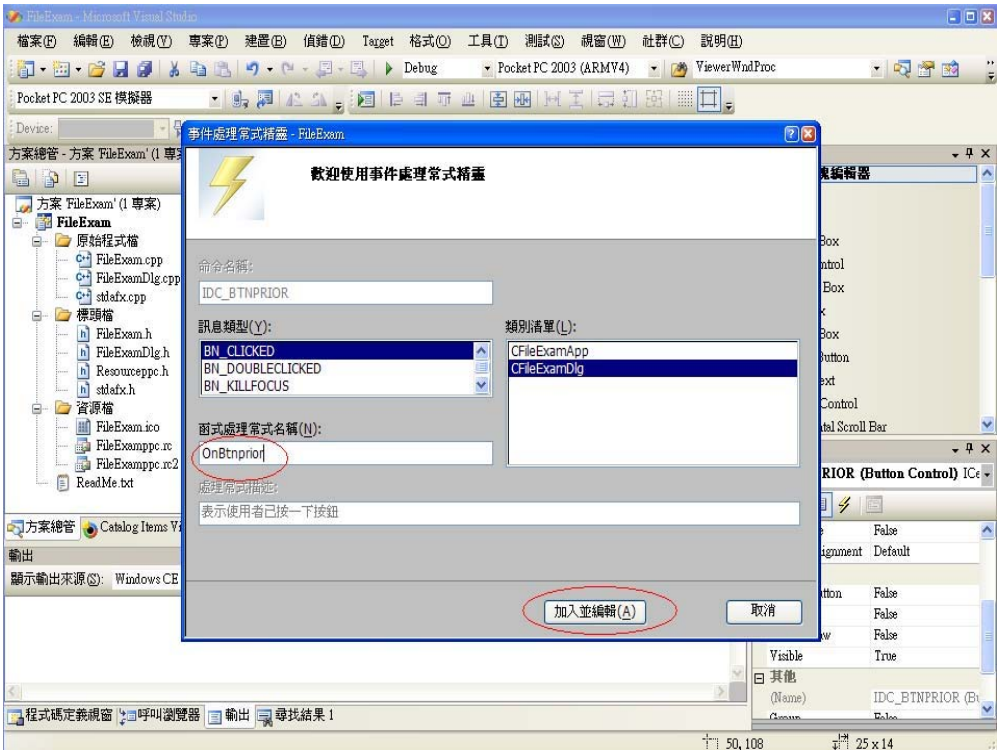
FF. 程式編輯視窗開啟在**FileExamDlg.cpp**，在**void CFileExamDlg::OnBtnfirst()**後，加入事件處理常式程式碼。



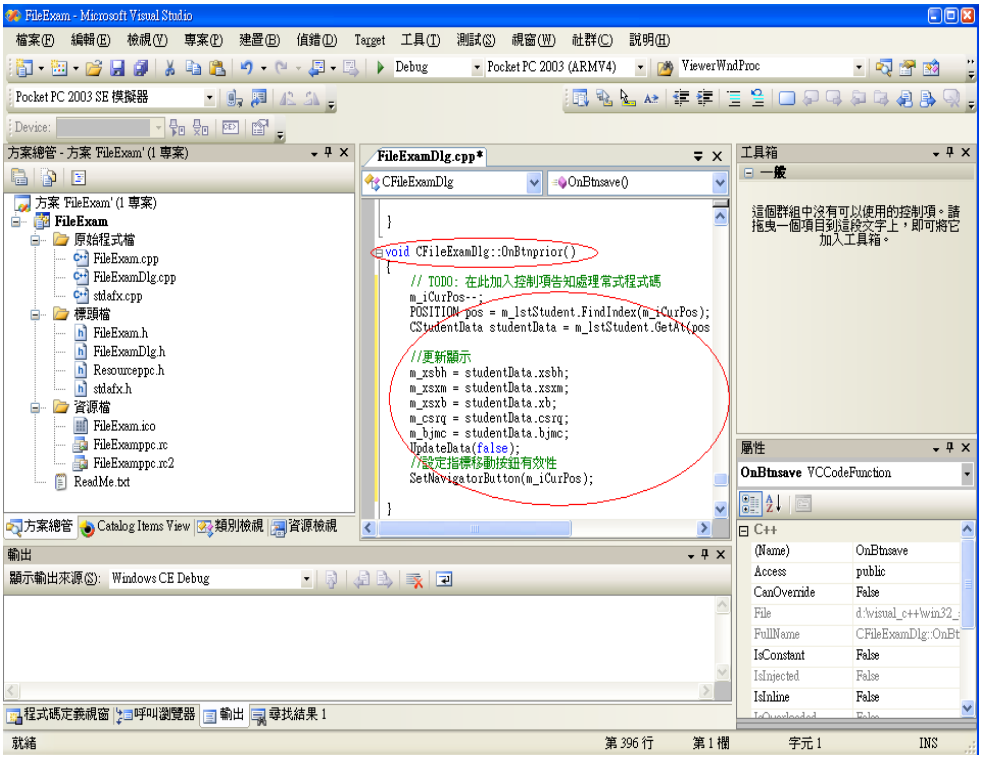
加入的程式碼 ：

```
CStudentData studentData = m_lstStudent.GetHead();
m_iCurPos = 0;

//  更新顯示
m_xsbh = studentData.xsbh;
m_xsxm = studentData.xsxm;
m_xsxb = studentData.xb;
m_csrq = studentData.csrq;
m_bjmc = studentData.bjmc;
UpdateData(false);

//  設定指標移動按鈕有效性
SetNavigatorButton(m_iCurPos);
```

GG. 選取「前一筆」事件訊息類型後，更改函式處理常式名稱為 **OnBtnprior**，接下來按下［加入並編輯］按鈕。



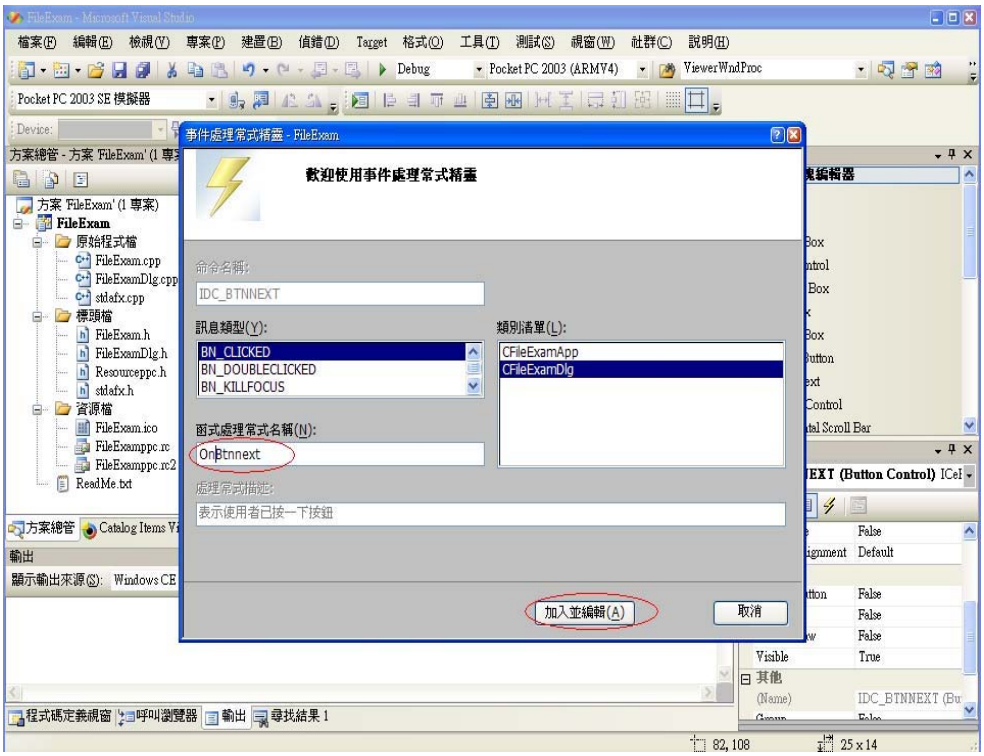
HH. 程式編輯視窗開啟在**FileExamDlg.cpp**，在**void CFileExamDlg::OnBtnprior()**後，加入事件處理常式程式碼。



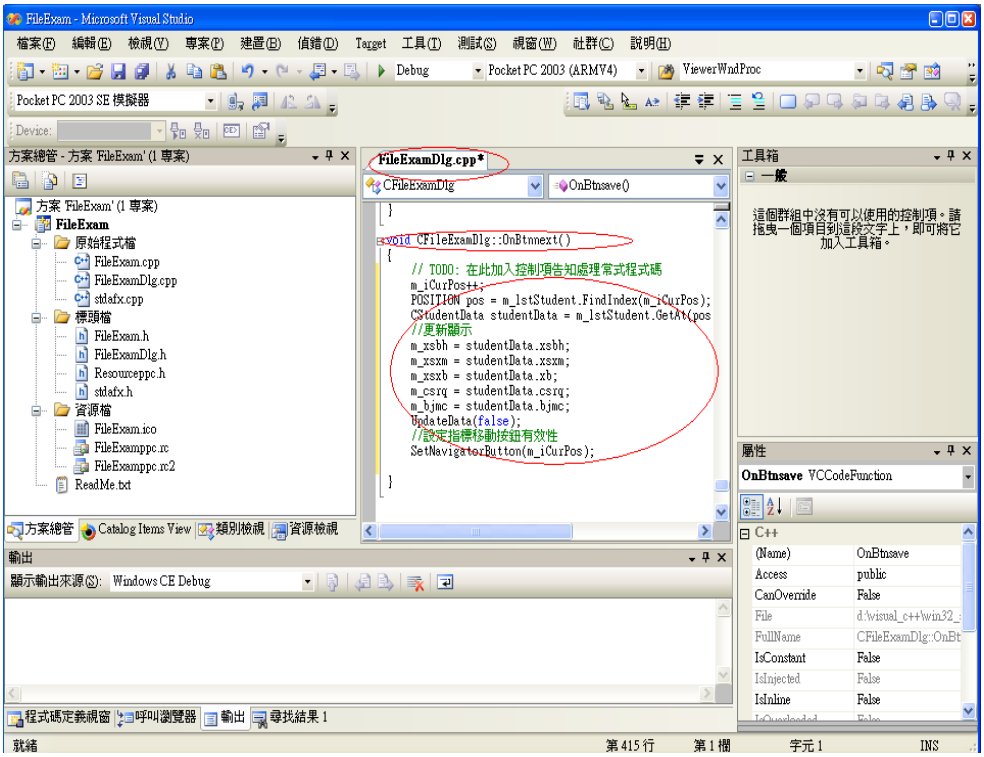
加入的程式碼 ：

```
m_iCurPos--;
POSITION pos = m_lstStudent.FindIndex(m_iCurPos);
CStudentData studentData = m_lstStudent.GetAt(pos);

//  更新顯示
m_xsbh = studentData.xsbh;
m_xsxm = studentData.xsxm;
m_xsxb = studentData.xb;
m_csrq = studentData.csrq;
m_bjmc = studentData.bjmc;
UpdateData(false);

//  設定指標移動按鈕有效性
SetNavigatorButton(m_iCurPos);
```

II. 選取「下一筆」事件訊息類型後，更改函式處理常式名稱為 **OnBtnnext**，接下來按下［加入並編輯］按鈕。



JJ. 程式編輯視窗開啟在**FileExamDlg.cpp**，在**void CFileExamDlg::OnBtnnext()**後，加入事件處理常式程式碼。

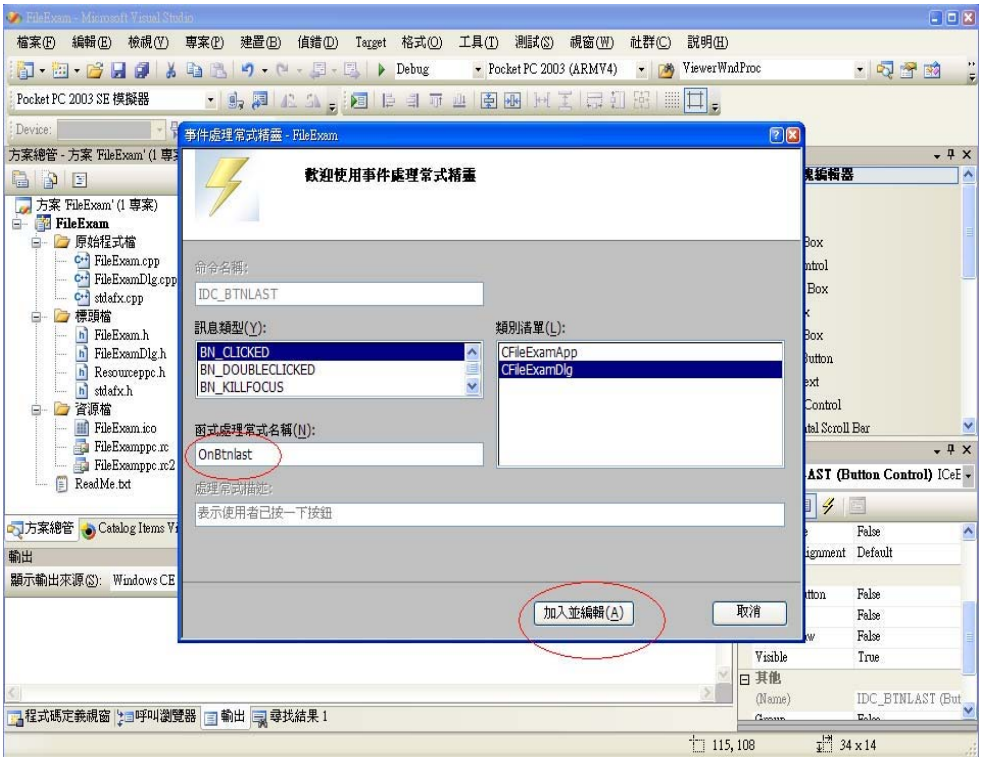


加入的程式碼 ：

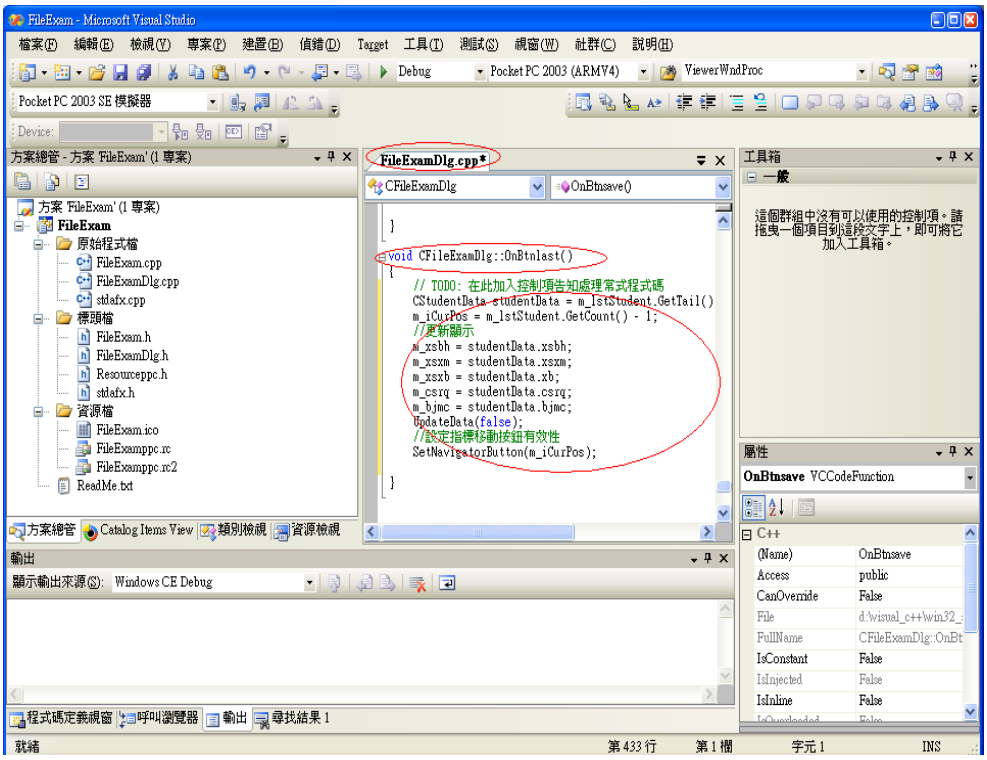
m_iCurPos++;

```
POSITION pos = m_lstStudent.FindIndex(m_iCurPos);
CStudentData studentData = m_lstStudent.GetAt(pos);

//  更新顯示
m_xsbh = studentData.xsbh;
m_xsxm = studentData.xsxm;
m_xsxb = studentData.xb;
m_csrq = studentData.csrq;
m_bjmc = studentData.bjmc;
UpdateData(false);

//  設定指標移動按鈕有效性
SetNavigatorButton(m_iCurPos);
```

KK. 選取「最後一筆」事件訊息類型後，更改函式處理常式名稱為 **OnBtnlast**，接下來按下［加入並編輯］按鈕。



LL. 程式編輯視窗開啟在**FileExamDlg.cpp**，在**void CFileExamDlg::OnBtnlast()**後，加入事件處理常式程式碼。

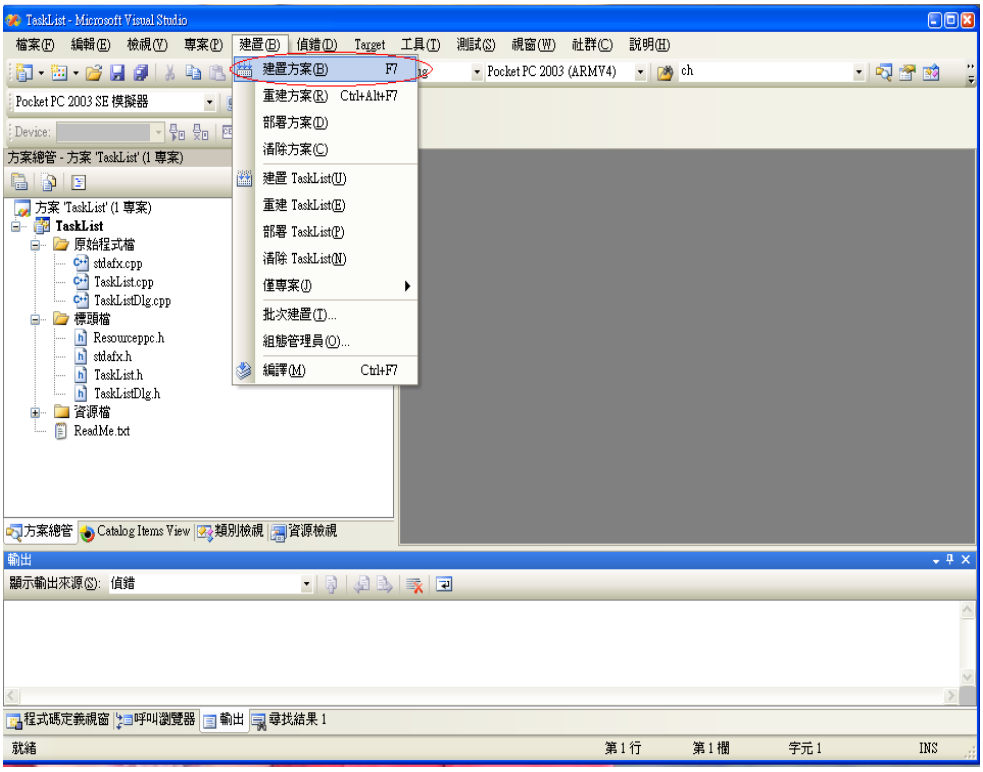


加入的程式碼 ：

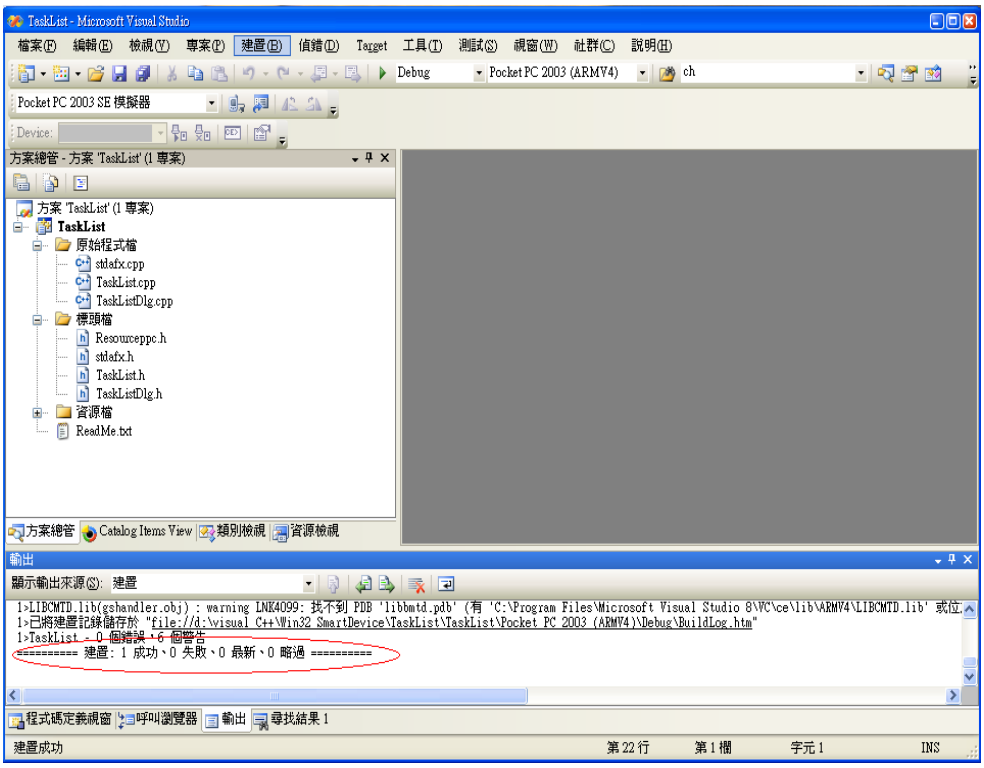
```
CStudentData studentData = m_lstStudent.GetTail();
m_iCurPos = m_lstStudent.GetCount() - 1;

//  更新顯示
m_xsbh = studentData.xsbh;
m_xsxm = studentData.xsxm;
m_xsxb = studentData.xb;
m_csrq = studentData.csrq;
m_bjmc = studentData.bjmc;
UpdateData(false);

//  設定指標移動按鈕有效性
SetNavigatorButton(m_iCurPos);
```

4. 編譯和執行

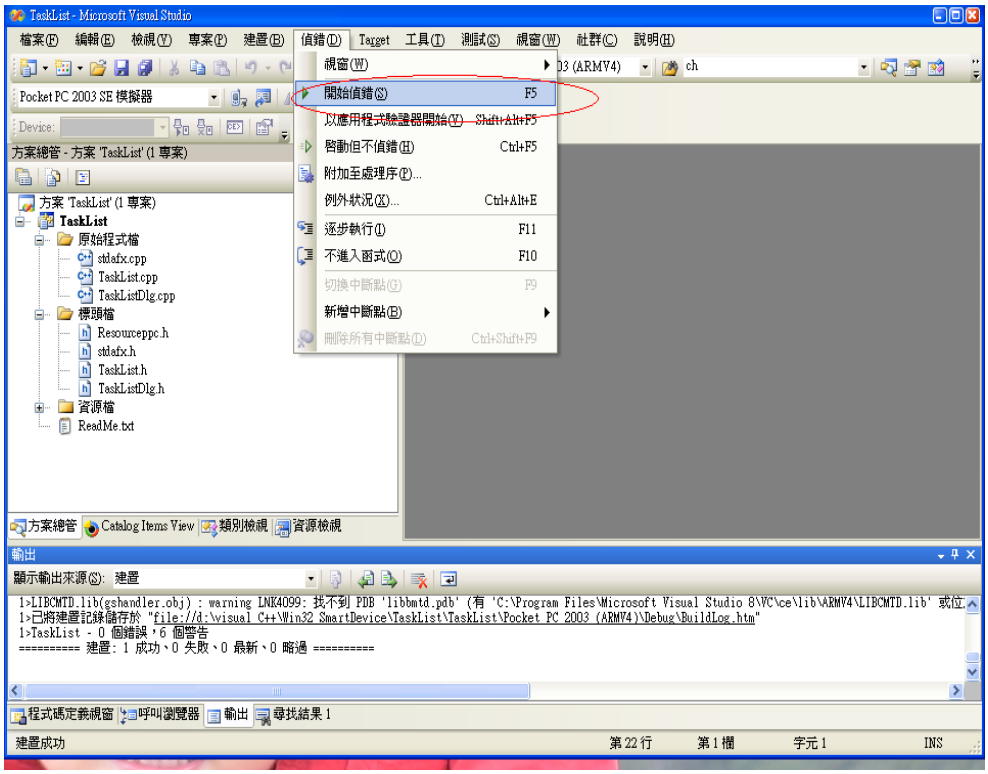
A. 在**Visual Studio 2005** 整合式開發環境中，點選[建置] -> [建置專案]後便會開始編譯整個專案。



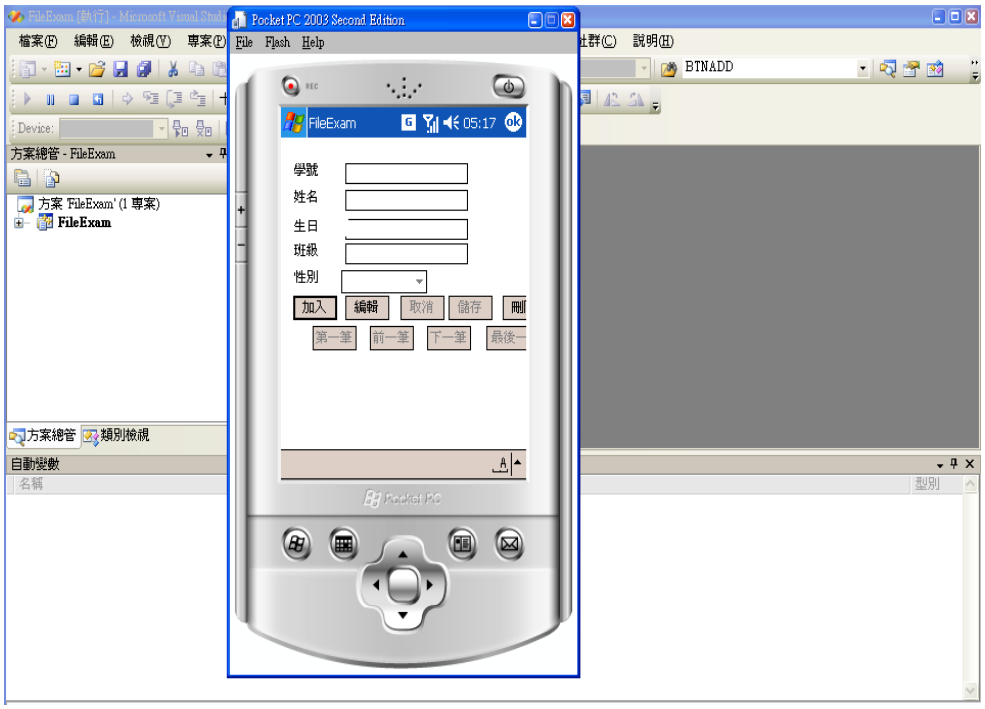
B. 編譯成功後，在**Visual Studio 2005** 整合式開發環境下方的輸出視窗中顯示建置成功。



C. 在**Visual Studio 2005** 整合式開發環境中，點選[偵錯] -> [開始偵錯]後便會開始對此專案執行並偵錯。


TaskList - Microsoft Visual Studio
檔案(F) 編輯(E) 檢視(V) 專案(P) 建置(B) 偵錯(D) Target 工具(T) 測試(S) 視窗(W) 社群(C) 說明(H)
視窗(W)
開始偵錯(S) F5
以應用程式驗證器開始(Y) Shift+Alt+F5
啟動但不偵錯(H) Ctrl+F5
附加至處理序(P)...
例外狀況(X)... Ctrl+Alt+E
逐步執行(I) F11
不進入函式(O) F10
切換中斷點(G) F9
新增中斷點(B)
刪除所有中斷點(D) Ctrl+Shift+F9
Pocket PC 2003 SE 模擬器
方案總管 - 方案 'TaskList' (1 專案)
方案 'TaskList' (1 專案)
TaskList
原始程式檔
stdafx.cpp
TaskList.cpp
TaskListDlg.cpp
標頭檔
Resourceppc.h
stdafx.h
TaskList.h
TaskListDlg.h
資源檔
ReadMe.txt
方案總管 Catalog Items View 類別檢視 資源檢視
輸出
顯示輸出來源(S): 建置
1>LIBCMTD.lib(gshandler.obj) : warning LNK4099: 找不到 PDB 'libbmtd.pdb' (有 'C:\Program Files\Microsoft Visual Studio 8\VC\ce\lib\ARMV4\LIBCMTD.lib' 或位
1>已將建置記錄儲存於 "file://d:\visual C++\Win32 SmartDevice\TaskList\TaskList\Pocket PC 2003 (ARMV4)\Debug\BuildLog.htm"
1>TaskList - 0 個錯誤，6 個警告
========== 建置: 1 成功、0 失敗、0 最新、0 略過 ==========
程式碼定義視窗 呼叫瀏覽器 輸出 尋找結果 1
建置成功
第 22 行
第 1 欄
字元 1
INS

D. 執行結果如下圖所示。


Pocket PC 2003 Second Edition
File Flash Help
FileExam 05:17
學號
姓名
生日
班級
性別
加入 編輯 取消 儲存
第一筆 前一筆 下一筆
方案總管 - FileExam
方案 'FileExam' (1 專案)
FileExam
方案總管 類別檢視
自動變數
名稱
型別