iPush® Embedded 教育訓練教材－ IO Module Framework

著 作 人：艾揚科技股份有限公司
(ICE Technology Corporation)

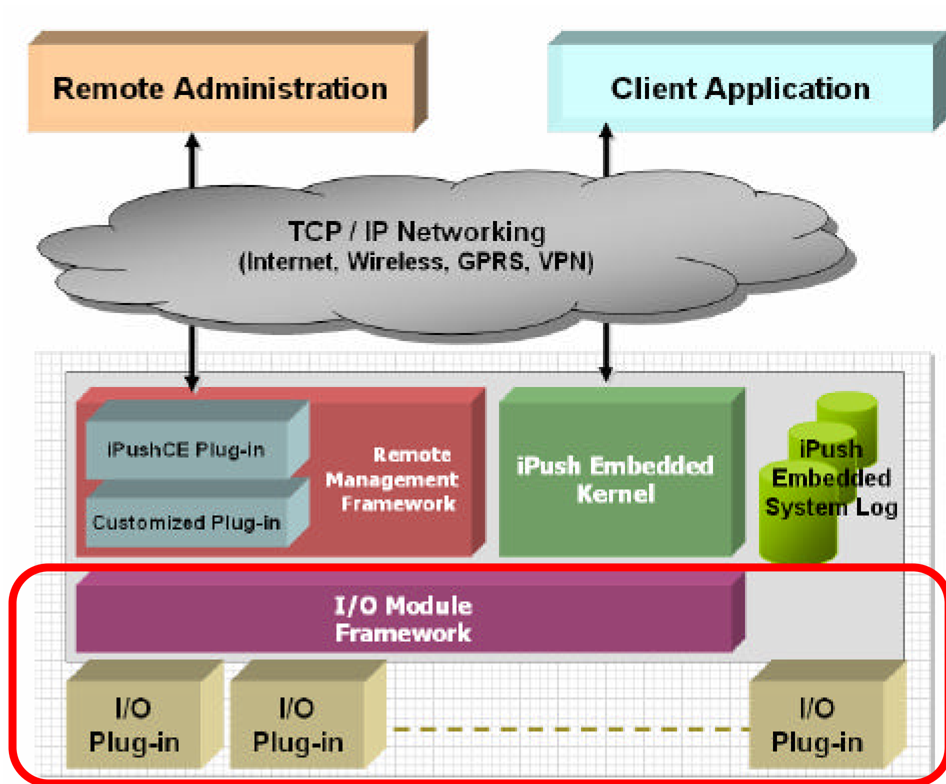
文件編號：TEmbedded-10-003-tw
版　　次：V1.0
出版時間：2004-06-28

IO Module Framework Programming

IO Module Framework 架構介紹......3
工具準備......3
IO Module Framework 基本工作原理......5
Module Kernel 提供的主要功能......6
Kernel Module 的命名慣例......7
引入的標頭檔......7
常用函式呼叫......7
從 DoReport 開始......8
Scan Kernel Module 的起始 (DoInit) 部分......9
DoStart 啟動......12
DoStop 停止工作......13
DoFinalize 結束執行......13
DoReflesh 重新起始並更新狀態......15
接收外來訊息的 DoProcessMsg......16
建立 TagObject 物件的 MyTa gCreateHandler 程序......18
讀取設定屬性的 MyPropertyHandler 程序......20
檢查裝置存在狀態的 Check State() 函式......21
Check State 檢查裝置的狀態......26

IO Module Framework 架構介紹

iPush Embedded 的訊息核心，是透過 IO Module 來對 WinCon-8000 上的 I/O 模組做存取。整體架構如下：



圖一、在 iPush Embedded 中的 IO Module Framework 位置

在接下來的內容中，我們將對 iPush Embedded 系統中的 IO Module Framework 如何透過執行緒 (Thread) 去存取 Scan Kernel 的資料，以及如何透過資料傳送給 iPush Embedded 訊息核心的方式。

工具準備

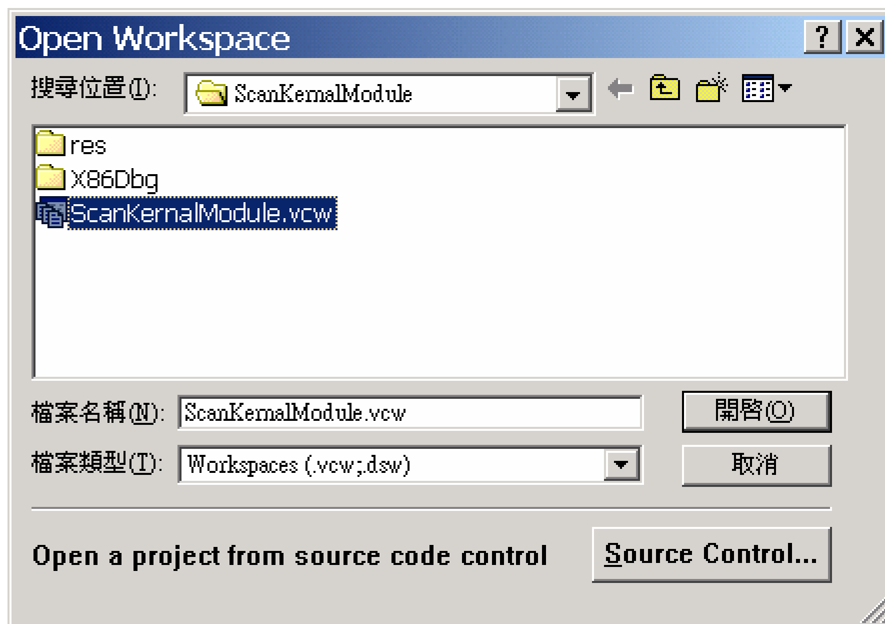
在接下來閱讀以下內容時，建議先閱讀 ICE Technology 的 IO Module Framework 說明手冊，以及 ICPDAS 的 Scan Kernel SDK 使用手冊。同時，建議最好具備 C++ 相關的程式設計經驗，以及指標、Struct、和多執行緒工作的概念。

同時，要編譯 Scan Kernel Module，您必須先建立好以下的環境：

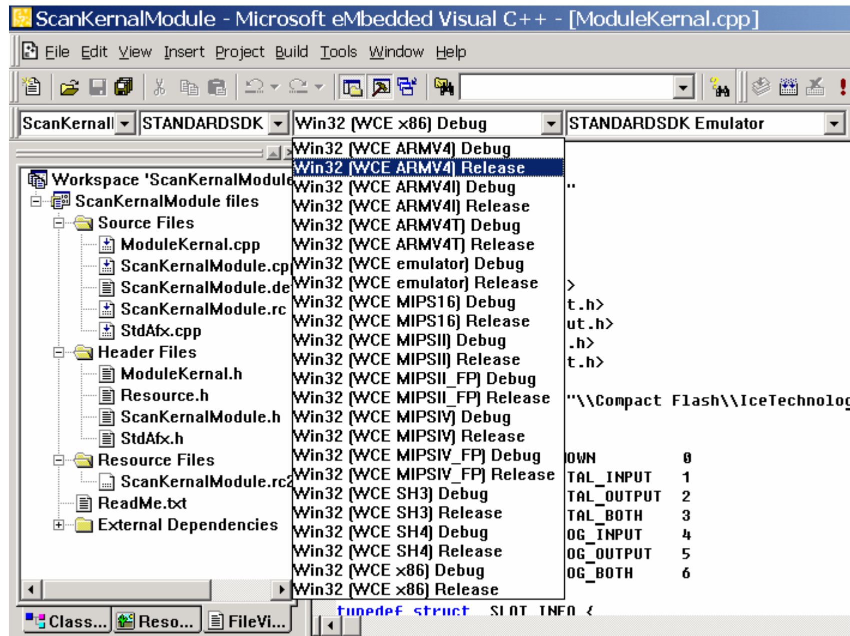
- 使用 Microsoft eMbedded Visual C++ 4.0 版
- Service Pack 2 和 QFE 更正檔，或是 Service Pack 3
- 安裝 StrongARM 的 Platform SDK 來進行程式的開發工作，安裝 eMbedded C++ 4.0 版時，預設就會安裝 WinCon-8000 所需的 Platform SDK

上面提到的軟體，都可以在 http://msdn.microsoft.com/downloads/ 上找到，檔名分別是 eVC4.exe、eVC4SP3.exe，可以利用輸入 eMbedded Visual C++ 來搜尋相關的結果。

您必須先建立以上的環境，才能順利的完成 Scan Kernel 模組的編譯工作。請在安裝 eMbedded Visual C++ 4.0 及 Services Pack 後，開啟光碟 Samples/ScanKernelModule 資料夾下的 ScanKernelModule.vcw 工作檔案。



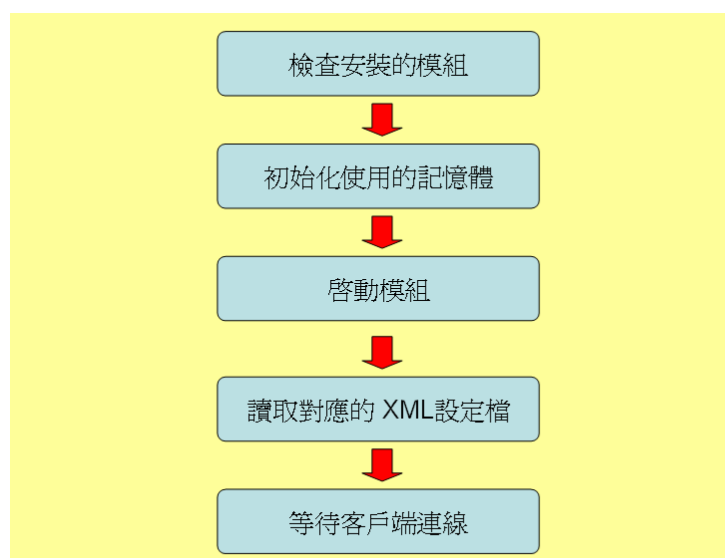
圖二、開啟 eMbedded VC++ 的 vcw workspace 檔案



圖三、編譯時要能選擇 WCE ARMV4 的選項

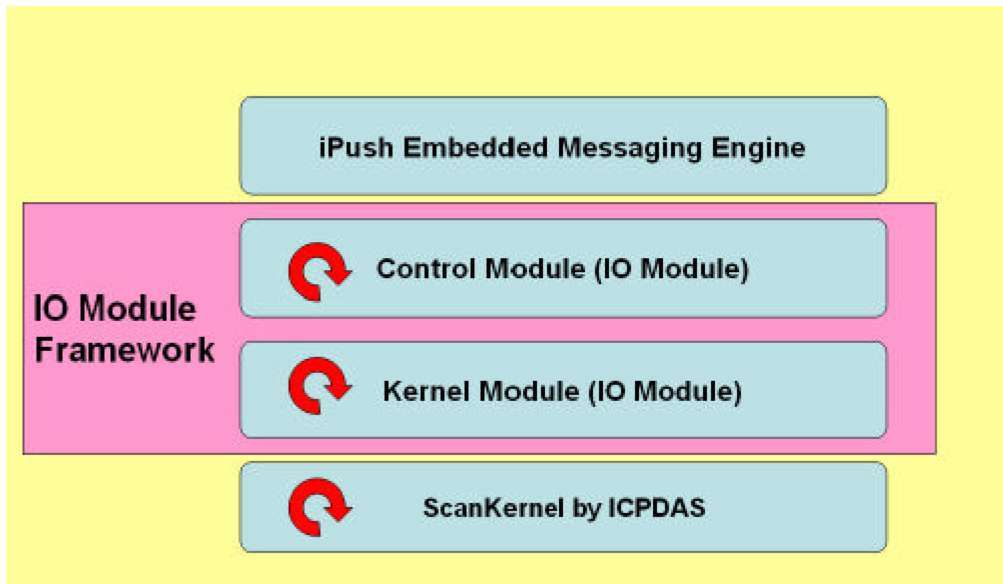
IO Module Framework 基本工作原理

系統啟動時，會透過 Scan kernel 取得目前 WinCon-8000 上安裝了哪一些 I/O 模組，接著會嘗試著去抓 I/O Module 的 XML 設定檔，然後依著 XML 的檔案設定，做 I/O 模組的資料讀取及設定工作。



圖四、初始化工作流程

IO Module 模組本身是以執行緒執行的程式，會自動的定時檢查 WinCon-8000 及從客戶端上面是否新的資料傳入。並且做資料的交換。I/O 是透過 Message Object 來做資料的傳遞。所以數值必須先轉換成資料物件後，才會進行實際的資料傳遞工作。



圖五、每個模組都有獨立執行緒工作的 IO Module Framework

在 ScanKernelModule.cpp 中，我們定義了六個供 IO Module Main Program 呼叫的函式。主要是：

- Init( )：提供初始化
- Start( )：啟動 DLL
- Stop( )：暫停 DLL 的服務
- Finalize( )：結束 DLL 執行時做的處理
- Refresh( )：重新啟動 DLL 時
- ProcessMsg( )：當 Main Program 通知 I/O Module 有資料需要被處理時的呼叫作業。

Module Kernel 提供的主要功能

- 接收從 Main Program 來的 ProcessMsg 訊息，提供給 I/O 模組處理
- 經由執行緒呼叫 CheckDevice( ) 及 CheckState( ) 偵測並處理來自 I/O 裝置的資料

Module Kernel 還會經由 MyTagCreatHander 及 MyPropertyHandler 讀取 XML 檔中的屬性設定，決定 I/O 模組的形式，以及掃描頻率 <Frequency> 以及敏感度 <Diff> 設定。

Kernel Module 的命名慣例

在 Module Kernel 的變數以及函式命名方式是依循以下的規則：

- g_ 代表全域變數、函式、物件
- ar_ 代表陣列型態的變數、函式、物件
- n_ 代表數值型態的變數、函式、物件
- proc_ 程序
- str_ 字串
- h_ Handle
- CB_ Callback 回呼函式，在此通常是用來傳送資料給 iPush Embedded 訊息傳送服務，或是透過紀錄服務，執行將資料寫入紀錄檔的工作
- f_ 浮點數變數
- dw_ Double Word 型態的資料
- uc_ unsigned char

其他的函式，例如 GetAtriOf8KModule 或是 Atri[0] 等陣列與參考值的定義，請參閱 Scan Kernel 的函式說明文件，以獲得進一步的資訊。

引入的標頭檔

在引入的 WinConAgent.h 檔案中，包含了許多 Scan Kernel 的函式。在 Module Kernel.cpp 中，我們僅使用了 GetNameOf8KModule 部份 Scankernel 的函式，使用者可以依據 WinCon-8000 上的組態，增加其他裝置(如 87k、7k) 的支援。

常用函式呼叫

- g_procCBWriteLog 執行 Log 檔寫入的工作
- g_procCBPublish 將資料提供給 iPush Embedded 傳送給客戶端的函式
- atoi stdlib 中，將字串轉換成 Integer 的函式

接下來, 我們將介紹如何由客戶端送一個 Report 命令開始, 描述整個 IO Module 工作的流程。

從 DoReport 開始

我們首先以簡單的 DoReport( ) 函式為例，介紹一下整個 I/O Module 資料收送的流程。

首先，客戶端的命令物件會被送到 ProcessMsg( ) 函式做處理。然後 ProcessMsg( ) 會根據命令物件的 isPropertyExist (lpMsgObj->isPropertyExist) 檢查 Property 是否存在，並做進一步的處理。

```
void DoProcessMsg(char *strTagPath, MessageObject *lpMsgObj)
{
    if(lpMsgObj->isPropertyExist("Report") && (!strTagPath || !strlen(strTagPath)))
    {
        DoReport();
    }
    // 以下程式碼本節省略

}
```

當 DoProcessMsg 從 Message Object 得到一個 Report 值時，就對所需的 Subject 執行 Report () 的動作，而 Report 實際執行的函式如下。

```
void DoReport()
{
    char strName[80];
    unsigned char iAtri[8];
    for(int i=1; i<MAX_SLOTS; i++)
    {
        if(GetNameOf8KModule(i, strName)==0 && strlen(strName))
        {
            MessageObject MsgObj;
            MsgObj.setStringProperty("DeviceInfo", strName);
            if(GetAtriOf8KModule(i, iAtri)==0)
            {
```

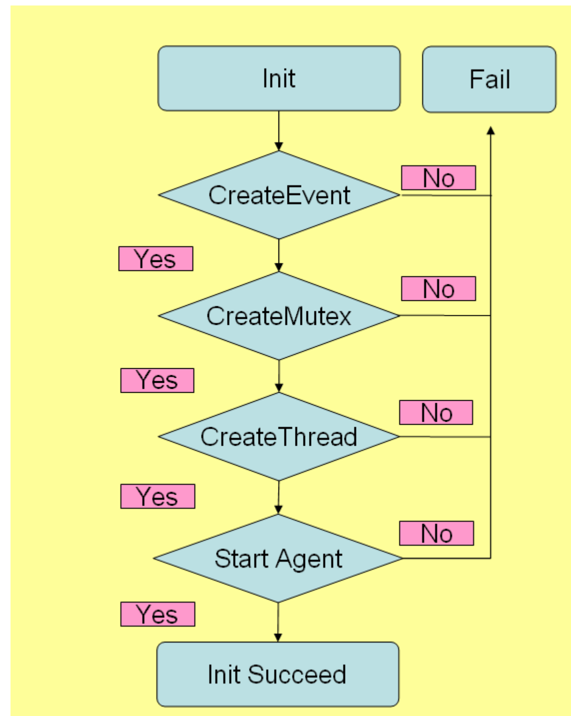
```
                MsgObj.setIntProperty("#DI", iAtri[0]);
                MsgObj.setIntProperty("#DO", iAtri[1]);
                MsgObj.setIntProperty("#AI", iAtri[2]);
                MsgObj.setIntProperty("#AO", iAtri[3]);
                MsgObj.setIntProperty("#Counter", iAtri[4]);
            }
            char strPostfix[224];
            sprintf(strPostfix, "%s_%d", strName, i);
            g_procCBPublish(g_nMyID, MESSAGE_TYPE_SUBJECT, strPostfix,
&MsgObj);
        }
    }
}
```

在上面這個部分中，我們使用了 Scan Kernel 的 GetNameOf8KModule 函式，取得 WinCon-8000 函式上的資料。也利用 GetAtriOf8Kmodule 取得模組的屬性值，並將屬性值設定給 Message Object，最後將 Message Object 的內容，利用 g_procCBPublishing 發送給 iPush Embedded 的訊息核心處理。

I/O Module 擁有自己的執行緒 Thread Worker 來執行多執行緒的工作，並透過定時 Polling 的方式來進行裝置狀態的變更及取出變化狀態。

Scan Kernel Module 的起始 (DoInit) 部分

在起始 Scan kernel Module 時，主程式會先呼叫 DoInit ( ) 工作，並執行初始化是否正常，以及建立執行緒的檢查工作。若是工作不正常，則呼叫 g_procCBWriteLog() 方法將失敗的訊息寫入，並回傳 FALSE 跳出起始程序。在確認整個起始程序可以完成之後，會將起始成功的訊息寫入，並傳回一個 True 的訊息。



圖六、Init 函式的流程圖

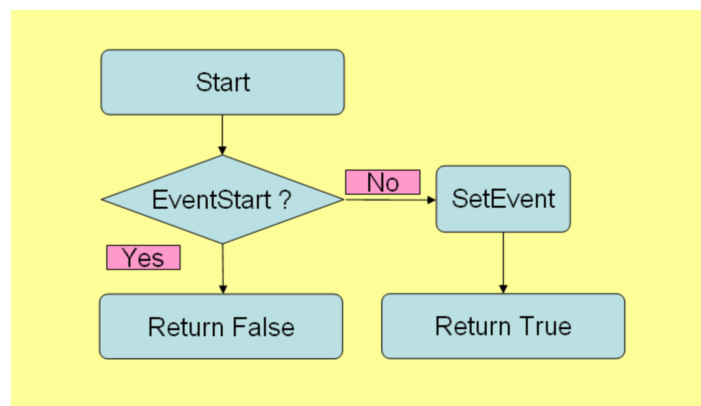
```
//DoInit
//This function call by Service when the servcice was be loaded.
//A developer could add his own initial procedures in it.
BOOL DoInit(int nID, DWORD dwParam, CBPublish procCBPublish, CBWriteLog
procCBWriteLog)
{
	if(nID<0)
		return FALSE;
	g_nMyID=nID;
//	g_nMySlot=dwParam;
	g_procCBPublish=procCBPublish;
	g_procCBWriteLog=procCBWriteLog;
	memset(g_arSlotInfo, 0, sizeof(SLOT_INFO)*MAX_SLOTS);

	for(int i=0; i< MAX_SLOTS;i++)
		g_arSlotInfo[i].fDiff = 0.0;

	g_hEventStart=CreateEvent(NULL, FALSE, FALSE, NULL);
	if(!g_hEventStart)
```

```
{
    g_procCBWriteLog(g_nMyID, "Module Init fail(0).");
    return FALSE;
}
g_hEventStop=CreateEvent(NULL, FALSE, FALSE, NULL);
if(!g_hEventStop)
{
    CloseHandle(g_hEventStart);
    g_procCBWriteLog(g_nMyID, "Module Init fail(1).");
    return FALSE;
}
g_hEventTerminate=CreateEvent(NULL, TRUE, FALSE, NULL);
if(!g_hEventTerminate)
{
    CloseHandle(g_hEventStart);
    CloseHandle(g_hEventStop);
    g_procCBWriteLog(g_nMyID, "Module Init fail(2).");
    return FALSE;
}
g_hMutexContainer=CreateMutex(NULL, FALSE, NULL);
if(!g_hMutexContainer)
{
    CloseHandle(g_hEventStart);
    CloseHandle(g_hEventStop);
    CloseHandle(g_hEventTerminate);
    g_procCBWriteLog(g_nMyID, "Module Init fail(2).");
    return FALSE;
}

g_hThread=CreateThread(NULL, 0, ThreadWorker, (LPVOID)NULL, 0, NULL);
if(!g_hThread)
{
    CloseHandle(g_hEventStart);
    CloseHandle(g_hEventStop);
    CloseHandle(g_hEventTerminate);
    CloseHandle(g_hMutexContainer);
    g_procCBWriteLog(g_nMyID, "Module Init fail(3).");
```

```
        return FALSE;
    }
    CloseHandle(g_hThread);

    // Add your initial procedures here
    unsigned char ret=StartAgent(3, 0);
    if(ret==0)
        g_procCBWriteLog(g_nMyID, "ScanKernal Agent start successfully.");
    else
        g_procCBWriteLog(g_nMyID, "ScanKernal Agent has been started.");
    Sleep(10000);
    g_procCBWriteLog(g_nMyID, "Module Initialized.");
    return TRUE;
}
```

DoStart 啟動

這一段主要是在執行 g_hEventStart 的檢查，若是 g_hEventStart 已經起始，則跳過起始動作。否則，執行 SetEvent 後寫入執行成功的訊息。



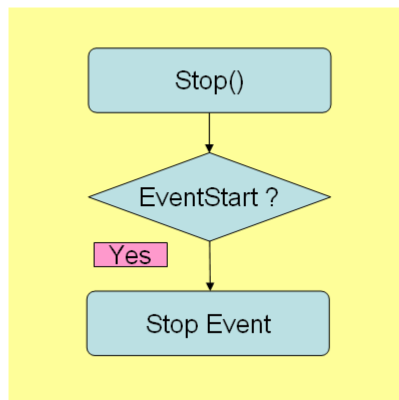
圖七、檢查是否已經啟動過了

```
BOOL DoStart()
{
    if(!g_hEventStart)
        return FALSE;
    SetEvent(g_hEventStart);
```

```
    g_procCBWriteLog(g_nMyID, "Module Started.");
    return TRUE;
}
```

DoStop 停止工作

檢查 g_hEventStop 是否為 TRUE，若是 TRUE 則執行 SetEventStop。

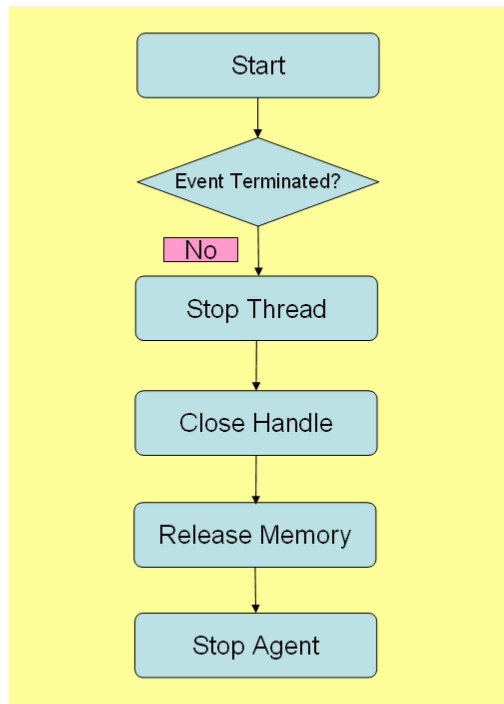


圖八、停止模組的流程

```
//DoStop
//call by Service when service was been stopped.
void DoStop()
{
    if(g_hEventStop)
        SetEvent(g_hEventStop);
    g_procCBWriteLog(g_nMyID, "Module Stopped.");
}
```

DoFinalize 結束執行

執行 g_hEventTerminate 的程序，並關閉 g_hEventStart、g_hEventStop、g_hEventTerimate、及 g_hMutexContainer 的 Handle，然後將 g_arSlotInfo 的 TagContainer 內容，一一刪除，然後停止程式的 Agent()。

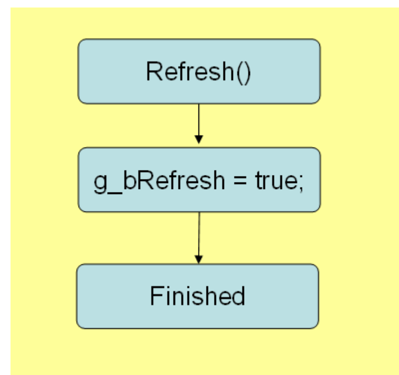


圖九、結束模組並釋放記憶體的程序

```
//DoFinalize
//call by Service when service was be unloaded.
void DoFinalize()
{
	if(g_hEventTerminate)
		SetEvent(g_hEventTerminate);
	WaitForSingleObject(g_hThread, 5000);
	CloseHandle(g_hEventStart);
	CloseHandle(g_hEventStop);
	CloseHandle(g_hEventTerminate);
	CloseHandle(g_hMutexContainer);
	for(int i=0; i<MAX_SLOTS; i++)
		if(g_arSlotInfo[i].lpTagContainer)
			delete g_arSlotInfo[i].lpTagContainer;
	StopAgent();
	g_procCBWriteLog(g_nMyID, "Module Terminated.");
}
```

DoReflesh 重新起始並更新狀態

DoReflesh 所做的是重新起始一個 TagContainer 指標 lpNewTagContainer，若成功重新建立一個 TagCobtainer，接著就從設定好的 CONFIG_FILEPATH 讀取相關的 XML 資訊及屬性。將目前的執行緒暫停後，然後將 g_lpTagContainer 指向新的 lpNewTagContainer，接著重新啟動模組執行緒，刪除舊的 lpOldTagContainer 物件，釋放出相對的空間。



圖九、更新狀態流程

```
//DoRefresh
//call by Service when service need to refresh.
void DoRefresh()
{
/*
	TagContainer *lpNewTagContainer=new TagContainer;
	if(!lpNewTagContainer)
	{
		g_procCBWriteLog(g_nMyID, "Module failed to Refresh.");
		return;
	}
	if(!lpNewTagContainer->LoadProfile(g_nMySlot, CONFIG_FILEPATH,
MyPropertyHandler, MyTagCreateHandler))
	{
		g_procCBWriteLog(g_nMyID, "Load profile FAIL.");
		return;
	}

	WaitForSingleObject(g_hMutexContainer, INFINITE);
```

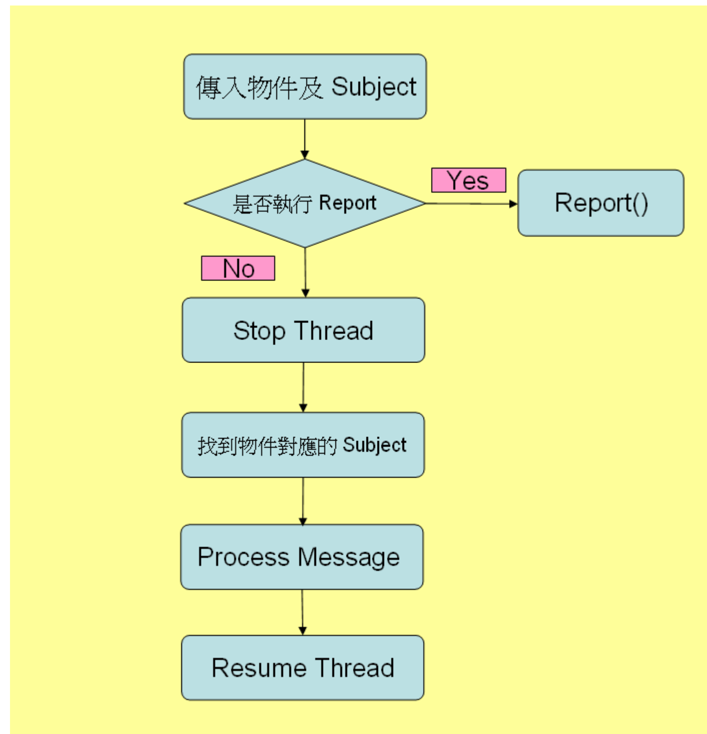
```
	TagContainer *lpOldContainer=g_lpTagContainer;
	g_lpTagContainer=lpNewTagContainer;
	ReleaseMutex(g_hMutexContainer);
	delete lpOldContainer;
*/
	g_bRefresh = true;
	g_procCBWriteLog(g_nMyID, "Module Refreshed.");
}
```

接收外來訊息的 DoProcessMsg

在這個函式中，我們主要會處理來自客戶端送來的訊息物件 (Message Object)。

這個函式會傳入兩個參數。一個是來源的 Subject 位址，另一個則是傳入的資料物件。在原始的 Module Kernel 版本中，我們會先檢查這個命令是不是 Report 命令，然後直接呼叫 DoReport () 函式，執行 DoReport () 函式的內容。

若不是執行 Report 命令，則會進行剩下的函式處理程序。先取得執行緒的優先權限後，接著執行判斷並進行 ProcessMessage 的動作，將資料解開並依照 Tag 的 Subject 路徑做進一步的處理。處理結束後，再呼叫 ReleaseMutex 復原原有執行緒的工作。



圖十、處理來自客戶端訊息的程序

```
//DoProcessMsg
//Call by Service when Service received a message and should be processed by this module.
//Any memory space in lpMsgObj will be released after this function returned, you should
// copy those data you need to process by other threads.

void DoProcessMsg(char *strTagPath, MessageObject *lpMsgObj)
{
    if(lpMsgObj->isPropertyExist("Report") && (!strTagPath || !strlen(strTagPath)))
    {
        DoReport();
    }

    int nIDLen;
    WaitForSingleObject(g_hMutexContainer, INFINITE);
    for(int i=1; i<MAX_SLOTS; i++)
    {
        if(g_arSlotInfo[i].lpTagContainer)
        {
            nIDLen=strlen(g_arSlotInfo[i].strPostfix);
```

```
                if(!strTagPath || !strlen(strTagPath))
                {
                    g_arSlotInfo[i].lpTagContainer->ProcessMessage(NULL, i, lpMsgObj);
                }
                else if(strncmp(g_arSlotInfo[i].strPostfix, strTagPath, nIDLen)==0)
                {
                    if(strTagPath[nIDLen]=='.')
                    {

    g_arSlotInfo[i].lpTagContainer->ProcessMessage(strTagPath+nIDLen+1, i, lpMsgObj);
                    }
                    else if(strTagPath[nIDLen]=='\0')
                    {
                        g_arSlotInfo[i].lpTagContainer->ProcessMessage(NULL, i, lpMsgObj);
                    }
                }
            }
        }
    ReleaseMutex(g_hMutexContainer);
}
```

建立 TagObject 物件的 MyTagCreateHandler 程序

這一部分的敘述主要依照 XML **檔案的設定，建立對應的** Digital **及** Analog Tag, **並進行設定名稱的工作後回傳** Tag。

```
TagObject* MyTagCreateHandler(const char **atts)
{
    if(g_nCurSlot < 0 || g_nCurSlot >= MAX_SLOTS)
        return NULL;

    char *strName=TagContainer::AttrGetValue(atts, "name");
    char *strType=TagContainer::AttrGetValue(atts, "type");
    char *strChannel=TagContainer::AttrGetValue(atts, "channel");
    if(strName && /*strType &&*/ strChannel)
    {
```

```
if(strcmp(strType, "DigitalOutput")==0)
{
	TagDigitalOutput *lpDigitalTag=new TagDigitalOutput;
	lpDigitalTag->SetName(strName);
	return lpDigitalTag;
}
else if(strcmp(strType, "BitOutput")==0)
{
	TagBitOutput *lpBitTag=new TagBitOutput(atoi(strChannel));
	lpBitTag->SetName(strName);
	return lpBitTag;
}
else if(strcmp(strType, "DigitalInput")==0)
{
	TagDigitalInput *lpDigitalTag=new TagDigitalInput;
	lpDigitalTag->SetName(strName);
	return lpDigitalTag;
}
else if(strcmp(strType, "BitInput")==0)
{
	TagBitInput *lpBitTag=new TagBitInput(atoi(strChannel));
	lpBitTag->SetName(strName);
	return lpBitTag;
}
else if(strcmp(strType, "AnalogInput")==0)
{
	TagAnalogInput *lpTag=new TagAnalogInput(strName,
	                                         atoi(strChannel),
	                                         g_arSlotInfo[g_nCurSlot].fDiff);
	return lpTag;
}
else if(strcmp(strType, "AnalogOutput")==0)
{
	TagAnalogOutput *lpTag=new TagAnalogOutput(strName, atoi(strChannel));
	return lpTag;
}
else
```

```
            {
                char strLog[512];
                sprintf(strLog, "Unknown Tag type=%s", strType);
                g_procCBWriteLog(g_nMyID, strLog);
            }
        }
        else
            g_procCBWriteLog(g_nMyID, "Error Tag profile data.");
        return NULL;
}
```

讀取設定屬性的 MyPropertyHandler 程序

MyPropertyHandler 會處理來自於 XML 檔案中的 <Property> 標籤的內容。然後針對屬性標籤的內容，做進一步的資料處理。在這一段程式碼中，我們會處理 XML 中的 Frequency 頻率標籤，以及變化敏感度的 Diff 標籤。

在 Frequency 標籤中，會針對讀到的 Frequency 標籤，以 100 ms 為單位處理後，最後將頻率的實際值設定給計數器 g_arSlotInfo[g_nCurSlot].nFreqCounter。

針對 [Diff] 標籤而言，KernelModule 會將得到的 Float 型態的設定值，設定給 g_arSlotInfo[g_nCurSlot].fDiff，然後在接下來的 CheckDevice() 以及 CheckState () 做判斷。

```
void MyPropertyHandler(char *strPropertyName, char *strPropertyValue)
{

    if(g_nCurSlot < 0 || g_nCurSlot >= MAX_SLOTS)
        return;

    if(strcmp(strPropertyName, "Frequency")==0)
    {
        //g_nCheckFrequency=atoi(strPropertyValue);
        g_arSlotInfo[g_nCurSlot].nFrequency = atoi(strPropertyValue);
        // Frequency
        int nRemain = g_arSlotInfo[g_nCurSlot].nFrequency % 100;
        int nMultiple = g_arSlotInfo[g_nCurSlot].nFrequency / 100;
```

```
        if(nRemain != 0)
            nMultiple += 1;
        g_arSlotInfo[g_nCurSlot].nFrequency = nMultiple * 100;
        g_arSlotInfo[g_nCurSlot].nFreqCounter = g_arSlotInfo[g_nCurSlot].nFrequency;

        char strLog[512];
        sprintf(strLog, "Change Slot%d Frequency property value=%d",
                g_nCurSlot,
                g_arSlotInfo[g_nCurSlot].nFrequency);
        g_procCBWriteLog(g_nCurSlot/*g_nMySlot*/, strLog);
    }
    else if(strcmp(strPropertyName, "Diff")==0)
    {
        //g_fDiff=(float)atof(strPropertyValue);
        g_arSlotInfo[g_nCurSlot].fDiff = (float)atof(strPropertyValue);
        char strLog[512];
        sprintf(strLog, "Change Slot%d Diff property value=%f",
                g_nCurSlot,
                g_arSlotInfo[g_nCurSlot].fDiff);
        g_procCBWriteLog(g_nCurSlot/*g_nMySlot*/, strLog);
    }
    /*
    else if(strcmp(strPropertyName, "DiffCurrent")==0)
    {
        g_fDiffCurrent=(float)atof(strPropertyValue);
        char strLog[512];
        sprintf(strLog, "Change DiffCurrent property value=%f", g_fDiffCurrent);
        g_procCBWriteLog(g_nMySlot, strLog);
    }
    */
}
```

檢查裝置存在狀態的 Check State() 函式

在這一整段程式碼中，最主要的是提供裝置存在與否的機制。可偵測 I/O 模組是否存在於

WinCon-8000 上。整段的程式邏輯如下：

依序檢查每一個插槽

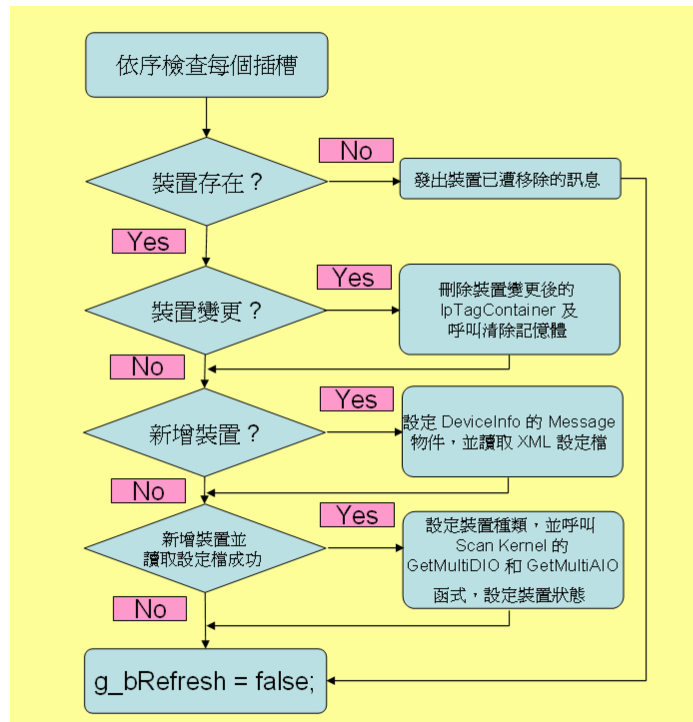
if (**裝置存在**)

  if(**裝置變更**){**刪除裝置變更後的** IpTagContainer **及呼叫清除記憶體**}

  if(**新增裝置**){**設定** DeviceInfo **的** Message **物件，並讀取** XML **設定檔**}

    if(**新增裝置並讀取設定檔成功**){**設定裝置種類，並呼叫** Scan Kernel **的** GetMultiDIO **和** GetMultiAIO **的函式，設定裝置狀態**}

else (**裝置不存在**){**發出裝置已遭移除的訊息**}



圖十一、Check Device **的流程圖**

```
//CheckDevice
//Thread Worker call this function each g_nCheckFrequency m.seconds.
//Do device check and publish any needed information here.
//A developer could modify this function if needed.
void CheckDevice()
{
```

```
char strName[80];
unsigned char iAtri[8];
for(int i=1; i<MAX_SLOTS; i++)
{
    if(GetNameOf8KModule(i, strName)==0 && strlen(strName))
    {//device exist
        if(strlen(g_arSlotInfo[i].strDeviceName) &&
            strcmp(strName, g_arSlotInfo[i].strDeviceName)!=0)
        {//device changed
            //publish DeviceRemoved message
            MessageObject MsgObj;
            MsgObj.setStringProperty("DeviceRemove",
g_arSlotInfo[i].strDeviceName);
            char strPostfix[224];
            sprintf(strPostfix, "%s_%d", strName, i);
            g_procCBPublish(g_nMyID, MESSAGE_TYPE_SUBJECT, strPostfix,
&MsgObj);
            delete g_arSlotInfo[i].lpTagContainer;
            memset(&g_arSlotInfo[i], 0, sizeof(SLOT_INFO));
            //g_arSlotInfo[i].fDiff = 0.0;
        }
        if(g_bRefresh || !strlen(g_arSlotInfo[i].strDeviceName))
        {//add new
            //publish DeviceInfo message
            MessageObject MsgObj;
            MsgObj.setStringProperty("DeviceInfo", strName);
            if(GetAtriOf8KModule(i, iAtri)==0)
            {
                MsgObj.setIntProperty("#DI", iAtri[0]);
                MsgObj.setIntProperty("#DO", iAtri[1]);
                MsgObj.setIntProperty("#AI", iAtri[2]);
                MsgObj.setIntProperty("#AO", iAtri[3]);
                MsgObj.setIntProperty("#Counter", iAtri[4]);
            }
            char strPostfix[224];
            sprintf(strPostfix, "%s_%d", strName, i);
```

```
            //arrange data
            strcpy(g_arSlotInfo[i].strDeviceName, strName);
            sprintf(g_arSlotInfo[i].strPostfix, "%s_%d", strName, i);

            if(g_arSlotInfo[i].lpTagContainer != NULL)
                delete g_arSlotInfo[i].lpTagContainer;
            g_arSlotInfo[i].lpTagContainer=new TagContainer;

            char strProfileName[256];
            sprintf(strProfileName, "%s%s.xml", CONFIG_FILEPATH, strName);

            if(g_arSlotInfo[i].lpTagContainer)
            {
                g_nCurSlot = i;
                if(!g_arSlotInfo[i].lpTagContainer->LoadProfile(i,
(char*)LPCTSTR(strProfileName), MyPropertyHandler, MyTagCreateHandler))
                {
                    char strLog[256];
                    sprintf(strLog, "Load profile %s FAIL.", strProfileName);
                    g_procCBWriteLog(g_nMyID, strLog);
                }
                else
                {//load OK
                    if(iAtri[0] && iAtri[1])//Digital Device
                    {

    g_arSlotInfo[i].nDeviceType=DEVICE_TYPE_DIGITAL_BOTH;
                        GetMultiDIO(i, g_arSlotInfo[i].ucDigitalState, 1);

    g_arSlotInfo[i].lpTagContainer->SetState((LPVOID)g_arSlotInfo[i].ucDigitalState);
                    }
                    else if(iAtri[0])
                    {

    g_arSlotInfo[i].nDeviceType=DEVICE_TYPE_DIGITAL_INPUT;
                        GetMultiDIO(i, g_arSlotInfo[i].ucDigitalState, 1);
```

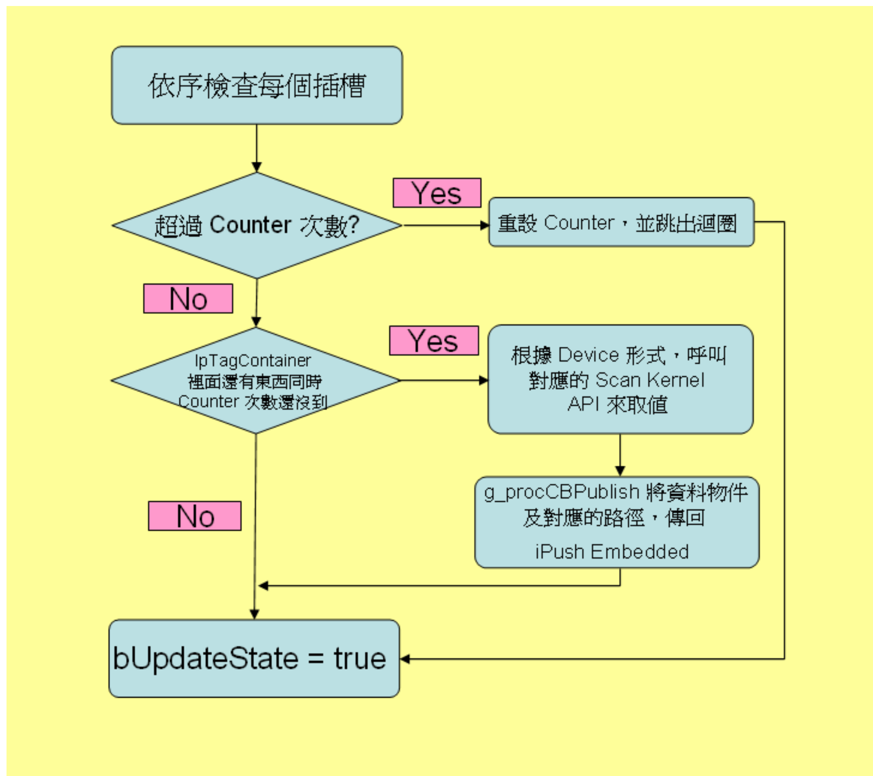
```
g_arSlotInfo[i].lpTagContainer->SetState((LPVOID)g_arSlotInfo[i].ucDigitalState);
                    }
                    else if(iAtri[1])
                    {
g_arSlotInfo[i].nDeviceType=DEVICE_TYPE_DIGITAL_OUTPUT;
                        GetMultiDIO(i, g_arSlotInfo[i].ucDigitalState, 1);
g_arSlotInfo[i].lpTagContainer->SetState((LPVOID)g_arSlotInfo[i].ucDigitalState);
                    }
                    else if(iAtri[2] && iAtri[3])//Analog Device
                    {
g_arSlotInfo[i].nDeviceType=DEVICE_TYPE_ANALOG_BOTH;
                        GetMultiAIO(i, g_arSlotInfo[i].fAnalogState, 1);
g_arSlotInfo[i].lpTagContainer->SetState((LPVOID)g_arSlotInfo[i].fAnalogState);
                    }
                    else if(iAtri[2])//Analog Device
                    {
g_arSlotInfo[i].nDeviceType=DEVICE_TYPE_ANALOG_INPUT;
                        GetMultiAIO(i, g_arSlotInfo[i].fAnalogState, 1);
g_arSlotInfo[i].lpTagContainer->SetState((LPVOID)g_arSlotInfo[i].fAnalogState);
                    }
                    else if(iAtri[3])//Analog Device
                    {
g_arSlotInfo[i].nDeviceType=DEVICE_TYPE_ANALOG_OUTPUT;
                        GetMultiAIO(i, g_arSlotInfo[i].fAnalogState, 0);
g_arSlotInfo[i].lpTagContainer->SetState((LPVOID)g_arSlotInfo[i].fAnalogState);
                    }
                }
            }
            else
```

```
                        g_procCBWriteLog(g_nMyID, "Module container allocate FAIL.");

                    g_procCBPublish(g_nMyID, MESSAGE_TYPE_SUBJECT, strPostfix,
&MsgObj);
                }
            }
            else
            {//device not exist
                if(strlen(g_arSlotInfo[i].strDeviceName))
                {//device removed
                    //publish DeviceRemoved message
                    MessageObject MsgObj;
                    MsgObj.setStringProperty("DeviceRemove",
g_arSlotInfo[i].strDeviceName);
                    char strPostfix[224];
                    sprintf(strPostfix, "%s_%d", strName, i);
                    g_procCBPublish(g_nMyID, MESSAGE_TYPE_SUBJECT, strPostfix,
&MsgObj);
                    delete g_arSlotInfo[i].lpTagContainer;
                    memset(&g_arSlotInfo[i], 0, sizeof(SLOT_INFO));
                }
            }
        }

        g_bRefresh = false;
}
```

Check State 檢查裝置的狀態

CheckState() 會檢查目前裝置的狀態，然後對讀取裝置的狀態，並執行更新的工作。在這邊我們使用一個 bUpdateState 旗標及 g_arSlotInfo[nSlot].nFreqCounter 來控制程式執行的狀態。首先若是 Counter 已經小於 0，代表已經 Update 過，接著就是恢復 Counter 的數值。否則便呼叫 Scan Kernel API 的 GetMultiDIO 及 GetMultiAIO 做裝置狀態的檢查工作。

最後呼叫 g_procCBPublish **將資料物件當參數傳給** iPush Embedded **的訊息核心。整個程式的設計流程如下：**



圖十二、Check State **的流程圖**

依序檢查每一個插槽

if(**檢查是否已經超過** Counter **的次數**){**重設** Counter, **並跳出迴圈**}

if(lpTagContainer **裡面還有東西，同時** Counter **次數還沒到**)
{**根據** Device **形式，呼叫對應的** Scan Kernel API **來取值**}}

呼叫 g_procCBPublish **將資料物件及對應的路徑，傳回** iPush Embedded**。遠端的使用者，可以經由使用客戶端的** iPush API **及解訊息的物件，解開由** IO Module **傳回的資訊。**

```
 void CheckState()
{
     for(int nSlot=1; nSlot<8; nSlot++)
     {
          bool bUpdateState = false;
```

```
g_arSlotInfo[nSlot].nFreqCounter -= 100;
if(g_arSlotInfo[nSlot].nFreqCounter <= 0)
{
	g_arSlotInfo[nSlot].nFreqCounter = g_arSlotInfo[nSlot].nFrequency;
	bUpdateState = true;
}

if(g_arSlotInfo[nSlot].lpTagContainer && bUpdateState)
{

	switch(g_arSlotInfo[nSlot].nDeviceType)
	{
	case DEVICE_TYPE_DIGITAL_INPUT:
		GetMultiDIO(nSlot, g_arSlotInfo[nSlot].ucDigitalState, 1);
		break;
	case DEVICE_TYPE_DIGITAL_OUTPUT:
		GetMultiDIO(nSlot, g_arSlotInfo[nSlot].ucDigitalState, 0);
		break;
	case DEVICE_TYPE_DIGITAL_BOTH:
		GetMultiDIO(nSlot, g_arSlotInfo[nSlot].ucDigitalState, 1);
		break;
	case DEVICE_TYPE_ANALOG_INPUT:
		GetMultiAIO(nSlot, g_arSlotInfo[nSlot].fAnalogState, 1);
		break;
	case DEVICE_TYPE_ANALOG_OUTPUT:
		GetMultiAIO(nSlot, g_arSlotInfo[nSlot].fAnalogState, 0);
		break;
	case DEVICE_TYPE_ANALOG_BOTH:
		GetMultiAIO(nS lot, g_arSlotInfo[nSlot].fAnalogState, 1);
		break;
	default:
		break;
	}
	char strPath[224];
	strcpy(strPath, g_arSlotInfo[nSlot].strPostfix);
	MessageObject MsgObj;
	while(g_arSlotInfo[nSlot].lpTagContainer->GetMessageObject(strPath,
```

```
&MsgObj))
                {
                    if(strlen(strPath))
                    {
                        char strPostfix[224];
                        sprintf(strPostfix, "%s.%s", g_arSlotInfo[nSlot].strPostfix, strPath);
                        g_procCBPublish(g_nMyID, MESSAGE_TYPE_SUBJECT, strPostfix,
&MsgObj);
                    }
                    else
                        g_procCBPublish(g_nMyID, MESSAGE_TYPE_SUBJECT,
g_arSlotInfo[nSlot].strPostfix, &MsgObj);
                    strPath[0]='\0';
                }
            }
        }
}
```