[百度分享](http://www.kaifa6.com/v/HuibianJiaocheng/tip-serial.htm)

|  |
| --- |
| **在 Win32 下 Serial Port 的通訊** |

以下的東西是我在閱讀 "Communications Programming for Windows 95" 時做的一點筆記, 所以在 topic 上大致上都和這書上的第三章一樣. 一些 structure 和 API 的宣告都是 copy from VC 5.0 的 on-line help.

這些東西主要談的是如何在 Win32 的平台下對 serial port 的通訊. 可能可以帶來幫助的是那些了解 serial port 的通訊, 但是不清楚在 Win32 平台究竟有那些 API 可用的 programer (就像我? :p) 我只是做做整理罷了, 更清楚的內容可能要再翻翻 on-line help 或是書本. 如果你要嘗試在 Win32 下做像是對 modem 做控制(例如撥號, 通訊)的程式, 但是對於 serial port 並不了解, 那你應該再去找一本有講到這些東西的書看看.

|  |
| --- |
| **開啟一個 Serial Port** |

利用一般開啟檔案的 CreatFile() 即可開啟 serial port device

HANDLE CreateFile(

LPCTSTR lpFileName, // pointer to name of the file

DWORD dwDesiredAccess, // access (read-write) mode

DWORD dwShareMode, // share mode

LPSECURITY\_ATTRIBUTES lpSecurityAttributes, // pointer to security attributes

DWORD dwCreationDistribution, // how to create

DWORD dwFlagsAndAttributes, // file attributes

HANDLE hTemplateFile // handle to file with attributes to copy

);

* 用 CreateFile() API.
* lpFileName 為 "COM1" 或是 "COM2"
* dwDersiredAccess 一般為 GENERIC\_READ|GENERIC\_WRITE
* dwShareMode "必須"為 0, 即不能共享, 但同一個 process 中的不同 thread 在一開啟之後就可以共享.
* lpSecurityAttributes 一般為 NULL
* dwCreateionDistributon 在這裡"必須"為 OPEN\_EXISTING
* dwFlagsAndAttributes 定義了開啟的屬性, 若是設成 FILE\_FLAG\_OVERLAPPED 則可使用非同步的 I/O.
* hTemplateFile "必須"為 NULL
* 傳回檔案 handle

|  |
| --- |
| **設定 Serial Port 傳送及接收緩衝區的大小** |

在開啟完 serial port 之後, 可以藉由呼叫 SetupComm() 來進行配置傳送時的緩衝區及接收時的緩衝區. 如果沒有呼叫 SetupComm() 的話, Win95 會配置內定的緩衝區.

BOOL SetupComm(

HANDLE hFile, // handle of communications device

DWORD dwInQueue, // size of input buffer

DWORD dwOutQueue // size of output buffer

);

|  |
| --- |
| **關閉 Serial Port file** |

利用一般的 CloseHandle() 即可.

BOOL CloseHandle(

HANDLE hObject // handle to object to close

)

|  |
| --- |
| **取得 Seial Port 的資訊** |

在 Win32 裡頭, 將一些通訊時會用到的資訊用 COMMPROP 這個結構來表示. (當然不僅僅是 Serial Port) 可以用 GetCommProperties() 來取得:

BOOL GetCommProperties(

HANDLE hFile, // handle of communications device

LPCOMMPROP lpCommProp // address of communications properties structure

);

COMMPROP 長的像這個樣子:

typedef struct \_COMMPROP { // cmmp

WORD wPacketLength; // packet size, in bytes

WORD wPacketVersion; // packet version

DWORD dwServiceMask; // services implemented

DWORD dwReserved1; // reserved

DWORD dwMaxTxQueue; // max Tx bufsize, in bytes

DWORD dwMaxRxQueue; // max Rx bufsize, in bytes

DWORD dwMaxBaud; // max baud rate, in bps

DWORD dwProvSubType; // specific provider type

DWORD dwProvCapabilities; // capabilities supported

DWORD dwSettableParams; // changable parameters

DWORD dwSettableBaud; // allowable baud rates

WORD wSettableData; // allowable byte sizes

WORD wSettableStopParity; // stop bits/parity allowed

DWORD dwCurrentTxQueue; // Tx buffer size, in bytes

DWORD dwCurrentRxQueue; // Rx buffer size, in bytes

DWORD dwProvSpec1; // provider-specific data

DWORD dwProvSpec2; // provider-specific data

WCHAR wcProvChar[1]; // provider-specific data

} COMMPROP;

在這裡, lpCommProp 需要 programmer 自行配置空間. 有趣的問題是, 系統在這個結構之後會需要額外的空間. 但是配置者也就是 programmer 卻不知道系統會需要多少. 很簡單的做法是配置一大塊絕對會夠的空間. 另一個聰明的做法是執行兩次 GetCommProperties() , 第一次只配置 sizeof(COMMPROP) 這麼大的空間, 因為還沒有開始執行一些動作, 所以系統並不會嘗試著在後面填東西, 所以不會出問題. 接著執行第一次的 GetCommProperties(), 得到結果, 取出結構中的 wPacketLength, 這個 member 代表實際上需要的大小, 然後依據這個大小重新配置一個新的. 這樣的話 , 就不會有浪費任何空間的問題了.

至於上述 COMMPROP 結構的成員所代表的意思, on-line help 中應該寫的都滿清楚的 .

|  |
| --- |
| **設定及取得通訊狀態** |

你可以利用 COMMPROP 來取得一些狀態, 但是當你想改變目前的設定時你需要兩個 API 來完成:

BOOL GetCommState(

HANDLE hFile, // handle of communications device

LPDCB lpDCB // address of device-control block structure

);

BOOL SetCommState(

HANDLE hFile, // handle of communications device

LPDCB lpDCB // address of device-control block structure

);

你可以用 GetCommState() 來取得目前 Serial Port 的狀態, 也可以用 SetCommState() 來設定 Serial Port 的狀態.

DCB 的結構就請自行翻閱 help 囉.

另外, programmer 最常控制的幾個設定就是 baud rate, parity method, data bits, 還有 stop bit. BuildCommDCB() 提供了對於這幾個常見設定的控制.

BOOL BuildCommDCB(

LPCTSTR lpDef, // pointer to device-control string

LPDCB lpDCB // pointer to device-control block

);

lpDef 長的像這樣: "baud=2400 parity=N data=8 stop=1"

|  |
| --- |
| **通訊設定對話盒** |

Win32 API 中提供了一個開啟通訊設定對話盒的 API: CommConfigDialog(), 當呼叫這個 API 時, 會蹦現一個可供設定 Baud Rate, Data Bits, Parity .. 等資訊的對話盒, programmer 可以利用它來讓使用者設定一些資訊, 並且取得結果.

BOOL CommConfigDialog(

LPTSTR lpszName, // pointer to device name string

HWND hWnd, // handle to window

LPCOMMCONFIG lpCC // pointer to comm. configuration structure

);

其中 lpCC 被用來存放設定值的結果.

typedef struct \_COMM\_CONFIG {

DWORD dwSize;

WORD wVersion;

WORD wReserved;

DCB dcb;

DWORD dwProviderSubType;

DWORD dwProviderOffset;

DWORD dwProviderSize;

WCHAR wcProviderData[1];

} COMMCONFIG, \*LPCOMMCONFIG;

在我們呼叫 CommConfigDialog() 之前, dwSize 要設為 sizeof(COMMCONFIG), wVersion 的值在這邊似乎不重要(我不清楚, VC5 的 on-line help 說可以設為 1, 我手中的 book 的範例是設為 0x100), 呼叫完 CommConfigDialog() 之後, 成員 dcb 中的 BaudRate, ByteSize, StopBits, Parity 就是使用者的設定.

|  |
| --- |
| **Timeout 的機制** |

因為傳輸時並不會維持一個絕對穩定的速率. 因為傳輸品質的關係, programer 會需要 timeout 的機制來協助他們做一些控制. 在 Win32 通訊 Timeout 的機制中, timeout 的性質共分為兩類, 先來看看 COMMTIMEOUTS 這個結構:

typedef struct \_COMMTIMEOUTS { // ctmo

DWORD ReadIntervalTimeout;

DWORD ReadTotalTimeoutMultiplier;

DWORD ReadTotalTimeoutConstant;

DWORD WriteTotalTimeoutMultiplier;

DWORD WriteTotalTimeoutConstant;

} COMMTIMEOUTS,\*LPCOMMTIMEOUTS;

programmer 可以利用 GetCommTimeouts() 和 SetCommTimeouts() 來讀取或是設定目前的 timeout 值.

BOOL GetCommTimeouts(

HANDLE hFile, // handle of communications device

LPCOMMTIMEOUTS lpCommTimeouts // address of comm. time-outs structure

);

BOOL SetCommTimeouts(

HANDLE hFile, // handle of communications device

LPCOMMTIMEOUTS lpCommTimeouts // address of communications time-out structure

);

第一種 timeout 的機制稱為 interval timeout, 從字面上的意義很容易可以理解這種 timeout 的機制是讀取字元之間的間隔時間的 timeout, 只有讀取字元時才能夠使用interval timeout. 也就是在這個結構中的 ReadIntervalTimeout, 單位為 ms, 當讀取完一個字元後, 超過了 ReadIntervalTimeout 的值, 卻還沒有讀到下一個字元時, timeout 就發生了.

第二種 timeout 的機制稱為 total timeout, 顧名思義即是傳輸的總時間的 timeout . 在這種 timeout 的機制下, Win32 提供了一個具有彈性的方式來設定 total timeout. 以讀取的 total timeout 為例, 利用 ReadTotalTimeoutMultiplier 和 ReadTotalTimeoutConstant 構成了一個線性的上限值. 什麼意思呢? 實際上的 total timeout 應該是這樣的一個式子:

ReadTotalTimeout = ReadTotalTimeOutMultiplier\*BytesToRead+ReadTotalTimeoutConstant

WriteTotalTimeout 用同樣的公式來計算. 這樣的話, 不僅可以用一個固定的值來做為 timeout 值, 也可以用條線來做為 timeout 的值, 而隨著要讀取或是要寫的 bytes 數而變動.

如果不想使用 timeout, 就把 COMMTIMEOUTS 裡頭的資料成員都填為 0.

如果你將 ReadIntervalTimeout 設為 MAXDWORD, 且將 ReadTotalTimeOutMultiplier 和 ReadTotalTimeoutConstant 都設為 0 的話, 那麼讀取時, 如果 receive queue 裡頭並沒有資料, 讀取的動作將會馬上返回, 而不會停滯在讀取的動作.

這裡有一個和 BuildCommDCB() 很像的 API 叫 BuildCommDCBAndTimeouts():

BOOL BuildCommDCBAndTimeouts(

LPCTSTR lpDef, // pointer to the device-control string

LPDCB lpDCB, // pointer to the device-control block

LPCOMMTIMEOUTS lpCommTimeouts // pointer to comm. time-out structure

);

lpDef 一樣是控制字串, 可以給像 BuildCommDCB() 中的 lpDef 那樣格式的字串, 但是多了 "TO=XXX" 這個設定. 如果 "TO=ON", 這個 API 會依據 lpCommTimeouts 裡頭的值來設定讀和寫的 timeout 值. 如果 "TO=OFF", 則會設定這個 device 沒有 timeout. 如果是 "ON" 和 "OFF" 之外的其它值, 則 lpCommTimeouts 的設定將會被忽略.

對了, 在設定完 timeout 值之後, 記得要檢查 COMMPROP 裡的 dwProvCapabilities 中的 PCF\_INTTIMEOUTS 和 PCF\_TOTALTIMEOUTS 兩個 flags 是否有被 set, 以確認 interval timeout 和 total timeout 是否有支援.

|  |
| --- |
| **讀取資料** |

從 serial port 裡頭讀取資料就跟讀取一般的檔案一樣, 使用 ReadFile() 來達成.

BOOL ReadFile(

HANDLE hFile, // handle of file to read

LPVOID lpBuffer, // address of buffer that receives data

DWORD nNumberOfBytesToRead, // number of bytes to read

LPDWORD lpNumberOfBytesRead, // address of number of bytes read

LPOVERLAPPED lpOverlapped // address of structure for data

);

要注意的是, nNumberOfBytesToRead 設定的是一次最多的讀取量, 很有可能所讀取的值(檢查 lpNumberOfBytesRead)小於這個值. 通常在錯誤發生或是 timeout 發生時這個 API 就會返回.

PurgeComm() 這個 API 可以用來終止目前正在進行的讀或寫的動作, 也可以 flush 掉 I/O buffer 內等待讀或寫的資料.

BOOL PurgeComm(

HANDLE hFile, // handle of communications resource

DWORD dwFlags // action to perform

);

其中 dwFlags 共有四種 flags:

* PURGE\_TXABORT: 終止目前正在進行的(背景)寫入動作
* PURGE\_RXABORT: 終正目前正在進行的(背景)讀取動作
* PURGE\_TXCLEAR: flush 寫入的 buffer
* PURGE\_TXCLEAR: flush 讀取的 buffer

而使用 FlushFileBuffers() 可以確保所有的資料都被送出, 這個 API 才會返回.

另外一個有趣的 API 是 ClearCommError(), 從字面上的意思看來, 它是用來清除錯誤情況用的, 但是實際上它還可以拿來取得目前通訊設備的一些資訊.

BOOL ClearCommError(

HANDLE hFile, // handle to communications device

LPDWORD lpErrors, // pointer to variable to receive error codes

LPCOMSTAT lpStat // pointer to buffer for communications status

);

呼叫這個 API 之後, 關於通訊設備的一些資訊會被儲存在 lpStat 中, COMSTAT 的結構如下:

typedef struct \_COMSTAT { // cst

DWORD fCtsHold : 1; // Tx waiting for CTS signal

DWORD fDsrHold : 1; // Tx waiting for DSR signal

DWORD fRlsdHold : 1; // Tx waiting for RLSD signal

DWORD fXoffHold : 1; // Tx waiting, XOFF char rec'd

DWORD fXoffSent : 1; // Tx waiting, XOFF char sent

DWORD fEof : 1; // EOF character sent

DWORD fTxim : 1; // character waiting for Tx

DWORD fReserved : 25; // reserved

DWORD cbInQue; // bytes in input buffer

DWORD cbOutQue; // bytes in output buffer

} COMSTAT, \*LPCOMSTAT

藉由 fCtsHold, fDsrHold, fRlsdHold, fXoffHold, fXoffSent 可以知道目前因為什麼因素而使通訊阻礙住了.( 跟 handshaking 和 flow control 有關) cbInque 和 cbOutQue 則可以顯示出還有多少 bytes 在讀取或是寫入 queue 中.

|  |
| --- |
| **寫入資料** |

和讀取資料一樣, programmer 可以使用 WriteFile() 來將資料寫入 serial port.

BOOL WriteFile(

HANDLE hFile, // handle to file to write to

LPCVOID lpBuffer, // pointer to data to write to file

DWORD nNumberOfBytesToWrite, // number of bytes to write

LPDWORD lpNumberOfBytesWritten, // pointer to number of bytes written

LPOVERLAPPED lpOverlapped // pointer to structure needed for overlapped I/O

);

關於通訊設備的寫入有三個很有趣的 API, 它們分別是 SetCommBreak(), ClearCommBreak, 和 TransmitCommChar().

BOOL SetCommBreak(

HANDLE hFile // handle of communications device

);

BOOL ClearCommBreak(

HANDLE hFile // handle to communications device

);

BOOL TransmitCommChar(

HANDLE hFile, // handle of communications device

char cChar // character to transmit

);

SetCommBreak() 是用來暫停目前的傳輸作業, 它會使 buffer 中的資料都不再被送出, 這個時候, program 可以去做些雜七雜八的事, 之後, 再利用 ClearCommBreak() 回復傳輸作業.

TransmitCommChar() 是用來立即性的趕在所有 buffer 資料被送出去之前, 傳輸一個字元的資料出去, 即使 buffer 中還有資料. 換句話說, 這個字元有最高的優先權被送出去.

|  |
| --- |
| **事件驅動式的 I/O** |

在 Win32 裡頭, 對於通訊設備的 I/O 可以用像是事件驅動式的方法來達成. 主要是利用一個叫 WaitCommEvent() 的 API. 呼叫這個 API 之後, 會一直 block 到設定的事件發生之後才會返回. 我們先來看看如何設定事件, 再回過頭來看 WaitCommEvent() .

programer 可以用 GetCommMask() 和 SetCommMask() 來取得或是設定目前設定的通訊事件.

BOOL GetCommMask(

HANDLE hFile, // handle of communications device

LPDWORD lpEvtMask // address of variable to get event mask

);

BOOL SetCommMask(

HANDLE hFile, // handle of communications device

DWORD dwEvtMask // mask that identifies enabled events

);

可以設定的事件有 EV\_BREAK, EV\_CTS, EV\_DSR, EV\_ERR, EV\_RING, EV\_RLSD, EV\_RXCHAR, EV\_RXFLAG, EV\_TXEMPTY.(其意義請自行參考 help), 當然, 你可以把它們 or 起來成為組合的事件.

在設定完想要處理的事件之後, 可以使用 WaitCommEvent()

BOOL WaitCommEvent(

HANDLE hFile, // handle of communications device

LPDWORD lpEvtMask, // address of variable for event that occurred

LPOVERLAPPED lpOverlapped, // address of overlapped structure

);

WaitCommEvent() 會一直 block 到你所設定的通訊事件發生為止. 所以當 WaitCommEvent() 返回時, 你可以由 lpEvtMask 取得究竟是那一事件發生, 再來決定要如何處理.

舉例來說, 可以用 SetCommMask() 設定事件為 EV\_RXCHAR, 那麼在呼叫 WaitCommEvent() 時, 它會等到有字元可供讀取時才會返回, 那麼在它返回之後, 可以檢查一下 lpEvtMask 中是否 set 了 EV\_RXCHAR, 如果是的話就可以用 ReadFile() 去讀取. 這樣的話, 可以避免掉某些情形之下, 需要做 polling 所引起效率不彰的問題.

|  |
| --- |
| **錯誤的處理** |

前面提過的 ClearnCommError() 可以用來取得目前發生錯誤的原因.(請參見 help)

|  |
| --- |
| **硬體的控制命令** |

Win32 中提供了 EscapeCommFunction() 允許 programer 對幾個硬體訊號做控制.

BOOL EscapeCommFunction(

HANDLE hFile, // handle to communications device

DWORD dwFunc // extended function to perform

);

其中 dwFunc 可以是:

* CLRDTR : 讓 DTR OFF
* CLRRTS : 讓 RTS OFF
* SETDTR : 讓 DTR ON
* SETRTS : 讓 RTS ON
* SETXOFF : "模擬" 接收到 XOFF 字元
* SETXON : "模擬" 接收到 XON 字元
* SETBREAK : 和 SetCommBreak() 的意思相同
* CLRBREAK : 和 ClearCommBreak() 的意思相同

|  |
| --- |
| **Reference** |

[1]Charles A. Mirho and Andre Terrisse, "Communications Programming for Windows 95" , Microsoft Press

[站长统计](http://www.cnzz.com/stat/website.php?web_id=1126421)