직렬 통신 컴포넌트의 제작

(Creating Serial Communication Components)

이번 장에서는 윈도우 API 함수를 이용해서 시리얼 통신을 제어할 수 있는 방법을 소개하고, 이를 바탕으로 간단한 시리얼 통신 컴포넌트를 제작하도록 할 것이다.
시리얼 통신과 관련해서는 여러가지 프리웨어 컴포넌트를 인터넷에서 찾을 수 있으므로, 자신에게 맞는 컴포넌트를 찾아서 사용하는 것도 하나의 방법이 될 것이다. 그러나, 기본적으로 직렬 통신 포트에 접근해서 이를 사용하는 방법은 익혀두는 것이 좋다.

시리얼 통신의 기초

시리얼 포트는 데이터를 보내고 받는 두가지 일을 한다. 이것이 대단히 간단하게 생각되겠지만, 실제로는 이를 위해서 여러가지 일들이 벌어진다. 시리얼 포트는 컴퓨터의 연산 속도에 비해 훨씬 느리게 동작하기 때문에, 파일 등을 시리얼 포트로 보낸다고 생각할 때 적절한 버퍼의 활용과 흐름 제어(flow control)가 필수적이다.
이러한 흐름 제어의 방법으로 가장 흔히 쓰이는 것이 RTS/CTS 와 XON/XOFF 제어이다.
RTS(request to send)와 CTS(clear to send)는 시리얼 포트의 하드웨어적인 흐름 제어 방법이다. 포트의 RTS 라인은 원격 디바이스의 CTS 라인과 연결되어 있고, CTS 라인은 RTS 라인과 연결되어 있다. 원격 디바이스가 데이터를 받을 준비가 되면 RTS 라인을 활성화 시키며, 이렇게 되면 이와 연결된 CTS 라인에서 데이터를 보내기 시작한다. 원격 디바이스가 충분한 양의 데이터를 받으면 RTS 라인을 끊게 되고, 이것이 데이터를 보내지 말라는 신호가 된다. 이러한 사이클이 데이터가 모두 전송될 때까지 반복된다.
XON/XOFF 방법은 소프트웨어적인 흐름 제어 방법으로 XON/XOFF 문자를 보내서 제어하는 방법이다. 일단 시리얼 포트에서 데이터를 전송하기 시작해서 버퍼가 차게 되면 XOFF 문자(ASCII 13h)를 전송한다. 이 문자를 받으면 시리얼 포트는 데이터의 전송을 일단 중단한다. 데이터를 계속 받아서 버퍼에 여유가 생기면 이번에는 XON 문자(ASCII 11h)를 전송해서 시리얼 포트가 데이터 전송을 계속하게 된다.
이 두가지 방법은 서로 다른 방식으로 결국 비슷한 역할을 하게 되는데, 동시에 두가지 모두를 사용할 수도 있다.
일단 데이터가 전송되는 도중에는 보내고, 받은 데이터가 제대로 전송했는지 검사하는 것도 중요하다. 데이터의 전송이 제대로 이루어 졌는지 검사하는 방법에는 여러가지가 있는데, 그중 가장 원시적인 것인 패리티(parity) 에러를 검사하는 것이다. 패리티 에러 검사의 방법에는 even, odd, mark, space 등의 것이 있는데, 실용성이 많이 떨어지기 때문에 거의 사용되지 않는다.

이러한 패리티 검사 방법 외에 실제로 사용되는 것으로 CRC(cyclic redundancy check) 검사를 많이 한다. 이 방법은 전송된 데이터의 순서와 양을 체크 섬을 이용해서 검사하는 것으로 비교적 효과적이면서도 정확하다.
흐름 제어 만큼이나 중요한 것이 연결된 두 시리얼 포트가 서로의 상태를 알아보고 이를 제어 하는 방법이다. 여기에 대한 방법에도 하드웨어적인 방법과 소프트웨어적인 방법이 있다. 하드웨어적인 방법은 DSR(data set ready)/DTR(data terminal ready) 라인을 이용하는 것이다. 예를 들어 두 개의 모뎀이 연결되면 각각의 모뎀은 DTR 라인을 활성화 시킨다(이를 'hi' 또는 'hot' 이라고도 한다.). 한 모뎀의 DTR 라인은 다른 모뎀의 DSR 라인과 연결되어 있는데, 양쪽 모뎀이 다른 모뎀으로부터 신호를 받게 되면 바로 하드웨어적인 핸드 쉐이킹을 하게 된다. 그렇지만 이 방법은 비교적 오래된 방법으로, 최근에는 소프트웨어적인 방법이 더욱 효과적이고 더 많이 쓰인다. DSR/DTR 방법은 흐름 제어의 한 수단으로 사용되고 있다.
소프트웨어적인 방법은 모뎀이 연결될 때 잡음이 많이 섞이게 되는데 흐름 제어, 압축 레벨, 보드 전송율 등을 조절해서 여기에 대처하게 된다. 간단한 시리얼 통신에서는 이러한 방법이 사용되지 않지만, 파일 전송 등의 비교적 복잡한 작업을 할 때에는 필수적으로 사용된다.
Win32 API 에는 이러한 시리얼 통신을 지원하기 위해 비교적 많은 함수가 준비되어 있다. 지금 부터 이들을 하나씩 알아보고, 이를 쉽게 사용할 수 있는 컴포넌트를 하나 만들어 보자.

포트 열기

가장 중요한 것이 처음으로 포트를 여는 것이다. 이를 위해서 CreateFile 함수가 사용된다. 이 함수의 선언부는 다음과 같다.

```
function CreateFile(lpFileName: PChar; dwDesiredAccess, dwShareMode: Integer;
  lpSecurityAttributes: PSecurityAttributes; dwCreationDisposition,
  dwFlagsAndAttributes: DWORD; hTemplateFile: THandle): THandle;
```

이 함수가 성공적으로 수행되면 시리얼 포트에 대한 핸들을 돌려주게 된다. 첫번째 파라미터로 사용되는 lpFileName 에는 사용할 포트의 이름을 지정한다, 예를 들어 'COM1', 'COM2' 등을 지정한다. dwDesiredAccess 파라미터에는 포트에 접근하는 방법을 지정하는데, 포트를 통해서 데이터를 읽고, 쓰기를 모두 한다면 'GENERIC_READ OR GENERIC_WRITE'로 설정한다. dwShareMode 파라미터에는 지정한 포트를 다른 어플리케이션과 공유할 것인지 여부를 정한다. 보통의 경우에는 0 으로 지정하여 다른 어플리케이션이 접근할 수 없도록 한다.
lpSecurityAttibutes 파라미터는 핸들에 대한 보안 레벨을 지정하는 구조체의 포인터를 넘

겨주게 되는데, 보통의 경우 nil 로 설정한다. dwCreationDisposition 파라미터는 윈도우가 파일을 열거나 생성하는 방법을 지정하는데, 시리얼 통신의 경우에는 파일이 언제나 지정되어 있고, 존재하는 상태이므로 'OPEN_EXISTING'으로 설정된다.
dwFlagsAndAttributes 파라미터는 시리얼 포트가 통신하는 방법을 지정할 수 있는데, 예를 들어 FILE_FLAG_OVERLAPPED 로 지정된 경우 시리얼 포트가 동시에 읽기, 쓰기가 가능한 비동기(asynchronous) 통신을 지원한다. 보통의 경우에는 0 으로 지정해서 동기(synchronous) 통신을 지정한다. 마지막으로 hTemplateFile 파라미터는 시리얼 통신과 아무런 관계가 없기 때문에 보통 0 으로 설정한다.

장치 제어 블록 (Device Control Block, DCB)

시리얼 포트를 제어하는데 가장 중요한 데이터 구조가 장치 제어 블록(DCB) 구조체이다. 이 구조체에는 시리얼 포트에 적용할 모든 설정 사항이 저장된다. Windows.pas 유닛에 선언되어 있는 DCB 구조체의 선언부는 다음과 같다.

```
type
  TDCB = record
    DCBLength: DWORD;              //DCB 구조체의 크기
    BaudRate: DWORD;               //Baud rate
    Flags: LongInt;          //비트 플래그
    wReserved: Word;
    XONLim: Word;            //XON 스위치에 대한 바이트 한계
    XOFFLim: Word;                 //XOFF 스위치에 대한 바이트 한계
    ByteSize: Byte;                //바이트의 비트 수
    Parity: Byte;                  //패리티 종류
    StopBits: Byte;                //스톱 비트
    XONChar: Char;                 //XON 문자
    XOFFChar: Char;                //XOFF 문자
    ErrorChar: Char;               //패리티 에러 대체 문자
    EOFChar: Char;           //EOF 문자
    EvtChar: Char;                 //이벤트 문자
    wReserved1: Word;
  end;
```

DCB 파라미터를 얻어오고, 설정하는데 사용되는 함수가 GetCommState, SetCommState 함수이다. 이들 함수에 포트에 대한 핸들을 넘겨 주면 DCB 구조체의 주소를 얻을 수 있다.

이들 함수의 선언부는 다음과 같다.

```
function GetCommState(hFile: THandle; var lpDCB: TDCB): BOOL; stdcall;
function SetCommState(hFile: THandle; const lpDCB: TDCB): BOOL; stdcall;
```

GetCommTimeouts, SetCommTimeouts 함수

데이터를 얻어올 때 몇 가지 문제로 데이터가 전송되지 않을 수가 있다. 예를 들어, 불의의 사고로 선로가 끊어진 경우 시리얼 포트는 데이터를 전송받을 수가 없다. 이때 적절한 시간이 넘게 지나가면 더이상 데이터를 기다리지 않는다. 이러한 시간을 읽거나 설정할 때에 GetCommTimeouts, SetCommTimeouts 함수를 사용한다. 이 함수에 포트의 핸들과 TCommTimeouts 데이터 형의 레코드를 전달하면 제한 시간이 설정된다.
이들의 선언부는 다음과 같다.

```
function GetCommTimeouts(hFile: THandle; var lpCommTimeouts: TCommTimeouts):
  BOOL; stdcall;
function SetCommTimeouts(hFile: THandle; const lpCommTimeouts: TCommTimeouts):
  BOOL; stdcall;

TCommTimeouts = record
  ReadIntervalTimeout: DWORD;
  ReadTotalTimeoutMultiplier: DWORD;
  ReadTotalTimeoutConstant: DWORD;
  WriteTotalTimeoutMultiplier: DWORD;
  WriteTotalTimeoutConstant: DWORD;
end;
```

보통의 경우에는 마이크로소프트의 디폴트 값을 그대로 사용한다.

PurgeComm, CloseHandle, ClearCommError 함수

PurgeComm 함수를 이용하면 읽기, 쓰기를 진행하거나 취소할 수 있다. 또한 입력, 출력 버퍼를 비우게 할 수도 있다. 선언부는 다음과 같다.

```
function PurgeComm(hFile: THandle; dwFlags: DWORD): BOOL; stdcall;
```

첫번째 파라미터에는 포트의 핸들을 지정하면 되고, 두번째 파라미터에 플래그를 설정한다. 플래그로 사용할 수 있는 것으로 PURGE_TXABORT, PURGE_TXCLEAR, PURGE_RXABORT, PURGE_RXCLEAR 등이 있다. 여기서 'abort'는 진행 중인 작업을 즉시 중지하라는 의미이며, 'clear'는 해당 버퍼를 비우라는 의미이다. TX 와 RX 는 각각 'transfer', 'receive'를 의미한다.

이렇게 사용한 포트를 닫을 때에는 CloseHandle 함수를 사용한다. 이 함수가 성공적으로 수행되면 True 가 반환된다. 선언부는 다음과 같다.

```
function CloseHandle(hFile: THandle): BOOL; stdcall;
```

ClearCommError 함수는 지정된 장치의 현재 상태를 검사해서 에러가 있으면 이를 리포트한다. 또한, 통신 에러가 발생하면 호출되어 장치의 에러 플래그를 지우고 데이터의 읽기, 쓰기 작업을 계속하게 한다. 이 함수의 lpStat 파라미터는 TCommStat 구조체의 포인터로, 이 구조체에 발생한 에러의 내용과 현재 장치의 상태에 대한 내용이 담기게 된다. 함수와 TCommStat 구조체의 선언부는 다음과 같다.

```
function ClearCommError(hFile: THandle; var lpErrors: DWORD; lpStat: PComStat):
            BOOL; stdcall;
TComStat = record
  Flags: TCommStateFlags;
  Reserved: array[0..2] of Byte;
  cbInQue: DWORD;
  cbOutQue: DWORD;
end;
```

데이터 읽기와 쓰기 (ReadFile and WriteFile)

데이터를 읽고 쓰는데 가장 중요한 함수는 ReadFile, WriteFile 함수이다. 이 함수들은 시리얼 포트에서 실제 데이터를 일고 쓸 때 사용된다. 선언부는 다음과 같다.

```
function WriteFile(hFile: THandle; const Buffer: nNumberOfBytesToWrite: DWORD;
  var lpNumberOfBytesWritten: DWORD; lpOverlapped: POverlapped): BOOL; stdcall;
function ReadFile(hFile: THandle; var Buffer: nNumberOfBytesToRead: DWORD;
  var lpNumberOfBytesRead: DWORD; lpOverlapped: POverlapped): BOOL; stdcall;
```

기본적으로 파라미터가 의미하는 부분은 거의 비슷하다. hFile 파라미터는 사용할 포트의 핸들이고, Buffer 는 데이터를 보내거나 읽어 와야 되는 버퍼의 포인터를 가리킨다. nNumberOfBytesWritten, nNumberOfBytesRead 파라미터는 실제로 포트에 전송되거나 받게 되는 데이터의 바이트 수를 나타낸다. 그리고, lpNumberOfBytesWritten, lpNumberOfBytesRead 파라미터는 실제로 읽고 쓴 바이트 수를 나타낸다.
마지막으로 lpOverlapped 파라미터는 시리얼 포트를 비동기로 사용할 때 사용되는 TOverlapped 구조체에 대한 포인터이다.

TSerialPort 컴포넌트

그러면, 이러한 지식을 바탕으로 시리얼 통신을 쉽게 사용할 수 있는 컴포넌트를 하나 구현해 보자. 다음부터 설명하는 컴포넌트는 Jason “Wedge” Perry 가 공개한 TSerialPort 컴포넌트에 기초한 것임을 미리 밝혀 둔다.
그러면 이 컴포넌트에서 사용할 프로퍼티를 미리 정의하도록 한다. 사실 시리얼 포트에 대한 프로퍼티로 정의할 만한 것들은 대부분 DCB 구조체의 내용이다. 비교적 중요한 프로퍼티를 나열하면 아래와 같다.

- COM 포트 (COM1 ~ COM8)
- Baud Rate (110 ~ 256,000)
- 패리티 검사의 종류 (None, Even, Odd, Mark, Space)
- 스톱 비트 (1, 1.5, 2)
- 데이터 비트 (4, 5, 6, 7, 8)
- XON 문자 (디폴트 11h)
- XOFF 문자 (디폴트 13h)
- XON Limit (1024k)
- XOFF Limit (2048k)
- 에러 문자 (0)
- 흐름 제어 종류 (RTS/CTS, XON/XOFF, DSR/DTR)

이들을 각각의 프로퍼티로 제어하려면 Set 메소드를 정의해야 한다. 이를 바탕으로 컴포넌트를 다음과 같이 선언한다.

```
unit Serial;
interface
  uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
```

```
type
  TCommPort = (cpCOM1, cpCOM2, cpCOM3, cpCOM4, cpCOM5, cpCOM6,
    cpCOM7, cpCOM8);
  TBaudRate = (br110, br300, br 600, br1200, br2400, br4800, br9600, br14400, br19200,
    br38400, br56000, br128000, br256000);
  TParityType = (pcNone, pcEven, pcOdd, pcMark, pcSpace);
  TStopBits = (sbOne, sbOnePtFive, sbTwo);
  TDataBits = (db4, db5, db6, db7, db8);
  TFlowControl = (fcNone, fcXON_XOFF, fcRTS_CTS, fsDSR_DTR);
  TNotifyTXEvent = procedure (Sender: TObject; Data: String) of Object;
  TNotifyRXEvent = procedure (Sender: TObject; Data: String) of Object;

const
  dflt_CommPort = cpCOM2;
  dflt_BaudRate = br14400;
  dflt_ParityType = pcNone;
  dflt_ParityErrorChecking = False;
  dflt_ParityErrorChar = 0;
  dflt_ParityErrorReplacement = False;
  dflt_StopBits = sbNone;
  dflt_DataBits = db8;
  dflt_XONChar = $11;
  dflt_XOFFChar = $13;
  dflt_XONLim = 1024;
  dflt_XOFFLim = 2048;
  dflt_ErrorChar = 0;
  dflt_FlowControl = fcNone;
  dflt_StripNullChars = False;
  dflt_EOFChar = 0;

TSerialPort = class(TComponent)
private
  hCommPort: THandle;
  FCommPort: TCommPort;
  FBaudRate: TBaudRate;
  FParityType: TParityType;
```

```
FParityErrorChecking: Boolean;
FParityErrorChar: Byte;
FParityErrorReplacement: Boolean;
FStopBits: TStopBits;
FDataBits: TDataBits;
FXONChar: Byte;
FXOFFChar: Byte;
FXONLim: Word;
FXOFFLim: Word;
FErrorChar: Byte;
FFlowControl: TFlowControl;
FStripNullChars: Boolean;
FEOFChar: Byte;
FOnTransmit: TNotifyTXEvent;
FOnReceive: TNotifyRXEvent;
FAfterTransmit: TNotifyTXEvent;
FAfterReceive: TNotifyRXEvent;
FRXBytes: DWORD;
FTXBytes: DWORD;
ReadBuffer: String;
procedure SetCommPort(Value: TCommPort);
procedure SetBaudRate(Value: TBaudRate);
procedure SetParityType(Value: TParityType);
procedure SetParityErrorChecking(Value: Boolean);
procedure SetParityErrorChar(Value: Byte);
procedure SetParityErrorReplacement(Value: Boolean);
procedure SetStopBits(Value: TStopBits);
procedure SetDataBits(Value: TDataBits);
procedure SetXONChar(Value: Byte);
procedure SetXOFFChar(Value: Byte);
procedure SetXONLim(Value: Word);
procedure SetXOFFLim(Value: Word);
procedure SetErrorChar(Value: Byte);
procedure SetFlowControl(Value: TFlowControl);
procedure SetStripNullChars(Value: Boolean);
procedure SetEOFChar(Value: Value);
```

```
    procedure Initialize_DCB;
  protected
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function OpenPort(MyCommPort: TCommport): Boolean;
    function ClosePort: Boolean;
    procedure SendData(Data: PChar; Size: DWORD);
    function GetData: String;
    function PortIsOpen: Boolean;
    procedure FlushTX;
    procedure FlushRX;
  published
    property CommPort: TCommPort read FCommPort write SetCommPort default dflt_CommPort;
    property BaudRate: TBaudRate read FBaudRate write Set FBaudRate default dflt_BaudRate;
    property ParityType: TParityType read FParityType write SetParityType default dflt_ParityType;
    property ParityErrorChecking: Boolean read FParityErrorChecking write SetParityErrorChecking
      default dflt_ParityErrorChecking;
    property ParityErrorChar: Byte read FParityErrorChar write SetParityErrorChar
      default dflt_ParityErrorChar;
    property StopBits: TStopBits read FStopBits write SetStopBits default dflt_StopBits;
    property DataBits: TDataBits read FDataBits write SetDataBits default dflt_DataBits;
    property XONChar: Byte read FXONChar write SetXONChar default dflt_XONChar;
    property XOFFChar: Byte read FXOFFChar write SetXOFFChar default dflt_XOFFChar;
    property XONLim: Word read FXONLim write SetXONLim default dflt_XONLim;
    property XOFFLim: Word read FXOFFLim write SetXOFFLim default dflt_XOFFLim;
    property ErrorChar: Byte read FErrorChar write SetErrorChar default dflt_ErrorChar;
    property FlowControl: TFlowControl read FFlowControl write SetFlowControl
      default dflt_FlowControl;
    property StripNullChars: Boolean read FStripNullChars write SetStripNullChars
      default dflt_StripNullChars;
    property EOFChar: Byte read FEOFChar write SetEOFChar default dflt_EOFChar;
    property OnTransmit: TNotifyTXEvent read FOnTransmit write FOnTransmit;
    property OnReceive: TNotifyRXEvent read FOnReceive write FOnReceive;
    property AfterTransmit: TNotifyTXEvent read FAfterTransmit write FAfterTransmit;
    property AfterReceive: TNotifyRXEvent read FAfterReceive write FAfterReceive;
```

```
end;

procedure Register;
```

이들 각각의 프로퍼티에 대해 앞에서 보다시피 dflt_ 로 시작하는 디폴트 상수값을 지정하고, Set 메소드를 정의하였다. 이들 Set 메소드의 구현 방법은 기본적으로 다음과 같은 형태를 가진다.

```
procedure TSerialPort.SetProperty(Value: TDefinedType);
begin
  if Value <> FProperty then
  begin
    FProperty := Value;
    Initialize_DCB;
  end;
end;
```

프로퍼티를 설정하는 방법은 비교적 쉽게 이해할 수 있을 것이다. 이번에는 이벤트를 정의할 차례이다. TSerialPort 클래스에서는 OnTransmit, OnReceive, AfterTransmit, AfterRecieve 라는 4 개의 이벤트를 제공한다. 각각의 이벤트는 Sender 와 Data 라는 파라미터를 가진다.
이제부터 앞에서 선언한 컴포넌트의 메소드를 구현해보자. 먼저 Create, Destroy, PortIsOpen 메소드를 다음과 같이 구현한다.

```
constructor TSerialPort.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  hCommPort := INVALID_HANDLE_VALUE;
  FCommPort := dflt_CommPort;
  FBaudRate := dflt_BaudRate;
  FParityCheck := dflt_ParityCheck;
  FStopBits := dflt_StopBits;
  FDataBits := dflt_DataBits;
  FXONChar := dflt_XONChar;
  FXOFFChar := dflt_XOFFChar;
  FXONLim := dflt_XONLim;
```

```
  FXOFFLim := dflt_XOFFLim;
  FErrorChar := dflt_ErrorChar;
  FFlowControl := dflt_FlowControl;
  FOnTransmit := nil;
  FOnReceive := nil;
end;

destructor TSerialPort.Destroy;
begin
  ClosePort;
  inherited Destroy;
end;

function TSerialPort.PortIsOpen: Boolean;
begin
  Result := hCommPort <> INVALID_HANDLE_VALUE;
end;
```

Create 메소드에서는 데이터 필드의 초기값을 설정하게 된다. 여기서 눈여겨 볼 것은 hCommPort 핸들을 INVALID_HANDLE_VALUE 로 설정한 부분이다. 이는 시리얼 포트가 성공적으로 열렸는지 테스트하고, 포트에 특정 작업이 이루어지는 것을 막는 등의 역할을 하게 된다. 이와 연관되어서 PortIsOpen 메소드가 사용된다.
Destroy 메소드에서는 열린 포트를 닫는 ClosePort 메소드를 호출한다.
그러면 실제로 포트를 열고, 닫는 OpenPort 와 ClosePort 메소드를 구현하도록 하자.
OpenPort 메소드는 앞에서 설명한 CreateFile API 함수를 이용해서 포트에 대한 핸들을 얻어오게 되며, 성공적으로 이 작업이 이루어지면 True 를 반환한다. 이때 포트를 열기 위해서는 이미 열려있는 포트를 닫아야 하므로 ClosePort 메소드를 먼저 호출한다. 그리고, Initialize_DCB 메소드를 호출해서 포트를 초기화 한다.
ClosePort 메소드는 CloseHandle API 함수를 호출해서 포트에 대한 핸들을 해제하게 되는데, 이때 PurgeComm API 함수를 사용해서 버퍼의 값을 비우게 되는 FlushTX, FlushRX 메소드를 먼저 호출한다. 그리고, 포트에 대한 핸들에 아무것도 지정되지 않았다는 것을 나타내는 INVALID_HANDLE_VALUE 값을 지정한다.

```
function TSerialPort.OpenPort(MyCommPort: TCommPort): Boolean;
var
  MyPort: PChar;
```

```
begin
  ClosePort;
  MyPort := PChar('COM' + IntToStr(Ord(MyCommPort) + 1));
  hCommPort := CreateFile(MyPort, GENERIC_READ or GENERIC_WRITE, 0, nil,
          OPEN_EXISTING, 0, 0);
  Initialize_DCB;
  if hCommPort <> INVALID_HANDLE_VALUE then
  begin
    Result := True;
    FCommPort := MyCommPort;
  end
  else Result := False;
end;

function TSerialPort.ClosePort: Boolean;
begin
  FlushTX;
  FlushRX;
  Result := CloseHandle(hCommPort);
  hCommPort := INVALID_HANDLE_VALUE;
end;

function TSerialPort.FlushRX;
begin
  if hCommPort = INVALID_HANDLE_VALUE then Exit;
  PurgeComm(hCommPort, PURGE_RXABORT or PURGE_RXCLEAR);
  ReadBuffer := '';
end;

function TSerialPort.FlushTX;
begin
  if hCommPort = INVALID_HANDLE_VALUE then Exit;
  PurgeComm(hCommPort, PURGE_TXABORT or PURGE_TXCLEAR);
end;
```

포트를 초기화하는 부분은 Initialize_DCB 메소드에 의해서 구현된다. 이 메소드는 프로퍼

티가 바뀔 때마다 이 변화를 DCB 구조체에 반영하는 역할을 한다. 실제 구현 방법은 다음 코드를 보면 쉽게 이해할 수 있을 것이다.

```
procedure TSerialPort.Initialize_DCB;
var
  MyDCB: TDCB;
begin
  if hCommPort = INVALID_HANDLE_VALUE then Exit;
  GetCommState(hCommPort, MyDCB);
  case FBaudRate of
    br110: MyDCB.BaudRate := 110;
    br300: MyDCB.BaudRate := 300;
    br600: MyDCB.BaudRate := 600;
    br1200: MyDCB.BaudRate := 1200;
    br2400: MyDCB.BaudRate := 2400;
    br4800: MyDCB.BaudRate := 4800;
    br9600: MyDCB.BaudRate := 9600;
    br14400: MyDCB.BaudRate := 14400;
    br19200: MyDCB.BaudRate := 19200;
    br38400: MyDCB.BaudRate := 38400;
    br56000: MyDCB.BaudRate := 56000;
    br128000: MyDCB.BaudRate := 128000;
    br256000: MyDCB.BaudRate := 256000;
  end;

  case FParityType of
    pcNone: MyDCB.Parity := NOPARITY;
    pcEven: MyDCB.Parity := EVENPARITY;
    pcOdd: MyDCB.Parity := ODDPARITY;
    pcMark: MyDCB.Parity := MARKPARITY;
    pcSpace: MyDCB.Parity := SPACEPARITY;
  end;

  if FParityErrorChecking then inc(MyDCB.Flags, $0002);
  if FParityErrorReplcement then inc(MyDCB.Flags, $0021);
  MyDCB.ErrorChar := Char(FErrorChar);
```

```
  case FStopBits of
    sbOne: MyDCB.StopBits := ONESTOPBIT;
    sbOnePtFive: MyDCB.StopBits := ONE5STOPBITS;
    sbTwo: MyDCB.StopBits := TWOSTOPBITS;
  end;

  case FDataBits of
    db4: MyDCB.ByteSize := 4;
    db5: MyDCB.ByteSize := 5;
    db6: MyDCB.ByteSize := 6;
    db7: MyDCB.ByteSize := 7;
    db8: MyDCB.ByteSize := 8;
  end;

  case FFlowControl of
    fcXON_XOFF: MyDCB.Flags := MyDCB.Flags or $0020 or $0018;
    fcRTS_CTS: MyDCB.Flags := MyDCB.Flags or $0004 or
          $0024 * RTS_CONTROL_HANDSHAKE;
    fcDSR_DTR: MyDCB.Flags := MyDCB.Flags or $0008 or
          $0010 * DTR_CONTROL_HANDSHAKE;
  end;

  if FStripNullChars then inc(MyDCB.Flags, $0022);
  MyDCB.XONChar := Char(FXONChar);
  MyDCB.XOFFChar := Char(FXOFFChar);
  MyDCB.XONLim := FXONLim;
  MyDCB.XOFFLim := FXOFFLim;
  if FEOFChar <> 0 then MyDCB.EOFChar := Char(EOFChar);
  SetCommState(hCommPort, MyDCB);
end;
```

이 메소드를 구현하기 위해서 DCB 구조체에 비트 플래그를 사용하였다. 이러한 플래그에는 여러가지 종류가 있는데, 대부분은 흐름 제어와 패리티 검사에 사용된다. 사용 가능한 플래그를 나열하면 다음과 같다.

```
fParity = $0002;                    //설정되면 패리티 검사를 한다.
fOutxCtsFlow = $0004;               //CTS 가 high 가 아니면 데이터가 전송되지 않음
fOutxDsrFlow = $0008;               //DSR 이 high 가 아니면 데이터가 전송되지 않음
fDtrControl = $0010;
        //DTR_CONTROL_ENABLE, DTR_CONTROL_DISABLE, DTR_CONTROL_HANDSHAKE
fDsrSensitivity = $0012;            //DSR 이 high 가 아니면 모든 바이트가 무시됨
fTxContinueOnXOff = $0014;
        //이 플래그가 설정되어 있으면 XON 문자가 전송되어도 데이터를 보내며,
          그렇지 않으면 XONLim 이 도달할 때까지 데이터를 보내지 않는다.
fOutX = $0018;                      //전송시 XON/XOFF 흐름 제어를 사용
fInX = $0020;                       //수신시 XON/XOFF 흐름 제어를 사용
fErrorChar := $0021;                //패리티 에러가 발생하면 이 문자로 교체
fNull = $0022;                      //Null 문자를 잘라냄
fRtsControl = $0024;                //RTS 흐름 제어 사용
```

이제 실제로 데이터를 받고, 전송하는 GetData, SetData 메소드를 구현해 보자. SendData 가 보다 쉽게 구현이 가능하므로, 여기에 대해서 먼저 알아본다. 이 메소드가 호출되면 먼저 OnTransmit 이벤트 핸들러를 호출해서 데이터를 전송하기 전에 여러가지 조작을 할 수 있도록 허용한다. 마찬가지로 이 메소드의 마지막에는 AfterTransmit 이벤트 핸들러를 호출해서 데이터 전송 후에 여러가지 작업을 할 수 있도록 한다. 실제로 데이터를 전송하는 작업은 WriteFile API 함수를 사용한다. 파라미터로 포트에 대한 핸들과 전송할 데이터의 포인터, 그리고 전송할 데이터의 크기를 넘겨주면 실제로 전송된 데이터의 크기가 결과값으로 반환된다. 마지막 파라미터에는 전송하는 방법을 지정할 수 있는데, nil 로 설정해서 동시에 읽고, 쓰는 작업을 할 수 없도록 하였다. 이 값을 'overlapped' 스타일의 비동기 통신 모드로 설정하면 동시에 읽고, 쓰는 작업을 할 수 있는데 이를 구현하기 위해서는 버퍼처리, 메모리 재할당 기법 등의 고급스런 테크닉을 사용해야 한다.

```
procedure TSerialPort.SendData(Data: PChar; Size: DWORD);
var
  NumBytesWritten: DWORD;
begin
  if hCommPort = INVALID_HANDLE_VALUE then Exit;
  if Assigned(FOnTransmit) then FOnTransmit(Self, Data);
  WriteFile(hCommPort, Data^, Size, NumBytesWritten, nil);
  if Assigned(FAfterTransmit) then FAfterTransmit(Self, Data);
end;
```

GetData 메소드에서도 프로시저의 처음과 끝 부분에 OnReceive, AfterReceive 이벤트 핸들러를 호출한다. 그리고, 데이터를 읽어 오는 데에는 ReadFile API 함수를 사용하는데, 이 함수에서 읽어온 데이터를 저장할 버퍼와 데이터의 크기를 지정한다. 이때 데이터의 크기를 적절하게 지정하기 위해 ClearCommError API 함수를 이용해서 TComStat 형의 데이터를 읽어와서, 버퍼의 크기를 결정하는 과정을 거친다.

```
function TSerialPort.GetData: String;
var
  NumBytesRead: DWORD;
  BytesInQueue: LongInt;
  oStatus: TComStat;                    //에러 코드를 담기 위해서 사용한다.
  dwErrorCode: DWORD;
begin
  if hCommPort = INVALID_HANDLE_VALUE then Exit;
  if Assigned(FOnReceive) then FOnReceive(Self, ReadBuffer);
  ClearCommError(hCommPort, dwErrorCode, @oStatus);
  BytesInQueue := oStatus.cbInQue;
  if BytesInQueue > 0 then
  begin
    SetLength(ReadBuffer, BytesInQueue + 1);
    ReadFile(hCommPort, PChar(ReadBuffer)^, BytesInQueue, NumBytesRead, nil);
    SetLength(ReadBuffer, StrLen(PChar(ReadBuffer)));
  end;
  if Assigned(FAfterReceive) then FAfterReceive(Self, ReadBuffer);
  Result := ReadBuffer;
end;
```

이렇게 해서 가장 기본적인 작동을 하는 시리얼 포트 컴포넌트를 하나 완성하였다. 이를 바탕으로 보다 기능을 강화해서 좋은 컴포넌트로 만들 수도 있는데, 예를 들어서 CRC 검사를 하고, 비동기 통신이 가능하며, 데이터를 읽고 쓸 때 멀티 쓰레딩 기법을 이용하는 컴포넌트 등을 생각할 수 있겠다.

TSerialPort 컴포넌트의 활용

이 컴포넌트를 이용하는 방법을 알아보자. TSerialPort 컴포넌트의 기능과 간단한 예제 코

드를 나열하면 다음과 같다.

1. 포트 열기:

```
SerialPort1.OpenPort(cpCOM2);
or if SerialPort1.OpenPort(cpCOM2) then do something.
```

2. 데이터 전송:

```
var
  S: PChar;
begin
  S := PChar(Edit1.Text);
  SerialPort1.SendData(S, SizeOf(s));
  SerialPort1.SendData((chr(13)), 1);                    //캐리지 리턴
end;
```

3. 데이터 읽기:

```
Memo1.Text := SerialPort1.GetData;
```

4. 버퍼 비우기:

```
SerialPort1.FlushRX;
SerialPort1.FlushTX;
```

5. 포트가 열려 있는지 확인:

```
if SerialPort1.PortIsOpen then do something.
```

6. 포트 닫기

```
SerialPort1.ClosePort;
```

정 리 (Summary)

이번 장에서는 직렬 포트를 이용한 통신을 하는 기본적인 방법에 대해서 알아보았다. 최근의 인터넷 환경의 발달로 모뎀과 직렬 포트를 직접 이용하는 사례가 많이 줄어들고 있는 것이 사실이다. 그렇지만, 직렬 포트는 새로운 기계를 제작하여 컴퓨터로 제어를 하거나, 그 밖에도 여러가지 아이디어를 가지고 가장 쉽게 컴퓨터와 연결할 수 있는 수단이다. 그러므로, 직렬 포트를 활용하는 방법을 알아두는 것은 의미가 있다.