# Практическое задание №1

# Программирование интерфейса RS-232C средствами Win32 API

# Синхронный режим

1. Ознакомиться с принципами программирования коммуникационных портов средствами Win32 API.
2. Написать консольное пользовательское приложение Win32 API.
   Программа должна осуществлять передачу *k* строк в **синхронном режиме** через последовательный порт COMn, каждая из которых представляет фамилию участника бригады из *k* человек (*k* вызовов функции записи в порт).
3. Запустить программу Advanced Serial Port Terminal (Пуск►Программы (Все программы) ►Eltima Software►Advanced Serial Port Terminal).
4. Для Advanced Serial Port Terminal создать новую сессию, выбрав порт COMm и установив его параметры в соответствии с вариантом задания. Открыть порт и установить режим Edit View.
5. Запустить разработанную программу на выполнение, в Advanced Serial Port Terminal наблюдать результат.

**Примечание**

На компьютере должен быть установлен виртуальный NULL-модем, соединяющий виртуальные порты COMn и COMm.

| Вариант | Порт | | Настройки портов |
|---|---|---|---|
| | Advanced Serial Port (COMn) | Программа пользователя (COMm) | |
| 1 | COM3 | COM4 | 38400/8o1 |
| 2 | COM4 | COM3 | 19200/8n2 |
| 3 | COM4 | COM3 | 9600/8s1 |
| 4 | COM3 | COM4 | 9600/8e1 |
| 5 | COM3 | COM4 | 38400/8o1 |
| 6 | COM4 | COM3 | 4800/8m1 |
| 7 | COM4 | COM3 | 4800/8e1 |
| 8 | COM3 | COM4 | 38400/8m1 |
| 9 | COM4 | COM3 | 9600/8o1 |
| 10 | COM4 | COM3 | 9600/8e1 |
| 11 | COM3 | COM4 | 38400/8n1 |
| 12 | COM4 | COM3 | 19200/8o1 |

# Программирование интерфейса RS-232C средствами Win32 API

## Часть I – Общие принципы

Последовательный порт очень часто используют как канал обмена информацией в промышленных системах управления и сбора данных, например, для обмена данными с контролерами.

Для написания приложения работающего с последовательным портом с использованием Win32 API наряду с фундаментальными понятиями Win32 необходимо иметь представление о многопоточном программировании и синхронизации потоков. В целом же работа с последовательными портами в Win32 идентична работе с файлами, а соответственно, будет состоять из нескольких этапов:

- открытие порта;
- установка его параметров;
- цикл чтения/записи данных в порт;
- закрытие порта.

### Открытие порта

Для открытия порта используется функция **`CreateFile`**.

```
HANDLE CreateFile(
       LPCTSTR               lpFileName,
       DWORD                 dwDesiredAccess,
       DWORD                 dwShareMode,
       LPSECURITY_ATTRIBUTES lpSecurityAttributes,
       DWORD                 dwCreationDistribution,
       DWORD                 dwFlagsAndAttributes,
       HANDLE                hTemplateFile
);
```

**`lpFileName`** — Указатель на строку с именем открываемого или создаваемого файла. Последовательные порты имеют имена "COM1", "COM2" и так вплоть до "COM9". Для доступа к последовательным портам с номерами больше 9 необходимо обращаться как "\\.\COM10", "\\.\COM11" и так далее.

| | |
|---|---|
| **dwDesiredAccess** | Задает тип доступа к файлу. Возможно использование следующих значений:<br>**0** – опрос атрибутов устройства без получения доступа к нему;<br>**GENERIC_READ** – файл открывается в режиме чтения;<br>**GENERIC_WRITE** – файл открывается в режиме записи;<br>**GENERIC_READ\|GENERIC_WRITE** – файл открывается в режиме чтения-записи. |
| **dwShareMode** | Задает параметры совместного доступа к файлу. Коммуникационные порты нельзя делать разделяемыми, поэтому данный параметр должен быть равен **0**. |
| **lpSecurityAttributes** | Задает атрибуты защиты файла. Поддерживается только в Windows NT. Однако при работе с портами должен быть **NULL**. |
| **dwCreationDistribution** | Управляет режимами автосоздания, автоусечения файла и им подобными. Для работы с коммуникационными портами должно задаваться **OPEN_EXISTING**. |
| **dwFlagsAndAttributes** | Задает атрибуты создаваемого файла. Так же управляет различными режимами обработки. Для работы с коммуникационными портами этот параметр должен быть или равным **0**, или **FILE_FLAG_OVERLAPPED**. Нулевое значение используется при синхронной работе с портом, а **FILE_FLAG_OVERLAPPED** при асинхронной (фоновой обработке ввода/вывода). |
| **hTemplateFile** | Задает дескриптор файла-шаблона. При работе с портами всегда должен быть равен **NULL**. |

Таким образом, в самом простейшем случае, фрагмент кода программы будет примерно следующим:

```
HANDLE hCom;

hCom = CreateFile("\\\\.\\COM10",
                  GENERIC_READ|GENERIC_WRITE,
                  0,
                  NULL,
                  OPEN_EXISTING,
                  0,
                  NULL);
```

В дальнейшем необходимо проверить результат работы функции **CreateFile**. При успешном открытии порта **CreateFile** возвращает дескриптор порта, а если произошла ошибка, то возвращает значение **INVALID_HANDLE_VALUE**.

```
if (hCom == INVALID_HANDLE_VALUE)
{
  // ошибка открытия порта

}
else
{
  // порт открыт успешно
}
```

## Установка параметров последовательного порта

Установка параметров последовательного порта осуществляется путём заполнения полей структуры **DCB** (**Device Control Block**). В ней задаются все наиболее важные свойства порта: скорость передачи данных, количество бит данных, стоповых бит, контроль чётности и др.

```
typedef struct _DCB {
        DWORD DCBlength;            // sizeof(DCB)
        DWORD BaudRate;             // current baud rate
        DWORD fBinary:1;            // binary mode, no EOF check
        DWORD fParity:1;            // enable parity checking
        DWORD fOutxCtsFlow:1;       // CTS output flow control
        DWORD fOutxDsrFlow:1;       // DSR output flow control
        DWORD fDtrControl:2;        // DTR flow control type
        DWORD fDsrSensitivity:1;    // DSR sensitivity
        DWORD fTXContinueOnXoff:1;  // XOFF continues Tx
        DWORD fOutX:1;              // XON/XOFF out flow control
        DWORD fInX:1;               // XON/XOFF in flow control
        DWORD fErrorChar:1;         // enable error replacement
        DWORD fNull:1;              // enable null stripping
        DWORD fRtsControl:2;        // RTS flow control
        DWORD fAbortOnError:1;      // abort reads/writes on error
        DWORD fDummy2:17;           // reserved
        WORD  wReserved;            // not currently used
        WORD  XonLim;               // transmit XON threshold
        WORD  XoffLim;              // transmit XOFF threshold
        BYTE  ByteSize;             // number of bits/byte, 4-8
        BYTE  Parity;               // 0-4=no,odd,even,mark,space
        BYTE  StopBits;             // 0,1,2 = 1, 1.5, 2
        char  XonChar;              // Tx and Rx XON character
        char  XoffChar;             // Tx and Rx XOFF character
        char  ErrorChar;            // error replacement character
        char  EofChar;              // end of input character
        char  EvtChar;              // received event character
        WORD  wReserved1;           // reserved; do not use
} DCB;
```

Исключая из рассмотрения ничего не значащие **fDummy2**, **wReserved** и **wReserved1**, получим 25 параметров, полностью конфигурирующих параметры последовательного порта.

| | |
|---|---|
| **DCBlength** | Размер структуры **DCB** в байтах. |
| **BaudRate** | Скорость передачи данных. Возможно указание следующих констант: **CBR_110**, **CBR_300**, **CBR_600**, **CBR_1200**, **CBR_2400**, **CBR_4800**, **CBR_9600**, **CBR_14400**, **CBR_19200**, **CBR_38400**, **CBR_56000**, **CBR_57600**, **CBR_115200**, **CBR_128000**, **CBR_256000** или просто числовых значений (не рекомендуется): **110**, **300**, **600**, **1200**, **2400**, **4800**, **9600**, **14400**, **19200**, **38400**, **56000**, **57600**, **115200**, **128000**, **256000**. |
| **fBinary** | Устанавливает режим передачи двоичных данных. Всегда **TRUE**, т.к. Win32 не поддерживает не двоичные режимы передачи данных. |
| **fParity** | Контроль чётности. Если **TRUE**, происходит проверка чётности и система сообщает о наличии ошибок при их возникновении. |
| **fOutxCtsFlow** | Указывает способ контроля линии CTS *(clear to send*) для управления передачей. Если **TRUE** и CTS имеет низкий потенциал, посылка данных приостанавливается до тех пор, пока на линии CTS не появится высокий потенциал. |
| **fOutxDsrFlow** | Указывает способ контроля линии DSR (*data set ready*) для управления передачей. Если **TRUE** и DSR имеет низкий потенциал, посылка данных приостанавливается до тех пор, пока на линии CTS не появится высокий потенциал. |
| **fDtrControl** | Устанавливает способ контроля над DTR (*data terminal ready*). Может принимать следующие значения **DTR_CONTROL_DISABLE**, **DTR_CONTROL_ENABLE** и **DTR_CONTROL_HANDSHAKE**. |
| **fDsrSensitivity** | Если **TRUE**, драйвер последовательного порта игнорирует полученную информацию при низком потенциале на линии DSR. |
| **fTXContinueOnXoff** | Если **TRUE**, после посылки символа XOFF передача продолжается. |
| **fOutX** | Если **TRUE**, передача останавливается при получении символа XOFF и возобновляется при получении XON. |
| **fInX** | Если **TRUE**, то при наличии во входном буфере **XoffLim** байт посылает символ XOFF, а при наличии **XonLim** посылается символ XON. |
| **fErrorChar** | Если **TRUE**, а также **fParity** тоже **TRUE**, то байты, полученные с ошибкой контроля четности, будут заменены символом указанным в поле **ErrorChar**. |
| **fNull** | Если **TRUE**, то нулевые байты отбрасываются. |
| **fRtsControl** | Устанавливает способ контроля над RTS (*request to send*). Может принимать следующие значения **RTS_CONTROL_DISABLE**, **RTS_CONTROL_ENABLE**, **RTS_CONTROL_HANDSHAKE** и **RTS_CONTROL_TOGGLE**. |
| **fAbortOnError** | Если **TRUE**, драйвер последовательного порта при возникновении ошибки прекращает все операции приёма и передачи данных. Драйвер сообщает об ошибке (**ERROR_IO_ABORTED**) и пользователь должен вызвать функцию **ClearCommError** для возобновления операций ввода-вывода. |

| | |
|---|---|
| **XonLim** | Указывает минимальное количество байт во входном буфере, при котором происходит посылка символа XON. |
| **XoffLim** | Указывает максимальное количество байт во входном буфере, при котором происходит посылка символа XOFF. |
| **ByteSize** | Количество информационных бит (от 4 до 8) в байте. |
| **Parity** | Способ контроля чётности, может принимать следующие значения: **NOPARITY**, **ODDPARITY**, **EVENPARITY**, **MARKPARITY**, **SPACEPARITY.** |
| **StopBits** | Количество стоповых бит, может принимать значения: **ONESTOPBIT**, **ONE5STOPBIT**, **TWOSTOPBIT.** |
| **XonChar** | Значение символа XON (для приёма и передачи). |
| **XoffChar** | Значение символа XOFF (для приёма и передачи). |
| **ErrorChar** | Значение символа, используемого для замены байт принятых с ошибкой чётности. |
| **EofChar** | Значение символа передаваемого при конце передачи данных. |
| **EvtChar** | Значение символа, которое вызывает наступление события **EV_RXFLAG**. |

Инициализировать **DCB** можно несколькими методами. Во-первых, можно заполнить все поля вручную, однако в этом случае нужно очень четко представлять, как работает последовательный порт. Во-вторых, можно получить текущие настройки порта (или же настройки порта по умолчанию) и изменить лишь необходимые параметры. Для этой цели необходимо воспользоваться функцией **GetCommState**. Прототип этой функции выглядит следующим образом:

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

| | |
|---|---|
| **hFile** | Дескриптор открытого коммуникационного порта, полученный функцией **CreateFile**. Следовательно, прежде чем получить параметры порта, он должен быть открыт. |
| **lpDCB** | Указатель на структуру **DCB**. Для **DCB** должен быть выделен блок памяти. |

К примеру, в следующем фрагменте кода устанавливаются такие параметры порта: скорость передачи данных 9600 байт/сек, пакет содержит восемь бит данных, один стоповый бит, контроль по чётности:

```
DCB dcb = {0};

if (!GetCommState(hCom, &dcb))
{
    // ошибка получения параметров DCB
}
```

```
else
{
    // структура DCB инициализирована,
    // можно устанавливать свои значения

    dcb.BaudRate = 9600;
    dcb.ByteSize = 8;
    dcb.Parity   = EVENPARITY;
    dcb.StopBits = ONESTOPBIT;
}
```

И, в-третьих, заполнить поля структуры **DCB** можно на основе текстовой строки, формат которой аналогичен функции **mode** DOS:

```
[baud=b][parity=p][data=d][stop=s][to={on|off}][xon={on|off}]
[odsr={on|off}][octs={on|off}][dtr={on|off|hs}][rts={on|off|hs|tg}]
[idsr={on|off}]
```

Например, следующая строка задает скорость 9600, с дополнением до чётного числа единиц, 8 бит данных и 1 стоповый бит.

```
baud=9600 parity=E data=8 stop=1
```

Для этого существует функция **BuildCommDCB**:

```
BOOL BuildCommDCB(
     LPCTSTR lpDef,
     LPDCB   lpDCB
);
```

**lpDef** — Указатель на строку с конфигурационной информацией в формате команды **mode** (см. выше).

**lpDCB** — Указатель на структуру **DCB**. При этом структура должна быть уже создана, и поле **DCBlength** должно содержать корректное значение.

```
DCB dcb = { 0 };
dcb.DCBlength = sizeof(dcb);
if(!BuildCommDCB("baud=9600 parity=E data=8 stop=1", &dcb))
{
  // ошибка заполнения DCB
}
else
{
  // структура DCB инициализирована
}
```

Функция **BuilCommDCB** может сформировать значения структуры **DCB** и на основе устаревшего синтаксиса, когда параметры работы коммуникационного порта задаются строкой вида

```
9600,n,8,1
```

Однако смешивать различные форматы в одном вызове функции **BuilCommDCB** не допускается.

Используются следующие соглашения:

- Для строк вида **9600,n,8,1** (не заканчивающихся символами **x** или **p**):

  **fInX**, **fOutX**, **fOutXDsrFlow**, **fOutXCtsFlow** устанавливаются в **FALSE**

  **fDtrControl** устанавливается в **DTR_CONTROL_ENABLE**

  **fRtsControl** устанавливается в **RTS_CONTROL_ENABLE**

- Для строк вида **9600,n,8,1,x** (заканчивающихся символом **x**):

  **fInX**, **fOutX** устанавливаются в **TRUE**

  **fOutXDsrFlow**, **fOutXCtsFlow** устанавливаются в **FALSE**

  **fDtrControl** устанавливается в **DTR_CONTROL_ENABLE**

  **fRtsControl** устанавливается в **RTS_CONTROL_ENABLE**

- Для строк вида **9600,n,8,1,p** (заканчивающихся символом **p**):

  **fInX**, **fOutX** устанавливаются в **FALSE**

  **fOutXDsrFlow**, **fOutXCtsFlow** устанавливаются TRUE

  **fDtrControl** устанавливается в **DTR_CONTROL_HANDSHAKE**

  **fRtsControl** устанавливается в **RTS_CONTROL_HANDSHAKE**

Чтобы изменения параметров вступили в силу необходимо использовать функцию **SetCommState**:

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

Параметры этой функции аналогичны функции **GetCommState**.

```
if(!SetCommState(hCom, &dcb))
{
  // ошибка установки параметров DCB
}
else
{
  // параметры DCB установлены
}
```

## Чтение и запись данных

Чтение и запись данных в последовательный порт в Win32 реализуются двумя способами синхронным (nonoverlapped) и асинхронным (overlapped). При синхронном способе возврат из вызванной функции ввода-вывода не происходит до тех пор, пока не закончится чтение или запись данных. Таким образом, поток, вызвавший функцию чтения записи, оказывается заблокированным до тех пор, пока не завершиться операции ввода-вывода. После завершения операции ввода-вывода поток разблокируется и его выполнение продолжается. В приложениях использующих многопотоковость Windows, эта проблема не является серьезным препятствием: другие потоки продолжают работать, в то время когда поток выполняющий ввод-вывод оказывается заблокированным. Но следует учитывать, что если другие потоки вызывают функции работы с портом, когда другой поток ждёт завершения операции ввода-вывода, то они тоже окажутся заблокированными.

При асинхронном способе происходит моментальный возврат из вызывающий функции, даже если операция ввода-вывода ещё не закончена. Программа может использовать время между вызовами функций инициализации и окончания ввода-вывода для выполнения какого-либо фонового кода.

В любом случае приём данных выполняется функцией **ReadFile**:

```
BOOL ReadFile(
      HANDLE        hFile,
      LPVOID        lpBuffer,
      DWORD         nNumOfBytesToRead,
      LPDWORD       lpNumOfBytesRead,
      LPOVERLAPPED lpOverlapped
);
```

а передача данных – функцией **WriteFile**:

```
BOOL WriteFile(
      HANDLE        hFile,
      LPVOID        lpBuffer,
      DWORD         nNumOfBytesToWrite,
      LPDWORD       lpNumOfBytesWritten,
      LPOVERLAPPED lpOverlapped
);
```

| | |
|---|---|
| **hFile** | Дескриптор открытого коммуникационного порта. |
| **lpBuffer** | Адрес буфера. Для операции записи данные из этого буфера будут передаваться в порт. Для операции чтения в этот буфер будут помещаться принятые из линии данные. |
| **nNumOfBytesToRead**<br>**nNumOfBytesToWrite** | Число ожидаемых к приему или предназначенных к передаче байт. |
| **lpNumOfBytesRead**<br>**lpNumOfBytesWritten** | Число фактически принятых или переданных байт. |

**lpOverlapped** — Адрес структуры **OVERLAPPED**, используемой для асинхронных операций. Для синхронных операций данный параметр должен быть равным **NULL**.

Работа с функциями чтения-записи существенно зависит от режима работы, поэтому примеры вызова таких функций будут приведены в соответствующих разделах.

## Закрытие порта

Открытый порт должен быть закрыт перед завершением работы программы. В Win32 закрытие объекта по его дескриптору выполняет функция **CloseHandle**:

```
BOOL CloseHandle(
     HANDLE hObject
);
```

Функция имеет единственный параметр – дескриптор закрываемого объекта. При успешном завершении функция возвращает не нулевое значение, при ошибке – нуль.

## Сервисные функции

В первый момент после открытия коммуникационного порта в его буферах приёма и передачи могут находиться некоторые данные – «мусор», анализировать которые бессмысленно. Поэтому первым делом необходимо сбросить порт, для чего предназначена функция **PurgeComm**:

```
BOOL PurgeComm(
     HANDLE hFile,
     DWORD  dwFlags
);
```

**hFile** — Дескриптор открытого коммуникационного порта, полученный функцией **CreateFile**.

**dwFlags** — Флаги, описывающие действия, выполняемые над портом:

**PURGE_TXABORT** – немедленно прекращает все операции записи, даже если они не завершены;

**PURGE_RXABORT** – немедленно прекращает все операции чтения, даже если они не завершены;

**PURGE_TXCLEAR** – очищает очередь передачи в драйвере;

**PURGE_RXCLEAR** – очищает очередь приема в драйвере.

Значения можно комбинировать, используя операцию побитовое ИЛИ.

```
     PurgeComm(hCom, PURGE_TXABORT | PURGE_TXCLEAR);
```

Вызов этой функции востребован в случаях возникновения ошибок, а также необходим для корректного завершения программы.

Очистка буфера передачи, как и экстренное завершение операции записи, не выполняют передачу данных находящихся в этом буфере – данные просто отбрасываются. Если же передача остатка данных необходима, то перед вызовом **PurgeComm** следует вызвать функцию **FlushFileBuffers**:

```
BOOL FlushFileBuffers(
      HANDLE hFile
);
```

```
      FlushFileBuffers(hCom);
      PurgeComm(hCom, PURGE_TXABORT | PURGE_RXABORT);
```

Внешние устройства управления объектами, чаще всего подключаемые к портам, обычно обмениваются с компьютером короткими сообщениями. В то же время, если устройство передаёт или принимает блоки данных длиной в несколько тысяч байт, рекомендуется задать размеры очередей приема и передачи драйвера, что позволит избежать потери данных и пауз в работе программы, если данные передаются слишком быстро. С этой целью используется функция **SetupComm**:

```
BOOL SetupComm(
      HANDLE hFile,
      DWORD  dwInQueue,
      DWORD  dwOutQueue
);
```

| | |
|---|---|
| **hFile** | Дескриптор открытого коммуникационного порта, полученный функцией **CreateFile**. |
| **dwInQueue** | Размер (в байтах) очереди приёма. |
| **dwOutQueue** | Размер (в байтах) очереди передачи. |

Размер очереди необходимо выбирать как минимум на 15–20% больше размера реальной посылки. Однако данная функция позволяет лишь указать пожелания пользователя в организации очереди данных – драйвер может внести коррективы или вообще отвергнуть устанавливаемое значение. В этом случае функция **SetupComm** возвращает **FALSE**.