Работа с COM-портом с помощью потоков

Содержание

Введение ... **1**
Основные определения ... **1**
Шаблоны потоков ... **2**
Цикл чтения байтов внутри потока ... 2
Краткое описание алгоритма цикла чтения байтов ... 5
Пояснение параметров функций ... 5
Параметры функции CreateEvent ... 5
Параметры функции SetCommMask ... 6
Параметры функции WaitCommEvent ... 6
Поток записи байтов ... 6
Создание потока ... **8**
Создание потока с помощью класса TThread ... 8
Поток чтения из порта ... 8
Использование метода Synchronize ... 9
Как использовать поток ... 10
Поток записи в порт ... 12
Использование методов Resume() и Suspend() ... 15
Создание потока средствами WinAPI ... 19
Создание потока чтения ... 19
Создание потока записи. Использование функций ResumeThread() и SuspendThread() ... 23
Сравнение использования класса TThread и средств WINAPI ... 25
Описание программы ... **26**
Функция открытия и инициализации порта COMOpen() ... 26
Функция закрытия порта COMClose() ... 31
Конструктор формы и обработчики событий формы и её элементов ... 33
Переменные, используемые в программе ... 36
Описание интерфейса ... **37**
Работа с программой ... **39**
Как протестировать программу ... **39**
Диспетчер задач ... 39
Заглушка для COM-порта ... 40
Список литературы ... **41**

Введение

Данная статья посвящена работе с COM-портом с помощью потоков под ОС Windows. Все приведённые здесь программы написаны в C++Builder 6 с использованием Windows-интерфейса, объектных классов и функций WINAPI.

Если вы ещё не работали с COM-портом в Windows, то рекомендуем сначала прочитать раздел "**Описание программы**", в котором приводится описание инициализации порта и работы с ним.

Основные определения

Перекрываемая операция (overlapped operation) – асинхронная операция, называется перекрываемой, так как функция, которая её активировала, сразу возвращает управление программе, и та может продолжить выполнение других операций как бы поверх запущенной, не дожидаясь её завершения.

Поток (thread) – часть процесса. Потоки представляют собой как бы несколько параллельных нитей выполнения процесса. У процесса всегда есть как минимум один поток – главный.

Функция ожидания – функция, которая на некоторое время переводит поток в режим ожидания события. Благодаря этому поток не занимает процессорное время.

Объект-событие (event) – сигнальный объект, который имеет два состояние: сигнальное состояние (событие наступило) и несигнальное состояние (событие не наступило). Используется для управления и синхронизации потоков, ресурсов и т.п.

 © PIClist RUS, 2007

Событие – в данном случае – некоторое событие, связанное с портом. Например, ожидание поступления байта.

Маска – в данном случае битовая комбинация, показывающая, какие события учитывать, а какие нет.

Шаблоны потоков

Для работы с COM-портом необходимы, как минимум, два потока – для чтения байтов из порта и для записи их в порт.

Цикл чтения байтов внутри потока

Наибольшую сложность представляет собой именно чтение байтов из порта, так как данные поступают извне в большинстве случаев асинхронно.

Ниже приведён шаблон кода, который используется в главной функции потока чтения байтов, а также подробное его объяснения. Поправки, которые необходимо внести при создании потока с помощью класса TThread или функций WinAPI, приводятся в соответствующих разделах.

Кроме функции потока приведены объявления используемых в ней переменных.

```
//-----------------------------------------------------------------------------

#define BUFSIZE 255     //ёмкость буфера

unsigned char bufrd[BUFSIZE], bufwr[BUFSIZE]; //приёмный и передающий буферы

//-----------------------------------------------------------------------------

HANDLE COMport;            //дескриптор порта

//структура OVERLAPPED необходима для асинхронных операций, при этом для операции чтения и записи нужно объявить разные структуры
//эти структуры необходимо объявить глобально, иначе программа не будет работать правильно
OVERLAPPED overlapped;            //будем использовать для операций чтения (см. поток ReadThread)
OVERLAPPED overlappedwr;          //будем использовать для операций записи (см. поток WriteThread)
//-----------------------------------------------------------------------------

//главная функция потока, реализует приём байтов из COM-порта
{
 COMSTAT comstat;  //структура текущего состояния порта, в данной программе используется для определения количества принятых в порт байтов
 DWORD btr, temp, mask, signal;   //переменная temp используется в качестве заглушки

 overlapped.hEvent = CreateEvent(NULL, true, true, NULL);      //создать сигнальный объект-событие для асинхронных операций
 SetCommMask(COMport, EV_RXCHAR);                //установить маску на срабатывание по событию приёма байта в порт
 while(условие)                       //пока поток не будет прерван, выполняем цикл
  {
   WaitCommEvent(COMport, &mask, &overlapped); //ожидать события приёма байта (это и есть перекрываемая операция)
   signal = WaitForSingleObject(overlapped.hEvent, INFINITE);  //приостановить поток до прихода байта
   if(signal == WAIT_OBJECT_0)                                 //если событие прихода байта произошло
    {
     if(GetOverlappedResult(COMport, &overlapped, &temp, true)) //проверяем, успешно ли завершилась перекрываемая операция WaitCommEvent
      if((mask & EV_RXCHAR)!=0)                         //если произошло именно событие прихода байта
       {
        ClearCommError(COMport, &temp, &comstat);              //нужно заполнить структуру COMSTAT
        btr = comstat.cbInQue;                          //и получить из неё количество принятых байтов
        if(btr)                                         //если действительно есть байты для чтения
        {
         ReadFile(COMport, bufrd, btr, &temp, &overlapped);     //прочитать байты из порта в буфер программы
        }
       }
    }
  }
 CloseHandle(overlapped.hEvent);          //перед выходом из потока закрыть объект-событие
}

//-----------------------------------------------------------------------------
```

Теперь подробно объясним приведённый код.

© PIClist RUS, 2007

Чтобы с портом можно было выполнять асинхронные операции, порт должен быть открыт (функцией **CreateFile**) с флагом **FILE_FLAG_OVERLAPPED**. Это действие выполняется в функции открытия порта COMOpen() (описание функции см. **ниже**).

```
HANDLE COMport;
COMport = CreateFile(cnum.c_str(),GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
FILE_FLAG_OVERLAPPED, NULL);
```

Кроме того, для этого необходима структура типа **OVERLAPPED**, которая используется для управления перекрываемыми асинхронными операциями ввода/вывода. В данном случае она объявлена статически.

Важное замечание: структуру OVERLAPPED необходимо объявлять **глобально**, а не внутри потока, иначе программа будет работать некорректно (это проверено).

```
OVERLAPPED overlapped;
```

В этой структуре будем использовать только поле **hEvent**. Другие поля либо зарезервированы для операционной системы, либо не используются при работе с коммуникационными портами, и поэтому нас не интересуют.

Асинхронная операция подразумевает работу по событию, поэтому необходимо создать **сигнальный объект-событие**, который будет устанавливаться в сигнальное состояние по завершении операции, таким образом указывая программе, что перекрываемая операция завершена. Кроме того, этот объект-событие будет указывать функциям ожидания, используемым для приостановки потока, когда следует пробудить поток для обработки полученной информации.

Объект-событие создаётся с помощью функции **CreateEvent**. Эта функция возвращает дескриптор объекта, который и заносится в структуру типа OVERLAPPED:

```
overlapped.hEvent = CreateEvent(NULL, true, true, NULL);
```

Так как работа с портом будет выполняться в асинхронном режиме, для создаваемого события в функции CreateEvent нужно задать следующие значения передаваемых параметров: NULL, true, true, NULL.

Первый параметр, равный **NULL**, означает, что дескриптор создаваемого события не будет наследоваться дочерними процессами и будет использоваться дескриптор безопасности, заданный по умолчанию[*].

Второй параметр, равный **true**, задаёт тип объекта-события **с ручным переключением** в несигнальное состояние[**]. Это необходимо, если мы хотим использовать функцию ожидания, такую как WaitForSingleObject (см. пояснения к параметрам функции CreateEvent).

Значение **true** третьего параметра указывает на то, что объект создаётся **в сигнальном состоянии**. Это нужно для того, чтобы предотвратить зависание программы в случае, если событие, ожидаемое функцией WaitCommEvent, произойдёт сразу после вызова функции (см. **пояснения ниже**).

Четвёртый параметр, равный **NULL**, означает, что объект создаётся без имени. Имя здесь не нужно, так как данный объект-событие используется внутри только одного процесса.

Создаваемый объект-событие связывается с дескриптором **hEvent** в структуре **OVERLAPPED**. Таким образом, перекрываемые операции могут использовать его, чтобы сообщать о своём завершении, установив его в сигнальное состояние, в то время как запустившая их функция может сразу возвратить управление программе, не дожидаясь завершения операции.

Чтобы отслеживать момент, когда в COM-порт пришёл байт, с помощью функции **SetCommMask** связываем с дескриптором порта маску события прихода байта **EV_RXCHAR**.

```
SetCommMask(COMport, EV_RXCHAR);
```

Функции, запускающие перекрываемые операции (такие как ReadFile, WriteFile, WaitCommEvent и т.п.), не дожидаются их завершения и сразу возвращают управление в программу. Если при этом операция не

[*] К сожалению, авторы данной статьи не знают, что представляет собой дескриптор безопасности.
[**] Переключение в несигнальное состояние в случае необходимости можно осуществить с помощью функции **ResetEvent**.

может быть завершена немедленно, функция вернёт FALSE, и система установит объект-событие в несигнальное состояние. Объект установится в сигнальное состояние только по завершении перекрываемой операции.

В данном случае перекрываемую операцию активирует функция **WaitCommEvent**. Она запускает ожидание события порта, заданного маской, и, если событие не наступает немедленно, передаёт управление обратно программе и возвращает FALSE, а система устанавливает объект-событие в **несигнальное** состояние.

```
WaitCommEvent(COMport, &mask, &overlapped);
```

Это позволяет запустить функцию ожидания, которая приостановит поток до того момента, пока перекрываемая операция не завершится, чтобы ожидающий событие поток не занимал процессорное время.

В качестве такой функции в данном случае выступает **WaitForSingleObject**.

```
signal = WaitForSingleObject(overlapped.hEvent, INFINITE);
```

Эта функция будет ждать, пока объект-событие, связанный со структурой OVERLAPPED, из несигнального состояния не перейдёт в сигнальное. Как только объект перейдёт в **сигнальное** состояние, WaitForSingleObject вернёт значение **WAIT_OBJECT_0** и активирует поток.

Кроме WAIT_OBJECT_0 эта функция может вернуть WAIT_FAILED (функция завершилась с ошибкой), WAIT_ABANDONED (связано с передачей не освобожденных мьютексов от завершившегося потока-владельца вызывающему потоку), WAIT_TIMEOUT (время ожидания завершено, а объект не установился в сигнальное состояние). Возникновение третьего случая предотвращаем установкой времени (таймаута) ожидания в INFINITE (переводится с английского как "бесконечно"). Второй случай нас не интересует (так как наш объект не является мьютексом, и такое значение не появится). А от первого случая ограждаемся проверкой возвращаемого значения на соответствие WAIT_OBJECT_0.

```
if(signal == WAIT_OBJECT_0)
```

Таким образом, как только произойдёт установленное маской событие, объект-событие перейдёт в сигнальное состояние и активирует приостановленный поток, который сможет продолжить своё выполнение.

Обратите внимание, что если объект-событие будет создан в **несигнальном** состоянии, а событие произойдёт сразу после запуска операции, объект **не** будет установлен в сигнальное состояние, и функция WaitForSingleObject будет вечно ждать его установки.

Как только поток активирован, функцией **GetOverlappedResult** нужно считать результат операции, запущенной WaitCommEvent. Функция GetOverlappedResult возвращает результат выполнения перекрываемой операции и в параметр mask, переданный в WaitCommEvent, заносит маску произошедших событий.

```
if(GetOverlappedResult(COMport, &overlapped, &temp, true))
```

Последний параметр функции GetOverlappedResult, равный **true**, заставляет её дожидаться, пока перекрываемая операция не завершится полностью.

После вызова GetOverlappedResult с помощью параметра mask следует выполнить проверку, произошло ли именно событие прихода байта.

```
if((mask&EV_RXCHAR)!=0)
```

Но так как в нашем случае отслеживается только одно событие, выполнять такую проверку не обязательно. Поэтому используем GetOverlappedResult только для определения успешности выполнения операции WaitCommEvent, чтобы выполнить чтение байтов только в случае успешного её завершения.

 © PIClist RUS, 2007

Теперь, когда известно, что байты получены, нужно выяснить, какое их количество находится в буфере. Для этого используем функцию **ClearCommError**, которая заполняет поля структуры **COMSTAT**. Одно из полей этой структуры, **cbInQue**, содержит количество байтов данных, содержащихся в буфере, но ещё не считанных функцией ReadFile.

```
ClearCommError(COMport, &temp, &curstat);
btr = curstat.cbInQue;
```

Это количество байтов и будем считывать функцией **ReadFile**.

```
ReadFile(COMport, buf, btr, &temp, &overlapped);
```

Краткое описание алгоритма цикла чтения байтов

Таким образом, должна выполняться следующая последовательность действий:

0) при открытии порта для асинхронных операций функцией CreateFile нужно использовать флаг FILE_FLAG_OVERLAPPED, а также необходима структура OVERLAPPED;

1) функцией CreateEvent создаём сигнальный объект-событие с ручным сбросом в несигнальное состояние;

2) функцией SetCommMask устанавливаем маску ожидаемого события для открытого порта;

3) запускаем цикл, который буде работать все время существования потока;

4) функцией WaitCommEvent запускаем перекрываемую операцию ожидания этого события, при этом сигнальный объект-событие перейдёт в несигнальное состояние;

5) функцией WaitForSingleObject помещаем поток в состояние эффективного ожидания (приостанавливаем) до тех пор, пока не произойдет событие, и объект-событие не установится в сигнальное состояние;

6) когда событие произошло (объект-событие установился в сигнальное состояние), и поток активировался, функцией GetOverlappedResult проверяем результат операции WaitCommEvent. Если результат успешный, выполняем следующие шаги (7, 8 и 9), иначе - переходим на начало цикла (шаг 4);

7) по маске событий, которая передавалась в функцию WaitCommEvent, проверяем, что произошло именно событие прихода байта. Если маска в функции SetCommMask указывала только на одно это событие, проверку можно не выполнять.

8) функцией ClearCommError заполняем структуру COMSTAT и из её поля cbInQue считываем количество доступных для чтения байтов.

9) функцией ReadFile считываем эти байты.

10) переходим на начало цикла

Пояснение параметров функций

Параметры функции CreateEvent

```
CreateEvent(NULL, true, true, NULL);
```

Первый параметр, равный **NULL**, означает, что дескриптор создаваемого события не будет наследоваться дочерними процессами и будет использоваться дескриптор безопасности, заданный по умолчанию*.

Второй параметр, равный **true**, означает, что тип объекта-события – объект с ручным переключением в несигнальное состояние**. Если поставить false, то есть задать тип объекта как объект с автопереключением в несигнальное состояние, то этот объект нельзя будет использовать в функциях ожидания (таких как WaitForSingleObject), находящихся **между перекрываемой операцией и функцией GetOverlappedResult**.

---

* К сожалению, авторы данной статьи не знают, что представляет собой дескриптор безопасности.

** При использовании объектов такого типа переключение объекта в несигнальное состояние осуществляется с помощью функции ResetEvent.

Если объект настроить как автопереключаемый, то WaitForSingleObject по завершении сбросит его в несигнальное состояние, и функция GetOverlappedResult заблокируется. Это произойдёт потому, что последний её параметр, установленный в true, заставляет её ждать до тех пор, пока перекрываемая операция не завершится, то есть сигнальный объект-событие не установится в сигнальное состояние, чего в данном случае не произойдёт. Установка последнего параметра функции GetOverlappedResult в true необходима, чтобы она дождалась завершения операции. Если этот параметр задать false, то если при вызове этой функции операция ещё не завершилась, она вернёт false, а функция GetLastError вернёт код ошибки ERROR_IO_PENDING (операция не завершена). В данном же случае нужно именно дождаться завершения операции, чтобы узнать её результат.

Значение **true** третьего параметра указывает на то, что объект создаётся в сигнальном состоянии. Это необходимо для предотвращения зависания программы, если событие, ожидаемое функцией WaitCommEvent, произойдёт немедленно после её вызова. В таком случае перекрываемая операция завершится сразу и не изменит состояние объекта-события. Это означает, что если он был создан в несигнальном состоянии, то в сигнальное состояние он не установится, и функция WaitForSingleObject будет ждать вечно его установки (то есть программа "зависнет"). А если объект-событие создан в сигнальном состоянии, то при немедленном завершении перекрываемой операции он не будет сброшен, и функция WaitForSingleObject выполнится и активирует поток.

Четвёртый параметр, равный **NULL**, означает, что объект создаётся без имени. Имя не нужно, так как объект-событие используется внутри только одного процесса.

Параметры функции SetCommMask

```
SetCommMask(comport, EV_RXCHAR);
```

Первый параметр – дескриптор открытого порта.

Второй параметр задаёт маску отслеживаемых событий. Может представлять комбинацию из нескольких значений (более подробно см. Builder Help). В данном случае отслеживается только одно событие – событие прихода байта (значение `EV_RXCHAR`).

Параметры функции WaitCommEvent

```
WaitCommEvent(comport, &mask, &overlapped);
```

Первый параметр – дескриптор порта.

Второй параметр – адрес переменной, в которую будет возвращена маска произошедших событий, если после завершения перекрываемой операции WaitCommEvent будет вызвана функция GetOverlappedResult. Причём маска будет содержать только те события, отслеживание которых было разрешено установкой маски функцией SetCommMask, и которые произошли.

Третий параметр – адрес структуры OVERLAPPED. В данном случае он задан, так как функция WaitCommEvent используется как асинхронная перекрываемая операция.

Поток записи байтов

Поток записи байтов в порт немного проще, чем поток чтения. Это связано с тем, что записью в порт мы можем управлять.

Ниже приведён шаблон потока записи в порт, поправки на TThread и WinAPI даны в соответствующих разделах.

```
//-----------------------------------------------------------------------------

//главная функция потока, выполняет передачу байтов из буфера в COM-порт

{
 DWORD temp, signal;       //temp - переменная-заглушка
```

```
 overlappedwr.hEvent = CreateEvent(NULL, true, true, NULL);       //создать событие
 WriteFile(COMport, bufwr, strlen(bufwr), &temp, &overlappedwr);  //записать байты в порт (перекрываемая операция!)
 signal = WaitForSingleObject(overlappedwr.hEvent, INFINITE);     //приостановить поток, пока не завершится перекрываемая операция WriteFile
 if((signal == WAIT_OBJECT_0) && (GetOverlappedResult(COMport, &overlappedwr, &temp, true))) fl = true; //если операция завершилась успешно, установить соответствующий флажок
 else fl = false;

}

//---------------------------------------------------------------------------
```

Так как здесь тоже используется асинхронная перекрываемая операция (запись в порт функцией WriteFile), то в этом случае структура OVERLAPPED также необходима. Так как операции чтения и записи в данном примере могут выполняться параллельно, для операции записи необходимо создать свою структуру **OVERLAPPED**:

```
OVERLAPPED overlappedwr;
```

Её также необходимо объявить глобально.

Как и в потоке чтения, для асинхронной операции записи необходимо создать сигнальный объект-событие с помощью функции **CreateEvent**:

```
overlappedwr.hEvent = CreateEvent(NULL, true, true, NULL);
```

Параметры используются такие же: NULL, true, true, NULL.

Далее запускаем перекрываемую операцию записи функцией **WriteFile**:

```
WriteFile(COMport, bufwr, strlen(bufwr), &temp, &overlappedwr);
```

Здесь в функцию передаются следующие параметры:

`COMport` - дескриптор порта;

`bufwr` - указатель на передающий программный буфер, содержащий данные, которые нужно записать в порт;

`strlen(bufwr)` - количество байтов передаваемых данных, в данном случае – длина строки в буфере;

`&temp` - адрес переменной, в которую будет помещено число фактически записанных байтов;

`&overlappedwr` - адрес структуры OVERLAPPED, содержащей дескриптор сигнального объекта-события, используемого перекрываемой функцией WriteFile.

Чтобы поток не занимал процессорное время, ожидая, пока запущенная перекрываемая операция записи байтов в порт не завершится, его следует приостановить. Для этого также используется функция **WaitForSingleObject**:

```
signal = WaitForSingleObject(overlappedwr.hEvent, INFINITE);
```

Как только перекрываемая операция записи в порт завершится, сигнальный объект-событие с дескриптором overlappedwr.hEvent установится в сигнальное состояние и функция WaitForSingleObject активирует поток. При этом она должна вернуть значение WAIT_OBJECT_0, которое свидетельствует о том, что асинхронная перекрываемая операция завершилась. Любое другое значение, как и в случае с потоком чтения, будет свидетельствовать об ошибке. Если все-таки возвращено значение WAIT_OBJECT_0, то функцией **GetOverlappedResult** проверяем, успешно ли завершилась операция.

```
if((signal == WAIT_OBJECT_0)&&(GetOverlappedResult(COMport, &overlappedwr, &temp, true))) fl = true;
else fl = false;
```

Если успешно, то GetOverlappedResult вернёт true, тогда устанавливаем флажок fl в true. Если же операция завершилась неуспешно (WaitForSingleObject вернула значение, отличное от WAIT_OBJECT_0,

 © PIClist RUS, 2007

либо GetOverlappedResult вернула false, либо оба случая сразу), флажок fl сбрасываем в false. По этому флажку потом можно будет определить, завершилась операция чтения успешно или нет.

На этом поток записи в порт завершается.

Создание потока

Для создания потоков можно использовать любой из двух способов:

1) создание потока с помощью класса **TThread**. В данном случае вы получаете возможность работы с потоком как с объектом – создавать, разрушать, выполнять инициализацию при создании, задавать для него дополнительные методы и свойства и тому подобное.

2) создание потока с помощью функций **WinAPI**. В этом случае поток создаётся в виде функции. То есть при создании потока ему передаётся функция, которую он будет исполнять.

Выбор способа создания потока зависит от конкретной задачи и пожеланий разработчика. В пределах одной программы даже (теоретически) возможно создание разных потоков, используя и первый, и второй способ.

В данном же примере описывается создание всех потоков программы либо только первым, либо только вторым способами.

Создание потока с помощью класса TThread

Сначала рассмотрим создание потока чтения, а затем по аналогии создадим поток записи.

Поток чтения из порта

В программе **C++ Builder** это можно сделать следующим образом: открыть меню **File->New->вкладка New->ThreadObject**. Затем в появившейся форме нужно задать имя нового класса-потока: **ReadThread**.

Появится новый файл с названием Unit2.cpp (или другой порядковый номер в зависимости от количества модулей). Если наличие нескольких модулей приемлемо, то этот файл просто нужно включить в проект через #include.

Если же нужно включить всё в один модуль (как и сделано в демонстрационной программе), то необходимо выполнить следующие действия:

1) Активировать вкладку с Unit2.cpp, если она еще не активирована.

2) Нажав на имя вкладки Unit2.cpp правой кнопкой мыши, выбрать пункт меню "Open Source/Header file" (Ctrl+F6). Затем скопировать из открывшегося файла Unit2.h следующее:

```
//---------------------------------------------------------------------------
class ReadThread : public TThread
{
private:
protected:
        void __fastcall Execute();   //главная функция потока
public:
        __fastcall ReadThread(bool CreateSuspended);     //конструктор потока
};

//---------------------------------------------------------------------------
```

и добавить в главный модуль программы желательно вверху, там, где объявлены переменные и функции (то есть до того, как пользоваться потоком).

Это объявление класса ReadThread. В данном объявлении указывается защищённая функция Execute(), которая является основной исполняемой функцией класса потока, которая используется только этим классом и не видна извне (является закрытой). Открытая функция ReadThread(bool CreateSuspended) – это конструктор класса. О нём сказано ниже.

3) из файла Unit2.cpp скопировать следующее (поместить тоже до использования потоков):

```
//---------------------------------------------------------------------------
__fastcall ReadThread::ReadThread(bool CreateSuspended) //конструктор потока, по умолчанию - пустой
        : TThread(CreateSuspended)
{
}
//---------------------------------------------------------------------------
```

Это конструктор класса ReadThread. По умолчанию он является пустым, но при необходимости в него можно добавить какие-либо действия, которые будут выполняться при создании потока. (Создаваемый поток является объектом класса). Параметр CreateSuspended (создать в остановленном состоянии) определяет, будет ли запущен поток сразу после создания (CreateSuspended = false), либо только после вызова метода Resume() (если CreateSuspended = true).

```
//---------------------------------------------------------------------------
void __fastcall ReadThread::Execute()
{
 //здесь вставляется код, который будет исполняться потоком
}
//---------------------------------------------------------------------------
```

Это – главная функция потока. Она начинает выполняться, когда поток запускается. В эту функцию и нужно добавить основной рабочий код потока. В данном случае – считывание данных из COM-порта:

```
//---------------------------------------------------------------------------

void __fastcall ReadThread::Execute()
{
 OVERLAPPED over;
 COMSTAT curstat;
 DWORD btr, temp, temp, mask, signal;    //переменная temp используется в качестве заглушки

 over.hEvent = CreateEvent(NULL, true, true, NULL);            //создать событие; true,true - для асинхронных операций
 SetCommMask(comport, EV_RXCHAR);                      //маска = если принят байт
 while(!Terminated)                                    //пока поток не будет прерван, выполняем цикл
  {WaitCommEvent(comport, &mask, &over);               //ожидать события приема байта
   signal = WaitForSingleObject(over.hEvent, INFINITE); //усыпить поток до прихода байта
   if(signal == WAIT_OBJECT_0)                         //если событие прихода байта произошло
    {if(GetOverlappedResult(comport, &over, &temp, true))      //проверяем, успешно ли завершилась перекрываемая операция WaitCommEvent
      if((mask&EV_RXCHAR)!=0)                                  //если произошло именно событие прихода байта
       {ClearCommError(comport, &temp, &curstat);              //нужно заполнить структуру COMSTAT
        btr = curstat.cbInQue;                         //получить количество принятых байтов
        if(btr)                                        //если в буфере порта есть непрочитанные байты
          ReadFile(comport, buf, btr, &temp, &over);   //прочитать байты из порта в буфер программы
       }
    }
  }
 CloseHandle(over.hEvent);       //закрыть объект-событие
}

//---------------------------------------------------------------------------
```

В случае использования потоков TThread в качестве условия в цикле ожидания и чтения байтов используется **(!Terminated)**. Это означает, что данный цикл будет выполняться, пока поток не будет прерван с помощью метода **Terminate()**. Этот метод устанавливает свойство потока Terminated в true, таким образом сообщая потоку, что он должен завершиться как только это возможно. Проверяя этот флаг, поток может определить, когда он должен завершиться, и выполнить перед завершением какие-нибудь действия. В данном случае, получив запрос на завершение, поток выходит из цикла чтения байта (так как Terminated становится равен true) и освобождает дескриптор события hEvent в структуре типа OVERLAPPED, которая использовалась при асинхронных операциях. (Напомним, что этот объект событие был создан с помощью функции CreateEvent). После чего поток прекращает выполнение.

Использование метода Synchronize

© PIClist RUS, 2007

В случае, когда поток делит графические компоненты и файлы с другими потоками, обращение к ним может привести к конфликту между потоками. Во избежание этих конфликтов используется метод **Synchronize(TThreadMethod &Method)**, который использует для обращения к компонентам очередь сообщений главного потока. Он выполняет переданный в него метод в главном потоке (этот поток отвечает за работу с графическими компонентами), причём поток, вызвавший метод Synchronize, приостанавливается на время выполнения этого метода.

Чтобы обратиться к графическим компонентам или файлу, сначала нужно создать отдельный метод потока и объявить его в объявлении класса:

```
//---------------------------------------------------------------------------

class ReadThread : public TThread
{
private:
protected:
        void __fastcall Execute();
        void __fastcall Printing();  //добавлено вручную для использования в Synchronize
public:
        __fastcall ReadThread(bool CreateSuspended);
};

//---------------------------------------------------------------------------
```

И написать соответствующую функцию. Например:

```
//---------------------------------------------------------------------------

//выводим принятые байты на экран и в файл (если включено)
void __fastcall ReadThread::Printing()
{

 Form1->Memo1->Lines->Add((char*)bufrd); //выводим принятую строку в Memo
 Form1->StatusBar1->Panels->Items[2]->Text = "Всего принято " + IntToStr(counter) + " байт"; //выводим счётчик в строке состояния

 if(Form1->CheckBox3->Checked == true)  //если включен режим вывода в файл
  {
   write(handle, bufrd, strlen(bufrd)); //записать в файл данные из приёмного буфера
  }
 memset(bufrd, 0, BUFSIZE);              //очистить буфер (чтобы данные не накладывались друг на друга)
}

//---------------------------------------------------------------------------
```

Эта функция выполняет вывод текущего состояния на форму и запись принимаемых данных в файл.

После этого в функции Execute() потока добавить метод Synchronize(функция) в месте, где требуется вызвать функцию, работающую с графическими компонентами или файлами. Например:

```
//---------------------------------------------------------------------------
void __fastcall ReadThread::Execute()
{

...

Synchronize(Printing);
}
//---------------------------------------------------------------------------
```

Как использовать поток

1) *Класс* – это тип данных, определяемый пользователем. *Объект* – конкретный экземпляр данного класса. Это аналогично типу данных (например, int) и переменной данного типа (например, i).

То есть перед тем, как можно будет создать и запустить поток, необходимо объявить объект класса, то есть некую переменную типа ReadThread, например:

```
ReadThread *reader;     //объект потока ReadThread
```

Нужно сделать ее глобальной, чтобы иметь возможность доступа к ней из других функций. Сам класс ReadThread должен быть объявлен раньше.

2) Теперь нужно создать поток:

```
reader = new ReadThread(false);      //создать и запустить поток чтения байтов
reader->FreeOnTerminate = true;  //установить это свойство потока, чтобы он автоматически уничтожался после завершения
```

В первой строке мы создаём объект класса ReadThread. Для этого вызывается конструктор потока ReadThread(), которому в качестве аргумента передаётся значение false. Это означает, что поток начинает работать сразу после создания (см. **описание конструктора выше**). Почему так сделано? Потому что в данной версии программы поток создаётся в функции открытия порта COMOpen(), то есть когда нужно начинать считывать байты (сразу после открытия порта).

Следует отметить, что потоки создаются только динамическим образом, то есть используя указатель (*reader) и оператор new (new ReadThread(false)).

Далее установим свойство потока **FreeOnTerminate** в true, чтобы объект потока освобождался после завершения его работы, так как здесь нам не нужно освобождать его вручную.

То есть в целом создание потока будет выглядеть так:

```
ReadThread *reader;     //объект потока ReadThread
reader = new ReadThread(false);      //создать и запустить поток чтения байтов
reader->FreeOnTerminate = true;  //установить это свойство потока, чтобы он автоматически уничтожался после завершения
```

После того, как поток больше не нужен, его следует завершить, используя метод **Terminate()**. Как говорилось выше, этот метод устанавливает флаг потока Terminated в true, сообщая потоку, что он должен завершиться.

То есть, вызываем завершение потока таким образом:

```
if(reader)reader->Terminate();
```

Проверка if(reader) необходима, если её не делать, будут возникать ошибки (это проверено).

Уничтожение потока в первой версии программы производится в функции закрытия порта COMClose(). Таким образом, при открытии порта необходимо создавать поток заново, как и было реализовано в функции COMOpen() в этой версии программы (см. **ниже**).

Для обработки завершения потока можно определить обработчик события OnTerminate, который вызывается между окончанием выполнения потока (когда он выполнил return) и его разрушением. В этом обработчике можно свободно работать с графическими компонентами, так как он исполняется в контексте основного потока программы. То есть в этом обработчике можно определить некоторый код, который, например, будет освобождать объект потока или выводить на форму какое-то сообщение.

Полностью код потока чтения выглядит так:

```
//---------------------------------------------------------------------------

//поток для чтения последовательности байтов из COM-порта в буфер
class ReadThread : public TThread
{
 private:
        void __fastcall Printing();      //вывод принятых байтов на экран и в файл
 protected:
        void __fastcall Execute();       //основная функция потока
 public:
        __fastcall ReadThread(bool CreateSuspended);    //конструктор потока
};
```

 © PIClist RUS, 2007

```
//---------------------------------------------------------------------------

//-----------------------------------------------------------------------------
//.............................. поток ReadThead ............................
//-----------------------------------------------------------------------------

ReadThread *reader;     //объект потока ReadThread

//---------------------------------------------------------------------------

//конструктор потока ReadThread, по умолчанию пустой
__fastcall ReadThread::ReadThread(bool CreateSuspended) : TThread(CreateSuspended)
{}

//---------------------------------------------------------------------------

//главная функция потока, реализует приём байтов из COM-порта
void __fastcall ReadThread::Execute()
{
 COMSTAT comstat;          //структура текущего состояния порта, в данной программе используется для определения
количества принятых в порт байтов
 DWORD btr, temp, mask, signal;   //переменная temp используется в качестве заглушки

 overlapped.hEvent = CreateEvent(NULL, true, true, NULL);       //создать сигнальный объект-событие для
асинхронных операций
 SetCommMask(COMport, EV_RXCHAR);                               //установить маску на срабатывание по событию
приёма байта в порт
 while(!Terminated)                                     //пока поток не будет прерван, выполняем цикл
  {
   WaitCommEvent(COMport, &mask, &overlapped);                  //ожидать события приёма байта (это и есть
перекрываемая операция)
   signal = WaitForSingleObject(overlapped.hEvent, INFINITE);  //приостановить поток до прихода байта
   if(signal == WAIT_OBJECT_0)                                  //если событие прихода байта произошло
    {
     if(GetOverlappedResult(COMport, &overlapped, &temp, true)) //проверяем, успешно ли завершилась перекрываемая
операция WaitCommEvent
      if((mask & EV_RXCHAR)!=0)                                 //если произошло именно событие прихода байта
       {
        ClearCommError(COMport, &temp, &comstat);               //нужно заполнить структуру COMSTAT
        btr = comstat.cbInQue;                          //и получить из неё количество принятых байтов
        if(btr)                                         //если действительно есть байты для чтения
        {
         ReadFile(COMport, bufrd, btr, &temp, &overlapped);     //прочитать байты из порта в буфер программы
         counter+=btr;                                          //увеличиваем счётчик байтов
         Synchronize(Printing);                                 //вызываем функцию для вывода данных на экран и в
файл
        }
       }
    }
  }
 CloseHandle(overlapped.hEvent);          //перед выходом из потока закрыть объект-событие
}

//---------------------------------------------------------------------------

//выводим принятые байты на экран и в файл (если включено)
void __fastcall ReadThread::Printing()
{

 Form1->Memo1->Lines->Add((char*)bufrd); //выводим принятую строку в Memo
 Form1->StatusBar1->Panels->Items[2]->Text = "Всего принято " + IntToStr(counter) + " байт"; //выводим счётчик в
строке состояния

 if(Form1->CheckBox3->Checked == true)  //если включен режим вывода в файл
  {
   write(handle, bufrd, strlen(bufrd)); //записать в файл данные из приёмного буфера
  }
 memset(bufrd, 0, BUFSIZE);              //очистить буфер (чтобы данные не накладывались друг на друга)
}

//---------------------------------------------------------------------------
```

Поток записи в порт

Поток записи в порт создаётся аналогично потоку чтения.
Точно так же как и для потока чтения, нужно создать объявление класса и для потока записи.

 © PIClist RUS, 2007

```
//---------------------------------------------------------------------------

//поток для записи последовательности байтов из буфера в COM-порт
class WriteThread : public TThread
{
private:
    void __fastcall Printing();      //вывод состояния на экран
protected:
        void __fastcall Execute();      //основная функция потока
public:
        __fastcall WriteThread(bool CreateSuspended);    //конструктор потока
};

//---------------------------------------------------------------------------
```

Здесь точно так же присутствуют конструктор класса потока записи WriteThread(bool CreateSuspended), главная функция потока Execute() и дополнительная функция Printing(), которая используется для вывода результатов операции в строке состояния.

Основная функция потока Execute() выполняет передачу данных в порт:

```
//---------------------------------------------------------------------------

//главная функция потока, выполняет передачу байтов из буфера в COM-порт
void __fastcall WriteThread::Execute()
{
 DWORD temp, signal;       //temp - переменная-заглушка

 overlappedwr.hEvent = CreateEvent(NULL, true, true, NULL);      //создать событие
 WriteFile(COMport, bufwr, strlen(bufwr), &temp, &overlappedwr);  //записать байты в порт (перекрываемая
операция!)
 signal = WaitForSingleObject(overlappedwr.hEvent, INFINITE);     //приостановить поток, пока не завершится
перекрываемая операция WriteFile
 if((signal == WAIT_OBJECT_0) && (GetOverlappedResult(COMport, &overlappedwr, &temp, true))) fl = true; //если
операция завершилась успешно, установить соответствующий флажок
 else fl = false;

 Synchronize(Printing);    //вывести состояние операции в строке состояния
 CloseHandle(overlappedwr.hEvent);       //перед выходом из потока закрыть объект-событие
}

//---------------------------------------------------------------------------
```

Функция Printing() выполняет вывод состояния операции в строке состояния:

```
//---------------------------------------------------------------------------

//вывод состояния передачи данных на экран
void __fastcall WriteThread::Printing()
{
 if(!fl)   //проверяем состояние флажка
  {
   Form1->StatusBar1->Panels->Items[0]->Text  = "Ошибка передачи";
   return;
  }
 Form1->StatusBar1->Panels->Items[0]->Text  = "Передача прошла успешно";
}

//---------------------------------------------------------------------------
```

Как и в случае с потоком чтения, для работы с потоком записи его сначала нужно создать.
Для этого выполняется следующее:

1) после объявления класса потока записи объявляется объект потока записи:

```
WriteThread *writer;     //объект потока WriteThread
```

Этот объект также должен быть объявлен глобально.

2) создать объект класса с помощью оператора new:

```
writer = new WriteThread(false);   //активировать поток записи данных в порт
writer->FreeOnTerminate = true;    //установить это свойство, чтобы поток автоматически уничтожался после завершения
```

Здесь также после создания потока устанавливается в true свойство FreeOnTerminate, чтобы поток автоматически уничтожался после завершения.

В программе поток создаётся при необходимости отправки данных – в обработчике нажатия кнопки "Передать":

```
//---------------------------------------------------------------------------

//кнопка "Передать"
void __fastcall TForm1::Button1Click(TObject *Sender)
{
 memset(bufwr,0,BUFSIZE);                    //очистить программный передающий буфер, чтобы данные не накладывались друг на друга
 PurgeComm(COMport, PURGE_TXCLEAR);              //очистить передающий буфер порта
 strcpy(bufwr,Form1->Edit1->Text.c_str());       //занести в программный передающий буфер строку из Edit1

 writer = new WriteThread(false);                //создать и активировать поток записи данных в порт
 writer->FreeOnTerminate = true;                 //установить это свойство, чтобы поток автоматически уничтожался после завершения

}

//---------------------------------------------------------------------------
```

Когда поток больше не нужен, он также уничтожается вызовом метода Terminate():

```
if(writer)writer->Terminate();       //если поток записи работает, завершить его; проверка if(writer) обязательна, иначе возникают ошибки
```

А полностью код потока записи в порт выглядит так:

```
//---------------------------------------------------------------------------

//поток для записи последовательности байтов из буфера в COM-порт
class WriteThread : public TThread
{
private:
    void __fastcall Printing();   //вывод состояния на экран
protected:
        void __fastcall Execute();      //основная функция потока
public:
        __fastcall WriteThread(bool CreateSuspended);   //конструктор потока
};

//---------------------------------------------------------------------------

//-----------------------------------------------------------------------------
//.............................. поток WriteThead ...........................
//-----------------------------------------------------------------------------

WriteThread *writer;     //объект потока WriteThread

//---------------------------------------------------------------------------

//конструктор потока WriteThread, по умолчанию пустой
__fastcall WriteThread::WriteThread(bool CreateSuspended) : TThread(CreateSuspended)
{}

//---------------------------------------------------------------------------

//главная функция потока, выполняет передачу байтов из буфера в COM-порт
void __fastcall WriteThread::Execute()
{
 DWORD temp, signal;      //temp - переменная-заглушка
```

 © PIClist RUS, 2007

```
 overlappedwr.hEvent = CreateEvent(NULL, true, true, NULL);      //создать событие
 WriteFile(COMport, bufwr, strlen(bufwr), &temp, &overlappedwr);  //записать байты в порт (перекрываемая
операция!)
 signal = WaitForSingleObject(overlappedwr.hEvent, INFINITE);     //приостановить поток, пока не завершится
перекрываемая операция WriteFile
 if((signal == WAIT_OBJECT_0) && (GetOverlappedResult(COMport, &overlappedwr, &temp, true))) fl = true; //если
операция завершилась успешно, установить соответствующий флажок
 else fl = false;

 Synchronize(Printing);    //вывести состояние операции в строке состояния
 CloseHandle(overlappedwr.hEvent);       //перед выходом из потока закрыть объект-событие
}

//---------------------------------------------------------------------------

//вывод состояния передачи данных на экран
void __fastcall WriteThread::Printing()
{
 if(!fl)    //проверяем состояние флажка
  {
   Form1->StatusBar1->Panels->Items[0]->Text  = "Ошибка передачи";
   return;
  }
 Form1->StatusBar1->Panels->Items[0]->Text  = "Передача прошла успешно";
}

//---------------------------------------------------------------------------
```

Использование методов Resume() и Suspend()

Вторая версия программы с использованием TThread демонстрирует использование методов **Resume()** и **Suspend()** для запуска и остановки потоков. В целом эта версия программы практически аналогична первой, все отличия связаны с использованием этих методов для управления потоком записи. Рассмотрим эти отличия:

1) В данном случае поток записи создаётся непосредственно в функции открытия порта COMOpen() после всех действий по открытию и инициализации порта (в конце функции), вместе с потоком чтения. При этом в конструктор потока чтения ReadThread(), как и в первой версии программы, передаётся параметр false, чтобы запустить поток чтения сразу после создания. А вот в конструктор потока записи WriteThread() передаётся параметр true, чтобы поток создался в остановленном состоянии. Будем запускать его по мере необходимости.

```
reader = new ReadThread(false);       //создать и запустить поток чтения байтов
reader->FreeOnTerminate = true;  //установить это свойство потока, чтобы он автоматически уничтожался
после завершения

writer = new WriteThread(true);  //создать поток записи данных в порт
writer->FreeOnTerminate = true;  //установить это свойство, чтобы поток автоматически уничтожался
после завершения
```

2) Главное отличие этой версии в том, что поток записи не разрушается после отправки данных, а существует в течение всего времени, пока открыт порт. Благодаря этому не нужно создавать новый поток, а затем разрушать его, каждый раз, когда требуется отправить данные. Вместо этого поток приостанавливается на то время, пока он не нужен, и активируется, когда необходимо передать данные.

Таким образом, поток записи writer будем включать, когда необходимо передать данные, то есть при нажатии кнопки "Передать". Для этого используется метод **Resume()** (переводится с английского как "возобновить"):

```
//кнопка "Передать"
void __fastcall TForm1::Button1Click(TObject *Sender)
{
 memset(bufwr,0,BUFSIZE);                     //очистить программный передающий буфер, чтобы данные не
накладывались друг на друга
 PurgeComm(COMport, PURGE_TXCLEAR);             //очистить передающий буфер порта
 strcpy(bufwr,Form1->Edit1->Text.c_str());      //занести в программный передающий буфер строку из
Edit1
 writer->Resume();      //активировать поток записи в порт
```

 © PIClist RUS, 2007

```
}
```

Но каким-то образом нужно выключить поток, когда он больше не нужен. Самый лучший способ – поручить это самому потоку. То есть поток после выполнения передачи данных будет останавливать сам себя. Это поможет избежать повторной передачи одних и тех же данных.

Для этого построим код потока следующим образом:

```
//---------------------------------------------------------------------------

//главная функция потока, выполняет передачу байтов из буфера в COM-порт
void __fastcall WriteThread::Execute()
{
 DWORD temp, signal;      //temp - переменная-заглушка

 overlappedwr.hEvent = CreateEvent(NULL, true, true, NULL);      //создать событие

 while(!Terminated)     //пока поток не будет завершён, выполнять цикл
  {
  WriteFile(COMport, bufwr, strlen(bufwr), &temp, &overlappedwr);  //записать байты в порт (перекрываемая операция!)
  signal = WaitForSingleObject(overlappedwr.hEvent, INFINITE);    //приостановить поток, ппока не завершится перекрываемая операция WriteFile
  if((signal == WAIT_OBJECT_0) && (GetOverlappedResult(COMport, &overlappedwr, &temp, true))) fl = true; //если операция завершилась успешно, установить соответствующий флажок
  else fl = false;

  Synchronize(Printing);  //вывести состояние операции в строке состояния
  }
 CloseHandle(overlappedwr.hEvent);       //перед выходом из потока закрыть объект-событие
}

//---------------------------------------------------------------------------
```

В данном случае используется бесконечный цикл while(!Terminated), который будет выполняться, пока потоку не будет передана команда завершения (с помощью метода потока Terminate()). Это сделано для того, чтобы поток сам по себе не завершился после отправки данных, как это было сделано в первой версии программы. В первой версии поток просто последовательно выполнял все команды и завершался, когда они заканчивались. И каждый раз при необходимости отправить данные создавался новый поток. В этой же версии нам нужно, чтобы поток существовал всё время, пока открыт порт. Поэтому в нём создаётся бесконечный цикл.

В цикле, как и в первой версии программы, выполняются операции по передаче данных в порт, после чего вызывается функция потока Printing для вывода результата операции в строке состояния (Synchronize(Printing)). Именно в этой функции мы и будем останавливать поток, так как если мы остановим его сразу после записи данных в порт, но перед вызовом функции Printing, то мы сможем увидеть результат операции в строке состояния только при следующем запуске потока, так как поток запустится с точки останова. Поэтому и делаем останов потока в функции Printing следующим образом (для этого используем метод потока **Suspend()** (переводится с английского как "приостановить")):

```
//---------------------------------------------------------------------------

//вывод состояния передачи данных на экран
void __fastcall WriteThread::Printing()
{
 if(!fl)  //проверяем состояние флажка
  {
   Form1->StatusBar1->Panels->Items[0]->Text  = "Ошибка передачи";
   return;
  }
 Form1->StatusBar1->Panels->Items[0]->Text  = "Передача прошла успешно";

 writer->Suspend();     //приостановить поток записи в порт, пока он не потребуется снова

}

//---------------------------------------------------------------------------
```

 © PIClist RUS, 2007

Метод Suspend() останавливает выполнение потока на некоторое время, пока не понадобится возобновить его работу с помощью метода Resume().

Когда после остановки мы снова возобновим поток с помощью метода Resume(), он продолжит выполнение с точки останова, то есть перейдёт на следующий оператор после writer->Suspend(), то есть вернёт управление функции Execute() и перейдёт на начало цикла while(!Terminated), после чего выполнит запись данных в порт, вывод результата в строке состояния и снова остановится.

Была обнаружена интересная особенность – если метод Suspend() для потока записи вызывался в функции Printing, то информация о результате операции, а также введённая в функцию в самом конце тестовая строка выводились независимо от того, в каком месте находился вызов метода. То есть работало даже так:

```
//вывод состояния передачи данных на экран
void __fastcall WriteThread::Printing()
{
 writer->Suspend();     //приостановить поток записи в порт, пока он не потребуется снова
 if(!fl)  //проверяем состояние флажка
  {
   Form1->StatusBar1->Panels->Items[0]->Text  = "Ошибка передачи";
   return;
  }
 Form1->StatusBar1->Panels->Items[0]->Text  = "Передача прошла успешно";
 Form1->Memo1->Lines->Add("Тест!");
}
```

и так тоже:

```
//вывод состояния передачи данных на экран
void __fastcall WriteThread::Printing()
{
 if(!fl)  //проверяем состояние флажка
  {
   Form1->StatusBar1->Panels->Items[0]->Text  = "Ошибка передачи";
   return;
  }
 Form1->StatusBar1->Panels->Items[0]->Text  = "Передача прошла успешно";
 writer->Suspend();     //приостановить поток записи в порт, пока он не потребуется снова
 Form1->Memo1->Lines->Add("Тест!");
}
```

Это связано с тем, что метод Synchronize выполняет переданный ему метод **в контексте главного потока** приложения, а поток, вызвавший Synchronize, на это время **приостанавливается**. Поэтому и не имеет значения, в каком месте функции выполнять останов потока, если функция вызывается с помощью метода Synchronize.

Но вот если поставить вызов метода Suspend в **главной** функции потока следующим образом:

```
while(!Terminated)     //пока поток не будет завершён, выполнять цикл
 {
  ............
  writer->Suspend();    //приостановить поток записи в порт, пока он не потребуется снова
  Synchronize(Printing);      //вывести состояние операции в строке состояния
 }
```

то в данном случае функция Printing выполнялась только после повторного запуска потока, что доказывает, что после вызова метода Resume() поток продолжает своё выполнение с точки останова.

Аналогичный эффект можно достичь следующим образом: в функции Printing вызвать метод Suspend()

```
//вывод состояния передачи данных на экран
void __fastcall WriteThread::Printing()
{
 if(!fl)  //проверяем состояние флажка
  {
   Form1->StatusBar1->Panels->Items[0]->Text  = "Ошибка передачи";
```

```
     return;
    }
   Form1->StatusBar1->Panels->Items[0]->Text  = "Передача прошла успешно";

   writer->Suspend();     //приостановить поток записи в порт, пока он не потребуется снова
  }
```

а в главной функции потока добавить тестовую строку после вызова функции Printing:

```
//главная функция потока, выполняет передачу байтов из буфера в COM-порт
void __fastcall WriteThread::Execute()
{
 while(!Terminated)     //пока поток не будет завершён, выполнять цикл
 {
  ..........
  Synchronize(Printing);      //вывести состояние операции в строке состояния
  Form1->Memo1->Lines->Add("Тест!");
 }
 CloseHandle(overlappedwr.hEvent);         //перед выходом из потока закрыть объект-событие
}
```

В этом случае информация о состоянии операции выведется в панель состояния, а вот тестовая строка появится только при следующем запуске потока.

Поэтому вызов метода Suspend лучше поместить так, чтобы поток выполнил всю необходимую работу (передача данных и вывод результата операции в строке состояния), и только потом был остановлен, чтобы при последующем запуске сразу перейти на начало цикла и выполнить такую же работу.

И всё же лучше всего вызов метода Suspend() поместить в конце цикла в главной функции, после вызова функции Printing:

```
//главная функция потока, выполняет передачу байтов из буфера в COM-порт
void __fastcall WriteThread::Execute()
{
 while(!Terminated)     //пока поток не будет завершён, выполнять цикл
 {
  ..........
  Synchronize(Printing);      //вывести состояние операции в строке состояния
  writer->Suspend();    //приостановить поток записи в порт, пока он не потребуется снова
 }
 CloseHandle(overlappedwr.hEvent);         //перед выходом из потока закрыть объект-событие
}
```

Вы можете сами сделать так, как вам удобнее. Этот случай лучше, потому что здесь останов потока происходит именно в конце цикла, а не в функции. Иначе, если вам понадобится добавить ещё какую-нибудь функцию после функции Printing, то вызов этого метода вам придётся переносить в новую функцию. Это не очень удобно и вносит некоторую путаницу.

3) Интересный момент – это уничтожение потока методом Terminate(). Вот как выглядит код завершения потока в случае использования данных методов:

```
//если поток записи существует, подать ему команду на завершение и запустить его, чтобы он выполнил
завершение
 if(writer) //проверка if(writer) обязательна, иначе возникают ошибки
  {
   writer->Terminate();
   writer->Resume();
  }
```

Так как поток записи активируется только когда необходимо передать данные, велика вероятность, что когда потребуется его завершить, поток будет находиться в остановленном состоянии и не сможет выполнить завершение. Тогда после вызова метода Terminate(), который установит в true свойство потока Terminated, необходимо вызвать метод Resume(), чтобы активировать поток. В этом случае поток сможет выполнить проверку на значение Terminated=true (в цикле while(!Terminated)) и завершиться.

В качестве особенностей описанных здесь методов можно отметить, что они имеют свойство накладываться. То есть если вы вызвали метод Suspend() несколько раз, то чтобы возобновить работу потока, вам нужно будет столько же раз вызвать метод Resume().

Создание потока средствами WinAPI

Сначала рассмотрим создание потока средствами WINAPI на примере потока чтения данных из порта, а потом по аналогии создадим поток записи.

Создание потока чтения

Перед тем как создать поток, используя средства WinAPI, необходимо объявить и создать функцию, которая будет выполняться потоком. Эта функция имеет прототип

```
DWORD WINAPI ThreadFunc(LPVOID);
```

где ThreadFunc – имя функции, задаваемое пользователем. Например:

```
DWORD WINAPI ReadThread(LPVOID);
```

В эту функцию и запишем основной код потока чтения:

```
//---------------------------------------------------------------------------

//главная функция потока, реализует приём байтов из COM-порта
DWORD WINAPI ReadThread(LPVOID)
{
 COMSTAT comstat;          //структура текущего состояния порта, в данной программе используется для определения количества принятых в порт байтов
 DWORD btr, temp, mask, signal;   //переменная temp используется в качестве заглушки

 overlapped.hEvent = CreateEvent(NULL, true, true, NULL);      //создать сигнальный объект-событие для асинхронных операций
 SetCommMask(COMport, EV_RXCHAR);                              //установить маску на срабатывание по событию приёма байта в порт
 while(1)                                  //пока поток не будет прерван, выполняем цикл
  {
   WaitCommEvent(COMport, &mask, &overlapped);                 //ожидать события приёма байта (это и есть перекрываемая операция)
   signal = WaitForSingleObject(overlapped.hEvent, INFINITE);  //приостановить поток до прихода байта
   if(signal == WAIT_OBJECT_0)                                 //если событие прихода байта произошло
    {
     if(GetOverlappedResult(COMport, &overlapped, &temp, true)) //проверяем, успешно ли завершилась перекрываемая операция WaitCommEvent
      if((mask & EV_RXCHAR)!=0)                                //если произошло именно событие прихода байта
       {
        ClearCommError(COMport, &temp, &comstat);              //нужно заполнить структуру COMSTAT
        btr = comstat.cbInQue;                         //и получить из неё количество принятых байтов
        if(btr)                                        //если действительно есть байты для чтения
         {
         ReadFile(COMport, bufrd, btr, &temp, &overlapped);     //прочитать байты из порта в буфер программы
         counter+=btr;                                          //увеличиваем счётчик байтов
         ReadPrinting();                               //вызываем функцию для вывода данных на экран и в файл
         }
       }
    }
  }
}

//---------------------------------------------------------------------------
```

Здесь цикл реализуется как while(1), то есть бесконечный цикл. Это связано с особенностями завершения потоков, которые будут рассмотрены ниже. Сам же алгоритм потока чтения при использовании WINAPI в целом практически больше ничем не отличается от алгоритма, реализованного на TThread, за исключением

разве только использования дополнительных функций (наподобие Printing() в TThread, об этом подробнее сказано ниже).

Сам поток создаётся функцией **CreateThread**. При успешном завершении эта функция возвращает дескриптор потока, который необходимо запомнить в переменную типа **HANDLE**. Например:

```
HANDLE reader;
reader = CreateThread(NULL, 0, ReadThread, NULL, 0, NULL);
```

В вызове функции используются следующие параметры:

- первый параметр, равный NULL означает, что возвращаемый дескриптор не наследуется и используется дескриптор безопасности по умолчанию (то есть аналогично первому параметру функции CreateEvent).

- второй параметр, равный 0, представляет собой размер стека потока в байтах. Нулевая величина стека означает, что используется величина стека по умолчанию, в качестве которой выступает размер стека главного потока процесса.

- третий параметр является указателем на функцию, которую будет выполнять поток. В данном случае это функция ReadThread. Следует заметить, что функция должна быть объявлена с соглашением вызова WINAPI* и возвращать 32-битный код выхода (параметр типа DWORD). Прототип функции см. выше.

- четвёртый параметр, равный NULL, - это 32-битное значение параметра, передаваемого в поток. В данном случае этот параметр не используется, поэтому равен NULL.

- пятый параметр, равный 0, означает, что поток запускается сразу после создания.

- шестой параметр представляет собой указатель на 32-битную переменную (типа DWORD), в которую будет помещён идентификатор создаваемого потока. Так как нам идентификатор потока не нужен, в качестве этого параметра указываем NULL.

Поток чтения создаётся в функции открытия порта COMOpen.

Теперь обратим внимание на использование дополнительной функции для вывода принятых байтов в файл, а результатов операции – в строку состояния. Основное отличие WINAPI состоит в том, что в этом случае можно работать с графическими компонентами и файлами сразу в главной функции потока, так как при использовании WINAPI все сообщения поступают в очередь главного потока программы и им же обрабатываются. Поэтому вывод в файл и в строку состояния можно включить прямо в цикл чтения байтов потока чтения. Но в данном случае мы оформили вывод в отдельную функцию ReadPrinting(), чтобы сохранить наглядность кода потока чтения и сделать его отличие от кода потока чтения, использующего TThread, наименьшим. Функция ReadPrinting() представляет собой обычную функцию, которая объявляется и вызывается также как и обычная функция.

```
void ReadPrinting(void);

//---------------------------------------------------------------------------

//выводим принятые байты на экран и в файл (если включено)
void ReadPrinting()
{
 Form1->Memo1->Lines->Add((char*)bufrd);  //выводим принятую строку в Memo
 Form1->StatusBar1->Panels->Items[2]->Text = "Всего принято " + IntToStr(counter) + " байт"; //выводим счётчик в строке состояния

 if(Form1->CheckBox3->Checked == true)  //если включен режим вывода в файл
  {
   write(handle, bufrd, strlen(bufrd)); //записать в файл данные из приёмного буфера
  }
 memset(bufrd, 0, BUFSIZE);              //очистить буфер (чтобы данные не накладывались друг на друга)
}
//---------------------------------------------------------------------------
```

После того как поток больше не нужен, его следует завершить. Поток чтения завершается при закрытии порта, в функции COMClose(). Для этого используется функция TerminateThread().

* К сожалению, авторы данной статьи не знают, что это такое.

```
BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode);
```

Первый её параметр – дескриптор завершаемого потока, второй параметр – 32-х битный код завершения (выхода) потока.

Например:

```
TerminateThread(reader,0);
```

Прерывание потока не обязательно удалит объект потока. Для того чтобы объект потока был удалён, необходимо, чтобы все дескрипторы потока были освобождены.

Следует отметить, что при прерывании потока, созданного средствами WINAPI, поток прерывается жёстко, и это может произойти в любом месте его выполнения, а значит, освобождать дескриптор сигнального объекта-события внутри кода потока, как это сделано в программе, использующей класс TThread, в данном случае не очень надёжный способ. (К тому же внутри потока используется бесконечный цикл while(1).) Поэтому, используя функцию TerminateThread(), после неё нужно дополнительно освобождать дескриптор объекта события, находящийся в структуре типа OVERLAPPED, связанной с потоком, а также ещё вручную освобождать и дескриптор потока.

```
if(reader)          //если поток чтения работает, завершить его; проверка if(reader) обязательна, иначе возникают ошибки
  {TerminateThread(reader,0);
   CloseHandle(overlapped.hEvent);   //нужно закрыть объект-событие
   CloseHandle(reader);
  }
```

По этой причине внутри потока используется бесконечный цикл while(1), а не цикл с условием проверки, прерван поток или нет, как это делается при использовании класса потоков TThread.

В принципе использование функции TerminateThread() – это грубый метод завершения потока. Здесь его можно применить, так как поток выполняет простую работу, не связанную с критическими секциями, ядром (kernel32) или динамическими библиотеками (DLL). Но для серьёзных проектов, использующих именно эти методы работы, TerminateThread не подойдёт.

Теоретически можно сделать по аналогии со свойством Terminated потоков TThread. То есть создать глобальную переменную-флаг (например, flag), которую устанавливать в единицу, когда нужно выполнить завершение потока, а в цикле самого потока поставить вместо while(1) условие while(!flag). Тогда получится мягкое завершение потока, как и в случае с TThread и Terminated, то есть можно будет выполнить какой-нибудь код перед завершением потока (например, освобождение дескриптора сигнального объекта-события). Но в этом случае после установки флага в единицу, нужно дождаться, когда поток завершиться, чтобы затем освободить его дескриптор. Например, для этих целей можно сделать второй флаг flag2 и с помощью while(!флаг2) ждать, когда поток установит его в единицу, что будет означать, что поток завершился.

То есть примерно код будет выглядеть так (например, для потока reader класса ReadThread):

```
//пример кода завершения потока WinAPI "мягким" способом
flag=1;
while(!flag2){}
flag=0; flag2=0;
CloseHandle(reader);

//как для этого нужно изменить код потока:
//главная функция потока, реализует приём байтов из COM-порта
DWORD WINAPI ReadThread(LPVOID)
{
 ..........

 while(!flag)    //пока поток не будет прерван, выполняем цикл
  {
   .........
  }
CloseHandle(overlapped.hEvent);
```


© PIClist RUS, 2007

```
flag2=1;
}
```

Можно сделать и без второго флага: сделать так, чтобы поток просто сбрасывал flag в ноль:

```
//пример кода завершения потока WinAPI "мягким" способом
flag=1;
while(!flag){}
CloseHandle(reader);

//как для этого нужно изменить код потока:
//главная функция потока, реализует приём байтов из COM-порта
DWORD WINAPI ReadThread(LPVOID)
{
 .......
 CloseHandle(overlapped.hEvent);
 flag=0;
}
```

Следует отметить, что данный случай подходит, если в программе реализован только один поток. Попробуйте изменить код так, чтобы он работал для двух потоков. Может, вы сможете придумать и более удобный способ. Например, для управления потоками WINAPI можно использовать сигнальные объекты-события и другие подобные объекты.

В нашем же случае используется именно функция TerminateThread.

Полностью поток чтения выглядит так:

```
HANDLE reader;     //дескриптор потока чтения из порта

DWORD WINAPI ReadThread(LPVOID);

//-----------------------------------------------------------------------------
//.............................. поток ReadThead .............................
//-----------------------------------------------------------------------------

void ReadPrinting(void);

//-----------------------------------------------------------------------------

//главная функция потока, реализует приём байтов из COM-порта
DWORD WINAPI ReadThread(LPVOID)
{
 COMSTAT comstat;          //структура текущего состояния порта, в данной программе используется для определения количества принятых в порт байтов
 DWORD btr, temp, mask, signal;   //переменная temp используется в качестве заглушки

 overlapped.hEvent = CreateEvent(NULL, true, true, NULL);      //создать сигнальный объект-событие для асинхронных операций
 SetCommMask(COMport, EV_RXCHAR);                              //установить маску на срабатывание по событию приёма байта в порт
 while(1)                                        //пока поток не будет прерван, выполняем цикл
  {
   WaitCommEvent(COMport, &mask, &overlapped);                 //ожидать события приёма байта (это и есть перекрываемая операция)
   signal = WaitForSingleObject(overlapped.hEvent, INFINITE);  //приостановить поток до прихода байта
   if(signal == WAIT_OBJECT_0)                                 //если событие прихода байта произошло
    {
     if(GetOverlappedResult(COMport, &overlapped, &temp, true)) //проверяем, успешно ли завершилась перекрываемая операция WaitCommEvent
      if((mask & EV_RXCHAR)!=0)                                //если произошло именно событие прихода байта
       {
        ClearCommError(COMport, &temp, &comstat);              //нужно заполнить структуру COMSTAT
        btr = comstat.cbInQue;                         //и получить из неё количество принятых байтов
        if(btr)                                        //если действительно есть байты для чтения
        {
         ReadFile(COMport, bufrd, btr, &temp, &overlapped);     //прочитать байты из порта в буфер программы
         counter+=btr;                                          //увеличиваем счётчик байтов
         ReadPrinting();                               //вызываем функцию для вывода данных на экран и в файл
        }
       }
    }
  }
```

```
}

//---------------------------------------------------------------------------

//выводим принятые байты на экран и в файл (если включено)
void ReadPrinting()
{
 Form1->Memo1->Lines->Add((char*)bufrd);  //выводим принятую строку в Memo
 Form1->StatusBar1->Panels->Items[2]->Text = "Всего принято " + IntToStr(counter) + " байт"; //выводим счётчик в строке состояния

 if(Form1->CheckBox3->Checked == true)  //если включен режим вывода в файл
  {
   write(handle, bufrd, strlen(bufrd)); //записать в файл данные из приёмного буфера
  }
 memset(bufrd, 0, BUFSIZE);              //очистить буфер (чтобы данные не накладывались друг на друга)
}

//---------------------------------------------------------------------------
```

Создание потока записи. Использование функций ResumeThread() и SuspendThread()

Для демонстрации потоков на WINAPI была сделана только одна версия программы, в которой для потока записи сразу реализована демонстрация использования запуска и останова потока.

В целом реализация останова и запуска потоков на средствах WINAPI аналогична использованию останова и запуска потоков на TThread с помощью методов Resume() и Suspend(), только в данном случае используются функции ResumeThread() и SuspendThread().

Прототипы этих функций:

```
DWORD ResumeThread(HANDLE hThread);

DWORD SuspendThread(HANDLE hThread);
```

Эти функции получают в качестве аргумента дескриптор потока, который нужно запустить (ResumeThread()) или остановить (SuspendThread()). Рассмотрим последовательно создание потока и использование этих функций.

Поток записи создаётся аналогично потоку чтения с помощью функции CreateThread().

Для этого сначала объявляем и создаём рабочую функцию потока:

```
//---------------------------------------------------------------------------

//главная функция потока, выполняет передачу байтов из буфера в COM-порт
DWORD WINAPI WriteThread(LPVOID)
{
 DWORD temp, signal;       //temp - переменная-заглушка

 overlappedwr.hEvent = CreateEvent(NULL, true, true, NULL);       //создать событие
 while(1)
  {WriteFile(COMport, bufwr, strlen(bufwr), &temp, &overlappedwr);  //записать байты в порт (перекрываемая операция!)
   signal = WaitForSingleObject(overlappedwr.hEvent, INFINITE);  //приостановить поток, пока не завершится перекрываемая операция WriteFile

   if((signal == WAIT_OBJECT_0) && (GetOverlappedResult(COMport, &overlappedwr, &temp, true)))      //если операция завершилась успешно
     {
      Form1->StatusBar1->Panels->Items[0]->Text  = "Передача прошла успешно";    //вывести сообщение об этом в строке состояния
     }
   else {Form1->StatusBar1->Panels->Items[0]->Text  = "Ошибка передачи";}      //иначе вывести в строке состояния сообщение об ошибке

   SuspendThread(writer);
  }
}

//---------------------------------------------------------------------------
```

 © PIClist RUS, 2007

Как вы можете заметить, здесь вывод состояния операции в панель состояния осуществляется сразу в главной функции потока. То есть в отличие от программы с TThread в версии с WINAPI поток записи состоит только из одной функции.

Кроме того, вы должны обратить внимание, что здесь также как и в потоке чтения на WINAPI используется бесконечный цикл while(1) из-за той же особенности завершения работы потока.

Самая последняя операция внутри цикла потока – это останов потока с помощью функции SuspendThread(), который аналогичен использованию метода Suspend() в потоке записи на TThread. То есть здесь поток также останавливает сам себя.

После того как создана функция потока, необходимо создать сам поток:

```
HANDLE writer;
writer = CreateThread(NULL, 0, WriteThread, NULL, CREATE_SUSPENDED, NULL);
```

Поток записи, как и поток чтения, создаётся в функции открытия порта COMOpen().

Обратите внимание, что при создании потока записи в порт в качестве пятого параметра указывается CREATE_SUSPENDED, это означает, что поток будет создан в остановленном состоянии, и для его запуска необходимо будет использовать функцию ResumeThread().

Эта функция вызывается, когда требуется отправить данные в порт. Подготовка к отправке данных и активирование потока осуществляется в обработчике нажатия кнопки "Передать":

```
//---------------------------------------------------------------------------

//кнопка "Передать"
void __fastcall TForm1::Button1Click(TObject *Sender)
{
 memset(bufwr,0,BUFSIZE);                 //очистить программный передающий буфер, чтобы данные не
накладывались друг на друга
 PurgeComm(COMport, PURGE_TXCLEAR);            //очистить передающий буфер порта
 strcpy(bufwr,Form1->Edit1->Text.c_str());      //занести в программный передающий буфер строку из
Edit1

 ResumeThread(writer);               //активировать поток записи данных в порт

}

//---------------------------------------------------------------------------
```

Здесь всё аналогично обработчику кнопки "Передать" в программе с TThread второй версии (где используются останов и запуск потоков), за тем лишь исключением, что для запуска потока используется WINAPI функция ResumeThread().

Когда поток заканчивает передачу данных, он останавливает сам себя функцией SuspendThread() (см. выше).

Поток записи, как и поток чтения, существует всё время, пока открыт COM-порт. Когда порт закрывается, и поток записи больше не нужен, он уничтожается (в функции COMClose()) таким же образом, как и поток чтения:

```
if(writer)       //если поток записи работает, завершить его; проверка if(writer) обязательна, иначе
возникают ошибки
  {TerminateThread(writer,0);
   CloseHandle(overlappedwr.hEvent); //нужно закрыть объект-событие
   CloseHandle(writer);
  }
```

Здесь следует отметить отличие от кода уничтожения потока записи для TThread (вторая версия программы), которое заключается в том, что здесь не нужно активировать поток для его уничтожения, так как функция TerminateThread уничтожает поток без проверки каких-либо условий. Зато если вы будете использовать "мягкое" уничтожение потока (как описывалось выше), то в этом случае вам необходимо будет также использовать пробуждение потока для его завершения.

Функцию TerminateThread также можно использовать и для немедленного принудительного завершения потоков TThread, для этого в неё передаётся дескриптор потока (свойство Handle), например:

```
TerminateThread((*void)reader->Handle, 0);
```

Если завершить поток записи writer на TThread с помощью этой функции, а не с помощью метода Terminate(), то активировать поток методом Resume() для уничтожения не требуется.

Приведём полный код потока записи в порт на WINAPI.

```
HANDLE writer;     //дескриптор потока записи в порт

DWORD WINAPI WriteThread(LPVOID);

//---------------------------------------------------------------------------
//.............................. поток WriteThead ...........................
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------

//главная функция потока, выполняет передачу байтов из буфера в COM-порт
DWORD WINAPI WriteThread(LPVOID)
{
 DWORD temp, signal;      //temp - переменная-заглушка

 overlappedwr.hEvent = CreateEvent(NULL, true, true, NULL);      //создать событие
 while(1)
  {WriteFile(COMport, bufwr, strlen(bufwr), &temp, &overlappedwr);  //записать байты в порт (перекрываемая операция!)
   signal = WaitForSingleObject(overlappedwr.hEvent, INFINITE);  //приостановить поток, пока не завершится перекрываемая операция WriteFile

   if((signal == WAIT_OBJECT_0) && (GetOverlappedResult(COMport, &overlappedwr, &temp, true)))      //если операция завершилась успешно
     {
      Form1->StatusBar1->Panels->Items[0]->Text  = "Передача прошла успешно";    //вывести сообщение об этом в строке состояния
     }
   else {Form1->StatusBar1->Panels->Items[0]->Text  = "Ошибка передачи";}     //иначе вывести в строке состояния сообщение об ошибке

   SuspendThread(writer);

  }
}

//---------------------------------------------------------------------------
```

Сравнение использования класса TThread и средств WINAPI

Одно из отличий заключается в том, что в потоках WINAPI можно использовать обращение к графическим компонентам и файлам, так как сообщения потоков WINAPI ставятся в очередь сообщений главного потока процесса. Таким образом избегаются конфликты между потоками при обращении к разделяемым компонентам или файлам. А вот в TThread для этого используется метод Synchronize(), который выполняет то же самое – ставит сообщения в очередь главного потока процесса, за счёт чего можно избежать конфликтов между потоками.

Второе отличие – это наличие у потоков TThread возможности "мягкого" завершения с использованием метода Terminate() и свойства Terminated, об этом подробно рассказывалось выше. Для потоков WINAPI организовать "мягкое" завершение сложнее, поэтому используется принудительное завершение потоков с помощью функции TerminateThread. (По крайней мере, авторам данной статьи способ завершения потока WINAPI, аналогичный "мягкому" завершению потока TThread, за исключением примера с использованием флагов, не известен.)

Самое же главное отличие состоит в том, что потоки TThread строятся на основе классов, а потоки WINAPI – на основе функций.

© PIClist RUS, 2007

Описание программы

Так как все три версии программ отличаются только в плане использования потоков, остальные функции программ являются общими.

Кроме потоков в программе используются функции открытия и закрытия порта, а также обработчики событий для элементов формы. Рассмотрим их по порядку.

При запуске программы вызывается конструктор формы, в котором происходит начальная инициализация элементов формы. Здесь отключаются некоторые элементы формы.

```
//конструктор формы, обычно в нём выполняется инициализация элементов формы
__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner)
{
 //инициализация элементов формы при запуске программы
 Form1->Label5->Enabled = false;
 Form1->Label6->Enabled = false;
 Form1->Button1->Enabled = false;
 Form1->CheckBox1->Enabled = false;
 Form1->CheckBox2->Enabled = false;
}
```

Кроме функций потоков главными при работе с портом являются функции открытия и закрытия порта. Эти функции должны быть объявлены до их использования, поэтому мы объявляем их сразу после объявлений переменных.

```
void COMOpen(void);             //открыть порт
void COMClose(void);            //закрыть порт
```

Функция открытия и инициализации порта COMOpen()

Сначала приведём код этой функции, а затем подробно её рассмотрим. В коде жирным шрифтом выделены основные моменты, на которые нужно обратить внимание.

```
//---------------------------------------------------------------------------

//функция открытия и инициализации порта
void COMOpen()
{
 String portname;           //имя порта (например, "COM1", "COM2" и т.д.)
 DCB dcb;                   //структура для общей инициализации порта DCB
 COMMTIMEOUTS timeouts;     //структура для установки таймаутов

 portname = Form1->ComboBox1->Text;      //получить имя выбранного порта

 //открыть порт, для асинхронных операций обязательно нужно указать флаг FILE_FLAG_OVERLAPPED
 COMport = CreateFile(portname.c_str(),GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
FILE_FLAG_OVERLAPPED, NULL);
 //здесь:
 // - portname.c_str() - имя порта в качестве имени файла, c_str() преобразует строку типа String в строку в виде
массива типа char, иначе функция не примет
 // - GENERIC_READ | GENERIC_WRITE - доступ к порту на чтение/записть
 // - 0 - порт не может быть общедоступным (shared)
 // - NULL - дескриптор порта не наследуется, используется дескриптор безопасности по умолчанию
 // - OPEN_EXISTING - порт должен открываться как уже существующий файл
 // - FILE_FLAG_OVERLAPPED - этот флаг указывает на использование асинхронных операций
 // - NULL - указатель на файл шаблона не используется при работе с портами

 if(COMport == INVALID_HANDLE_VALUE)            //если ошибка открытия порта
  {
   Form1->SpeedButton1->Down = false;           //отжать кнопку
   Form1->StatusBar1->Panels->Items[0]->Text = "Не удалось открыть порт";       //вывести сообщение в строке
состояния
   return;
  }

 //инициализация порта
```

```
 dcb.DCBlength = sizeof(DCB);     //в первое поле структуры DCB необходимо занести её длину, она будет
использоваться функциями настройки порта для контроля корректности структуры

 //считать структуру DCB из порта
 if(!GetCommState(COMport, &dcb)) //если не удалось - закрыть порт и вывести сообщение об ошибке в строке
состояния
  {
   COMClose();
   Form1->StatusBar1->Panels->Items[0]->Text  = "Не удалось считать DCB";
   return;
  }

 //инициализация структуры DCB
 dcb.BaudRate = StrToInt(Form1->ComboBox2->Text);       //задаём скорость передачи 115200 бод
 dcb.fBinary = TRUE;                                    //включаем двоичный режим обмена
 dcb.fOutxCtsFlow = FALSE;                              //выключаем режим слежения за сигналом CTS
 dcb.fOutxDsrFlow = FALSE;                              //выключаем режим слежения за сигналом DSR
 dcb.fDtrControl = DTR_CONTROL_DISABLE;                 //отключаем использование линии DTR
 dcb.fDsrSensitivity = FALSE;                           //отключаем восприимчивость драйвера к состоянию линии
DSR
 dcb.fNull = FALSE;                                     //разрешить приём нулевых байтов
 dcb.fRtsControl = RTS_CONTROL_DISABLE;                 //отключаем использование линии RTS
 dcb.fAbortOnError = FALSE;                             //отключаем остановку всех операций чтения/записи при
ошибке
 dcb.ByteSize = 8;                                      //задаём 8 бит в байте
 dcb.Parity = 0;                                        //отключаем проверку чётности
 dcb.StopBits = 0;                                      //задаём один стоп-бит

 //загрузить структуру DCB в порт
 if(!SetCommState(COMport, &dcb)) //если не удалось - закрыть порт и вывести сообщение об ошибке в строке
состояния
  {
   COMClose();
   Form1->StatusBar1->Panels->Items[0]->Text  = "Не удалось установить DCB";
   return;
  }

 //установить таймауты
 timeouts.ReadIntervalTimeout = 0;              //таймаут между двумя символами
 timeouts.ReadTotalTimeoutMultiplier = 0;//общий таймаут операции чтения
 timeouts.ReadTotalTimeoutConstant = 0;       //константа для общего таймаута операции чтения
 timeouts.WriteTotalTimeoutMultiplier = 0;    //общий таймаут операции записи
 timeouts.WriteTotalTimeoutConstant = 0;      //константа для общего таймаута операции записи

 //записать структуру таймаутов в порт
 if(!SetCommTimeouts(COMport, &timeouts))//если не удалось - закрыть порт и вывести сообщение об ошибке в строке
состояния
  {
   COMClose();
   Form1->StatusBar1->Panels->Items[0]->Text  = "Не удалось установить тайм-ауты";
   return;
  }

//установить размеры очередей приёма и передачи
 SetupComm(COMport,2000,2000);

 //создать или открыть существующий файл для записи принимаемых данных
 handle = open("test.txt", O_CREAT | O_APPEND | O_BINARY | O_WRONLY, S_IREAD | S_IWRITE);

 if(handle==-1)           //если произошла ошибка открытия файла
  {
   Form1->StatusBar1->Panels->Items[1]->Text = "Ошибка открытия файла";      //вывести сообщение об этом в
командной строке
   Form1->Label6->Hide();                                            //спрятать надпись с именем файла
   Form1->CheckBox3->Checked = false;                                //сбросить и отключить галочку
   Form1->CheckBox3->Enabled = false;
  }
 else { Form1->StatusBar1->Panels->Items[0]->Text = "Файл открыт успешно"; } //иначе вывести в строке состояния
сообщение об успешном открытии файла

 PurgeComm(COMport, PURGE_RXCLEAR);      //очистить принимающий буфер порта

 reader = new ReadThread(false);         //создать и запустить поток чтения байтов
 reader->FreeOnTerminate = true;       //установить это свойство потока, чтобы он автоматически уничтожался после
завершения

}

//---------------------------------------------------------------------------
```


© PIClist RUS, 2007

Итак, для того, чтобы можно было работать с портом, его нужно открыть и правильно проинициализировать. Какие действия нужно для этого выполнить:

1) Сначала объявляются необходимые переменные: строковая переменная для имени порта, а также структуры типа **DCB** и **COMMTIMEOUTS**, которые понадобятся для инициализации порта.

```
String portname;        //имя порта (например, "COM1", "COM2" и т.д.)
DCB dcb;                         //структура для общей инициализации порта DCB
COMMTIMEOUTS timeouts;           //структура для установки таймаутов
```

2) из выпадающего списка с именами портов получаем имя порта. Для этого необходимо, чтобы свойство Style выпадающего списка (ComboBox) было установлено в csDropDownList, чтобы исключить возможность его редактирования. При этом имена портов должны быть записаны именно так, как они будут использоваться в функции CreateFile, иначе такой способ не сработает.

```
portname = Form1->ComboBox1->Text;   //получить имя выбранного порта
```

3) теперь с помощью функции CreateFile открываем порт.

```
//открыть порт, для асинхронных операций обязательно нужно указать флаг FILE_FLAG_OVERLAPPED
 COMport = CreateFile(portname.c_str(),GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
FILE_FLAG_OVERLAPPED, NULL);
 //здесь:
 // - portname.c_str() - имя порта в качестве имени файла, c_str() преобразует строку типа String в
строку в виде массива типа char, иначе функция не примет
 // - GENERIC_READ | GENERIC_WRITE - доступ к порту на чтение/записть
 // - 0 - порт не может быть общедоступным (shared)
 // - NULL - дескриптор порта не наследуется, используется дескриптор безопасности по умолчанию
 // - OPEN_EXISTING - порт должен открываться как уже существующий файл
 // - FILE_FLAG_OVERLAPPED - этот флаг указывает на использование асинхронных операций
 // - NULL - указатель на файл шаблона не используется при работе с портами
```

Функция открывает порт как файл. Она возвращает дескриптор открытого порта.

В функцию передаются следующие параметры:

1) portname.c_str() – это имя порта. c_str() – метод класса TString, который преобразует строковую переменную в массив типа char, так как функция принимает имя порта только в виде массива char.

2) GENERIC_READ | GENERIC_WRITE – этот параметр означает, что доступ к порту будет осуществляться на чтение/запись.

3) 0 – означает, что порт не может быть общедоступным (shared) – этот параметр для порта всегда должен иметь такое значение.

4) NULL – дескриптор порта не наследуется, используется дескриптор безопасности по умолчанию.

5) OPEN_EXISTING – порт обязательно должен открываться как уже существующий файл.

6) FILE_FLAG_OVERLAPPED – этот флаг указывает на то, что с портом будут выполняться асинхронные перекрываемые операции.

7) NULL – означает, что указатель на файл шаблона не используется – это обязательно для работы с портами.

В переменную COMport типа HANDLE запоминается возвращаемый дескриптор порта. Далее работа с портом будет осуществляться именно через него.

Затем проверяем, удалось ли открыть порт. Если попытка была неудачной, функция CreateFile вернёт значение INVALID_HANDLE_VALUE. Тогда отжимаем кнопку "Открыть-Закрыть порт", выводим соответствующее сообщение об ошибке и выходим из функции. Если порт открыт успешно, переходим к его инициализации.

```
if(COMport == INVALID_HANDLE_VALUE)            //если ошибка открытия порта
  {
   Form1->SpeedButton1->Down = false;           //отжать кнопку
   Form1->StatusBar1->Panels->Items[0]->Text = "Не удалось открыть порт";       //вывести сообщение в
строке состояния
```

```
    return;
   }
```

5) Инициализация начинается со структуры dcb. Для начала нужно считать её значение. Для этого сначала в первое поле структуры DCBlength заносим значение размера структуры:

```
dcb.DCBlength = sizeof(DCB);  //в первое поле структуры DCB необходимо занести её длину, она будет использоваться функциями настройки порта для контроля корректности структуры
```

А затем из порта с помощью функции GetCommState считываем в неё текущие настройки:

```
//считать структуру DCB из порта
if(!GetCommState(COMport, &dcb))     //если не удалось - закрыть порт и вывести сообщение об ошибке в строке состояния
 {
  COMClose();
  Form1->StatusBar1->Panels->Items[0]->Text  = "Не удалось считать DCB";
  return;
 }
```

Если попытка чтения настроек была неудачной, тогда закрываем порт, в строке состояния выводим сообщение об ошибке и выходим из функции.

Зачем считывать настройки из порта? Это полезно в том случае, если нужно изменить только те из них, которые нам нужны, а значения остальных нам неизвестны.

После того как считали структуру dcb, заполняем нужные нам поля настройками.

```
//инициализация структуры DCB
 dcb.BaudRate = StrToInt(Form1->ComboBox2->Text);       //задаём скорость передачи 115200 бод
 dcb.fBinary = TRUE;                                    //включаем двоичный режим обмена
 dcb.fOutxCtsFlow = FALSE;                              //выключаем режим слежения за сигналом CTS
 dcb.fOutxDsrFlow = FALSE;                              //выключаем режим слежения за сигналом DSR
 dcb.fDtrControl = DTR_CONTROL_DISABLE;                 //отключаем использование линии DTR
 dcb.fDsrSensitivity = FALSE;                           //отключаем восприимчивость драйвера к состоянию линии DSR
 dcb.fNull = FALSE;                                     //разрешить приём нулевых байтов
 dcb.fRtsControl = RTS_CONTROL_DISABLE;                 //отключаем использование линии RTS
 dcb.fAbortOnError = FALSE;                             //отключаем остановку всех операций чтения/записи при ошибке
 dcb.ByteSize = 8;                                      //задаём 8 бит в байте
 dcb.Parity = 0;                                        //отключаем проверку чётности
 dcb.StopBits = 0;                                      //задаём один стоп-бит
```

Самые важные для нас параметры, которые нужно настроить – это скорость передачи (BaudRate), разрешение приёма нулевых байтов (так как они могут появиться среди передаваемых данных) (fNULL), установка восьми бит данных в байте (ByteSize) и одного стоп-бита (StopBits) и отключение проверки чётности (Parity). Также нам необходимо отключить использование линий CTS, DSR, DTR, RTS (fOutxCtsFlow, fOutxDsrFlow, fDtrControl, fDsrSensitivity, fRtsControl), а также остановку выполнения операций чтения/записи при ошибке (fAbortOnError) и включить двоичный режим обмена (fBinary).

Теперь необходимо загрузить структуру с установленными настройками в порт. Для этого используем функцию SetCommState:

```
//загрузить структуру DCB в порт
 if(!SetCommState(COMport, &dcb))    //если не удалось - закрыть порт и вывести сообщение об ошибке в строке состояния
   {
    COMClose();
    Form1->StatusBar1->Panels->Items[0]->Text  = "Не удалось установить DCB";
    return;
   }
```

Также выполняем проверку, удалось ли выполнить операцию по записи структуры с настройками в порт. Аналогично – если попытка записи структуры в порт была неудачной, закрываем порт, отжимаем кнопку и выходим из функции. В случае удачной попытки – продолжаем инициализацию.

6) Теперь инициализируем структуру COMMTIMEOUTS. Так как таймауты использовать не будем, заполняем все поля структуры нулями:

```
//установить таймауты
 timeouts.ReadIntervalTimeout = 0;          //таймаут между двумя символами
 timeouts.ReadTotalTimeoutMultiplier = 0;   //общий таймаут операции чтения
 timeouts.ReadTotalTimeoutConstant = 0;        //константа для общего таймаута операции чтения
 timeouts.WriteTotalTimeoutMultiplier = 0;     //общий таймаут операции записи
 timeouts.WriteTotalTimeoutConstant = 0;       //константа для общего таймаута операции записи
```

И записываем структуру таймаутов в порт:

```
//записать структуру таймаутов в порт
 if(!SetCommTimeouts(COMport, &timeouts))  //если не удалось - закрыть порт и вывести сообщение об
ошибке в строке состояния
  {
   COMClose();
   Form1->StatusBar1->Panels->Items[0]->Text  = "Не удалось установить тайм-ауты";
   return;
  }
```

Если запись структуры в порт неудачна, закрываем порт, выводим сообщение об ошибке в строке состояния и выходим из функции.

Иначе – продолжаем инициализацию.

Возникает вопрос: зачем вообще нужны таймауты. Они нужны а том случае, если устройство (программа на ПК) ожидает приём определённого числа байтов, а приходит только часть их. Поэтому чтобы устройство (программа) не ждало неизвестное количество времени приёма недостающих байтов, устанавливают таймауты. Тогда, не получив недостающие байты, устройство (программа) отключится по истечении таймаута и не "зависнет".

Но так как тут у нас работают потоки, которые постоянно находятся в режиме приёма данных, и чтобы не усложнять программу, мы таймауты не используем.

7) Теперь устанавливаем размеры очередей приёма и передачи:

```
//установить размеры очередей приёма и передачи
SetupComm(COMport,2000,2000);
```

Эта функция устанавливает внутренний буфер драйвера устройства. В принципе её использование не обязательно. Здесь мы ставим большой объём данных на случай быстрого заполнения буфера устройства при использовании высоких скоростей передачи данных и непрерывного потока данных. Но как показывает практика – они не нужны, так как принимается по 8 байтов (редко больше) – но пока непонятно, с чем это связано. Возможно с тем, что эти значения являются как бы рекомендацией драйверу устройства, и могут им игнорироваться.

8) Теперь открываем файл для записи принимаемых данных:

```
//создать или открыть существующий файл для записи принимаемых данных
 handle = open("test.txt", O_CREAT | O_APPEND | O_BINARY | O_WRONLY, S_IREAD | S_IWRITE);

 if(handle==-1)         //если произошла ошибка открытия файла
  {
   Form1->StatusBar1->Panels->Items[1]->Text = "Ошибка открытия файла";     //вывести сообщение об
этом в командной строке
   Form1->Label6->Hide();                                                 //спрятать надпись с именем
файла
   Form1->CheckBox3->Checked = false;                                     //сбросить и отключить
галочку
   Form1->CheckBox3->Enabled = false;
  }
```

```
 else { Form1->StatusBar1->Panels->Items[0]->Text = "Файл открыт успешно"; } //иначе вывести в строке состояния сообщение об успешном открытии файла
```

Если файл открыть не удалось – в строке состояния выводим сообщение об ошибке, прячем имя файла на форме, сбрасываем и отключаем галочку "Сохранить в файл".

Иначе – выводим сообщение, что файл открыт успешно.

9) Сбрасываем буферы порта:

```
PurgeComm(COMport, PURGE_RXCLEAR);   //очистить принимающий буфер порта
```

Это чтобы не передался и не принялся какой-нибудь находящийся в буфере мусор.

10) На этом шаге создаём необходимые потоки. Если это версия программы без использования методов Suspend() и Resume(), то создаётся (и сразу запускается) только поток чтения.

```
reader = new ReadThread(false);      //создать и запустить поток чтения байт
reader->FreeOnTerminate = true; //установить это свойство потока, чтобы он автоматически уничтожался после завершения
```

Если это программа, в которой используются методы Suspend() и Resume(), то создаются оба потока – и чтения, и записи. При этом поток чтения запускается сразу, а поток записи создаётся в остановленном состоянии.

```
reader = new ReadThread(false);      //создать и запустить поток чтения байт
reader->FreeOnTerminate = true;  //установить это свойство потока, чтобы он автоматически уничтожался после завершения

writer = new WriteThread(true);  //создать поток записи данных в порт
writer->FreeOnTerminate = true;  //установить это свойство, чтобы поток автоматически уничтожался после завершения
```

А для потоков на WINAPI создание потоков выглядит так:

```
reader = CreateThread(NULL, 0, ReadThread, NULL, 0, NULL);                  //создаём поток чтения, который сразу начнёт выполняться (предпоследний параметр = 0)
writer = CreateThread(NULL, 0, WriteThread, NULL, CREATE_SUSPENDED, NULL);  //создаём поток записи в остановленном состоянии (предпоследний параметр = CREATE_SUSPENDED)
```

Здесь также создаются оба потока, но с помощью функции WINAPI CreateThread. Поток чтения также запускается сразу (предпоследний параметр в функции равен нулю), а поток записи создаётся в остановленном состоянии (предпоследний параметр имеет значение CREATE_SUSPENDED).

Функция закрытия порта COMClose()

В этой функции выполняется завершение потоков, закрытие дескрипторов порта, файла и объектов-событий. Для всех трёх программ эта функция немного различается:

1) Функция COMClose() для программы на TThread без использования методов Resume() и Suspend():

```
//функция закрытия порта
void COMClose()
{
 if(writer)writer->Terminate();             //если поток записи работает, завершить его; проверка if(writer) обязательна, иначе возникают ошибки
 if(reader)reader->Terminate();          //если поток чтения работает, завершить его; проверка if(reader) обязательна, иначе возникают ошибки

 CloseHandle(COMport);                    //закрыть порт
 COMport=0;                                   //обнулить переменную для дескриптора порта
 close(handle);                        //закрыть файл для записи принимаемых данных
 handle=0;                             //обнулить переменную для дескриптора файла
}
```

 © PIClist RUS, 2007

Здесь выполняется проверка, активны ли потоки, и если да – они завершаются методом Terminate(). Затем закрываются порт и файл для записи данных, обнуляются дескрипторы порта и файла. Это нужно, чтобы в обработчике закрытия формы (см. **ниже**) можно было проверить, открыты порт и файл, или нет. Это связано с тем, что после освобождения дескриптора переменная не обнуляется, а сохраняет его номер, и в этом случае проверка не сработает. Если при этом попытаться закрыть дескриптор, то функция, закрывающая дескриптор, выдаст ошибку. На это можно не обращать внимания, так как в этом случае функция просто возвращает ошибку и не предпринимает никаких действий. Но лучше, если программа будет обрабатывать такие ситуации, такая программа является более грамотной и надёжной.

2) Функция COMClose() для программы на TThread с использованием методов Resume() и Suspend():

```
//функция закрытия порта
void COMClose()
{
 //если поток записи существует, подать ему команду на завершение и запустить его, чтобы он выполнил
завершение;
 if(writer) //проверка if(writer) обязательна, иначе возникают ошибки;
  {
   writer->Terminate();
   writer->Resume();
  }
 if(reader)reader->Terminate();         //если поток чтения работает, завершить его; проверка
if(reader) обязательна, иначе возникают ошибки

 CloseHandle(COMport);                  //закрыть порт
 COMport=0;                                 //обнулить переменную для дескриптора порта
 close(handle);                       //закрыть файл для записи принимаемых данных
 handle=0;                            //обнулить переменную для дескриптора файла
}
```

Здесь отличие от первой версии состоит в том, что после того как для потока записи writer вызывается метод Terminate(), также дополнительно вызывается метод Resume() – это необходимо, потому что поток может находиться в остановленном состоянии и не сможет выполнить проверку свойства Terminate, чтобы завершиться. Все остальное – как и в первой версии.

3) Функция COMClose() для программы на WINAPI с использованием функций ResumeThread() и SuspendThread():

```
//функция закрытия порта
void COMClose()
{
//Примечание: так как при прерывании потоков, созданных с помощью функций WinAPI, функцией TerminateThread
//         поток может быть прерван жёстко, в любом месте своего выполнения, то освобождать дескриптор
//         сигнального объекта-события, находящегося в структуре типа OVERLAPPED, связанной с потоком,
//         следует не внутри кода потока, а отдельно, после вызова функции TerminateThread.
//         После чего нужно освободить и сам дескриптор потока.
 if(writer)        //если поток записи работает, завершить его; проверка if(writer) обязательна, иначе возникают
ошибки
  {TerminateThread(writer,0);
   CloseHandle(overlappedwr.hEvent);     //нужно закрыть объект-событие
   CloseHandle(writer);
  }
 if(reader)          //если поток чтения работает, завершить его; проверка if(reader) обязательна, иначе
возникают ошибки
  {TerminateThread(reader,0);
   CloseHandle(overlapped.hEvent);       //нужно закрыть объект-событие
   CloseHandle(reader);
  }

 CloseHandle(COMport);            //закрыть порт
 COMport=0;               //обнулить переменную для дескриптора порта
 close(handle);           //закрыть файл, в который велась запись принимаемых данных
 handle=0;                //обнулить переменную для дескриптора файла
}
```

Эта функция отличается от первых двух, и отличие связано с особенностями использования WINAPI. Здесь потоки прерываются "грубым" способом – с помощью функции TerminateThread(), которая

немедленно завершает поток, потому что у потоков WINAPI нет встроенных средств для "мягкого" завершения. "Мягкое" завершение можно обеспечить вручную, например – с помощью флагов, но это усложняет программу. Здесь мы не стали прибегать к этому, а воспользовались именно функцией TerminateThread(). У этой функции плюс в том, что она завершает даже спящий поток, то есть после её вызова не нужно активировать поток функцией ResumeThread(), как в предыдущем примере. Но из-за неё нам приходится после её вызова закрывать дескрипторы сигнальных объектов-событий, а также дескрипторы самих потоков.

А затем также как и в остальных функциях, выполняется закрытие порта и файла.

Конструктор формы и обработчики событий формы и её элементов

Здесь опишем конструктор формы и различные обработчики событий элементов формы и самой формы, которые мы используем в нашей программе. Все обработчики описаны в том порядке, в котором они расположены в коде.

Конструктор формы не представляет ничего сложного – в нём происходит инициализация элементов формы, представляющая собой деактивирование элементов, которые не могут быть доступны, если не открыт порт или не установлена галочка "Сохранить в файл".

```
//---------------------------------------------------------------------------

//конструктор формы, обычно в нём выполняется инициализация элементов формы
__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner)
{
 //инициализация элементов формы при запуске программы
 Form1->Label5->Enabled = false;
 Form1->Label6->Enabled = false;
 Form1->Button1->Enabled = false;
 Form1->CheckBox1->Enabled = false;
 Form1->CheckBox2->Enabled = false;
}

//---------------------------------------------------------------------------
```

Обработчик нажатия на кнопку "Открыть порт". Кнопка "Открыть порт" – это кнопка SpeedButton, её интересная особенность – то, что она может оставаться в нажатом состоянии, и, чтобы её отжать, нужно нажать эту кнопку ещё раз (похоже на включение/выключение галочки CheckBox). Чтобы такое стало возможно, в свойствах кнопки нужно выставить следующие настройки: AlowAllUp=true, GroupIndex=1. Также на эту кнопку можно прикрепить иконку. Для этого в Glyph загрузить иконку и установить NumGlyph=1. А с помощью свойства Layout выбирать, с какой стороны должна находиться иконка.

Итак, при нажатии на кнопку в обработчике выполняется проверка – если кнопка нажата, тогда открываем порт и активируем кнопку "Передать", а также флажки сигналов "DTR" и "RTS". При этом меняем надпись на нажатой кнопке на "Закрыть порт". Затем сбрасываем счётчик принятых байтов. И вызываем обработчики флажков CheckBox1 и Checkbox2, чтобы включить линии DTR и RTS, если эти флажки были установлены.

Если кнопка отжалась, выполняем обратные действия: закрываем порт, меняем надпись на кнопке на "Открыть порт", очищаем строку состояния на форме, активируем списки настроек порта и отключаем кнопку "Передать" и галочки "DTR" и "RTS".

```
//---------------------------------------------------------------------------

//обработчик нажатия на кнопку "Открыть порт"
void __fastcall TForm1::SpeedButton1Click(TObject *Sender)
{
 if(SpeedButton1->Down)
  {
   COMOpen();                    //если кнопка нажата - открыть порт

   //показать/спрятать элементы на форме
   Form1->ComboBox1->Enabled = false;
```

```
   Form1->ComboBox2->Enabled = false;
   Form1->Button1->Enabled = true;
   Form1->CheckBox1->Enabled = true;
   Form1->CheckBox2->Enabled = true;

   Form1->SpeedButton1->Caption = "Закрыть порт"; //сменить надпись на кнопке

   counter = 0;  //сбрасываем счётчик байтов

   //если были включены флажки DTR и RTS, установить эти линии в единицу
   Form1->CheckBox1Click(Sender);
   Form1->CheckBox2Click(Sender);
  }

 else
  {
   COMClose();                  //если кнопка отжата - закрыть порт

   Form1->SpeedButton1->Caption = "Открыть порт"; //сменить надпись на кнопке
   Form1->StatusBar1->Panels->Items[0]->Text = "";      //очистить первую колонку строки состояния

   //показать/спрятать элементы на форме
   Form1->ComboBox1->Enabled = true;
   Form1->ComboBox2->Enabled = true;
   Form1->Button1->Enabled = false;
   Form1->CheckBox1->Enabled = false;
   Form1->CheckBox2->Enabled = false;
  }
}

//---------------------------------------------------------------------------
```

Обработчик закрытия формы выполняется при закрытии формы (когда вы нажимаете крестик или закрываете программу ещё каким-нибудь способом). В этом обработчике выполняются те же действия, что и в функции COMClose(), отличие состоит в том, что здесь при закрытии порта и файла выполняется проверка – если переменная дескриптора содержит 0, то закрытие дескриптора не выполняется. Это необходимо, чтобы избежать попытки повторного закрытия уже закрытого дескриптора. Такой подход более грамотный, чем просто всегда пытаться закрыть порт, игнорируя сообщения функций об ошибках. (См. **описание функции COMClose()**).

```
//---------------------------------------------------------------------------

//обработчик закрытия формы
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
 if(reader)reader->Terminate();      //завершить поток чтения из порта, проверка if(reader)
обязательна, иначе возникают ошибки
 if(writer)writer->Terminate();  //завершить поток записи в порт, проверка if(writer) обязательна,
иначе возникают ошибки
 if(COMport)CloseHandle(COMport);       //закрыть порт
 if(handle)close(handle);               //закрыть файл, в который велась запись принимаемых данных

}
//---------------------------------------------------------------------------
```

Обработчик нажатия галочки "Сохранить в файл". Он срабатывает в ответ на нажатие галочки. При этом проверяется – если галочка установлена, активируются надписи "Имя файла:" и "test.txt", а во второй раздел строки состояния выводится сообщение "Вывод в файл!". Это означает, что включена запись принимаемых данных в файл.

Если галочка отключена, надписи дезактивируются, а сообщение из строки состояния убирается.

```
//---------------------------------------------------------------------------

//галочка "Сохранить в файл"
void __fastcall TForm1::CheckBox3Click(TObject *Sender)
```

```
{
 if(Form1->CheckBox3->Checked)       //если галочка включена
  {
   //активировать соответствующие элементы на форме
   Form1->Label5->Enabled = true;
   Form1->Label6->Enabled = true;

   //вывести индикатор записи в файл в строке состояния
   Form1->StatusBar1->Panels->Items[1]->Text = "Вывод в файл!";
  }

 else                         //если галочка выключена
  {
   //отключить соответствующие элементы на форме
   Form1->Label5->Enabled = false;
   Form1->Label6->Enabled = false;

   //убрать индикатор записи в файл из строки состояния
   Form1->StatusBar1->Panels->Items[1]->Text = "";
  }

}
//---------------------------------------------------------------------------
```

Обработчик нажатия кнопки "Передать". Выполняется при нажатии на кнопку "Передать". Эта кнопка доступна, только когда открыт порт.

В обработчике с помощью функции memset() очищается программный передающий буфер для того, чтобы от предыдущих данных не оставался мусор, если они занимали больше ячеек массива, чем новые.

Затем с помощью функции PurgeComm() с параметром PURGE_TXCLEAR очищаем передающий буфер порта, чтобы избежать передачи какого-нибудь мусора, который может там оказаться.

После этого с помощью функции strcpy() копируем в буфер строку из поля ввода Edit1 (предварительно преобразовав её в массив типа char с помощью метода AnsiString-строки c_str()). Затем запускаем поток записи в порт. В данном случае обработчик кнопки "Передать" приведён для версии без использования методов Resume() и Suspend(), поэтому здесь поток записи создаётся заново.

```
//---------------------------------------------------------------------------

//кнопка "Передать"
void __fastcall TForm1::Button1Click(TObject *Sender)
{
 memset(bufwr,0,BUFSIZE);                 //очистить программный передающий буфер, чтобы данные не
накладывались друг на друга
 PurgeComm(COMport, PURGE_TXCLEAR);         //очистить передающий буфер порта
 strcpy(bufwr,Form1->Edit1->Text.c_str());   //занести в программный передающий буфер строку из Edit1

 writer = new WriteThread(false);      //создать и активировать поток записи данных в порт
 writer->FreeOnTerminate = true;       //установить это свойство, чтобы поток автоматически
уничтожался после завершения

}

//---------------------------------------------------------------------------
```

В версии с использованием этих методов вместо двух строчек создания потока будет одна строка с вызовом метода Resume():

```
writer->Resume();//активировать поток записи в порт
```

А в программе с потоками на WINAPI будет строка с вызовом функции ResumeThread():

```
ResumeThread(writer);               //активировать поток записи данных в порт
```

 © PIClist RUS, 2007

Обработчик нажатия кнопки "Очистка поля". Вызывается при нажатии на кнопку "Очистка поля". В нём выполняется очистка поля Memo1 с помощью его метода Clear().

```
//---------------------------------------------------------------------------

//кнопка "Очистить поле"
void __fastcall TForm1::Button3Click(TObject *Sender)
{
 Form1->Memo1->Clear(); //очистить Memo1
}

//---------------------------------------------------------------------------
```

Обработчик нажатия на галочку "DTR". Выполняется при нажатии на галочку. В обработчике проверяется – если галочка установлена – тогда с помощью функции EscapeCommFunction с параметром SETDTR устанавливаем линию DTR в единицу. Иначе – с помощью той же функции, но с параметром CLRDTR, сбрасываем линию DTR в ноль.

```
//---------------------------------------------------------------------------

//галочка "DTR"
void __fastcall TForm1::CheckBox1Click(TObject *Sender)
{
 //если установлена - установить линию DTR в единицу, иначе - в ноль
 if(Form1->CheckBox1->Checked) EscapeCommFunction(COMport, SETDTR);
 else EscapeCommFunction(COMport, CLRDTR);
}

//---------------------------------------------------------------------------
```

Обработчик нажатия на галочку "RTS". Выполняется при нажатии на галочку "RTS". В обработчике проверяется, установлена ли галочка. Если она установлена, тогда с помощью функции EscapeCommFunction с параметром SETRTS устанавливаем на линию RTS. Иначе – с помощью той же функции, но с параметром CLRRTS, сбрасываем линию RTS.

```
//---------------------------------------------------------------------------

//галочка "RTS"
void __fastcall TForm1::CheckBox2Click(TObject *Sender)
{
 //если установлена - установить линию RTS в единицу, иначе - в ноль
 if(Form1->CheckBox2->Checked) EscapeCommFunction(COMport, SETRTS);
 else EscapeCommFunction(COMport, CLRRTS);
}

//---------------------------------------------------------------------------
```

Переменные, используемые в программе

Для работы программы понадобилось создать несколько переменных. Некоторые из них обязательны, а некоторые – не обязательны.

1) Буфер приёма **bufrd** и буфер передачи **bufwr**. Используются для хранения принимаемых и передаваемых данных. Размер для них установлен в 255. Вы можете его изменить, поставив в директиве #define BUFSIZE другую величину. Следует отметить, что для буфера передачи размер может быть любой – это зависит от объёма передаваемых данных. Здесь мы просто ограничили его числом 255. При этом в Edit1 тоже поставили ограничение на количество вводимых символов, но в 254 (на всякий случай, чтобы последний байт в буфере был зарезервирован, например, под нулевой символ конца строки).

А вот с приёмным буфером ситуация иная. В принципе, для него можно поставить и меньший размер, так как выяснилось, что почти всегда данные считываются из порта по восемь байтов, причём на любой

 © PIClist RUS, 2007

скорости. Очень редко бывает больше. И с чем это связано, к сожалению, остаётся непонятно. Поэтому пусть размер буфера останется 255, на всякий случай.

Буферы не являются обязательными переменными, но с ними проще работать.

```
#define BUFSIZE 255      //ёмкость буфера
unsigned char bufrd[BUFSIZE], bufwr[BUFSIZE]; //приёмный и передающий буферы
```

2) Переменная **COMport** типа HANDLE для хранения дескриптора порта. Так как порт открывается как файл с помощью функции CreateFile, то работа с портом-файлом осуществляется, как и с обычным файлом, с помощью обращения к нему через дескриптор, который и возвращает эта функция. Дескрипторы, используемые функциями WINAPI, имеют тип HANDLE. Дескриптор содержит номер открытого приложением файла (порт открывается как файл).

```
HANDLE COMport;         //дескриптор порта
```

Имейте в виду, что когда вы закрываете порт функцией CloseHandle(COMport), то значение переменной, в которой хранится дескриптор, не обнуляется. Если попытаться закрыть дескриптор ещё раз, функция вернёт ошибку (подробнее об этом см. выше).

Переменная для дескриптора порта является обязательной.

3) Ещё две обязательные переменные – структуры **overlapped** и **overlappedwr** типа OVERLAPPED для потоков чтения и записи. Структура OVERLAPPED необходима для асинхронных операций, при этом для операции чтения и записи нужно объявить разные структуры. Их необходимо объявить глобально, иначе программа будет работать неправильно (протестировано).

```
OVERLAPPED overlapped;        //будем использовать для операций чтения (см. поток ReadThread)
OVERLAPPED overlappedwr;             //будем использовать для операций записи (см. поток WriteThread)
```

4) Приведённые ниже переменные являются необязательными.

Переменная **handle** типа int для хранения дескриптора файла, в который записываются принимаемые данные. Эта переменная является необязательной, так как запись в файл можно не делать.

```
int handle;                   //дескриптор для работы с файлом с помощью библиотеки <io.h>
```

Флаг **fl** типа bool, сигнализирующий об успешности операции записи в порт, выполненной в потоке записи. Используется, чтобы передать в функцию Printing() потока записи результат асинхронной операции записи в порт. Контроль успешности записи в порт можно не использовать, либо передавать в функцию Printing() результат операции каким-либо другим способом* – в этом случае этот флаг не нужен.

```
bool fl=0; //флаг, указывающий на успешность операций записи (1 - успешно, 0 - не успешно)
```

Переменная-счётчик **counter** для хранения количества принятых байтов за время открытия порта. Счётчик сбрасывается при нажатии на кнопку "Открыть порт". Количество принятых байтов выводится в правой части строки состояния. Использовать его не обязательно.

```
unsigned long counter;  //счётчик принятых байтов, обнуляется при каждом открытии порта
```

Описание интерфейса

Интерфейс всех трёх версий программ одинаков, поэтому опишем его на примере первой версии (см. **Рис. 1**).

* Из-за использования метода Synchronize для вызова этой функции передать в неё параметры в качестве аргументов нам не удалось. Авторам статьи в данном случае другой способ кроме глобальных переменных не известен.

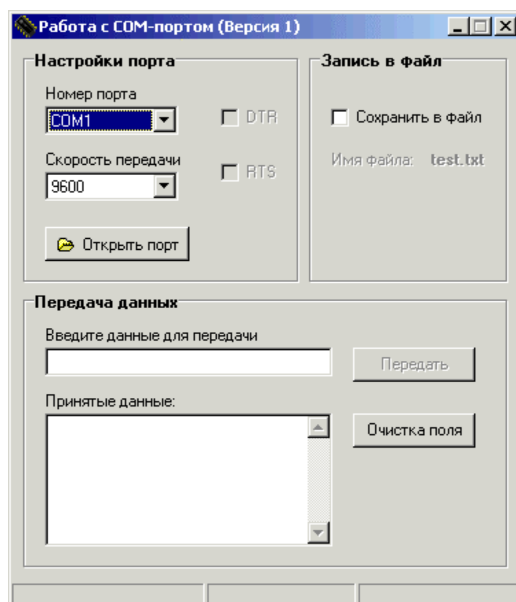

Рис. 1. Интерфейс программы

Интерфейс состоит из одного окна, разделённого на четыре части:

1) Секция "Настройка порта".

Здесь располагаются два списка с настройками порта. Первый список, "**Номер порта**", позволяет выбрать имя порта – COM1 или COM2. Второй список, "**Скорость передачи**", позволяет выбрать скорость передачи в бит/секунду (бод). В этом списке приведена полная линейка скоростей порта – от 110 бод до 115 200 бод.

Оба списка доступны, когда порт не открыт. При открытии порта списки становятся недоступными.

Под списками располагается кнопка "**Открыть порт**". При нажатии на эту кнопку происходит открытие порта с текущими установленными настройками. После нажатия надпись на кнопке меняется на "**Закрыть порт**", и кнопка остаётся нажатой. Чтобы закрыть порт, нужно нажать кнопку ещё раз, тогда она вернётся в исходное состояние, порт будет закрыт.

Сбоку от списков располагаются две галочки сигналов **DTR** и **RTS**. Они доступны, только когда порт открыт. Установкой и сбросом этих галочек можно соответственно устанавливать и сбрасывать сигналы DTR и RTS. При этом состояния сигналов при открытии порта восстанавливаются соответственно состоянию галочек. То есть если вы оставите их включенными, при открытии порта эти линии будут установлены в единицу*.

2) Секция "Запись в файл".

Здесь можно включить сохранение в файл принимаемых из порта данных. Для этого предусмотрена галочка "**Сохранить в файл**". При нажатии галочки активируется надпись с именем файла, и программа будет производить запись данных в файл. Если при открытии файла для записи данных (которое происходит при открытии порта) по каким-либо причинам произошла ошибка, то галочка будет недоступна, а надпись пропадёт. Галочка записи в файл доступна всё время работы с программой, если при открытии файла не произошла ошибка.

Сброс галочки отключает запись данных в файл.

3) Секция "Передача данных".

В этой секции находится **текстовое поле ввода**, в которое можно ввести текст для передачи. Длина вводимого текста ограничена 254 символами в связи с ограничением передающего буфера.

Справа от текстового поля находится кнопка "**Передать**", при нажатии на которую выполняется передача данных из текстового поля. После передачи данных набранная строка остаётся в поле ввода (не стирается). В отличие от поля ввода кнопка "Передать" доступна, только если порт открыт.

* Это не совсем грамотно, так как правильнее будет дать пользователю возможность проконтролировать включение/выключение этих линий до открытия порта. Но не будем усложнять программу.

 © PIClist RUS, 2007

Ниже располагается текстовое окно для отображения принимаемых данных и кнопка "**Очистить поле**", при нажатии на которую текстовое окно очищается. Поле принимаемых данных и кнопка "Очистить поле" доступны всё время работы программы.

4) Строка состояния.

Используется для вывода различной служебной информации, разделена на три части. В первой части выводится результат выполнения операции открытия порта, файла для записи или передачи данных. В средней части выводится индикатор сохранения принимаемых данных в файл, если галочка "Сохранить в файл" включена. В третьей части выводится количество принятых байтов за всё время, пока порт был открыт.

Работа с программой

Запустив программу, вам, прежде всего, необходимо настроить порт – выбрать имя порта и скорость работы с ним. При этом используется следующий формат байта: длина 8 бит, с одним стоп-битом и без бита контроля чётности.

Если желательна запись принимаемых данных в файл, то нужно включить галочку "Сохранить в файл". При этом данные будут сохраняться в файл с именем test.txt в каталоге с программой. Рекомендуется активировать запись в файл до открытия порта, так как приём данных начинается сразу, и если в порт сразу будут поступать данные, они могут не успеть попасть в файл.

Так как файл открывается при открытии порта, то, в случае ошибки открытия файла, запись в файл станет недоступной.

Затем нужно открыть порт, нажав на кнопку "Открыть порт". При этом становятся доступными галочки DTR и RTS. Вы можете устанавливать и сбрасывать их для управления соответствующими линиями.

Также становится доступна передача данных. Чтобы передать какой-нибудь текст, наберите его в поле ввода и нажмите кнопку "Передать", данные будут переданы.

Чтобы закрыть порт, нажмите на кнопку "Закрыть порт".

Как протестировать программу

Для тестирования программы вам понадобятся: диспетчер задач (либо встроенный для Windows 2000 (см. **Рис. 2**), либо в виде какой-нибудь утилиты (например, в составе TurnUp Utilities)), а также заглушка для COM-порта.

Диспетчер задач

Диспетчер задач вам понадобится для наблюдения за корректным управлением потоками и дескрипторами.

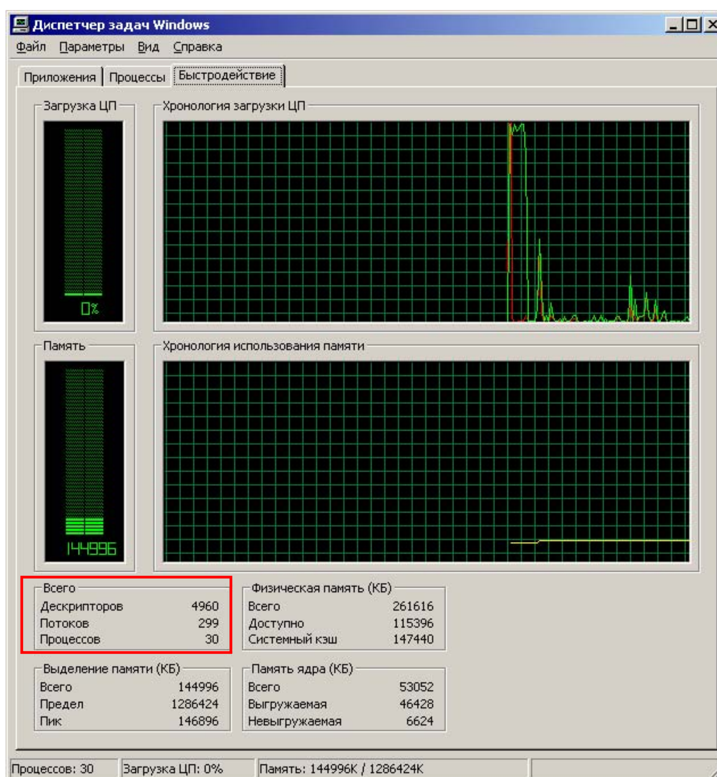

Рис. 2. Окно диспетчера задач Windows 2000

В окне диспетчера задач нас интересует следующее:

1) **Загрузка процессора** – по ней видно, что в случае использования усыпления потоков, они почти не загружают процессор.

2) **Окошко "Всего"** – здесь содержится информация о количестве открытых дескрипторов, процессов и потоков. Пользуясь этим окошком при отладке программ, мы можем отследить, правильно ли создаются и уничтожаются потоки, закрываются и открываются дескрипторы. Например, таким способом авторы отлавливали ошибки по закрытию потоков – при нажатии кнопки создавалось два потока, а уничтожался только один, и, таким образом, при каждом включении/выключении порта число потоков в системе увеличивалось. Такая же ситуация наблюдалась с дескрипторами – когда дескриптор события или файла освобождался не там, где надо, и в результате количество дескрипторов также увеличивалось.

В принципе, после закрытия программы, все дескрипторы и потоки, связанные с ней, автоматически уничтожаются. Но лучше, если вы будете полностью контролировать дескрипторы и потоки, созданные программой, и не позволять вашей программе "мусорить". Так вы сможете избежать различных ошибок, и программа будет работать стабильнее.

Ну а количество процессов вам пригодится, если вы кроме потоков работаете ещё и с процессами. В данном же случае здесь можно посмотреть, как на количество процессов влияет наша программа. При её запуске должен создаваться один процесс, а при её закрытии – уничтожаться.

Учтите также, что при открытии программы обязательно создаётся один поток – это главный поток программы, а также несколько дескрипторов, в число которых входят и дескрипторы элементов управления, расположенных на форме.

Заглушка для COM-порта

В качестве аппаратного вспомогательного средства для тестирования программы вам понадобится заглушка для COM-порта. Она представляет собой разъём DB-9-F, у которого контакты 2 (RD) и 3 (TD) соединены проводом (который желательно припаять). Эта заглушка вставляется в нужный COM-порт. Работает она очень просто – когда программа посылает в COM-порт какие-то данные, они через RX тут же попадают на TX и программа сразу же их принимает. То есть получается что-то наподобие эха.

Недостаток этой заглушки в том, что на ней будет сложно протестировать обмен с устройством сложными пакетами данных, для которых нужно формировать определённые ответы. Для этого лучше иметь устройство, которое может формировать пакеты и отвечать на принимаемые данные. Либо иметь второй компьютер, COM-порт которого нужно соединить с COM-портом другого с помощью нуль-модемного соединения, и на обоих запустить программу, в которой затем вручную формировать нужные пакеты.

На **Рис. 3** приведены схемы контактов разъёма и соединения выводов для получения заглушки.

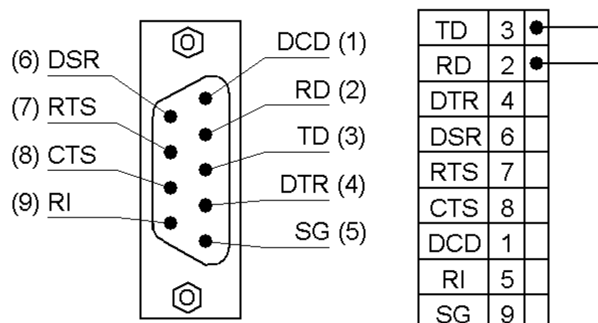

Рис. 3. Схемы контактов разъёма DB-9 и соединения контактов заглушки

Теперь вы сможете написать и протестировать свою программу работы с COM-портом. На этом и закончим нашу статью.

Список литературы

1) Агуров П. "*Последовательные интерфейсы ПК. Практика программирования*".

Книга посвящена в основном COM-порту и интерфейсу RS-232. Книга содержит теоретическую информацию о последовательных интерфейсах, примеры кодов для работы с COM-портом, а также справочник по функциям для работы с COM-портом (занимает достаточно большую часть книги).

Достоинства книги:

- именно кусок кода по работе с потоками из этой книги помог нам написать программу, а затем и эту статью.
- неплохой справочник по функциям.
- работа с драйвером IO.
- немного теоретической информации поможет понять, что такое интерфейс RS-232 (если только не запутает читателя).

Недостатки:

- автор практически не рассматривает аппаратную часть RS-232.
- коды программ написаны на Delphi и почти не имеют пояснений. Будто бы автор торопился выпустить книгу.

Эту книгу можно использовать для начального изучения COM-порта.

2) М.Титов. *Статья*.

Статья посвящена программной работе с COM-портом. В ней очень хорошо рассмотрена инициализация порта, а также всякие хитрые функции для работы с портом. Приведены подробные описания функций WINAPI (взятые из help, но хорошо переведённые и снабжённые комментариями). Также рассматриваются функции для приёма-передачи данных (ReadFile и WriteFile). Но передаче данных уделяется меньше внимания, а о перекрываемых (overlapped) операциях написано непонятно. А про работу с потоками вообще только упоминается.

Эту статью можно использовать для начального изучения работы с COM-портом, а также чтобы разобраться с инициализацией и другими интересными функциями.

3) Архангельский А.Я, Тагин М.А. "*Приёмы программирования в C++ Builder. Механизмы Windows, сети*". В этой книге освещаются некоторые вопросы по работе с потоками, сигнальными объектами и COM-портами, а также много других интересных приёмов работы в C++ Builder.

 © PIClist RUS, 2007

4) *Help к программе Borland C++ Builder 6.0*. Благодаря ему авторы данной статьи смогли разобраться во всём, о чём здесь написали. Материал в нём изложен в краткой и понятной форме, при этом оставаясь довольно содержательным. Одно "но" - всё на английском языке.

 © PIClist RUS, 2007