Павел Агуров

bhv

Последовательные интерфейсы ПК

Практика программирования

+ CD-ROM

- Протоколы последовательной связи
- Стандарты RS232/RS485
- Работа в DOS, Windows 98/ME/NT/2000/XP
- Примеры на языке Pascal
- Все уровни доступа
- Поддержка Plug and Play

МАСТЕР_ПРОГРАММ_

УДК 681.3.06
ББК 32.973-018
А23

Агуров П. В.

А23 Последовательные интерфейсы ПК. Практика программирования. — СПб.: БХВ-Петербург, 2004. — 496 с.: ил.

ISBN 5-94157-468-1

Книга представляет собой практическое руководство по программированию последовательных интерфейсов. В первой части книги представлены теоретические сведения о последовательных интерфейсах, во второй — практические примеры и листинги программ на языках Pascal и Delphi, а третья содержит справочные данные, облегчающие поиск необходимой информации. В приложениях приведены дополнительные сведения и ответы на часто задаваемые вопросы. Большое количество практических советов, примеров программ, а также последовательность и простота изложения позволят читателю уверенно овладеть изложенным в книге материалом. Для удобства читателей все исходные коды приводятся на прилагаемом к книге компакт-диске.

Для программистов, занятых в сфере промышленной автоматизации

УДК 681.3.06
ББК 32.973-018

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Алексей Семенов*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Оформление серии	*Via Design*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 13.02.04.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 39,99.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02 от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-94157-468-1

Содержание

Глава 2

Протоколы

"Теперь позвольте пару фраз без протокола..."

В. Высоцкий, "Милицейский протокол"

В этой главе описываются типы протоколов обмена, методы организации передачи данных и методы повышения достоверности передачи.

2.1. Что такое протокол обмена

Чтобы принимающее устройство умело правильно интерпретировать полученные данные, необходима договоренность между приемником и передатчиком — *протокол обмена*. Протокол обмена состоит из нескольких частей:

- договоренность о параметрах связи (скорость, четность, число бит в байте и т. д.);
- договоренность о последовательности передачи данных и форматах представления чисел;
- договоренность об инициаторе обмена (если необходимо);
- процедура восстановления связи (если необходимо);
- дополнительные договоренности (например, передача прав главного хоста в сетях RS-485).

Параметры связи обязывают приемник и передатчик работать с одинаковыми аппаратными настройками. Обычно промышленные устройства используют строго определенные настройки обмена, некоторые — позволяют задавать скорость обмена (как программно, так и с помощью перемычек). Настройки четности и число бит в байте чаще всего фиксируются и не могут настраиваться со стороны аппаратуры.

Иногда, для согласования скорости работы, устройства выбирают фиксированную скорость начала обмена, устанавливают соединение и, обмениваясь данными на этой скорости, "договариваются" об использовании других скоростей и параметров обмена.

Для того чтобы принимающее устройство могло воспользоваться полученной информацией, ему необходимо знать, как ее интерпретировать. Для этого приемник и передатчик декларируют последовательность передачи данных. Часто именно эту часть договоренностей называют протоколом обмена, подразумевая наличие остальных частей.

При использовании приемником и передатчиком одной линии связи (как, например, в интерфейсе RS-485), необходима договоренность об инициаторе обмена. Это тем более необходимо в тех случаях, когда приемник один, а передатчиков много.

При разрыве связи одно из устройств может ждать ответа другого, а то, в свою очередь, ответа первого. Таким образом, без договоренности о действиях в случае разрыва связи устройства не смогут связаться друг с другом повторно. Часто выбирается очень простой способ — если в течение некоторого времени устройства не получают данных друг от друга, они переходят в режим "первого запуска". Однако возможны и более экзотические договоренности.

С точки зрения формата передаваемых данных, типы протоколов обмена можно поделить на две категории — строковые и бинарные. В первом случае информация передается обычными ASCII-символами (American Standart Code for Informational Interchange, американский стандартный код для обмена информацией), а во втором — бинарным потоком данных. Далее мы рассмотрим оба эти типа, а их реализацией займемся в *гл. 14*.

2.2. ASCII-протокол передачи данных

В таком протоколе управляющие символы и данные передаются с помощью ASCII-символов.

Пример пакета данных в таком формате приведен в табл. 2.1. Пакет данных начинается со стартовой последовательности — одного или нескольких символов, обозначающих начало данных. Символы стартовой последовательности не должны появляться внутри блока данных. Получив байты начала посылки, все абоненты настраиваются на получение полного пакета данных. Затем, получив и сравнив номер абонента со своим номером, абонент либо продолжает обработку пакета, либо снова переходит в режим ожидания начала посылки. Таким образом, пакет обрабатывает только один абонент сети. После получения байта конца посылки абонент проверяет контрольную сумму и обрабатывает полученные данные. Конец посылки также может представляться несколькими символами. Разумеется, строка конца посылки не должны встречаться внутри самих данных.

Естественно, если связь осуществляется между двумя абонентами, поле "номер абонента" может отсутствовать.

Таблица 2.1. *Структура ASCII-протокола*

Байт	Описание	Примеры	
Начало посылки	Идентификатор начала посылки. Любой символ (или несколько символов), не встречающийся в символах данных	$	!!!
Номер абонента	Две десятичные цифры номера абонента (дополненные при необходимости нулем). В некоторых случаях могут оговариваться специальные номера абонентов, например, "**", означающая посылку всем абонентам сети	01	15
Данные	Последовательность цифр данных. Числа с плавающей точкой разделяются знаком ".". Части данных могут разделяться пробелом	011 03.1	155 17.5
Контроль	Контрольная сумма данных. Обычно один или два байта суммы всех байт данных с переполнением. Байты суммы также передаются в десятичном виде (три десятичные цифры с выравниванием нулями слева)	002	104
Конец посылки	Идентификатор конца посылки. Для ASCII-протоколов обычно символ LR (перевод строки)	$0D	***

Достоинство такого протокола в удобстве отладки: прием и передачу команд и данных можно осуществлять с помощью обычной программы-терминала (например, Telnet).

Основной недостаток — большой объем данных, необходимых для передачи. Например, для передачи одного байта шестнадцатеричной строкой потребуется два байта (число 230 будет передано последовательностью $E, $6). А при передаче обычными десятичными цифрами — три байта (число 230 будет передано тремя цифрами "2,3,0"). Передача же чисел с плавающей точкой потребует количество байт, пропорциональное точности передаваемого числа (плюс десятичный разделитель). Например, число 221,73 будет передаваться с помощью последовательности символов "2,2,1,.,7,3", т. е. с помощью 6 байт.

Второй недостаток — необходимость преобразования двоичных данных в ASCII-строки и обратно. Если на языке высокого уровня (C, Pascal) это не проблема, то, например, для 8-битового контроллера может потребоваться значительный объем кода (пример кода, реализующего такое преобразование приводится в *прил. 8*).

Хорошим примером ASCII-протокола является MODBUS-ASCII *(см. разд. 4.1.1)*, а программной реализацией своего ASCII-протокола мы займемся в *гл. 14*.

2.3. Бинарный протокол передачи данных

Бинарная передача значительно более компактна. Например, для передачи числа с плавающей точкой типа `SINGLE` *(см. прил. 3)* требуется 4 байта независимо от числа знаков после десятичной точки.

Так же как в ACSII-протоколе, выбирается некоторый символ для обозначения начала посылки. Завершение ожидания данных производится либо по получению необходимого числа байт, либо по тайм-ауту: принимающее устройство отсчитывает время с получения последнего байта до приема следующего. Если эта пауза превышает некоторый интервал (обычно 1,5—2 байта), посылка считается потерянной и приемный буфер сбрасывается. После получения необходимого числа байт (или символа конца посылки) проверяется правильность полученных данных и производится их обработка.

Примером бинарного протокола может служить протокол MODBUS-RTU *(см. разд. 4.1.2)*, а реализацией своего бинарного протокола мы займемся в *гл. 14*.

2.4. Предотвращение потери данных

Причины потери данных при передаче или приеме можно разделить на две категории: аппаратные и программные.

К *аппаратным причинам потери данных* относятся помехи на линии, некачественный проводник, несоответствие длины провода и скорости передачи и т. д. Бороться с такими причинами, конечно, надо, но не на программном уровне. Правила прокладки сетей и расчет максимальной длины проводника можно найти в соответствующей литературе, например, см. в [2] и [9]. От программной составляющей в этом случае требуется только определить, что данные были переданы с ошибкой. Способы обнаружения ошибок мы рассмотрим дальше.

Потеря данных может возникать и при неправильной организации программы. Последовательная передача подразумевает, что все биты, посланные с одной стороны, будут приняты с другой. Более того, во время передачи и приема байта нельзя допускать никаких задержек: если байт не считан вовремя, он теряется и его место занимает следующий переданный байт. Аналогично, если хотя бы часть пакета данных будет потеряна, это, чаще всего, означает потерю всего пакета данных.

2.4.1. Прерывания и потоки

Основная причина потери данных — программа занята обработкой других данных и не опрашивает порт. Возможно, в этот момент идет обработка предыдущего полученного пакета данных или программа занята диалогом с

пользователем. Все эти причины сводятся к одной — неправильной организации программы.

Рассмотрим, например, алгоритм, приведенный в листинге 2.1.

Листинг 2.1. Неправильная организация опроса порта

```
Begin
 While (True) do begin
  { Опрос клавиатуры }
  If KeyPressed then
   Case ReadKey of
    #27: Break; {выход}
    'q': Begin
      Write('Введите K'); ReadLn(K);
    End;
   End;
  End;

  { Опрос порта }
  DoReadPort;

  { Есть новые данные! }
  If DataAvailable then begin
   WriteLn('Значение температуры:', Code_ADC * K);
   DataAvailable:= False;
  End;

 End;
End.
```

При всей заманчивой простоте такого кода так делать не надо! Все будет хорошо, пока пользователь не нажмет клавишу <q>, вызывающую диалог ввода нового коэффициента. А вот во время диалога порт опрашиваться не будет. В программе для Windows можно сделать аналогичную ошибку, вызвав метод `DoModal` формы диалога параметров.

Конечно, в Windows можно воспользоваться компонентом `TTimer` и делать опрос порта в обработчике `OnTimer` с некоторой фиксированной частотой. Однако и этот способ далек от совершенства. Дело в том, что, во-первых, при большой скорости передачи данных буфер приема драйвера порта может переполниться, и часть данных будет потеряна. А, во-вторых, таймер

работает с помощью посылки сообщения WM_TIMER окну программы. Это сообщение имеет низший приоритет, и вероятность того, что считывание порта будет произведено вовремя, достаточно мала. В Windows 95/98 достаточно, щелкнув на заголовке окна, удерживать кнопку "мыши" в нажатом состоянии — и сообщения WM_TIMER не будут обрабатываться. Таким образом, если пользователь будет перетаскивать окно на другое место, порт опрашиваться не будет. И еще один недостаток: несмотря на то, что редактор свойств позволяет установить свойство Interval (частоту вызова обработчика таймера) в любое значение, большее нуля, реально работающим минимальным значением будет 70—100 мс.

Раскритиковав все способы работы, стоит предложить какой-нибудь правильный. Для DOS таким способом является обработка прерываний, рассматриваемая в гл. 8. А для Windows правильным способом обработки порта является создание отдельного потока для чтения данных, как будет показано в *гл. 11* и *12*.

2.4.2. Буферизация

При большой скорости передачи данных, очевидно, может возникнуть ситуация, когда время на обработку полученного пакета данных (например, рисование графика, анализ данных и сохранение их в файл) больше, чем время между получением пакетов.

В таких случаях необходимо некоторое промежуточное хранилище полученных данных — буфер.

Буферизация позволяет основному модулю не заботиться о скорости обработки данных. Конечно, размер буфера тоже ограничен, но обычно его выбирают достаточно большим.

На уровне приема одного байта существуют свои буферы приема:

- в DOS можно включить специальный режим буферизации FIFO;
- Windows имеет свои буферы, размер которых можно изменять вызовом функции SetupComm.

Однако чаще всего программе важнее иметь свой буфер, хранящий полные пакеты данных, а системные буферы использовать как дополнительный метод повышения надежности *(см. гл. 14)*.

Единственная сложность буферизации — необходимость синхронизации процедуры выборки данных из буфера и процедуры добавления в буфер. Синхронизация необходима, т. к. в DOS добавление в буфер будет производиться из прерывания, а в Windows — из другого потока. В обоих случаях вероятность того, что основная программа будет прервана на обновление буфера в середине процедуры выборки данных, достаточно высока. Очевидно, в таком случае возникнет ошибка.

Синхронизация в DOS

В DOS синхронизация возможна только одним способом — запрещением прерываний на время выборки данных из порта, как показано в листинге 2.2.

Листинг 2.2. Синхронизация в DOS

```
{Обработчик прерывания}
Procedure OnCOMInterrupt(B : Byte);
Begin
 { Добавить байт в буфер }
 AddByteToBuffer(B);
End;

Var
 B : Byte;
Begin
 {Основной цикл программы}
 While (True) do begin
  {Запретить прерывание}
  DisableIRQ;
  {Забрать байт, если он есть}
  If GetSizeBuffer > 0 then B:= GetByteFromBuffer;
  {Разрешить прерывания}
  EnableIRQ;

... обработка нового байта B ...

 End;
End.
```

Обратите внимание, что обработка байта производится вне блока [`DisableIRQ`, `EnableIRQ`]. Таким образом, запрещение прерываний происходит на очень маленький промежуток времени выборки данных из буфера.

Синхронизация в Windows

В Windows существуют свои методы синхронизации. Для синхронизации двух потоков нужно создать специальный объект синхронизации, например, `TCriticalSection`. Скелет такой программы приводится в листинге 2.3.

Листинг 2.3. Пример синхронизации в Windows

```
Uses SyncObjs; {нужно для использование объектов синхронизации}
Var
 CS : TCriticalSection;

{Создание объекта синхронизации}
procedure TForm1.FormCreate(Sender: TObject);
begin
 CS:= TCriticalSection.Create;
end;

{Уничтожение объекта синхронизации}
procedure TForm1.FormDestroy(Sender: TObject);
begin
 CS.Free;
end;

{Добавление байта в буфер}
procedure TThread1.DoAddToBuffer(:);
begin
 CS. Acquire;
 Try
   :добавить в буфер:
 Finally
  CS. Release;
 End;
end;

{Выборка байта из буфера}
procedure TThread2.DoGetByteFromBuffer(:);
begin
 CS. Acquire;
 Try
   :забрать из буфера:
 Finally
  CS. Release;
 End;
end;
```

Подробнее про объекты синхронизации можно прочитать в [11].