RocketPort/ISA API (6508) for the MS-DOS Operating System

First Edition, December 1993
Revised April 1994

Trademarks

Document Number: 6508D1.ELE

Before You Begin

Scope

This guide describes the following information about the DOS application program interface (API) for RocketPort controllers:

- Installing the software and hardware
- Running the sample application
- Developing applications

***Note**: If you want to install the Interrupt 14 device driver, see the Reference Card.*

Purpose

This guide explains installing and using the API functions.

Audience

This guide is for people who develop applications for the MS-DOS system.

Prerequisites

This guide assumes that you are running an ISA-based personal computer with the following:

- MS-DOS operating system (level 5.0 or higher)
- One of the following compilers:
  - A Borland® C++ compiler (level 3.1 and higher)
  - A Microsoft® C/C++ compiler (level 7.0 and higher)

Suggestions

Use Chapter 1 to install the API. Use Chapter 2 and Appendix A to develop applications that run with the API. If you have any problems, see Chapter 3.

Organization

Section 1. Installing RocketPort Systems
Provides you with the following information:

- Product introduction
- Software and hardware installation overview
- Installing the software and hardware
- Configuring controllers
- Running the sample application

Section 2. Developing Applications
Provides you with information about how to develop applications using the API.

Section 3. Troubleshooting and Technical Support
Provides you with troubleshooting and technical support information for your RocketPort series controller.

Appendix A. API Functions
Contains the API asynchronous functions available for writing the application.

Appendix B. Double Buffering Example
Illustrates the double buffering example on your diskette.

Software or Document Changes

For information that is not in this guide, see the **README.API** file on the software diskette. If this file is empty, that means that this guide reflects the API on the diskette.

Table of Contents

Examples

Flowcharts

Tables

Section 1. Installating RocketPort/ISA Systems

This section contains a product overview and discusses installing the API for your system. The DOS API and Interrupt 14 device driver are delivered on one diskette.

***Note**: See the Int 14 Reference Card for Interrupt 14 information.*

1.1. Product Introduction

The RocketPort multiport serial controller series fits into a 16-bit ISA slot of a personal computer. The RocketPort series uses a 36 MHz processor specifically designed to process asynchronous serial communications, thereby maximizing performance and eliminating bottlenecks.

RocketPort series uses Application Specific Integrated Circuits (ASICs) technology to replace most hardware components, including:

- The processor
- Serial controller
- Bus interface logic and other miscellaneous logic

The RocketPort series is I/O mapped eliminating memory mapping conflicts.

The RocketPort series supports RS-232 or RS-422 mode and connects easily to the interface box or your peripherals, depending on the type of RocketPort controller you purchased.

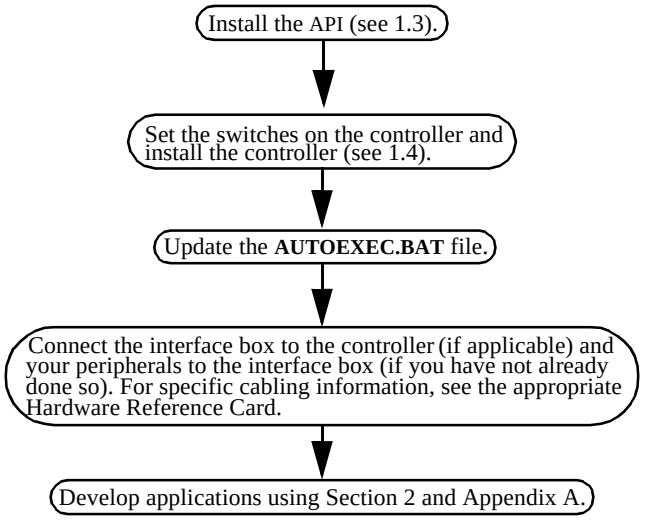
The device driver supports up to four RocketPort controllers (128 ports) in one PC. You can install any combination of the series, which includes the following:

Name	Number of Ports	Interface Type
RocketPort 4*	4	Requires interface box
RocketPort 4J	4	RJ45 cables not included
RocketPort Quadcable*	4	Includes a fanout cable with standard DB25 or DB9 connectors
RocketPort 8	8	Requires interface box
RocketPort 8J	8	RJ11 cables not included
RocketPort Octacable	8	Includes a fanout cable with standard DB25 or DB9 connectors
RocketPort 16	16	Requires 16-port interface box (Standard DB25 or Rack Mount RJ45 available)
RocketPort 32	32	Requires two 16-port interface boxes (Standard DB25 or Rack Mount RJ45 available)

The RocketPort series is easy to install using Subsection 1.2.

1.2. Software and Hardware Installation Procedures

Use Flowchart 1-1 for an overview of installing a RocketPort series system.



Flowchart 1-1. Hardware and Software Installation Overview

***Note**: If you have an installation or operations problem, see Chapter 3.*

1.3. Installing the Software

You may want to install the API in a directory named **\ROCKET** so that the examples illustrated in this guide match your environment.

The following shows a sample installation onto your hard disk (assuming the hard disk is drive C):

1. Insert the Comtrol *API and Device Driver for MS-DOS* diskette into the appropriate drive.
2. Change to the drive that you installed the diskette on.
3. Enter the following:

   install

4. Select the **API** button by pressing <**Enter**> or <**Click**>. <**Click**> means that you should move the cursor over the item and press the mouse button.

***Note**: Press <F1> on any item for button-sensitive Help.*

5. Select the I/O address range for each RocketPort series controller.
   a. Use the<**Tab**> key and the <**ALT**> <**Down Arrow**> key combination or <**click**> on the arrow next to the I/O Address Range box to view the I/O address ranges.
   b. Use an <**Arrow**> key or the mouse cursor to highlight the I/O range you want to select.
   c. Press <**Enter**> or <**Click**> to execute the selection.

   The I/O address identifies the location in the system's I/O space used to pass control information between the system and the controller.

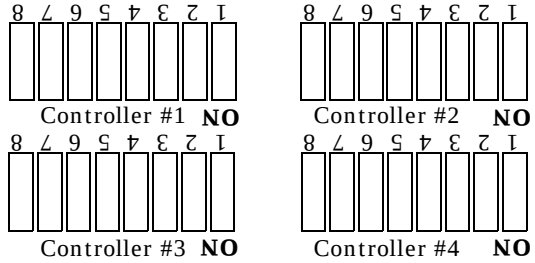
   For the first controller, you will select a 68-byte I/O address range. For subsequent controllers, you will select a 64-byte range.

   Most peripherals use I/O address ranges between 0 and 3FF hexadecimal. If you have peripherals installed above 400h, you may experience an I/O conflict.

   RocketPort controllers use I/O address ranges at 400h intervals above the I/O address range. Make sure that other peripherals in the system do not use these I/O address ranges. See Table 3-1 for information about common I/O usage.
4. Enter a path name for the API directory, if you do not want to use the default path, **\ROCKET**.
5. Select an interrupt (IRQ) for the controller that does not conflict with an existing interrupt.
6. Select <**Ok to Install**>.
7. Select <**OK**> at the confirmation screen.
8. Set the DIP switches on the controller as directed in the summary screen.

   You may want to fill in the blank switches provided for you or place a check mark in Table 1-1, which illustrates common I/O ranges.

   Press <**ENTER**> to view the DIP switch settings for additional controllers.



***Notes**: You may want to set the DIP switches for the controllers while looking at the summary screen.*

You can also use the **\ROCKET\INSTALL.LOG** *file to set the switches, if you do not set them at this time.*

9. Make sure that you note the line that you must add to the **AUTOEXEC.BAT** file. For example:

   SET ROCKETCFG=C:\ROCKET\CONFIG.DAT

   This path is the same path where the API is installed.

   ***Note**: After you create your own applications, you may need to change the configuration file (see Subsection 2.3).*
10. When your cursor returns to the DOS prompt, remove the diskette from the drive.
11. Edit the **AUTOEXEC.BAT** file as directed in Step 9.

Go to the next subsection to install the controller.

Table 1-1. Common Switch Settings

Controller #1 I/O Address Range	DIP Switch Settings Controller #1 determines other controller settings
100 - 143 hex	1st ISA NO; 2nd ISA NO; 3rd ISA NO; 4th ISA NO
140 - 183 hex	1st ISA NO; 2nd ISA NO; 3rd ISA NO; 4th ISA NO
180 - 1C3 hex (Default)	1st ISA NO; 2nd ISA NO; 3rd ISA; 4th ISA

Table 1-1. Common Switch Settings (Continued)

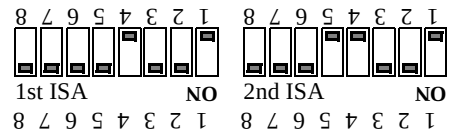
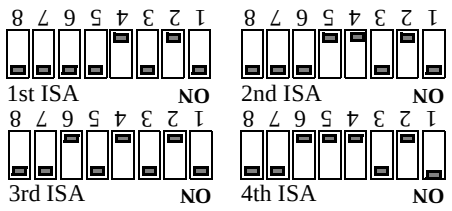
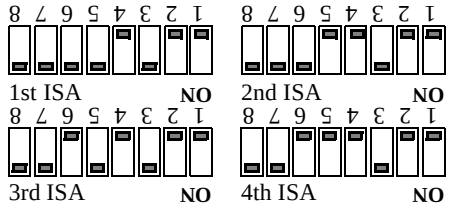
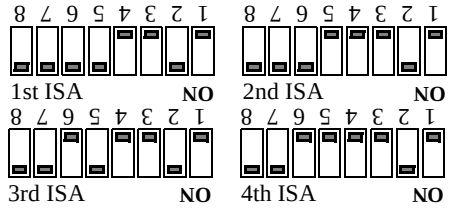
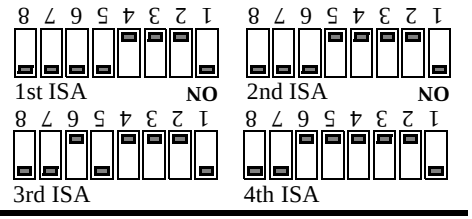
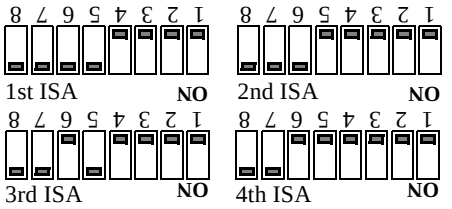
Controller #1 I/O Address Range	DIP Switch Settings Controller #1 determines other controller settings
200 - 243 hex	
240 - 283 hex	
280 - 2C3 hex	
300 - 343 hex	
340 - 383 hex	

Table 1-1. Common Switch Settings (Continued)

Controller #1 I/O Address Range	DIP Switch Settings Controller #1 determines other controller settings
380 - 3C3 hex	

1.4. Installing the Controller

To prepare your controller for installation, you may need to set the I/O address DIP switch. The default I/O address range is 180 through 1C3. You must change the I/O address settings on any additional controllers, even if you select the default address range.

If you did not set the DIP switch on the controller or controllers during the software installation, do so at this time. Make sure that you set each controller as advised during the software installation or use the information in the **\ROCKET\INSTALL.LOG** file.

After you set the I/O DIP switch, you are ready to install the controller. Use the following steps to install the controller:

1. Turn the power switch for the system unit to the OFF position.
2. Remove the system unit cover.
3. Select a slot to install the controller.
4. Remove the expansion slot cover.
5. Insert the controller in the expansion slot, make sure that it is properly seated.
6. Attach the controller to the chassis with the expansion slot screw. Repeat Steps 3 through 5 for each controller.
7. Replace the cover on the system unit.

If connecting a system with an interface box:

a. Attach the male end of the RocketPort cable to the controller and the female end to the connector on the interface box labeled *Host*.

Note: *If you have a RocketPort 32, the connector labelled J1 corresponds to ports 0 through 15 on the interface box and the connector labeled J2 (closest to the bus) corresponds to ports 16 through 31.*

b. Connect the peripherals to the interface box.

Note: *The ports on the interface box are numbered from 0 to 3, 7, or 15 on the standard DB25 interface. The Rack Mount RJ45 interface is numbered*

from 1 to 16.

c. If applicable, set each port to the appropriate communications mode (RS-232 or RS-422) for your peripheral using the slide switch.

If connecting a system with a Quad/Octacable:

a. Attach the male end of the Quad/Octacable to the controller.

b. Connect the Quad/Octacable to the peripherals.

If connecting a RocketPort 4J or 8J controller:

a. Connect your peripheral devices to the RJ style connector on the controller.

After installing and configuring the controller, you are ready to attach your peripherals. Refer to the *Hardware Reference Card* if you need information about the pinouts for the connectors.

After connecting the peripherals, you can go to the next subsection to run the sample application. The sample application shows you how to use the API.

Use Section 2 and Appendix A to develop applications.

1.5. Running the Sample Application

The sample program, **TERM**, is a simple terminal emulator program which uses one RocketPort port at a time. **TERM** uses an ASCII terminal connected to port 0 of the Comtrol interface box with an RS-232 cable. This allows testing for both transmit and receive. The terminal should be configured for 9,600 baud, 8 data bits, 1 stop bit, and no parity.

Optionally, if you do not have an available terminal to run the sample application, you can use the loopback plug that came with your controller.

***Note**: If your configuration is different, you must change the parameters in the* **aaOpen** *call to match your requirements. Make sure you recompile before running the sample program.*

Use the following procedure to run the sample program:

At the DOS prompt, change to the **c:\ROCKET\SAMPLE** directory.

12. Type **TERM** at the DOS prompt. The following displays:

    Serial Device Number:

    Optionally, insert the loopback plug in Port 0 of the interface box.

13. Type **0** and then press <**Enter**>. The following displays:

    Serial Device Number 0 **Hit F10 to Quit**

    The **TERM** application allows you to type any character on the PC keyboard and have it appear on the terminal, and type any character on the terminal and have it appear on the PC screen.

    Optionally, if you are using the loopback plug, any character that you type on the keyboard appears on the screen.

14. Enter several characters using the PC keyboard. You should see these keystrokes appear on the ASCII terminal.

15. Enter several characters using the ASCII terminal keyboard. You should see these keystrokes appear on the PC screen.

If the sample fails, see Section 3.

Use Section 2 and Appendix A to develop applications.

Section 2. Developing Applications

This section describes the following topics:

- API features and functions
- Writing the configuration file
- Using the API (flowchart and example)
- Include files
- Configuring the controllers
- Using API calls to write the application
  - Understanding device numbers
  - Configuration parameters for serial devices
- Writing serial data
- Exiting the application
- Reading serial data
- Installing and detecting events
- Building applications

2.1. API Features

The API contains the following features:

- Supports up to 4 RocketPort controllers in a PC.
- Supports up to 32 serial devices per controller.
- Provides baud rates from 50 to 230.4 K baud.
- Supports all modem control lines available on the controller.
- Provides detection of modem control line changes.
- Provides direct control of modem control outputs.
- Provides direct read of modem control inputs.
- Provides detection of receive errors:
  - Parity
  - Receiver overrun
  - Framing
  - Buffer overflow
- Supports 1K bytes of receive data buffering and 256 bytes of transmit data buffering.
- Supports hardware (RTS/CTS) flow control.
- Supports software (XON/XOFF) flow control.
- Provides read counts of buffered transmit and receive data.
- Provides send break and receive break detection.
- Provides installable event functions for the following:
  - Receive data available
  - Modem control changes
  - Periodic event

For information about event functions see Subsection 2.12.

2.2. API Functions

Table 2-1 lists API functions that are available to a system application. For detailed information about the functions, see Appendix A.

Table 2-1. API Functions

Function Name	Description
aaChangeModemState	Changes the state of modem output lines.
aaClose	Closes a device.
aaEnPeriodicEvent	Enables/disables dispatching of periodic event function.
aaExit	Performs cleanup when exiting application.
aaFlush	Flushes transmit or receive buffer, or both.
aaGetCtlStatus	Gets controller status.
aaGetModemStatus	Gets modem status.
aaGetRxCount	Gets count of data bytes available in receive buffer.
aaGetRxStatus	Gets status of receive buffer.
aaGetTxCount	Gets count of data bytes in transmit buffer.
aaInit	Executes controller and API initialization.
aaInstallCtrlCHandler	Installs a handler for the **CTRL+C** key interrupt.

Table 2-1. API Functions (Continued)

Function Name	Description
aaInstallMdmChgEvent	Installs an event function to handle modem change events.
aaInstallPeriodicEvent	Installs a periodic event function.
aaInstallRxEvent	Installs an event function to handle Rx data available events.
aaOpen	Opens a device for reading or writing, or both.
aaRead	Reads serial data.
aaReadWithStatus	Reads serial data and status.
aaReconfigure	Reconfigures communication parameters.
aaSendBreak	Sends a break signal.
aaSetCloseDelay	Sets the maximum **aaClose**() transmit drain delay.
aaWrite	Writes serial data.
EvModemChange*	Modem control input change event function.
EvPeriodic*	Periodic event function.
EvRxData*	Receive data available event function.

* *These are not part of the API, but are part of the application.*

2.3. Writing the Configuration File

The configuration file is used by the **aaInit**() function to obtain information about all the RocketPort controllers installed in the system. The **aaInit**() function checks the **ROCKETCFG** environment variable to determine the name and path of this file.

When you installed the API, the configuration file was created for you, and you were instructed to place the following line in your **AUTOEXEC.BAT** file:

SET ROCKETCFG=*filepath*

where ***filepath*** is the complete path to the configuration file. This path is the same path where the API was installed.

The initial configuration file allows you to run the sample application program (**TERM**), but when you create and distribute your own application you may wish to use a different configuration file.

The configuration file contains between two and five lines:

- The first line gives the IRQ number that is used by all RocketPort controllers.
- The second through fifth lines give the starting I/O addresses for the first through the fourth controllers

The first controller uses a 68-byte block of I/O address space, subsequent controllers use 64-byte blocks. I/O address lines should be placed in the file only for controllers that are actually installed in the system. Table 2-2 summarizes this information.

Table 2-2. Configuration File Parameters

Line Number	Parameter	Allowable Values	Block Size
1	IRQ number	3, 4, 5, 9, 10, 11, 12, 15	NA
2	Ctrl 1 I/O	100, 140, 180, 1C0, 200, 240, 280, 2C0, 300, 340, 380	68 bytes
3	Ctrl 2 I/O	100, 140, 180, 1C0, 200, 240, 280, 2C0, 300, 340, 380	64 bytes
4	Ctrl 3 I/O	100, 140, 180, 1C0, 200, 240, 280, 2C0, 300, 340, 380	64 bytes
5	Ctrl 4 I/O	100, 140, 180, 1C0, 200, 240, 280, 2C0, 300, 340, 380	64 bytes

Each RocketPort controller uses up to three additional “alias” I/O address ranges located at 400h intervals above the address ranges described above.

For example, if the first controller is addressed at 100, the I/O address ranges used by that controller are:

- 100 - 143
- 500 - 543

- 900 - 943
- D00 - D43

This is normally of no concern because ISA peripherals often use only 10 bits of I/O addressing, meaning they are limited to addresses below 400h.

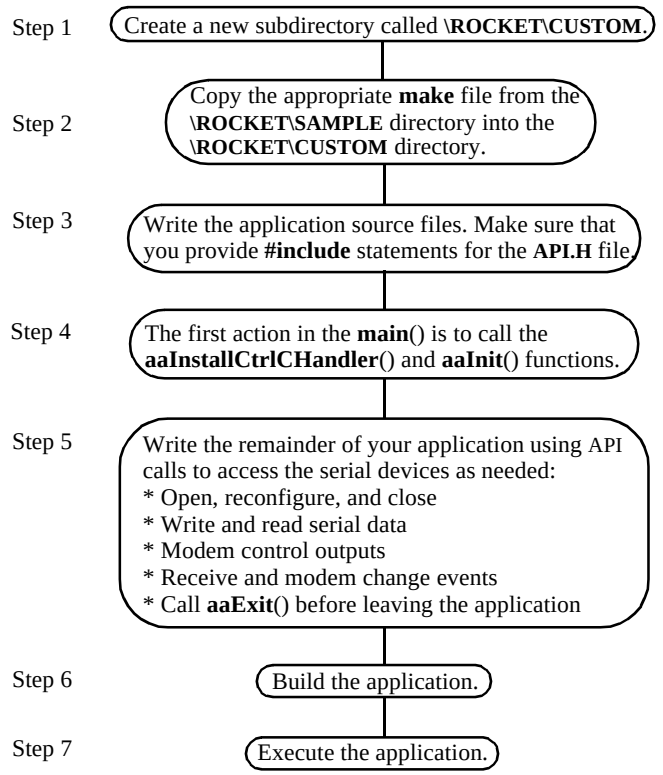
***Note**: In order for your application to locate the configuration file, the* **ROCKETCFG** *environment variable must point to it. This is done with the* **DOS SET** *command, usually placed in the* **AUTOEXEC.BAT** *file.*

2.4. Flowchart for Using the API

This subsection contains the steps required to write and execute an application program using the API. Each of these steps are described in detail in the following subsections.

The remainder of this chapter assumes that the API is installed in a directory named **\ROCKET**, and that you will place your application source code in a new directory called **\ROCKET\CUSTOM**.

You may wish to create your own directory structure for source code. In that case, these instructions and the **make** files must be adjusted accordingly. A complete application demonstrating the use of the API is provided in the **\ROCKET\SAMPLE** directory.



Flowchart 2-1. How to Use the API

2.5. Application Example

The following application corresponds to the previous flowchart and the following subsections explain specific steps in detail.

```
#include <stdio.h>
#include <dos.h>
#include <process.h>
#include <stdlib.h>
#include <conio.h>
#include "api.h"                                          Step 3
                                                          (2.6)
main(void)
{
  unsigned int InitReturn;
  char Buf[80];
  int i,Dev,Cnt,Err;
```

```
/* Initialize API */
aaInstallCtrlCHandler();
 if((InitReturn = aaInit()) != NO_ERR)                    Step 4
{                                                         (2.7)
  printf("Init fail: %x\n",InitReturn);
  aaExit();
  exit(1);
}

/* Get serial device and display terminal emulator screen */
printf("Serial Device Number (0-15): ");
gets(Buf);                          /* get serial device */    Step 5
sscanf(Buf,"%d",&Dev);                                         (2.8-
system("cls");                       /* clear screen */         2.12)
printf("Serial Device Number %d\t\t\tHit F10 to Quit\n",Dev);

/* Open the device */
if((Err = aaOpen(Dev,
     COM_TX | COM_RX,
     COM_BAUD_9600,
     COM_PAR_NONE,
     COM_DATABIT_8,
     COM_STOPBIT_1,
     COM_FLOW_NONE,
     COM_DEN_PARITY | COM_DEN_RXOVR | COM_DEN_FRAME,
     COM_MDM_DTR)) != 0)
{
  printf("Failure - Could not open device number %d, Error
          %d\n",Dev,Err);
  aaExit();          /* required API call before exiting */    Step 6
  exit(1);                                                     (2.13)
}
/* Infinite loop to handle console I/O and serial I/O */
while(1)
{
  /* Attempt to read char from serial device and write to screen */
  if((Cnt = aaRead(Dev,80,(unsigned char *)Buf)) > 0)
  {
    for(i = 0;i < Cnt;i++)
      putch(Buf[i]);
  }

  /* Attempt to read char from keyboard and write to serial device */
  if((bdos(11,0,0) & 0xff) == 0xFF) /* if char waiting */
  {
    Buf[0] = bdos(8,0,0) & 0xff;   /* read keybd char */
    if((Buf[0] == '\0') && ((bdos(11,0,0)&0xff) == 0xff)) /* 2 char key */
    {
      Buf[1] = bdos(8,0,0) & 0xff; /* 2nd key */
        if(Buf[1] == 0x44)       /* F10 = quit */
          break;
    }
    aaWrite(Dev,1,(unsigned char *)Buf); /* write char to serial device */
  }
}
  aaClose(Dev,COM_MDM_RTS | COM_MDM_DTR); /* close device */
  aaExit();                    /* required API call before exiting */
  return(0);
}
```

2.6. Include Files (Step 3)

The **API.H** file must be included in the .C source code files.

2.7. Configuring RocketPort Controllers (Step 4)

Configuration of the RocketPort controllers and the API is done using **aaInit**(), as shown in the previous example. The **aaInit**() function must be called once before any other API function (except **aaInstallCtrlCHandler**()) can be called. It performs the configuration using the information in the configuration file given in the **ROCKETCFG** environment variable. See Subsection 2.3 for information about the format and placement of the configuration file.from the system configuration. See Subsection 2.3 for information about the system configuration.

Many applications also require that the DOS default **CTRL+C** key handler be replaced with a handler that calls **aaExit**() (see Subsection 2.10). This is done using **aaInstallCtrlCHandler**(). Once installed, this handler calls **aaExit**() if the user terminates the application by pressing the **CTRL+C** or **CTRL+BREAK** keys. If the application prevents program termination with these keys, the **aaIntallCtrlCHandler**() function does not need to be called.

2.8. Using API Calls (Step 5)

The following subsections provide details about Step 5 of the API sample. The topics include:

- Device numbers
- Configuration parameters for opening, closing, and reconfiguring serial devices
  - Open type
  - Baud
  - Parity
  - Data bits
  - Stop bits
  - Flow control
  - Detect enable
- Modem Control (output only)

2.8.1. Understanding Device Numbers

Each serial device is identified by a device number. Most API functions take the device number as a parameter. The number of ports that exist on each controller determines which device numbers map to which serial ports on which controllers.

The device numbers always count sequentially from 0, with the first port on the first controller in the configuration file assigned device number 0. Each subsequent controller in the configuration file begins counting where the previous controller left off. If there are more than one RocketPort controllers in the system, the controller in the lowest numbered slot is the first controller.

For example, if there are three controllers located in slots 2, 4, and 5, having 8, 16, and 8 ports respectively, the device numbers would map out as shown in Table 2-3.

Table 2-3. Mapping Device Numbers

Device Number	Controller Number	Slot	Port Number on the Controller
0 through 7	1	2	0 through 7
8 through 23	2	4	0 through 15
24 through 31	3	5	0 through 7

You can determine how many controllers are installed in your system, the first device number on each controller, and the number of devices on each controller with the **aaGetCtlStatus**() function.

2.8.2. Configuration Parameters for Serial Devices

Before the application can use a serial device, it must be opened with **aaOpen**(). To change the communication parameters while the device is open, use **aaReconfigure**(). Once the line is no longer in use it should be closed with **aaClose**().

There are a number of communication parameters used with one or more of the **aaOpen**(), **aaReconfigure**(), and **aaClose**() functions. Each of these parameters is described in the following subsections.

2.8.2.1. Open Type Parameter

The open type parameter is used in **aaOpen**() to identify whether the line is being opened for transmit, receive, or both. The flags used for open type are given in Table 2-4. This parameter is declared as follows:

unsigned int OpenType;

Table 2-4. Open Type Flags

Flag	Meaning When the Flag is Set
COM_TX	Open for transmit
COM_RX	Open for receive

2.8.2.2. Baud Parameter

The baud parameter is used with **aaOpen**() and **aaReconfigure**() to set the baud rate that the channel will operate at. You can assign only one of the flags shown in Table 2-5 to the baud parameter. The baud parameter is declared as follows:

unsigned char Baud;

Table 2-5. Baud Flags

Flag	Meaning When the Flag is Set
COM_BAUD_50	50 baud
COM_BAUD_75	75 baud
COM_BAUD_110	110 baud
COM_BAUD_134	134 baud
COM_BAUD_150	150 baud
COM_BAUD_200	200 baud
COM_BAUD_300	300 baud
COM_BAUD_600	600 baud
COM_BAUD_1200	1,200 baud
COM_BAUD_1800	1,800 baud
COM_BAUD_2400	2,400 baud
COM_BAUD_3600	3,600 baud
COM_BAUD_4800	4,800 baud

Table 2-5. Baud Flags (Continued)

Flag	Meaning When the Flag is Set
COM_BAUD_7200	7,200 baud
COM_BAUD_9600	9,600 baud
COM_BAUD_19200	19,200 baud
COM_BAUD_38400	38,400 baud
COM_BAUD_57600	57,600 baud
COM_BAUD_76800	76,800 baud
COM_BAUD_115200	115,200 baud
COM_BAUD_230400	230,400 baud

2.8.2.3. Parity Parameter

The parity parameter is used by **aaOpen**() and **aaReconfigure**() to set the type of parity checking done on receive and parity generation done on transmit. You can assign only one of the flags shown in Table 2-6 to the parity parameter. The parity parameter is declared as follows:

unsigned char Parity;

Table 2-6. Parity Flags

Flag	Meaning When Flag is Set
COM_PAR_NONE	No parity
COM_PAR_EVEN	Even parity
COM_PAR_ODD	Odd parity

2.8.2.4. Data Bits Parameter

The data bits parameter is used by **aaOpen**() and **aaReconfigure**() to set the number of data bits in each transmitted and received character. You can assign only one of the flags shown in Table 2-7 to the data bits parameter. The data bits parameter is declared as follows:

unsigned DataBits

Table 2-7. Data Bits Flags

Flag	Meaning When Flag is Set
COM_DATABIT_7	7 data bits
COM_DATABIT_8	8 data bits

2.8.2.5. Stop Bits Parameter

The stop bits parameter is used by **aaOpen**() and **aaReconfigure**() to set the number of stop bits used in the framing of each transmitted and received character. You can assign only one of the flags shown in Table 2-8 to the stop bits parameter. The stop bits parameter is declared as follows:

unsigned char StopBits;

Table 2-8. Stop Bits Flags

Flag	Meaning When Flag Set
COM_STOPBIT_1	1 stop bit
COM_STOPBIT_2	2 stop bits

2.8.2.6. Flow Control Parameter

The flow control parameter is used by **aaOpen**() and **aaReconfigure**() to set the flow control method. You can assign either COM_FLOW_NONE or any combination of the remaining flags shown in Table 2-9 to the flow control parameter.

The flow control parameter is declared as follows:

unsigned int FlowCtl;

Table 2-9. Flow Control Flags

Flag	Meaning When Flag Set
COM_FLOW_NONE	No flow control
COM_FLOW_IS	Enable input software flow control
COM_FLOW_IH	Enable input hardware flow control using RTS

Table 2-9. Flow Control Flags (Continued)

Flag	Meaning When Flag Set
COM_FLOW_OS	Enable output software flow control
COM_FLOW_OH	Enable output hardware flow control using CTS
COM_FLOW_OXANY	Enable restart output on any Rx character

2.8.2.7. Detection Enable Parameter

The detection enable parameter is used by **aaOpen**() and **aaReconfigure**() to set which events are detected by the API.

If a detection enable flag is set, an event function within the application is dispatched when that event is detected. This assumes that the application has installed the event function. See Subsection 2.12 for information about event functions.

You can assign any combination of the flags shown in Table 2-10 to the detection enable parameter. The detection enable parameter is declared as follows:

unsigned int DetectEn;

Table 2-10. Detection Enable Flags

Flag	Meaning When Flag Set
COM_DEN_NONE	No event detection enabled
COM_DEN_MDM	Enable modem control change detection
COM_DEN_RDA	Enable receive data available detection

2.8.2.8. Modem Control Parameter

The modem control parameter is used by **aaOpen**() to determine the initial state of the modem control outputs. If a flag is set, the modem control line is turned ON; otherwise it is OFF.

It is also used by **aaClose**() to determine which modem control outputs must be cleared. If a flag is set, that modem control line is turned OFF; otherwise it is not changed.

The modem control output flags are given in Table 2-11. The modem control parameter is declared as follows:

unsigned ModemCtl;

Table 2-11. Modem Control Output Flags

Flag	Modem Control Line
COM_MDM_RTS	Request to send
COM_MDM_DTR	Data terminal ready

2.9. Writing Serial Data

After a device is open, serial data can be written to it using **aaWrite**(). The number of data bytes from previous **aaWrite**() calls that are still awaiting transmission can be obtained with **aaGetTxCount**().

2.10. Exiting the Application

You must call **aaExit**() before exiting an application. This does final cleanup, including removing the interrupt service routine (ISR) used by the API.

2.11. Reading Serial Data

After a device is open serial data can be read from it using **aaRead**(). The number of receive data bytes that are buffered by the device can be obtained with **aaGetRxCount**().

Using **aaRead**() by itself does not return any receive error information. If error information is needed, you can determine if there are any errors in the device's receive buffer by calling **aaGetRxStatus**(). If errors exist, you can obtain the error status of each receive data byte by reading the data with **aaReadWithStatus**().

2.12. Installing and Detecting Events

When the controller needs to notify the system that something important has occurred, it generates an interrupt and causes an event function to execute on the system.

Event functions tell you what has happened and provide the appropriate information for that event, which you can then process as needed.

The following receive events can occur on the system:

- Modem change event, one of the modem lines has changed for a serial device.
- Receive data event, data has been received on a serial device.
- Periodic event, occurs 274 times per second.

You need a way to tie your application to these events. This is accomplished by calling the **aaInstall*xxx*Event** functions. By using an **aaInstall*xxx*Event** function,

you can give the system software the name of an application program function that executes when a particular event occurs. The following **aaInstall*xxx*Event** functions are available:

- **aaInstallMdmChangeEvent**
- **aaInstallPeriodicEvent**
- **aaInstallRxEvent**

Example 2-1 provides event function examples and shows how to install event functions. Notice that installing event functions is done shortly after the controller is initialized.

Even after an event function is installed, it will not be dispatched unless that event has been enabled. Modem change and receive data events are enabled or disabled using the **DetectEn** parameter in the **aaOpen**() function. Periodic events are enabled or disabled using the **aaEnPeriodicEvent**() function.

Example 2-1. Sample Event Function

```
#define NUMDEV 32              /* max number devices this app supports */

int FirstDev, MaxDev;          /* first and maximum device numbers */
unsigned char CD_Change[NUMDEV]; /* indicates changes to CD modem input */
unsigned char ModemState[NUMDEV]; /* state of modem inputs for each device */

main()                         /* application main() fragment */
{   int NumDev;

  /* Initialize controller */
  aaInstallCtrlCHandler();
  if(aaInit() != NO_ERR)
  {
    printf("Initialization Failure\n");
    aaExit();
    exit(1);
  }

  /* Get device number range for 1 controller */
  if(aaGetCtlStatus(0,&FirstDev,&NumDev) != NO_ERR)
  {
    printf("Controller Status Failure\n");
    aaExit();
    exit(1);
  } }

MaxDev = FirstDev + NumDev - 1;

  /* Set up application event functions */
  aaInstallRxEvent(EvRxData);
  aaInstallMdmChangeEvent(aaModemChg);
  aaInstallPeriodicEvent(EvPeriodic);
  aaEnPeriodicEvent(TRUE);
  .
  .
  .
}

#pragma check_stack(off)       /* Microsoft C only */

void ExRxData(int Dev)         /* receive event function */
{
  int Count;

  Count = aaGetRxCount(Dev);   /* get number bytes available */
  if(Count > BUF_SIZE)
    Count = BUF_SIZE);
  GetRxData(Dev,Count);        /* application function to read the data */
}

void EvMdmChg(int Dev,unsigned char MdmChange,unsigned char MdmState)
{
  if(MdmChange & COM_MDM_CD)   /* CD changed */
  {
    CD_Change[Dev]++;          /* indicate change occurred */
  }
  ModemState[Dev] = MdmState;  /* save current state of modem inputs */
}

void EvPeriodic(void)          /* periodic event function */
{
  int Dev;

  for(Dev = FirstDev;Dev <= MaxDev;Dev++) /* check all devs for Tx data*/
  {
    SendTxData(Dev);           /* application function to transmit data */
  }
}

#pragma check_stack(on)        /* Microsoft C only */
```

Each of the previously described event functions require different parameters. For example, the receive data event function only passes a device number to the application, whereas the modem change event function passes a device, a modem state, and a modem change parameter to the application's event function.

These parameters are described in Appendix A under the function names prefixed with **Ev**.

The periodic event is different from the other events in that it occurs on regular intervals regardless of what is occurring on the controller. One use for the periodic event function is to allow the application to write data to devices in the background. See Subsection 2.13 for more information.

Warnings: *The event functions you write for your applications are actually executing during a system interrupt service routine (ISR). It is very important that you keep these event functions as short as possible. Also, there are many standard C library functions that do not work within an ISR, such as* **printf***(). Using these functions can cause unpredictable results and can even hang your system.*

If using the Microsoft C compiler, stack checking must be disabled

during event functions and any functions called by event functions. Stack checking can be turned off and on with:

#pragma check_stack(off)

#pragma check_stack(on)

2.13. Double Buffering Transmit and Receive Data

Each serial device on the RocketPort controller internally provides 250 bytes of buffering for transmit data and 1K bytes of buffering for receive data. In some applications this may not be sufficient.

For example, an application program may need to write large blocks of data at infrequent intervals. If the application calls **aaWrite**() directly, only 250 bytes are taken, and the device's internal transmit buffer may empty before the next **aaWrite**() call occurs, leaving a period of time where no data is being transmitted.

In cases like the one described above, additional buffering is needed. To accomplish this, the data can be double buffered using event functions (see Subsection 2.12). This allows the application to move serial data to and from the buffers rather than directly accessing the device using **aaWrite**() and **aaRead**(). The event function handles moving data between the device and the buffer. Double buffering as described in this subsection adds additional overhead, so it should only be done when an application requires it.

A sample program (**\ROCKET\SAMPLE\DBUF.C**) shows an example of double buffering. Also included is a Borland C++ make file called **MAKEDBUF.BC**. The source code is reproduced in this guide in Appendix B .

For double buffering of transmit data, use the periodic event function. This function polls each device's buffer for data, and if data is available writes it to the device using **aaWrite**(). The **EvPeriodic**() function in **DBUF.C** shows how to do this.

The **EnqTxData**() is used in **DBUF.C** to write data into the transmit buffer. The application calls **EnqTxData**() instead of writing directly to the device with **aaWrite**(). Notice that **EnqTxData**() disables interrupts while manipulating the write buffer pointers. This is necessary because **EvPeriodic**() is part of an interrupt service routine (ISR) and you do not want it to suddenly interrupt and change these pointers until you are completely done updating them.

For double buffering of receive data, the receive event function should be used. This event function is not called unless the device has receive data available. The event function then reads the data with **aaRead**() or **aaReadWithStatus**() and places it in the receive buffer. A simple example using only **aaRead**() is shown in the **EvRxData**() function in **DBUF.C**.

The **DeqRxData**() function is used in **DBUF.C** to read data from the receive buffer. The application calls **DeqRxData**() instead of reading directly from the device with **aaRead**(). Notice that **DeqRxData**() disables interrupts while manipulating the read buffer pointers. **EvRxData**() is part of an ISR, so this is necessary for the same reason interrupts were disabled in **EnqTxData**().

2.14. Building Applications (Step 6)

The application is built by executing the compiler's **make** utility and a **make** file. The **make** file contains the rules that the **make** utility uses to build the application. If the application is contained entirely in a single source file called **TERM.C**, then the **make** file copied in from the **\ROCKET\SAMPLE** directory can be used as is. Otherwise, you must modify the **make** file to build using your application source file names.

Section 3. Troubleshooting

3.1. Resolving Installation Problems

If installation fails or you are trying to resolve a problem, you should try the following before calling the Comtrol technical support line:

- Check the signals between your peripherals and the interface box to verify that they match (if applicable). See the appropriate *Hardware Reference Card* for information.
- Check to make sure the serial and interface cables are connected properly.
- Check to see if the DIP switch is set to the desired address by checking the **/ROCKET/INSTALL.LOG** file with an editor against the settings on each controller.
- Reseat the controller in the slot.
- Make sure that the expansion slot screw was replaced after inserting the controller.
- Reinstall the API, selecting a different I/O address range for the controller. For possible I/O address conflicts, see Tables 3-1 and 3-2.

Table 3-1 defines the 64-byte I/O address blocks from 0 through 3FFh and their known uses. Table 3-2 defines the 64-byte I/O address blocks from 400 through FFFh and their known uses.

Table 3-1. System I/O Addresses – Up to 3FF

Address Block	Addresses Used	Description
000 – 03F		Reserved for Motherboard
040 – 07F		Reserved for Motherboard
080 – 0BF		Reserved for Motherboard
0C0 – 0FF		Reserved for Motherboard
100 – 13F		
140 – 17F		
180 – 1BF		
1C0 – 1FF	1F0 – 1F8	Fixed Disk
200 – 23F		
240 – 27F	278 – 27F	LPT2, IDE controllers, multifunction boards (game ports)
280 – 2BF		
2C0 – 2FF	2E8 – 2EF 2F8 – 2FF	COM4 COM2
300 – 33F		
340 – 37F	378 – 37F	LPT1
380 – 3BF	3B0 – 3BF	Monochrome Display and LPT3
3C0 – 3FF	3D0 – 3DF 3E8 – 3EF 3F0 – 3F7 3F8 – 3FF	Graphics Monitor Adapter COM3 Floppy Disk Controller COM1

Table 3-2. System I/O Address Aliases – Above 3FF

Address Block	1st Alias	2nd Alias	3rd Alias
000 – 03F	400 – 43F	800 – 83F	C00 – C3F
040 – 07F	440 – 47F	840 – 87F	C40 – C7F
080 – 0BF	480 – 4BF	880 – 8BF	C80 – CBF
0C0 – 0FF	4C0 – 4FF	8C0 – 8FF	CC0 – CFF
100 – 13F	500 – 53F	900 – 93F	D00 – D3F
140 – 17F	540 – 57F	940 – 97F	D40 – D7F
180 – 1BF	580 – 5BF	980 – 9BF	D80 – DBF

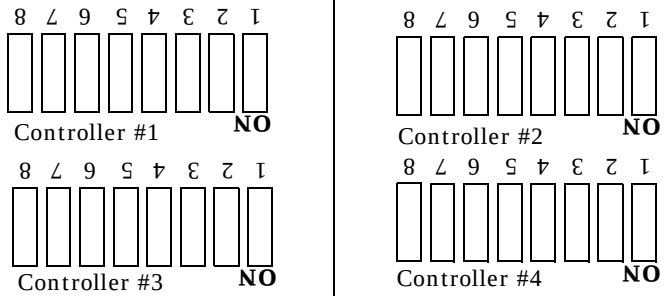
Table 3-2. System I/O Address Aliases – Above 3FF (Continued)

Address Block	1st Alias	2nd Alias	3rd Alias
1C0 – 1FF	5C0 – 5FF	9C0 – 9FF	DC0 – DFF
200 – 23F	600 – 63F	A00 – A3F	E00 – E3F
240 – 27F	640 – 67F	A40 – A7F	E40 – E7F
280 – 2BF	680 – 6BF	A80 – ABF	E80 – EBF
2C0 – 2FF	6C0 – 6FF	AC0 – AFF	EC0 – EFF
300 – 33F	700 – 73F	B00 – B3F	F00 – F3F
340 – 37F	740 – 77F	B40 – B7F	F40 – F7F
380 – 3BF	780 –7BF	B80 – ABF	F80 – FBF
3C0 – 3FF	7C0 – 7FF	BC0 – BFF	FC0 – FFF

3.2. Placing a Support Call

Before you place a technical support call to Comtrol, please make sure that you have the following information.

Table 3-3. Support Call Information

Item	Your System Information
Controller type	4-port, 8-port, 16-port, or 32-port model
Interface type	DB25, RJ45, or RJ11
Mark your I/O address selections	
Operating system type and release	
Device driver release number (to verify, view the **VERSION.DAT** file)	
PC make, model, and speed	
List of other devices in the PC and their addresses	

Contact Comtrol using one of the following methods.

Corporate Headquarters:

- email: support@comtrol.com
- FAX: (612) 631-8117
- Phone: (612) 631-7654
- BBS: (612) 631-8310 (for device driver updates)
- FTP Site: ftp://ftp.comtrol.com

Note: The BBS supports modem speeds up to 28.8 Kbps with 8 bits, and no parity.

Comtrol Europe:

- email: support@comtrol.co.uk
- FAX: +44 (0) 1 869-323-211
- Phone: +44 (0) 1 869-323-220
- BBS: +44 (0) 1 869-243-687

3.3. Retrieving Future Software Updates

Comtrol supports a BBS that provides software updates for our customers.

***Note**: The BBS supports modem speeds up to 14.4Kbps with 8 bits and no parity.*

BBS: (612) 631-8310

Appendix A. API Functions

This appendix contains reference pages for the RocketPort API. Table A-4 lists all of the API functions.

Table A-4. API Function Reference

Function	Description
aaChangeModemState	Changes the state of modem output lines.
aaClose	Closes a device.
aaEnPeriodicEvent	Enables or disables dispatching of the periodic event function.
aaExit	Performs cleanup when exiting from an application program.
aaFlush	Flushes the transmit or receive buffer, or both for a device.
aaGetCtlStatus	Gets controller status.
aaGetModemStatus	Gets a device's modem status.
aaGetRxCount	Gets the count of data bytes available in the receive buffer.
aaGetRxStatus	Gets the status of the device's receive buffer.
aaGetTxCount	Gets the count of data bytes in the transmit buffer waiting to be transmitted.
aaInit	Executes controller and API initialization.
aaInstallCtrlCHandler	Installs a handler for the **CTRL+C** key interrupt.
aaInstallMdmChgEvent	Installs an application level event function to handle modem change events.
aaInstallPeriodicEvent	Installs a periodic application level event function.
aaInstallRxEvent	Installs an application level event function to handle receive data available events.
aaOpen	Open a device for reading or writing, or both.
aaRead	Reads serial data from a device.
aaReadWithStatus	Reads serial data and status from a device.
aaReconfigure	Reconfigures a device's communications parameters.
aaSendBreak	Sends a break signal.
aaSetCloseDelay	Sets the maximum time **aaClose**() waits for a device's transmit buffer to drain before flushing the transmit buffer and completing the close.
aaWrite	Writes serial data out to a device.
EvModemChange*	Modem control input change event function.
EvPeriodic*	Periodic event function.
EvRxData*	Receive data available event function.

* These are not part of the API, but are part of the application.

Function **aaChangeModemState**

Purpose Changes the state of modem output lines.

Call **aaChangeModemState(*Dev,RTSState,DTRState*)**

int *Dev*	Device number
int *RTSState*	State of RTS line: ON, OFF, or NOCHANGE
int *DTRState*	State of DTR line: ON, OFF, or NOCHANGE

Return **int**: **NO_ERR** if successful
ERR_DEV if device out of range

Function **aaClose**

Purpose Closes a device.

Call **aaClose(*Dev,ModemCtl*)**

int *Dev*	Device number
unsigned char *ModemCtl*	Modem control lines to turn OFF, can be COM_MDM_RTS or COM_MDM_DTR. If the flag is not set the state of the modem line is not changed.

Return **int:** **NO_ERR** if successful
ERR_DEV if device number out of range
ERR_MDMCTL if invalid modem control flag
ERR_NOTOPEN if device not open

Comments This function waits for the device's transmit buffer to drain before completing the close. The maximum wait time defaults to **CLOSE_TBEDLY**, but can be changed with the **aaSetCloseDelay**() function.

Warning This function disables and enables interrupts.

Function **aaEnPeriodicEvent**

Purpose Enables or disables dispatching of the periodic event function.

Call **aaEnPeriodicEvent(*State*)**

int *State*	TRUE to enable dispatching of the periodic event function, FALSE to disable dispatching.

Return **void**

Comments The periodic event function is called 274 times a second. Once installed, the periodic event function is not dispatched until it is enabled with the **aaEnPeriodicEvent**() function. The **aaEnPeriodicEvent**() function can also be used to disable dispatching of the periodic event function.

Warning The event function must be installed with **aaInstallPeriodicEvent**() before enabling dispatching.

aaExit

Purpose Performs cleanup when exiting from an application program.

Call **aaExit()**

Return **void**

Comments This function does cleanup tasks required when exiting from an application, such as halting controller interrupts and restoring the IRQ vector used by the controller.

Warning Once **aaInit**() has been called, **aaExit**() must be called before exiting the application program.

If the application program can be exited using the **CTRL+C** or **CTRL+BREAK** keys, then the default DOS **CTRL+C** handler must be replaced with a handler that calls **aaExit**(). You can use the **aaInstallCtrlCHandler**() function for this purpose.

Function **aaFlush**

Purpose Flushes the transmit or receive buffer, or both for a device.

Call **aaFlush(*Dev*,*FlushFlags*)**

int *Dev*	Device number
unsigned char *FlushFlags*	COM_TX or COM_RX, or both

Return **int:**

NO_ERR	if successful
ERR_DEV	if device is out of range
ERR_OPENTYPE	if ***FlushFlags*** is out of range

Function **aaGetCtlStatus**

Purpose Gets controller status, including the first device number on the controller and the number of devices on the controller.

Call **aaGetCtlStatus(*CtlNum*,*FirstDevP*,*NumDevP*)**

int *CtlNum*	Controller number to get status on.
int **FirstDevP*	Pointer to variable where first device number on this controller will be returned.
int **NumDevP*	Pointer to variable where number of devices on this controller will be returned

Return **int:**

NO_ERR	Controller is installed
ERR_NOCTL	Controller is not installed

Comments The ***CtlNum*** parameter identifies which RocketPort controller to get the status of. Controllers are numbered sequentially beginning with 0. Controller 0 will be the first controller whose address appears in the configuration file given by the **ROCKETCFG** environment variable. The contents of the ***FirstDevP*** and ***NumDevP*** parameters are modified only if **NO_ERR** is returned.

Function **aaGetModemStatus**

Purpose Gets a device's modem status.

Call **aaGetModemStatus(*Dev*)**
int *Dev*: Device number

Return **unsigned char** State of the modem control inputs using the COM_MDM_CTS, COM_MDM_DSR, and COM_MDM_CD flags. If a flag is set that modem line is ON, if a flag is not set that modem line is OFF.

Function **aaGetRxCount**

Purpose Gets the count of data bytes available in the receive buffer.

Call **aaGetRxCount(*Dev*)**
int *Dev* Device number

Return **int:** Receive byte count

Function **aaGetRxStatus**

Purpose Gets the status of the device's receive buffer.

Call **aaGetRxStatus(*Dev*)**
int *Dev* Device number

Return **int:** **NO_ERR** if there are no errors in the device's receive buffer

ERR_RX if there are errors in the device's receive buffer

ERR_DEV if device number out of range

ERR_NOTOPEN if device not open for receive

Comments If there are errors in the device's receive buffer, the exact error and the errored data byte can be determined using the **aaReadWithStatus**() function.

Function **aaGetTxCount**

Purpose Gets the count of data bytes in the transmit buffer waiting to be transmitted.

Call **aaGetTxCount(Dev)**
int Dev Device number

Return **int:** Transmit byte count

Function **aaInit**

Purpose Executes controller and API initialization.

Call **aaInit()**

Return **unsigned int**

NO_ERR	if no initialization errors
ERR_ALLOCDEV	if it can not allocate device structure
ERR_CTLINIT	if controller initialization error
ERR_CHANINIT	if channel initialization error
ERR_DEVSIZE	if invalid number of devices found
ERR_CTLSIZE	if invalid number of controllers found

Comments This function must be called once before calling any other API function except **aaInstallCtrlCHandler**(). The controller initialization parameters is obtained from the configuration file given by environment variable **ROCKETCFG**.

Warning Once **aaInit**() has been called, **aaExit**() must be called before exiting the application program.

If the application program can be exited using the **CTRL+C** or **CTRL+BREAK** keys, then the default DOS **CTRL+C** handler must be replaced with a handler that calls **aaExit**(). You can use the **aaInstallCtrlCHandler**() function can be used for this purpose.

Function **aaInstallCtrlCHandler**

Purpose Installs a handler for the **CTRL+C** key interrupt.

Call **aaInstallCtrlCHandler()**

Return **void**

Comments This function replaces the existing **CTRL+C** (interrupt 23H) handler with a handler that performs the following actions:

1. Calls **aaExit**().
2. Sets the carry flag to signal DOS to terminate the application.
3. Executes a **far** return.

DOS restores the original **CTRL+C** handler when terminating the application.

aaInstallCtrlCHandler() is the only API function that can be called before calling **aaInit**(). If you plan on using the **aaInstallCtrlCHandler**() function, we recommend calling it either immediately before or immediately after the call to the **aaInit**() function.

If you want different **CTRL+C** processing, you must write and install your own custom **CTRL+C** handler. Refer to the *Microsoft MS-DOS Programmer's Reference* for more information. To aid in writing your own handler, the source code for **aaInstallCtrlCHandler**() and the handler it installs are given below:

```
void aaInstallCtrlCHandler(void)
{
dos_setvect(0x23,(void (interrupt far
*)())aaCtrlCIntHandler);
}

void far aaCtrlCIntHandler(void)
 {
   aaExit();
   asm stc;
}
```

Warning If the application program can be exited using the **CTRL+C** or **CTRL+BREAK** keys, then the default DOS **CTRL+C** handler must be replaced with a handler that calls **aaExit**(). You can use the **aaInstallCtrlCHandler**() function for this purpose.

Function **aaInstallMdmChgEvent**

Purpose Installs an application level event function to handle modem change events.

Call **aaInstallMdmChgEvent(*EvFuncP*)**
void (evFuncP*)(*Dev*,unsigned char *MdmChange*,**
unsigned char *MdmState*) Ptr to the event function

Return **void**

Comments See the **EvModemChange**() function for a description of the event function.

Warning The function installed here is called during an interrupt service routine (ISR). Keep your code short and remember that many standard C library calls do not work in ISRs, such as printf().

If using the Microsoft C compiler, stack checking must be disabled during the event function and any functions called by the event function.

Stack checking can be turned off and on with:

#pragma check_stack(off)

#pragma check_stack(on)

Function **aaInstallPeriodicEvent**

Purpose Installs a periodic application level event function.

Call **aaInstallPeriodicEvent(*EvFuncP*)**
void (EvFuncP*)(void)** Ptr to the event function.

Return **void**

Comments The periodic event function is called 274 times a second. Once installed, the periodic event function is not dispatched until it is enabled with **aaEnPeriodicEvent**(). The **aaEnPeriodicEvent**() function can also be used to disable dispatching of the periodic event function.

Comments See the **EvPeriodic**() function for a description of the event function.

Warning The function installed here will be called during an interrupt service routine (ISR). Keep your code short and remember that many standard C library calls do not work in ISRs, such as **printf**().

If using the Microsoft C compiler, stack checking must be disabled during the event function and any functions called by the event function.

Stack checking can be turned off and on with:

#pragma check_stack(off)

#pragma check_stack(on)

Function **aaInstallRxEvent**

Purpose Installs an application level event function to handle receive data available events.

Call **aaInstallRxEvent(*EvFuncP*)**
void (EvFuncP*)(int *Dev*)**; Ptr to the event function.

Comments See the **EvRxData**() function for a description of the event function.

Return **void**

Warning The function installed here is called during an interrupt service routine (ISR). Keep your code short and remember that many standard C library calls do not work in ISRs, such as **printf**().

If using the Microsoft C compiler, stack checking must be disabled during the event function and any functions called by the event function.

Stack checking can be turned off and on with:

#pragma check_stack(off)

#pragma check_stack(on)

Function **aaOpen**

Purpose Open a device for reading or writing, or both.

Call **aaOpen(*Dev,OpenType,Baud,Parity,DataBits,StopBits,FlowCtl, DetectEn,ModemCtl*)**

int *Dev*	Device Number
unsigned int *OpenType*	COM_TX for transmit or COM_RX for receive, or both.
unsigned char *Baud*	One of the COM_BAUD_XX flags defined in API.H.
unsigned char *Parity*	One of: COM_PAR_NONE, COM_PAR_EVEN, COM_PAR_ODD.
unsigned char *DataBits*	One of COM_DATABIT_7, COM_DATABIT_8.
unsigned char *StopBits*	One of COM_STOPBIT_1, COM_STOPBIT_2.
unsigned int *FlowCtl*	Flow control flag, can be COM_FLOW_NONE or any combination of COM_FLOW_NONE, COM_FLOW_IS, COM_FLOW_OS, COM_FLOW_IH, COM_FLOW_OH, COM_FLOW_OXANY.
unsigned int *DetectEn*	Detection enable flags, can be any combination of the following:

COM_DEN_NONE	No error detection enabled
COM_DEN_RDA	Enable Rx data available detection
CON_DEN_MDM	Enable modem input (DSR, CD, or CTS) change detection

unsigned char *ModemCtl*	Modem control lines to turn ON, can be COM_MDM_RTS or

COM_MDM_DTR, or both. If the flag is not set the line is OFF. If hardware flow control is in use for a modem line, it's flag has no effect.

Return

int:	**NO_ERR**	if successful
	ERR_DEV	if device number out of range
	ERR_OPENTYPE	if invalid open type flag
	ERR_BAUDRATE	if invalid baud rate flag
	ERR_PAR	if invalid parity bits flag
	ERR_DATAB	if invalid data bits flag
	ERR_STOPB	if invalid stop bits flag
	ERR_FLOW	if invalid flow control bits flag
	ERR_DETECT	if invalid detect enable flag
	ERR_MDMCTL	if invalid modem control flag
	ERR_ALREADYOPEN	if device already open error flag

Comments

If this device has been opened previously and is still open, this function fails and returns an **ERR_ALREADYOPEN** error.

Warning

This function disables and enables interrupts.

Function

aaRead

Purpose

Reads serial data from a device.

Call

aaRead(*Dev,Cnt,Buf*)

int *Dev*	Device number
int *Cnt*	Maximum number of bytes that can be read
unsigned char **Buf*	Buffer to store the data in

Return

int:	**Number of bytes read**	if successful
	0	if no data available to be read
	ERR_DEV	if device number out of range
	ERR_NOTOPEN	if device not open for receive

Comments

This function reads data from the device's receive buffer without checking for receive errors. If receive error information is needed, use the **aaGetRxStatus**() and **aaReadWithStatus**() functions.

Warning

The ***Cnt*** parameter should not be greater than the size of the ***Buf*** receive buffer.

Function **aaReadWithStatus**

Purpose Reads serial data and status from a device.

Call **aaReadWithStatus(*Dev,Cnt,Buf)***

int *Dev*	Device number
int *Cnt*	Max number of bytes that can be read
unsigned int **Buf*	Buffer to store the data and status. The low byte of each array element in *Buf* contains the data byte, and the high byte contains the status for that data byte. The status may be 0 indicating no error, or any combination of the following flags:

ERR_PARITY	parity error
ERR_OVRRUN	receiver over run error
ERR_FRAME	framing error
ERR_BREAK	break

Return **int:** **Number of bytes read** if successful
0 if no data available to be read
ERR_DEV if device number out of range
ERR_NOTOPEN if device not open for receive

Warning The ***Cnt*** parameter should not be greater than the number of array elements in the ***Buf*** receive buffer.

Function **aaReconfigure**

Purpose Reconfigures a device's communications parameters.

Call **aaReconfigure(*Dev,Baud,Parity,DataBits,StopBits,FlowCtl,DetectEn*);**

int *dev*;	Device Number
unsigned char *Baud*	One of the baud rate flags defined in API.H.
unsigned char *Parity*	One of: COM_PAR_NONE, COM_PAR_EVEN, COM_PAR_ODD.
unsigned char *DataBits*	One of COM_DATABIT_7, COM_DATABIT_8.
unsigned char *StopBits*	One of COM_STOPBIT_1, COM_STOPBIT_2.
unsigned int *FlowCtl*	Flow control flag, can be COM_FLOW_NONE or any combination of: COM_FLOW_IS, COM_FLOW_OS, COM_FLOW_IH, COM_FLOW_OH, COM_FLOW_OXANY.
unsigned int *DetectEn*	Detection enable flags, can be any combination of the following:

COM_DEN_NONE	No error detection enabled
COM_DEN_RDA	Enable Rx data available detection
COM_DEN_MDM	Enable modem input (DSR,CD, or CTS) change detection

Return **int:** **NO_ERR** if successful
ERR_DEV if device number out of range
ERR_BAUDRATE if invalid baud rate flag

ERR_PAR	if invalid parity bits flag
ERR_DATAB	if invalid data bits flag
ERR_STOPB	if invalid stop bits flag
ERR_FLOW	if invalid flow control bits flag
ERR_DETECT	if invalid detect enable flag
ERR_NOTOPEN	if device not open error flag

Warning This function disables and enables interrupts.

Function **aaSendBreak**

Purpose Sends a break signal.

Call **aaSendBreak(Dev,Time)**

int Dev	Device number
int Time	Time in milliseconds to send the break

Return **int:**

NO_ERR	if successful
ERR_DEV	if device is out of range.

Function **aaSetCloseDelay**

Purpose Sets the maximum time **aaClose()** waits for a device's transmit buffer to drain before flushing the transmit buffer and completing the close.

Call **aaSetCloseDelay(*Dev,MaxDelay*)**

int *Dev*	Device number
int *MaxDelay*	Maximum time aaClose will wait for a device's transmit buffer to drain in seconds. For no delay use 0. Maximum value is 32,767 seconds.

Return **int:**

NO_ERR	if successful.
ERR_DEV	if device number out of range.

Comments The device does not need to be open to execute this function.

Function **aaWrite**

Purpose Writes serial data out a device.

Call **aaWrite(*Dev,Cnt,Buf*)**

int *Dev*:	Device number
int *Cnt*:	Number of bytes to be written
unsigned char **Buf*:	Buffer of data to write

Return **int:**

Number of bytes written	if successful
0	if no data bytes written
ERR_DEV	if dev number out of range
ERR_NOTOPEN	if dev not open for transmit

Warning The ***Cnt*** parameter should not be greater than the size of the ***Buf*** transmit buffer.

Function **EvModemChange**

Purpose Application modem input change event function

Call **EvModemChange(*Dev*,unsigned char *MdmChange*,unsigned char *MdmState*)**

int *Dev*	Device number
unsigned char *MdmChange*	Modem input lines which changed. Can be any combination of the flags: COM_MDM_DSR, COM_MDM_CTS, or COM_MDM_CD. If a flag is set that modem line is changed, if a flag is not set that modem line did not change.
unsigned char *MdmState*	Current state of the modem inputs. Can be any combination of the COM_MDM_CTS, COM_MDM_DSR, and COM_MDM_CD flags. If a flag is set that modem line is ON, if a flag is not set that modem line is OFF.

Return **void**

Comments This function is not part of the API, it must be written by the developer as part of the application program.

The function name **EvModemChange** is an example name only, this event function can be given any name desired.

This function is not called directly by the application. Instead, it is dispatched by the API's internal ISR (interrupt service routine) when it detects that receive data is available. Before this function will be dispatched it must be installed with **aaInstallMdmChgEvent**(), and modem input change detection must be enabled. Detection is enabled using the **DetectEn** parameter of **aaOpen**() or **aaReconfigure**().

Warning The function installed here is called during an interrupt service routine (ISR). Keep your code short and remember that many standard C library calls do not work in ISRs, such as **printf**().

If using the Microsoft C compiler, stack checking must be disabled during the event function and any functions called by the event function.

Stack checking can be turned off and on with:

#pragma check_stack(off)

#pragma check_stack(on)

Function **EvPeriodic**

Purpose Application periodic event function

Call **EvPeriodic()**

Return **void**

Warning This function is not part of the API, it must be written by the developer as part of the application program.

The function name **EvPeriodic** is an example name only, this event function can be given any name desired.

This function is not called directly by the application. Instead it is dispatched by the API's internal ISR (interrupt service routine) when it detects that receive data is available. Before this function will be dispatched it must be installed with **aaInstallPeriodicEvent**(), and periodic events must be enabled with **aaEnPeriodicEvent**().

Once installed and enabled, the periodic event function is called 274 times a second regardless of the state of controller.

Warning The function installed here is called during an interrupt service routine (ISR). Keep your code short and remember that many standard C library calls do not work in ISRs, such as **printf**().

If using the Microsoft C compiler, stack checking must be disabled during the event function and any functions called by the event function.

Stack checking can be turned off and on with:

#pragma check_stack(off)

#pragma check_stack(on)

Function **EvRxData**

Purpose Application receive data available event function

Call **EvRxData(Dev)**
int *Dev*: Device number

Return **void**

Warning This function is not part of the API, it must be written by the developer as part of the application program.

The function name **EvRxData** is an example name only, this event function can be given any name desired.

This function is not called directly by the application. Instead, it is dispatched by the API's internal ISR (interrupt service routine) when it detects that receive data is available. Before this function will be dispatched it must be installed with **aaInstallRxEvent**(), and receive data available detection must be enabled. Detection is enabled using the **DetectEn** parameter of **aaOpen**() or **aaReconfigure**().

Warning The function installed here is called during an interrupt service routine (ISR). Keep your code short and remember that many standard C library calls do not work in ISRs, such as **printf**().

If using the Microsoft C compiler, stack checking must be disabled during the event function and any functions called by the event function.

Stack checking can be turned off and on with:

#pragma check_stack(off)

#pragma check_stack(on)

Appendix B. Double Buffering Example

This appendix contains a copy of the **\ROCKET\SAMPLE\DBUF.C** file for your convenience.

File: DBUF.C
Project: RocketPort DOS API
Purpose: Double buffering sample program
Comments: This program shows how to use the periodic event function to double buffer Tx data, and how to use the receive event function to double buffer Rx data. The double buffering is done in a pair of queues for each device, these are defined in the Q_T structure.
Operation: Install a loopback plug on port 0. At the DOS command line type "DBUF." Each time transmit data is enqueued to the device 0, the Tx buffer the count is displayed. Each time data is dequeued from the device 0; the Rx buffer, the count, and Rx string are displayed.

```
**********************************************************/
#include <stdio.h>
#include <process.h>
#include <stdlib.h>
#include <mem.h>
#include <string.h>
#include <conio.h>
#include <dos.h>
#include "api.h"

#define NUMDEV 8      /* num devices this app supports */
#define TXBUF_SIZE 1024      /* transmit buffer size */
#define RXBUF_SIZE 1024      /* receive buffer size */

void EvRxData(int);     /* function prototypes */
void EvPeriodic(void);
int EnqTxData(int,unsigned char *,int);
int DeqRxData(int,unsigned char *,int);

typedef struct‘          /* transmit and receive queues */
{
  int OpenTx;            /* TRUE if device open for Tx */
  int TxIn;              /* index to add Tx data at */
  int TxOut;             /* index to remove Tx data at */
  unsigned char TxBuf[TXBUF_SIZE]; /* buffer for Tx data */
  int RxIn;              /* index to add Rx data at */
  int RxOut;             /* index to remove Rx data at */
  unsigned char RxBuf[RXBUF_SIZE]; /* buffer to Rx data */
} Q_T;
Q_T q[NUMDEV];/* Tx and Rx queues for each dev */

/**********************************************************
Function:    main
Purpose:     Initialization, test Tx and Rx double buffering.
*/
main()
{
  int Dev;
  int Err;
  unsigned char Buf[100];
  int Cnt;

  /* Initialize controller */
  aaInstallCtrlCHandler();
  if((Err = aaInit()) != NO_ERR)
  {
    printf("Initialization Failure %x\n",Err);
    aaExit();
    exit(1);
  }

  /* Clear queues */
  for(Dev = 0;Dev < NUMDEV;Dev++)
  {
    q[Dev].OpenTx = FALSE;
    q[Dev].TxIn = 0;
    q[Dev].TxOut = 0;
    q[Dev].RxIn = 0;
    q[Dev].RxOut = 0;
  }

  /* Set up application event functions */
  aaInstallRxEvent(EvRxData);
  aaInstallPeriodicEvent(EvPeriodic);
```

```
  aaEnPeriodicEvent(TRUE);

  /* Test background transmit and receive on device 0.  A loopback
     plug can be installed on device 0 so that all transmitted data is
    received on the same device. */
  printf("To stop test press any key\n");
  if((Err = aaOpen(0,
              COM_TX | COM_RX,
              COM_BAUD_38400,
              COM_PAR_NONE,
              COM_DATABIT_8,
              COM_STOPBIT_1,
              COM_FLOW_NONE,
              COM_DEN_RDA,
              COM_MDM_RTS | COM_MDM_DTR)) != 0)
  {
    printf("Open Failure - Device %d, Error %d\n","0",Err);
    aaExit();
    exit(1);
  }
  q[0].OpenTx = TRUE;

  while(!kbhit())          /* test loop */
  {
    Cnt = EnqTxData(0,
          (unsigned char *)"This string is being written to device 0",
          40);
    if(Cnt > 0)
      printf("Tx %d bytes\n",Cnt);

    delay(100);            /* wait for loopback data  */
    Cnt = DeqRxData(0,Buf,RXBUF_SIZE-1);
                           /* dequeue all Rx data available */
    Buf[Cnt] = NULL;  /* null terminate received string */
    if(Cnt > 0)
      printf("Rx %d bytes, String = %s\n",Cnt,Buf);
  }
  getch();

  /* Exit application */
  q[0].OpenTx = FALSE;
  aaClose(0,COM_MDM_RTS | COM_MDM_DTR);
  aaExit();
  return(0);
}
```

```
/*****************************************************
Function:   EnqTxData
Purpose:    Add data to a Tx queue.
Call:       EnqTxData(Dev,Buf,Cnt)
            int Dev; Device number
            unsigned char *Buf; Buffer with data to add
            int Cnt; Count of bytes to add
Return:     int: Number of bytes added to Tx queue
*/
int EnqTxData(int Dev,unsigned char *Buf,int Cnt)
{
  int i;                   /* balance of bytes to copy after q wrap */
  int NumOpen;             /* num bytes open in Tx buffer */
  int In;                  /* In index into Tx buffer */

  asm cli;                 /* no interrupts until done, do not want
                              periodic event function modifying q
                              while we are working on it */
  In = q[Dev].TxIn;        /* local copy of In index */

  /* Get number bytes open in Tx buffer */
  if((NumOpen = q[Dev].TxOut - In - 1) < 0)
    NumOpen += TXBUF_SIZE;    /* adjust for q wrap */
  if(NumOpen > Cnt)
    NumOpen = Cnt;    /* don't move more than are incoming */
  if(NumOpen == 0)
    return(0);               /* no room in Tx buffer */
  i = NumOpen - (TXBUF_SIZE - In);    /* i = whats left after wrap around */
  if (i < 0)
    i = 0;

  /* Copy to end of Tx buffer */
  memcpy(&q[Dev].TxBuf[In],Buf,NumOpen - i);


  /* Update In index, pnt to beginning of buff if already at end of it */
  In = (In + (NumOpen - i)) % TXBUF_SIZE;

  /* Copy the rest of the buffer, if any left */
  if (i != 0)
  {
    memcpy(q[Dev].TxBuf,&Buf[NumOpen - i],i);
    In = i;
  }
```

```
  /* Update Tx queue In index */
  q[Dev].TxIn = In;
  asm sti;                  /* enable interrupts */
  return(NumOpen);
}

/*********************************************************
Function:   DeqRxData
Purpose:    Remove data from a Rx queue.
Call:       DeqRxData(Dev,Buf,Cnt)
            int Dev; Device number
            unsigned char *Buf;
                        Buffer takes data removed from Rx queue.
            int Cnt; Count of bytes to remove
Return:     int: Number of bytes removed from Rx queue
*/
int DeqRxData(int Dev,unsigned char *Buf,int Cnt)
{
  int i;                    /* balance of bytes to copy after q wrap */
  int Out;                  /* Out index into Rx buffer */
  int BCnt;                 /* count of bytes copied */

  asm cli;                  /* no interrupts until done, do not want
                               periodic event function modifying q
                               while we are working on it */
  Out = q[Dev].RxOut;/* local copy of Out index */

  /* Get number of bytes in Rx buffer */
  if((BCnt = q[Dev].RxIn - Out) < 0)
    BCnt += RXBUF_SIZE;     /* adjust for queue wrap */
  else if(BCnt == 0)
    return(BCnt);           /* nothing in Rx buffer */
  if(Cnt < BCnt)
    BCnt = Cnt;             /* do not move more than asked for */
  i = BCnt - (RXBUF_SIZE - Out);    /* i = whats left after wrap around */
  if(i < 0)
    i = 0;

  /* Copy to end of Rx buffer */
  memcpy(Buf,&q[Dev].RxBuf[Out],BCnt - i);

  /* Updata Out index, point to beginning of buffer if already at end of it */
  Out = (Out + (BCnt - i)) % RXBUF_SIZE;

  /* Copy the rest of the buffer, if any left */
  if (i != 0)
  {
    memcpy(&Buf[BCnt - i],q[Dev].RxBuf,i);
    Out = i;
  }

  /* Update Rx queue Out index */
  q[Dev].RxOut = Out;
  asm sti;                  /* enable interrupts */
  return(BCnt);
}

/*********************************************************
Function:   EvRxData
Purpose:    Receive event function, read data from a serial device
            and add it to a Rx queue.
Call        EvRxData(Dev)
            int Dev; Device number
Return:     void
*/
void EvRxData(int Dev)    /* receive event function */
{
  int i;                    /* balance of bytes to copy after q wrap */
  int NumOpen;              /* num bytes open in Rx buffer */
  int In;                   /* In index into Rx buffer */
  int Cnt;                  /* total count of bytes read */

  In = q[Dev].RxIn;     /* local copy of In index */

  /* Get number bytes open in Rx buffer */
  if((NumOpen = q[Dev].RxOut - In - 1) < 0)
    NumOpen += RXBUF_SIZE;          /* adjust for q wrap */
  if(NumOpen == 0)
    return;                 /* no room in Rx buffer */
  i = NumOpen - (RXBUF_SIZE - In);    /* i = whats left after wrap around */
  if (i < 0)
    i = 0;

  /* Read data in up to end of Rx buffer */
  Cnt = aaRead(Dev,NumOpen - i,&q[Dev].RxBuf[In]);

  /* Update In index, point to beginning of buffer if already at end of it */
  In = (In + Cnt) % RXBUF_SIZE;
```

```
  /* Read more data if any room left at front of buffer and if device wasn't
     already emptied */
  if((i != 0) &&
     (Cnt == NumOpen - i))
  {
     In = aaRead(Dev,i,q[Dev].RxBuf);  /* read balance of data */
  }

  /* Update Rx queue In index */
  q[Dev].RxIn = In;
}

/*********************************************************
Function:    EvPeriodic
Purpose:     Periodic event function, remove data from Tx queues
              and write it to serial devices.
Call:        EvPeriodic(void)
Return:      void
*/
void EvPeriodic(void)
{
  int Dev;              /* device number */
  int i;                /* balance of bytes to copy after q wrap */
  int Out;              /* Out index into Tx buffer */
  int Cnt;              /* number of bytes to write */
  int WCnt;             /* number of bytes actually written */

  for(Dev = 0;Dev < NUMDEV;Dev++) /* check all devs for data to Tx */
  {
    if(!q[Dev].OpenTx)      /* device not open for Tx */
      continue;
    Out = q[Dev].TxOut;     /* local copy of Out index */

    /* Get number of bytes in Tx buffer */
    if((Cnt = q[Dev].TxIn - Out) < 0)
      Cnt += TXBUF_SIZE;      /* adjust for queue wrap */
    else if(Cnt == 0)
      return;               /* nothing in Tx buffer */
    i = Cnt - (TXBUF_SIZE - Out);     /* i = whats left after wrap around */
    if(i < 0)
      i = 0;

    /* Write data to end of Tx buffer */
    WCnt = aaWrite(Dev,Cnt - i,&q[Dev].TxBuf[Out]);

    /* Updata Out index, point to start of buffer if already at end of it */
    Out = (Out + WCnt) % TXBUF_SIZE;

    /* Write more data if any left at front of buffer and if device wasn't
       already filled */
    if((i != 0) &&
       (WCnt == Cnt - i))
    {
      Out = aaWrite(Dev,i,q[Dev].TxBuf);  /* write balance of data */
    }

    /* Update Tx queue Out index */
    q[Dev].TxOut = Out;
  }
}
```