THE IO SYSTEM

December 7, 1984

2600 Liberty Avenue
P. O. Box 2600
Pittsburgh, PA 15230
(412) 355-0900

Accent is a trademark of Carnegie-Mellon University.

Accent and many of its subsystems and support programs were originally developed by the CMU Computer Science Department as part of its Spice Project.

The information in this document is subject to change without notice and should not be construed as a commitment by PERQ Systems Corporation. The company assumes no responsibility for any errors that may appear in this document.

PERQ Systems Corporation will make every effort to keep customers apprised of all documentation changes as quickly as possible. The Reader's Comments card is distributed with this document to request users' critical evaluation to assist us in preparing future documentation.

PERQ, PERQ2, LINQ, and Qnix are trademarks of PERQ Systems Corporation.

Table of Contents Page

List of Figures

List of Tables

1. Theory

The IO (Input/Output) servers allow processes to access the IO devices on a PERQ workstation. A standard interface exists that provides a common mechanism for acquiring use of the device that an IO server manages, for making requests to the server for use of the device, and for relinquishing use of the device. By supplying fairly open-ended definitions for the parameters to the main IO request routines, most of the servers allow processes to manipulate their devices at the lowest level of functionality (i.e., the actual hardware registers). Allowing this low level of control best provides the flexibility that processes may require for their applications.

The primary role of an IO server is to manage the details of device interaction (including physical memory requirements), to provide a convenient means for processes to perform IO operations, to handle the allocation and deallocation of a device, and to ensure the integrity of a device by restricting certain operations and recovering the device when it appears to be hung.

Each IO server is registered with the local Name Server using a name/port pair. The name part of the registration includes the local machine's name, thus identifying the server (and its corresponding device) with the machine and allowing access to devices on remote machines.

2. Use

The IO system provides access to all the IO devices except the hard disk and network. There is one device server for each of the supported IO devices. The supported IO devices are:

Floppy
GPIB
RS232A
RS232B (PERQ2 workstations only)
Speech

These devices are currently supported by a Z80 microprocessor in the workstation that acts as an IO processor.

There is a single set of standard interface routines that provide access to all of the device servers. Each device server has a special port registered with the Name Server that provides unique access to the associated server. To select which server a request is directed to, the port associated with the server is specified as a parameter to one of the standard interface routines.

The definitions throughout this document are given in Pascal. If you are programming in the C language, please refer also to the document "C System Interfaces" in the *Accent Languages Manual*. If you are programming in the Lisp language, see the document "Lisp Interaction with the Accent Operating System" in the *Accent Lisp Manual*. When FORTRAN bacomes available under Accent, the definitions will be the same as in the C language.

2.1. Basic Routines

The user interface to an IO server is provided through a small set of standard routines:

```
Procedure InitIO( ReplyPort: Port );

Function IO_Version(
          ServerPort : ServerNamePort
         ): String;

Function OpenIO(
          ServerPort : ServerNamePort;
       Var IOPort    : ServerIOPort;
          UserPort   : UserEventPort
        ): GeneralReturn;

Function CloseIO(
          IOPort : ServerIOPort
         ): GeneralReturn;

Function SyncIO(
          IOPort          : ServerIOPort;
          Command         : IOCommand;
          CmdBlk          : Pointer;
          CmdBlk_Cnt      : Long;
       Var DataBuf        : Pointer;
       Var DataBuf_Cnt    : Long;
          DataTransferCnt : Long;
          TimeOut         : Long;
       Var Status         : IOStatusBlk
         ): GeneralReturn;

Procedure AsyncIO(
          IOPort             : ServerIOPort;
          Command            : IOCommand;
          CmdBlk             : Pointer;
          CmdBlk_Cnt         : Long;
          DataBuf            : Pointer;
          DataBuf_Cnt        : Long;
          DataTransferCnt    : Long;
          TimeOut            : Long;
          NotifyOnCompletion : Boolean
         );
```

```
Function IOAsynch( IOMsg : Pointer ): Boolean;

  {this raises the following exceptions}

    Exception IOCompletion(
              TransactionID : Long;
              DataBuf       : Pointer;
              DataBuf_Cnt   : Long;
              Status        : IOStatusBlk
             );

    Exception AsyncData(
              SequenceNumber : Long;
              DataBuf        : Pointer;
              DataBuf_Cnt    : Long;
              Status         : IOStatusBlk
             );

    Exception Distress(
              TransactionID : Long;
              DataBuf       : Pointer;
              DataBuf_Cnt   : Long;
              Status        : IOStatusBlk
             );

    Exception Acknowledge( TransactionID : Long );

    Exception ServerFull( TransactionID : Long );
```

These routines are discussed in Section 3.2.

Except for InitIO, AsyncIO, and IOAsynch, each of these is actually a remote procedure call to a server and is implemented by sending a message to and receiving a reply from the server. The message passing interface is provided by modules created by the Matchmaker utility.

2.2. Modes of Interaction

Two modes of interaction--Synchronous and Asynchronous--are available for accessing a server to perform IO operations. The terms Synchronous and Asynchronous, as used here, do not imply anything about the type of data that the device is transferring (e.g., for RS232, it says nothing about whether the data is Bisync or Async); they refer to the style of user interaction with the server.

In the Synchronous mode, the user process makes IO requests through the SyncIO routine. The user process is then blocked until it receives the reply message with the results of the operation. This mode is useful if you do not want to deal with the details of the message passing system, since the Matchmaker-generated interface modules handle the packing, sending, receiving, and unpacking of the messages that implement the remote procedure call. The Synchronous mode is also useful when the user process wants to block itself until a certain operation has been performed.

In the Asynchronous mode, the user process queues IO requests using the AsyncIO routine. The user process is not blocked; it continues with other processing while the IO operation is being performed. Up to ten IO requests can be queued for each server. When the server completes a requested operation, it sends the results back to the user process in a message as an asynchronous event that the process must handle. This requires doing an explicit Receive in order to obtain the event. Event messages are detected either by polling with a Receive on the UserEventPort, or by enabling emergency message interrupts to cause an exception to be raised when an event message arrives. Whichever method is used, the IOAsynch routine is then called. IOAsynch unpacks the message and generates an exception that represents the given event and supplies the information in the message as parameters to the exception call. This makes it unnecessary to know the details of IPC (Inter-Process Communications) message formats.

One important aspect about asynchronous event messages must be pointed out here. When the user process receives the event messages that have been sent by a server, it may find that the messages are not received in the order in which they were sent by the server. This may occur because of the way the Accent Kernel queues up the messages sent to a port. For an emergency message, the Kernel places the message at the head of the

message queue for that port. Thus, if several emergency messages are queued on a port, the receiving process will get them in reverse order. Since all event messages are sent by the server as emergency messages, a means to identify the order of event messages is provided. For all the different event messages (except AsyncData events), the TransactionID supplied with the event provides the correspondence between the event and the IO request (made through AsyncIO) associated with it. For AsyncData events (which are not directly associated with an IO request), the server supplies a SequenceNumber with the event to establish an order among the AsyncData events that the user process receives. The TransactionID and SequenceNumber are discussed in greater detail in Sections 2.3, 2.7, and 3.2.7.

2.3. Function of Basic Routines

This section discusses in more detail the standard routines and their parameters.

Except for the InitIO and IOAsynch routines, the first parameter to each of the basic routines is a port that is created and owned by the particular server and thus identifies the device server to which the user's request is directed. Each server owns two important ports, a ServerNamePort and a ServerIOPort, that provide the access to users. The ServerNamePort is registered with the Name Server and associated with a unique string name identifying the particular server.

The current existing IO servers are registered as:

"[MachineName]GPIBServer"
"[MachineName]FloppyServer"
"[MachineName]RS232AServer"
"[MachineName]RS232BServer"
"[MachineName]SpeechServer"

where MachineName is the name of the machine upon which the device resides. This provides the means to access the devices on other machines across the network as well as access to local devices.

User programs acquire send access to a ServerNamePort through a LookUp call to the NameServer. The ServerNamePort can only be used in the call to open the device or within the IO_Version call. The server grants access to the device to the OpenIO caller by returning its ServerIOPort. The ServerIOPort is then used as the first parameter in all other calls to the server. Some servers (e.g., the Z80 supported device servers) may grant exclusive use of the device and not permit it to be opened again until it has been closed; other servers may permit multiple access.

The InitIO call is not directed to the server. The InitIO procedure is created by Matchmaker and is part of the Matchmaker-generated user interface module that implements (using the message system) the remote procedure calls. The user calls InitIO with the parameter ReplyPort, which is a port in the user's space to which the server is given send access for those remote calls (IO_Version, OpenIO, CloseIO, and SyncIO) that are implemented with a message send followed by an explicit receive. If the ReplyPort equals NullPort in the user call, then InitIO will allocate a port in the user's space for the reply messages for remote calls. The user must call InitIO just once, before the first actual remote call (which should be OpenIO) to a server.

The last parameter in the OpenIO call is the UserEventPort. This is a port owned by the user and to which the server will send asynchronous event messages. If the user specifies NullPort for the UserEventPort in his call to OpenIO, then the Asynchronous interface will not be enabled. This means that the server will ignore calls to AsyncIO and will not pass any other asynchronous

events to the user.

The parameters for the actual IO calls are fairly straightforward. CmdBlk is a generic pointer to which the user can recast his own device specific command block pointer (see Section 3.1 for the definition of IOCmdBlk). CmdBlk_Cnt indicates the number of command bytes to which CmdBlk points. The CmdBlk is required to have a long integer occupying its first 2 words which can be set by the user to provide a transaction ID tag for the IO command request. This tag is then copied into the first 2 words of the Status block when the response to the IO request is made to the user. The ID tag is useful for matching up server responses to IO requests made through AsyncIO. The ID tag is supplied explicitly as the TransactionID parameter in the asynchronous events sent by a server. The remainder of the CmdBlk is completely device specific. DataBuf points to a buffer for data transfers and DataBuf_Cnt is the size of the buffer in bytes. The number of data bytes to transfer is indicated by DataTransferCnt.

DataBuf_Cnt is used as an indicator of how many bytes pointed to by DataBuf are actually transmitted to/from the server in the message associated with the remote procedure call. Thus, for example, in a SyncIO call to read N data bytes, DataBuf should be set to Nil, DataBuf_Cnt should be 0, and DataTransferCnt should be N when the user invokes the remote procedure. Upon return from SyncIO and assuming the server successfully carried out the request, DataBuf will point to a buffer holding the N bytes of data. This buffer will have been created by the kernel when it handles the receipt of the server-generated response message to the remote call for SyncIO. The kernel maps the data pointed to in the response message into the user's address space and the Matchmaker-generated interface module sets DataBuf to point to that piece of memory.

TimeOut, when applicable to a given command, indicates how

long to wait for the command to complete before giving up. In most cases, a value of -1 means to not wait, zero means to wait indefinitely, and a positive value means to wait that many clock ticks, where a tick is approximately one millisecond. The Status block holds information about the success or failure of the IO command along with any available device status bytes. In the event of failure, the information in the Status block may also show how much of the command was performed before failure occurred. The set of IOCommands available will include those like IORead, IOWrite, IOReset, IOSense, etc., that are generally applicable to most devices, as well as some that are device specific. Commands include those for data transfer and control, those for device control and configuration, and simple directives to the server. Three important server commands--IOAbort, IOResume, and IOSuspend--are directives to the servers. They are applicable to all servers and are issued through the AsyncIO call only. These will be discussed in Section 2.7.

The NotifyOnCompletion parameter in AsyncIO provides a means to specify whether or not the IOCompletion event message for the given AsyncIO request will be sent to the user. If NotifyOnCompletion is true, the server sends the results of the operation in an IOCompletion message as soon as the operation completes. If NotifyOnCompletion is false, the server does not send a completion message at all. If, however, the IO operation fails, the server will send a Distress event message to signal the failure, regardless of the value of NotifyOnCompletion.

2.4. Example of SyncIO for GPIB

In this section, the service provided through the SyncIO call, along with descriptions of the parameters, is presented in the context of a specific example. Full definitions for new types are provided in Section 3.1. For GPIB the call and parameters are:

```
SyncIO(     IOPort          : ServerIOPort;
            Command         : IOCommand;
            CmdBlk          : Pointer;
            CmdBlk_Cnt      : Long;
        Var DataBuf         : Pointer;
        Var DataBuf_Cnt     : Long;
            DataTransferCnt : Long;
            TimeOut         : Long;
        Var Status          : IOStatusBlk
       ): GeneralReturn;
```

IOPort — As previously mentioned, this is the GPIB server port returned to the user upon successfully executing the OpenIO call.

Command — This is the user requested IO command. Valid commands for GPIB are:

IOSense	IORead	IODevRead
IOReset	IOWrite	IODevWrite
IOWriteRegisters	IOWriteEOI	IOSetBufferSize
IOFlushInput	IOReadHiVol	
IOFlushOutput	IOWriteHiVol	

These commands are explained in Section 2.5.

CmdBlk — This points to a record holding the CmdIDTag (also known as the transaction ID) as well as device specific command bytes that are required for the command. The user's pointer to his IOCmdBlk for GPIB should be recast to a generic pointer. The definition for CmdBlk is given in Section 3.1.

CmdBlk_Cnt — The number of valid bytes in the CmdBlk. This is always at least 4 to account for the CmdIDTag.

DataBuf — This points to a buffer for the data which is to be sent or received. It is only used with the IORead, IOWrite, IOReadHiVol, IOWriteHiVol, and IODevWrite commands.

DataBuf_Cnt — The number of data bytes held in the buffer pointed to by DataBuf.

DataTransferCnt
: The number of bytes to read / write to / from the data buffer.

TimeOut
: The maximum number of clock ticks to wait before giving up on the command. A TimeOut value that is zero means to wait indefinitely. A TimeOut value of -1 means to not wait at all for some commands (e.g. IORead to get data from the ring buffer) and means to wait indefinitely for other commands (i.e., where to not wait would make no sense). A positive TimeOut value means to wait that many clock ticks for the IO operation to complete. A clock tick is approximately one millisecond. For some commands, the TimeOut is irrelevant and is ignored.

  In some cases, when a command times out, the server will issue a device reset automatically. This is done for those commands for which a message is sent to the Z80. Since the Z80 always gives an ACK / NAK for each command message, a time out would indicate that the device is hung. The only command that can free the device for subsequent commands is device reset. Note that after a device reset, any particular commands that were previously issued to configure the device's operating mode will need to be repeated.

Status
: This shows the success or failure of the Command and, in either case, indicates how much of the command was performed. Also included is device status from the most recently issued IOSense command since the last device reset was issued. The device status can be empty.

  Status is defined by IOStatusBlk, IOSenseStatusBlk, and GPIBSenseStatus (see the definitions in Section 3.1). In the IOStatusBlk, HardStatus is status information provided by the device for the given Command. For IORead, for example, it is the error byte that is supplied with each character in the input ring buffer. SoftStatus is supplied by the server and provides a logical indication of the Command completion status. A list of the values used for SoftStatus is exported by the IODefs module. IOSuccess is the value returned in SoftStatus when the Command is successful. When command bytes are present, CmdBytesTransferred indicates how many of the bytes were actually transferred. DataBytesTransferred serves a similar function when the Command involves data transfer. DeviceStatus for GPIB indicates the last available status from the internal registers of the TMS 9914 GPIB Controller chip that provides the

interface to the GPIB. Refer to the Texas Instruments "TMS 9914 GPIB Controller Data Manual" for a detailed description of this chip.

2.5. Commands for GPIB

In a number of the GPIB commands, the Z80 will wait until the data in its GPIB output ring buffer has been transmitted over the GPIB bus before actually initiating the command. Since some commands seize control and/or reconfigure the bus, it is necessary for the Z80 to wait until its GPIB output buffer has been drained before initiating those commands. This ensures the integrity of previous IOWrite commands which have sent output data for GPIB.

The standard TimeOut mechanism used for most commands is:

```
TimeOut <= 0      Wait indefinitely

TimeOut >  0      Wait TimeOut clock
                  ticks
```

Commands that do time out are followed immediately by a server-issued device reset.

SoftStatus is returned for each command and indicates success or the reason for failure. A list of SoftStatus codes with their meanings can be found in Section 3.1. A command whose parameters are in error will be rejected and SoftStatus will indicate why. This type of error will not be listed in the discussions below. Assuming no parameter errors, the standard values for SoftStatus for most GPIB commands will be:

IOSuccess Command completed successfully.

IOTimeOut Command did not complete within the TimeOut period.

IOUndefinedError
Command was NAKed (i.e., it failed or was rejected) by the Z80.

The following are valid commands for GPIB. Refer to the TMS 9914 Data Book for technical information about GPIB.

IOSense	IOReset
IOWriteRegisters	IOFlushInput
IOFlushOutput	IORead
IOWrite	IOWriteEOI
IOSetBufferSize	IOReadHiVol
IOWriteHiVol	IODevRead
IODevWrite	

IOSense Provides 10 bytes of status information from the Z80. Upon return, DeviceStatus in Status holds the count and the status bytes. The first 6 status bytes represent the register values in the TMS 9914 GPIB Controller chip at the time of the last Z80 GPIB interrupt and the remaining 4 bytes show values current with the issued IOSense. (These are shown in the GPIBSenseStatus record in Section 3.1.) Note that current values for IntStat0 and IntStat1 cannot be obtained since reading those registers dismisses the interrupts that the bit maps in those registers represent. (See TMS 9914 Data Book for more details.) The server also maintains a copy of these 10 status bytes and copies them into the DeviceStatus field of Status for subsequent IOCommands. The server will always zero its copy of DeviceStatus after a device reset. TimeOut for IOSense is standard. SoftStatus will be IOSucess, IOTimeOut, or IOUndefinedError.

IOReset Puts the GPIB Controller into the idle state by issuing a device reset for GPIB to the Z80. This clears the Z80's GPIB input and output buffers (any data is discarded) and puts the TMS 9914 into the idle state by performing a Software Reset aux command. The TMS 9914 interrupt mode is reset for Data In and Data Out interrupts only, the Hdfa/Hdfe aux commands are disabled (in case they were previously set), and any data holdoff is released using Rhdf aux command. The PERQ workstation's GPIB input ring buffer is not affected. This command is also issued implicitly by the server for some of the other commands when they time out. TimeOut for IOReset is ignored since the device reset should never fail.

IOWriteRegisters
Programs the TMS 9914 registers. CmdBlk points to the user's

IOCmdBlk which, for IOWriteRegisters, contains an array of GPIBWriteRegister elements. Each GPIBWriteRegister is a pair of bytes where the first byte indicates the TMS 9914 register and the second byte is the value to be written. CmdBlk_Cnt indicates the total number of bytes and, thus, must be even. TimeOut and SoftStatus are standard. CmdBytesTransferred shows the total number of bytes transferered.

IOFlushInput

Suspends GPIB bus activity and extracts all remaining GPIB input data held in the Z80. This is done by issuing a Tca aux command to the GPIB Controller chip to suspend bus activity and sending all data accumulated in the Z80 GPIB input ring buffer to the PERQ workstation. In addition, any byte that is being held in the controller chip's Data In register is removed and also sent to the PERQ workstation. All data returned is deposited in the workstation's GPIB input ring buffer and can be obtained by the user with the IORead command. Note that the Tca aux command is not issued by the Z80 until the Z80's GPIB output buffer has been drained. TimeOut and SoftStatus are standard.

IOFlushOutput

Flushes the Z80's GPIB output data ring buffer by waiting until all the data has been drained from the buffer. TimeOut and SoftStatus are standard.

IORead Extracts data from the PERQ workstation's GPIB input ring buffer. This does not require any interaction with the Z80 and, thus, a time out does not result in a device reset. When the user makes the remote call, DataBuf should be Nil, DataBuf_Cnt 0, and DataTransferCnt should indicate the number of bytes to read. Upon return from the call, DataBuf will have been set to point to the buffer holding the data sent by the server (and may be Nil). Each character in the PERQ workstation input ring buffer has a status byte (the value of IntStat0 at the time of the Data In interrupt) associated with it. The server extracts characters and puts them into the DataBuf until either the DataTransferCnt is satisfied, a character's status byte shows an error, or no characters remain and the TimeOut expires. DataBytesTransferred is set to the number of characters returned in the DataBuf. SoftStatus is IOSuccess if the command succeeds completely. IONoDataFound is returned if no characters were found and the TimeOut expires. IOTimeOut is returned if the requested DataTransferCnt was not satisfied and the

TimeOut expired. If the server finds a character with its status byte showing an error, it terminates further reading, puts the character into the DataBuf, sets HardStatus with the character's status byte, and sets SoftStatus to the appropriate error (IOCircBufOverflow or IOEndOfInput). For TimeOut, a value of 0 means wait indefinitely, -1 means don't wait, and > 0 means wait that many clock ticks.

IOWrite Sends data to the Z80 to be transmitted by the GPIB controller chip using Data Out interrupts. The Z80 buffers the data and sends an ACK when it has room in its buffer for another packet of data from the PERQ workstation (12 bytes is the most that can be sent in a single data packet). The server handles the user's IOWrite command by packaging the user's data pointed to by DataBuf into 12-byte packets and sending them to the Z80 as indicated above. (IOWrite will be optimal when the DataBuf_Cnt is a multiple of 12.) Transmission of the last of the data bytes onto the GPIB bus can only be verified by the user following up with an IOFlushOutput command or some other command that waits until the Z80 GPIB output buffer has been drained. TimeOut and SoftStatus are standard. DataBytesTransferred will indicate how many bytes were actually sent to the Z80 for transmission.

IOWriteEOI

The same as IOWrite except that the last data byte will be sent with the GPIB bus EOI line set. The user must take care not to send more output data until the Z80 GPIB output buffer drains--otherwise, EOI may be set on the wrong byte. Draining of the buffer can be verified explicitly with IOFlushOutput or implicitly with one of the other commands.

IOSetBufferSize

Sets the size of the physical buffer that is used for the IOReadHiVol and IOWriteHiVol commands. The default size is set to 1024 bytes when the device is allocated by the server in the OpenIO call. CmdBlk_Cnt should be 8 for this command.

IOReadHiVol

Reads data from the GPIB using a DMA channel and thus provides a high transfer rate. As with IORead, DataBuf is Nil and DataBuf_Cnt is 0 when the user makes the remote call and are set appropriately upon return. DataTransferCnt must be greater than 1 (since the Z80 DMA cannot handle a byte count of 1). TimeOut and SoftStatus are standard. DataBytesTransferred indicates the

number of bytes actually read. IOReadHiVol may be programmed to terminate early if EOI is raised by the sending device. When this feature is enabled and occurs, a completion of IOEndofInput is returned for SoftStatus and DataBytesTransferred must be checked to determine the amount of data actually received.

IOWriteHiVol

Writes data to the GPIB using a DMA channel. DataBuf points to a buffer holding DataBuf_Cnt bytes. DataTransferCnt has the same restriction as for IOReadHiVol. TimeOut and SoftStatus are standard. DataBytesTransferred indicates the number of bytes actually written to GPIB and only needs to be checked when SoftStatus is not IOSuccess. The Z80 will wait for the Z80 GPIB output data ring buffer to drain before starting the HiVol operation.

IODevRead and IODevWrite

In their simplest form, can be used just to configure the GPIB bus to change the Talker/Listener device on the bus. The CmdBlk points to a record containing the CmdIDTag followed by a GPIBDevCmdBlk. See Section 3.1 for the definition of GPIBDevCmdBlk.

By setting all fields in GPIBDevCmdBlk to 0 or false except the PrimAddr and SecAddr, IODevRead would simply configure the bus with the device in PrimAddr and SecAddr as the new Talker; IODevWrite would configure a new Listener. SecAddr should be set to 255 if not used. These commands are useful since they reduce a commonly used sequence of IOWriteRegister and IOWrite commands to a single command. They can also be used to configure the bus and set up the GPIB controller chip's interrupt mask registers appropriately for HiVol Read/Write commands. The basic algorithms for DevRead and DevWrite are given at the end of this section.

For IODevRead, CmdBlk_Cnt should be 11. If ReadCount in GPIBDevCmdBlk is non-zero, the Z80 will wait until ReadCount bytes have been read (using Data In interrupts) from the configured GPIB Talker device and the Z80 will then hold off further input using the Hdfa aux command. As the data accumulates in the Z80, it is sent up to the PERQ workstation's GPIB input ring buffer. The Z80 returns (and thus the IODevRead completes) only after the requested number of bytes have been read and sent to the PERQ workstation. To obtain the data, the user must subsequently use the IORead command (i.e., for IODevRead, DataBuf is NIL

and DataBuf_Cnt is 0).

For IODevWrite CmdBlk_Cnt should be 10. DataBuf points to DataBuf_Cnt bytes of data to be transmitted on GPIB using Data Out interrupts. As with the IOWrite command, the data is shipped to the Z80 in packets. Note that the first packet holds the 5 byte GPIBDevCmdBlk plus only 7 data bytes.

For both IODevRead and IODevWrite, TimeOut and SoftStatus are standard. CmdBytesTransferred should be the same as CmdBlk_Cnt. DataBytesTransferred is valid only for IODevWrite and indicates the number of data bytes actually sent to the Z80 for transmission.

The Z80 will wait before initiating the IODevRead and IODevWrite commands until the Z80 GPIB output ring buffer has drained. The basic algorithm for each command is then given below in pseudo-Pascal code fragments and using the following definitions:

AuxReg	9914 Auxiliary Command Register
DataOut	9914 Data Out Register
DataIn	9914 Data In Register
Mask0	9914 Interrupt Mask Register 0
Mask1	9914 Interrupt Mask Register 1

DevWrite:

```
With GPIBDevCmdBlk, GPIBDevCmdBlk.Options do
  begin
    AuxReg := Tca;
    AuxReg := Ton, off;
    AuxReg := Lon, off;
    If SetInt0Mask then Mask0 := Int0Mask;
    If SetInt1Mask then Mask1 := Int1Mask;
    If not OmitBusConfig then
       begin
         DataOut := Unt;
         If not OmitUnlisten then DataOut := Unl;
         If (0 <= PrimAddr) and (PrimAddr <= 30) then
           begin
             DataOut := Mla + PrimAddr;
             If (0 <= SecAddr) and (SecAddr <= 31) then
                 DataOut := Msa + SecAddr;
           end;
         { Note:  We load the DataOut with the first   }
         {        interface command (Unt) and the rest }
         {        are sent using Bus Out interrupts.   }
       end;
```

```
    AuxReg := Ton, on;
    If not OmitGoToStandby then AuxReg := Gts;
    {Now handle the data bytes }
    If DataTransferCnt does not equal 0 then
      begin
        DataOut := first data byte;
        { Remaining data bytes are transmitted using  }
        { Bus Out interrupts                          }
        If ForceEOI then last data byte is sent with EOI;
        If WaitOnData then
           wait till all data sent before returning
        else
           return without waiting;
      end;
  end { with };
```

DevRead:

```
With GPIBDevCmdBlk, GPIBDevCmdBlk.Options do
  begin
    AuxReg := Tca;
    AuxReg := Ton, off;
    AuxReg := Lon, off;
    If not OmitBusConfig then clear DataIn;
    If SetInt0Mask then Mask0 = Int0Mask;
    If SetInt1Mask then Mask1 := Int1Mask;
    If not OmitBusConfig then
      begin
        DataOut := Unt;
        If not OmitUnlisten then DataOut := Unl;
        If (0 <= PrimAddr) and (PrimAddr <= 30) then
          begin
            DataOut := Mta + PrimAddr;
            If (0 <= SecAddr) and (SecAddr <= 31) then
               DataOut := Msa + SecAddr;
          end;
        { Note: We load the DataOut with the first  }
        {       interface command (Unt) and the     }
        {       rest are sent using  Bus Out        }
        {       interrupts.                         }
      end;

    if HoldOffOnEOI then
       AuxReg := Hdfe, on
    else
       AuxReg := Hdfe, off;
    AuxReg := Hdfa, off;
    AuxReg := Rhdf;          {Release any previous holdoff }
    AuxReg := Lon, on;
    AuxReg := Gts;
    If ReadCount <> 0 then
      begin
        Wait until we have input ReadCount data bytes using
        Bus In interrupts and issue
```

```
        AuxReg := Hdfa, on before the last byte is input.
      end
  end { with };
```

2.6. Commands for RS232, Speech and Floppy

The valid SyncIO commands for the other servers are discussed here. The descriptions of the parameters to SyncIO for the other servers are essentially the same as those given for GPIB in Section 2.4. Also, the descriptions of the commands for GPIB in Section 2.5 are applicable except for the differences noted below for each device. The timeout mechanism, use of the status block, and the delay waiting for the device's output ring buffer to drain before initiating certain commands are all similar to the case for GPIB.

2.6.1. RS232A

The valid commands for RS232A are:

IOSense	IORead
IOReset	IOWrite
IOFlushInput	IOSetBufferSize
IOFlushOutput	IOReadHiVol
IOSetBaud	IOWriteHiVol
IOWriteRegisters	

For technical information about RS232, refer to the *ZILOG Z80 SIO Technical Manual*.

IOSense — Only 2 bytes of status information are held in the DeviceStatus record. These correspond to the last available values obtained for the ReadRegisters 0 and 1 of the Serial IO (SIO) chip that implements the RS232 interface.

IOReset — Configures the RS232A channel to handle 9600 baud, 8-bit asynchronous data using 1 1/2 stop bits and no parity. The Data Terminal Ready (DTR) and Request To Send (RTS) signals (RS232 pins 20 and 4) are turned on and the SIO chip is programmed with Auto Enables mode. Auto Enables means that the Clear To Send (CTS) and Data Carrier Detect (DCD) inputs (RS232 pins 5 and 8) are used as the enable signals for the

respective transmission and reception of RS232 data bytes. This means that CTS must be on before the SIO chip will transmit a byte, and the DCD signal must be on before the SIO chip will actually assemble data bytes from the incoming bit stream.

IOFlushInput
: Sends all accumulated data held in the Z80 RS232A input ring buffer to the PERQ, where it is deposited in the PERQ's RS232A input ring buffer.

IOFlushOutput
: Same as for GPIB.

IOSetBaud
: Modifies the rate of the internal baud rate clock used for transmitting and receiving asynchronous RS232 data bytes in their bit-serial form. The baud rate codes to use are defined in Section 3.1 and include a code (RSExt) for synchronous RS232 data where external clocks are used to synchronize the character bit stream. CmdBlk_Cnt should be 6 for IOSetBaud.

IOWriteRegisters
: Permits you to program all SIO registers except WriteRegisters 1 and 2. These two registers are used to control the interrupt modes of the SIO chip. Registers 6 and 7 are used to specify the sync characters used for synchronous RS232 data transfers. Bit definitions for the other four registers are given in Figures 1 through 4.

  The byte masks used, for example, to program the chip for the default settings obtained by IOReset are:

Register #	Byte Mask (in octal)
0	030
0	100
4	110
3	341
5	352

IORead
: Similar to the GPIB example except that the per character status byte is the status byte provided by the SIO chip that accompanies each input data byte assembled by the chip. This byte is the value of SIO ReadRegister 1 which shows such errors as parity, framing, and overrun for the given input data byte as well as the end of frame condition and residue codes for SDLC data. Thus the SoftStatus code returned for IORead will be one of IOSuccess,

Figure 1: Write Register 0 Bit Functions

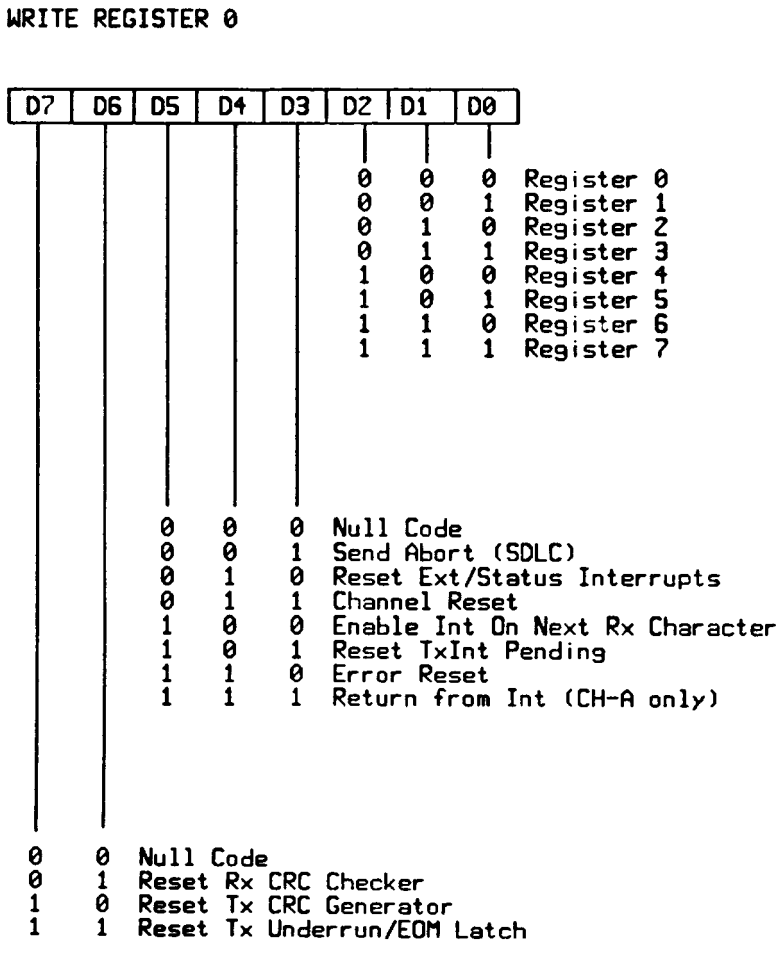


Figure 2: Write Register 3 Bit Functions

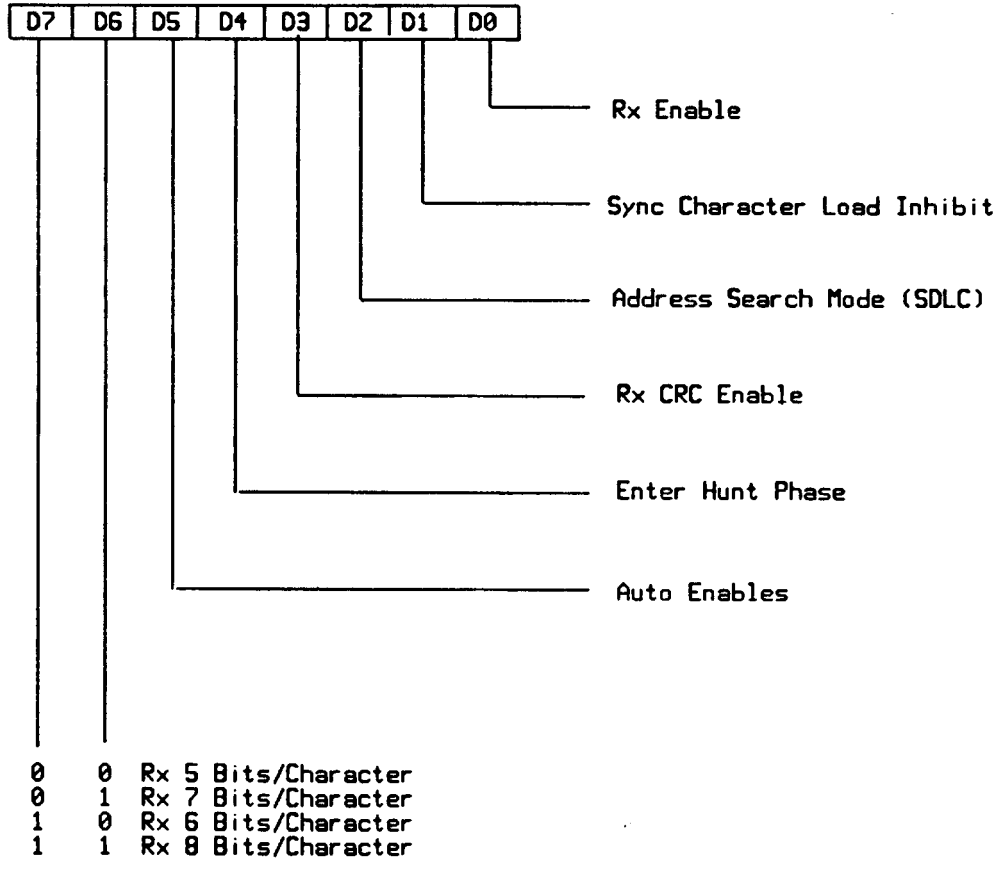


Figure 3: Write Register 4 Bit Functions

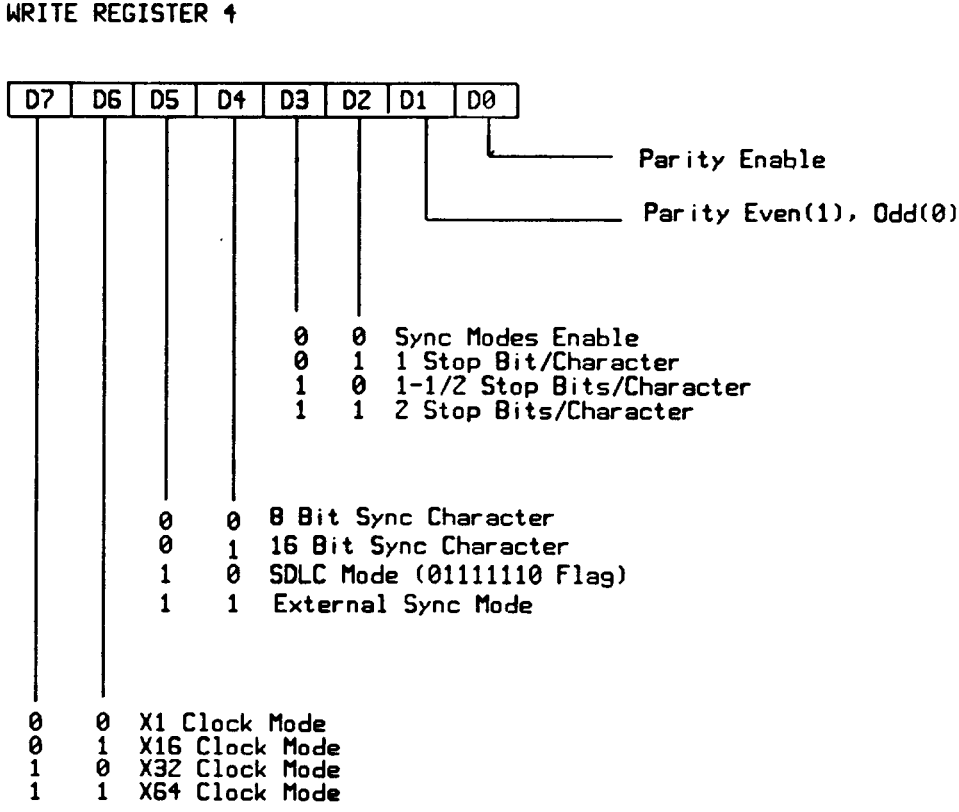
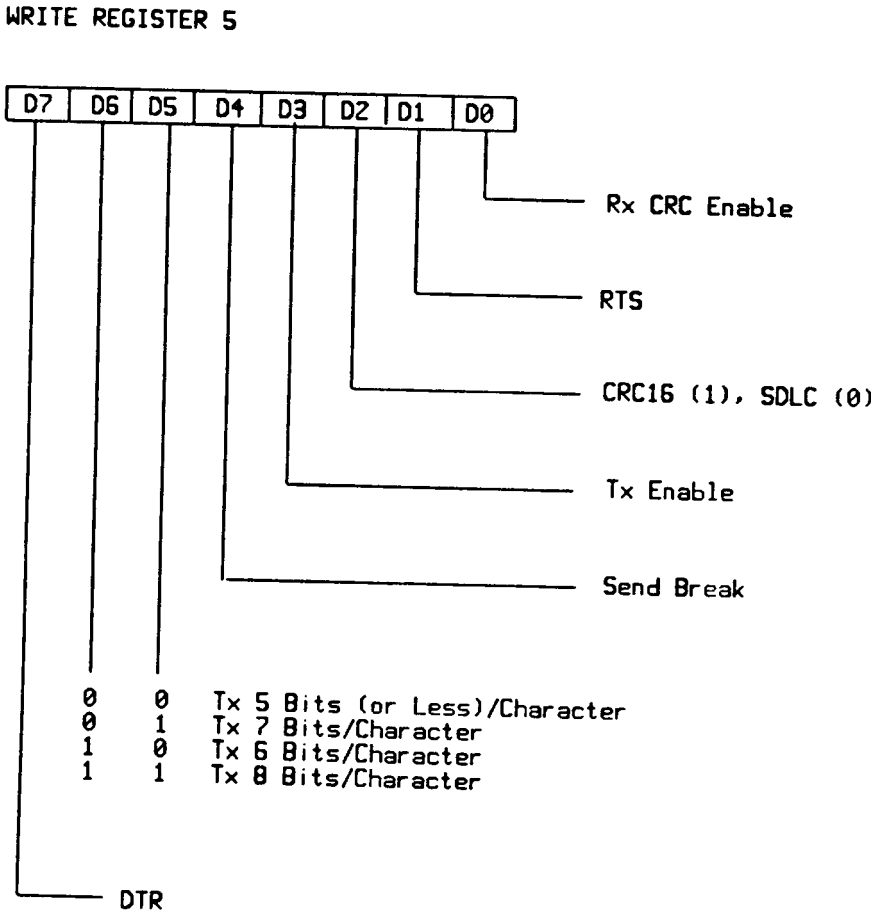


Figure 4: Write Register 5 Bit Functions



IOTimeOut, IONoDataFound, IOCircBufOverFlow, IOParityError, IOFramingError, or IOEndOfFrame.

IOWrite Same as for GPIB.

IOSetBufferSize
Same as for GPIB.

IOReadHiVol
Similar to GPIB. Note, however, that this command is not very useful for RS232 and that the EOI feature of GPIB is not present.

IOWriteHiVol
Same as for GPIB.

2.6.2. RS232B

The description of RS232B is the same as RS232A except that the IOSetBufferSize, IOReadHiVol, and IOWriteHiVol commands are not present. Note that RS232B exists only on PERQ2 workstations.

2.6.3. Speech

The valid commands for speech are:

IOSense	IOSetBaud
IOWrite	IOWriteHiVol
IOReset	IOFlushOutput
IOSetBufferSize	

The descriptions of these commands are the same as for RS232A except as noted below. The speech data output is implemented using the same type of SIO chip as for RS232 but configured instead for synchronous 8-bit data transmission using a single sync character of 252 (octal) which is automatically transmitted at the start of data transfers. The default baud rate is set at 32 KHz.

IOReset Configures the hardware for the default state described above and clears the speech output ring buffer.

IOSetBaud Changes the baud rate for bit-serial transmission of speech data to the actual speech output hardware in the PERQ. The SpeechTxRate field in the CmdBlk is set to obtain the desired bit

rate in the following manner:

PERQ:

$$\text{SpeechTxRate} = \frac{2.456 * 10\hat{}6}{16 * (\text{Desired Baud Rate})}$$

PERQ2:

$$\text{SpeechTxRate} = \frac{4 * 10\hat{}6}{\text{Desired Baud Rate}}$$

A SpeechTxRate of 5 for PERQ and 125 for PERQ2 will give a baud rate of approximately 32 KHz. Note also that workstations that have a portrait/landscape electromagnetic ranging tablet instead of a Summagraphics BitPad tablet have a further restriction. Since the electromagnetic ranging tablet shares the same baud rate clock with Speech, modifying the Speech baud rate may have unpredictable effects on the behavior of the tablet. You may have to unplug the tablet while your application runs, and you should restore the baud rate to 32KHz before your program terminates. This can easily be done by issuing the IOReset command.

2.6.4. Floppy

The valid commands for floppy are given in Table 1, along with some other information that is explained below.

The parameters to SyncIO for Floppy are similar to those for GPIB in Section 2.5. The main difference is in the Status parameter. For Floppy, the DeviceStatus record in the IOStatusBlk is only valid as indicated in Table 1. In the table, "on failure" means that the command failed for reasons other than IOTimeOut, IOUndefinedError, or some parameter error (that is, the floppy controller hardware initiated the command but

Table 1: Valid Commands for Floppy

IO Command	CmdBlk_Cnt	Returns Device Status	Returns Hard Status
IOReset	4	on failure	on failure
IOSense	4	on success	no
IOSenseDrive	4	on success	no
IOReadID	6	on success	no
IOSetDensity	5	no	no
IORecalibrate	4	on failure	on failure
IOSeek	7	on failure	on failure
IOFormat	8	on failure	on failure
IORead	8	on failure	on failure
IOWrite	8	on failure	on failure

failed to complete it). The controller always provides status information at the termination of each command. These bytes are returned to you "on failure" as the DeviceStatus record, which for floppy is a FloppyResultStatus record as defined in IODefs.Pas. Also "on failure," HardStatus in Status will be assigned the first byte of the device status, which is Status Register 0 from the controller and is represented in the FloppyResultStatus record as:

```
Unit        : Bit2     {always 0}
Head        : Bit1
NotReady    : Boolean
EquipFault  : Boolean
SeekEnd     : Boolean
IntrCode    : ( 00 - Normal
              01 - Abnormal
              10 - InvalidCmd
              11 - DriveRdyChange)
```

DeviceStatus is also returned "on success" for those commands listed in the table whose function it is to explicitly extract status about the last floppy operation or current drive state. Also shown in the table is the CmdBlk_Cnt required for each of the floppy commands. The command bytes for the commands involving actual floppy IO include a device address specifying the Unit, Head, Cylinder, and Sector (see the definition of IOCmdBlk in Section 3.1). The ranges of these parameters are:

Unit:	always 0
Head:	0 ... 1
Cylinder:	0 ... 76
Sector:	1 ... 26

For read/write commands, the logical mapping of floppy sectors is done by first increasing the sector number, then the cylinder number, and lastly the head number. For IORead and IOWrite, the starting address on the floppy is given in the CmdBlk and DataTransferCnt indicates how many bytes should be read/written. To satisfy the requested number of bytes, the server will continue to read/write consecutively numbered sectors and will continue, if necessary, by advancing to the next cylinder on the same side. When the last cylinder on side 0 is read/written, it switches to side 1 at cylinder 0 and sector 1 and continues until the byte count is satisfied. Thus the server will implicitly do seek operations in order to satisfy the DataTransferCnt. If the DataTransferCnt given is greater than the number of bytes that could possibly be read/written from/to the floppy from the starting address to the end of the floppy, then the server will not attempt to start the command and will reject it with an error code of IONotEnoughData / IONotEnoughRoom.

The other floppy commands are fairly straightforward. The IOFormat command is used to format all 26 sectors of a single track designated by the address in the CmdBlk. You must also specify in the CmdBlk the default data pattern to be written into the formatted sectors of that track. DataBuf for IOFormat must point to a buffer holding 26 4-byte SectorIDs for each of the 26 consecutive sectors on the track. Each 4-byte SectorID is given as:

Byte	Information	Range
1	Cylinder	0...76
2	Head	0...1
3	Sector	1...26
4	SectorSizeCode	0...1

where a SectorSizeCode of 0 is used for single density with 128 bytes per sector and 1 is used for double density with 256 bytes per sector. The server does not check the 104 (26 x 4) byte buffer of SectorIDs for correctness. You can also perform hardware interleaving of floppy sectors by simply specifying the desired sector number in each SectorID. Thus if the list of sector numbers specified in the 26 consecutive SectorIDs is given by 1, 14, 2, 15, 3, 16, . . ., 12, 25, 13, 26, then a hardware interleave factor of 2 is achieved on that track. The IORecalibrate command forces the floppy to seek to head 0 and cylinder 0. IOReset does a device reset followed by a recalibrate. The timeout mechanism for floppy is the same as that described for GPIB in Section 2.5.

2.7. Features of AsyncIO

The IOCommands for AsyncIO include all those available for SyncIO. These commands are used in the same way as for SyncIO, but results are returned differently. There are also three additional commands for AsyncIO that are applicable to all servers: IOAbort, IOSuspend, and IOResume. These three commands are directives to the server and affect the handling of queued IO requests. Their actions are:

IOAbort
: Aborts all queued requests

IOSuspend
: Suspends further processing of queued requests

IOResume
: Resumes the processing of suspended requests

These commands are typically used when the server sends a Distress event to the user process (see Section 3.2.6). When the server receives one of these commands, it is not queued like other commands; instead, it acts on it immediately and returns an Acknowledge event to confirm the action requested.

Another command, IOSetStream, is available for GPIB and RS232. IOSetStream is used to turn on/off the default handling of input data that arrives spontaneously from a device once the device has been enabled. The CmdBlk for IOSetStream includes two parameters, EnableStream and BlockingFactor. If EnableStream is false, the server simply lets data accumulate in the device input buffer; the data can only be obtained with explicit IORead commands. This is the state of a device when it is first opened. If EnableStream is true, the server automatically sends the spontaneous input data to the user process without requiring an IORead command. The data is sent as an AsyncData event message (see Section 3.2.6) which the user must handle. An order for these AsyncData events is provided by the SequenceNumber parameter supplied with the event (see Section 3.2.7). BlockingFactor is used when EnableStream is true. BlockingFactor specifies the number of data bytes that are sent in a single AsyncData event message. The server will accumulate the data and send AsyncData event messages containing the specified number of data bytes. Note that using IOSetStream only provides a default handling for the input data stream. IORead commands may still be used, especially if you want to get a number of bytes greater or less than the current BlockingFactor value. When a server has an IORead command in its queue, it will not send any AsyncData event messages until after it has processed and satisfied the queued IORead request. Currently, the maximum value for BlockingFactor is 1024 bytes.

3. Definitions

The data type definitions in Section 3.1 are found in module IODefs in the file IODefs.Pas in LibPascal. The routine definitions in Section 3.2 are found in module IO in IOUser.Pas in LibPascal.

3.1. Type Definitions

```
Module IODefs;
{-----------------------------------------------------------------------
{ IODefs.Pas
{
{ Dirk Kalp
{
{ Feb 24, 1984
{
{ Copyright (C) Perq Systems Corporation, 1984
{
{-----------------------------------------------------------------------
{
{ Abstract:
{   This module provides the definitions needed for the
{   IO system.  It is intended for use by Clients and
{   Servers.
{
{-----------------------------------------------------------------------


{*****************}    Exports    {********************}

imports AccentType from AccentType;


const
   {}
   { Define return values for IO errors.
   {}
   IOBaseMsgID              = 4000;

   IOSuccess                = SUCCESS;

   IOErr                    = IOBaseMsgID;
   IOUndefinedError         = IOErr + 1;
   IOTimeOut                = IOErr + 2;
   IODeviceNotFree          = IOErr + 3;
```

```
   IOInvalidIOPort            = IOErr + 4;
   IOBadUserEventPort         = IOErr + 5;
   IOBadPortReference         = IOErr + 6;
   IOIllegalCommand           = IOErr + 7;
   IOBadCmdBlkCount           = IOErr + 8;
   IOBadDataByteCount         = IOErr + 9;
   IOBadRegisterNumber        = IOErr + 10;
   IONotEnoughRoom            = IOErr + 11;
   IOBadBaudRate              = IOErr + 12;
   IONoDataFound              = IOErr + 13;
   IOOverRun                  = IOErr + 14;
   IOParityError              = IOErr + 15;
   IOFramingError             = IOErr + 16;
   IOCircBufOverFlow          = IOErr + 17;
   IOEndOfFrame               = IOErr + 18;
   IOEndOfInput               = IOErr + 19;
   IOBadSectorNumber          = IOErr + 20;
   IOBadCylinderNumber        = IOErr + 21;
   IOBadHeadNumber            = IOErr + 22;
   IOUndeterminedEquipFault   = IOErr + 23;
   IODeviceNotReady           = IOErr + 24;
   IOMissingDataAddrMark      = IOErr + 25;
   IOMissingHeaderAddrMark    = IOErr + 26;
   IODeviceNotWritable        = IOErr + 27;
   IOSectorNotFound           = IOErr + 28;
   IODataCRCError             = IOErr + 29;
   IOHeaderCRCError           = IOErr + 30;
   IOBadTrack                 = IOErr + 31;
   IOCylinderMisMatch         = IOErr + 32;
   IODriveReadyChanged        = IOErr + 33;
   IONotEnoughData            = IOErr + 34;
   IOBadBufferSize            = IOErr + 35;
   IOExclusionFailure         = IOErr + 36;
   IOCancelled                = IOErr + 37;
   IOStartIOComplete          = IOErr + 38;
   IOMustBeAsynchronous       = IOErr + 39;
   IOBadBlockingFactor        = IOErr + 40;
   IOMustEnableAsyncIO        = IOErr + 41;
   IOServerFull               = IOErr + 42;
   IOAborted                  = IOErr + 43;


type
   ServerNamePort = Port;
   ServerIOPort   = Port;
   UserEventPort  = Port;

   IOCommand      =
     (IOSense,          IOReset,         IOWriteRegisters,
      IOFlushInput,     IOFlushOutPut,   IORead,
      IOWrite,          IOWriteEOI,      IOReadHiVol,
      IOWriteHiVol,     IODevRead,       IODevWrite,
      IOSetBaud,        IOSetStream,     IOSetAttention,
      IOAbort,          IOSuspend,       IOResume,
      IOSeek,           IORecalibrate,   IOFormat,
```

```
   IOReadID,          IOSenseDrive,    IOSetDensity,
   IOSetBufferSize,   IONullCmd);

pIOMessage        ^IOMessage;
IOMessage         = record
                      Head : Msg;
                      Body : array[0..1023] of integer;
          {}
          { Note: The size of the Body is dependent
          {       upon the maximum size of messages
          {       that are passed between the Client
          {       and Server. This size is determined
          {       by the number and type of the parameters
          {       in the remote calls, the IPC conventions
          {       for packing data in-line, and the added
          {       parameters in the message that MatchMaker
          {       inserts for coded return values.
          {       The actual size needed for IO messages
          {       is 46 given current IPC conventions,
          {       MatchMaker requirements, and actual
          {       parameters for the remote calls. Future
          {       changes to any of the above may require
          {       modifying the size for the Body. Accent
          {       gurus say that 1024 is a reasonably safe
          {       maximum for the time being and that we
          {       should just use that here.
          {}
                  end;



GPIBWriteRegister = packed record
                      case RegNum: Bit8 { really 0..6 } of
                        0: ( { To be defined later! } );
                        1: (RegVal : Bit8)
                    end;

SIOWriteRegister = packed record
                      case RegNum: Bit8 { really 0..7 } of
                        0: ( { To be defined later! } );
                        1: (RegVal : Bit8)
                    end;

GPIBDevCmdHead = packed record
  Options: packed record  { Bitmap to select cmd actions }
             SetInt0Mask     : boolean;
             SetInt1Mask     : boolean;
             OmitBusConfig   : boolean;
             OmitUnListen    : boolean;
             case boolean of
               true : ( { for IODevRead cmd }
                        HoldOffOnEOI     : boolean;
                        unused2          : Bit3
                      );
               false: ( { for IODevWrite cmd }
```

```
                        OmitGoToStandby : boolean;
                        WaitOnData      : boolean;
                        ForceEOI        : boolean;
                        unused1         : Bit1
                       );
            end;
  IntOMask  : Bit8;   { Mask for 9914 Interrupt Reg 0 }
  IntlMask  : Bit8;   { Mask for 9914 Interrupt Reg 1 }
  PrimAddr  : Bit8;   { Primary Address of device }
  SecAddr   : Bit8;   { Secondary Address of device }
  case boolean of
     true:  ( { for IODevRead }
              ReadCount : Bit8
            );
     false: ( { for IODevWrite }
              { nothing }
            );
                end;

DensityType    = (SingleDensity, DoubleDensity);

pIOCmdBlk = ^IOCmdBlk;
IOCmdBlk  = packed record
   CmdIDTag: long;
   case integer {IODevice} of
      0: {No Device for generic access}
         (case integer of
            0: {Byte access}
               (CmdByte: packed array[0..0] of Bit8);

            1: {Word access}
               (CmdWord: packed array[0..0] of integer);

            2: {Register access}
               (WriteReg: packed array[stretch(0)..stretch(0)]
                                 of packed record
                                    RegNum: Bit8;
                                    RegVal: Bit8
                                 end)
          );


      1: {GPIB}
         (case IOCommand of
            IOSetAttention: { AsyncIO only }
               (GPIBEnableATN: boolean);

            IOSetStream: { AsyncIO only }
               (GPIBEnableStream: boolean;
                GPIBBlockingFactor: integer);

            IOSetBufferSize:
               (GPIBBufferSize: long);

            IOWriteRegisters:
               (GPIBWriteReg: packed array
```

```
                                    [stretch(0)..stretch(0)]
                                     of  GPIBWriteRegister);

      IODevRead, IODevWrite:
         (GPIBDevCmdBlk: GPIBDevCmdHead)
   );


2: {RS232A, RS232B}
   (case IOCommand of
      IOSetAttention: { AsyncIO only }
         (RS232EnableATN: boolean);

      IOSetStream: { AsyncIO only }
         (RS232EnableStream: boolean;
          RS232BlockingFactor: integer);

      IOSetBufferSize: { RS232A only }
         (RS232BufferSize: long);

      IOSetBaud:
         (RS232TxBaud: Bit8;
          RS232RxBaud: Bit8);

      IOWriteRegisters:
         (RS232WriteReg: packed array
                            [stretch(0)..stretch(0)]
                             of  SIOWriteRegister)
   );


3: {Speech}
   (case IOCommand of
      IOSetAttention: { AsyncIO only }
         (SpeechEnableATN: boolean);

      IOSetBufferSize:
         (SpeechBufferSize: long);

      IOSetBaud:
         (SpeechTxRate: integer);

      IOWriteRegisters:
         (SpeechWriteReg: packed array
                            [stretch(0)..stretch(0)]
                             of  SIOWriteRegister)
   );


4: {Floppy}
   (case IOCommand of
      IOSetAttention: { AsyncIO only }
         (FloppyEnableATN: boolean);

      IOSetDensity:
```

```
            (FloppyDensity: DensityType);

         IORead, IOWrite, IOFormat, IOSeek, IORecalibrate,
         IOReadID, IOSenseDrive:
            (FloppyUnit   : Bit8;
             FloppyHead   : Bit8;
             case IOCommand of
                IORead, IOWrite, IOFormat, IOSeek:
                   (FloppyCylinder: Bit8;
                    case IOCommand of
                       IORead, IOWrite:
                          (FloppySector: Bit8);
                       IOFormat:
                          (FloppyFmtData: Bit8)
                   )
            )
      )
end;


GPIBSenseStatus = packed record
                        { IOSense to GPIB provides }
     { first 6 bytes are status at time of last interrupt }
        IntStat0    : Bit8;   { Interrupt Status 0 }
        IntStat1    : Bit8;   { Interrupt Status 1 }
        IntAddrStat : Bit8;   { Address Status }
        IntBusStat  : Bit8;   { Bus Status }
        IntAddrSwch : Bit8;   { Address Switch }
        IntCmdPass  : Bit8;   { Command Pass Through }

        { next 4 bytes are current status  }
        CurAddrStat : Bit8;   { Address Status (now) }
        CurBusStat  : Bit8;   { Bus Status (now) }
        CurAddrSwch : Bit8;   { Address switch (now) }
        CurCmdPass  : Bit8;   { CurCmdPass (now) }
      end;

SIOSenseStatus = packed record
        { Read General Status - SIO Chip's
          Read Register #0 }
        RxCharAvailable  :  boolean;
        IntPending       :  boolean;
        TxBufferEmpty    :  boolean;
        DCD              :  boolean;
        SyncHunt         :  boolean;
        CTS              :  boolean;
        TransmitUnderRun :  boolean;
        BreakAbort       :  boolean;

        { Read Special Condition - SIO Chip's
          Read Register #1 }
        AllSent          :  boolean;
        Residue          :  Bit3;
        ParityError      :  boolean;
        RxOverRun        :  boolean;
        CrcFramingError  :  boolean;
```

```
        EndOfFrame      : boolean;
    end;

FloppyResultType  = (NoStatus,    {no status available}
         HeadChange,  {status from Seek or Recalibrate}
         DriveSense,  {status from SenseDrive}
         CmdResults   {status from other drive cmds}
        );

FloppyResultStatus = packed record
           { Floppy device status is always returned as  }
           { part of the result phase of a cmd to the    }
           { controller.  Z80 simply maintains a copy of }
           { the status results for the last cmd.        }
           StatusType: FloppyResultType;
           Unused    : Bit6;
           case FloppyResultType of
              NoStatus: ({nothing});
              HeadChange, DriveSense, CmdResults:
                 (Unit: Bit2;
                  Head: Bit1;
                  case FloppyResultType of
                     DriveSense:
                        (TwoSided      : boolean;
                         AtTrack0      : boolean;
                         DriveReady    : boolean;
                         WriteProtected: boolean;
                         DriveFault    : boolean
                        );
                     HeadChange, CmdResults:
                        (NotReady  : boolean;
                         EquipFault: boolean;
                         SeekEnd   : boolean;
                         IntrCode  : (Normal, Abnormal,
                                      InvalidCmd,
                                      DriveRdyChange);
                         case FloppyResultType of
                            HeadChange:
                              (PresentCylinder: Bit8);
                            CmdResults:
                              (NoAddrMark     : boolean;
                               NotWritable    : boolean;
                               NoData         : boolean;
                               Unused1        : Bit1;
                               Overrun        : boolean;
                               DataError      : boolean;
                               Unused2        : Bit1;
                               TrackEnd       : boolean;
                               NoDataAddrMark: boolean;
                               BadTrack       : boolean;
                               ScanFail       : boolean;
                               ScanHit        : boolean;
                               WrongCylinder  : boolean;
                               DataCRCError   : boolean;
                               ControlMark    : boolean;
                               Unused3        : Bit1;
```

```
                                    CylinderID     : Bit8;
                                    HeadID         : Bit8;
                                    SectorID       : Bit8;
                                    SectorSizeCode: Bit8
                                   )
                            )
                    )
             end;

   IOSenseStatusBlk = packed record
           StatusCnt: Bit8;
           case integer {IODevice} of
               1: {GPIB}
                  (GPIBStatus: GPIBSenseStatus);
               2: {RS232A, RS232B, Speech}
                  (SIOStatus: SIOSenseStatus);
               3: {Floppy}
                  (FloppyStatus: FloppyResultStatus);
               0: {otherwise} { for generic access }
                  (case integer of
                     1: (StatusByte: packed array
                           [stretch(1)..stretch(1)]
                           of Bit8);
                     2: (StatusWord: packed array
                           [stretch(1)..stretch(1)]
                           of integer);
                     3: (SByte     : packed array
                        [stretch(1)..stretch(12)]
                           of Bit8)
                  )
          end;

   IOStatusBlk  = packed record
                     CmdIDTag             : long;
                     HardStatus           : integer;
                     SoftStatus           : integer;
                     CmdBytesTransferred  : long;
                     DataBytesTransferred : long;
                     DeviceStatus         : IOSenseStatusBlk;
                  end;


const
   {}
   { Baud rate codes for RS232.
   {}
   RSExt          = 0;
   RS110          = 1;
   RS150          = 2;
   RS300          = 3;
   RS600          = 4;
   RS1200         = 5;
   RS2400         = 6;
   RS4800         = 7;
   RS9600         = 8;
   RS19200        = 9;   { only for PERQ2 }
```

3.2. Routine Definitions

3.2.1. Specifying interface reply port

```
Procedure InitIO( ReplyPort: Port );
```

Abstract:

Specifies the port in the user process's port space that will be used to accomplish the remote procedure calls to the IO servers. The port specified serves as the reply port upon which the response from the server will be received. This procedure should be called once and before any of the other routines are used.

Parameters:

ReplyPort Port in your process's port space. If NullPort is specified, then InitIO will create a new port

Returns:

None

3.2.2. Getting version number of a server

```
Function IO_Version(
     ServerPort : ServerNamePort
    ): String;
```

Abstract:

Returns a string that identifies the version number of the server.

Parameters:

ServerPort

Port that identifies the IO server to which the request is directed. It is the port obtained from the Name Server with a LookUp call

Returns:

The server's version number in string form

3.2.3. Acquiring right to use a device

```
Function OpenIO(
     ServerPort : ServerNamePort;
 Var IOPort     : ServerIOPort;
     UserPort   : UserEventPort
   ): GeneralReturn;
```

Abstract:

Acquires the right to use the device managed by the selected IO server. The server grants access to a process by returning a port that the process then uses in the other IO calls. Some servers grant exclusive use to a single process and thus OpenIO performed by other processes will be rejected until the current process relinquishes access with the CloseIO call. OpenIO will always enable the Synchronous interface to the server and will only enable the Asynchronous interface if a UserEventPort is specified. To enable the Synchronous interface only, specify NullPort for the UserEventPort.

Parameters:

ServerPort
: Port identifying the IO server to which your request is directed. It is the port obtained from the Name Server

IOPort
: Port returned by the server (if the OpenIO request is granted) that the caller uses to make subsequent IO requests.

UserPort
: Port upon which the calling process wishes to receive Asynchronous events. Specifying NullPort disables the Asynchronous interface

Returns:

IOSuccess
: If the request is granted

IODeviceNotFree
: If an exclusive-use device is already allocated

3.2.4. Closing a device

```
Function CloseIO(
     IOPort : ServerIOPort
    ): GeneralReturn;
```

Abstract:

CloseIO is called when you are through using the device managed by the selected server. The specified ServerIOPort identifies to which server the CloseIO is being directed. CloseIO is executed immediately and thus any queued IO requests or asynchronous event messages will be aborted and discarded.

Parameters:

IOPort — ServerIOPort that uniquely identifies the server to which the CloseIO is directed. It is the ServerIOPort returned by the server in the OpenIO call

Returns:

IOSuccess if no errors; otherwise returns an identifying error code.

3.2.5. Performing synchronous IO operations

```
Function SyncIO(
     IOPort          : ServerIOPort;
     Command         : IOCommand;
     CmdBlk          : Pointer;
     CmdBlk_Cnt      : Long;
 Var DataBuf         : Pointer;
 Var DataBuf_Cnt     : Long;
     DataTransferCnt : Long;
     TimeOut         : Long;
 Var Status          : IOStatusBlk
    ): GeneralReturn;
```

Abstract:

SyncIO is a primary call to be used in performing IO operations on the device managed by the selected IO server. The specified ServerIOPort identifies the server to which the request is directed. The other parameters are specified according to the particular requirements for the device and the valid set of commands applicable to the device. These are discussed in more detail in Chapter 2 and are listed in Section 3.1. The server performs the operation and the SyncIO call returns whenever the operation completes or times out. A

device-dependent status block is returned that shows the degree of success in the completion of the operation. The set of IO error codes as well as the status block definition can also be found in Section 3.1. In most cases the server will automatically issue a device reset when an operation times out. For calls that don't involve the return of data, the server always sets the DataBuf parameter to Nil. Thus you must be careful about the parameter you supply for DataBuf so that you don't lose your reference to a piece of memory inadvertently.

Parameters:

IOPort — ServerIOPort that uniquely identifies the server to which the SyncIO is directed. It is the ServerIOPort returned by the server in the OpenIO call

Command — IOCommand to perform

CmdBlk — Pointer to the device specific command block. For most commands, you can create a pointer to an IOCmdBlk and just recast CmdBlk to point to it. For other commands (like IOWriteRegisters) that have an arbitrary number of command bytes, you will need to create a large enough buffer to accommodate your command bytes and set CmdBlk to point to it. You may also want to treat your command buffer as an IOCmdBlk for purposes of using the record field names to assign the command byte values. Note that the first two words of the command block are always treated as a CmdIdTag (or Transaction ID) by the server and will be copied into the status block prior to return. The CmdIdTag is primarily useful in the AsyncIO interface to the server

CmdBlk_Cnt — Number of command bytes pointed to by the CmdBlk parameters

DataBuf — Pointer to the data. For commands that read data from a device, DataBuf should be Nil. Upon return, DataBuf will point to a buffer holding the data read. When you are through with the buffer pointed to by DataBuf, you should use the InvalidateMemory call (see the document "Kernel Interface" in this manual) to deallocate the buffer. For commands that write data to a device, DataBuf will point to your buffer of data to output. Since the server will always set DataBuf to Nil prior to return for commands that don't return input data, you must take care here not to lose your pointer reference to the buffer used in

the call. This is easily done by using a generic pointer variable for DataBuf that has also been set to point to your output buffer

Databuf_Cnt
: Number of bytes in the buffer pointed to by DataBuf. For read commands, this is the number of data bytes the server has returned to you. For read commands you should set DataBuf_Cnt to 0 prior to the call. (This eliminates the overhead of passing useless data in the message to the server and prevents any potential invalid memory references.) For write commands, this is the number of bytes in your output buffer

DataTransferCnt
: Number of bytes that you want to read/write from/to the device

TimeOut
: Number of milliseconds that the server should wait for the operation to complete. A value of 0 means to wait indefinitely. A negative value means to not wait (when that makes sense) and otherwise has the same meaning as 0

Status
: Completion status of the operation. Included here is the DeviceStatus, which is device dependent and may hold the values of device status registers when they were last obtained from the device. The current values are explicitly obtained when the IOSense command is issued

Returns:

IOSuccess if the operation succeeds completely; otherwise returns an identifying error code. If the command fails, Status can also be checked for more detailed error information as well as an indication of how much of the command succeeded.

3.2.6. Performing asynchronous IO operations

```
Procedure AsyncIO(
    IOPort              : ServerIOPort;
    Command             : IOCommand;
    CmdBlk              : Pointer;
    CmdBlk_Cnt          : Long;
    DataBuf             : Pointer;
    DataBuf_Cnt         : Long;
    DataTransferCnt     : Long;
    TimeOut             : Long;
    NotifyOnCompletion  : Boolean
   );
```

Abstract:

AsyncIO is a primary call to be used in performing actual IO operations on the device managed by the selected IO server. The specified ServerIOPort identifies the server to which the request is directed. The other parameters are specified according to the particular requirements for the device and the valid set of commands applicable to the device. These are discussed in more detail in Chapter 2 and are also presented in Section 3.1.

Unlike the SyncIO call, the AsyncIO call simply results in the server queuing the IO request and returning immediately to the caller without returning any results. This permits the caller to continue with other processing while the IO request is being handled. The number of IO requests that may be queued within a server is limited to 10. The server processes the IO requests from the queue in the order of their arrival and returns the results of each operation in an IO Event message which is sent to the user's UserEventPort (previously specified in the OpenIO call) as an Emergency message which the user must handle. As mentioned in Section 2.2, the user may receive these event messages by explicit polling or through the Emergency message interrupt mechanism.

Once the actual Receive has been performed to obtain the event message, the user should call the IOAsynch routine, which unpacks the message and generates the appropriate exception representing the event. For an operation that has completed normally, IOAsynch generates the IOCompletion exception with the actual results of the operation being supplied as parameters to the exception. IO requests that do not complete normally result in a different exception being generated by the call to IOAsynch in accordance with the type of error being reported (see the explanation of IOAsynch in Section 3.2.7).

Some IO requests can only be issued through the AsyncIO call and are handled immediately rather than queued. These are the commands IOAbort, IOSuspend, and IOResume which serve as directives to the server to control the handling of the queued IO requests. These have been discussed in Section 2.7 and in Section 3.2.7.

Intermixing AsyncIO and SyncIO calls is permitted. Of course, one must realize that a SyncIO call does not return until its IO request has been carried out. And that means that any queued IO requests made prior to the SyncIO will have been handled along with the sending of the appropriate IO event messages before the SyncIO returns.

Parameters:

IOPort — ServerIOPort that uniquely identifies the server to which the AsyncIO is directed. The port to use here is the ServerIOPort returned by the server in the OpenIO call

Command — IOCommand to perform

CmdBlk — Pointer to the device specific command block. For most commands, you can create a pointer to an IOCmdBlk and just recast CmdBlk to point to it. For other commands (like IOWriteRegisters) that have an arbitrary number of command bytes, you will need to create a large enough buffer to accommodate your command bytes and set CmdBlk to point to it. You may also want to treat your command buffer as an IOCmdBlk for purposes of using the record field names to assign the command byte values. Note that the first two words of the command block are always treated as a CmdIDTag (or Transaction ID) by the server. When the server processes the request and returns the resultant IO event message, it copies the CmdIDTag into the status block and also supplies it explicitly as the TransactionID parameter in the event message. Thus the Transaction ID provides the correspondence between IO requests and results

CmdBlk_Cnt — Number of command bytes pointed to by CmdBlk

DataBuf — Pointer to the data. This should be Nil except for commands that write data to the device

DataBuf_Cnt — Number of bytes pointed to by DataBuf. It should be set to 0 except for write commands

DataTransferCnt — Number of bytes that you want to read / write to / from the device

TimeOut Number of milliseconds that the server should wait for the operation to complete after the request is removed from the queue for actual processing. A value of 0 means to wait indefinitely. A negative value means to not wait (when that makes sense) and otherwise has the same meaning as 0

NotifyOnCompletion
One of:

True server will respond with the IOCompletion event message holding the results of the operation as soon as the operation completes

False server does not send an IOCompletion message for this request

If an IO request does not complete normally, the server will send the Distress event message showing the error regardless of the NotifyOnCompletion parameter

Returns:

Nothing returned. Results of the operations are asynchronously returned later as IO event messages

3.2.7. Handling IO event messages

```
Function IOAsynch(
     IOMsg : Pointer
    ): Boolean;
```

Abstract:

IOAsynch is used in conjunction with the Asynchronous IO interface. When an IO event message is received from a server, call IOAsynch to unpack the message contents. IOAsynch determines what kind of event was sent and raises one of six exceptions, passing to the exception any parameters or results contained in the event message. The six exceptions are: IOCompletion, AsyncData, Attention, Distress, Acknowledge, and ServerFull. These exceptions are explained later in this section.

Parameters:

IOMsg Pointer to the IO event message you obtained from the server. These Emergency messages are obtained by polling with a Receive

on the UserEventPort (set by the OpenIO routine) or by using Emergency message interrupts to signal their arrival and then performing the Receive

Returns:

True if the IOMsg given it is indeed an IO event message, false otherwise.

Exceptions:

```
Exception IOCompletion(
     TransactionID : Long;
     DataBuf       : Pointer;
     DataBuf_Cnt   : Long;
     Status        : IOStatusBlk
    );
```

IOCompletion is sent by the server when the given IO request has completed normally. Return data and status is provided in the parameters.

```
Exception Acknowledge( TransactionID : Long );
```

Acknowledge is sent to confirm the completion of a command. Unlike IOCompletion, Acknowledge does not return any data or status information. The Acknowledge event is used to confirm only the IOAbort, IOSuspend, and IOResume commands.

```
Exception ServerFull( TransactionID : Long );
```

ServerFull is sent if an AsyncIO request would exceed the permitted number of queued requests (ten for each device). The request is rejected.

```
Exception Distress(
     TransactionID : Long;
     DataBuf       : Pointer;
     DataBuf_Cnt   : Long;
     Status        : IOStatusBlk
    );
```

Distress is sent for all other errors that occur in handling an AsyncIO request. When Distress is sent, the server responds by suspending the processing of queued IO requests until the user responds by sending either an IOAbort or an IOResume command to the server. The server will subsequently send an Acknowledge event to confirm that it has carried out the IOAbort or IOResume.

```
Exception AsyncData(
     SequenceNumber : Long;
     DataBuf        : Pointer;
     DataBuf_Cnt    : Long;
     Status         : IOStatusBlk
    );
```

AsyncData is sent whenever data arrives asynchronously from a device and when the server has been explicitly directed to supply this data (by the IOSetStream command described in Section 2.7). If there is an IORead request in the server's queue, AsyncData events are delayed until after the queued IORead has been processed.

Exception Parameters:

SequenceNumber
: Number giving the chronological order of the AsyncData event message. Because AsynchData messages are emergency messages, they can become out of order; therefore SequenceNumber identifies their order. When the device is opened, the server initializes the first SequenceNumber to 1.

TransactionID
: The CmdIDTag field of the CmdBlk of the corresponding AsyncIO call

DataBuf
: Pointer to any data supplied in the event

DataBuf_Cnt
: Number of bytes pointed to by DataBuf

Status
: Completion status of the operation. Included is the DeviceStatus, which is device dependent and may hold the values of device status registers when they were last obtained from the device. The current DeviceStatus values are explicitly obtained when the IOSense command is issued

INDEX

KEY TO ABBREVIATIONS - PROGRAMMING MANUAL

Volume 1

TH - Theory of Operations
KE - Kernel Interface
FS - File System

Volume 2

PR - Process Manager
WM - Window Manager
EN - Environment Manager
NT - Network Server
NM - Name Server
TM - Time Server
TY - Typescript Manager
IO - IO System