



THE KERNEL INTERFACE

December 7, 1984

Copyright © 1984 PERQ Systems Corporation
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 system described in this manual is based upon the Kernel program by Richard F. Rashid.

This document is adapted from the paper by Richard F. Rashid, *Accent Kernel Reference Manual*. Carnegie-Mellon University, Pittsburgh, PA, 1984.

This document is not to be reproduced in any form or transmitted in whole or in part without the prior written authorization of PERQ Systems Corporation.

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

<u>1. Theory</u>	<u>KE-1</u>
<u>2. Use</u>	<u>KE-5</u>
<u>3. Accent Message Primitives</u>	<u>KE-7</u>
3.1. Overview	KE-7
3.2. Accent Primitives	KE-18
3.2.1. Sending a message from current process to a remote port	KE-18
3.2.2. Setting ports on which to receive messages (obsolete)	KE-20
3.2.3. Receiving next message from a port queue	KE-21
3.2.4. Receiving next message (operating system use only)	KE-23
3.2.5. Locking or unlocking ports	KE-24
3.2.6. Returning ports with messages waiting	KE-25
3.2.7. Returning ports with messages waiting (obsolete)	KE-26
3.2.8. Moving data in address space	KE-27
3.2.9. Resetting the enabling of software interrupts	KE-28

3.3. IPC-Related Non-Primitive Functions	KE-29
3.3.1. Allocating a new port	KE-29
3.3.2. Changing maximum queue length	KE-30
3.3.3. Deallocating access to a port	KE-31
3.3.4. Interposing an index (obsolete)	KE-32
3.3.5. Interposing a port (obsolete)	KE-32
3.3.6. Getting the index status (operating system use only)	KE-32
3.3.7. Getting the status (operating system use only)	KE-33
3.3.8. Extracting rights	KE-33
3.3.9. Inserting rights	KE-34
 4. Accent Process Primitives	 KE-35
4.1. Overview	KE-35
4.2. Process Management Routines	KE-36
4.2.1. Forking a process	KE-36
4.2.2. Creating a process	KE-37
4.2.3. Terminating a process	KE-38
4.2.4. Associating a debugger with a process	KE-39
4.2.5. Returning current process status	KE-39
4.2.6. Setting the priority of a process	KE-40
4.2.7. Setting a limit on computation time	KE-41
4.2.8. Suspending a process	KE-41
4.2.9. Resuming a process	KE-42
4.2.10. Examining the microstate of a process	KE-42
4.2.11. Depositing the microstate of a process	KE-43

4.2.12. Enabling software interrupts for messages	KE-44
4.2.13. Getting the SleepID	KE-45
<hr/>	
5. Accent Virtual Memory Primitives	KE-47
 5.1. Overview	KE-47
 5.2. Virtual Memory Management Routines	KE-48
5.2.1. Creating a segment	KE-48
5.2.2. Truncating a segment	KE-50
5.2.3. Destroying a segment	KE-50
5.2.4. Reading selected pages of a segment	KE-51
5.2.5. Overwriting selected pages of a segment	KE-52
5.2.6. Intercepting all calls to the segment system	KE-54
5.2.7. Specifying a segment as disk backup	KE-55
5.2.8. Determining available storage for virtual memory	KE-55
5.2.9. Validating a given part of address space	KE-56
5.2.10. Invalidating a given part of address space	KE-57
5.2.11. Setting protection of valid pages	KE-58
5.2.12. Reading another process's virtual memory	KE-59
5.2.13. Writing to another process's virtual memory	KE-60
5.2.14. Determining if an address is valid	KE-60
<hr/>	
6. Accent Disk Handling	KE-63

6.1. Overview	KE-63
6.2. Disk Management Routines	KE-64
6.2.1. Getting information about partition structure	KE-64
6.2.2. Making a partition accessible	KE-65
6.2.3. Making a partition inaccessible	KE-66
6.2.4. Performing disk I/O while by-passing paging	KE-67
 7. Accent Display Options	 KE-69
7.1. Overview	KE-69
7.2. Display Management Routines	KE-69
7.2.1. Creating a rectangle for a given port	KE-70
7.2.2. Destroying a rectangle for a given port	KE-71
7.2.3. Enabling a list of rectangles	KE-71
7.2.4. Setting-up a window for kernel typescript	KE-72
7.2.5. General RasterOp between rectangles	KE-73
7.2.6. Drawing a line in a rectangle	KE-74
7.2.7. Drawing a character string in a rectangle	KE-75
7.2.8. Coloring a rectangle	KE-77
7.2.9. Scrolling a rectangle	KE-78
7.2.10. Returning a rectangle's parameters	KE-79
 8. Summary of Calls	 KE-81

1. Theory

Accent is a communication-oriented, kernel-based operating system. It provides:

- multiple processes with a flexible process scheduling facility,
- a large, paged virtual memory for each user process,
- both synchronous and asynchronous message-based interprocess communication,
- transparent network extensibility, and
- a flexible capability-based approach to security and protection.

Features of the Accent operating system include:

- support for multiple virtual machines, each capable of using its own micro-interpreter and each with up to 2^{32} bytes of paged virtual memory,
- transparently pageable microcode overlays for special purpose operations, and,
- protected access to rectangular areas of the PERQ workstation 1024x768 or 1024x1280 bitmap displays, including graphics rasterop capability and special string display functions.

The basic unit of computation in Accent is the process. Processes consist of an address space (2^{32} bytes of paged virtual memory), process state (including the state of any macrocode and

microcode registers essential to execution) and access to one or more ports.

An Accent port is a kernel-managed and protected queue of messages. A port is both a single-direction communication channel and the basic object reference mechanism in Accent. Ports are used to refer to objects, and operations on objects are requested by sending messages to the ports that represent them.

An Accent message is a discrete typed collection of data objects and may include access rights to ports.

Message-passing is the sole means of communication both between processes and between the processes and the operating system kernel. The only primitive functions are those directly concerned with message communication. These primitive functions are implemented as traps into the kernel.

This document describes both the basic kernel message primitives provided by Accent and the functions made available by the Accent kernel through messages. Facilities provided by Accent processes (such as the file system) are described in other documents in this manual.

This document is organized as follows:

Chapter 2, Use, describes how to convert a Pascal specification to either a message exchange or a kernel trap.

Chapter 3, Accent Message Primitives, describes the message primitives and support facilities.

Chapter 4, Accent Process Primitives, details process creation and management facilities.

Chapter 5, Accent Virtual Memory Primitives, explains virtual memory management functions.

Chapter 6, Accent Disk Handling, lists the disk handling functions.

Chapter 7, Accent Display Options, details the function calls for updating Accent screen displays.

Chapter 8, Summary of Calls, lists the Pascal calls to Accent.

For clarity, all message primitives and Accent kernel functions provided by messages are described as calls to a PERQ Pascal library of kernel interface procedures and functions. All Pascal code is the PERQ extended language, not standard Pascal.

C and Lisp libraries are provided which have a similar form and make available the same set of functions.

For more information on using Pascal and C on the PERQ workstation, see the *Accent Languages Manual*. For more information on using Lisp, see the *Accent Lisp Manual*.

2. Use

The rule of thumb for converting the Pascal specification found in this document to either a message exchange or kernel trap is as follows:

- In the case of kernel primitives, the arguments to the function are packaged into a record and a pointer to the record is placed onto the PERQ workstation expression stack to be picked up and interpreted by the operating system after a switch to kernel state occurs. The passed record contains slots for return values. Return values are stored by the operating system before returning to the user process.
- In the case of message calls, the input arguments to the function are composed into a message and sent to the port described in the first argument of the function. The arguments are placed in the message in the order that they appear, each described by a different descriptor (see below). Reply messages contain a sixteen bit integer result code and all output arguments and are sent to the port which was used as the local port field of the message as sent to the kernel. In the procedural specifications below, the reply port field is left out of the call for clarity. In the Accent Pascal interface, the reply port is allocated by an initialization routine and implicit in all calls.

In addition, Pascal records are used as a descriptive language for the contents of the messages. In interpreting these records as bits of information laid out in memory, the following packing rules must be applied:

- In records which are not labeled 'packed', all quantities which are 16 bits or less in size are packed in 16 bits.
- In packed records, several quantities may be packed into a 16 bit word if the sum of the bit sizes is less than or equal to 16.
- The bit size of a quantity is the minimum number of bits required to represent that quantity. Thus a boolean has bit size one. A character has bit size eight. An integer is 16 bits. A long is 32 bits.
- Information is packed into words from right to left (least significant to most significant bit). No quantity may straddle a 16 bit boundary with the exception of a long which is 32 bits.
- Identifiers which are defined as part of enumerated types start with the value zero and increase by one. Thus <type foo = (bar, baz)> means that <bar> will have the value zero and <baz> the value one.

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 becomes available under Accent, the definitions will be the same as in the C language.

3. Accent Message Primitives

Accent message primitives manipulate two distinct objects:

1. Ports - protected kernel objects to which messages may be sent and logically queued until reception, and
2. Messages - ordered collections of typed data consisting of a fixed size message header and a variable size message body.

3.1. Overview

Ports have finite queues and processes may exercise several options for handling the case of message transmission to a full queue (refer to the Send function in Section 3.2.1).

Processes refer to ports through the use of 32-bit numeric values that represent access rights to send to, receive from, or own ports. A process can hold any combination of these rights. When a process creates a port it holds all three. As long as a process retains access to a port it may use the same numeric value to refer to it. The numeric value representing the same port may be different for different processes. A process can only refer to ports to which it has been given access.

The three different kinds of port access rights are defined as follows:

Send access to a port implies that a process can send a message to that port. Should the port be destroyed during the time a process has send access, a message will be sent to that process by the kernel indicating that the port has disappeared as well as the cause

of the disappearance (e.g. explicit destruction, process death, network failure, etc.). Any process that knows about a port has send rights to it.

Receive access to a port allows a process to receive a message from that port. Only one process may have receive access for a given port at a time.

Ownership of a port implies that, should the process with receive access relinquish the port either through explicit deallocation or process death, the receive rights to the port will be sent to the port's owner. Likewise, should ownership be relinquished, the ownership rights are sent by the kernel in a message to the receiving process.

Port access rights can be passed in messages. They are interpreted by the kernel and transferred from the sender to the kernel upon message transmission and to the receiver upon message receipt.

At the time of creation, each process normally has access to at least two ports:

A kernel port for which the new process has send access and which can be used to send messages to the system kernel, and

A data port for which the new process has receive access and ownership and for which the kernel has send access. The data port is the port normally used by the kernel to send emergency messages about port destruction and other error conditions.

A message consists of a fixed part, describable as a Pascal record, followed by a variable size part. The Pascal record (from the definitions file AccentType.Pas) describing the fixed size message head is as follows:

```

type
  Msg =
    record
      SimpleMsg : boolean; { True if message uses
                            { no pointers and
                            { transmits no port
                            { rights
      MsgSize     : long;   { Number of contiguous
                            { bytes in the message}
      MsgType     : long;   { NORMALMSG or
                            { EMERGENCYMSG
      LocalPort   : Port;  { For send: optional
                            { port of sender
      RemotePort  : Port;  { For receive: port msg
                            { received on
      ID          : long;  { For send: port to
                            { send message to
      { For receive: optional
      { port of sender
      ID          : long;  { Arbitrary value - by
                            { convention used
                            { to identify kind
                            { of message.
      { MsgSize = ByteSize(Msg) bytes of message }
      { data follow, see below. }
    end;
  
```

15	0
SimpleMsg	word 0
MsgSize (lsw)	word 1
MsgSize (msw)	word 2
MsgType (lsw)	word 3
MsgType (msw)	word 4
LocalPort (lsw)	word 5
LocalPort (msw)	word 6
RemotePort (lsw)	word 7
RemotePort (msw)	word 8
ID (lsw)	word 9
ID (msw)	word 10

The variable data part of a message consists of an array of descriptors for typed objects. Each typed object descriptor is of the form:

		31	0
		TypeType	
(optional)	TypeName	TypeSizeInBits	
(optional)	NumElts		
		Pointer to Data (32 bits) or Inline Data (TypeSizeInBits * NumElts bytes)	

The record describing the TypeType field is:

type

```
TypeType =  
  packed record  
    TypeName      : Bit8;  
    TypeSizeInBits : Bit8;  
    NumObjects    : Bit12;  
    InLine        : boolean;  
    LongForm      : boolean;  
    Deallocate    : boolean;  
  end;
```

31	30	29	28	27	16	15	8	7	0

U	D	L	I	NUM-	TYPESIZE-	TYPENAME
N	E	O	N	OBJECTS	INBITS	
U	A	N	L			
S	L	G	I			
E	L	F	N			
D	O	O	E			
C	R					
A	M					
T						
E						

The TypeName, TypeSizeInBits, and NumElt fields exist only if the LongForm bit of the DataType field is set. This provides potentially 16 bits for both the TypeName and the TypeSizeInBits and allows a single object to be an array of up to $2^{32}-1$ primitive data elements. If LongForm is false, then the number of elements is defined by the NumObjects field of DataType.

System defined data types (i.e., values which may be included in the TypeName field of a DataDescriptor) include:

TYPEUNSTRUCTURED	=	0;
TYPEBIT	=	0;
TYPEBOOLEAN	=	0;
TYPEINT16	=	1;
TYPEINT32	=	2;
TYPEPTOWNSERSHIP	=	3;
TYPEPTRECEIVE	=	4;
TYPEPTALL	=	5;
TYPEPT	=	6;
TYPECHAR	=	8;
TYPEINT8	=	9;
TYPEBYTE	=	9;
TYPEREAL	=	10;
TYPEPSTAT	=	11;
TYPESTRING	=	12;

TYPESEGID = 13;

TYPEPAGE = 14;

The Deallocate bit applies to port rights and data blocks pointed to by a descriptor. If the Deallocate bit is set and a port is referred to, that port is deallocated from the name space of the current process when the message is sent. If the Deallocate bit is set and a pointer is referred to, all full pages of data pointed to by the pointer will be removed from the address space of the sending process when the message is sent.

The following is an example of the creation of a message in Pascal. The message contains both pointer and inline data blocks as well as a port and a long integer. Note that in practice users do not normally construct messages by hand, but use the Accent remote procedure call generator to define type-checked message interfaces between processes. (For more information see the document "Matchmaker: The Accent Remote Procedure Call Language" in the *Accent Languages Manual*.)

```
const
  MYMESSAGEID = 100;

type
  ptrBlock  = ^Block;
  Block     = array [0..255] of integer;

MyMessage =
  record
    Header          : Msg;
    APortType       : TypeType;
    APort          : Port;
    ALongType       : TypeType;
    ALong          : long;
    BlockPtrType   : TypeType;
    BlockPtr       : ptrBlock;
    InlineBlockType: TypeType;
    InlineBlockTypeName: Integer;
    InlineBlockTypeSize: Integer;
    InlineBlockNumObjects: Long;
    InlineBlock    : Block;
  end;
```

```

var
  message      : MyMessage;
  MyPort       : Port;
  HisPort      : Port;
  MyOtherPort  : Port;
  MyLong        : long;
  PtrToSomeBlock: ptrBlock;

begin
  { Get information to send from somewhere. }

  message.Header.SimpleMsg          := false;
  message.Header.MsgSize            := WordSize (message)*2;
  message.Header.MsgType            := NORMALMSG;
  message.Header.LocalPort          := MyPort;
  message.Header.RemotePort         := HisPort;
  message.Header.ID                 := MYMESSAGEID;

  message.APortType.TypeName        := TYPEPT;
  message.APortType.NumObjects      := 1;
  message.APortType.InLine          := true;
  message.APortType.LongForm         := false;
  message.APortType.Deallocate       := false;
  message.APortType.TypeSizeInBits  := 32;
  message.APort                      := MyOtherPort;

  message.ALongType.TypeName        := TYPELONG;
  message.ALongType.NumObjects      := 1;
  message.ALongType.InLine          := true;
  message.ALongType.LongForm         := false;
  message.ALongType.Deallocate       := false;
  message.ALongType.TypeSizeInBits  := 32;
  message. ALONG                     := MyLong;

  message.BlockPtrType.TypeName     := TYPEINT16;
  message.BlockPtrType.NumObjects   := 256;
  message.BlockPtrType.InLine        := false;
  message.BlockPtrType.LongForm      := false;
  message.BlockPtrType.Deallocate    := false;
  message.BlockPtrType.TypeSizeInBits := 16;
  message.BlockPtr                  := PtrToSomeBlock;

  { Now an example using the long form of TypeType }

  message.InlineBlockType.InLine     := true;
  message.InlineBlockType.LongForm    := true;
  message.InlineBlockType.Deallocate  := false;
  message.InlineBlockType.Name        := TYPEINT16;
  message.InlineBlockType.Size        := 16;
  message.InlineBlockNumObjects      := 256;
  message.InlineBlock                := PtrToSomeBlock`;

  { Send the message }

end;

```

In addition to the message record definitions, the following constant and type definitions are used by the various message interface routines:

```
const
    WAIT          = 0;
    DONTWAIT     = 1;
    REPLY         = 2;

    PREVIEW       = 0;
    RECEIVEIT     = 1;
    RECEIVEWAIT   = 2;

    DEFAULTPTS    = 0;
    ALLPTS        = 1;
    LOCALPT       = 2;

    DEFAULTBACKLOG = 0; { Installation dependent }
    MAXBACKLOG    = 63; { Installation dependent }

    {}

    { Message types:
    {}
    NORMALMSG     = 0;
    EMERGENCYMSG  = 1;

    {}

    { Kernel generated message ids:
    {}
    M_PORTDELETED      = #100 + 1;
    M_MSGACCEPTED      = #100 + 2;
    M_OWNERSHIPRIGHTS  = #100 + 3;
    M_RECEIVERIGHTS    = #100 + 4;
    M_GENERALKERNELREPLY = #100 + 6;
    M_KERNELMSGERROR   = #100 + 7;
    M_PARENTFORKREPLY  = #100 + #10;
    M_CHILDFORKREPLY   = #100 + #11;
    M_DEBUGMSG         = #100 + #12;

type
    Port           = long;
    Milliseconds   = long;
    SendOption     = WAIT..REPLY;
    ReceiveOption  = PREVIEW..RECEIVEWAIT;
    PortOption     = DEFAULTPTS..LOCALPT;
    LPortArray     = array [stretch(0)..stretch(#777777)] of Port;
    PtrLPortArray  = ^LPortArray;
    BacklogValue   = 0..MAXBACKLOG;
```

In addition, the possible return values for all of the routines are

found in file AccentType.Pas, and listed below. For kernel primitives, these values are returned in the kernel trap argument block. For messages these values are returned as the first element in the data portion of the message as a 16 bit integer. The user sees both these types of return values as general return values.

```
const
  AccErr          = 100;
  Dummy          = AccErr+0;
  Success        = AccErr+1;
  TimeOut        = AccErr+2;
  PortFull       = AccErr+3;
  WillReply      = AccErr+4;
  TooManyReplies = AccErr+5;
  MemFault       = AccErr+6;
  NotAPort       = AccErr+7;
  BadRights      = AccErr+8;
  NoMorePorts   = AccErr+9;
  IllegalBacklog = AccErr+10;
  NetFail        = AccErr+11;
  Intr           = AccErr+12;
  Other          = AccErr+13;
  NotPortReceiver = AccErr+14;
  UnrecognizedMsgType = AccErr+15;
  NotEnoughRoom  = AccErr+16;
  NotAnIPCCall   = AccErr+17;
  BadMsgType     = AccErr+18;
  BadIPCName    = AccErr+19;
  MsgTooBig      = AccErr+20;
  NotYourChild   = AccErr+21;
  BadMsg          = AccErr+22;
  OutOfIPCSpace  = AccErr+23;
  Failure         = AccErr+24;
  MapFull         = AccErr+25;
  WriteFault     = AccErr+26;
  BadKernelMsg   = AccErr+27;
  NotCurrentProcess = AccErr+28;
  CantFork        = AccErr+29;
  BadPriority     = AccErr+30;
  BadTrap          = AccErr+31;
  DiskErr          = AccErr+32;
  BadSegType      = AccErr+33;
  BadSegment       = AccErr+34;
  IsParent         = AccErr+35;
  IsChild          = AccErr+36;
  NoAvailablePages = AccErr+37;
  FiveDeep         = AccErr+38;
  BadVPTable      = AccErr+39;
  VPExclusionFailure = AccErr+40;
  MicroFailure    = AccErr+41;
  EStackTooDeep   = AccErr+42;
  MsgInterrupt    = AccErr+43;
```

UncaughtException	= AccErr+44;
BreakPointTrap	= AccErr+45;
ASTInconsistency	= AccErr+46;
InactiveSegment	= AccErr+47;
SegmentAlreadyExists	= AccErr+48;
OutOfImageSegments	= AccErr+49;
NotASystemAddress	= AccErr+50;
NotAUserAddress	= AccErr+51;
BadCreateMask	= AccErr+52;
BadRectangle	= AccErr+53;
OutOfRangeBounds	= AccErr+54;
IllegalScanWidth	= AccErr+55;
CoveredRectangle	= AccErr+56;
BusyRectangle	= AccErr+57;
NotAFont	= AccErr+58;
PartitionFull	= AccErr+59;

3.2. Accent Primitives

The primitives described in this section can be found in the module AccCall in the file AccCall.Pas.

3.2.1. Sending a message from current process to a remote port

```
function Send(
  var MsgHdr : Msg;
  MaxWait   : long;
  Option    : SendOption
) :GeneralReturn;
```

Abstract

Send transmits a message from the current process to the RemotePort specified in the message header.

Arguments

- MsgHdr The header portion of the message to be sent.
MaxWait The maximum time in milliseconds to wait if the destination port is full. A wait time of zero implies wait forever.
Option A constant specifying whether the send operation should WAIT, DONTWAIT or REPLY if the destination port is full at the time of

the send. The options are as follows:

• **WAIT**

should be used when the sending process wishes to be suspended for MaxWait milliseconds if the queue is full. If by that time the port is still full, the call returns without having sent the message.

• **DONTWAIT**

should be used if the sending process does not wish to wait for any length of time in the case of a full destination port.

• **REPLY**

allows the sender to give exactly one message to the operating system without being suspended if the destination port is full. When another message can be posted to the destination port's queue, a message is sent to the data port of the sending process notifying it that another message can be sent. A second message, sent to a full port before this notification arrives, results in an error.

Returns

Success	The message has been queued for the RemotePort specified in MsgHdr.
Timeout	The message was not sent since the destination port was still full after MaxWait milliseconds.
Failure	The message was not sent because the destination port was full and DONTWAIT was specified.
WillReply	The destination port was full but the REPLY option was specified. An emergency message will be sent when the message can be posted.
TooManyReplies	The REPLY option was specified but a reply request is already outstanding for this process and destination port.
MemFault	The message pointed to data not in the address space of the sender.
NotAPort	The message refers to a port number which is not available to the current process.
BadRights	The message contains a reference to port access rights not possessed by the current process.

3.2.2. Setting ports on which to receive messages (obsolete)

```
function SetPortsWaiting(  
var Ports : PortBitArray  
) : GeneralReturn;
```

NOTE

SetPortsWaiting is still in the system but has been made obsolete by the removal of limitations on the maximum number of ports accessible to a process. This call will succeed only if the total

number of ports in the current Accent system is less than the old per process maximum of 255. The call LockPorts replaces SetPortsWaiting.

SetPortsWaiting passes a boolean array to the operating system kernel specifying the ports on which the current process wishes to receive messages. Each port is represented as a bit in the array and a value of true implies that, should a Receive call be made, a message can be accepted from that port. A value of false implies that even if a message is waiting on that port at the time of the Receive call, it should not be received.

3.2.3. Receiving next message from a port queue

```
function Receive(
var MsgHdr : Msg; { inout parameter }
  MaxWait : long;
  PortOpt : PortOption;
  Option   : ReceiveOption
) :GeneralReturn;
```

Abstract

Receive retrieves the next message from a port queue on which the current process is waiting.

Arguments

MsgHdr The header part of a message data structure into which a message can be received.

MaxWait The maximum time in milliseconds to wait for a message before giving up. A wait time of zero implies an infinite wait. A wait time of -1 implies no wait.

PortOpt A constant specifying what ports the receive should be done on.

- **DEFAULTPTS**

the ports that have been previously set by SetPortsWaiting and LockPorts should be used.

- **ALLPTS**

look for a message on all the ports to which this process has Receive access.

- **LOCALPT**

receive a message only on the port specified by `MsgHdr.LocalPort`.

Option A constant specifying whether the receive operation should be `PREVIEW`, `RECEIVEWAIT` or `RECEIVEIT`

- **PREVIEW**

allows the header of a message to be looked at before receiving the data. Once previewed, a message has in fact been received, but the operating system will hold the data until a subsequent Receive with the `RECEIVEIT` option is performed. No other messages can be received until the previewed message has been received.

- **RECEIVEWAIT**

simply waits until a message can be received. It does not receive the message or modify the port queues in any way.

- **RECEIVEIT**

is the option actually used for receiving both the header and data part of a message. Once received, the message is removed from its port queue and all port capabilities contained in the message are passed to the receiving process.

Returns

Success The message has been received.

Timeout The message was not received after `MaxWait` milliseconds.

MemFault The message pointed to a data area inaccessible to the receiver.

3.2.4. Receiving next message (operating system use only)

```
function EReceive(
    var xxmsg      : Msg;
    MaxWait       : long;
    PortOpt       : PortOption;
    Option        : ReceiveOption
) :GeneralReturn;
```

Abstract

EReceive is a version of Receive that is used by the operating system. It is not to be used by other clients of the kernel.

Arguments

xxmsg The header part of a message data structure into which a message can be received.

MaxWait The maximum time (in milliseconds) to wait for a message before giving up. A wait time of zero implies an infinite wait. A wait time of -1 implies no wait.

PortOpt A constant specifying what ports the receive should be done on.

- DEFAULTPTS

The ports that have been previously set by SetPortsWaiting should be used.

- ALLPTS

Look for a message on all the ports to which this process has EReceive access.

- LOCALPT

Receive a message only on the port specified by MsgHdr.LocalPort.

Option A constant specifying whether the receive operation should be PREVIEW, RECEIVEWAIT or RECEIVEIT.

- PREVIEW

Allows the header of a message to be looked at before receiving the data. Once previewed, a message has in fact been received, but the operating system will hold the data until a subsequent EReceive with the RECEIVEIT option is performed. No other message can be received until the previewed message has been

received.

• **RECEIVEWAIT**

Simply waits until a message can be received. It does not receive the message or modify the port queues in any way.

• **RECEIVEIT**

This is the option actually used for receiving both the header and data part of a message. Once received, the message is removed from its port queue and all port capabilities contained in the message are passed to the receiving process.

Returns

Success The message has been received.

Timeout The message was not received after MaxWait milliseconds.

MemFault The message pointed to a data area inaccessible to the receiver.

3.2.5. Locking or unlocking ports

```
function LockPorts(
    LockOrUnlock : boolean;
    Ports        : PtrLPortArray;
    PortsCount   : long
) : GeneralReturn;
```

Abstract

LockPorts either locks (if LockOrUnlock is true) or unlocks the ports pointed to by the array Ports. Messages may not be received from a locked port and no interrupts occur if messages arrive at a locked port. A port may only be locked by the process with receive access.

Arguments

LockOrUnlock

True if we are locking, false otherwise.

Ports A port array used to specify the ports to be locked.

PortsCount

The number of ports to be locked.

Returns

Success Ports are locked or unlocked.

MemFault Illegal data area specified for returned data.

Failure Illegal request to lock or unlock a port.

3.2.6. Returning ports with messages waiting

```
function MessagesWaiting(
    MsgType      : long;
    var Ports     : PtrLPortArray;
    var PortsCount : long
        { inout parameter }
) : GeneralReturn;
```

Abstract

MessagesWaiting returns an array of ports on which there is a message waiting whose type matches MsgType.

Arguments

MsgType Message type field of messages of interest.

Ports A port array used to return ports with messages waiting.

PortsCount

The maximum number of ports to return. On return, the actual number of ports returned.

Returns

Success Ports are returned.

NotEnoughRoom

Number of ports is greater than the maximum given by PortsCount.

3.2.7. Returning ports with messages waiting (obsolete)

```
function PortsWithMessages(
  MsgType : long;
var Ports   : PortBitArray
) : GeneralReturn;
```

NOTE

PortsWithMessages is still in the system but has been made obsolete by the removal of limitations on the maximum number of ports accessible to a process. This call will succeed only if the total number of ports in the current Accent system is less than the old per process maximum of 255. The call MessagesWaiting replaces PortsWithMessages.

PortsWithMessages returns a boolean array in the var parameter Ports which has a true value for every port containing at least one message of type MsgType.

3.2.8. Moving data in address space

```
function MoveWords(
    SrcAddr      : VirtualAddress;
var DstAddr      : VirtualAddress;  { inout }
    NumWords     : long;
    Delete       : boolean;
    Create       : boolean;
    Mask         : long;
    DontShare    : boolean
) :GeneralReturn;
```

Abstract

MoveWords moves data from one place in the current process's address space to another. It differs from a macroarchitecture move instruction in that it can be used to remap areas of memory in addition to moving data in the conventional sense. Note, however, that source and destination areas cannot overlap.

Arguments

SrcAddr The source address.

DstAddr The destination address. If Create (see below) is true then this argument is ignored on input and supplied by the system on

output.

NumWords

The number of words to move.

- Delete** If true, the source words will be deleted from the current process's address space. Memory is only deleted in full pages. The pages deleted start with the page containing SrcAddr and end with the page containing the address SrcAddr + NumWords - 1.
- Create** If true, new space will be allocated for the copy and a pointer returned in DstAddr.
- Mask** A 32-bit mask which, if Create is true, determines where in memory allocated space can be used (see the Section 5.2.9 "ValidateMemory").
- DontShare** If true, implies that data will be copied copy-on-write. If false it implies that the data references in the destination area will be synonymous with data references in the source area. It is an error if DontShare is false and the source and destination address have different word offsets in a page.

Returns

- Success** The copy was performed.
- MemFault** The arguments are illegal.

3.2.9. Resetting the enabling of software interrupts

```
function SoftEnable(
    NormOrEmerg : boolean;
    EnOrDis      : boolean
) :GeneralReturn;
```

Abstract

SoftEnable resets the enabling of software interrupts. When a software interrupt occurs software interrupts are disabled by the system. This routine is intended to be called by software interrupt handlers just before they exit in

order to allow the handling of further software interrupts.

Arguments

NormOrEmerg

True if software interrupts are enabled on normal messages, false if software interrupts are enabled on emergency messages.

EnOrDis True if interrupts are enabled; false if interrupts are disabled.

Returns

Success This is the only possible return.

3.3. IPC-Related Non-Primitive Functions

The primitives described in this section can be found in the module AccInt in the file AccentUser.

3.3.1. Allocating a new port

```
function AllocatePort(
    KernelPort      : Port;
var LocalPort      : Port; { out }
    Backlog        : BacklogValue
) :GeneralReturn;
```

Abstract

AllocatePort requests that a new port be allocated with an initial maximum queue size (backlog) of Backlog. If no port can be returned, LocalPort is left with a value of NULLPORT.

Arguments

KernelPort The kernel port of a process. The port is created in the process designated by KernelPort.

LocalPort A var parameter into which the allocated port is returned.

Backlog The maximum queue length (number of messages) for the newly allocated port. A value of zero will set the backlog to

DEFAULTBACKLOG. The maximum backlog value is MAXBACKLOG.

Returns

Success A port has been allocated.

NoMorePorts

No more port slots available for this process.

MemFault The arguments are illegal.

3.3.2. Changing maximum queue length

```
function SetBacklog(
    KernelPort      : Port;
    LocalPort       : Port;
    Backlog         : BacklogValue
) :GeneralReturn;
```

Abstract

SetBacklog requests that the maximum queue length for the specified port be changed to Backlog.

Arguments

KernelPort The kernel port of the process whose port space the current process wishes to alter.

LocalPort The port queue to be changed.

Backlog The new backlog. Should the new backlog be less than the current queue length, no messages are released or destroyed, but future send requests will respect the new backlog. A value of zero will set the backlog to DEFAULTBACKLOG. The maximum backlog value is MAXBACKLOG.

Returns

Success The backlog has been changed.

IllegalBacklog

The backlog is either too small or too large.

The backlog of a port in another process's port space can be changed, if you have any access to that process's kernel port.

3.3.3. Deallocating access to a port

```
function DeallocatePort(
    KernelPort      : Port;
    LocalPort       : Port;
    Reason          : long
) :GeneralReturn;
```

Abstract

DeallocatePort requests that the current process's access to a port be released. If the current process is both the receiver and owner for the port, then the port is destroyed and all other processes with send access are notified both of the port's destruction and the reason for that destruction.

Arguments

KernelPort The kernel port of the process.

LocalPort The port to be deallocated.

Reason An arbitrary 32 bit number which will be returned to those processes which have access to the port should it be destroyed by this call. There are 3 system defined reasons for port death: EXPLICITDEALLOCATION, PROCESSDEATH, and NETWORKTROUBLE.

Returns

Success The port has been deallocated.

NotAPort The port is not allocated to this process.

3.3.4. Interposing an index (obsolete)

```
function IndexInterpose(
    ServPort      : port;
    MyPort        : Port;
    HisIndex      : Long;
var HisPort     : Port
) :GeneralReturn;
```

NOTE

This function has been made obsolete. It will always return FAILURE.

3.3.5. Interposing a port (obsolete)

```
function PortInterpose(
    ServPort      : port;
    MyPort        : Port;
    HisPort       : port;
var MyNewPort   : Port
) :GeneralReturn;
```

NOTE

This function has been made obsolete. It will always return FAILURE.

3.3.6. Getting the index status (operating system use only)

```
function GetPortIndexStatus(
    ServPort      : port;
    PortIndex     : Long;
var BackLog     : Integer;
var NWaitingMsgs : Integer;
var EWaitingMsgs : Integer;
var PortRight   : port;
var PortType    : Integer
) :GeneralReturn;
```

NOTE

GetPortIndexStatus provides information about a port given its index into the kernel port table. It is intended to be used by system implementors and not by normal clients of the kernel.

3.3.7. Getting the status (operating system use only)

```
function GetPortStatus(
    ServPort      : port;
    PortRight     : port;
var BackLog      : Integer;
var NWaitingMsgs : Integer;
var EWaitingMsgs : Integer;
var PortIndex    : Long;
var PortType     : Integer;
) :GeneralReturn;
```

NOTE

GetPortStatus provides information about a port given its index into the kernel port table. It is intended to be used by system implementors and not by normal clients of the kernel.

3.3.8. Extracting rights

```
function ExtractAllRights(
    ServPort      : port;
    PortIndex    : Long;
var PortRight    : port;
var PortType     : Integer;
) :GeneralReturn;
```

NOTE

This call is not currently implemented.

3.3.9. Inserting rights

```
function InsertAllRights(
    ServPort      : port;
    PortIndex     : Long;
var PortRight   : port;
var PortType    : Integer
) :GeneralReturn;
```

NOTE

This call is not currently implemented.

4. Accent Process Primitives

Accent supports a potentially large number of processes, each with independent paged address spaces.

4.1. Overview

Process scheduling is round-robin within a priority level. Accent supports 16 priority levels, each with an installation dependent time quota. Once a process exhausts its time quota at a given priority level, its priority is decreased, allowing a form of aging. Process priorities are reinstated at their default level following a message send or receive. Processes may be created either by forking or by explicit process creation. Processes may only fork themselves, but may terminate either themselves or other processes that they have created.

The following type definitions are used by the various message interface routines:

```
const
    READONLY      = 0;
    READWRITE     = 1;
    MAXPROCS     = 63;
    NUMPRIORITIES = 16;
    NUMSLEEPQS   = 32;
    NUMQUEUES    = NUMSLEEPQS +
                    NUMPRIORITIES + 5;

type
    Port          = long;
    VirtualAddress= long;
    Microseconds  = long;
    PriorID       = 0..NUMPRIORITIES-1;
    ProcState     = (Supervisor, Privileged,
                    BadSupervisor, User);
    QID           = 0..NUMQUEUES;
```

```
MemProtection = READONLY..READWRITE;
PStatus       = record
    State          : ProcState;
    Priority       : PriorID;
    MsgPending     : boolean;
    EMsgPending   : boolean;
    MsgEnable      : boolean;
    EMsgEnable    : boolean;
    LimitSet       : boolean;
    SVStkInCore   : boolean;
    QueueID        : QID;
    SleepID        : ptrInteger;
    RunTime         : long;
    LimitTime      : long;
end;
```

4.2. Process Management Routines

The routines which follow in this section may be found in the module AccInt in the file AccentUser.Pas.

4.2.1. Forking a process

```
function Fork(
    KernelPort      : Port;
    var HisKernelPort : Port;
                           { out parameter }
    var HisDataPort  : Port;
                           { out parameter }
    Ports           : PtrPortArray;
    PortsCount      : long
) : GeneralReturn;
```

Abstract

Fork creates a running clone of the process whose kernel port is used for the call. The ports owned by the child of the current process created by the Fork are provided by the parent. A process may only Fork itself. HisKernelPort and HisDataPort are created by the kernel and give the parent a handle on the child.

Arguments

KernelPort The kernel port of the process to be copied.

HisKernelPort

On return, the kernel port of the child.

HisDataPortOn return, the data port of the child.

Ports A pointer to an array of port capabilities to send to the child. Only SEND rights to these ports are sent.

PortsCount

The number of ports to be sent to the newly created process.

Returns

IsParent The current process is the parent.

IsChild The current process is the child.

TooManyProcesses

The system can no longer allocate processes.

NotCurrentProcess

Kernel port does not belong to the current process.

NotAPort A port mentioned in the message is not accessible to the parent.

4.2.2. Creating a process

```
function CreateProcess(
    KernelPort      : Port;
    var HisKernelPort : Port; { out parameter }
    var HisDataPort   : Port { out parameter }
) : GeneralReturn;
```

Abstract

CreateProcess creates a new process which has no state. This call can be used to create a process which may then be loaded with memory and the state of the registers through the use of WriteProcessMemory and Deposit (which are described in detail in Sections 5.2.13 and 4.2.11, respectively).

Arguments

KernelPort The kernel port of the parent process.

HisKernelPort

On return, the kernel port of the child.

HisDataPortOn return, the data port of the child.

Returns

Success The call succeeded.

Failure The request could not be performed.

NotCurrentProcess

Kernel port does not belong to the current process.

4.2.3. Terminating a process

```
function Terminate(  
    KernelPort : Port;  
    Reason     : long;  
) : GeneralReturn;
```

Abstract

Terminate destroys the process associated with KernelPort. If that process is not the current process, the call returns.

Arguments

KernelPort The kernel port of the process to be killed.

Reason An arbitrary 32 bit number that will be returned to those processes which have access to ports affected by this call.

Returns

Success The process has been killed.

4.2.4. Associating a debugger with a process

```
function SetDebugPort(
    KernelPort    : Port;
    DebugPort     : Port
) :GeneralReturn;
```

Abstract

Associate a debugger (defined by DebugPort) with a process (identified by KernelPort, its kernel port) in such a way that if that process should fail, the process will be suspended in its failed state and a message will be sent to DebugPort. The message sent by the kernel will be an Emergency Message and will contain as its data a port (SendersKernelPort), a 32-bit numeric value (always 0), and an error code (a GeneralReturn value) as a 32 bit numeric value.

Arguments

KernelPort Kernel port of process to be debugged.

DebugPort Port to which error messages should be sent when the process fails.

Returns

Success DebugPort has been associated with the process.

Failure KernelPort was not a kernel port for an active process.

4.2.5. Returning current process status

```
function Status(
    KernelPort    : Port;
var Status       : PStatus
) :GeneralReturn;
```

Abstract

Status returns the current process status of the process associated with KernelPort.

Arguments

KernelPort The kernel port of the process to be queried.

Status A var parameter used to return the process's state.

Returns

Success The data has been retrieved.

MemFault Illegal address reference.

4.2.6. Setting the priority of a process

```
function SetPriority(
    KernelPort    : Port;
    Priority      : PriorID
) : GeneralReturn;
```

Abstract

SetPriority is used to change the priority of the process associated with **KernelPort**. The highest priority is 15, the lowest is 0.

Arguments

KernelPort The kernel port of the process whose priority is to be changed.

Priority The new priority.

Returns

Success Priority changed.

BadPriority Illegal priority value. A process may only decrease its own priority.

4.2.7. Setting a limit on computation time

```
function SetLimit(
    KernelPort    : Port;
    ReplyPort     : Port;
    Limit         : long
) :GeneralReturn;
```

NOTE

The SetLimit function has not been implemented yet.

SetLimit sets a limit on the computation time available to the process associated with KernelPort. When the time limit has been exceeded, a message will be sent to ReplyPort indicating that the process has used its allotted computational resources.

4.2.8. Suspending a process

```
function Suspend(
    KernelPort    : Port;
) :GeneralReturn;
```

Abstract

Suspend suspends the process associated with KernelPort. If a process suspends itself, it may be awakened by the receipt of a message (as well as a *Resume*). If a process suspends another process, the suspended process may only be awakened by a call to *Resume*.

Arguments

KernelPort The kernel port of the process to be suspended.

Returns

Success Process suspended.

4.2.9. Resuming a process

```
function Resume(
    KernelPort : Port;
) : GeneralReturn;
```

Abstract

Resume resumes the process associated with KernelPort.

Arguments

KernelPort The kernel port of the process to be resumed.

Returns

Success Process resumed.

4.2.10. Examining the microstate of a process

```
function Examine(
    KernelPort : Port;
    RegOrStack : boolean;
    Index      : integer;
    var Value   : integer
) : GeneralReturn;
```

Abstract

Examine looks at the microstate of the process associated with KernelPort. If the process examined is not suspended, the values returned may not be accurate.

Arguments

KernelPort The kernel port of the process to be examined.

RegOrStack True if the microregister values are to be examined, false if the

expression stack values are to be examined.

- Index The index of the register to be examined.
Value A var parameter to return the desired value.

Returns

- Success Process state returned.
MemFault Illegal address specified.

4.2.11. Depositing the microstate of a process

```
function Deposit(  
    KernelPort : Port;  
    RegOrStack : boolean;  
    Index      : integer;  
    Value      : integer  
) : GeneralReturn;
```

Abstract

Deposit changes the microstate of the process associated with KernelPort. If the process changed is not suspended, the values of the registers may not actually be changed.

Arguments

- KernelPort The kernel port of the process to be changed.
RegOrStack True if the microregister values are to be changed, false if the expression stack values are to be changed.

- Index The index of the register to be changed.
Value The new value of the register.

Returns

- Success Process state changed.
MemFault Illegal address specified.

4.2.12. Enabling software interrupts for messages

```
function SoftInterrupt (
    KernelPort      : Port;
    NormOrEmerg    : boolean;
var EnOrDisable   : boolean
) :GeneralReturn;
```

Abstract

SoftInterrupt enables or disables software interrupts for either normal or emergency messages.

It is suggested that SoftInterrupt be used on other processes. For your own processes SoftEnable is recommended because it contains a trap to prevent messages from getting out of order.

Arguments

KernelPort The kernel port of the process to be affected.

NormOrEmerg

True if the Normal message interrupt is to be enabled / disabled,
false if the Emergency message interrupt is to be
enabled / disabled.

EnOrDisable

True if enabled, false if disabled. Used to store previous value of
enable / disable flag on return.

Returns

Success Action performed. Previous value of flag in EnOrDisable.

4.2.13. Getting the SleepID

```
function GetIOSleepID(
  var SleepID      : long
) :GeneralReturn;
```

Abstract

GetIOSleepID returns the SleepID that would be used to suspend the calling process if it did an IPC receive. It is intended to be used by device driver processes that wish to use the kernel sleep trap and be awakened if a message is received. The following code could be used to put the current process to sleep for 1 jiffy (1/60 of a second) or until a message arrives.

```
begin
  loadexpr(SleepID);
  loadexpr(1);           { NTicks = 1 means
                         1 jiffy }
  loadexpr(0);           { ProcID = 0 means
                         this process }
  loadexpr(0); loadexpr(0); { FlagPtr = Nil }
  nopagefault(2);
  inlinebyte(KOPS);
  inlinebyte(KSleep);
end
```

Arguments

SleepID Returns the value of the current process IPC Sleep ID.

Returns

Success SleepID has been set.

5. Accent Virtual Memory Primitives

Accent provides two kinds of memory storage to processes: temporary virtual memory and more permanent kernel memory objects which may be directly backed by disk or other storage media.

5.1. Overview

Each Accent process may have up to 2^{32} bytes of paged virtual memory. Virtual memory may be written or read directly by a process using macro instructions defined by its micro-interpreter. The size of a page on a PERQ is 512 bytes.

Accent also provides as a service the notion of a segment which may be created, destroyed, read or written by explicit message operations. Segments have unique 32-bit identifiers and may be backed by physical memory, disk, or by a process through a port identifier. Disk memory is divided by Accent into logical disk devices called partitions and a separate port is provided for access to each partition.

The following type definitions are used by the various message interface routines:

```
type
    Port          = long;
    VirtualAddress = long;
    SpiceSegKind   = (Temporary, Permanent, Bad,
                      SegPhysical,
                      Imaginary, Shadow);
    SegID         = long;
```

5.2. Virtual Memory Management Routines

The functions described in this section are contained in the module AccInt in the file AccentUser.Pas.

5.2.1. Creating a segment

```
function CreateSegment(
  SpecialPort : Port;
  ImagSegPort : Port;
  SegmentKind : SpiceSegKind;
  InitialSize : integer;
  MaximumSize : integer;
  Stable      : boolean;
var Segment    : SegID
) : GeneralReturn;
```

Abstract

CreateSegment creates a segment that may be permanent, segphysical, temporary or imaginary:

- Permanent segments are allocated on disk and survive rebooting, etc. Their disk storage is allocated on the partition with which SpecialPort is associated.
- SegPhysical segments are contiguous regions of physical memory and are normally used for handling I/O devices, the bitmap display, etc. No disk storage is allocated for these segments.
- Temporary segments are the same as permanent segments but are deallocated when all processes having access to their pages are gone. Disk storage for these segments is allocated on the TemporarySegmentPartition as set by SetTempSegPartition.
- Imaginary segments are segment ids without storage. They can be read into an address space or written, but instead of disk operations being performed to get or write the data, messages are sent to the ImagSegPort specified in the CreateSegment call asking to read and/or write the specified pages of the imaginary segment. The imaginary segment handling process can then respond directly to read / write requests and manage directly the virtual memory of

another process. This is also a tool which can be used to support network paging. There is no disk storage for these segments. Imaginary segments are not implemented in this release.

Arguments

SpecialPort

The kernel port of the current process (in the case of temporary or imaginary segments) or a specially protected port (in the case of segphysical segments) or a port associated with a partition (in the case of permanent segments). Partition ports and the protected port are usually available only to system processes.

ImagSegPort

A port for read and write operations on imaginary segments. May be a NULLPORT for other types of segments.

SegmentKind

One of physical, temporary, permanent, or imaginary.

InitialSize

The size of the new segment in pages.

MaximumSize

The maximum size to which the new segment will be allowed to grow.

Stable

Whether or not the new segment is to be stable (not currently implemented).

Segment

A var parameter for returning the id of the created segment.

Returns

Success

Segment created.

BadRights

The SpecialPort used does not have the right to create the kind of segment desired.

PartitionFull

The partition specified does not have enough disk space to accommodate the segment.

5.2.2. Truncating a segment

```
function TruncateSegment(
    SpecialPort      : Port;
    Segment          : SegID;
    NewSize          : integer
) : GeneralReturn;
```

Abstract

TruncateSegment truncates the segment specified by Segment.

Arguments

SpecialPort

A protected port available only to selected processes (for Permanent or Physical segments) or a kernel port (for Temporary or Imaginary segments).

Segment The segment id of the segment to be truncated.

NewSize The new size of the segment in pages.

Returns

Success Segment truncated.

BadRights The SpecialPort used does not have the right to truncate the kind of segment desired.

NotASegment

The segment does not exist.

5.2.3. Destroying a segment

```
function DestroySegment(
    SpecialPort      : Port;
    Segment          : SegID
) : GeneralReturn;
```

Abstract

DestroySegment destroys the segment specified by Segment.

Arguments

SpecialPort

A protected port available only to selected processes (for Permanent or Physical segments) or a kernel port (for Temporary or Imaginary segments).

Segment The segment ID of the segment to be destroyed.

Returns

Success Segment destroyed.

BadRights The SpecialPort used does not have the right to destroy the kind of segment desired.

NotASegment

The segment does not exist.

5.2.4. Reading selected pages of a segment

```
function ReadSegment(
    SpecialPort : Port;
    Segment      : SegID;
    Offset       : integer;
    NumPages    : integer;
var Data        : pointer;
var DataCount   : long
) :GeneralReturn;
```

Abstract

ReadSegment reads the selected pages of Segment. The data read is pointed to by Data.

Arguments

SpecialPort

A protected port available only to selected processes (for

Permanent or Physical segments) or a kernel port (for Temporary or Imaginary segments).

- Segment The segment ID of the segment to be read.
- Offset The page offset of the first page to be read (file pages start at zero). Can be set to -1 to return the FIB (File Information Block). (For more information on FIB, see Section 5.4 "File Information Block" in the *File System* document in this manual.)
- NumPages The number of pages to be read. Specifying -1 returns the entire segment.
- Data On return, Data points to the data area into which the segment has been read.
- DataCount The number of bytes actually read.

Returns

- Success Segment data read.
- BadRights The SpecialPort used does not have the right to read the kind of segment desired.
- NotASegment The segment does not exist.
- MemFault Illegal address specified.

5.2.5. Overwriting selected pages of a segment

```
function WriteSegment(
    SpecialPort      : Port;
    Segment          : SegID;
    Offset           : integer;
    Data             : pointer;
    DataCount        : long
) : GeneralReturn;
```

Abstract

WriteSegment overwrites the selected pages of Segment with the data pointed to by Data.

Arguments

SpecialPort

A protected port available only to selected processes (for Permanent or Physical segments) or a kernel port (for Temporary or Imaginary segments).

Segment The segment ID of the segment to be written.

Offset The page offset of the first page to be written (file pages start at zero). Specifying -1 allows you to write only FSData sections of -1 block. (For more information on FIB, see Section 5.4 "File Information Block" in the *File System* document in this manual.)

Data A pointer to the data to be written to the segment.

DataCount The number of bytes to be written.

Returns

Success Segment data written.

BadRights The SpecialPort used does not have the right to write the kind of segment desired.

NotASegment

The segment does not exist.

PartitionFull

The partition specified does not have enough disk space to accommodate the segment.

MemFault Illegal address specified.

Writing block -1 will only change the FSData area of the block.

5.2.6. Intercepting all calls to the segment system

```
function InterceptSegmentCalls(
    ServPort      : port;
var OldSysPorts   : PtrAllPortArray;
var OldSysPorts_Cnt : long;
var SysPorts     : PtrPortArray;
var SysPorts_Cnt : long
) :GeneralReturn;
```

Abstract

This call is used by a process that wishes to intercept all calls to the segment system. It is intended to be used by trusted system processes only.

Arguments

ServPort This is the port that is to be used to make the call. It is the Kernel port of the process making the call.

OldSysPorts

This array will be sent to contain the current ports that are being used to make segment calls. All segment calls by other processes in the system will be received on this set of ports.

OldSysPorts_Cnt

This is the count of valid ports in OldSysPorts.

SysPorts SysPorts will contain the new set of ports that the calling process can use to make segment system requests.

SysPorts_Cnt

This is the count of the number of valid ports that are in SysPorts.

Returns

Success This is the only possible return.

5.2.7. Specifying a segment as disk backup

```
function SetPagingSegment(
    ServPort      : port;
    Segment       : SegID
) :GeneralReturn;
```

Abstract

This call is used to specify a segment that can be used as the disk backup for virtual memory. This segment is only used if the system cannot find a paging partition at system initialization time.

Arguments

ServPort The port of the process that is currently handling the paging system.

Segment This is the segment ID of the segment that is to be used for virtual memory backing store.

Returns

Success The call completed without error.

BadSegment
SegID was not a valid segment.

5.2.8. Determining available storage for virtual memory

```
function AvailableVM(
    KernelPort   : Port;
    var NumBytes  : long   { out parameter }
) : GeneralReturn;
```

Abstract

AvailableVM returns the size in bytes of secondary storage remaining as backing store for virtual memory.

Arguments

KernelPort Kernel port of calling process.

NumBytes Amount of remaining secondary storage available for virtual memory backing store.

Returns

Success Size (in bytes) of secondary storage returned.

5.2.9. Validating a given part of address space

```
function ValidateMemory(
    KernelPort : Port;
var Address   : VirtualAddress;
    NumBytes   : long;
    CreateMask : long
) : GeneralReturn;
```

Abstract

ValidateMemory marks a given part of a process address space as valid. References to data in this area will succeed and initial reads will return zero.

WARNING

Note that Validate Memory will not issue a warning if you have previously validated the same space. Using ValidateMemory to validate existing addresses may cause the content of those addresses to be overwritten.

Arguments

KernelPort Kernel port of process to be affected.

Address Starting address. If this address is 0, the kernel will find an area NumBytes in size and whose starting address is consistent with CreateMask and return that value in Address. This parameter should always be initialized (usually to 0) to prevent unexpected errors.

NumBytes Number of bytes to validate (rounded by the system to validate

only full pages).

CreateMask A means of easily specifying a certain alignment for the memory to be validated. The kernel will begin validating memory at the first address greater than or equal to Address such that for every one bit in CreateMask, the corresponding bit in the address is either zero or one, and for every zero bit in CreateMask, the corresponding bit in the address is also zero. Specifying -1 will give you the most convenient alignment. Specifying -256 will give you page alignment.

Returns

Success Memory validated.
MemFault Illegal address specified.

5.2.10. Invalidating a given part of address space

```
function InvalidateMemory(
    KernelPort      : Port;
    Address         : VirtualAddress;
    NumBytes        : long
) : GeneralReturn;
```

Abstract

InvalidateMemory marks a given part of a process address space as invalid. References to data in this area will fail. Any secondary storage being used to support this area will be released.

Arguments

KernelPort Kernel port of process to be affected.
Address Starting address.
NumBytes Number of bytes to invalidate (rounded up by the system to invalidate only full pages).

Returns

Success Memory invalidated.

MemFault Illegal address specified.

WARNING: InvalidateMemory always invalidates the full pages containing the address range given to it. It will, therefore, invalidate *more* memory than the address range, unless the address range begins and ends on a page boundary. It is safest to only invalidate memory allocated by ValidateMemory or MoveWrods.

Invalidateing out-of-line data received in a message is always safe.

One should *not* use InvalidateMemory to free memory allocated by the Pascal procedure NEW or the C function malloc.

5.2.11. Setting protection of valid pages

```
function SetProtection(
  KernelPort      : Port;
  Address         : VirtualAddress;
  NumBytes        : long;
  Protection      : integer
) : GeneralReturn;
```

Abstract

SetProtection changes the protection of valid pages in a process's address space.

Arguments

KernelPort Kernel port of process to be affected.

Address Starting address.

NumBytes Number of bytes to change protection of (rounded up by the system to change protection of only full pages).

Protection The new protection bits (ReadOnly, ReadWrite).

Returns

Success Memory protected.
MemFault Illegal address specified.

5.2.12. Reading another process's virtual memory

```
function ReadProcessMemory(  
    KernelPort : Port;  
    Address     : VirtualAddress;  
    NumBytes    : long;  
var Data       : pointer;  
var DataCount  : long  
) : GeneralReturn;
```

Abstract

ReadProcessMemory allows a process with access to another process's kernel port to read its virtual memory.

Arguments

KernelPort Kernel port of process whose data is to be read.

Address Starting address.

NumBytes Number of bytes to read.

Data Pointer to area into which data is to be read.

DataCount Number of bytes actually read.

Returns

Success Memory read.

MemFault Illegal address specified.

5.2.13. Writing to another process's virtual memory

```
function WriteProcessMemory(
    KernelPort      : Port;
    Address         : VirtualAddress;
    Data            : pointer;
    DataCount       : long
) : GeneralReturn;
```

Abstract

WriteProcessMemory allows one process with access to another process's kernel port to write that other process's virtual memory.

Arguments

KernelPort Kernel port of process whose memory is to be written.

Address Starting address in process to be affected.

Data Pointer to data to be written.

DataCount Number of bytes to write.

Returns

Success Memory written.

MemFault Illegal address specified.

5.2.14. Determining if an address is valid

```
function Touch(
    KernelPort      : Port;
    Address         : VirtualAddress
) : GeneralReturn;
```

Abstract

Touch determines if a given location Address is a valid address for the process specified by KernelPort.

Arguments

KernelPort Kernel port of process whose memory is to be queried.

Address Address to check.

Returns

Success Memory exists.

Failure Memory is invalid.

6. Accent Disk Handling

6.1. Overview

The following type definitions are used by the disk handling functions:

```
const
    MAXPARTCHARS = 8;   { maximum length for a
                          partition name }
    MAXDPCHARS   = 25;  { maximum length for dev:part
                          name }
    MAXPARTITIONS = 30; { maximum partitions on one
                         device }
    MAXDEVICES    = 5;   { maximum number of devices }

type
    Port          = long;
    PartString    = string[MAXPARTCHARS];
    DevPartString = string[MAXDPCHARS];
    PartitionType = (Root,UnUsed,Segment,PLX {...} );
    DiskAddr      = long;
    SegID         = long;

    PartInfo      =
        record
            PartHeadFree : DiskAddr; {pointer to Head of
                                      Free List }
            PartTailFree : DiskAddr; {pointer to Tail of Free
                                      List }
            PartInfoBlk : DiskAddr; {pointer to PartInfoBlock}
            PartRootDir : SegID;   {SegID of Root Directory }
            PartNumOps  : integer; {how many operations done
                                      since last update of
                                      PartInfoBlock }
            PartNumFree : long;    {HINT of how many free
                                      pages }
            PartInUse   : boolean; {this entry in PartTable
                                      is valid }
            PartMounted : boolean; {this partition is mounted}
            PartDevice  : integer; {which disk this partition
                                      is in }
            PartStart   : DiskAddr; {Disk Address of 1st page}
            PartEnd     : DiskAddr; {Disk Address of last page}
            PartKind    : PartitionType; {Root or Leaf}
            PartName    : PartString; {name of this partition}
            PartExUse   : boolean;  {Opened exclusively}
            Unused      : long;    {Port is not returned}
        end;
```

```
PartList = array[1..MAXPARTITIONS] of PartInfo;  
ptrPartList = ^PartList;
```

6.2. Disk Management Routines

The functions described in this section can be found in the module AccInt in the file AccentUser.Pas.

6.2.1. Getting information about partition structure

```
function GetDiskPartitions(  
    ServPort      : port;  
    interface     : DiskInterface;  
    log_unit      : InterfaceInfo;  
var unitnum     : Integer;  
var DevName     : DevPartString;  
var PartL       : Pointer;  
var PartL_Cnt   : long  
) :GeneralReturn;
```

Abstract

GetDiskPartitions will return information about the partition structure of a specific disk. This call is used by the file system. No other processes have access to this call.

Arguments

ServPort The port of the disk server.
Interface Specifies which disk interface is to be used.
log_unit The drive number on which the disk is attached.
unitnum The unit number as mounted.
DevName Set to be the name of the device that is at the unit Interface.
PartL A pointer to a structure that contains a list of the partitions that are on the device indexed by Interface.
PartL_Cnt This is the count of the number of valid partitions in PartL.

Returns

Success Operation completed.

6.2.2. Making a partition accessible

```
function PartMount(
    SpecialPort : Port;
    PartName    : string;
    ExUse       : boolean;
var RootID     : SegID;
var PartKind   : PartitionType;
var PartPort   : Port
) : GeneralReturn;
```

Abstract

PartMount makes the specified partition accessible through the port returned as PartPort.

Arguments

SpecialPort

A privileged port to the system kernel. Usually only system processes have access to this port.

PartName A string name for the partition to be mounted. Includes the device name.

ExUse An option specifying whether this process is willing to share access to the partition. If ExUse is true only one port at a time is allowed to access the partition. If ExUse is false, multiple shared mounts are allowed.

RootID The Segment ID of the root directory for this partition.

PartKind The Partition type of the partition. This is used as a hint to the formatting of the partition.

PartPort The port that is associated with the partition. It is required to be the SpecialPort in all the segment access calls to permanent segments on this partition.

Returns

Success Partition mounted.

NotADev Device part of the name is unrecognized.

NotAPart The partition does not exist.

PartNotAvail

Either some other process has the partition mounted for exclusive access, or ExUse is true and the partition is already mounted.

6.2.3. Making a partition inaccessible

```
function PartDismount(  
    PartPort      : Port  
) : GeneralReturn;
```

Abstract

PartDismount makes the specified partition inaccessible on this partition port.

Arguments

PartPort A port associated with a partition. Only system processes have access to partition ports.

Returns

Success Partition dismounted.

PartStillMounted

This port can no longer access the partition, but some other ports can.

6.2.4. Performing disk I/O while by-passing paging

```
function DirectIO(
    ServPort      : port;
var CmdBlk       : DirectIOArgs;
var DataHdr     : Header;
var Data        : DiskBuffer;
) :GeneralReturn;
```

Abstract

This call is used to perform disk I/O while bypassing the paging mechanism. It should be used with care so that it does not interfere with the pager's view of the disk.

Arguments

ServPort This must be one of the kernel's special ports.

CmdBlk A record describing the command to be executed is sent to the pager. It returns the IOStatus on completion.

DataHdr The 16 bit header to be written is passed in.

Data A single block can be read / written.

Returns

Success It worked.

Failure It did not. IOStatus in CmdBlk has the error code.

PERQ Systems Corporation
Accent Operating System

Kernel Interface
Accent Disk Handling

7. Accent Display Options

The Accent kernel provides a number of facilities that are used to quickly update uncovered portions of the display.

7.1. Overview

These operations are executed by the process that is making the call. In general, a client process makes a call to the window manager to update some portion of the display. The client address space window manager code will trap to the kernel and execute the appropriate function to perform the requested operation.

If the portion of the display that was to be updated is on the screen and uncovered, the kernel will perform the operations. If the kernel cannot perform the operation for some reason, it will return a failure indication. At this point, the client interface code will make an IPC request of the window manager process to do the display operation.

7.2. Display Management Routines

The functions described in this section can be found in the module AccCall in the file AccCall.Pas, and in the module AccInt in the file AccentUser.Pas.

7.2.1. Creating a rectangle for a given port

```
function CreateRectangle(
    ServPort      : port;
    RecPort       : port;
    BaseAddr     : VirtualAddress;
    ScanWidth    : Integer;
    BaseX        : Integer;
    BaseY        : Integer;
    MaxX         : Integer;
    MaxY         : Integer;
    IsFont        : Boolean
) :GeneralReturn;
```

Abstract

CreateRectangle allocates a rectangle (that can be used for screen operations) and associates it with a given port. The rectangle is Enabled to receive input.

Arguments

ServPort Service port (should be the permanent segment port).

RecPort Port to which the rectangle is attached. If the port already has a rectangle that rectangle is replaced.

BaseAddr Physical address of memory containing the rectangle.

ScanWidth Words per scan line for memory containing the rectangle.

BaseX, BaseY

Upper left corner of the rectangle relative to BaseAddr.

MaxX, MaxY

Lower right corner of the rectangle.

IsFont If TRUE, this rectangle is really a font. The BaseAddr points to the bitmap area of the font; the font header precedes it. RectDrawByte will only draw characters from a source rectangle that is a font.

Returns

Success Operation completed.

Failure Port was not the Permanent Segment Port.

BusyRectangle

RectPort has messages outstanding.

7.2.2. Destroying a rectangle for a given port

```
function DestroyRectangle(
    ServPort      : port;
    RectPort      : port
) :GeneralReturn;
```

Abstract

Deallocate a rectangle record for the specified port.

Arguments

ServPort Service port (should be the permanent segment port).

RectPort Port from which the rectangle is to be detached.

Returns

Success Operation completed.

Failure Port is not the permanent segment port, or the specified port does not have a rectangle.

7.2.3. Enabling a list of rectangles

```
function EnableRectangles(
    ServPort      : port;
    RectList      : PtrPortArray;
    RectList_Cnt  : long;
    Enable        : Boolean
) :GeneralReturn;
```

Abstract

Enables or disables an entire list of rectangles. An enabled rectangle can have graphics operations performed on it with the kernel graphics calls. This call will not enable the rectangles if any rectangle has messages waiting (to ensure

that all requests are processed in the proper order).

Arguments

ServPort Service port (should be the permanent segment port).

RectList Pointer to an array of ports whose rectangles are to be enabled or disabled.

RectList_Cnt

Number of ports in the array whose rectangles are to be enabled or disabled.

Enable Set to True to enable; False to disable.

Returns

Success Rectangles were enabled or disabled.

Failure Port was not the permanent segment port, or any of the ports do not correspond to the rectangles.

BusyRectangle

Messages were waiting on one of the ports.

7.2.4. Setting-up a window for kernel typescript

```
function SetKernelWindow(
    ServPort      : port;
    LeftX         : Integer;
    TopY          : Integer;
    Width         : Integer;
    Height        : Integer;
    Inverted      : Boolean
) :GeneralReturn;
```

Abstract

Sets up a window for the kernel typescript.

Arguments

ServPort Service port (should be the permanent segment port).

LeftX, TopY

Upper left corner of the kernel window in screen coordinates.

Width, Height

The size of the window.

Inverted If TRUE, the kernel typescript has white letters on a black background. If FALSE, it has black letters on a white background.

7.2.5. General RasterOp between rectangles

```
function RectRasterOp(
    DstRectangle      : port;
    Action            : Integer;
    DstX              : Integer;
    DstY              : Integer;
    Width             : Integer;
    Height            : Integer;
    SrcRectangle      : port;
    SrcX              : Integer;
    SrcY              : Integer
) : GeneralReturn;
```

Abstract

Kernel protected RasterOp. Performs a RasterOp from the source rectangle to the destination rectangle, clipped to the rectangle boundaries. Will fail if any of the source area is outside the source rectangle.

Arguments

DstRectangle

The port for the destination rectangle.

Action RasterOp function.

DstX, DstY, Width, Height

The area within the destination rectangle for RasterOp destination.
Relative to upper left corner of destination rectangle.

SrcRectangle

The port for the source rectangle.

SrcX, SrcY The area within the source rectangle for RasterOp source. Relative to upper left corner of source rectangle.

Returns

Success Operation completed.

OutOfRectangleBounds

Source area is not completely inside source rectangle.

CoveredRectangle

Source or destination covered. Must call window manager to perform operation.

BusyRectangle

Messages are queued up for the viewport. Must call window manager to keep queued operations synchronized.

BadRectangle

Port is not a rectangle.

7.2.6. Drawing a line in a rectangle

```
function RectDrawLine(
  DstRectangle    : port;
  Kind           : Integer;
  X1,Y1,X2,Y2   : Integer
) : GeneralReturn;
```

Abstract

Kernel protected Line Draw routine. Draws a line in the rectangle clipped to the rectangle boundaries. (RectDrawLine does not return OutOfRectangleBounds).

Arguments

DstRectangle

Port for destination rectangle.

Kind Kind of drawing operation:

- 0 Erase line
- 1 Draw line
- 2 Invert line

X1, Y1 One end point of the line.

X2, Y2 The other end of the line.

Returns

Success Line was drawn.

CoveredRectangle

The rectangle is covered. Must call window manager to perform operation.

BusyRectangle

Messages are queued up for the viewport. Must call window manager to keep queued operations synchronized.

BadRectangle

Port is not a rectangle.

7.2.7. Drawing a character string in a rectangle

```
function RectPutString(
    DstRectangle      : port;
    FontRectangle     : port;
    Action            : Integer;
    var FirstX        : Integer;
    var FirstY        : Integer;
    StrPtr            : Pointer;
    FirstChar         : Integer;
    var MaxChar       : Integer
) :GeneralReturn;
```

Abstract

Draws a character string in a rectangle. As much of the string as will fit is displayed, and the amount displayed is returned.

Arguments

DstRectangle

The port for the destination rectangle.

FontRectangle

The port for the Font. If NullPort, it uses the system font.

Action RasterOp Function for drawing characters.

FirstX, FirstY

Drawing position for the origin of the first character. Returns the position of the origin of the next character to draw.

StrPtr Pointer to a packed array of characters containing the string to draw.

FirstChar Position of the first character in the string to draw (from 0).

MaxChar Position of the last character to draw. Returns the number of the last character actually drawn.

Returns

Success String was drawn.

OutOfRectangleBounds

Area is not completely inside destination.

CoveredRectangle

Destination covered. Must call the window manager to perform the operation.

BadRectangle

Port is not a rectangle.

BusyRectangle

Messages are queued up for the viewport. Must call the window manager to keep queued operations synchronized.

NotAFont FontRectangle was not a font.

7.2.8. Coloring a rectangle

```
function RectColor(
    Rectangle      : port;
    Action         : Integer;
    X              : Integer;
    Y              : Integer;
    Width          : Integer;
    Height         : Integer
) :GeneralReturn;
```

Abstract

Operates on one rectangle to set, clear, or invert all its bits. If part of the area to be colored is outside the rectangle, it is ignored (RectColor does NOT return OutOfRectangleBounds).

Arguments

Rectangle Port for destination rectangle.

Action Function to use, as follows:

0 or ord(RectWhite) - set all bits to 0

1 or ord(RectBlack) - set all bits to 1

2 or ord(RectInvert) - invert all bits

X, Y Upper left corner of destination (relative to Rectangle's boundaries).

Width, Height

Width and height of destination.

Returns

Success Operation performed. May be clipped to DstRect's boundaries.

BadRectangle

Rectangle is not a rectangle.

CoveredRectangle

Rectangle is not enabled (covered).

BusyRectangle

Rectangle has messages queued.

7.2.9. Scrolling a rectangle

```
function RectScroll(
    Rectangle      : port;
    X              : Integer;
    Y              : Integer;
    Width          : Integer;
    Height         : Integer;
    Xamt           : Integer;
    Yamt           : Integer
) : GeneralReturn;
```

Abstract

Scrolls a portion of a rectangle up, down, left, or right and erases the part that remains.

Arguments

Rectangle The port for the destination rectangle.

X, Y Upper left corner of destination (relative to Rectangle's boundaries).

Width, Height

Width and height of destination.

Xamt Number of bits to move the area horizontally; negative numbers to move to the left, positive numbers to move to the right.

Yamt Number of bits to move the area vertically; negative numbers to move up, positive numbers to move down.

Returns

Success Operation performed. May be clipped to Rectangle's boundaries.

BadRectangle

Rectangle is not a rectangle.

CoveredRectangle

Rectangle is not enabled (covered).

BusyRectangle

Rectangle has messages queued.

OutOfRectangleBounds

Area to be moved (starting position) is partially outside Rectangle.

7.2.10. Returning a rectangle's parameters

```
function GetRectangleParms(
    ServPort      : port;
    RectPort      : port;
var BaseAddr     : VirtualAddress;
var ScanWidth    : Integer;
var BaseX        : Integer;
var BaseY        : Integer;
var MaxX         : Integer;
var MaxY         : Integer;
var IsFont       : Boolean
) :GeneralReturn;
```

Abstract

Returns the rectangle parameters for a specified port.

Arguments

ServPort Service port(should be the permanent segment port).

RectPort Port for which the rectangle is to be returned.

BaseAddr Returns physical address of memory containing the specified rectangle.

ScanWidth Returns words per scan line for memory containing rectangle.

BaseX, BaseY

Returns upper left corner of rectangle, relative to BaseAddr.

MaxX, MaxY

Returns lower right corner of rectangle.

IsFont Returns "This rectangle is really a font". The BaseAddr points to the bitmap area of the font; the font header precedes it.
RectDrawByte will only draw characters from a source rectangle that is a font.

Returns

Success Parameters are returned.

Failure The ServPort was not the permanent segment port.

BadRectangle

RectPort does not have a rectangle.

8. Summary of Calls

The following is a summary of the Pascal calls to Accent.

```
function Send(var MsgHdr:Msg;MaxWait:long;Option:SendOption)
             :GeneralReturn;

function SetPortsWaiting(var Ports:PortBitArray):GeneralReturn;

function Receive(var MsgHdr:Msg{inout parameter};MaxWait:long;
                 PortOpt:PortOption;Option:ReceiveOption):
                 GeneralReturn;

function EReceive(var xxmsg:Msg;MaxWait:long;PortOpt:PortOption;
                  Option:ReceiveOption):GeneralReturn;

function PortsWithMessages(MsgType:long;var Ports:PortBitArray)
                           :GeneralReturn;

function LockPorts(LockOrUnlock:boolean;Ports:PtrLPortArray;
                  PortsCount:long):GeneralReturn;

function MessagesWaiting(MsgType:long;var Ports:PtrLPortArray;
                        var PortsCount:long):GeneralReturn;

function MoveWords(SrcAddr:Virtual Address;var DstAddr:
                   VirtualAddress:{inout}NumWords:long;
                   Delete:boolean;Create:boolean;
                   Mask:long;DontShare:boolean):GeneralReturn;

function SoftEnable(NormOrEmerg:boolean;EnOrDis:boolean):
                   GeneralReturn;
```

```
function AllocatePort(KernelPort:Port;var LocalPort:Port:{out}
                      Backlog:BacklongValue):GeneralReturn;

function SetBacklog(KernelPort:Port;LocalPort:Port;Backlog:
                      BacklogValue):GeneralReturn;

function DeallocatePort(KernelPort:Port;LocalPort:Port;Reason:long)
                      :GeneralReturn;

function IndexInterpose(ServPort:Port;MyPort:Port;HisIndex:Long;
                        var HisPort:Port):GeneralReturn;

function PortInterpose(ServPort:port;MyPort:Port;HisPort:port;
                        var MyNewPort:Port):GeneralReturn;

function GetPortIndexStatus(ServPort:port;PortIndex:Long;
                            var BackLog:Integer;var NWaitingMsgs:Integer;
                            var EWaitingMsgs:Integer;var PortRight:port;
                            var PortType:Integer):GeneralReturn;

function GetPortStatus(ServPort:port;PortRight:port;var BackLog:
                        Integer;var NWaitingMsgs:Integer;
                        var EWaitingMsgs:Integer;var PortIndex:Long;
                        var PortType:Integer):GeneralReturn;

function ExtractAllRights(ServPort:Port;PortIndex:Long;
                           var PortRight:port;var PortType:Integer):
                           GeneralReturn;

function InsertAllRights(ServPort:port;PortIndex:Long;
                           var PortRight:port;var PortType:Integer):
                           GeneralReturn;

function Fork(KernelPort:Port;var HisKernelPort:Port;
              {out parameter}var HisDataPort:Port;
              {out parameter}Ports:PtrPortArray;
              PortsCount:long):GeneralReturn;
```

```
function CreateProcess(KernelPort:Port;var HisKernelPort:Port;
                      {out parameter}var HisDataPort:Port;
                      {out parameter}):GeneralReturn;

function Terminate(KernelPort:Port;Reason:long):GeneralReturn;

function SetDebugPort(KernelPort:Port;DebugPort:Port):GeneralReturn;

function Status(KernelPort:Port;var Status:PStatus):GeneralReturn;

function SetPriority(KernelPort:Port;Priority:PriorID)
                     :GeneralReturn;

function SetLimit(KernelPort:Port;ReplyPort:Port;Limit:long)
                  :GeneralReturn;

function Suspend(Kernel:Port):GeneralReturn;

function Resume(KernelPort:Port):GeneralReturn;

function Examine(KernelPort:Port;RegOrStack:boolean;Index:Integer;
                 var Value:Integer):GeneralReturn;

function Deposit(KernelPort:Port;RegOrStack:boolean;Index:Integer;
                 var Value:Integer):GeneralReturn;

function SoftInterrupt(KernelPort:Port;NormOrEmerg:boolean;
                       var EnOrDisable:boolean):GeneralReturn;

function GetIOSleepID(var SleepID:long):GeneralReturn;

function CreateSegment(SpecialPort:Port;ImagSegPort:Port;
```

```
SegmentKind:SpiceSegKind;InitialSize:  
Integer;MaximumSize:Integer;Stable:boolean;  
var Segment:SegID):GeneralReturn;
```



```
function TruncateSegment(SpecialPort:Port;Segment:SegID;NewSize:  
Integer):GeneralReturn;
```



```
function DestroySegment(SpecialPort:Port;Segment:SegID):  
GeneralReturn;
```



```
function ReadSegment(SpecialPort:Port;Segment:SegID;Offset:integer;  
NumPages:Integer;var Data:pointer;  
var DataCount:long):GeneralReturn;
```



```
function WriteSegment(SpecialPort:Port;Segment:SegID;Offset:Integer;  
Data:pointer;DataCount:long):GeneralReturn;
```



```
function InterceptSegmentCalls(ServPort:port;  
var OldSysPorts:PtrAllPortArray;  
var OldSysPorts_Cnt:long;var SysPorts:  
PtrPortArray;var SysPorts_Cnt:long):  
GeneralReturn;
```



```
function SetPagingSegment(ServPort:port;Segment:SegID):  
GeneralReturn;
```



```
function AvailableVM(KernelPort:Port;var NumBytes:long  
out parameter:GeneralReturn;
```



```
function ValidateMemory(KernelPort:Port;var Address:VirtualAddress;  
NumBytes:long;CreateMask:long):  
GeneralReturn;
```



```
function InvalidateMemory(KernelPort:Port;Address:VirtualAddress;  
NumBytes:long):GeneralReturn;
```

```
function SetProtection(KernelPort:Port;Address:VirtualAddress;
                      NumBytes:long;Protection:integer):
                      GeneralReturn;

function ReadProcessMemory(KernelPort:Port;Address:VirtualAddress;
                           NumBytes:long;var Data:pointer;var DataCount:
                           long):GeneralReturn;

function WriteProcessMemory(KernelPort:Port;Address:VirtualAddress;
                            Data:pointer;DataCount:long):
                            GeneralReturn;

function Touch(KernelPort:Port;Address:VirtualAddress):
                GeneralReturn;

function GetDiskPartitions(ServPort:Port;interface:DiskInterface;
                           log_unit:InterfaceInfo;var unitnum:Integer;
                           var DevName:DevPartString;var PartL:Pointer;
                           var PartL_Cnt:long):GeneralReturn;

function PartMount(SpecialPort:Port;PartName:string;
                   ExUse:boolean;var RootID:SegID;
                   var PartKind:PartitionType;var PartPort:Port)
                   :GeneralReturn;

function PartDismount(PartPort:Port):GeneralReturn;

function DirectIO(ServPort:Port;var CmdBlk:pointer;var CmdBlk_Cnt:
                  long)var DataHdr:pointer;
                  var DataHdr_Cnt:long;
                  var Data:pointer;var Data_Cnt:long):
                  GeneralReturn;

function CreateRectangle(ServPort:port;RecPort:port;BaseAddr:
                        VirtualAddress;ScanWidth:Integer;
                        BaseX:Integer;BaseY:Integer;
                        MaxX:Integer;MaxY:Integer;IsFont:Boolean):
                        GeneralReturn;
```

```
function DestroyRectangle(ServPort:Port;RecPort:Port):GeneralReturn;

function EnableRectangles(ServPort:Port;RecList:PtrPortArray;
                           RecList_Cnt:long;Enable:Boolean):
                           GeneralReturn;

function SetKernelWindow(ServPort:port;LeftX:Integer;
                         TopY:Integer;Width:Integer;Height:Integer;
                         Inverted:Boolean):GeneralReturn;

function RectRasterOp(DstRectangle:port;Action:Integer;
                      DstX:Integer;DstY:Integer;Width:Integer;
                      Height:Integer;SrcRectangle:port;SrcX:Integer;
                      SrcY:Integer):GeneralReturn;

function RectDrawLine(DstRectangle:port;Kind:Integer;X1,Y1,X2,Y2:
                      Integer):GeneralReturn;

function RectPutString(DstRectangle:port;FontRectangle:port;
                       Action:Integer;var FirstX:Integer;
                       var FirstY:Integer;StrPtr:Pointer;
                       FirstChar:Integer;var MaxChar:Integer)
                       :GeneralReturn;

function RectColor(Rectangle:port;Action:Integer;X:Integer;
                   Y:Integer;Width:Integer;Height:Integer):
                   GeneralReturn;

function RectScroll(Rectangle:port;X:Integer;Y:Integer;
                     Width:Integer;Height:Integer;Xamt:Integer;
                     Yamt:Integer):GeneralReturn;

function GetRectangleParams(ServPort:port;RectPort:port;
                           var BaseAddr:VirtualAddress;
                           var ScanWidth:Integer;var BaseX:Integer;
                           var BaseY:Integer;var MaxX:Integer;
                           varMaxY:Integer;var IsFont:Boolean):
                           GeneralReturn;
```