PERQ Systems Corporation

ACCENT PROGRAMMING MANUAL Volume 1

December 7, 1984

This manual is for use with Accent Release S6.

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.

ACCENT PROGRAMMING MANUAL

PREFACE

This manual contains documentation on programming with Accent, the operating system for the PERQ workstation. Accent currently supports the Pascal, C, and Lisp languages. Accent was developed jointly by PERQ Systems Corporation and the Spice Project in the Computer Science Department at Carnegie-Mellon University. "Spice" is an acronym for Scientific Personal Integrated Computing Environment.

This manual contains general information needed by all programmers. Before you use this manual you should be familiar with the material in the *Accent User's Manual*. In addition to this manual, you may need to refer to the following manuals which contain information pertinent to particular programming tasks:

- Accent Microprogramming Manual
- Accent Languages Manual (Pascal and C)
- Accent Lisp Manual

Definitions and routines for the servers are covered in this manual. Other definitions and routines that are available to application programs are covered in the documents "The Pascal

Library" and "C System Interfaces." All of the public modules in the Pascal library are listed in the document "The Module Index." All of these documents are in the *Accent Languages Manual*.

Other manuals for Accent are the *Accent Qnix Manual* (forthcoming in a future release) and the *Accent System Administration Manual*.

Throughout the manuals the term "PERQ" refers to all models of the PERQ workstation unless stated otherwise. When a distinction is made between the PERQ workstation and the PERQ2 workstation, the term "PERQ2" refers to both Model LN-3000 and LN-3500.

The following symbols have been used throughout the Accent manuals:

< >	Material that is to be replaced by symbols or text as explained in the accompanying text. Do not type the angle brackets. Example: <filename> indicates that you should type the name of your file.
[ ]	Optional feature. Do not type the square brackets.
{ }	0 to n repetitions of an optional item. Do not type the braces.
CAPITALS	Literal, to be reproduced exactly as shown (although it may be reproduced in upper-case or lower-case). Example: <filename.CMD> indicates that the filename must contain the extension .cmd.
|	"Or"--choice between the items shown on either side of the symbol.
CTRL	Control key
ESC, INS	Escape key (labeled as ACC ESC or INS on various models)
DEL	Delete key (labeled as REJ DEL on some models)
HELP	Help key
LF	Linefeed Key

RETURN	Carriage return
italics	Input to be typed by the user

ACCENT PROGRAMMING MANUAL

TABLE OF CONTENTS

THEORY OF OPERATIONS

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.

This document is adapted from the paper by R. Rashid and G. Robertson, "Accent: A Communications Oriented Network Systems Kernel," published in the Proceedings of the Eighth Symposium on Operating Systems Principles.

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.

Table of Contents Page

List of Figures

1. Introduction

Accent is an operating system which runs on the PERQ Systems family of single user graphic workstations. It is the Accent operating system that permits a group of individual workstations to become a unified computing environment distributed across many workstations connected via Ethernet. To accomplish this, the design of Accent is based on a powerful communications abstraction.

1.1. Organization of this Document

This chapter gives an overview of the system, and we recommend that all programmers read it.

Chapter 2 describes in detail the operation of the kernel of the Accent system. If you are not interested in the details of the operation of the system, you may skip this chapter.

Chapter 3 provides an overview of the major servers in the system.

Chapter 4 provides information about each of the three different environments available on the system.

Chapter 5 provides programming examples. This is the most interesting chapter for most users of this document.

1.2. Messages and Communications

Accent is a message-based operating system. All communication is provided through the sending and receiving of messages. The object upon which this abstraction is based is the Interprocess Communication (IPC) port. Ports are protected objects. The only way to gain access to a port is to have the Accent kernel provide that access.

The Accent IPC system supports multiple senders and a single receiver on IPC ports. In addition to the multiple senders and single receiver, an Accent IPC Port also has an owner. When the process that has the receive rights to a port dies, the owner of that port is notified. This allows a third controlling process to establish the communication channel between two other processes, by retaining ownership and passing receive rights to the subordinate processes.

The IPC system is built into the operating system at the lowest levels. There are a number of ways in which the IPC system and other portions of the operating system interact. This will become apparent in later discussions of the IPC and Memory systems.

1.3. Processes

Accent supports the concurrent execution of multiple processes. A priority scheduling system with preemption and aging is used. The aging system is used to lower the priority of compute-bound processes. When a process completes a full time slice without blocking, the priority of that process is lowered. When the process blocks, its priority is raised to its original level. This keeps the priority of highly interactive processes high while lowering the priority of compute-bound processes. In order to supply a degree of fairness, the time quantum is longer for the lower priorities. Processes are arranged in a tree of descendants. The operating system provides facilities that can be used to

control either single processes or entire trees of processes.

Accent provides a large paged virtual address space for processes. Each process has a 2^32 byte virtual address space. This space is broken into a number of 512 byte pages. In Accent it is not possible for processes to share portions of their address spaces. If a process wants to give another process access to part of its address space, the initial process sends that portion of the address space to the second process in an IPC message. This is a simple mapping operation; no data is actually moved.

If both processes want to have access to the data, a logical copy is made. In this case each of the pages is marked CopyOnWrite. As long as neither process wants to write any of the data, no copy is made. If either process performs a write operation to some portion of the data, the page that contains the data to be written is copied. It is important to note that neither process knows that this has happened. As far as both processes are concerned, they each have a separate copy of the data.

1.4. The Network

Accent is a communications oriented operating system. All requests made by clients are in the form of IPC messages. The IPC facilities are extended across the Ethernet by a set of servers. It is important to note that the semantics of inter-machine and intra-machine IPC facilities are exactly the same. There is no need for a process to know if the other processes with which it is interacting are on the local machine or on a remote machine. Because these facilities are built into the lowest levels of the operating system, all operating system facilities can be provided by local or remote machines. Under Accent neither the user nor the applications writer needs to know that the network is present.

1.5. The Window Manager

Accent provides a sophisticated window management facility. This facility is used to provide processes with a virtual display. The set of all processes' virtual displays are mapped onto the single PERQ physical display.

The Window Manager fully supports the covered window paradigm. Processes need not know the current state of their windows. The Window Manager is responsible for displaying only the portions of a process window that are visible at any given time.

If a window is a simple text window the Window Manager will "remember" what is in the window. When the state of the window changes the Window Manager will redisplay the newly uncovered portions of the window.

If the window is a graphics window the application program can request the same type of facilities as a simple text window. This is done by keeping off-screen copies of the covered portions of the window. In addition to this method, applications that wish to have more control over the display can request a notification when the state of their windows changes.

In addition to text and graphics output, the window manager is responsible for the control of the system keyboard and pointing device. Key strokes and puck presses are provided to the application as a set of abstract events. The events that are generated when keys or buttons are pressed can be defined by the application. The mapping of actual key and puck presses into abstract events is achieved by a user-alterable translation table. PERQ provides translation tables for all system functions. If you do not like these translations, you are free to change them, but in general we suggest that for system functions you use the standard

translations.

1.6. The File System

Accent provides a virtual memory mapped file system. All access to files are the same as access to virtual memory.

When a file is "opened" the file is mapped into the virtual memory of the process that opened the file. This is only a mapping operation; no data is moved. When a portion of the file is accessed, that portion is faulted into physical memory.

To write a file, the application program first builds the file in its virtual address space. The application program then requests the file system to save that portion of the address space on the disk with a specific name. (At this point, the data is copied to the disk.)

Of course all File System facilities work in the same manner regardless of the actual location of the file. Access to remote files is achieved in exactly the same manner as access to local files.

1.7. Environments

At the current time Accent provides its user with three different environments. They are: a) Native Accent, b) Qnix, an implementation of the UNIX System V operating system, and c) AI workstation environment running AccentLisp.

1.8. Native Accent Environment

The native Accent environment is started when the operating system is booted. The user interface to this environment is the Accent Shell. This Shell provides command processing, as well as a number of simple commands, for the Accent environment. Most commands used in this environment are interactive.

1.9. Qnix

Qnix is an emulator that provides a UNIX System V system call interface. This interface allows most UNIX software programs to be ported to run on the PERQ workstation by simply recompiling the source.

The user interface to the Qnix system is provided by the UNIX software shell. This provides a well known environment to a large number of users. There are very few differences between the Qnix environment and the environment provided by the UNIX System V operating system.

It is possible to have all of these environments running at the same time on the workstation. Each environment runs in a different window on the screen. To change from environment to environment it is only necessary to change to a different window. In a number of cases it is possible to make cross use of the environments. It is possible to run Accent programs from a UNIX software shell. It is also possible to run UNIX software programs from an Accent shell.

2. System Structures

This chapter describes in detail a number of the underlying system structures used and the facilities provided by the Accent kernel. Described in this chapter are: a) the process system, b) the memory system, c) the queuing system and d) the IPC system.

The modules VMTypes.Pas and VMIPCTypes.Pas define most of the structures referenced here.

2.1. Process System

Accent provides for the concurrent execution of multiple processes. Each process in the system has a 2^32 byte address space. All address spaces are disjoint.

The Accent scheduler provides a multiple priority scheduling system with preemption. The system provides 16 priority levels, with level zero being the lowest and level fifteen being the highest. Scheduling is round robin within a priority level. In addition the scheduler implements an aging algorithm that is used to lower the priority of compute-bound processes. If at any time a process finishes its time quantum without blocking, the priority of that process is lowered. This lowering will happen each time the process completes a time slice without blocking. Thus it is possible for a process to move from priority fifteen to priority zero in sixteen time slices. The next time the process blocks, its priority is raised to its original level. To implement a degree of fairness, the time quantum is made larger as the priority level decreases. This system provides rapid response to highly interactive processes.

The process management system interacts with the virtual

memory management and inter-process communication systems to provide the execution environment for processes running on a host machine.

All communication between a user process and the kernel is through a port, called its kernel port, which is created when the process is created. Since ports can be sent in messages to other processes, it is possible for process A to send its kernel port to process B. The process system is designed so that process B can manage process A's behavior, much the same way the virtual memory system allows one process to manage another's virtual memory. This mechanism forms the basis for remote debugging and monitoring systems.

2.1.1. Structures

2.1.1.1. PCBHandle (VMTypes.Pas)

The PCBHandle provides all of the information about a process that must be available to the Accent microkernel. There is one PCB handle for each process in the system. The set of all PCBHandles is kept in a PCBTable, which is an array 0..MaxProc of PCBHandles. This array is indexed using the process ID.

Figure 1 gives the definition of the PCBHandle.

2.1.1.2. MicroContext (VMTypes.Pas)

The MicroContext is used to hold information about a process when that process does not have control of the CPU. It holds the saved registers for the process as well as the EStack (contents of the hardware's expression stack) for the process. Figure 2 gives the definition of the MicroContext.

Figure 1: Definition of PCBHandle

```
PCBHandle = packed record
      NextEntry      : ProcID;          { Word 0, 16 bits }
      PrevEntry      : ProcID;          { Word 1, 16 bits }
      QueueID        : QID;             { Word 2, 6 bits }
      MsgPending     : boolean;         { Word 2, 1 bit  }
      EMsgPending    : boolean;         { Word 2, 1 bit  }
      InUse          : boolean;         { Word 2, 1 bit  }
      SVCEntry       : boolean;         { Word 2, 1 bit  }
      LimitSet       : boolean;         { Word 2, 1 bit  }
      SVStkInCore    : boolean;         { Word 2, 1 bit  }
      Priority       : PriorID;         { Word 2, 4 bits }
      MsgEnable      : boolean;         { Word 3, 1 bit  }
      EMsgEnable     : boolean;         { Word 3, 1 bit  }
      Fill3          : 0..16383;        { Word 3, 14 bits }
      SVStkPtr       : ptrMicroContext; { Words 4,5 }
      SleepID        : ptrInteger;      { Words 6,7 }
      RunTime        : Bit32;           { Words 8,9 }
      LimitTime      : Bit32;           { Words 10,11 }
      SleepTime      : Bit32;           { Words 12,13 }
      SPMPtr         : ptrSPM           { Word 14,15 }
    end;

  ptrPCBTable = ^PCBTable;
  PCBTable    = array [0..MAXPROCS] of PCBHandle;
```

Figure 2: Definition of MicroContext

```
  MicroContext  = packed record
                    EStk        : array [0..15] of integer;
                    Reg         : array [0..63] of integer;
                    Fill1       : array [1..32] of integer;
                    TrapCode    : integer;
                    TrapArgs    : VirtualAddress;
                    EStkCnt     : integer;
                    State       : ProcState;
                    Fill2       : Bit14;
                    RegCnt      : integer;
                    Fill3       : array [1..8] of integer;
                    Fill4       : Bit8;
                    BPC         : Bit4;
                    Fill5       : Bit4;
                    Fill6       : integer
                  end;
```

2.2. Memory System

This section describes in detail the way in which memory is addressed in the Accent system. The section is divided into two parts.

The first part describes the way in which a process does addressing. It describes all of the structures that make up the memory map for a process and how those structures are accessed using a process virtual address.

The second part describes the way that the system translates a virtual address supplied by a process into an address in physical memory or causes a page fault.

2.2.1. Process address space

2.2.1.1. Virtual memory management

In Accent, virtual memory, file storage and IPC are integrated together in a way that preserves the logical structure of inter-process communication while providing significant performance advantages over previous communication-based operating systems. This same melding of virtual memory with IPC makes it possible for Accent to allow one process to manage the virtual address space of another (either by allocating virtual memory from the kernel and sending it to another process or by explicitly managing page faults) and in so doing provides an easy mechanism for cross-network paging.

2.2.1.2. Virtual memory and files

The virtual address space of an Accent process is flat and linearly addressable. On the PERQ this address space is 2^32 sixteen bit words. An Accent segment is the basic unit of virtual memory allocation and secondary storage management. All randomly accessible secondary storage is considered part of the virtual memory of the system and is organized into Accent segments and managed by the kernel.

There are two kinds of Accent segments: temporary and permanent. Temporary segments are allocated by processes as required for their memory needs and are released when all processes which have access to them are terminated. The storage contained in permanent segments form the basis for the Accent file system. Permanent segments are allocated by sending messages to a special port normally supplied only to special processes. They do not disappear except by explicit request.

Normally, new segments are allocated by the kernel in response to a CreateSegment message, with the kernel responding in a message with the newly created segment's identifier. A segment can be explicitly destroyed through the use of a DestroySegment message. The data contained in a segment can be read into a process's virtual address space using a ReadSegment message. The reply to a ReadSegment message contains the newly allocated pages which are mapped into the requesting process's address space through reception of the message. Similarly, data can be explicitly transferred out of a process's address space through the use of a WriteSegment message.

2.2.1.3. Process directed management of virtual memory and network paging

An advantage of the Accent approach to virtual memory is that it allows virtual memory to be considered a process-provided resource. A process can create an imaginary segment and "read" it into an unused part of its address space. It is then possible for a process to send another process these 'imaginary pages' in a message. When the message is received, these pages are placed into the address map of the receiving process. When these pages are referenced by the receiving process, messages requesting their contents are sent to a port belonging to the process which created the segment. When a changed 'imaginary page' is about to be purged from main memory, its new contents are also sent in a message. This allows a process which is not the kernel to provide kernel-like management of secondary storage. In particular, it makes cross-network paging possible using the standard IPC facility.

2.2.1.4. Virtual address make-up

As indicated earlier, a virtual address under Accent is a 32 bit value. This value is made up a of a number of fields that are used during the translation process. This section will give a brief overview of those fields.

Bit 31 of the virtual address is the Supervisor / User bit. If the bit is set the process is currently executing in supervisor space. If the bit is clear the process is running in user space.

Bits 30 through 24 are used to pick one of 128 level 2 blocks. This value is used as an index into the Lev2 field of the process control block.

Bits 23 through 16 are used as an index into the specific level 2 block.

Bits 15 through 8 are used to pick a single page from a list of Spice Chunk Descriptors.

Bits 7 through 0 are used to pick a single 16-bit word in a page.

2.2.1.5. Structures

This section describes the data structures that define the virtual address space of a process. These structures are searched when a page fault occurs. They are not used for addressing when the desired page is in memory. There is a set of system tables that is used by the kernel to provide addressing when no faults are going to be taken (see Section 2.2.2). The tables provide much faster access to the required page than could be obtained by searching the three-level process map.

2.2.1.5.1. Spice Process Map (VMTypes.Pas)

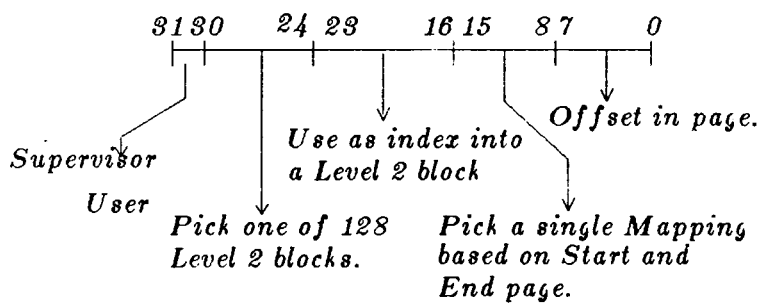
The Spice Process Map, SPM, is the starting point for the translation of a process virtual address into a physical address. The SPM is pointed to by a field in the PCBHandle for the process.

The F field is used to link a set of free SPMs. SCDFreeList is used to keep a list of free Spice Chunk Descriptors. (more about the SCDs later). MapBlks is a virtual address in the kernel virtual address space. It is the address of the start of the Map Blocks for this process. Map Blocks are a set of pages that are used to hold information about the process address mapping. Any single page in the set of pages that make up the map blocks holds a single type of information. A single map block can hold either Spice Chunk Descriptors or SCDIndices. The set of pages that make up the process map blocks are contiguous in kernel virtual address space. This implies that given the start address of the map blocks, simple address arithmetic can be done to get to a given map block. The Lev2 portion of the SPM holds a number of Lev2Indices.

Figure 3: Virtual Address

Virtual Address Makeup

Each process has a 2^{32} bit virtual address space. A VA is interpreted in the following way:



The Virtual Address Space of a process is described by a 3 level map. The levels are 1) Lev2Index, 2) SCDIndex 3) Spice Chunk Descriptor.

These three levels describe a portion of an active segment. The system wide Active Segment Table keeps track of all active segments.

Figure 4: Lev2 Portion of SPM

```
SPM   = packed record
          F             : ptrSPM;
          SCDFreeList   : SCDIndex;
          Proc          : ProcessNumber;
          MapBlks       : VirtualAddress;
          MapSize       : long;
          NextBlk       : Bit8;
          Lev2Size      : Bit8;
          NextFree      : VirtualAddress;
          EndFree       : VirtualAddress; { Next allocated - 1 }
          Lev2          : array [0..127] of Lev2Index
        end;
```

2.2.1.5.2. Lev2Index (VMTypes.Pas)

Each of these indices points to a map block that holds SCDIndices.

A Lev2 index is picked by bits 30-24 of the process virtual address. This number is used as an index into the Lev2 array in the SPM. It is possible for a process to validate a 32 mega-word portion of virtual address space by making a single entry in the Lev2 portion of the SPM. No other structures need to be created until a portion of the address spaces is referenced.

The Blk field of the Lev2Index picks a map block that contains a set of SCDIndices.

The WriteProtect and the CopyOnWrite fields in the Lev2Index provide the protection on the entire portion of the process virtual address space.

Figure 5: Lev2Index

```
Lev2Index       = packed record
                    Blk           : Bit7;
                    InUse         : boolean;
                    Free          : boolean;
                    case integer of
                    1:
                       (
                         WriteProtect  : boolean;
                         CopyOnWrite   : boolean;
                         Valid         : boolean
                       );
                    2: (
                         StatusBits    : Bit3
                       );
                    3: (
                         ProtectBits   : Bit2
                       )
                  end;
```

2.2.1.5.3. SCDIndex (VMTypes.Pas)

The Lev2Block is the second level in the translation map. There are a number of SCDIndices in each map block. Bits 23-16 of the virtual address pick a specific SCDIndex. Once again it is possible for a process to validate a 128k byte portion of address space by simply creating a single SCDIndex.

The WriteProtect and CopyOnWrite fields of the SCDIndex provide the protection information for that portion of the address space.

The Blk field picks the map block that contains an SCD. The Blkindex picks a particular SCD in that map block.

Figure 6: SCDIndex

```
SCDIndex        = packed record
                    case integer of
                    1:
                       (
                         Blk            : Bit7;
                         BlkIndex       : Bit6;
                         WriteProtect   : boolean;
                         CopyOnWrite    : boolean;
                         Valid          : boolean
                       );
                    2: (
                         Index          : Bit13;
                         StatusBits     : Bit3
                       );
                    3: (
                         Blank          : Bit13;
                         ProtectBits    : Bit2
                       );
                    4: (
                         All            : integer
                       )
                  end;
```

2.2.1.5.4. SCD (VMTypes.Pas)

The SCD describes a "chunk" of the process's virtual address space. Each SCD describes a variable size portion of the address space of a process. The smallest portion described is a single page. The largest portion described is 256 pages.

A larger portion of virtual address space is described by linking together a number of SCDs. Bits 15-8 of the virtual address are used to pick the SCD that describes the pages containing the page picked by these bits.

The SegmentPtr points to a structure called the Active Segment Table. This is a system-wide table that describes all of the currently active segments. SCDs for a number of different processes may point to the same AST entry. This allows the system to keep a single physical copy of a segment while allowing the different processes to see what they think are different copies of that segment.

If a process changes a page that is shared by other processes, the process that changed the page must get a new copy of it. This is done by the Shadow Segment facilities described later.

Figure 7: SCD

```
SCD             = packed record
                    StartPage      : Bit8;
                    EndPage        : Bit8;
                    SegmentPtr     : ptrASTRecord;
                    SegmentOffset  : integer;
                    case integer of
                      1: (
                           SCDNext        : SCDIndex
                         );
                      2: (
                           Blank          : Bit13;
                           WriteProtect   : boolean;
                           CopyOnWrite    : boolean;
                           Blank1         : Bit1
                         );
                      3: (
                           Blank2         : Bit13;
                           ProtectBits    : Bit2;
                           Blank3         : Bit1
                         )
                  end;
```

2.2.1.6. AST (VMTypes.Pas)

The Active Segment Record is used to describe a memory segment that is being accessed by one or more processes. The set of all active segments is kept in the Active Segment Table. This is a system-wide table.

If a process changes a page of a segment, the system will insure that the process is the only one that sees the change. This is accomplished using the shadow segment facilities. A new segment, a shadow, is created. This segment is described by the SCD that was used to gain access to the page that was changed. The ShadowOf field of the new segment points to the AST record of the original segment. The IndexPtr points to the index blocks for the new shadow segment. There will be a valid index block for the pages of the shadow that have been changed. Pages that

have not been changed have Index Blocks that contain a null value. When a reference is made to one of these non-changed pages that are described by the shadow, the system will follow the ShadowOf field to the original segment. The original segment will contain the desired page. This process allows the system to reduce the number of copies of pages that are kept, while still maintaining the "no shared memory" semantics of the system.

Figure 8: ASTRecord

```
ASTRecord  = packed record
               case integer of
                 1: (
                      F          : ptrASTRecord;
                      ShadowOf   : ptrASTRecord;
                      SegmentID  : SegID;
                      UseCount   : Bit13;
                      SegKind    : SpiceSegKind;
                      case integer of
                        1: ( PhysSize : 0..MEMPAGES-1 );
                        2: ( ImagID   : 0..MAXIMAG-1 );
                        3: ( SegOffset: integer)
                    );
                 2: (
                      IndexPtr           : ptrIndexBlock;
                      Blank              : ptrASTRecord;
                      OffsetInPagingArea : Bit16;
                      SizeInPagingArea   : Bit16
                    );
                 3: (
                      IndexAddr          : DiskAddr
                    )
             end;
```

2.2.2. System addressing

This section describes the set of system tables that is used by the kernel to provide addressing when no faults are going to be taken.

2.2.2.1. PVRecord

The PVRecord is used to maintain information about the physical memory pages in the system. There is a single PVRecord for each page in physical memory. When a memory request is made the PVRecord for the page that contains the memory location is checked to see if the request is a valid one.

A second important feature provided by the PVTable is that it maintains a list of all process virtual addresses that point to a physical page. This is used when a page is to be removed from physical memory. When the page is removed all virtual addresses that point to that physical page must be removed from the VPTable (see section 2.2.2.2). The PVList field of the PVRecord is an index into a second table maintained by the pager process.

Figure 9: PVRecord

```
PVRecord        = packed record
                    WriteProtect  : boolean;
                    CopyOnWrite   : boolean;
                    Used          : boolean;
                    Dirty         : boolean;
                    MappedToDisk  : boolean;
                    Free          : boolean;
                    Locked        : boolean;
                    DontTouch     : boolean;
                    HeaderChanged : boolean;
                    SwapIn        : boolean;
                    SwapOut       : boolean;
                    InStack       : boolean;
                    SVStkFlag     : boolean;
                    UseCnt        : Bit2;
                    Flush         : boolean;
                    PVList        : VPIndex
                  end;

ptrPV           = ^PV;
PV              = array [0..MEMPAGES-1] of PVRecord;
```

2.2.2.2. VPRecord

The VPTable is used to map a process virtual address into a system physical address. The VPTable is made up of a set of VPRecords. When a process presents a virtual address, the kernel translates that virtual address into a physical address or causes a page fault. The VPTable is the data structure used during that translation.

When an address request is made, a hash is computed using the virtual address and the process ID. The hash table used is the VPTable. The size of the hash table is determined by the number of physical pages in memory. In general there are about twice the number of hash entries in the table as there are physical pages in the memory. Hash collisions are chained together using the VPnext field in the VPRecord. The VA / ProcID pair is used as input into the hash function. The function returns an index into the VPTable. If the VA and ProcID presented to the hash match the VAddr and Proc in the VPRecord, then the VPRecord provides the desired physical address. If they do not match, the chain of VPRecord, linked on VPNext, is searched. If no match is found, the requested page is not in memory and the process will be forced to take a page fault.

2.3. Queue System

The queue system is used to provide process queueing. It is a very simple system. A process must be on one and only one queue at any given time. The set of all queues is kept by the kernel. Figure 11 gives the definition of the system queue tables.

The meaning of each queue in the queue table is defined by a set of constants in VMTypes.Pas (see Figure 12).

Figure 10: Definition of VPRecord

```
VPRecord          = packed record
                      case integer of
                        1: ( VAddr         : VirtualAddress;
                             PAddr         : PhysicalAddress
                           );
                        2: ( Proc          : ProcessNumber;
                             Blank1        : Bit8;
                             Blank2        : Bit16;
                             WriteProtect  : boolean;
                             CopyOnWrite   : boolean;
                             Used          : boolean;
                             Dirty         : boolean;
                             InUse         : boolean;
                             MappedToDisk  : boolean;
                             InPlace       : boolean;
                             EndOfVPList   : boolean;
                             PLsw          : Bit8;
                             PMsw          : Bit4;
                             VPNext        : ShortVPIndex
                           );
                        3: ( VPFreeList    : PageList;
                             StatusBits    : integer
                           )
                    end;
```

Figure 11: Definition of System Queue Tables

```
Queue           = record
                    QHead          : ProcID;
                    QTail          : ProcID;
                    TimeSlice      : integer;
                    Filler         : integer
                  end;

ptrQueueTable = ^QueueTable;
QueueTable    = array [0..NUMQUEUES] of Queue;
```

Figure 12: System Queue Constants

```
NUMSLEEPQS  = 32;          { must be power of two }
QNIL        = 0;           { Nil Process ID or Nil QueueID}
Front       = true;
Back        = false;

All         = true;
One         = false;
NUMQUEUES   = NUMSLEEPQS + NUMPRIORITIES + 6;
RUNQS       = 1;           { Run Queues }
                           {  One for each priority }
                           {  PCB locked in real memory }

PENDINGQ    = RUNQS + NUMPRIORITIES;
     { Pending Queue }
                           {  Either timeslice end }
                           {  or PCB not in real memory }

SLEEPQ      = PENDINGQ + 1;
     { Sleep Queues }
                           {  Blocked on SleepID }

KILLQ       = SLEEPQ + NUMSLEEPQS;
                           { Process termination queue }

FREEQ       = KILLQ + 1;   { Unallocated process queue }

LIMITQ      = FREEQ + 1;   { Queue for processes which
                           {  have exceeded time limit }

SVCWAITQ    = LIMITQ + 1; { Waiting on Supervisor busy }
```

2.4. IPC System

Although processes are the active components of the system, the notion of process is a poor one on which to base a communication facility. Different languages may define multiple concurrent tasks within a single kernel-supported process. Moreover, a given service may be provided over a period of time by a number of different processes.

The basic transport abstraction of the IPC is the notion of a port. A port is a protected kernel object into which messages may be placed by processes and from which messages may be removed. All services and facilities provided by a process are made available to other processes through one or more ports.

Ports are intended to be used by processes to represent specific services or data structures. For example, a file system process might associate a separate port with each open file, or a virtual terminal handling process might allocate a port to represent each virtual terminal. However, no restriction is placed on their use. Ports may be used by processes to communicate information in any mutually convenient way.

Logically associated with each port is a queue on which resides messages sent to that port but not yet removed from it by a process. The ability to remove messages from a port is called receive access. Only one process may have receive access to a port at a time, although receive access to a port may be transferred to another process in a message.

Ports cannot be directly manipulated or named by a process. Instead, the kernel provides processes with a secure capability to send a message to a port and/or receive a message from it. This capability is a local name for a system object, much in the way a UNIX software file descriptor is a local name for a system

maintained file, pipe, or device. Port capabilities may be passed in messages, handed down to process children, or destroyed. A given process may have only one local name for a given port at a time. Whenever a port name is passed in a message the system kernel must map that name from the local name space of the sending process into the name space of the receiving process.

The ability to manipulate access to ports allows for the redirection of communication from one process to another and the explicit management of communication between two processes by a third process. It also allows a port to be used to refer to a specific service or process-provided object even in situations in which that service or object is handled by different servers at different times. Neither the address (i.e. the location) nor the name of a process can be determined from the capability to send a message to one of its ports.

2.4.1. Creating, accessing and destroying ports

A port is created by a process through an AllocatePort system call. It is a system object, distinct from the process which created it but initially owned by the creating process. The result of the AllocatePort call is a local port name which refers to the created port.

Initially the owner of a port also has receive access to it. Ownership of a port and receive access are, however, logically distinct. Ownership of a port may be passed in a message from one process to another, but not shared. Receive access to a port may also be passed in a message but not shared. These restrictions prevent multiple servers from managing the same queue, and are necessary to avoid serious problems which occur when access to a single queue is shared by processes on different machines.

If a single process both owns and has receive access to a port,

that process may destroy the port by performing a DeallocatePort system call. A process with a capability (local name) to a port may release the capability via the same system call.

A port is automatically destroyed when its owner and the process with receive access to it both die. In either case, all processes with access to that port are notified via emergency messages. If the same process does not both own and have receive access to a port, then the deallocation of the port name by either process results in an emergency error message being sent to the other accompanied by full access rights to that port (i.e., both ownership and receive rights). All user programs that may receive port died messages should actually do the receive as the pile-up of port died messages will lead to system degradation. This is most true of processes that may run a long time and interact with processes that may deallocate ports or pass away.

The purpose of distinguishing receive access from ownership is to allow a process to take over services or functions provided by other processes in the event those processes should die or malfunction. This is particularly important when writing fail-soft software and can also be used to provide orderly shut-down of services after catastrophic failures.

2.4.2. Software or pseudo interrupts

In the case where a process does not wish to wait for messages explicitly, it is possible to enable a software interrupt which will be triggered upon message reception. The interrupt service routine, executing in the context of that process, can then receive the incoming message and process it or notify the main program of the event in a user-defined manner. A mechanism such as this can allow processes which are inherently compute-bound to react quickly to incoming messages without using some form of message polling.

2.4.3. Flow control

The message queues attached to ports have a finite length. This prevents a sending process from queueing more messages to a receiving process than can be absorbed by the system and provides a means for controlling the flow of data between processes of mismatched processing speed.

Subject to implementation restrictions on maximum port size, the process owning a port is allowed to specify its backlog--the maximum number of normal messages which may be queued for that port at one time. Should a process attempt to send a message to a full port, one of three things may happen depending on options specified by the sender:

The process is suspended until the message can be placed in the queue.

The process is notified of an error condition (after perhaps allowing itself to be suspended for up to a specific period of time waiting for the message to be sent).

The message is accepted and the kernel sends a message to the sending process when that message can actually be placed in the queue. A maximum of one message per sending process per receiving port may be outstanding in this fashion.

These three options correspond to three different programming situations:

The first option is the one most likely to be used by a 'user process' when communicating with a 'server process.' In this situation the user process does not care if it is suspended for some time waiting for a message to be delivered to the server (as in the case of a remote procedure call).

The second option is used when a process does not care whether a

particular message is sent to a destination, but is using the message only to wake up a dormant partner. The fact that other messages are in the queue for the partner's port indicates that the partner is already scheduled to be activated.

The third option is the one most likely to be used by a service process when dealing with a user process. The server probably cannot afford to be suspended waiting on a user to clear its queue. It may also not want to just throw away the message or poll the user explicitly until the message can be sent. The system provides an explicit message event corresponding to the unblocking of the user's port queue.

2.4.4. Waiting for messages

Although particular protocols may specify synchronous behavior, the receipt of a message is inherently an asynchronous event. In a network environment, in particular, it becomes possible for error conditions to cause messages to be sent to a process at almost any time. Mechanisms must therefore be provided for a process to check the state of its ports, to wait for activity on one or more of its ports, and to receive messages selectively. It is also possible for a process to specify a maximum amount of time to wait for a message before resuming.

Four basic primitives are provided for accomplishing this task.

Receive(SetOfPorts,Message,Timeout) waits for at most Timeout milliseconds and if a message is available during that time from any of the ports designated by SetOfPorts then the message is read into Message and a boolean value of true is returned.

MessageWait(SetOfPorts,Timeout) performs a similar function, but does not receive the pending message. It returns the capability of the port from which the next message would be received.

Preview(SetOfPorts,Message,Timeout) is again similar to Receive, but it reads only the header of the next waiting message and does

not actually dequeue that message from the port queue.

PortsWithMessagesWaiting(SetOfPorts) is an informational routine which checks the status of all ports and returns the set of ports with messages waiting.

2.4.5. Sending and receiving kernel messages

Each newly created process has access to two ports whose primary purpose is to allow messages to be sent to and received from the Accent kernel. The first is called the kernel port of the process, and the kernel logically has the receive rights for this port while the created process has the send rights. The second is called the data port, and it is normally used by a process to receive messages from the kernel.

The parent of a process can, at the time of process creation, ask that other ports to which it has access be given to its child process. The parent can also get access to the kernel and data port of the child process. This, taken together with the ability to send port access rights in messages, is the basic mechanism for establishing communication between processes.

2.4.6. Messages

A message is logically a collection of typed data objects. At the time of a Message Send call, the message is copied from the address space of the sender into the address space of the receiver who has performed a Receive Call. copied from the address space of the sender at the time of a message send call into the address space of the receiver when a receive call is performed. Physically, a message is divided into two parts: 1) the message header, which contains information normally associated with all messages and 2) an optional description of structured data to be sent. The purpose of this division is primarily to optimize the transmission of short control messages and to make it easier for the kernel to find critical information which must always be

contained in a message--such as its destination port.

2.4.7. The message header

The message header contains a small amount of system required message information and is used to form an anchor for the structured part of a message. At minimum it specifies the message type and contains a capability for the destination port of a message and a field for a capability (which may be empty) to be used for a reply. The destination and reply ports are more commonly referred to as the remote and local ports, respectively. The header also contains an ID field to be used for discriminating message types and a pointer to the structured data part of the message.

The header may be checked without actually receiving the data portion of a message by calling Preview. The purpose of this facility is to allow a process to check the ID of the message being received and select an appropriate message structure for receiving it.

The message type contains information which determines the kind of service it requires of the IPC. Two types have been given special names and play a dominate role in communication:

> Normal messages - A 'normal' message is flow controlled, sequential, reliable, not secure, of lowest priority, and has no maximum age. This is the default message type and satisfies most communication requirements.
>
> Emergency messages - Emergency messages are specially flow controlled, reliable, not secure, of highest priority, and have no maximum age. Emergency messages play an important role in error handling. Because of their high priority, they are guaranteed to be received before any normal messages are sent to a port. Their purpose is to allow urgent information to be delivered to a process regardless of that process's current message backlog or

message queue. They are used for error notification, special event processing, and debugging. If two emergency messages are in a port the Receive call will return them in order.

The notion of message type allows the programmer to specify the exact requirements of his IPC use. This gives the underlying system more information about a message and thus makes possible optimizations in the delivery and management of messages. This use of message types is consistent with the overall goal of making as much of the inner workings of the communication system as possible visible to the 'outside world' rather than hidden inside compiled algorithms, thus allowing greater flexibility in optimization, management, and monitoring.

2.4.8. Exceptional condition handling

Distributed programming imposes a heavy responsibility to handle a multitude of error conditions. Message activity can be pipelined or multiplexed, and the relationships between incoming and outgoing messages can be much richer than in a conventional programming environment (i.e., one in which subroutines are used as the primary structuring mechanism). A faulty process could conceivably crash many processes by sending illegal messages, making it very hard to identify the source of the problem. As a practical matter, it is difficult to ensure complete compatibility between similar programs written by different individuals in different languages. Supposedly interchangeable processes may differ in subtle ways.

A variety of facilities for protection, error detection, and error handling has been built into the IPC. Illegal process addresses are noted within the send and receive primitives, and the appropriate error returned to the offending process. In addition, processes are protected from accidental or malicious access through the use of port capabilities rather than global port or process identifiers. Whenever a port is destroyed the kernel

notifies all processes which still have access to that port via an emergency message. Emergency messages themselves are important because they give processes a way to reliably communicate errors in situations where normal communication channels are blocked. Software interrupts allow processes to handle error conditions without interfering with normal execution.

2.4.9. Optimizing message transfers

Messages are logically copied from a process's address space into a kernel message data structure upon transmission and are logically copied from the kernel to a process's address space upon message reception. Since data is never shared between processes, message communication over a network has precisely the same semantics as local message communication.

Double copy semantics need not, however, imply actually copying the data twice. If a process sends a message pointing to pages in its virtual memory and either releases the memory or simply never changes it, double copy semantics can be preserved by marking the pages referenced in the message as copy-on-write and not actually copying them into the supervisor's address space. Moreover, if the receiver of the message doesn't care about its placement or desires that it be placed in its memory on the same page boundaries, no copy of data need be performed at all. Instead, the data pages may be placed directly into the address map of the receiving process.

Thus, when a process sends a message to a port, whole virtual memory pages referenced in the message are not copied but instead marked as copy-on-write in the address space of the sender. Should the sender attempt to change any or all of these pages, new virtual pages will be allocated, filled with the appropriate data, and placed in its address space. On reception into an area of the receiver's virtual memory which is properly aligned, pages referred to in the incoming message are mapped in

rather than copied. If the receiver desires a different alignment of data than that specified by the sender, a copy operation will be performed.

The utilization of virtual memory by the IPC represents a functionally transparent optimization which is not required either for the functioning of the IPC or the virtual memory management facility. Nevertheless, this optimization can be of enormous value in increasing the speed of communication between processes on the same host.

Example:

The advantage of integrating virtual memory, file storage, and IPC is graphically illustrated by the example of file system access provided through messages sent to and from a file system process. The file system process can read secondary storage by sending a message to the kernel. The kernel then sends back a message which contains the virtual pages requested by the file system. This message need look no different than any other message containing data. Moreover the requested pages need never be copied, but are simply placed into the address space of the file system in the normal way that data is placed into the address space of any process when received in a message. If a user requests a block of data, however large, the file system can send it that block in a message, again without ever having the underlying data referenced by either the kernel or the file system. In this way the speed advantages of large memory mapped files can be obtained through the reception of normal IPC messages without resorting to special file mapping primitives.

2.4.10. Structures

This section describes the structures that are used by the IPC system.

2.4.10.1. Port structures

Figure 13 gives the definition of the structure that is used to define an IPC port. There is a single PortRec for each port in the system. The record keeps all information needed by the IPC system on a per port basis.

It is interesting to note that the queue of messages for a port is not kept with the port. That queue is kept in the per process IPCRecord (see Section 2.4.10.2).

The SecondaryMsgQ is used to hold a single message. This is the message that can be sent to a port when that port is full. When a process puts a message into this queue, the sending process will be notified when it is possible to place that message into its standard queue.

The Senders field of the record contains a bit array that has a bit for each process in the system. If a process has send access to the port, the bit in the Senders field that corresponds to the ProcessID of the sending process will be set.

Figure 13: Definition of Structure Used to Define an IPC Port

```
PortRec  = record
             SecondaryMsgQ : ptrKMsg;
             NumQueued     : integer;
             Backlog       : integer;
             Owner         : ProcID;
             Receiver      : ProcID;
             Locked        : boolean;
             InUse         : boolean;
             Senders       : ProcBitArray;
             RectanglePtr  : ptrRectangle
           end;
```

2.4.10.2. Process IPC structures

For each process in the system there is a per-process table that is used by the IPC system. This table contains information about the IPC state for that process.

All messages for a process are linked using a field in the IPCRecord. In general the number of messages that are waiting for a process to receive at any given time is small. In addition to this fact, most receives are done on the set of all ports to which the process has receive access. Based on these two facts there is a performance improvement to be had by linking the message with the process. When a process does a receive, the system need only look at the process IPC table. The system does not have to search the list of ports to which this process has receive access.

The DebugPort is a field that contains a port on which the system can send a message if this process dies. The Limit and Wait ports are ports on which the system can send a message to notify some process that something unusual has happened to this process.

MsgsWaiting is a pointer to the first message in a linked list of messages for this process.

KPorts contains the Kernel and Data ports for the process.

Figure 14: Definition of IPCRecord

```
IPCRecord = record
              Active       : boolean;
              State        : IPCState;
              DebugPort    : Port;
              MsgsWaiting  : ptrKMsg;
              PreviewMsg   : ptrKMsg;
              LimitPort    : Port;
              WaitOption   : integer;
              WaitPt       : Port;
              UseHashTable: boolean;
              KPorts       : array [KERNELPORT..DATAPORT]
       of Port
            end;

ptrIPCRecord = ^IPCRecord;

ptrIPCInfo   = ^IPCInfo;
IPCInfo      = array[KERNELPROC..NPROC+1] of IPCRecord;
```

3. Major SubSystems

This chapter gives an overview of the major subsystems that make up the Accent operating system. Each major subsystem, except the Message Server, is discussed in more detail in its own document in this manual.

Accent is structured as a set of cooperating processes. All requests are made by building a message and sending that message to a server request port.

This method of making requests is used throughout the operating system. Even requests of the Kernel are made by building a message and executing a Send. The Send code will look at the message to determine the port to which the message is to go. If the port is the kernel port of the process, the send is turned into the correct Kernel trap for the requested function.

For details on how to make use of these servers, see the documents on the specific server or the Programming Examples portion of this document.

For the name of ports that provide most of these functions, see Appendix B.

For system defined server ports, see Appendix D.

3.1. Kernel

The kernel of the Accent operating system provides the virtual machine on which Accent processes execute. The kernel is implemented in a combination of microcode and Pascal. In theory the only functions that must be provided by the kernel are the primitive IPC facilities. For performance reasons a number of other facilities are currently provided by the kernel.

The kernel provides the following interfaces to the Accent virtual machine:

1. IPC System facilities:
    a. allocation and deallocation of ports
    b. changing port backlog
    c. interposing a port for another port
    d. send and receive primitives
    e. control of software interrupts associated with the IPC system.
2. Process control facilities:
    a. Fork
    b. Terminate
    c. Priority
    d. Setting limits
    e. Suspend and resume process execution
3. Memory management facilities:
    a. Validate and Invalidate (i.e., Allocate and Deallocate) process virtual address space

b. Reading and writing of process Stack and register sets

c. Reading and writing of process virtual memory space.

4. Segment control facilities:

a. Create, Destroy and Truncate segment

b. Read and Write Segments to and from backing store

5. Backing store control and status

a. Get information about the disks

b. Mount and DisMount disk partitions

c. Direct Disk I/O operations

6. Display operations

a. RasterOp

b. Line

c. DrawByte

The interfaces to the kernel-provided facilities are found in AccentUser and AccCall in LibPascal.

3.2. File System

Accent provides a single level store view of the memory hieriarchy. In the Accent system there is only virtual memory. Some of that virtual memory can reside on disk and have a name. This long term storage of virtual memory is the responsibility of the file system. When a file is to be read, a request is made to the file system to provide that file. The file system will arrange to have the entire file mapped into the requester's address space.

No actual data transfers occur during this operation; only mapping operations are performed. When the process touches a piece of the data, the page that contains the data is brought into memory using the normal page fault facilities. When a file is to be written to disk, it is first created in virtual memory. The client process then asks the file system to save that piece of virtual memory on the disk with a specific name.

The file system provides access to files that are on the local disk. In addition, it provides access to files that may reside on other disks that are attached to other machines on the network. When a request is made for a file that does not reside on the local machine, the file system does a number of operations. First it must look to see if it has ever accessed a file on the remote machine before. If so, it will have access to a request port for the remote machine. If this is the first access, the file system must try to find the desired machine. It does this by doing a simple IPC look up of the remote file access port for that machine. Once the port is obtained, the request is made for the file. If the desired file is small, the entire file is sent to the requesting machine in a single message. If the file is large the remote machine sends an IOU for the data. Segments of this type are called imaginary segments. They are a piece of memory that is backed by another process. When the local process touches a page of the imaginary segment, a request is made to the remote machine for the desired page.

A client of the file system does not need to know if a file is on a local disk or on a remote disk. All operations on the file are accomplished in the same manner.

The direct interfaces to the file system in SesUser.Pas are very low level. The module PathName.Pas provides a much more user-oriented interface to the filing system. This is the recommended interface for users of the file system.

3.3. Window Manager

The Window Manager provides the facilities that are used to place text and graphics on the screen and to obtain input from users. These facilities are provided by a set of three different processes. These processes act in close cooperation to provide the user oriented input / output facilities of the system.

The Window system is made up of three different abstractions. They are:

- Windows - Windows are a user abstraction. They are the objects that the user deals with on the screen. Windows can be moved to different locations on the screen in all three dimensions. Windows can be uncovered, partially covered, completely covered, or off screen. In addition to a virtual display the window also provides a control point for a process or a group of processes. The Window Manager provides process group control operations that can be invoked on the processes that are associated with a window. The icons provided by the system describe the state of windows and of the processes associated with those windows.

- ViewPorts - ViewPorts are output abstractions that are presented to clients of the window manager That is, programs using the Window Manager). All output is directed to a viewport and clipped to the boundaries of the viewport. In general most windows are made up of two viewports. One viewport (the "outer" viewport) covers the entire area of the window, including its borders and title line. The second viewport is the client usable area of the window. This viewport is usually called the "inner viewport."

- TypeScripts - A TypeScript is a special viewport used only to display textual input and output.

3.3.1. General

The Window Manager is the process that handles the window and display system. It supports a full implementation of covered windows. The Window Manager also provides additional facilities through the use of icons.

The icon window is controlled by the Window Manager. Clients of the Window Manager can make requests for changes to be made to the icon that is associated with that client.

3.3.2. TypeScript

The TypeScript manager maintains standard text windows (Typescripts), providing line editing and redisplay functions for user programs that do not require graphics output or elaborate input control.

TypeScript remembers the last several pages (a window's worth) of text output by the program using a typescript. When a text window changes state, size or coveredness, TypeScript will redisplay the changed portion of the window. TypeScript will also redisplay on request information that may have scrolled from the window.

TypeScript allows the user to edit input lines using a subset of the commands available in the system editor (all of the single-line editing functions) and to recall previous input lines to be edited into new input lines.

TypeScript also handles Escape Completion for each typescript. The user may type a partial filename and press the Escape key; TypeScript will ask the file system to complete the file name by

finding the longest unambiguous match to the partial name. It will then add the name to the line of input, so that the user can specify the rest of the name or add more to the line.

TypeScript can either stop at the end of each screenful of output ('more' mode) or scroll output continuously. The user can select which mode to use, under keyboard control.

In 'More' mode, when a full page of output has been displayed, a black bar appears at the bottom of the window. The user then presses LineFeed to display the next page of output.

In continuous scroll mode, the user can use the Process Control Functions Suspend and Resume to stop or start output.

3.3.3. Tracker

Tracker is the process that actually listens to the keyboard and the tablet. It receives data from these input devices and then routes that data to the appropriate process. In addition it is Tracker that keeps the cursor tracking the puck or mouse.

3.4. Environment Manager

The Environment Manager provides processes with an execution environment. This environment can include any number of user and system defined variables. Environment Manager variables can be of two types: a) simple strings or b) searchlists.

It is possible for a client of the Environment Manager to set an environment variable and then to later obtain the value of that variable. In addition it is possible to ask the Environment Manager to make the environment variable known to other processes in the system.

Environment variables can be defined as simple strings or they

can be defined in terms of other environment variables.

The client interface to the Environment Manager is in EnvMgrUser.Pas in LibPascal.

3.5. Net Server

The Network Server is the process that is responsible for driving the Ethernet hardware. It provides facilities that allow multiple client processes to use the network concurrently. The client interface to the Net Server is found in Net10MBUser in LibPascal.

3.6. Message Server

The Message Server is the process that is responsible for extending the IPC facilities across the network. In addition the Message Server provides a simple name service facility.

3.6.1. Name service

The only client interface to the Message Server is for using the Naming Services. The client interface to the name server portions of the message server are found in the file MsgNUser.Pas in LibPascal.

The interface to the Name Server is very simple:

1. A process can register a port with a given name.
2. A process can request the port associated with a name.
3. A process can destroy the name/port connection. In addition a call is provided to return status information associated with a port.

3.6.2. Network IPC operation

The major function provided by the Message Server is to extend the IPC system across the network. An example will be used to show the manner in which this is done.

3.6.2.1. Finding a name

1. Process A on Machine A executes a LookUp. The LookUp is for a port provided by Process B on Machine B.
2. The Message Server on Machine A notices that it does not have an entry for Process B in its name space.
3. The Message Server on Machine A sends a broadcast message on the network asking all other message servers if they have an entry for name B.
4. The Message Server on Machine B receives the message and determines that it does have an entry for name B.
5. It will reply to the request from Message Server A saying "I have a port for B. Here is an ID that you can use to talk to me about B."
6. The Message Server on Machine A receives the message from the Message Server on B. It then allocates a port, A', that it associates with the ID provided by the Message Server on Machine B.
7. Finally the message server on Machine A returns the port A' to Process A.

Note that Process A does not know that the port A' is a port to the Message Server and not to Process B. The fact that Process B is on another machine is transparent to Process A.

3.6.2.2. Sending a message

1. Process A wants to send a message to Process B. It executes a Send command using Port A' as the destination.
2. Message Server A receives the message on port A'. It looks in its internal tables and notices that port A' is associated with a process on another machine.
3. Message Server A sends the message, using the appropriate network protocol, to Message Server B.
4. Message Server B looks at the ID provided by Message Server A and finds the port on Machine B that is associated with that ID.
5. Message Server B then sends the message to Process B on the port that Process B supplied during a CheckIn command.

3.6.2.3. Deleting ports

When a remote port is deleted, the following actions take place:

1. The owner of the Port B deletes it.
2. The Message Server B gets a Port Death message.
3. Message Server B sends a message to Message Server A, telling it that Port B has died.
4. Message Server A deletes the port A'.
5. Process A gets a Port died message for Port A'. The semantics of the Port Deleted processing is exactly the same as if the port were on the local machine. This is because, in fact, it is a local port that is deleted.

3.7. Time Server

The Time Server provides all of the time of day facilities provided by the operating system. It can return the time and date in a number of different formats. The interface to this server is found in the file TimeUser.Pas in LibPascal.

3.8. I/O System

The I/O system provides access to all I/O devices except the disk and the network. There is a single standard interface to all of the devices. The way in which a client picks the device to be accessed is by sending the requests to the appropriate server. There is one server for each of the supported I/O devices. The interface to the I/O servers is found in IOUser.Pas in LibPascal.

4. Accent Environments

Accent provides a number of environments for the user of the system. Each of the environments provides its own user interface, command line processing, error processing, and style of operation. All of the environments make use of the virtual machine provided by the Accent kernel. In addition, all of the environments make use of the input and output facilities provided by the Accent window manager.

Any number of these environments may be active at any given time. In addition, it is possible for processes running in one environment to communicate with processes in other environments using the Accent IPC facilities.

At the current time Accent provides its user with three different environments. They are: a) Native Accent, b) Qnix, an implementation of the UNIX System V operating system, and c) AI workstation environment running Accent Lisp.

4.1. Native Accent Environment

The native Accent environment is started when the operating system is booted. The user interface to this environment is the Accent Shell. This Shell provides command processing, as well as a number of simple commands, for the Accent environment. Most commands used in this environment are interactive.

4.2. Qnix

Qnix is an emulator that provides a UNIX System V system call interface. This interface allows most UNIX software programs to be ported to run on the PERQ workstation by simply recompiling the source.

The user interface to the Qnix system is provided by the UNIX software shell. This provides a well known environment to a large number of users. There are very few differences between the Qnix environment and the environment provided by the UNIX System V operating system.

It is possible to run native Accent programs from the Qnix environment in the same manner as running Qnix programs. The only limitation is that it is not possible to pipe between the two environments.

To start the Qnix environment execute the command file QnixInit.Cmd.

4.3. AI Workstation

The AI workstation environment is provided by running Accent Lisp on the machine. AccentLisp is a superset of Common Lisp.

Lisp is a programming language widely used for Artificial Intelligence research. It was conceived by John McCarthy in 1958. Because of its built-in facilities for symbol processing and its interactive programming environment, the language is increasingly being used for such applications as compilers, CAD systems, and editors.

Common Lisp is a new dialect of Lisp, closely related to MacLisp, Lisp Machine Lisp (Zetalisp), and (somewhat less closely) Franz Lisp. It was developed jointly by several Lisp

groups to meet the need for a modern Lisp dialect that is stable, well-documented, and suitable for implementation on a variety of machines.

AccentLisp is the implementation of the Common Lisp language for the PERQ workstation. It was developed by the Spice Lisp Group at Carnegie-Mellon University; it is nearly identical to the Common Lisp implementations on VAX/VMS and VAX/UNIX.

The AccentLisp release includes Hemlock, an editor written in Common Lisp. It is based on TOPS-20 EMACS, an editor written by Richard M. Stallman of the Massachusetts Institute of Technology.

5. Programming Examples

This chapter gives a number of example programs. These programs, in general, do not do anything useful. They do, however, show how to use a number of the facilities of the Accent operating system.

5.1. Messages

Below are two programs that send and receive IPC messages.

The first program, Message1, sends and receives both simple and complex messages. The program performs the following functions:

1. Define types that can be used for messages.
2. Allocate ports.
3. Register a port/name pair with the name server.
4. Present a name to the name server and receive a port.
5. Send a simple message.
6. Receive a simple message.
7. Send a complex message.
8. Receive a complex message.

The second program, Message2, is slightly different than Message1. In Message2 the program will be interrupted when a message arrives. The program will not do a receive until the message is there. This method of interaction allows the program to perform other computations while waiting for a message.

Generally, programs do not do actual IPC sends and receives. Most interprocess interaction is done using Remote Procedure

Call interfaces that were generated by Matchmaker. In these cases it is the Matchmaker code that does the sends and receives. Using Matchmaker interfaces is the preferred way of doing multiple process programs.

```
program Message1;

{-------------------------------------------------
{
{ Abstract:
{    This program demonstrates the use of the message system.
{    It will send and receive both "simple" and "complex"
{    messages.
{
{ Written by: Don Scelza
{
{ Copyright (C) PERQ Systems Corp., 1984
{
{-------------------------------------------------}

imports PascalInit   from PascalInit;  { Many useful ports are
                                           defined here. }
imports MsgN         from MsgNUser;    { For access to the
                                           name server }
imports AccCall      from AccCall;     { For Send and Receive }
imports AccInt       from AccentUser;  { AllocatePort }
imports AccentType   from AccentType;  { GR values }
imports SaltError    from SaltError;   { Error routines }
imports CommandDefs  from CommandDefs; { Get the value for
                                           ErAnyError. }

type
    DataArray  = array[0..10] of integer;
    pDataArray = ^DataArray;

{
{ A message has the following form:
{
{    1) There is a message header.  This defines the entire
{       message to the system.
{
{    2) The header is followed by zero or more objects.  An
{       object is made up of two parts.
{
{       a) A descriptor.  This defines all of the information
{          about a single object.
{
{       b) The data for the object.
{}

    MySimpleMessage = record     { A simple message does }
                                 { not contain any ports }
                                 { or pointers to "out of }
        Head:  msg;              { line" data. }
        MType: TypeType;         { A descriptor for "Data" }
        Data:  DataArray;        { The actual data to be }
                                 { passed. }
        end;

    pMySimpleMessage = ^MySimpleMessage;
```

```
    MyComplexMessage =  record  { A complex message }
                                { may contain ports or }
                                { "out of line" data. }
        Head:  msg;             { In this case the }
        MType: TypeType;        { Kernel may have to }
        Data:  pDataArray;      { translate the Ports or }
        end;                    { make new memory map
                                { entries for the out }
                                { of line data }

    pMyComplexMessage = ^MyComplexMessage;

var
    MyPort:             Port;   { We will do receives on
                                      this port. }
    MsgExamplePort:     Port;   { We will do sends on
                                      this port. }
```

```
const
    MyName          = 'MsgExample'; { Just a name to use }
    PortBackLog     = 3;  { Number of messages that may }
                          { be queued on a port }

    SimpleIndex     = 12345;  { An id for a simple }
                              { message }
    ComplexIndex    = 54321;  { An id for a complex }
                              { message }
```

```
procedure RegisterName;
{-----------------------------------------------}
{
{ Abstract:
{    Register a named port with the Name Server.
{
{ Design:
{    The port that we will register is MyPort.  It is
{    assumed that MyPort is a valid port at the time that
{    this procedure is called.  It is also assumed that the
{    interface to the NameServer has been initialized.
{
{-----------------------------------------------}
  var Gr: GeneralReturn;

    begin
    writeln('Register ', MyName, ' with the name server.');
    Gr := CheckIn(NameServerPort,{ NameServer from }
                                 { PascalInit }
                  MyName,        { The name for this port}
                  NullPort,      { Not used currently. }
                                 { Must be NullPort }
```

```
                MyPort);        { The port to give to }
                                { others }
    if Gr <> Success then
      begin
      GRWriteStdError(Gr, GR_FatalError, 'Could not "CheckIn"');
      end;
    end;

procedure LookUpMyName;
{-------------------------------------------------
{
{ Abstract:
{    Lookup the port name MyName.  Place the port returned
{    by the NameServer into MsgExamplePort.
{
{-------------------------------------------------}
  var Gr: GeneralReturn;

    begin
    writeln('Lookup ', MyName, ' from the name server.');
    Gr := LookUp(NameServerPort,     { NameServer from }
                                     { PascalInit }
                 MyName,             { The name that we }
                                     { want to find. }
                 MsgExamplePort);    { Will hold the desired }
                                     { port }
    if Gr <> Success then
      begin
      GRWriteStdError(Gr, GR_FatalError, 'Could not "LookUp"');
      end;
    end;


procedure AllocateMyPort;
{-------------------------------------------------
{
{ Abstract:
{    This procedure is used to allocate a new port.  The port
{    that is allocated will be placed into MyPort.
{
{-------------------------------------------------}
  var Gr: GeneralReturn;

    begin
    writeln('Allocate port.');
    Gr := AllocatePort(Kernelport,   { My port to the Kernel }
                       MyPort,       { Put the new port in }
                                     { MyPort }
                       PortBackLog); { Allow this many }
                                     { messages to be }
        { queued }
    if Gr <> Success then
        begin
        GRWriteStdError(Gr, GR_FatalError, 'Could not allocate
                         MyPort');
```

```
        end;
    end;

procedure SendSimple;
{-------------------------------------------------
{
{ Abstract:
{    Set-up and send a simple message on port MsgExamplePort
{
{-------------------------------------------------}
  var Gr: GeneralReturn;
      SendMyP: pMySimpleMessage;
      I: integer;

    begin
    writeln('Send a simple message.');
    new(SendMyP);                  { Allocate the message }

{
{ First set up the message header.  This describes the entire
{ message that is to be sent.
{}

    with SendMyP^.Head do
        begin
        SimpleMsg := true;
                    { This is a simple message.  No ports }
                    { or pointers }
        MsgType := NormalMsg;
                    { This is a normal, not emergancy, }
                    { message. }
        LocalPort := NullPort;
                    { The port for a reply.  We don't }
                    { expect to get one. }
        RemotePort := MyPort;
                    { Destination for the message }
        ID := SimpleIndex;
                    { An ID that the receiver can check }
        MsgSize := wordsize(MySimpleMessage) * 2;
        end;

{
{ Now set up the object descriptor for the data of the message.
{}


    with SendMyP^.MType do
        begin
        TypeName := TypeUnStructured;  { Just bits.  No }
                                       { ports or pointers }
        NumObjects := 1;               { Only one object }
                                       { in the message }
        InLine := true;       { The data is in the message. }
                              { No pointers }
        LongForm := false;    { This is a simple description }
        Deallocate := false;
```

```
                { Don't remove the message from the }
                { senders address space }
        TypeSizeInBits := wordsize(DataArray) * 16;
        end;

{
{ Put in some values for the data.
{}

    with SendMyP^ do
        begin
        for I := 0 to 10 do Data[I] := I + 23;
        end;

{
{ Now send the message
{}

    Gr := Send(SendMyP^.Head,       { The message header }
               0,                   { Don't time out }
               Wait);               { Wait for it to finish }

    if Gr <> Success then
        begin
        GRWriteStdError(Gr, GR_FatalError, 'Could
                        not Send SimpleMessage');
        end;

    dispose(SendMyP);
    end;


procedure SendComplex;
{-----------------------------------------
{
{ Abstract:
{    Set-up and send a Complex message on port
{    MsgExamplePort
{
{-----------------------------------------}
  var Gr: GeneralReturn;
      SendMyP: pMyComplexMessage;
      DataP: pDataArray;
      I: integer;

    begin
    writeln('Send a complex message.');
    new(SendMyP);               { Allocate the message }
    new(DataP);                 { Allocate the data memory }

{
{ First set up the message header.  This describes the entire
{ message that is to be sent.
{}
```

```
    with SendMyP^.Head do
      begin
      SimpleMsg := false;      { This is a complex message. }
      MsgType := NormalMsg;    { This is a normal message. }
      LocalPort := NullPort;   { The port for a reply.  We }
                               { don't expect to get one. }
      RemotePort := MyPort;    { Destination for the message }
      ID := ComplexIndex;      { An ID that the receiver }
                               { can check }
      MsgSize := wordsize(MyComplexMessage) * 2;
      end;

{
{ Now set up the object descriptor for the data of the message.
{}

    with SendMyP^.MType do
      begin
      TypeName := TypeUnStructured;
               { Just bits.  No ports }
                               { or pointers }
      NumObjects := 1;         { Only one object in }
                               { the message }
      InLine := false;         { The data is referenced }
                               { by a pointer }
      LongForm := false;       { This is a simple }
                               { description }
      Deallocate := false;     { Don't remove the message }
                               { from the senders address }
                               { space }
      TypeSizeInBits := wordsize(pointer) * 16;
      end;

{
{ Put in some values for the data.
{}

    for I := 0 to 10 do DataP^[I] := I + 23;
    SendMyP^.Data := DataP;

{
{ Now send the message
{}

    Gr := Send(SendMyP^.Head,      { The message header }
               0,                  { Don't time out }
               Wait);              { Wait for it to finish }

    if Gr <> Success then
        begin
        GRWriteStdError(Gr, GR_FatalError, 'Could not
                        Send ComplexMessage');
        end;

    dispose(SendMyP);
```

```
    dispose(DataP);
    end;


procedure ReceiveSimpleMsg;
{-------------------------------------------------------
{
{ Abstract:
{    This procedure will receive a message.  It will check the
{    ID of the message to see if it was one that was expected.
{
{-------------------------------------------------------}
  var Gr: GeneralReturn;
      SimpleP: pMySimpleMessage;

    begin
    writeln('Receive a simple message.');
    new(SimpleP);
    SimplePˆ.Head.MsgSize := wordsize(MySimpleMessage) * 2;
    Gr := Receive(SimplePˆ.Head,  { Receive the message into }
                                  { SimpleP. }
                  0,              { Don't time out.  Just }
                                  { wait. }
                  AllPts,         { Receive any messages  }
                                  { on any ports. }
                  ReceiveIt);     { Actually receive the }
                                  { message. }
    if Gr <> Success then
        begin
        GRWriteStdError(Gr, GR_FatalError,
           'Could not receive a simple message.');
        end;

    if SimplePˆ.Head.ID <> SimpleIndex then
        begin
        GRWriteStdError(ErAnyError, GR_FatalError,
           'Message was not a simple message. ');
        end;
    end;


procedure ReceiveComplexMsg;
{-------------------------------------------------------
{
{ Abstract:
{    This procedure will receive a message.  It will check the
{    ID of the message to see if it was one that was expected.
{
{-------------------------------------------------------}
  var Gr: GeneralReturn;
      ComplexP: pMyComplexMessage;

    begin
    writeln('Receive a complex message.');
    new(ComplexP);
```

```
    ComplexP^.Head.MsgSize := wordsize(MySimpleMessage) * 2;
    Gr := Receive(ComplexP^.Head,{ Receive the message }
                                  { into ComplexP. }
                  0,              { Don't time out.  Just }
                                  { wait. }
                  ALlPts,         { Receive any messages }
                                  { on any ports. }
                  ReceiveIt);     { Actually receive the }
                                  { message. }
    if Gr <> Success then
        begin
        GRWriteStdError(Gr, GR_FatalError,
           'Could not receive a complex message.');
        end;

    if ComplexP^.Head.ID <> ComplexIndex then
        begin
        GRWriteStdError(ErAnyError, GR_FatalError,
           'Message was not a complex message. ');
        end;
    end;


begin


{
{ Allocate a new port.  Register that port with the NameServer.
{ LookUp the port and get the Send rights to it.
{
{ Actually we already have all of the rights to the port.
{ In addition the values of MyPort and MsgExamplePort will be
{ the same because this is a single process.  If the call to
{ LookUpMyName was in a different process the values of MyPort
{ and MsgExamplePort might be different.  In that case the
{ process that did the LookUpMyName would have send rights to
{ MsgExamplePort.  It would not have receive rights to it.
{}

AllocateMyPort;
RegisterName;
LookUpMyName;

{
{ Send a simple message
{}

SendSimple;

{
{ Receive a message.  We know that there is one there because
{ we just sent it.
{}

ReceiveSimpleMsg;
```

```
{
{ Send a complex message
{}

SendComplex;

{
{ Receive a message.  We know that there is one there because
{ we just sent it.
{}

ReceiveComplexMsg;

end.
```

```
program Message2;

{-------------------------------------------------------------
{
{ Abstract:
{    This program demonstrates the use of the message system.
{    It will send and receive a simple message using the
{    software interrupt facility to be notified when a message
{    is available.
{
{ Written by: Don Scelza
{
{ Copyright (C) PERQ Systems Corp., 1984
{
{-------------------------------------------------------------}

imports PascalInit   from PascalInit;  { Many useful ports }
                                       { are defined here. }
imports MsgN         from MsgNUser;    { For access to the }
                                       { name server }
imports AccCall      from AccCall;     { For Send and Receive }
imports AccInt       from AccentUser;  { AllocatePort }
imports AccentType   from AccentType;  { GR values }
imports SaltError    from SaltError;   { Error routines }
imports CommandDefs  from CommandDefs; { Get the value for }
                                       { ErAnyError. }
imports Except       from Except;      { For Message  }
                                       { exceptions }

type
    DataArray  = array[0..10] of integer;

{
{ A message has the following form:
{
{    1) There is a message header.  This defines the entire
{       message to the system.
{
{    2) The header is followed by zero or more objects.  An
{       object is made up of two parts.
{
{       a) A descriptor.  This defines all of the information
{          about a single object.
{
{       b) The data for the object.
{}


    MySimpleMessage = record   { A simple message does }
                               { not contain any }
                               { ports or pointers to }
        Head:  msg;            { "out of line" data. }
        MType: TypeType;       { A descriptor for "Data"}
        Data:  DataArray;      { The actual data to be }
                               { passed. }
        end;
```

```
    pMySimpleMessage = ^MySimpleMessage;


var
    MyPort:             Port;      { We will do receives on }
                                   { this port. }
    MsgExamplePort:     Port;      { We will do sends on }
                                   { this port. }
    NoMessage:          boolean;   { True if waiting for }
                                   { a message }
    BoolDum:            boolean;   { A dummy }
    Gr:                 GeneralReturn;




const
    MyName          = 'MsgExample'; { Just a name to use }
    PortBackLog     = 3;    { Number of messages that may }
                            { be queued on a port }
    SimpleIndex     = 12345;  { An id for a simple message }


procedure RegisterName;
{-------------------------------------------------
{
{ Abstract:
{    Register a named port with the Name Server.
{
{ Design:
{    The port that we will register is MyPort.  It is
{    assumed that MyPort is a valid port at the time that
{    this procedure is called.  It is also assumed that the
{    interface to the NameServer has been initialized.
{
{-------------------------------------------------}
  var Gr: GeneralReturn;

    begin
    writeln('Register ', MyName, ' with the name server.');
    Gr := CheckIn(NameServerPort,  { NameServer from }
                                   { PascalInit }
                  MyName,          { The name for this }
                                   { port}
                  NullPort,        { Not used currently. }
                                   { Must be NullPort }
                  MyPort);         { The port to give to }
                                   { others }
    if Gr <> Success then
      begin
      GRWriteStdError(Gr, GR_FatalError, 'Could not "CheckIn"');
      end;
    end;
```

```
procedure LookUpMyName;
{-------------------------------------------------------
{
{ Abstract:
{    Lookup the port name MyName.  Place the port returned
{    by the NameServer into MsgExamplePort.
{
{-------------------------------------------------------}
  var Gr: GeneralReturn;

    begin
    writeln('Lookup ', MyName, ' from the name server.');
    Gr := LookUp(NameServerPort,     { NameServer from }
                                     { PascalInit }
                 MyName,             { The name that we }
                                     { want to find. }
                 MsgExamplePort);    { Will hold the desired }
                                     { port }
    if Gr <> Success then
      begin
      GRWriteStdError(Gr, GR_FatalError, 'Could not "LookUp"');
      end;
    end;


procedure AllocateMyPort;
{-----------------------------------------------
{
{ Abstract:
{    This procedure is used to allocate a new port.  The port
{    that is allocated will be placed into MyPort.
{
{-----------------------------------------------}
  var Gr: GeneralReturn;

    begin
    writeln('Allocate port.');
    Gr := AllocatePort(Kernelport,   { My port to the Kernel }
                       MyPort,       { Put the new port in }
                                     { MyPort }
                       PortBackLog);{ Allow this many }
                                     { messages to be }
       { queued }
    if Gr <> Success then
        begin
        GRWriteStdError(Gr, GR_FatalError, 'Could not
                           allocate MyPort');
        end;
    end;

procedure SendSimple;
{-----------------------------------------
{
{ Abstract:
{    Set-up and send a simple message on port MsgExamplePort
```

```
{
{-------------------------------------------}
  var Gr: GeneralReturn;
      SendMyP: pMySimpleMessage;
      I: integer;

    begin
    writeln('Send a simple message.');
    new(SendMyP);                { Allocate the message }

{
{ First set up the message header.  This describes the entire
{ message that is to be sent.
{}

    with SendMyP^.Head do
        begin
        SimpleMsg := true;{ This is a simple message. }
                          { No ports or pointers }
        MsgType := EmergencyMsg; { Send an Emergency }
                          { message to be different. }
        LocalPort := NullPort; { The port for a reply.  We }
                          { don't expect to get one. }
        RemotePort := MyPort;  { Destination for the message }
        ID := SimpleIndex;{ An ID that the receiver can }
                          { check }
        MsgSize := wordsize(MySimpleMessage) * 2;
        end;

{
{ Now set up the object descriptor for the data of the message.
{}


    with SendMyP^.MType do
        begin
        TypeName := TypeUnStructured;
        { Just bits.  No ports }
                              { or pointers }
        NumObjects := 1;      { Only one object in
        { the message }
        InLine := true;       { The data is in the message. }
                              { No pointers }
        LongForm := false;    { This is a simple description }
        Deallocate := false;  { Don't remove the message from }
                              { the senders address space }
        TypeSizeInBits := wordsize(DataArray) * 16;
        end;

{
{ Put in some values for the data.
{}

    with SendMyP^ do
        begin
        for I := 0 to 10 do Data[I] := I + 23;
```

```
        end;

{
{ Now send the message
{}

    Gr := Send(SendMyP^.Head,       { The message header }
               0,                   { Don't time out }
               Wait);               { Wait for it to finish }

    if Gr <> Success then
        begin
        GRWriteStdError(Gr, GR_FatalError, 'Could not
                            Send SimpleMessage');
        end;

    dispose(SendMyP);
    end;



procedure ReceiveSimpleMsg;
{-------------------------------------------------
{
{ Abstract:
{    This procedure will receive a message.  It will check the
{    ID of the message to see if it was one that was expected.
{
{-------------------------------------------------}

  var Gr: GeneralReturn;
      SimpleP: pMySimpleMessage;

    begin
    writeln('Receive a simple message.');
    new(SimpleP);
    SimpleP^.Head.MsgSize := wordsize(MySimpleMessage) * 2;
    Gr := Receive(SimpleP^.Head,  { Receive the message into }
                                  { SimpleP. }
                  0,              { Don't time out.  Just }
                                  { wait. }
                  AllPts,         { Receive any messages }
                                  { on any ports. }
                  ReceiveIt);     { Actually receive the }
                                  { message. }
    if Gr <> Success then
        begin
        GRWriteStdError(Gr, GR_FatalError,
           'Could not receive a simple message.');
        end;

    if SimpleP^.Head.ID <> SimpleIndex then
        begin
        GRWriteStdError(ErAnyError, GR_FatalError,
           'Message was not a simple message. ');
```

```
        end;
    end;


handler EmergMsg;
{-------------------------------------------
{
{ Abstract:
{    This handler will be called when an emergency message is
{    received.
{
{-------------------------------------------}
    begin
    writeln('Enter message exception handler.');
    ReceiveSimpleMsg;
    NoMessage := false;
    end;


begin

{
{ Allocate a new port.  Register that port with the NameServer.
{ LookUp the port and get the Send rights to it.
{
{ Actually we already have all of the rights to the port.
{ In addition the values of MyPort and MsgExamplePort will be
{ the same because this is a single process.  If the call to
{ LookUpMyName was in a different process the values of MyPort
{ and MsgExamplePort might be different.  In that case the
{ process that did the LookUpMyName would have send rights to
{ MsgExamplePort.  It would not have receive rights to it.
{}

AllocateMyPort;
RegisterName;
LookUpMyName;

NoMessage := true;

{
{ Tell the system to interrupt the process when an
{ emergency message is received.  When the message
{ is ready an exception is raised.
{}

BoolDum := true;
Gr := SoftInterrupt(KernelPort,    { Send the request to the }
                                   { Kernel }
                    false,         { Allow exceptions on }
                                   { emergency messages }
                    BoolDum);      { Turn them on. Return }
                                   { the old value in }
      { BoolDum }
```

```
{
{ Send a simple message
{}

SendSimple;

{
{ Now process until a message arrives.
{}

while NoMessage do ;

end.
```

5.2. Matchmaker

Matchmaker is a language for defining remote procedure calls. Remote procedure calls are message-based interfaces with declarations. These interfaces are generally between multiple processes in a multiple-language programming environment. Once an interface between two processes has been declared, then Matchmaker can be used to generate procedures for sending and receiving messages. Both processes can understand the messages even though the processes may be written in different languages. Matchmaker does all the work of appropriately packing the procedure arguments into messages and extracting message fields for incoming procedure arguments.

Matchmaker allows a programmer to write a server process and declare the types of all data to be exchanged between the server and its user processes. Procedural interfaces can be declared based on those types for sending data between processes in messages. When these procedures are implemented they can begin to send and receive messages. Remote procedure call interfaces simplify and increase the reliability of writing message based code.

Examples are given in the document "Matchmaker: The Accent Remote Procedure Call Language" in the *Accent Languages Manual*.

5.3. Graphics

This section provides two simple graphics programs.

The first, Graphics1, performs some simple graphics operations. It draws lines and does rasterops.

The second program, Graphics2, shows how to enable and deal with notification that the size or coveredness of a window has

changed.

```
program Graphics1;

{-------------------------------------------------------------
{
{ Abstract:
{    This program demonstrates some simple uses of the window
{    manager.  The program does all operations in the window
{    from which it was started.  This program will NOT handle
{    any of the emergency messages that the window manager
{    provides to the client when a viewport changes.
{
{ Written by: Don Scelza
{
{ Copyright (C) PERQ Systems Corp., 1984
{
{-------------------------------------------------------------}

imports PascalInit   from PascalInit;  { Many useful ports }
                                       { are defined here. }
imports AccentType   from AccentType;  { GR values }
imports SaltError    from SaltError;   { Error routines }
imports CommandDefs  from CommandDefs; { Get the value for }
                                       { ErAnyError. }

{
{ Get the interface to the Window Manager.
{}

imports ViewPt            from ViewPtUser;
imports Sapph             from SapphUser;
imports ViewKern          from ViewKern;
imports WindowUtils       from WindowUtils;

var
    MaxX:       integer;         { Max X coordinate }
    MaxY:       integer;         { Max Y coordinate }
    VP:         ViewPort;        { Inner ViewPort of }
                                 { the user window. }



procedure GetViewPortInfo;
{-------------------------------------------------------------
{
{ Abstract:
{    Get the information needed to draw in the current window.
{    UserWindow is a variable in PascalInit.  It is the window
{    that the program was run in.  From this we can get the
{    information about the inner viewport of that window.  This
{    information includes:
{
{        VP:    The InnerViewPort
{        MaxX:  The MaxX point of the inner viewport.
{        MaxY:  The MaxY point of the inner viewport.
{
```

```
{-------------------------------------------------}
    begin
    WindowViewPort(UserWindow, VP, MaxX, MaxY);
    end;


procedure Clear;
{-------------------------------------------------
{
{ Abstract:
{    This procedure will clear the inner viewport of the
{    user window.
{
{-------------------------------------------------}
    begin
    VPColorRect(VP, RectWhite, 0, 0, MaxX, MaxY);
    end;

procedure WaitForCr;
{-------------------------------------------------
{
{ Abstract:
{    Set the window attention flag and wait for a <cr>.
{
{-------------------------------------------------}
  var S: string;

    begin
    ShowWindowAttentionFlag;              { Turn on the "!". }
    readln(S);
    RemoveWindowAttentionFlag;
    end;



procedure DrawLines;
{-------------------------------------------------
{
{ Abstract:
{    Draw a set of rectangles.  One inside the other.
{
{-------------------------------------------------}
  var StartX, StartY, EndX, EndY: integer;
      Done: boolean;

    begin
    Done := false;
    StartX := 0;
    StartY := 0;
    EndX := MaxX;
    EndY := MaxY;
    while not Done do
        begin
        VPLine(VP, DrawLine, StartX, StartY, EndX,   StartY);
```

```
        VPLine(VP, DrawLine, StartX, EndY,   EndX,   EndY);
        VPLine(VP, DrawLine, StartX, StartY, StartX, EndY);
        VPLine(VP, DrawLine, EndX,   StartY, EndX,   EndY);
        StartX := StartX + 4;
        StartY := StartY + 4;
        EndX := EndX - 4;
        EndY := EndY - 4;
        if (StartX >= EndX) or (StartY >= EndY)
           then Done := true;
        end;
    end;

procedure RasterOps;
{-------------------------------------------------
{
{ Abstract:
{    This procedure will do a set of rasterops. The
{    algorithm is the same as DrawLines.
{
{-------------------------------------------------}
  var StartX, StartY, EndX, EndY: integer;
      Done: boolean;

    begin
    Done := false;
    StartX := 0;
    StartY := 0;
    EndX := MaxX;
    EndY := MaxY;
    while not Done do
        begin
        VPRop(VP,    { Source ViewPort for the rasterop }
              RNot,          { Use a Not function }
              StartX,        { Start X and Y locations }
              StartY,
              EndX - StartX, { width and height }
              EndY - StartY,
              VP,      { Dest ViewPort for the rasterop }
              StartX,        { Start X and Y locations }
              StartY);
        StartX := StartX + 4;
        StartY := StartY + 4;
        EndX := EndX - 4;
        EndY := EndY - 4;
        if (StartX >= EndX) or (StartY >= EndY)
           then Done := true;
        end;
    end;

begin


GetViewPortInfo;  { Get the information about the viewport. }
Clear;

writeln('This is a simple graphics test program.');
```

```
writeln('It runs a set of tests and then waits for the user to');
writeln('Type a <cr>.  When the attention flag in the window');
writeln('icon is set type <cr> to go onto the next display.');

WaitForCr;

Clear;                                  { Clear the screen }
DrawLines;                              { Draw some lines }
WaitForCr;                              { Wait to go on. }

Clear;
RasterOps;
WaitForCr;

end.
```

```
program Graphics2;

{-------------------------------------------------------------
{
{ Abstract:
{   This program demonstrates some simple uses of the window
{   manager.  The program does all operations in the window
{   from which it was started.
{
{   This version of the program has the code needed to handle
{   the messages from the window manager that signify changes
{   to the state of the window.
{
{ Written by: Don Scelza
{
{ Copyright (C) PERQ Systems Corp., 1984
{
{-------------------------------------------------------------}

imports PascalInit   from PascalInit;  { Many useful ports }
                                       { defined here. }
imports AccentType   from AccentType;  { GR values }
imports SaltError    from SaltError;   { Error routines }
imports CommandDefs  from CommandDefs; { Get the value for }
                                       { ErAnyError. }

{
{ Get the interfaces needed to receive messages.
{}

imports Except       from Except;  { For Message exceptions }
imports AccCall      from AccCall; { For Send and Receive }

{
{ Get the interface to the Window Manager.
{}

imports ViewPt       from ViewPtUser;
imports Sapph        from SapphUser;
imports ViewKern     from ViewKern;
imports WindowUtils  from WindowUtils;

{
{ Get the portions of the Window manager that deal with
{ window state change notification.
{}

imports SaphEmrServer     from SaphEmrServer;
imports SaphEmrExceptions from SaphEmrExceptions;


var
    MaxX:       integer;          { Max X coordinate }
    MaxY:       integer;          { May Y coordinate }
    VP:         ViewPort;         { Inner VierPort of }
                                  { the user window. }
```

```
    BoolDum:    boolean;
    Gr:         GeneralReturn;



procedure GetViewPortInfo;
{-------------------------------------------------
{
{ Abstract:
{    Get the information needed to draw in the current window.
{    UserWindow is a variable in PascalInit.  It is the window
{    that the program was run in.  From this we can get the
{    information about the inner viewport of that window.  This
{    information includes:
{
{        VP:    The InnerViewPort
{        MaxX:  The MaxX point of the inner viewport.
{        MaxY:  The MaxY point of the inner viewport.
{
{-------------------------------------------------}
    begin
    WindowViewPort(UserWindow, VP, MaxX, MaxY);
    end;



procedure Clear;
{-------------------------------------------------
{
{ Abstract:
{    This procedure will clear the inner viewport of the
{    user window.
{
{-------------------------------------------------}
    begin
    VPColorRect(VP, RectWhite, 0, 0, MaxX, MaxY);
    end;


procedure WaitForCr;
{-------------------------------------------------
{
{ Abstract:
{    Set the window attention flag and wait for a <cr>.
{
{-------------------------------------------------}
  var S: string;

    begin
    ShowWindowAttentionFlag;              { Turn on the "!". }
    readln(S);
    RemoveWindowAttentionFlag;
    end;
```

```
procedure DrawLines;
{-------------------------------------------------
{
{ Abstract:
{    Draw a set of rectangles.  One inside the other.
{
{-------------------------------------------------}
  var StartX, StartY, EndX, EndY: integer;
      Done: boolean;

    begin
    Done := false;
    StartX := 0;
    StartY := 0;
    EndX := MaxX;
    EndY := MaxY;
    while not Done do
        begin
        VPLine(VP, DrawLine, StartX, StartY, EndX,   StartY);
        VPLine(VP, DrawLine, StartX, EndY,   EndX,   EndY);
        VPLine(VP, DrawLine, StartX, StartY, StartX, EndY);
        VPLine(VP, DrawLine, EndX,   StartY, EndX,   EndY);
        StartX := StartX + 4;
        StartY := StartY + 4;
        EndX := EndX - 4;
        EndY := EndY - 4;
        if (StartX >= EndX) or (StartY >= EndY)
            then Done := true;
        end;
    end;


handler EmergMsg;
{-----------------------------------------------
{
{ Abstract:
{    This handler will be called when an emergency message
{    is received.  The code will receive the message and then
{    pass it to the Client code of the Window Manager.  If
{    the message was a window manager message,  the code
{    will raise the correct exception for the type of change
{    that happened to the window.
{
{-----------------------------------------------}
  const MaxMsgSize = 2048;
  type space = array[0..MaxMsgSize div 2 - 1] of integer;
       pDummyMsg = ^DummyMsg;
       DummyMsg = record
                  head: msg;
                  RetType: typetype;
                  RetCode: integer;
                  body: space;
                  end;

  var Gr: GeneralReturn;
      pReply, pMsg: pDummyMsg;
```

```
    begin

{
{ Receive the message.
{}

    new(pReply);
    new(pMsg);
    pMsg^.Head.MsgSize := MaxMsgSize * 2;
    Gr := Receive(pMsg^.Head, 0, AllPts, ReceiveIt);

{
{ Call the Window Manager.  If the message was a Window
{ Manager message an exception will be raised and the call
{ will return true.  If it was not a Window Manager message
{ no exception will be raised and the call will return
{ false.
{}

    if  SaphEmrServer(pMsg, pReply) then
       begin
       end;
{
{ Turn on interrupts on messages receive again.
{}

    dispose(pReply);
    dispose(pMsg);
    Gr :=  SoftEnable(false, true);
    end;


handler EViewPtChanged(VP: ViewPort; X1, Y1,
                       W1, H1, R: integer);
{-----------------------------------------------
{
{ Abstract:
{    This exception is raised if the size of the viewport changes.
{    We will get the new size parameters and redisplay the lines.
{
{-----------------------------------------------}
    begin
    GetViewPortInfo;
    Clear;
    DrawLines;
    end;

handler EViewPtExposed(VP: ViewPort; RA: pRectArray;
                       NumRectangles: long);
{-----------------------------------------------
{
{ Abstract:
{    This exception is raised if the coveredness of the viewport
{    changes.
{
{-----------------------------------------------}
```

```
    begin
    GetViewPortInfo;
    Clear;
    DrawLines;
    end;


begin


{
{ Tell the system to interrupt the process when an emergency
{ message is received.  When the message is ready an exception
{ is raised.
{}

BoolDum := true;
Gr := SoftInterrupt(KernelPort,     { Send the request to }
                                    { the Kernel }
                    false,          { Allow exceptions on }
                                    { emergency messages }
                    BoolDum);       { Turn them on. Return }
                                    { the old value in }
      { BoolDum }


GetViewPortInfo;  { Get the information about the viewport. }
Clear;

{
{ Tell the window manager to generate exceptions when a
{ window state change happens.
{}

EnableNotifyExceptions(VP, DataPort, true, true);


writeln('This is a simple graphics test program.');
writeln('It will draw a display and wait for the user to type');
writeln('a <cr>.  While waiting, if the state of the window');
writeln('changes the display will be redrawn.');

WaitForCr;

Clear;                              { Clear the screen }
DrawLines;                          { Draw some lines }
WaitForCr;                          { Wait to go on. }


end.
```

5.4. File System

The Accent file system presents a single level view of the memory hierarchy. Files are mapped into the process virtual address space. This means that access to portions of a file are the same as access to portions of memory. When a file is read the contents of that file are mapped into the address space. The file system returns a pointer that is the virtual address of the start of the data. It also supplies the number of bytes of data that were in the file.

File1 is a simple program that will read a file and treat the contents of that file as integers.

```
program File1;

{-------------------------------------------------------------
{
{ Abstract:
{    This program demonstrates the use of the file system.
{    Read a file and print the first 100 integers in it.
{
{
{ Written by: Don Scelza
{
{ Copyright (C) PERQ Systems Corp., 1984
{
{-------------------------------------------------------------}

imports PascalInit  from PascalInit;  { Many useful ports are }
                                      { defined here. }
imports AccentType  from AccentType;  { GR values }
imports SaltError   from SaltError;   { Error routines }
imports CommandDefs from CommandDefs; { Get the value for }
                                      { ErAnyError. }
imports PathName    from PathName;    { High level interface }
                                      { to the file sys. }



type
    IntArray = array[0..0] of integer;  { Used to access the }
                                        { file data }
    pIntArray = ^IntArray;

var ArrayPtr:   pIntArray;
    I:          integer;
    FName:      Path_Name;
    NumBytes:   long;
    LastIndex:  long;
    Gr:         GeneralReturn;


begin

{
{ First map the file into our address space.
{}

FName := 'File1.Pas';
Gr := ReadFile(FName, recast(ArrayPtr, File_Data), NumBytes);
if Gr <> Success then
    begin
    GRWriteStdError(Gr, GR_FatalError, 'File1.Pas');
    end;
writeln('Full name of the file found is ', FName);

{
{ Now look at the first 100 values.  If there are less than 100
```

```
{ integers in the file only look at the ones that are there.
{
{ We must turn off range checking to access the data using the
{ defined type.
{}


if NumBytes > 200 then
    LastIndex := 100
else
    LastIndex := shrink(NumBytes div 2);


{$r-}
for I := 1 to LastIndex do
    writeln(ArrayPtr^[I]);
{$r=}

end.
```

There are two commonly desired changes to the above program that are simple to make in the Accent file system. The differences are: a) the use of a specific search list and b) having the system look for files with specific extensions. Both of these facilities are provided by the same system call.

The call to ReadFile in the above example can be replaced by a call to the procedure ReadExtendedFile. This procedure allows the client to specify both a list of extensions and a specific search list.

The Extension list is specified as a string. It has the form:

'<ext>;<ext>;...<ext>'

Remember that if a file without an extension is acceptable you must specify a null <ext> as the first value. To look for files that have a .Pas or a .Defs or no extension, the string that should be

supplied is:

```
';.Pas;.Defs'
```

To specify a different search list you must supply the Environment Manager variable name that is the search list. The standard search list is named <Default>.

5.5. Process Management

This section provides a number of examples of creating and using processes.

The first program, ProcessSplit, does a simple process split. After a split, one of the processes is the parent and the other process is the child. The two processes are exact copies of each other. The Split call returns an indication of which process is the parent and which process is the child.

The second program, ProcessExec, does an Exec. The Exec system call starts a new process running a specific program. The new process is not a copy of the parent, as in simpleSplit. This program will Exec a number of copies of itself. Each copy will allocate a port and register it with the name server. Because all of these processes are descendants of each other, they are all in the same process control group. To end the chain of Execs when a new copy of ProcessExec finds a specific name, instead of doing another Exec, that process will tell the process manager to kill the entire group of processes.

```
program ProcessSplit;

{-------------------------------------------------------
{
{ Abstract:
{    This is a simple program that does a Split.  After the call
{    to split there will be two processes.  One process will be
{    the parent while the other process will be the child.
{
{    This program does a simple split and then both processes
{    exit. If desired it is possible for the parent process to
{    wait for the  child to die.  When the child dies the kernel
{    will send a message to the parent on its data port.  This
{    will be an emergency message.
{
{ Written by: Don Scelza
{
{ Copyright (C) PERQ Systems Corp., 1984
{
{-------------------------------------------------------}

imports PascalInit   from PascalInit;  { Many useful ports are }
                                       { defined here. }
imports AccCall      from AccCall;     { For Send and Receive }
imports AccentType   from AccentType;  { GR values }
imports SaltError    from SaltError;   { Error routines }
imports CommandDefs  from CommandDefs; { Get the value for }
                                       { ErAnyError. }
imports Spawn        from Spawn;       { For Split }


var
    CKPort:     port;
    CDPort:     port;
    Gr:         GeneralReturn;


procedure IAmTheChild;
{-------------------------------------------------------
{
{ Abstract:
{    This procedure will be executed by the Child process
{    after the split.
{
{-------------------------------------------------------}
    begin
    writeln('I am the child');
    end;

procedure IAmTheParent;
{-------------------------------------------------------
{
{ Abstract:
{    This procedure will be executed by the Parent process
{    after the split.
{
```

```
{-------------------------------------------------------}
    begin
    writeln('I am the parent');
    end;

begin
Gr := Split(CKPort, CDPort);
if Gr = IsChild then
    begin
    IAmTheChild;
    end
else
    begin
    IAmTheParent;
    end;
end.
```

```
program ProcessExec;
{-------------------------------------------------------------
{
{ Abstract:
{    This program does an Exec.  It will Exec itself a number of
{    times.  Each time the new process will register its Data Port
{    with the name server.  The first process will register a
{    name of "1". The second process will register a name of
{    "2". Each time a process starts it will lookup these
{    names.  If the name "1" is not registered that process
{    will register it. If the name is registered the process
{    will go on to the next name.
{
{    All of these processes will be in the same process control
{    group.  When a process starts and is able to find a port
{    with the name "2" that process will kill the entire process
{    group.
{
{ Written by: Don Scelza
{
{ Copyright (C) PERQ Systems Corp., 1984
{
{-------------------------------------------------------------}

imports PascalInit  from PascalInit;  { Many useful ports are }
                                      { defined here. }
imports AccCall     from AccCall;     { For Send and Receive }
imports AccentType  from AccentType;  { GR values }
imports SaltError   from SaltError;   { Error routines }
imports CommandDefs from CommandDefs; { Get the value for }
                                      { ErAnyError. }
imports Spawn       from Spawn;       { For Split }
imports MsgN        from MsgNUser;    { To register ports }
imports ProcMgr     from ProcMgrUser; { To kill this set of }
                                      { processes }


var
    CKPort:     port;
    CDPort:     port;
    Gr:         GeneralReturn;
    MyProcName: string;
    TestP:      port;
    HisCommand: CommandBlock;

type
    MySimpleMessage = record    { A simple message does }
                                { not contain any ports }
                                { or pointers to "out }
        Head:  msg;             { of line" data. }
        MType: TypeType;        { A descriptor for "Data"}
        Data:  integer;         { The actual data to be }
                                { passed. }
        end;

    pMySimpleMessage = ^MySimpleMessage;
```

```
function FindNextProcName: string;
{-------------------------------------------------------
{
{ Abstract:
{    This procedure will find the next name to register with
{    the name server.  The names that will be searched for are:
{        "1" then "2" then "3".
{
{ Results:
{    Return the next name to be used.
{
{-------------------------------------------------------}
  var Gr: GeneralReturn;
      P: port;

    begin
    writeln('Looking for "1"');
    Gr := LookUp(NameServerPort, '1', P);
    if Gr <> Success then
        begin
        FindNextProcName := '1';
        exit(FindNextProcName);
        end;

    writeln('Looking for "2"');
    Gr := LookUp(NameServerPort, '2',  P);
    if Gr <> Success then
        begin
        FindNextProcName := '2';
        exit(FindNextProcName);
        end;

    writeln('Looking for "3"');
    Gr := LookUp(NameServerPort,  '3',  P);
    if Gr <> Success then
        begin
        FindNextProcName := '3';
        exit(FindNextProcName);
        end;
    end;


procedure RegisterName(Who: string);
{-------------------------------------------
{
{ Abstract:
{    Register a named port with the Name Server.
{
{ Parameters:
{    Who is the name that we are to register.
{
{ Design:
{    The port that we will register is the TestP.
{
{-------------------------------------------}
  var Gr: GeneralReturn;
```

```
    begin
    Gr := CheckIn(NameServerPort, Who,  NullPort, TestP);
    if Gr <> Success then
      begin
      GRWriteStdError(Gr, GR_FatalError, 'Could not "CheckIn"');
      end;
    end;

procedure AllocateMyPort;
{-----------------------------------------------
{
{ Abstract:
{    This procedure is used to allocate a new port.  The port
{    that is allocated will be placed into TestP.
{
{-----------------------------------------------}
  var Gr: GeneralReturn;

    begin
    Gr := AllocatePort(Kernelport, TestP, 1);
    if Gr <> Success then
      begin
      GRWriteStdError(Gr, GR_FatalError, 'Could not
                          allocate TestP');
      end;
    end;


procedure WaitToDie;
{-----------------------------------------------
{
{ Abstract:
{    This procedure will wait for a message on our Test port.
{    Because no process will ever send a message there, we will
{    wait until a process group signal is generated.
{
{-----------------------------------------------}
  var Gr: GeneralReturn;
      SimpleP: pMySimpleMessage;

    begin
    new(SimpleP);
    SimpleP^.Head.MsgSize := wordsize(MySimpleMessage) * 2;
    SimpleP^.Head.LocalPort := TestP;
     { We only want messages }
                                  { that are on TestP }
    Gr := Receive(SimpleP^.Head, { Receive the message into }
                                  { SimpleP. }
                  0,              { Don't time out.  Just }
                                  { wait. }
                  LocalPt,        { Only messages on TestP }
                  ReceiveIt);     { Actually receive the }
                                  { message. }
    if Gr <> Success then
        begin
        GRWriteStdError(Gr, GR_FatalError,
```

```
            'Could not receive a simple message.');
        end;
    end;


begin

writeln('New process is looking for a name:');
MyProcName := FindNextProcName;  { Look for the next name }
writeln('My name is ', MyProcName);

AllocateMyPort;                  { Get the port to be }
                                 { registered. }

if MyProcName = '3' then         { "3" is the last name. }
                                 { Kill the group }
    begin
    writeln(MyProcName, ' is the last process. Killing the group');
    PMGroupSignal(PMPort, UserWindow, SigLevel1Abort);
    end
else
    begin

    writeln('Register ', MyProcName, ' with the name server.');
    RegisterName(MyProcName);

    writeln('Exec the next process.');
    Gr := Exec(CKPort, CDPort, 'Exec.Run', HisCommand);

    writeln(MyProcName, ' is waiting to die.');
    WaitToDie;
    end;
end.
```

5.6. Network

All requests for network services go through the NetServer. This process provides the device driver for the network hardware.

The example network program is a simple process that will perform network time service. When a time request is received on the network, this process will get the time from its local time server and respond with a network packet to the machine that made the request.

```
program NetTimeServer;

{-------------------------------------------------------------
{
{ Abstract:
{    This program will respond to time requests from the
{    network.  The time will be put into a string and returned
{    to the machine that made the request.
{
{    The time string will be in the form:
{    <dd>-<mmm>-<yyyy> <hh>:<mm>:<ss>.<mmm>
{
{   e.g.        22-Jun-1982 13:05:21.001
{
{ Written By Don Scelza
{
{ Copyright (C) PERQ Systems Corp., 1984
{
{
{-------------------------------------------------------------}


imports PascalInit  from PascalInit;  { Many useful ports are }
                                      { defined here. }
imports MsgN        from MsgNUser;    { For access to the }
                                      { name server }
imports AccCall     from AccCall;     { For Send and Receive }
imports AccInt      from AccentUser;  { AllocatePort }
imports AccentType  from AccentType;  { GR values }
imports SaltError   from SaltError;   { Error routines }
imports CommandDefs from CommandDefs; { Get the value for }
                                      { ErAnyError. }
imports Except      from Except;      { For Message }
                                      { exceptions }
imports Time        from TimeUser;    { Time Server }
                                      { interface }

{
{ Get the interface to the ethernet.
{}

imports Net10MBRecvServer from Net10MbRecvServer;
                                      { Routines for }
                                      { receiving packets. }
imports Net10MB     from Net10MBUser; { General inter- }
                                      { face routines }


const
    Request         = 1;  { Code for time server request }
    Rply            = 2;  { Code for time server reply }
    TimeLength      = 24; { Expect 24 bytes for time  }
    TimeReqType     = 7;  { Ethernet type field for a }
                          { time request. }
```

```
type

    pTimePacket = ^TimePacket;
    TimePacket  = packed record
                    Dest:               E10Address;
                    Src:                E10Address;
                    PType:              E10Type;
                    ReplyOrRequest:     0..255;
                    Dummy:              0..255;
                    TimeString:         string[Timelength];
                    end;

var
    NetServerPort: port;          { Port for making netserver }
                                  { requests. }
    PacketPort:    port;          { The Net Server will send }
                                  { packets to use on this }
    { port. }
    pRecvPacket:   pTimePacket;   { Pointer to the received }
                                  { packet }
    pSendPacket:   pTimePacket;   { Pointer to the send packet. }
    pNetMessage:   pE10Message;   { Used to hold messages }
                                  { from the Net Server }
    pRMessage:     pE10Message;   { Reply message for }
                                  { Net10MBRecvServer }
    Gr:            GeneralReturn;



procedure InitNetTimeServer;
{-----------------------------------------------
{
{ Abstract:
{    Initialize the network time server.
{    The following initialization will be done:
{       1) Initialize all server interfaces.
{       2) Find a request port for the Net Server.
{       3) Allocate a port that will be used to receive
{          messages from the Net Server
{       4) Allocate memory needed for Net Server messages and
{          requests.
{       5) Set up Address and Type fields in the packets.
{       6) Tell the Net Server the types of packets that we are
{          interested in.
{
{-----------------------------------------------}
  var Gr: GeneralReturn;

    begin

{
{ First initialize the interfaces to the system servers.
{}

    InitNet10MB(NullPort);      { Initialize the interface to the }
```

```
                                { Net Server }

{
{ Find a port to talk to the Net Server.
{}

    Gr := LookUp(NameServerPort,  { NameServer from }
                                  { PascalInit }
                 'EtherServer',   { The name of the Net }
                                  { Server }
                 NetServerport);  { Will hold the desired }
                                  { port }
    if Gr <> Success then
        begin
        GRWriteStdError(Gr, GR_FatalError, 'EtherServer');
        end;


{
{ Allocate a port that the Net Server will use to send us
{ packets that were received on the network.
{}

    Gr := AllocatePort(Kernelport,  { My port to the Kernel }
                       PacketPort,  { Put the new port in }
                                    { PacketPort }
                       10);     { Allow this many messages }
                                { to be queued }
    if Gr <> Success then
        begin
        GRWriteStdError(Gr, GR_FatalError, 'Could not
                            allocate PacketPort');
        end;

{
{ Allocate the memory needed to hold packets and messages
{}

    new(pSendPacket);
    new(pNetMessage);
    new(pRMessage);

{
{ Set up the Address and type fields in the SendPacket.
{
{ First make a request of the Net Server to return the Ethernet
{ address of this machine.
{}

    Gr := E10GetAdd(NetServerPort,      { Port for making Net }
                                        { Server requests. }
                    pSendPacket^.Src); { The Address will }
                                        { be put here. }

    if Gr <> E100k then
        begin
```

```
        GRWriteStdError(Gr, GR_FatalError, 'Could not get
                                Ethernet Address');
        end;

    pSendPacket^.PType := TimeReqtype;  { Time packets are }
                                        { of this type }
    pSendPacket^.ReplyOrRequest := Rply;

{
{ Now tell the Net Server the type of packets that we want to
{ receive.  When we tell the server also give it a port.  This
{ port will be used by the server to send us packets of the
{ desired type.
{}

    Gr := E10SetFilter(NetServerPort,{ Net Server request port }
                       PacketPort,   { The NetServer --> }
                                     { Client packet port }
                       TimeReqType); { Packets of this type. }
    if Gr <> E100k then
        begin
        GRWriteStdError(Gr, GR_FatalError,
           'Could not set the Ethernet filter' );
        end;
    end;


procedure ReplyWithTime;
{-------------------------------------------------------
{
{ Abstract:
{    Reply to a request with the current time.
{    The request that we are to reply to will be in the packet
{    pointed to by pRecvPacket.
{
{-------------------------------------------------------}
  var TForm: integer;
      T: string;

    begin

{
{ Get the time from the Time Server.
{}

    TForm := TF_Dashes;
    TForm := lor(TForm, TF_FullYear);
    TForm := lor(TForm, TF_MilliSeconds);
    T := GetStringTime(TimePort, TForm);
    writeln('Responding with ', T);

{
{ Fill in the fields of the packet.  The other fields were
{ set during initialization.
{}
```

```
    pSendPacket^.Dest := pRecvPacket^.Src;
    pSendPacket^.TimeString := T;

{
{ Now send the packet.  Remember that the number of bytes
{ given to the send is the total number of bytes in the packet.
{ This also includes the headers.  Not just the data portions.
{
{}

    Gr := El0Send(NetServerPort, recast(pSendPacket,
                                        pEl0Packet),
                  El0BytesInHeader + El0MinDataBytes);
    if Gr <> El00k then
        begin
        GRWriteStdError(Gr, GR_FatalError, 'Could not send
                                time packet');
        end;
    end;


handler El0Receive(ServPort: El0Port;  Buff: pEl0Packet;
                   NumBytes: long);
{-------------------------------------------------------------
{
{ Abstract:
{    This exception will be raised by Net10MBRecvServer when
{    a packet is received from the Net Server.
{
{    This handler will check to make sure that the packet was
{    a time request.  If so it will reply to the requestor with
{    the current time.
{
{ Parameters:
{    Buff is a pointer to the packet that was received.
{
{    NumBytes is the number of valid bytes of memory that are
{    pointed to by Buff.
{
{-------------------------------------------------------------}
    begin

    pRecvPacket := recast(Buff, pTimePacket);
    if pRecvPacket^.ReplyOrRequest = Request then
         begin
         writeln('Time request received');
         ReplyWithTime;
         end;

    Gr := InvalidateMemory(KernelPort, recast(Buff,
                           VirtualAddress), NumBytes);
    if Gr <> Success then
        begin
        GRWriteStdError(Gr, GR_FatalError, 'Could not Deallocate
                                memory');
        end;
```

```
    end;


begin
   ,
InitNetTimeServer;

{
{ Now we will just loop receiving messages on NetServerPort.
{ When a message comes in hand it to the Net1OMBAsync.
{ If the message was from the Net Server, Net1OMBAsync.
{ will raise an exception.
{}

while true do
    begin

{
{ Try to receive a message from the Net Server.
{}

    pNetMessage^.Head.MsgSize := wordsize(E1OMessage) * 2;
    Gr := Receive(pNetMessage^.Head, 0, ALlPts, ReceiveIt);

{
{ If the Gr was success we got a message.  Give it to
{ Net1OMBrecvServer.
{}

    if Gr = Success then
      if Net1OMBRecvServer(recast(pNetMessage, pointer),
                           recast(pRMessage, pointer)) then ;

    end;

end.
```

5.7. Memory

Memory allocation and deallocation in Accent is a simple matter. There are two routines provided by the system for these functions, ValidateMemory and InvalidateMemory.

ValidateMemory provides a portion of virtual memory of a specific size. The amount of memory provided is rounded up to a page boundry.

InvalidateMemory removes a portion of a process's virtual address space.

```
program Memoryl;

{-------------------------------------------------------
{
{ Abstract:
{    This program provides a simple example of memory
{    allocation and deallocation.
{
{ Written by: Don Scelza
{
{ Copyright (C) PERQ Systems Corp., 1984
{
{-------------------------------------------------------}

imports PascalInit  from PascalInit;  { Many useful ports are }
                                      { defined here. }
imports AccInt      from AccentUser;  { For Send and Receive }
imports AccentType  from AccentType;  { GR values }
imports SaltError   from SaltError;   { Error routines }

const
    ArraySize = 1024;          { Size of the array in bytes. }
    ArrayLIndex = ArraySize - 1;  { Last index of the array }
type
    Byte = 0..255;                      { A byte }
    ByteArray = packed array[0..ArrayLIndex] of Byte;
    pByteArray = ^ByteArray;

var
    Gr:         GeneralReturn;
    Ptr:        pByteArray;
    VA:         VirtualAddress;
    I:          integer;

begin

{
{ Allocate a portion of address space.
{ We will allocate 1024 bytes at any address the system
{ would like.  In addition allow any alignment.
{}

VA := 0;                            { No specific address }
                                    { required. }
Gr := ValidateMemory(KernelPort,    { Port for the server }
                     VA,            { Put the address of }
                                    { the memory here. }
                     ArraySize,
                     -1);       { Any alignment will do. }
if Gr <> Success then
    begin
    GRWriteStdError(Gr, GR_FatalError, 'Could not
                      deallocate memory');
    end;

Ptr := recast(VA, pByteArray);          { Put the VA into Ptr }
```

```
{
{ Now go through the memory and touch each byte.
{}

for I := 0 to ArrayLIndex do Ptr^[I] := (I mod 256);
for I := 0 to ArrayLIndex do writeln(Ptr^[I]);

{
{ Dispose of the memory allocated.
{}

Gr := InvalidateMemory(KernelPort, VA, ArraySize);
if Gr <> Success then
    begin
    GRWriteStdError(Gr, GR_FatalError, 'Could not deallocate
                        memory');
    end;
end.
```

Appendix A. Process Numbers

There are a number of processes that are created when the system is booted. Because their order is well-defined, they have well-defined process numbers. The numbers provided here are the operating system internal process numbers. They are not the process numbers that are given by the Details -Systat command.

Process		
1	Pager	Provides page access to the disk.
2	Time Server	Provides time for the system.
3	File System	Provides File level access to the disk.
4	Environment Manager	Provides SearchList and Variables.
5	BootUp	Dies after creating StartUp.
6	StartUp	Provides Shell and Login creation.
7	Window Manager	Provides graphic access to the display.
8	Tracker	Keyboard and Tablet driver.
9	TypeScript	Provides normal text refresh of windows.
10	Process Manager	Provides process control.
11	Message Server	Extends IPC across the Network.
12	Net Server	Network device server.

Appendix B. Registered System Ports

This appendix provides a list of the system ports that are registered with the name server.

["MachineName"]SesNetPort

Port for remote file access

NetServer

Access to Net Server

NameServerPort

Access to the Name Server

["MachineName"]RS232AServer

Access to the RS232 A

["MachineName"]RS232BServer

Access to the RS232 B

["MachineName"]GPIBServer

Access to the GPIB

["MachineName"]FloppyServer

Access to the Floppy

["MachineName"]SpeechServer

Access to the Speech output device

NOTE: "MachineName" is the name in the SysName file.

Appendix C. Standard Environment Variables

This appendix gives a list of the standard environment variables that are used by the system.

Boot:

Name of the device and partition from which the system was booted.

Current:

Name of the current directory.

Default:

Name of the default search list.

Dev:

Name of the system device.

Run:

Name of the search list to find run files.

MachineName

Name of the machine.

ShellCommands

Name of the file that holds the Shell commands.

ShellName

Name of the program to use as a Shell.

ShellProfile

Name of the default profile.

UserName

Name of the current user of the machine.

FloppyServerName

Name that the Floppy utility will use to find Floppy Server.

Appendix D. Standard System Servers

This appendix gives a list of the standard system servers and their request ports, which are defined in PascalInit. The interfaces to these servers are initialized before control is given to client code. This means that the Init call to these servers does not need to be done in the client code.

Time Server	TimePort
File Server	SesPort
Environment Manager	EMPort
Process Manager	PMPort
Name Server	NameServerPort
TypeScript Server	TypeScriptPort
Window Manager	SapphPort

Appendix E. Inter-Program Argument Format

Program arguments are passed as a counted list of counted words, each potentially having an arbitrary number of characters, along with an index to the beginning of each word.

A. Implementation

1. The fields:

   WordCount: long; ! Actual number of words

   WordDirIndex: long; ! Byte index to dir in WordArrayPtr^

   WordArrayPtr: ^ packed array [*] of char; ! Words and dir

   WordArray_Cnt: long; ! Actual number of bytes in word array

   are part of the initial message (InitMsgType).

2. Within WordArrayPtr^, individual words are stored sequentially, with each word having the following fields:

   WordSize: long; ! Number of chars in the word

   Data: packed array [0..WordSize] of byte;

   The word must be null-terminated (i.e. Data[WordSize] = chr(0)).

3. All words must begin on the next even 32-bit boundary following the previous word.

4. A "directory" to each word is contained in WordArrayPtr^ at index WordDirIndex. The directory will consist of WordCount longs, each one

being the index into WordArrayPtr^ of the corresponding word (i.e. for each i such that 0 < i < WordCount, WordArrayPtr^[[WordDirIndex + 4*i]] is the low byte of WordSize for word i).

5. WordDirIndex must index to the next even 32-bit boundary following the last word.

B. Properties:

1. Arguments are arbitrary numbers of arbitrary length words.
2. Words can be built sequentially.
3. This is random access to all words.
4. Arguments and directory should almost always fit into one page.