RIG, Rochester's Intelligent Gateway:

System Overview

E. Ball, J. Feldman, J. Low,
R. Rashid, and P. Rovner
Computer Science Department
The University of Rochester

TR5

RIG, Rochester's Intelligent Gateway: System Overview

by

E. Ball, J. Feldman, J. Low, R. Rashid, and P. Rovner

Computer Science Department

University of Rochester

TR5

The RIG system provides convenient access to a wide range of computing facilities. The system includes five large mini-computers in a very fast internal network, disk and tape storage, a printer/plotter and a number of display terminals. These are connected to larger campus machines (IBM 360/65 and DEC KL10) and to the ARPANET. The operating system and other software support for such a system present some interesting design problems. This paper contains a high level technical discussion of the software designs, many of which will be treated in more detail in subsequent reports.

"When I opened my eyes I saw the Aleph ....
the place where, without any possible
confusion, all the places in the world
are found, seen from every angle."

--Jorge Luis Borges, "The Aleph"
(translation by Anthony Kerrigan)

In a world where networks of diverse computing resources are growing and intertwining, there is a pressing need for systems which provide access to a variety of computers and serve as intelligent gateways to their use. In response to our own needs we are developing an operating system which, after Borges' point containing all other points, we are calling Aleph. Aleph is based on a simple message-passing discipline for inter-process communication. When completed, Aleph will form the framework for Rochester's Intelligent Gateway (RIG), a system for uniform access to a variety of local and remote computing facilities.

1. RIG OVERVIEW

At the simplest (most abstract) level, a gateway system can be seen as a mechanism for connecting terminals to a variety of computers and computer networks. Much of the subsequent discussion applies to any such system. At this abstract level, RIG will look as in Figure 1.

The three large machine connections in Figure 1 (360, KL10, CERF) are representative of facilities one might expect to be available locally. The first is the University batch-processing computer which also has some interactive capabilities (WYLBUR, APL). The DEC KL10 is used as a general purpose time sharing system. CERF (Computer Engineering Research Facility) %Wilhelm75% is an experimental computer designed and being built by the University of Rochester Department of Electrical Engineering. All these machines are located several hundred meters away from the gateway processor.

Two network connections are shown (ARPANET, ETHERNET). They represent two distinctly different kinds of computer networks: the ARPANET being a large nationwide network of research computers, the ETHERNET representing a high speed (3 megahertz) in-house network of four 64K, 16 bit word mini-computers.

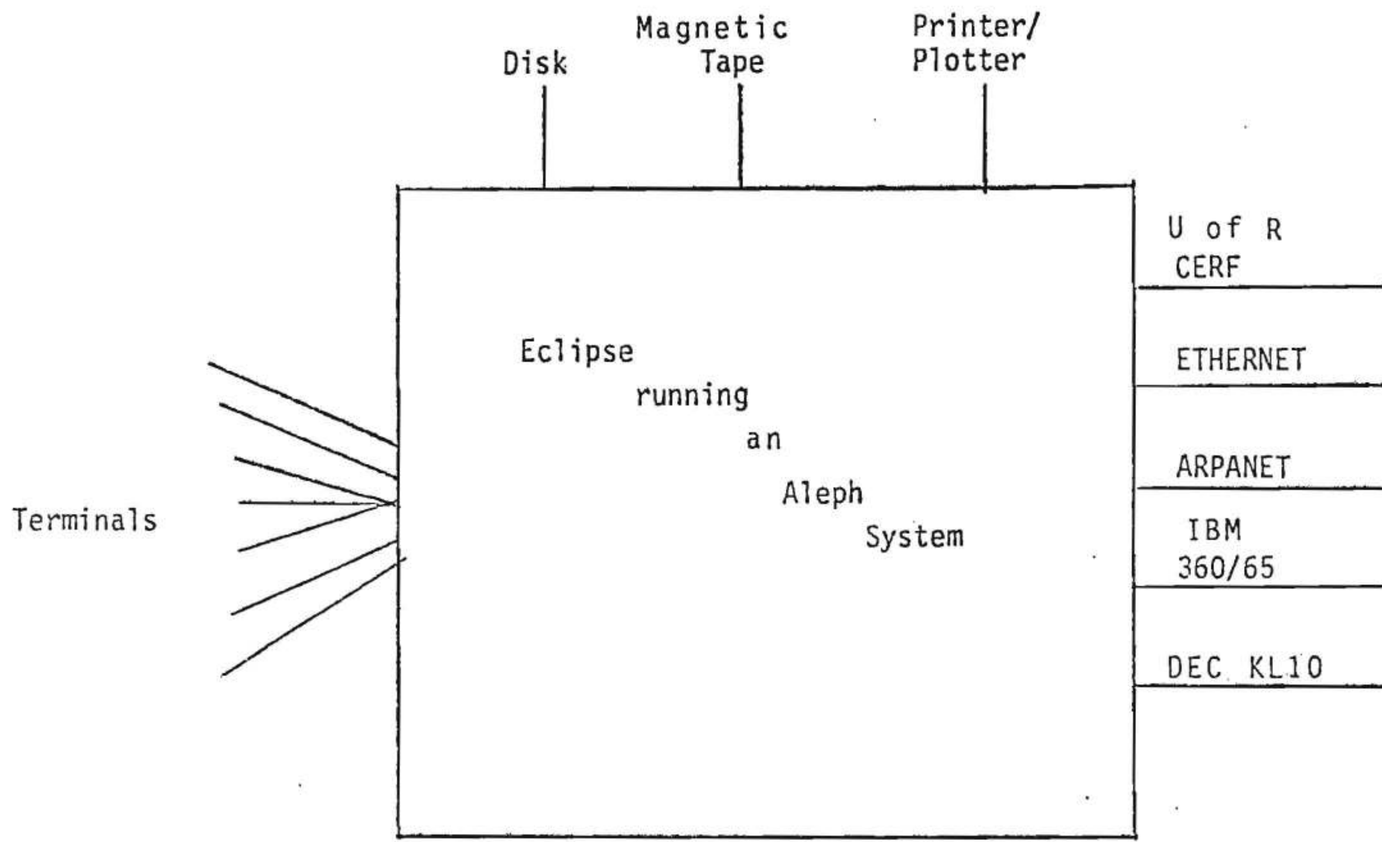


Figure 1. RIG System Overview

The RIG central processor is a Data General Eclipse -- a relatively powerful mini-computer. The use of a mini-computer for our gateway provides obvious cost advantages along with some implementation headaches.

RIG includes facilities for local file storage and backup and for printed and plotted output. The availability of these facilities is basic to the gateway concept. The RIG user will be able to create and edit files locally, with all the advantages that local computing offers: a single familiar editor, fast reliable response and better security and protection. The user can then choose among several larger machines to process his file.

2. INTELLIGENT GATEWAYS

These are some of the features we feel must be provided by an intelligent gateway system:

(1) It must be able to handle a number of terminals, each of which may be monitoring several tasks in varying stages of completion. These include editing, file manipulation, and communication with other computers on either a character by character or file basis.

(2) The user should be insulated whenever possible from the idiosyncrasies of host computers. He should be provided with a set of locally defined primitives (e.g. for requesting compilation and loading) which the gateway can convert into commands meaningful to the remote host.

(3) Response to modest requests should be rapid.

Certainly the ideal modern general purpose time-sharing system (GPTSS) should be capable of modification to meet all of these requirements. There is, however, a good reason not to turn a GPTSS into a gateway. A GPTSS is designed to provide reasonable response to any user program which does not demand excessive resources. The RIG processor is not intended to support arbitrary user programs. Instead, it provides easy access to remote computers where such programs may be run. As a result, a gateway operating system can be simpler and more efficient than a GPTSS, allowing us to utilize a smaller, less powerful machine.

3. ALEPH -- A GATEWAY OPERATING SYSTEM

Aleph is divided into 3 levels -- kernel, foreground, and background -- each with distinct functions and

	Communications with	Interrupt Discipline	Runs to Completion?	Availability
Kernel	Foreground only	Standard Multi-programming (~ 10 μ Sec)	Yes, except for hardware interrupts	Always available
Foreground	All	Polling (~ 1 m Sec)	To a polling (clean) point	Very rapid access by map switch
Background	Foreground only	Check-pointing (~ 10 Sec)	Preemptable	Secondary Storage

Figure 2. Properties of RIG processes at the three levels.

communication disciplines (Figure 2).

a) Foreground

The foreground is the locus of all RIG activity. RIG will be required to provide three kinds of service to the external world: full or half duplex character transmission between terminals and any of the gateway accessible computers, file transfer operations between any two systems or peripherals, and process to process communication among systems. We have chosen to dedicate an independent process within RIG to each external connection and to establish a uniform message-passing system for inter-process communication. Each process has responsibility for maintaining its external communication protocol and for performing any conversions necessary to enable it to present a standard interface to the rest of the system. Thus, knowledge of the idiosyncrasies of a particular connection will be required only within the process dedicated to it. A typical collection of foreground processes is given in Figure 3. As an example, the Arpanet process depicted in Figure 3 has various 'ports' of communication to other processes in the foreground. These correspond to 'connections' between remote systems and processes within RIG. The use of standardized message formats for character, file, and process communications, allows the high degree of

flexibility and modularity necessary to provide practical communications with widely varying computer systems.

Foreground interprocess communication is effected by a message queueing and distribution system similar to that described in (Walden72). Messages can also be sent to and from kernel device drivers and background jobs (to be described later). Typically, a device (e.g. the printer/plotter) will have a dedicated foreground process which communicates with other foreground processes and a kernel driver which actually runs the device. There is no pre-emption among foreground processes and time-slicing. Process control is based on a modifiable software priority scheme. Rescheduling is done only at "clean points" established by each process individually. This makes possible much faster context switching than one could have in a GPTSS, without sacrificing the integrity of the processes. Foreground processes will always be in main memory while they are in use.

b) Kernel

The Aleph kernel provides the basic functions used by all processes in the system. These functions include:

(1) control of I/O devices

(2) scheduling of foreground and background tasks

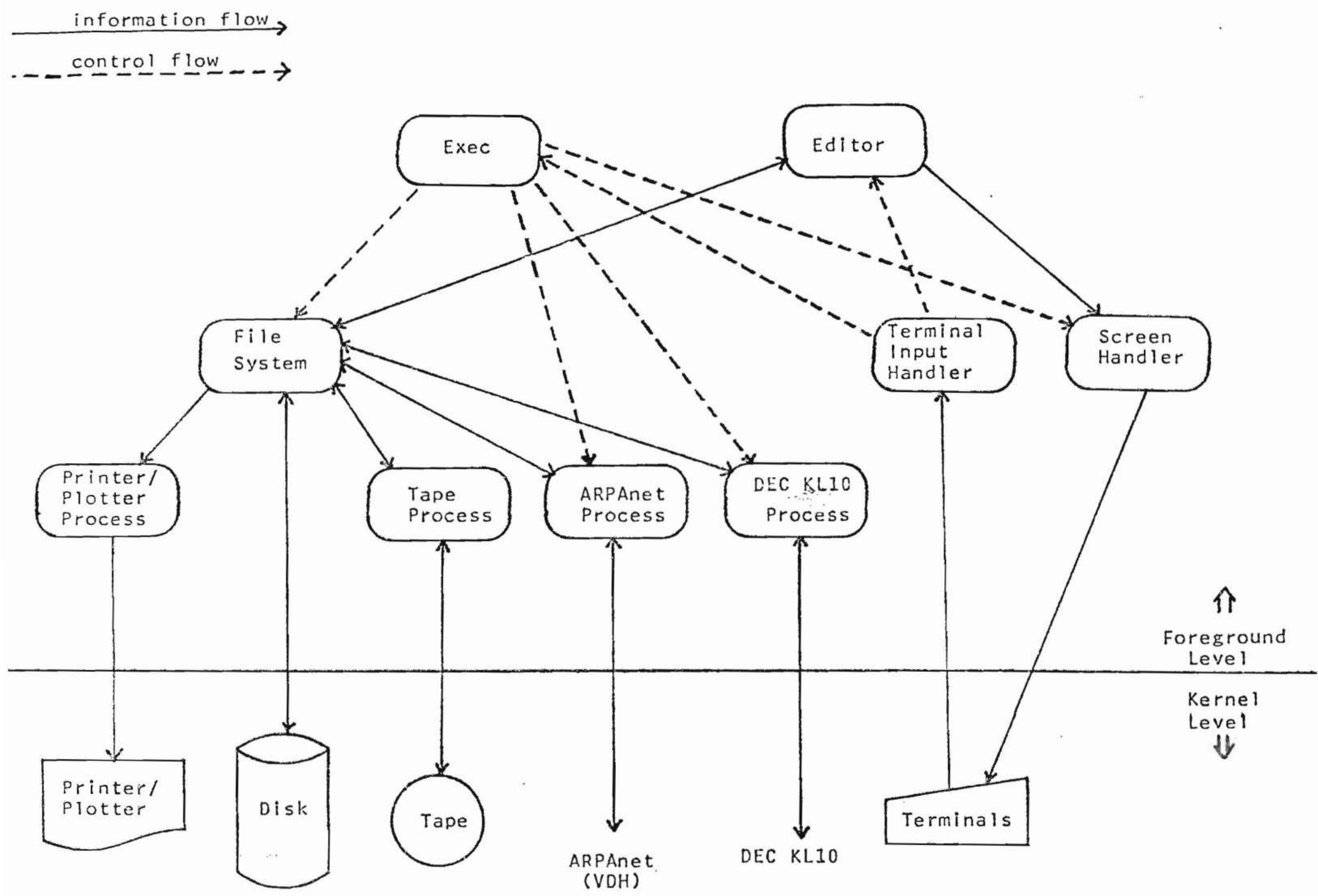


Figure 3. Typical Aleph process organization

(3) inter-process communication

(4) memory management and allocation.

I/O device drivers are controlled by a conventional priority interrupt mechanism. Each device driver has associated with it a single foreground process which handles I/O requests from other processes in the system. The kernel level driver can communicate directly with this process and can 'awaken' it (force it into the ready queue) if it is suspended.

The Aleph scheduler selects the highest priority ready foreground process for execution. If all foreground processes are suspended, control passes to a single background job until the foreground requires processing. The scheduler uses no time-slicing or pre-emption in the foreground. Therefore the foreground is rescheduled only when the currently executing process relinquishes control explicitly. I/O interrupts do not cause rescheduling so the effects of an I/O-complete are not felt in the foreground until the next rescheduling.

A foreground process allows rescheduling in one of three ways:

(1) By sending a message to another process

(2) By requesting a message from its input queue

(3) By performing a 'clean point' call

Clearly the performance of Aleph will depend upon the frequency of these rescheduling requests. Because the system functions primarily as an input/output distribution network very little processing ever occurs without the necessity of inter-process communication. This means that most processes effectively time-slice themselves by continually sending or receiving messages. In the few situations (such as formatting) where processing time may be somewhat greater, we require each process to periodically relinquish control. Since the Aleph foreground level is not intended to include 'user' programs (such programs will be executed on remote machines) such behavioral requirements can reasonably be made of the processes in the system.

The advantage to be gained by designing a request-driven scheduling system isolated from hardware interrupts is simplicity. The overhead of context switching, for interrupts or rescheduling, is reduced to a minimum. More importantly, because each process knows that it loses control only at its own request, it can avoid the synchronization and critical race problems that would arise if it were being time-sliced.

Communication between processes in the foreground is in the form of 'messages.' Each process has a unique process number and any other process can send it a message using this number as an address. The routines that support the message system are included in the Aleph kernel. These routines maintain an input queue of messages for each process in the foreground. When a process requests a message Aleph removes one from its input queue and returns it to the caller. If there are no messages waiting, the caller is suspended until a message is sent to it by another process. Sending a message causes it to be linked into the receiver's queue, and awakens the receiver if it has been suspended by a message request.

The address used to specify the destination (or source) of a message consists of a process number and a port number. The port mechanism allows a process to define several logical addresses within itself and communicate with other processes from each of them (Figure 4). These ports are normally allocated for communication concerning a single set of requests. For example, a file system process might allocate a port to represent a single open disk file. All messages requesting manipulation of that file would be sent to that port of the file system process. Since Aleph allows a process to receive the next message for a specified input

port, processes can easily control their own input by assigning ports to logical tasks and can allow the message switching system to perform any queueing that might be required.

Aleph also allows a process to wait for a message to arrive from a specified process-port. Thus a process can suspend itself until it has received an acknowledgement that a critical request has been completed, thereby allowing the processing of other messages to continue.

In addition to these message primitives, Aleph allows each process to inspect messages waiting in its input queue and to receive a particular message from any place in the queue. When requesting a message, a process also has the option of specifying a timeout period, and if no message arrives within that time, Aleph will notify it that a timeout occurred. This feature enables the system to recover from error situations in which a process fails to respond to requests.

The Aleph message protocol confines interactions between system modules to a well structured format that discourages poorly defined dependencies between processes. Because of this structure Aleph is able to provide powerful

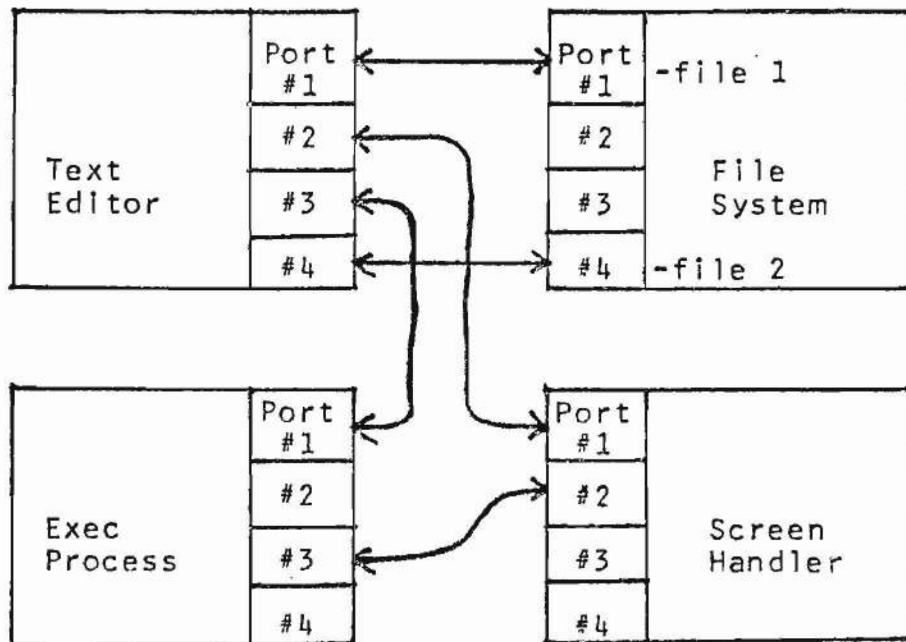


Figure 4. Aleph processes: Use of port numbers

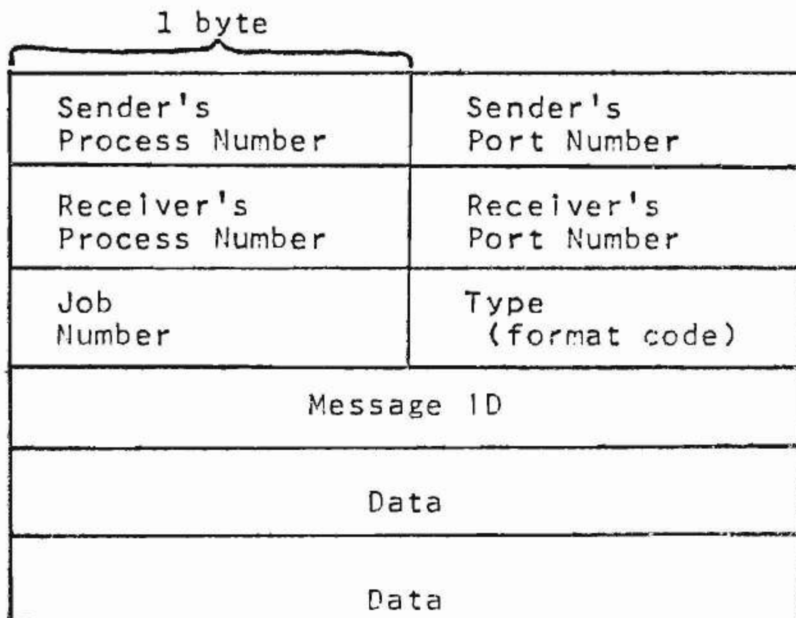


Figure 5. Aleph message format

methods of modifying information flow within the system. The 'shadow' facility defines a shadow process which receives a copy of every message sent to a specified target process. The shadow process can then monitor its activity, gather statistics, maintain logs, etc. without requiring any modification of the target process. The shadow facility can also be used to debug new versions of system modules. The 'interposer' facility allows a special process to intercept all messages sent to a particular process or to intercept all messages originating from it. Thus, an interposer could intercept all transactions between a pair of processes and perform additional processing on the messages without requiring any changes in the code of either process. This ability to add 'intelligent' processing to any information path in the system provides the flexibility necessary for a gateway system.

An Aleph message (Figure 5) contains six 16 bit words of information. Two words are used to specify the source and destination addresses (process-port). A one byte 'job number' identifies the user job that caused the message to be sent. That is, all messages that are sent in order to satisfy the requests of one user have that user's job number. One word contains a system unique 'message ID' that specifies the meaning of the message. For example, one

message ID means 'open a disk file,' and this message would be recognized as invalid if sent to the printer/plotter process. Thus, by requiring message ID's to have a unique system-wide meaning we have simplified debugging considerably. The one byte 'type' field indicates the format of the remaining two words of the message. In some cases these words hold the data of the message (characters, binary parameters, etc.). For larger transactions these words define the size and location of a memory buffer which contains the data to be transferred.

The Aleph memory manager (included in the kernel) maintains memory pools from which space for data buffers, system tables, and BCPL (Curry72) stack is allocated. Memory in each pool is allocated using a boundary tag system. Each buffer has two 'handles' that can be allocated to Aleph foreground processes, thus giving them access to the buffer. Only two processes can have simultaneous access to a buffer, but any process can elect to transfer its access rights to another process. A buffer is returned to the memory pool only after both of the processes having access to it release their handles. In a typical example, the tape drive process might ask the file system for the next block of an open file. The file system process would allocate a buffer with handles for itself and the tape

process. After filling the buffer, it would send the tape process a message that the transfer was complete and release its own handle. When the tape process was finished with the data it would release its handle and the buffer would be re-allocated.

For still larger data transfers, two processes can allocate a group of buffers that pass back and forth between them. This scheme requires the overhead of buffer allocation only once, and simplifies the implementation of a multiply buffered transfer strategy. Because only two processes have access to any buffer at one time, synchronization problems involving buffers are relatively local and easy to understand.

c) Background

Aleph should be able to maintain rapid response to a large number of widely varied requests. To insure this, we must explicitly relegate some longer operations to a background level. Typical background tasks include file transfer, long editor searches, etc. as well as traditionally batch operations such as compiling. Thus a program setting up a file transfer will be in the foreground, but the transfer itself will not have guaranteed response and can be done in the background. It is currently

planned that background tasks will normally run to completion. There will be some pre-emptive checkpointing of background tasks, but this should be kept to a minimum.

4. FOREGROUND SYSTEM FUNCTIONS

By the nature of our design, many of the functions commonly considered part of an operating system kernel are in RIG foreground processes. Among these functions are file system maintenance and peripheral management. We believe this improves RIG's overall flexibility by increasing its modularity. In this section we will describe some of the foreground processes already implemented as part of RIG which have a direct bearing on our goal of making RIG a true gateway. In particular we will discuss the RIG screen and text handling processes. Other RIG processes such as the various file system handlers, command interpreters and editors will be discussed in separate reports.

a) Screen Management

An important design goal of our gateway is to allow users to oversee several tasks in varying stages of completion. This kind of facility is important because many RIG tasks (e.g. file transfers, compilations, the monitoring

of number crunching programs on host computers, etc.) may require relatively little supervision and yet take a disproportionate amount of working time away from the user.

While not commonplace, similar mechanisms have been included in other systems (e.g. Tenex Telnet). A problem with previous implementations, however, has been one of informing the user about the progress of his activities. Typically the user has only one logical line of communication. Input and output are both multiplexed in time, forcing the user to periodically look in on each process to assess its state. His working context is at the best a long scroll of paper and at worst his fragile short term memory. Consequently, although it might increase his useful terminal time, a user may avoid using such a facility to the fullest simply to prevent himself from being burdened by its complexity.

b) Screens, Regions, and Subregions

The advent of inexpensive display oriented terminals suggests a solution to this problem: divide the user's visible screen into logical regions, each of which may be dedicated independently to different user tasks. Output could thereby be spatially as well as temporally related,

permitting the user to view the activity of more than one program at a time (Swinehart73).

We have adopted this solution for our gateway. A special process called the Screen Handler is responsible for the allocation of space on each terminal display screen.

In our system a 'screen' is defined to be those lines of text physically visible on a display device. A 'region' is a rectangular area within a screen in which the output of a single process may be displayed. A 'subregion' is a rectangular area within a region which is independent of all other areas of the screen. Subregions are the basic unit of screen space allocation. A region is simply a collection of contiguous subregions belonging to the same Aleph process, and a screen is a collection of regions visible on the same display device. No empty space is allowed in the sense that the physical screen is always allocated completely to one or more regions, each of which in turn consists of one or more subregions. New regions and subregions may be created only by splitting existing entities into two parts of variable size.

This hierarchical division of the screen permits us to satisfy the output requirements of several independent

processes without compromising the readability of the display. An 'Executive' process is associated with each user in the system. This process is the root of the tree of processes spawned by the user of the system and as such is the holder of special privileges in communication with the Screen Handler. The Executive alone, presumably at the request of the user, performs the task of sectioning the screen into regions. These regions are allocated to the processes spawned by the Executive which may then divide them into subregions as necessary. The result is positional constancy and spatial integrity of information. Regions of the screen devoted to user processes do not vary in size or change position except at the explicit request of the user acting through the Executive. Moreover, all output pertinent to the execution of a single process is contiguous on the screen. A typical 'snapshot' of a screen would look like Figure 6.

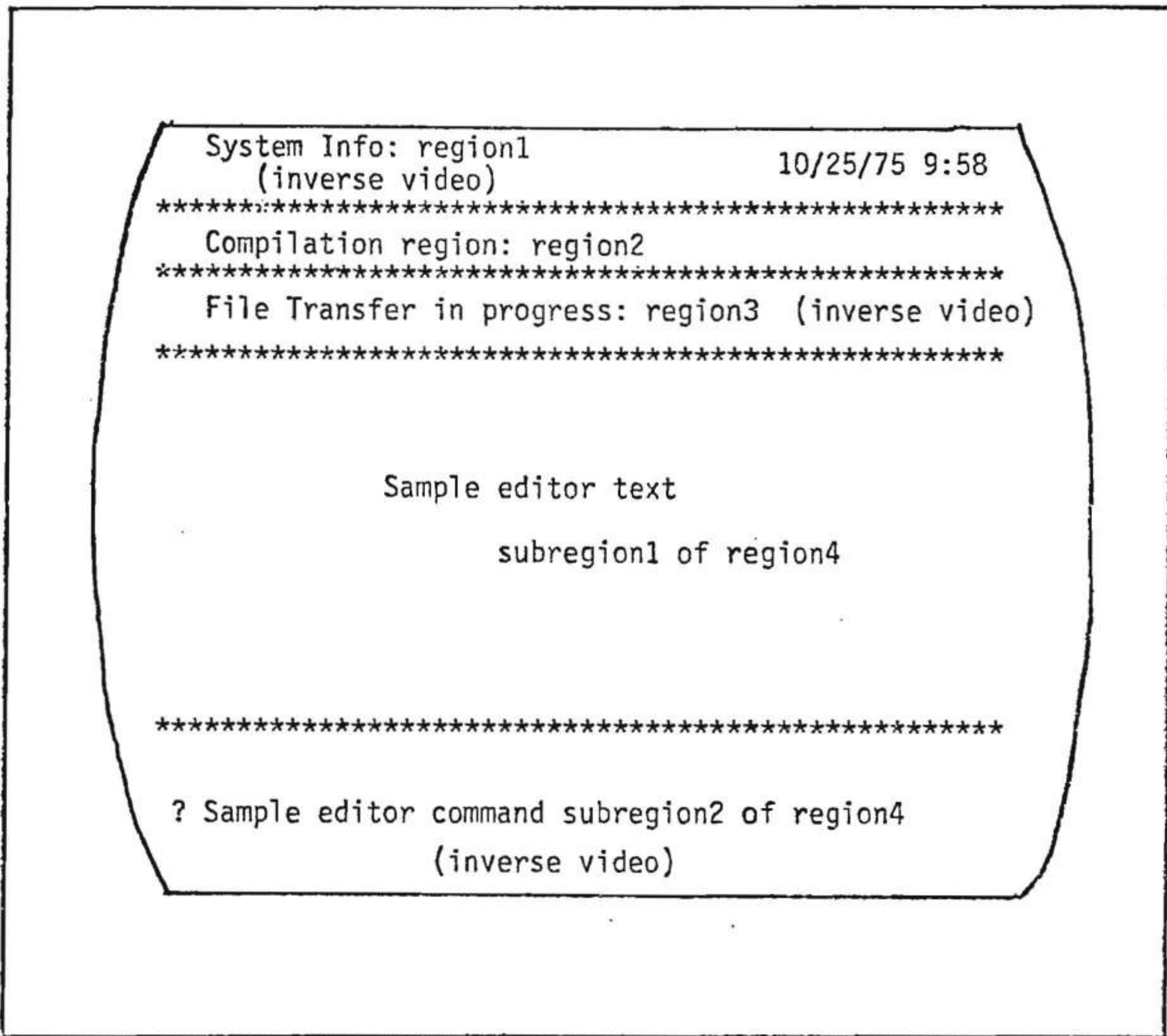
A serious drawback to the use of the screen to effectively inform the user about the state of his work is the relatively small amount of displayable text available on most terminals. Generally speaking, the entire context of a user's activities cannot always be visible on a single screen. Thus it becomes crucial for the mapping of lines to areas of the screen to be flexible. Important text, as

selected by the user or his program, should always be visible, but less recent text should not necessarily be lost. If that were the case we would have provided the user with no more than an array of small logical displays to replace his single physical terminal. It is for this reason that we have decided to isolate the function of terminal output from the task of collecting the outgoing information of an Aleph process. No Aleph process ever directly sends a message to the terminal output handlers. Instead, an Aleph process modifies a data structure we call a 'pad' which, if the pad is mapped into a subregion, results in such an output message.

c) Pads

A pad is both a data structure and an Aleph process. As a data structure it consists of a variable number of text lines which could be used, for example, as an input buffer, a window onto a file or a depository for temporary textual data. As a process a pad has the ability to send and receive messages. Through messages it provides a powerful set of text manipulation primitives to the foreground process which owns it, including most of the functions performed by simple text editors, e.g. character and line insertion and deletion, searching and substitution,

Figure 6



scrolling, overwriting, and others. Readers familiar with Simula may be more comfortable with this concept if they think of a pad as a Simula class structure. Protection is afforded to the system because all access to pad data structures is constrained. At the same time user processes are provided with a powerful tool for text manipulation.

Not only do pads provide for convenient text manipulation, they can also be used for communication between a foreground process and a user terminal. The textual content of a pad may be mapped onto any subregion of a screen. When this is done, all changes made to the pad are automatically reflected in changes made to its terminal image. Because a pad may contain more text than can be displayed in a given subregion, each pad maintains an internal pseudo-cursor which points to the current focus of attention (either as indicated by a cursor motion message sent by the foreground processes which owns the pad or by the last change made to the pad). Only that portion of text about the pad pseudo-cursor which will fit in the subregion is displayed. Using pad pseudo-cursor motion messages, the owning foreground process can select which portion of the pad is to be displayed, providing a working context through pad commands rather than screen space (a scarce resource!).

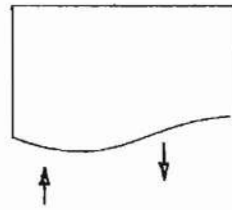
d) Pads and Files

In order to facilitate the use of a pad as a textual window onto a file, each pad is provided with an input and an output process-port pair by its owning process. Scrolling and pad overflow or underflow cause messages to be sent to these ports resulting in the input or output of new lines. If an output process-port pair is left unspecified, all output is discarded. The archetypal use of this facility is in file manipulation. In this case the pad's input-output ports correspond to an open file being handled by an Aleph file process (Figure 7).

5. DISTRIBUTED COMPUTING

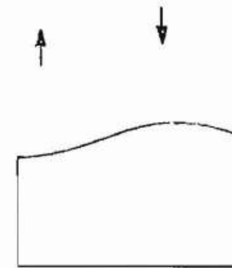
The major topic omitted from our previous discussions is the fast internal network of mini-computers. This currently consists of four machines (which we will call PALOs) in addition to the Eclipse. Each machine has 64K of 16 bit words, a disk, keyboard, and raster display. It can be used either as a stand-alone computer or as a very powerful terminal connected to RIG. Using the PALO as a terminal we can experiment with the ARPA network graphics protocol (Sproull74) and other 'intelligent terminal' functions.

Figure 7



Top of file

Pad



Bottom of file

The four PALOs and the Eclipse are linked by a three megahertz connection called the Ethernet (Metcalfe75). An overall system diagram is given in Figure 8. The Ethernet connection is fast enough to support distributed computing as well as more standard network operations. One current project is to use an Eclipse simulator in the PALO as an alternative background level. A task, e.g. a compilation, can be sent to a PALO for execution and the result returned to the Eclipse. The user need not be aware of where his compilation is performed. There is much more to be done along these lines, some of which is discussed in the next section.

6. INTELLIGENT GATEWAY FUNCTIONS

Very little has been presented so far that would justify the word 'intelligent' in the middle of 'RIG.' The services we have described are themselves in no way remarkable. One of our basic design goals was that the system should provide a great deal of help to users trying to cope with myriad systems. Although only a small part of this facility has been specified in detail, the general outlines are already clear.

We discussed the division of our users into two

classes: those whose access to the central computing facility is direct through alpha-numeric display oriented terminals and those whose major local resource is a relatively powerful stand-alone mini-computer linked to our central facility via a three megahertz network. In the latter case, the benefits of a message switching system in the central processor are fairly obvious and have been described by a number of authors. Our message switching system is potentially the communications skeleton of a resource sharing computer network, defined by David Walden to be "a set of autonomous, independent computer systems, interconnected to permit each computer system to utilize all the resources of the other computer systems much as it would normally call one of its own subroutines" (Walden72). Processes living on a peripheral computer (one of the PALOs) can access facilities provided by Eclipse subsystems (processes) in the same manner that other processes resident in the Eclipse may access those subsystems -- through switcher messages. Interprocess communication over the network can be viewed as a subcase of general switcher communication. Messages destined for non-local process-ports need simply be packaged into network mail to be sent to the appropriate computer. A sample scenario detailing how this can work in a file system interaction between a PALO and the Eclipse would proceed as follows:

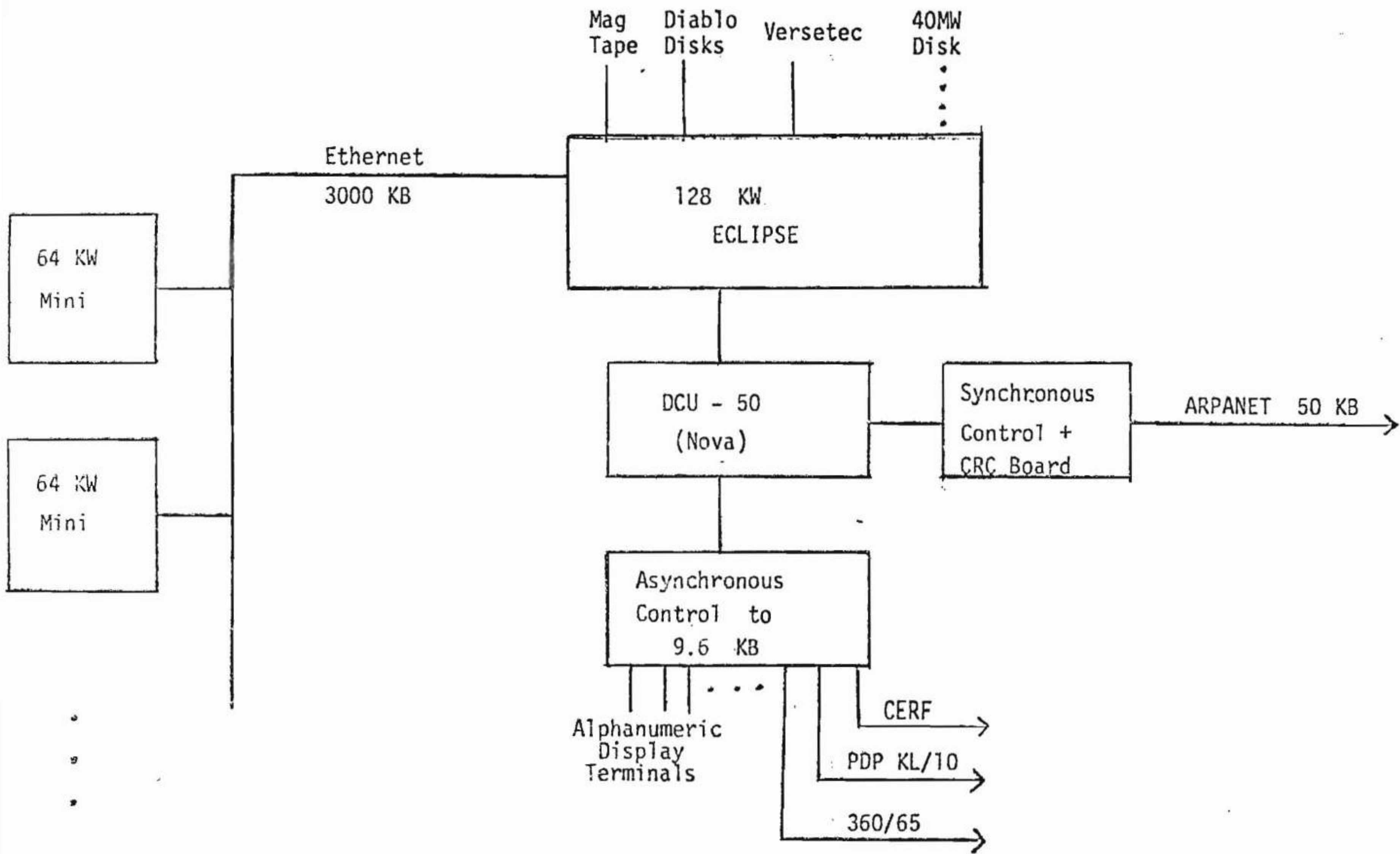


Figure 8. RIG Hardware Configuration

A PALO program would like file status information concerning a file resident on a disk pack serviced by the Eclipse. It sends a switcher message over the net to the file handling process-port for that device. This message is received by the Eclipse, converted to a local message, and posted. The file process receives this message and responds by sending its reply to the requesting process-port. This message is then packaged into network mail for the PALO and sent. The PALO receives the reply and continues processing.

The fact that the request was being made by a process outside the local environment was transparent to the file system program itself. The only interface between the PALO and Eclipse processes existed at the level of network input/output, although it was, of course, necessary for the PALO process to be able to understand and use proper protocol. Some such protocol knowledge would have been required for any form of communication.

Another example is a user at an alpha-numeric terminal composing a program to run on a remote machine. It is probably best to use the text editor on the local machine (better response, cheaper file storage). The compiler for a particular high-level language might be on a remote machine.

It would presumably be sent a correctly formatted text file and appropriate control commands. A request to list the output will normally be interpreted (by an intercessor process) as a request to get the compiled program back to the local machine for printing, perhaps with character conversion. If the code is to be debugged on a remote machine, one would want to use a character-by-character protocol across the network for the interactive debugger.

One way to view the intelligent gateway concept is that a user should be able to access a remote subsystem without knowing the operating system conventions of the machine on which the subsystem is running. An obvious extension is to have the gateway system dynamically choose the best set of resources for a given job. This is a natural extension of what we are doing with our internal network and will certainly be added to RIG, at least in a rudimentary form.

The overall shape of an intelligent terminal system is becoming clear. There are four principle components:

(1) the user interface;

(2) The data-structure (directory) for keeping track of system state;

(3) a set of rules of procedure; and

(4) a response handling capability.

We will discuss each of these briefly, describing its role and how we plan to implement it.

In most general terms, the user interface includes many pieces such as the editor, screen handler, etc. In the narrow sense considered here, the user interface is a command language. For an intelligent terminal system, one would like a more expressive and general language than is common in operating systems design. The user will want to provide descriptive information and rather complex instructions in addition to simple command sequences. We plan to use a compiler-like language and implement it using a production language (PL) interpreter. The PL system is natural to use, has a state-table nature, and can easily be made to accommodate different command languages.

The directory envisioned for RIG is an extension of the standard file directory which has a number of attributes such as location, size, format, protection, etc. A RIG directory extends this idea in two ways: (1) Directory entries are much richer because of the need to deal with many different systems; and (2) Entries are included for several entities (subsystems, parameters, keywords, etc.) in addition to files. From the point of view of a user, the directory structure is an associative store. We also expect

to extend the notion of hierarchical directories to be more like the context mechanisms (Bobrow73) of the AI languages. The system should be able to be context sensitive in its handling of completion, checking, defaulting, etc.

The rules for handling any particular situation in the RIG environment are not too complex, assuming that one has a reasonable set of primitives. This gives rise to the hope that a user can be made to understand and perhaps change what is being done to him. Recent work in AI has yielded a number of ways of expressing sets of rules of procedure. These range from totally undirected axiom schemes, through situation-action (production) rules to very specific routines. Our model is of about the generality of Schank-Abelson scripts (Schank75): there is an expected sequence of events and actions, but many details are expected to vary from case to case. Preliminary efforts suggest that we can build a readable yet efficient rule language for at least some simple intelligent terminal functions. The interpreter for this rule language will be implemented using the same PL system used for the command language.

Perhaps the most difficult task of an intelligent terminal system is responding appropriately to messages from

other systems. One could, with some justice, treat this as a restricted natural language understanding task and use the pertinent methods. We are choosing to try a simpler scheme based (mirabile dictu) on a PL interpreter. It seems that if RIG can keep enough context, then it can use the appropriate set of PL tables and handle many responses without employing complex methods. We do not anticipate that casual users will learn to write response handlers, but it shouldn't really be that difficult in most cases.

Each of the four components has been designed to use the simplest adequate techniques. This makes the intelligent terminal effort seem more like one in systems programming than one in AI. Many of the really difficult problems will involve AI techniques (e.g. trying various methods for achieving a goal), but we feel that a great deal of understanding can be attained by constructing a system which is conceptually straightforward.

References

Bobrow73 Bobrow, D. and Raphael, B. "New Programming Languages for AI Research". ACM Computing Surveys. Vol. 6, No. 3, 1974. pp.153-174.

Curry75 Curry, J. BCPL Reference Manual. Xerox Palo Alto Research Center, 1975.

Metcalfe75 Metcalfe, R.M. and Boggs, D.R. "Ethernet: Distributed Packet Switching for Local Computer Networks". Communications of the ACM forthcoming. 1975.

Schank75 Schank, R. and Abelson, R. "Scripts, Plans and Knowledge". Proceedings of the Fourth International Joint Conferences on Artificial Intelligence. 1975. pp. 151-157.

Sproull74 Sproull, R.F. and Thomas, E. "A Network Graphics Protocol". SIGGRAPH - ACM. Vol. 8, No. 3, 1974.

Swinehart74 Swinehart, D.C. "Copilot: A Multiple Process Approach to Interactive Programming Systems". PhD thesis, Stanford University. 1974.

Walden72 Walden, D.C. "A System for Interprocess Communication in a Resource Sharing Computer Network". Communications of the ACM, Vol. 15, No. 4, 1972. pp.221-230.

Wilhelm75 Wilhelm, N. "Computer Engineering Research Facility" University of Rochester. 1975.