UCAM-CL-TR-5
ISSN 1476-2986

Parrot – A replacement for TCAM

P. Hazel, A.J.M. Stoneley

April 1976

ISSN 1476-2986

Summary

The terminal driving software and hardware for the Cambridge TSO (Phoenix) system is described. TCAM and the IBM communications controller were replaced by a locally written software system and a PDP-11 complex. This provided greater flexibility, reliability, efficiency and a better "end-user" interface than was possible under a standard IBM system.

P. Hazel
A.J.M. Stoneley

April 1976

Contents

1 Introduction

The main computing engine of the Cambridge University Computing Service is an IBM 370/165 with an MVT* supervisor. One of the main means of access to the system is the Phoenix online system, which is based on TSO. In particular, those parts of the system concerned with swapping and sharing of time (TSC and the RCTs) are essentially as provided, whilst those parts concerned with the individual user (the Terminal Monitor Program, Editor, sundry utilities and the dynamic allocation routines) have been replaced.

The primary aim of the Phoenix system is to provide editing and job submission facilities for a large number of people, rather than to provide actual computing power for a few. To this end a large number of terminals must be active in a highly interactive fashion. We currently have over 200 connected terminals, of which any 80 may be logged on at any time. All these terminals, as well as various other devices, are connected to a PDP-11 system, which in turn is connected to the IBM 2870 multiplexor channel through a DX-11 interface. The 370/165 is aware only of the active terminals, which are selected for it from the 200 available by the PDP-11.

In the initial system (1972) the system software in the 370/165 was totally unaware of the PDP-11. TSO was connected to TCAM in the standard fashion and TCAM thought it was connected to half duplex Teletypes via a 2703 transmission control unit. The PDP-11 emulated the 2703 in all necessary respects. The reasons for this were of two classes: reasons for not having a real 2703 and reasons for pretending that we had. In the former class, the most prominent reason was cost. A bank of 2703s providing hardware connections for a projected 300 terminals would have been very expensive, and indeed this number is beyond the addressing range of a single channel. Other reasons were that we would have been restricted to devices which IBM chose to acknowledge and that the performance of IBM multiplexors of that era left much to be desired. They were no more than multiplexors and devoid of intelligence, leaving such requirements to TCAM. We required substantially greater flexibility, both in types of device and in facilities such as local line editing and escape character input. We also required facilities such as paper tape readers and punches which are unrelated to TSO. The choice of interface, the emulation of a 2703 driving Teletypes, was based on the need for independence of particular IBM operating systems.

*[There is a glossary of IBM terminology at the end of this report]

In that system, which survived for about three years, TCAM believed it was in contact with a fixed number of half duplex Teletypes, all of which could be logged on at the same time. In fact it was connected to devices of various types, all disguised by the PDP-11. All device dependent code was thus removed to the PDP-11, including character code translation (TCAM character translation was bypassed). As far as TCAM was concerned, all its terminals could be logged on at once and so the size and complexity of the TCAM MCP was that appropriate to the maximum number of terminals logged-on, rather than the total array.

In the course of time various causes for dissatisfaction arose, few severe but the total aggravating. The most severe problem was the shortage of device addresses. With all the other devices attached to the multiplexor channel, very little room was left for expansion. We envisaged ultimately about 100 logged on users. Main store was also in short supply and TCAM plus the TIOC seemed rather excessive in their requirements. A number of outright bugs were present in TCAM and the lack of documentation coupled with generally poor programming style defeated our efforts to find these bugs. There were also a number of infelicities. TCAM tended to swallow characters which it thought inappropriate to a Teletype and sometimes inserted various other characters. The most irritating insertions were the "write idles". These are non-printing characters which cycle the Teletype mechanism, the intent being to prevent the user typing at the keyboard when input cannot be accepted. The resultant 'chuntering' is extremely irritating and for us it was also futile: most of our Teletypes actually operated in full duplex mode, acceptance of input being indicated by the echo. Consequently we had the PDP-11 swallow the write idles, rendering it impossible to transmit these characters when really required, for example to a graphic device. This lost us four characters, each of which was used by a different part of TCAM for such purposes. A further infelicity lay in the treatment of "BREAK" (attention). BREAK appeared to TCAM as an I/O error during a read or write command. When a user's input buffers were full TCAM would cease to read from the terminal and BREAK could not then be sent. Common reasons for input buffers being full are an unintentional program loop or a never-ending WAIT, and this is precisely when one would like to be able to signal BREAK. In such a case the only remedy was to telephone the operators and request "cancellation" of one's session. Finally, it was felt that it was silly to have a sophisticated PDP-11 system pretending to be a totally stupid 2703. Several new features could be conceived if the 370/165 were to acknowledge the existence of the power of the PDP-11.

These considerations prompted us to replace TCAM as the TSO device driving program. This is commonly regarded as extremely difficult if not impossible and it is worth considering how we found it possible to do this economically. In the first place, we did not replace TCAM per se but the whole of the terminal I/O software. Secondly, we have not provided any facility not related to TSO. TCAM is in its own right a general purpose message switching system. Again, TCAM has to drive many different terminal types. We have a limited range and all are driven by a separate computer, the PDP-11. Finally, there are some functions which we have not (initially) provided, such as automatic line numbering and prompting.

TCAM Connection Logic

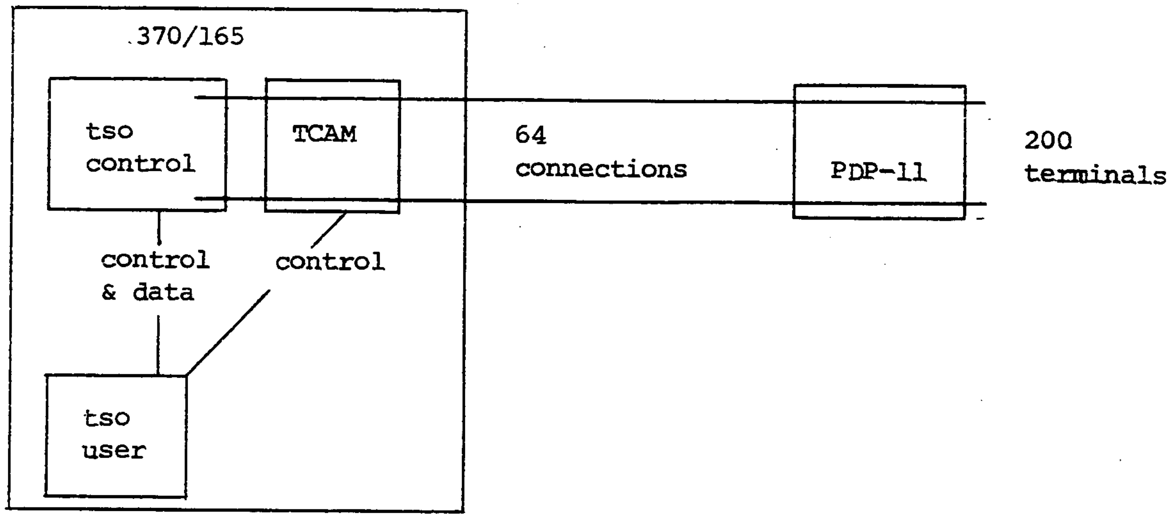


Figure 1

Parrot Connection Logic

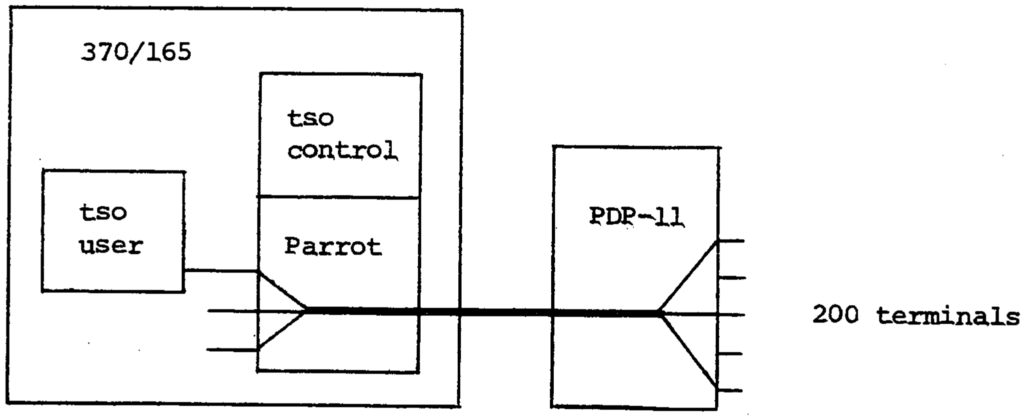


Figure 2

370/165 - PDP-11 transmission block

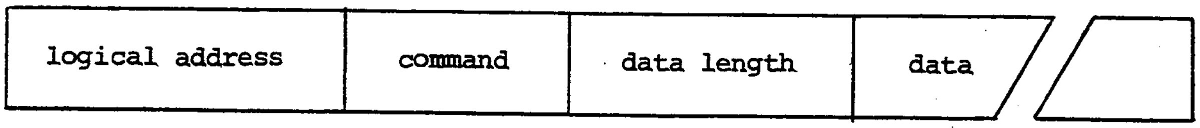


Figure 3

2 The New 370/165 - PDP-11 Interface

The 370/165 part of the software system has been named Parrot. It will be convenient to use this name here. Figures 1 and 2 depict the connection logic for TCAM and PARROT.

Since one of our aims was to reduce the range of 370/165 device addresses required, it was clear that we could not have one per logged on user. At first sight the logical alternative was one for the whole system, but on reflection we opted for two, over which full duplex communication is maintained. To keep MVT happy, these are notionally labelled as a card reader and a line printer, but this is not important. Within the 370/165 all I/O is through the standard EXCP mechanism. The general rule for the interface is that at all times either party should be able to speak. To this end the 370/165 should at (almost) all times keep a read command outstanding on the input address and the PDP-11 should always be prepared to accept data on the output address. If an I/O error occurs for any reason, such as a PDP-11 system crash, the I/O operation is restarted. The necessary flow control occurs at a higher level.

A single I/O operation results in the transfer of a 'transmission block' or 'buffer' of up to 80 bytes (see Figure 3). This block is headed by a 'logical process address' and upon receipt the block is immediately passed to the corresponding 'process'. At this stage a process is to be envisaged in a purely notional manner as a single thread activity of bounded scope. On each side of the interface there is one process devoted to each logged on user, the 'session process'. There is also a 'global process', which deals with events such as restarts.

During normal operation the session process sends transmission blocks solely to its counterpart in the other machine, from whence only it receives them. It is at this level that flow control is exerted. Each buffer sent between session processes contains a 'command' byte, immediately following the address. Not all commands are legal at all times; some depend on the current state.

The following is not intended as a complete description of the protocol, but should be sufficient to give the flavour.

Session output commands (from the 370/165 to the PDP-11):-

Write characters	- the remainder of the buffer contains the characters. This may not be repeated until a 'Write done' command is returned from the PDP-11.

Write characters now	- normally output does not interrupt a partially input line. This does. Notionally needed for urgent messages, actually unused.
Enable read	- After the PDP-11 has sent a data buffer it may not send another until it receives this.
Mend	- the acknowledgement to 'Break'.
Mode read	- read session details (eg line width). Response is 'Mode here'.
Mode write	- set session details. Response is 'Mode write done'.
Disconnect	- terminates a 'session'. The session is abandoned.
Connect	- request connection of a particular terminal. For use after PDP-11 restart.
Go	- enables reconnection for a new session after disconnect or initial startup and also acknowledges PDP-11 restart.

Session input commands (to the 370/165 from the PDP-11):-

Write done	- sic
Input chars read	- The remainder of the buffer contains input. This command may not be repeated until 'Enable read' is received.
Mode here	- buffer contains session details; response to 'Mode read'.
Mode write done	- response to 'Mode write'.
Break	- after 'Break' is sent, the PDP-11 session process ignores all buffers from the corresponding 370/165 process until 'Mend' appears. This overcomes synchronisation problems associated with the full duplex link.
Hangup	- The user has hung up his telephone. After 'Hangup', the PDP-11 automatically disconnects the terminal and ignores all commands until 'Go' appears.

Connect	- a terminal (i.e. a user) has been connected and a session has thus begun.
Input read complete	- as 'Write done', but the last character is a record terminator, i.e. RETURN or ESC.
Go	- acknowledges 370/165 restart.

Individual processes are not permitted to crash without bringing down the whole of the machine in which they are running, an attitude adopted in view of the lack of adequate storage protection in either machine. On the other hand it was felt unreasonable to demand a complete system restart when either only the PDP-11 or only the 370/165 crashed.

After a 370/165 restart, the PDP-11 automatically re-initialises itself, disconnecting all terminals. Users must then log on again. The more interesting case is when the PDP-11 is restarted.

When the PDP-11 is restarted, it loses all memory of its previous life. It announces the restart by means of a command sent to the global process in the 370/165. This process informs all session processes in the 370/165. Each of these session processes then sends a 'Go' command to its PDP-11 counterpart. In the meantime this PDP-11 session process ignores everything until the 'Go' command arrives. This is necessary since the pipeline may contain a substantial backlog of messages relating to activities before the fall was detected. If a corresponding TSO session was previously in progress, the 370/165 session process sends a 'Connect' command to its PDP-11 counterpart to acquire the correct terminal. When all 370/165 session processes have had a chance to reconnect, a global 'Go' is sent to the PDP-11 and normal running resumed. Until the global 'Go' is received, no terminal is permitted to initiate connection.

Confusion can arise since the communication pipeline can in principle hold a queue of several restart announcements and acknowledgements from either or both parties. One has no means of knowing whether an acknowledgement received refers to the most recent announcement made. This problem can be solved if one can generate unique incarnation numbers. We do not do this because in practice the problem does not arise. In the last resort a complete clean start can be made.

As far as was possible, we tried to banish all device dependent code into the PDP-11. Nonetheless, a "problem program" in the 370/165 does occasionally need to know device details and to exert special control. An immediate example is the maximum length line that can be typed. Programs may legitimately wish to produce output in different formats on wide and narrow typewriters, and the user of a dial-up device needs to be able to specify its width. To this end the special functions of 'Mode-read' and 'Mode-write' are provided in the protocol.

3 The 370/165 System - Parrot

One of the difficulties in replacing TCAM is the diffuse nature of the interface between TCAM and the TSO user. There is a misty interface between TPUT/TGET and the TIOC, a muddy one between the TIOC and TCAM, and several tenuous links between TPUT/TGET, TCAM, and the rest of TSO. This was not so discouraging as might appear since we had considerable freedom in deciding precisely where the hatchet was to fall. We were not so much replacing TCAM per se as the terminal I/O system as a whole. The main constraint was that the user interface to TGET and TPUT should remain constant, and since this is by far the narrowest interface in the system it was here that we cut. Internally we also used the standard interface to the TSO driver, and the standard flags and ECBs for signalling RCTs and TSC. QTIP, AQCTL and the TIOC were excised.

The main bulk of Parrot runs as an OS subtask of TSC, and in the same region. The code of Parrot is assembled as a single load module with a single CSECT whose sole entry point is that of the Parrot task. The code for SVC 93 (TPUT/TGET) and SVC 94 (terminal control) is also part of this CSECT, the entry points for these routines being planted in the TSCVT by the Parrot task during initialisation. The SVCLIB modules merely branch to these entry points.

It is clear that within Parrot several logically unconnected activities will take place and so some form of multiprogramming technique is required. Trivial estimates of cost are sufficient to rule out OS multitasking as a solution and so a multiprocess system was built within the Parrot task. The basis of this is the "coordinator", which is a set of routines for regulating the progress of Parrot "processes". A process in this context is rather more concrete than that described in the preceding section. A Parrot process bears the same sort of relation to the whole Parrot task as an OS task does to the whole machine. It is analogous to a Hasp processor. Any one process is normally oblivious of other processes and communicates with them solely through the coordinator. Activities which run asynchronously with Parrot, such as other OS tasks using TPUT, appear to Parrot as interrupt routines or routines executed by an asynchronous cpu.

The Parrot coordinator is strongly based on the coordinator written by M.J.T. Guy for the PDP-11 system.

The rules for processes are mainly dictated by the required economies. One envisages of the order of a hundred logged on users and a comparable number of processes. This means that the main storage dedicated to a process must be minimal and that the cpu time per process should not increase with the number of processes. For example, one cannot afford to store a set of 16 general registers for each process or to scan the entire list of processes at every coordinator entry.

Parrot Process Base

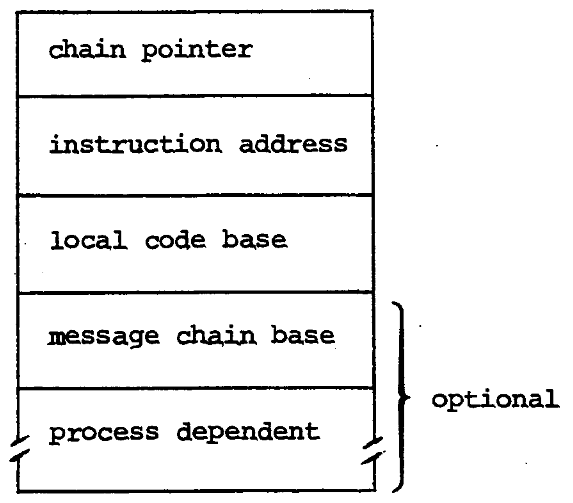


Figure 4

A Parrot process is physically represented by a block of store called a "process base", whose size depends on the particular process (see figure 4). The minimum requirement for a process base is three words, one of which is used for chaining, one for storing the instruction address, and one for storing the local code base register. When a process is running, one standard register, the "process base register", points to the base and one, the "local code base register" is used to address code local to the process. This code is often shared re-entrant code. At all times within Parrot, the "common code base register" may be used to address coordinator code or "central subroutines", and one other register standardly points to the TSCVT. When a process is halted, the only information preserved by the coordinator is that kept in the minimal process base, that is the local code base register and the current instruction address. The necessary corollary of this is that processes may only be halted of their own volition, which they signal by calling either the SLEEP or WAIT coordinator routine. Until such a routine is called, a process may assume that its registers are all its own and will not change under its feet, and further that no other Parrot process will run. Other OS tasks may of course run unless hardware interrupts are disabled.

When one process SLEEPs, the coordinator runs the process whose base is on the head of the "ready queue", first removing that base from the queue. Thus the way to cause a process to be run is to place its base on the ready queue, a function performed by the coordinator subroutine WAKE. It is but a mild simplification to say that the coordinator consists of two routines, CHAIN and UNCHAIN. If the ready queue is found empty, the coordinator enters an OS WAIT. If any external activity, such as TPUT, requires the services of Parrot, it calls the coordinator subroutine KICK, which WAKEs the appropriate process and OS POSTs Parrot.

Most process bases have one other word used by coordinator functions concerned with inter-process communication. The standard method of sending a process a message is to place the message in a buffer and "send" it to the target process, a coordinator routine SEND being provided for this purpose. The action is to chain the buffer to the target process base and wake the target process. When it eventually runs, the target process is expected to look for and unchain any buffer sent to it.

This leads to Buffer Management. There is a central subroutine which may be called at any time to obtain a standard buffer and another to get rid of it. If no buffer is available, hopefully a rare case, the routine signifies this by a return code. The requesting process is then expected to use a coordinator routine to WAIT on the buffer halt queue, the action being to chain the process base onto this queue. Eventually the Buffer Management Process will FREE all the bases on this queue. They should then try again for buffers. This is the only way in which store may be obtained dynamically.

If an external task requires a buffer and none is free it must make its own arrangements. For example, TPUT sets a flag, KICKs a process to get a buffer for it, and OWAITs.

The rest of the process base, which may be of any length, may be used freely by the process, and is naturally preserved when the process is halted.

The main processes are the transmission process, which accepts buffers and writes them to the PDP-11, the reception process, which reads buffers from the PDP-11 and distributes them, and the session processes.

The transmission and reception processes are very simple. For the transmission process the loop is:-

```
Until buffer queue is empty
        Write buffer to PDP-11
        Free buffer
Sleep
Repeat
```

For the reception process the loop is:-

```
Get buffer
Read block from PDP-11 to buffer
     (This is the main halt point.)
Send buffer to addressee
Repeat
```

There is one session process for each slot, that is for each TJB in the system. Thus each logged on user is connected to a unique session process, and there are typically some session processes idle awaiting connection. A session process remains permanently attached to a particular TJB.

The main cycle of action for a session process is:-

```
Sleep
Deal with any transmission blocks that have arrived from the
  PDP-11
If output is available and not in progress,
 send output text to the PDP-11
Unless too much input has accumulated,
 send 'Enable read' to the PDP-11
Check for miscellaneous events such as logoff
Repeat
```

TPUT, TGET (SVC 93) and the terminal status SVC (SVC 94) appear as interrupt processes. The code for these SVCs is written within the general orbit of the session process code, has the same register conventions, and shares some subroutines local to the session process as well as the universally available central routines.

Between the session process and the SVCs, text for transmission either way is buffered in "streams", that is to say it is densely packed into chains of buffers. Streams are handled by the stream handling subroutines, of which the main ones are READ and WRITE. One of the arguments to these routines is a pointer to the "stream base", from which everything knowable about the stream can be deduced. Any stream may be read from the point where the previous read left off, or written to at the tail. Simple estimation indicates that the cpu time spent packing and unpacking the text in streams is negligible.

Streams have a record structure embedded, so that, for example, one may read the shorter of a specified number of characters or to end of record. Such a structure is necessary since TGET operates in terms of records, which are strings terminated by some (variable) end of transmission character at the terminal. On output the record structure is mainly unused, since devices basically only accept strings of characters, one such being NEWLINE. Occasionally "control records" appear in the streams. A control record is identifiable as such independently of the characters it contains. It is used for activities connected with 'Mode-read' and 'Mode-write', such as 'setting terminal width', which must not leap-frog the normal character transfers.

There are two streams associated with each session process, the input stream and the output stream, the stream bases being embedded in the process base. The output is written to by TPUT and read from by the process; the input stream is written to by the process and read from by TGET. The size of a stream is normally restricted. If the input stream becomes over full, the process merely refrains from sending 'Enable read' to the PDP-11. If the output stream becomes full the caller of TPUT is forced into OWAIT in the standard TSO fashion, which is to set a flag in the TJB, a non-dispatchability bit in the TCB, to inform the TSO driver, and exit to the dispatcher. Similarly if TGET has to wait for input to appear in the input stream, it forces the user into standard IWAIT. IWAIT and OWAIT are ended by the session process when appropriate. If the user is still in core the waits are cleared directly. If he is swapped out the driver causes the RCT to clear them.

'Break', which may arrive at any time, causes the process to flush both streams and to signal the event to the user in the standard fashion.

A new feature, Input Mode, has been provided as an aid to efficiency. Much of the time a user is at a terminal is spent merely typing text into the machine, from which no interactive response is expected. This state is ended by typing some particular terminating string, such as "/*". In a standard system this activity results in the user being swapped in at the end of every line, necessarily so since the TSO/TCAM control functions do not know that no interaction is required. One would like to be able to buffer rather more input before swapping in. A user enters input mode by calling SVC 94 with a particular function code and pointing to a character string to be used as the terminating string. While input mode is set the user is swapped in only when the input stream becomes reasonably full. Input mode is normally terminated when the user types the terminating string. It is also ended by 'Break' or by a different entry to SVC 94.

No great difficulties were found in interfacing to the rest of the world. Such as were found were all the result of OS not using the proper interfaces. For example, OPEN for a terminal data set needs to know the line size. As supplied it does not use the proper interface (i.e. the GTSIZE macro, which expands into SVC 94); instead, it rousts about in TSO's control blocks. Less disreputably, but quite as confusingly, there are two modules in which this happens. The obvious one, the access method executor, is documented but the code is never exercised. The other, in the main line of OPEN, is not documented and is frequently exercised.

In the design stage, most of the problems arose from the inadequacies of OS inter-task communication and also its expense. Sometimes facilities are basically adequate but too expensive: such is ENQ. At other times they are inadequate, such as in WAIT/POST, which provides no help in the case when one party may vanish or be otherwise unreliable. The classic boner is inter-partition POST, which if used incautiously can bring the RCT down.

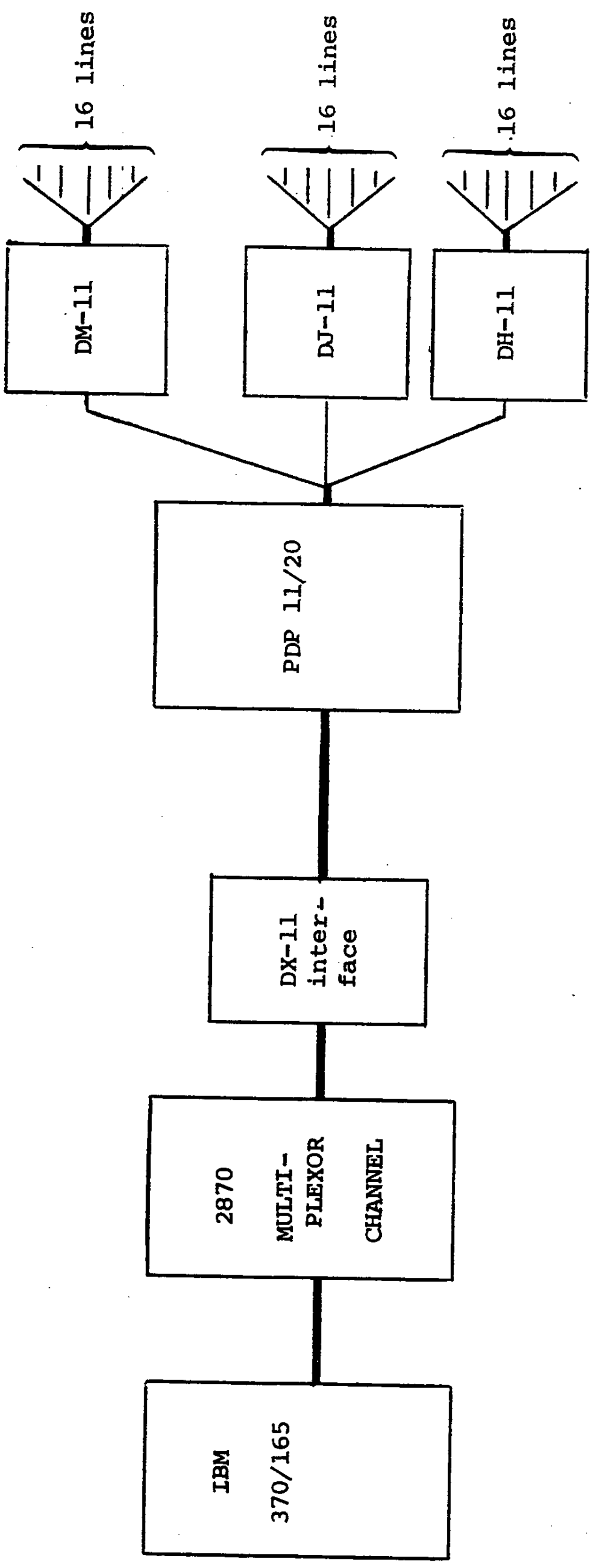

16 lines
16 lines
16 lines
DM-11
DJ-11
DH-11
PDP 11/20
DX-11 inter-face
2870 MULTI-PLEXOR CHANNEL
IBM 370/165

Figure 5

4 THE PDP-11 SYSTEM

4.1 Hardware

The PDP-11 hardware consists of a PDP 11/20 processor with 28K words of store, a DX-11 control unit to channel interface, and a number of synchronous and asynchronous line drivers. There are also paper tape readers and punches, a Tri-Data Cartrifile, and a local operator's typewriter.

The synchronous lines are used for HASP Remote Job Entry Workstations, which do not fall within the scope of this Report. The asynchronous line drivers are in the form of multiplexors, each driving 16 lines. The current configuration is:-

8	x	DM-11	multiplexors, all working at 110 baud
1	x	DJ-11	multiplexor with 8 lines at 1200 baud and 8 lines at 134.5 baud
5	x	DJ-11	multiplexors working at 110 baud
1	x	DH-11	multiplexor with all lines variable between 110 and 300 baud inclusive

Five of the lines on the DH-11 are connected to the public telephone network via Post Office Modems. One of each multiplexor type is shown in figure 5.

4.2 Software

The overall design of the PDP-11 software is due to M.J.T. Guy. The system is conceived as a number of processes and interrupt routines, some of which are interdependent. The processes are run under the control of a simple coordinator which contains much the same facilities as the Parrot coordinator described above. Processes that have been "woken up" are placed on a ready queue, and are run one by one when they reach the top of the queue. The most frequent cause of awakening is the arrival of an interrupt. However, a process sometimes needs to wait for some other external event. For example, it may be unable to proceed until buffers are available. In this case it places itself on a halt queue associated with the event in question. When the awaited event happens, the halt queue is freed, i.e. transferred to the ready queue. Processes run at the lowest hardware priority, so that any interrupt takes precedence. A process runs until it causes itself to be halted, which is a useful interlocking feature for inter-process communication. The information required by the coordinator in order to manage a process is kept in a control block called a process base, at which a standard register is pointed whenever the process is entered.

A number of central subroutines are provided for common use. These include routines for maintaining chains of buffers and process bases, which run in "disabled" state for interlocking reasons. The chains are one-way circular chains, with the base pointer pointing to the LAST buffer. The chain field in this buffer points to the FIRST buffer. It is thus a simple matter to find either end of the chain; in most cases this can be done in a single PDP-11 machine instruction.

4.3 Store Management

The store management routines manage a contiguous area of store. Requests to allocate and free blocks of arbitrary size are supported, but these facilities are currently used only during initialization. During normal running store allocation takes place from two free chains of fixed length buffers, of lengths 12 and 88 bytes respectively. The short buffers are used for characters transferred to or from a terminal, and the longer ones for transfers to or from the 370/165. Store management subroutines are provided for obtaining and freeing buffers. There is a store management process which is woken up at suitable times to replenish the supply of buffers from the free store if it gets low, or to return some buffers to the free store if there is a surplus. A process which requests a buffer and is refused must place itself on the Store Manager's halt queue. The entire queue is freed when the store becomes available, and each process must repeat its request.

4.4 Timing facilities

A special process called the "one second process" is woken up every second by the clock interrupt routine. Its sole task is to free its halt queue. Other processes may thus request to be woken up at the next clock "tick".

4.5 Interface to the 370/165 channel

The 2870 Multiplexor Channel performs I/O between 370/165 main storage and peripheral "devices". In this section we use the word "device" to mean a logical peripheral with a unique address as seen from the 370/165. In practice, what are actually connected to the channel are a number of "control units", which may in general control more than one device each. The channel, however, works entirely in terms of "device addresses", which lie in the range 0-255.

The 370/165 initiates a data transfer by executing a Start I/O (SIO) instruction, which specifies the address of the device in question. Before doing this, a Channel Program must be set up in 370/165 main storage. This consists of one or more Channel Command Words, each of which contains a command byte, data pointer, and data length. The channel operates autonomously under the control of the channel program until it completes, or an error occurs. After each channel command is executed, the device sends a "status" byte which indicates whether the command was successful or not.

The PDP-11 behaves as a control unit with a number of different devices on it. The DX-11 control unit to channel interface is the hardware connection between the PDP-11 and the channel. The DX-11 driver routines, which were written by C.J. Cheney, control the DX-11 on behalf of all the devices. They consist of an interrupt routine and a process.

Inside the PDP-11, an active device is always associated with a "device process". In addition to the coordinator information in the process base, there are additional fields for the use of the DX-11 driver. These include the 370/165 hardware address for the device, the current command byte, the command status, and the address of an "interrupt vector", which contains pointers to subroutines that are called asynchronously from the DX-11 interrupt routine whenever there is activity on the channel which changes the state of the device, such as the receipt of a command. There are two device processes associated with the Parrot interface -- one for transferring data in each direction. (However, these are not the only devices in the PDP-11 system -- the RJE lines, 370/165 Operator Console typewriters, and paper tape I/O have their own devices, all of which use the same channel interface simultaneously.)

A typical sequence of events which occurs during the processing of a command is as follows:-

a) The 370/165 obeys an SIO instruction to start a channel program for the device, and the channel fetches the first channel command word.

b) The DX-11 hardware recognizes the address propagated by the channel, acknowledges, and accepts the command byte.

c) The "command received" subroutine for the device is called from the DX-11 interrupt routine; this wakes up the device process.

d) The device process, when it starts to run, decodes the command.

e) For a read command, the process waits until data is available. Buffers are then sent to the DX-11 process for transmission to the 370/165. When a buffer has been transmitted, it is sent back to the device process. When all the data have been sent, a DX-11 subroutine is called to terminate the transfer.

f) For a write command, the process must send to the DX-11 process an empty buffer which is returned with data from the channel. This is repeated as many times as necessary. A device interrupt subroutine is called when the channel count is exhausted, and the transfer is terminated as before.

Slight variants on the above occur when events such as System Reset interrupt normal running.

The two Parrot "device processes" are little more than a software multiplexor and a de-multiplexor. The former receives buffers from the 80 "user processes" and sends then to the DX-11 process in response to read commands from the 370/165. Each transfer consists of one (not necessarily full) buffer only, with a maximum data length of 80 bytes. The latter device process receives buffers from the DX-11 process as a result of write commands, and passes them on to the relevant user process as indicated by the Parrot address fields in the buffers. Again, each transfer consists of a single buffer. This device process also acts as the "global" process. Global restart commands and System Resets are sent to ALL user processes. The communication between the device processes and user processes is entirely one-way.

The flows of data within the PDP-11 are thus:-

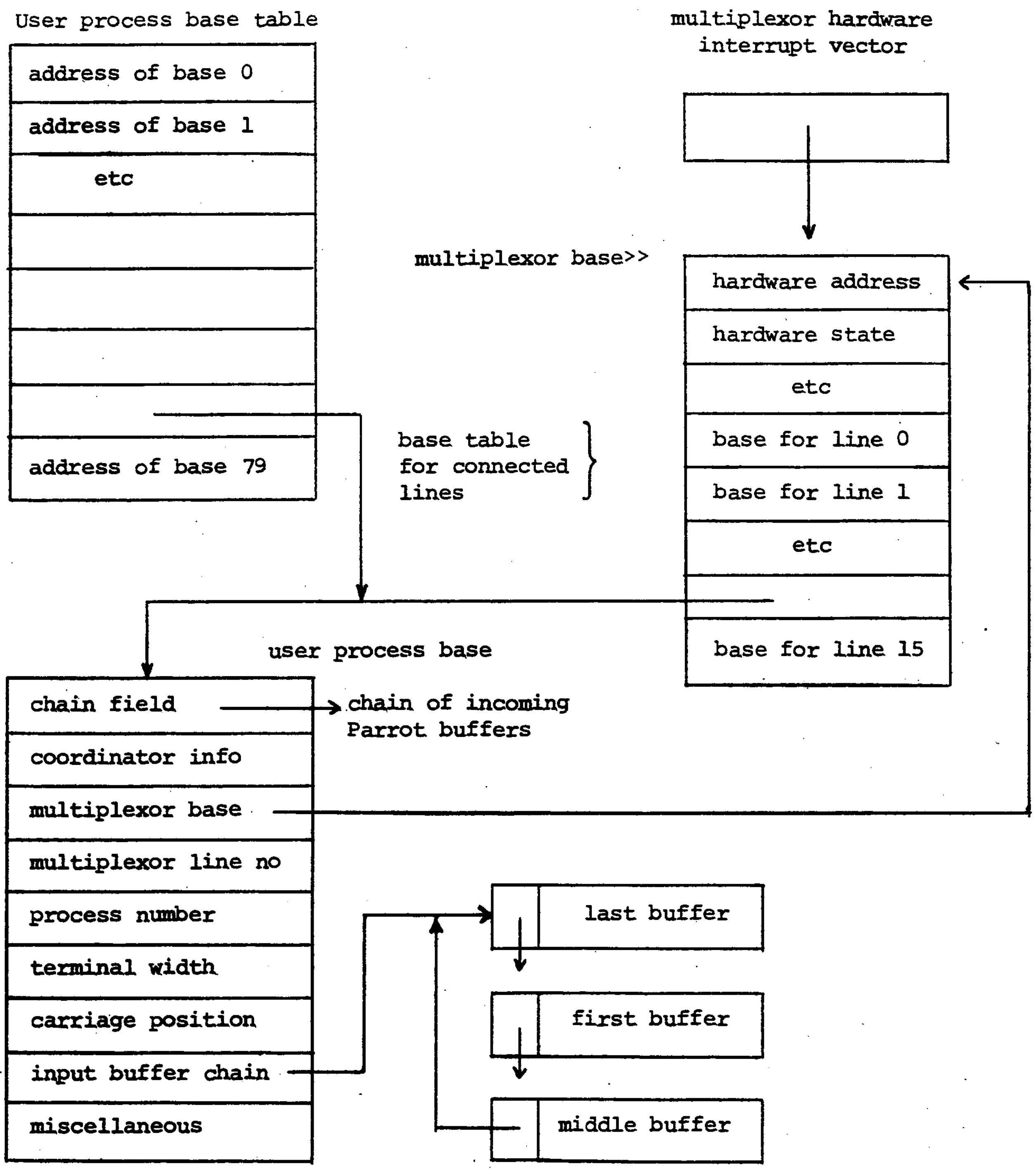
```
Output:   370/165 -> DX-11 process -> 370/165 output device process
                     -> user process -> terminal

Input:    terminal -> user process -> 370/165 input device process
                     -> DX-11 process -> 370/165
```

4.6 User Processes

The 80 user processes in the PDP-11 communicate with their corresponding session processes in the 370/165 on the one hand, and with physical terminals on the other. At system start up or immediately after LOGOFF (i.e. after a Parrot 'Disconnect' command), a user process enters a disabled state until 'Go' is received. This command makes the process available for connection to a terminal.

When a user wishes to log on to the Phoenix system, he types RETURN on his terminal. In the interrupt routine which is then entered, the table of user processes is searched to see whether there is one available for connection. If not, a suitable message is written to the terminal and the transaction is completed. (It is also possible to run the system in a testing mode in which only terminals that are situated in the computer room are allowed to log on. This mode can be set or unset by commands from the 370/165.) If a process is available for connection, the device is connected to it and it is woken up.

A simplified picture of some of the data structures used in controlling a process which is connected to a terminal is shown in figure 6.



User process base table
address of base 0
address of base 1
etc
address of base 79
multiplexor hardware interrupt vector
multiplexor base>>
hardware address
hardware state
etc
base for line 0
base for line 1
etc
base for line 15
base table for connected lines
user process base
chain field
chain of incoming Parrot buffers
coordinator info
multiplexor base
multiplexor line no
process number
terminal width
carriage position
input buffer chain
miscellaneous
last buffer
first buffer
middle buffer

Figure 6

When it is first woken up after a terminal is connected, the process causes the message "LOGON" to be written, waits for it to complete, and then waits for the user to type his userid. If this is not done within 30 seconds, the message "***ABANDONED" is written, the terminal is disconnected from the process, and the process base is marked as being available for connection again. This also happens if BREAK is pressed. Only when the process has a complete line in hand does it send a 'Connect' command to the 370/165. The data sent with the 'Connect' command includes the PDP-11 hardware address and the terminal width. The usual response is an 'Enable read' command, which causes the logon line to be transmitted. The terminal then remains connected to the process until a 'Disconnect' command or a System Reset is received.

When the user has completed a line of input, no further input is accepted until the complete line has been transmitted to the 370/165. Output received from the 370/165 is held for up to 60 seconds if the user is in the middle of typing a line of input. If the input line is not completed in this time, it is abandoned (!!RETURN, LINEFEED being output). If BREAK is received from the device at ANY time, all the current input and output in the PDP-11 is abandoned, a 'Break' command is sent to the 370/165, and ! written to the terminal. The process ignores all subsequent activity by the terminal and all commands from the 370/165 until 'Mend' is received. If a user on a dialled line hangs up, 'Hangup' is sent to the 370/165, and the process disconnects from the terminal automatically.

4.7 Terminal handling

The majority of asynchronous terminals connected to the PDP-11 operate in ASCII code in full duplex mode, i.e. all output on the terminal, including the reflection of input, is under the PDP-11's control. The system also supports half-duplex ASCII devices and IBM 2741 terminals; for simplicity, however, only the handling of full duplex ASCII devices will be described here.

When a character is received it is immediately reflected back to the terminal. The character is then checked for validity, and if it is invalid, a BELL character is "reflected" as well. Valid characters are translated to EBCDIC code and stored in short (12 byte) buffers. Completed buffers are chained together up to a maximum of 256 characters. The input line can be terminated by RETURN, which causes "carriage return, linefeed" to be sent to the terminal, or by ESC, which leaves the carriage where it is.

The character @ is used as an "escape character" when reading input. It signals the PDP-11 that the next character is to be treated in a special way. This mechanism is used for entering characters which do not exist on the terminal, and also for requesting special control functions from the PDP-11. Examples of the first use are @V for |, @- for _, @T for ~, @@ for @ itself, and @Xnn for the EBCDIC character with value X'nn'. The control functions include

@C	cancel the previous character
@L	cancel the whole line
@Y	convert subsequent letters to lower case
@U	convert subsequent letters to upper case
@RETURN	concatenate this line with the next as one input line
@I	interrupt -- has the same effect as BREAK
@Z	suppress character reflection until the end of the line
@D<char>	change device characteristics (see below)

The character RUBOUT has the same effect as @C (which it reflects). @L and @RETURN both cause RETURN, LINEFEED to be sent to the device. The @D combination signals a change in terminal characteristics to the PDP-11. Normally the characteristics of devices on fixed lines are known, so its main use is on dialled lines. The following options are supported:-

@DB	ignore BREAK characters
@DD	revert to default characteristics
@DF	change to full 96-character ASCII code
@DH	change to half-duplex working
@DR	ignore RUBOUT characters
@DS)	(insert idle characters after RETURN and/or LINEFEED;
@DT)	(the different letters request different numbers of
@DU)	(idle characters

@DB and @DR are provided to help overcome problems on noisy lines. The effects of BREAK and RUBOUT can always be achieved by typing @I or @C respectively. @DS, @DT and @DU are provided for terminals with slow carriage return mechanisms.

Output buffers for the terminal are translated from EBCDIC into terminal code, illegal characters being converted to ?. There is a special output character, GRAPHIC ESCAPE, which causes the following character to be output untranslated. By this means, control codes can be written to VDU's. The PDP-11 maintains a pointer to the current carriage position, and inserts a newline in the output if the right hand margin is exceeded. The right hand margin value can be changed by the 370/165 via the 'Mode set' command, and read via the 'Mode read' command.

While output is taking place at the terminal, input is not permitted and any characters received are ignored, with two exceptions:-

a) Typing BREAK or @I causes the rest of the output to be abandoned and subsequently a 'Break' command is sent to the 370/165.

b) Typing SPACE causes the PDP-11 to pause at the end of the next line of output. The pause lasts 60 seconds. It can be cancelled by typing @L, or alternatively by typing in a line of input, which will be sent to the 370/165 while subsequent output continues.

5 Glossary of IBM terminology

AQCTL	Routine for communication with TCAM.
CSECT	Program segment.
Dispatcher	OS task coordinator.
Driver	The program which does the short term scheduling within TSO.
ECB	Event Control Block, used by WAIT and POST.
ENQ	An interlock and reservation mechanism.
EXCP	The supervisor call for initiating I/O on a channel.
IWAIT	The state of being halted waiting for terminal input.
MCP	Message Control Program.
MVT	An IBM multiprogramming supervisor program.
OWAIT	The state of being halted waiting for terminal output to take place.
QTIP	TSO/TCAM communication routine.
POST	Ends WAIT.
RCT	Region Control Task. The task which connects and disconnects users from the rest of the operating system when they are swapped in and out of main store.
SVC	Supervisor call.
SVCLIB	Library of SVC routines.
TCAM	Telecommunications Access Method. An IBM system normally required for driving TSO terminals.
TCB	Task Control Block.
TGET	The supervisor call by which a problem program reads from a terminal.
TIOC	Terminal I/O Coordinator. A buffering interface between TPUT/TGET and TCAM.
TJB	A control block regulating a single user's session.

TSC	Time Sharing Control. The top task of the TSO subsystem. The task which actually performs the swapping.
TSCVT	The central control block and address table for TSO.
TPUT	The supervisor call by which a problem program writes to a terminal.
TSO	Time Sharing Option. An IBM multiple access system.
TSIP	An interface to the TSO Driver (q.v.).
WAIT	Suspend current task.