Edinburgh
Regional
Computing
Centre

DEIMOS: User Manual

A guide to the use of DEIMOS,
an operating system for the PDP11.

by B.A.C. Gilmore

Edited by J.M. Murison

3rd Edition: January, 1980

DEIMOS

AN OPERATING SYSTEM FOR THE PDP 11

USER MANUAL

by

B.A.C. Gilmore

Edited by J.M. Murison

1st Edition: August 1976
2nd Edition: May 1978
3rd Edition: January 1980

CONTENTS

GENERAL FEATURES

Design Aims

DEIMOS was designed for operation in a medium-sized PDP 11. At least 16K words of main store, a memory management unit, a disc or similar fast mass storage device, a terminal and a clock are required.

The system was designed with six main aims:

1) To run user programs

The system is designed to run general user programs. Normally, about twenty simultaneous programs can be supported, but this figure is a parameter at system generation. Each program runs in its own virtual memory environment (VM), not necessarily limited to the hardware's mapping limit of 32K words. The system, and other user programs, are fully protected from the failure of a user program.

2) To support multiple terminals

The system supports multiple terminals, each of which can be linked to a command language interpreter which will enable the user to initiate and control programs from the terminal.

3) To support peripherals

The system supports a wide range of peripherals, e.g. line printer, card reader, paper tape reader and punch, various discs, magnetic tape, graph plotter, asynchronous communication lines and a synchronous communication line using the HDLC protocol. New peripherals can be added with minimum disturbance to the rest of the system.

4) To be self-supporting

The system is self supporting on a medium main store configuration (approximately 32K words). Although the system only occupies about 6K words, 32K words are required to run the IMP compiler.

5) To use swapping

The number and size of user programs is currently limited to the physical store size of the machine; however, a limited swapping strategy will be implemented to support a virtual store size of two to three times the physical store size.

6) To have a small resident section

The size of the resident system is kept small to allow as much store as possible for user programs.

Constraints

The resident part of the system has been kept reasonably small to enable the system to run in small main store configurations. This constraint affects the overall design, distinguishing it from systems with a large set of facilities like UNIX and RSX11D, which require at least 48K words of main store to do useful work.

Structural Overview

The system is based on the concept of a number of tasks, each of which manages either a resource or a user program, and which communicate and synchronise with each other using 'messages'. The concept of messages comes from EMAS, the GEC 4080 and other systems.

The user interface is heavily influenced by a number of machines running in the Computer Science department.

The system has two main sections, a resident section and a potentially swappable section.

The resident section consists of a kernel and the mass storage device handler which runs otherwise as a standard system task.

The kernel does the following:

1) Controls the CPU allocation.

2) Passes interrupts to their device handlers.

3) Passes messages between tasks, storing them if necessary.

4) Supports the virtual memories, including mapping between them.

5) Provides clock and timer functions.

6) Controls main store allocation.

All peripherals and other system functions, e.g. the file storage handler, command language interpreter and loader, are handled by system tasks. The system tasks are 'privileged'; this entitles them to access parts of the real machine and other tasks.

A new task is created when a user program is run, and is deleted on its termination. A task consists of a virtual memory environment and a 'task descriptor block', held within the kernel. A task on this system does not have a 'task monitor': all interrupts and messages are processed by the task in its 'user' state.

The virtual memory environment of a task consists of a number of segments. These are used to hold the program code, data areas and shared system code. The hardware of the PDP11 allows eight segments to be mapped onto real store at any given time, giving a virtual memory address space of 32K words. However, the number of segments owned by a task is not limited to eight. A segment may be mapped in a read only mode or in a read/write mode.

The 'task descriptor block' contains the registers (when the task is not actually executing) and other information such as the state, priority level and message queue that constitutes the context of the task.

The list of segments used by all tasks is held in a Global Segment Table within the kernel, with the main store or disc address, access permission and the number of tasks using the segment. This table enables the kernel to maintain control over the usage of segments and can easily determine what parts of tasks may be swapped out. The main store address, access permission and a pointer into the Global Segment Table are also maintained within the task descriptor to optimise the context switch.

If a task fails with either a hardware fault (e.g. an address error or memory protection violation), or a fault detected by software (e.g. an illegal supervisor call or message), then the kernel generates a message to a 'system error task'; to allow later investigation the failed task is prevented from continuing. The 'error task' informs the user of the task failure and the reason for it. The 'error task' is also used by some system tasks to inform the operator about the state of devices.

All communication in the system is done by sending messages. These messages are queued by the kernel, if necessary, until requested. Interrupts are handled similarly, the kernel generating and queueing a message for the appropriate task. A table is used to determine which task a message (or interrupt) is for. A supervisor call is provided to enable tasks to 'link' themselves to a particular message number. This is slightly less efficient than direct ownership but enables device handlers to be configured into the system dynamically.

The address of a data area may be passed by a message. The segment containing this area may then be mapped from the caller's VM to the receiver's VM. Currently this mechanism is only used to share segments, which are eventually returned to the caller. There is no restriction, however, to stop segments actually being transferred by this method.

Input/output on this system uses a separate segment in each user's task to hold its I/O buffers. This allows the kernel to swap the major part of a task whilst slow I/O is in progress. The sharing of segments, as described above, is used by the device handlers to process the buffers, the segment being released and a reply sent on completion.

Implementation

This system was written in IMP. IMP was chosen for a variety of reasons:

1) A deliberate decision was made to write the system in a high level language for ease of expansion and maintainability.

2) IMP is a proven systems implementation language; for example, it was used to write both the System 4-75 and 2900 versions of EMAS.

3) A new implementation of IMP was available on the PDP11 which was of a high enough standard to consider using it for systems implementation.

Two modules of the system have been written in assembler. The first module is at the lowest level of the system, loading up the registers on context switching. Since there are no explicit register manipulations in IMP, it was necessary for this module to be in assembler. The second assembler module provides the run time support for IMP programs. This module could probably be converted to IMP later, but was written in assembler for bootstrapping reasons. Fortunately, these two sections are changed infrequently as they have proved to be a disproportionately small source of problems in relation to their size.

The rest of the system consists of six IMP modules, comprising the kernel and the system tasks. These modules are compiled separately and then 'linked' by a purpose-built Linker which also sets up the bootstrapping area.

Applications programs, with the exception of the editor - which was brought from a previous system - have been written in IMP. The sources are held and compiled on the system.

Operation

The first DEIMOS system, operational since May 1976, is used for a large spooling RJE system with the usual peripherals plus a magnetic tape drive and a graph plotter.

The second system, also operational since May 1976, is part of ERTE (Edinburgh Remote Terminal Emulation). See the paper "The Structure and Uses of the Edinburgh Remote Terminal Emulator" by J.C. Adams, W.S. Currie and B.A.C. Gilmore, in 'Software Practice and Experience', Volume 8, p. 451 (1978).

The third system, on a PDP 11/34 used as the Front End for the ERCC 2970 running EMAS 2900, became operational in December 1977.

The fourth and fifth systems are used part time as the drivers for an interactive benchmark of two ICL 2900s running VME/K and VME/B.

At this stage swopping has still to be implemented, though most of the necessary kernel features are already present.

All commands given to the system are interpreted by the Command Language Interpreter (CLI). The CLI indicates its readiness to accept a command by typing the prompt '%'. If a program is running, the CLI can be invoked by typing ESC (escape); the command prompt will then appear.

Input to the CLI has two forms:

1) A file name, followed by 'stream definitions' or

2) A command verb, possibly followed by parameters

---

1) File Name

If a file name is specified, that file is loaded and the program contained in it is entered. If the file does not exist, or exists but does not contain a program, the message '*NO FRED', where FRED is the name of the file, is output and the command prompt is reissued.

The 'stream definitions' are input in the form:

<input 1>,<input 2>,<input 3>/<output 1>,<output 2>,<output 3>

where <input 1>, etc. each represent a 'stream definition'. One or more spaces must be given between the file name and the stream definitions; no spaces may appear within the stream definitions.

A stream definition is of the form:

.TT - either input to or output from a terminal
.LP - output to a line printer
<any other devices on the system, specified in a similar way>
<a file definition>

A file definition is of the form:

<unit number>.<file name>(<file system number>)

<unit number> is a digit in the range '0' to '4' and refers to the physical disc drive. If neither is specified, '0' is assumed.

<file name> consists of an alphanumeric string, which must start with a letter, of up to 6 characters in length

<file system number> is the file system that the file comes from, or is created in. If none is specified the user's own is used (see the command LOGON).

Examples

FRED	the file called FRED on unit 0 in the user's file system
0.FRED	the same
1.FRED	the file FRED from disc unit one
FRED(0)	the file FRED from file system 0 (the system's own one)
1.FRED(25)	the file FRED from disc unit 1 and file system 25 (octal)
.TT	the user's terminal, defined for input if on the left hand side of the '/', and for output if on the right hand side

The stream definitions associate files or devices with the corresponding input and output streams, which are accessible to an IMP program. For example, if the program JOE is run with the following command:

```
JOE F1,F2/O1,O2,.TT
```

then input stream 1, selected within the program by use of SELECT INPUT(1), is mapped onto the file F1. A SELECT OUTPUT(2) will send output to the file O2, and SELECT OUTPUT(3) will send output to the terminal.

If a field is left blank, as with <input 3> in the above example, the stream is mapped to .NULL. This causes, on input, an 'end-of-file' signal to be generated, and on output all the output to be thrown away.

Input stream zero and output stream zero are always associated with the user's terminal.

Further examples

PROG .TT	uses one input stream, .TT; no output stream (apart from 0) is defined
PROG FILE	will read from a file called FILE on stream one
PROG /FILE	no input streams defined (apart from 0); will write to a file called FILE after a SELECT OUTPUT(1)
PROG FILE/FILE	will read data from a file called FILE and create an output file called FILE

Note: When a program does the first SELECT OUTPUT(n) for a perticular value of n, where a file is mapped to that stream, the system creates a temporary file with the same name as the defined file but with a '#' appended (thus FILE# in the previous example). When the program stops, or calls CLOSE OUTPUT(n), the original file, if one exists, is destroyed and the temporary file RENAMEd to the original file's name.

2) Command Verb

There are currently available the following system commands:

a) LOGON - Log on to a file system

b) TASKS - List the names of the programs currently executing

c) HOLD - Stop a task executing

d) FREE - Release a 'held' task

e) KILL - Stop a program in a controlled manner

f) PURGE - Immediately remove a task from the system (i.e. abort the task)

g) ABORT - Abort a device handler transfer

h) INT - Interrupt a task

---

a) LOGON

This command is called in the form:

LOGON nn

where 'nn' is a two digit octal number, representing the user's file system number.

b) TASKS

This command has no parameters. It lists out the tasks on the system in the form:

<task name> <task number> <task state>

The states of a task are as follows:

state	meaning
000001	Task is in the WAIT state
000002	Task has executed a POFF
000010	Task is on a CPU QUEUE awaiting execution
000020	The task is in the RUN state
000200	The task has been HELD

c) HOLD

This command is called in the form:

HOLD xxxx

where 'xxxx' represents the name of a task running on the system. If the task is currently executing, it is suspended. If the task is waiting for a message, it will be suspended when it next requests the CPU.

d) FREE

This command is the converse of HOLD and is called in the form:

FREE xxxx

It places the task 'xxxx' back on the CPU.

It is also used to tell a device handler to continue after a device error. For example, if a task attempts to write to a disc while the 'write protect' is on, the disc task will accept an error and wait. The user may put the 'write protect' off, and then type 'FREE OK' to tell the disc handler to retry the transfer.

e) KILL

This command is called in the form:

KILL xxxx

It has the effect of sending to the task called 'xxxx' a message to stop. The task should then stop in a controlled manner, tidying up all its streams.

Note: If a program is still running, HOLD should be called before either KILL or PURGE is used.

f) PURGE

This command is called in the form:

PURGE xxxx

It has an immediate effect, completely removing the task from the system. No attempt is made to tidy the open streams. This command should not normally be used to stop a program, but must be used to clear from the system a program which has crashed with an 'ADDRESS ERROR' or a 'SEG FAULT'.

g) ABORT

This command is called in the form:

ABORT xxxx

It is the equivalent of FREE, except that it informs task 'xxxx' to fail the transfer request instead of retrying it (see FREE).

h) INT

This command is called in the form:

INT y xxxx

It will place the single character 'y' in a fixed location in task 'xxxx'. This may be used to control the running of a task.

A task may examine and change its INT character by declaring:

constbyteintegername INT = K'160060'

and then using INT as a simple variable.

Processes may communicate with other processes by sending and receiving messages. A message consists of four 16-bit integers that are passed from one process to another. The general format may be expressed as an IMP record, as follows:

```
record format PF (byteinteger SERVICE, REPLY, integer A, B, C)
```

The two byte integers are used for message routeing and are defined below. The three integers are used in various ways, their uses depending on the context.

An example of an alternative format used is:

```
recordformat P2F (byteinteger SERVICE, REPLY, FN, PORT, c
        record (BUFF) name BUFFER, byteinteger LEN, SPARE)
```

A process sends a message to another process by setting SERVICE to the service number of the other process (the 'destination' process). REPLY should be set to the service number to which the reply to the message (if any) is to be sent. When a process is started up, the system assigns a unique process identification number to it. A copy of this number is held by the system in a fixed location within the virtual memory of the process. It may be accessed by:

```
constbyteintegername ID = K'160030'
```

This ID is the service number of the process. For processes which require more than one service number, or which provide a service to other processes, (e.g. a line printer handler which requires to have a 'fixed' service number), further service numbers can be claimed by declaring system routine LINKIN:

```
systemroutinespec LINKIN (integer SERVICE)
```

and then calling it to clain the service number. For example:

```
constinteger LP SER=25
    .
    .
LINKIN(LP SER)
```

The number supplied to LINKIN must lie in the range 1-30. A list of commonly used numbers is given in Appendix 1.

There are two system routines that are used to send messages. They are predeclared, and so may be used without a spec. Their form is given for information.

```
systemroutinespec PON (record (PF) name P)

systemroutinespec PONOFF (record (PF) name P)
```

The call of PON, for example, would be the form:

```
record (PF) P
    .
    .
P_SERVICE = LPSER; P_REPLY = ID
! P_A, P_B and P_C should be set up as required.
PON (P)
    .
    .
```

The effect of PON is to send the message in P and then to return control to the process to let it continue. When an immediate reply is required, PONOFF should be used: the process is then held until the specific reply to that message is sent back to the process; this is actually implemented by holding the process in suspension until a message is sent whose SERVICE and REPLY match those of the original message. A message is received by calling the predeclared system routine POFF. Its declaration, again given for information, is as follows:

```
systemroutinespec POFF (record (PF) name P)
```

Messages for a process are queued by the system. A call to POFF will cause the system to deliver to the process either the first message in the queue, or optionally the first message for a particular service provided by that process. If there is no message of the appropriate type in the message queue the process is suspended until a suitable message arises.

When a process wishes to receive any message, it calls POFF as follows:

```
P_SERVICE=0
POFF(P)
```

To receive a particular message, for example with SERVICE=ID and REPLY=LPSER, the following code is used:

```
P_SERVICE = ID; P_REPLY = LPSER
POFF(P)
```

BUFFER SHARING BETWEEN PROCESSES

In many instances it is necessary to pass more than 6 bytes of data to another process. A method of allowing processes to share segments of internal memory has therefore been provided.

A predeclared system routine of the form:

systemroutinespec MAP VIRT (integer HIS SER, HIS SEG, MY SEG)

is used to map segment number HIS SEG of process number HIS SER into segment number MY SEG of the running process. HIS SER may be the process id set by the system, or any service number which the process has declared (by a call to LINKIN). It should be noted that the whole segment is mapped in the same mode as the original segment.

For example, if a process with SER=a sends a buffer at its address 8_140322 (segment 6) to a process with ID=b, then process b may access the buffer by:

```
MAP VIRT(a, 6, 4)
```

The buffer will start at address 8_100322 (segment 4) in process b.

For processes with only one segment of code and one segment of stack, segments 3, 4 and 5 are available for other process's segments to map into.

When a process is finished with a segment it should release it, by calling MAP VIRT with HIS SEG set to zero.

THE EDITOR

The EDITOR is a PDP11 version of the Edinburgh Compatible Context Editor (ECCE). For general information on the Editor, see the document 'ECCE', published by ERCC (August 1978).

The command for calling the Editor has four possible forms, as illustrated below:

E /TEST	- to produce a new file called TEST
E TEST	- to edit an existing file called TEST
E TEST/TEST2	- to produce a new file called TEST2 from an existing file called TEST
E TEST/.TT	- used to examine, without modification, the file TEST

Notes

* The Editor prompts '>' when it is ready to accept a command and ':' when it expects a line of input (command GET).

* This version of the Editor uses a 'window' of the file. This will not normally be apparent, but it does mean that the command M-* will not necessarily return right to the top of the file.

WARNING

If you wish to terminate the Editor abnormally, always use the command PURGE rather than KILL. For example, if you are editing a file back into itself, as in the second form above, a KILL will delete the file TEST, and replace it with the temporary editor file which may well be empty.

THE IMP COMPILER

The IMP compiler is a three pass compiler; a fourth pass, a linking phase, is automatically called for begin ... endofprogram programs.

There are four main ways of calling it:

IMP A/B - which compiles source file A to object B

IMP A/,L - which only does one pass and creates a listing file L; note that L can be a filename, .TT or .LP

IMP A/B,L - which creates both an object file and a listing file

IMP A,.TT/B - this form is used to change the Linker defaults; for more details of these defaults see the section on the Linker, below

Notes

1) At present, the compilation of an external routine file will cause a 'FAULT 100' on exit from the third pass; this inhibits the entry to the Linker. The object file in this case is always called 0Y.

2) The first pass automatically uses a file of specs called PRIMS which is taken from the system's file system.

3) The first pass creates a temporary output file called 0 (the letter, not the digit) for use with the second pass. The second pass creates 02 and 03 and the third pass 0Y.

4) The output from the third pass, 0Y, is useful. It can be used as input to the programs RECODE and VIEW if it is necessary to de-compile the compiler output.

5) If the Linker is to be run as a separate fourth pass, the file 0Y is used as its input.

THE LINKER

The Linker is normally run automatically as a fourth pass to the IMP compiler. It is run individually by typing one of the following four forms of command:

```
    LINK A/B
or
    LINK A,.TT/B
or
    LINK A/B,C
or
    LINK A,.TT/B,C
```

All of the above forms use the file A (which must be the output from the third pass of the compiler) to create a runnable file B.

The second and fourth forms of the command override the standard Linker defaults. The Linker prompts for the new values as follows (defaults in brackets):

NAME: up to 4 characters are typed in, giving the Task Name for the program
(default: the first four characters of the object file name)

STACK: the desired stack size (in octal bytes), excluding the GLA
(default: 14000)

STREAMS: the maximum number of input and of output streams that the program will use, excluding input and output on stream 0
(default: 3)

The Linker will normally assign the code and stack segments as indicated in the table below. The Linker automatically uses segments 3 and 5 as overflow code and stack segments.

Seg no	Use
0	Unused
1	Shared PERM
2	1st code seg
3	
4	
5	
6	1st stack seg
7	Task I/O buffers

If a program has external references in it, the Linker will first attempt to satisfy them using library LIB000 in the user's file system. If there are still unsatisfied references, the Linker will look at LIB000 in file system 0 (the system's file system). For more information on libraries, see the next section.

If the optional second output parameter is specified, a link map (see the example below) is output to it. In the above examples this would be in a file called C. .TT or .LP could also be specified.

Sample Linker Output Map

```
TASK: TEST
CODE: 040000  GLA: 140020
  XREF: PON      POFF      PONOFF      MAPVIRT
  XDEF: 040000   #GO
FILE: SHARED
  CODE: 046506   GLA: 140752
  XREF: 004000   RUN
  XDEF: 022656   PON
  XDEF: 022700   POFF
  XDEF: 022736   MAPVIRT
  XDEF: 023006   MAPABS

TOTALS: CODE = 006506  GLA/STACK = 002016
```

The TASK name of the program (in this case TEST) is printed, along with the base address for its code (040000) and its GLA (140020). XREF refers to 'external references' from that section of code (in this case they are all declared by the system). XDEF is an 'external definition', in this case #GO (the entry point for the main program).

FILE indicates that the Linker has loaded an object file; it specifies its name (SHARED) and the start address of its code (046506) and GLA (140752) sections. The list thereafter gives the 'external definitions', in this case entry points which the file contains.

The last line gives the overall code length of the program (006506) and the total size of the GLA and the declared stack.

If any references are left undeclared, the Linker will list them along with an 'UNDEFINED REFERENCE' message.

If the Linker attempts to load a file which contains an entry point that has already been loaded, the message '*DOUBLE DEF' is output and the Linker stops.

LIBRARY MANIPULATION

When the Linker finds an external reference in a program file it will attempt to satisfy the reference, by:

a) searching the library LIB000 in the user's file system, loading object files as necessary, then

b) searching the library LIB000 in the system's file system (0).

There are three programs currently available to the user for manipulating libraries:

1) NEWLIB - creates a new library file

2) INSERT - inserts the entries of an object file into a library

3) INDEX - lists the contents of a library

Note: Each of these programs will manipulate libraries other than LIB000, but at present the Linker will not link them.

---

1) NEWLIB

The command for using NEWLIB is as follows:

```
NEWLIB /LIB
```

This command will create a new library file, LIB; if it already exists the old copy is destroyed.

2) INSERT

There are two main forms of this command, as follows:

```
INSERT TESTY,LIB/LIB
```

and

```
INSERT TESTY,LIB/LIB2
```

The first form will add the entries of file TESTY into library LIB. The second form will add the contents of file TESTY into library LIB, creating a new library called LIB2.

3) INDEX

The form of this command is as follows:

```
INDEX LIB
```

This command will print out all the entries in the library LIB.

COMPILER LISTINGS WITH CODE ADDRESSES

A program called ALIST is available to generate an IMP listing which includes the addresses of the generated code. It can either be run using the output from the third pass of the compiler (file OY) or by using the final object program (usually more convenient).

Example

```
ALIST TESTS,TEST/L
```

This takes the source file TESTS and its associated object file TEST, and creates the listing in file L. It is of course possible to specify .TT or .LP instead of file L.

CHECKING THE COMPILED OBJECT CODE

A program called RECODE exists to take either the output from the third pass of the compiler or the final object program, and recode it back into assembler and merge it with the source program.

Example

```
RECODE TESTS,TEST/L
```

This takes the source file TESTS and its associated object file TEST, and creates the listing in file L. It is of course possible to specify .TT or .LP instead of file L.

TRANSFER PROGRAM

A program called T exists to allow source or object files to be copied from file to file, device to file, file to device, or device to device. It can also be used to concatenate two files by specifying a second input parameter.

Examples

T FRED/2.FRED(20) - copies the file FRED on disc drive 0 (user's current file system) to file system 20 on disc unit 2.

T FRED/.LP - lists file FRED on the line printer.

T .PR/FRED - transfers a file from the paper tape reader to file FRED.

T FRED,JIM(21)/FRED - appends to file FRED file JIM on file system 21.

T .TT/FRED - reads a 'file' from the terminal and creates a file FRED. To terminate the transfer, it is necessary to HOLD and then KILL T. (Note, however, that the preferred way to create a file from .TT input is to use the editor.)

FILE SYSTEM INTERROGATOR

A file system interrogator program called F exists to enable a user to examine and modify the contents of directories. It should be called without any stream definitions. It then prompts the user for input.

The possible replies and their meanings are as follows:

A	- Lists name of all files in the current directory along with the start block, protect code and number of blocks in the file.
B filename	- Gives data as in A for a specific file.
C	- Gets current L values (see below).
D filename	- Deletes file. Specifying '?' instead of a filename lists all files and for each file requests a 'Y' to delete, or an 'N' to keep, unless the answer to the automatic request is 'Y'.
F	- Lists filenames in current directory.
G filename	- Searches all directories for specified file. If found, reports location.
L x,yy	- Alters current directory to disc x, file system yy (octal). L CR causes return to current logon.
O	- Lists the files in the current directory, in alphabetical order.
R	- Renames file; prompts for old and new filenames.
S	- Stops.
T filename	- Transfers file; prompts for destination: disc and file system, .TT, or .LP.
U	- Lists all files for each user in turn.

Notes

* The program assumes that the current logon file system is on disc 0 on entry.
* A '?' in a file name means "do the command with respect to all files starting with the letters given before the ?".
* Unwanted output can be interrupted by sending A to task F using the INT mechanism:

```
INT A F
```

MAGNETIC TAPE UTILITIES

The system supports a TU16 controller and tape drive, which enables 800 bpi or 1600 bpi tapes to be read or written.

The following general purpose programs exist to use the TU16:

ARCHDK - archives a complete RK05 disc pack to 1600 bpi tape in 4K word blocks.

RESTDK - restores to disc a tape written by ARCHDK.

STAR - standard labelled tape archive and restore.

STAN - standard labelled tape analysis.

UTAN[R] - tape analysis for unlabelled tapes.

INSL - standard tape labeller.

Note: In systems where the tape device handler (TU16) is not loaded by the system initialisation program LOADUP it is necessary to

```
RUN TU16
```

before using a tape program.

A description of each program follows.

ARCHDK

This program archives a complete RK05 disc drive to 1600 bpi tape in 4K blocks. It prompts:

```
DRIVE?
```

when it is loaded. The user then types in the unit number of the drive to be archived, and the program will then write the entire contents of the disc to tape, sector by sector in 4K blocks. An entire RK05 disc takes approximately 45 seconds. The existing program writes the tape at 1600 bpi but it is a simple modification to change this to 800 bpi.

In systems where there is a double density pack and the system treats both 'units' as one unit, the entire disc - 5.4Mb - is written to tape.

The program is run by typing:

```
ARCHDK
```

in response to the system prompt.

RESTDK

This program takes a tape with a disc image written by ARCHDK and transfers the tape contents back onto a RK05 pack. When the program is loaded it prompts:

DRIVE?

and the user responds with the number of the drive that holds the disc pack to be overwritten. As a sensible precaution the user is advised to 'write protect' all other drives.

Before the data transfer begins the program asks for confirmation, thus:

CONFIRMATION?

If any reply other than 'Y' is given the program stops.

The remaining tape programs are described in detail in Appendix 1. They and their descriptions were written by C.D. McArthur (ERCC).

STAR - Standard labelled Tape Archive and Restore

This program is used to create archives from disc to tape. It creates a standard IBM labelled tape, and files are written to and read from it.

STAN - Standard labelled Tape ANalyses

This is an interactive tape analysis program driven by commands from a console or a file. Bytes of data on the tape can be identified and dumped on the printer.

UTAN and UTANR - Unlabelled Tape ANalysis

These programs are similar to STAN, but no assumptions are made about the format.

INSL - INitialise/re-initialise Standard Labelled tape

This program is used to create (or rewrite) a standard IBM label in a magnetic tape.

THE DEBUGGING PROGRAM

Object file name: DBUG

This program is used as an aid in debugging programs. It will normally be 'linked' to a running program on the system using the command T (for details see below); all accesses to locations will then be made in that program's virtual memory.

It may be used to do the following:

1) Set and clear breakpoints

2) Dump out the psect, registers and/or the IMP stack

3) Examine and change locations in main store

4) Dump general areas of main store

DBUG indicates its readiness to accept commands by issuing the prompt 'DEBUG:'. The following commands may be used:

T Set task number of program to be debugged

A Execute 'P' (see below) and dump the virtual memory

B Set breakpoint

C Clear breakpoint

D Dump an area of main store

I Dump the IMP stack

N Set a new program code base

O Change the output device

P Dump the psect

R Dump the registers

S Stop DEBUG

W Wait DEBUG

? Print options

In addition to these commands, there is an implied command, activated by typing an octal digit, which enters the location examination & change part of DBUG.

A detailed description of each command follows.

T - SET TASK NUMBER

The prompt 'TASK ID:' is output and the (octal) ID of the program to be debugged should be entered. The TASK ID of a running program may be obtained by typing the command TASKS to the Command Language Interpreter.

Note: Only the commands 'N', 'O', 'S' and 'W' may be used before 'T' is used for the first time.

A - PRINT ALL

Executes a Print Psect ('P'), and dumps the program's virtual memory. This command is useful for dumping all relevant information about a program onto the line printer; see also command 'O'.

B - SET BREAKPOINT

The prompt 'ADDR:' is output; the reply must be the relative address (with respect to the start of the program) of the desired breakpoint. Debug will remember the contents and place the text 'BR .' in the location. This will cause the program to loop when it executes that instruction.

Debug replies: BP: n ADDR: n2 CONT= n3

where n is the breakpoint number (between 0 and 20), n2 is the virtual address, and n3 the original contents.

The message 'BP TABLE FULL' is output if more than 21 breakpoints are used. See the command N for setting breakpoints in external routines.

C - CLEAR BREAKPOINTS

Prompts 'NO?'. The breakpoint number should be typed; 'A' or '-1' causes all the breakpoints to be cleared. If the specific breakpoint has not been set, the symbol '?' is output. The original contents are replaced.

D - DUMP AN AREA OF MAIN STORE

Prompts 'FROM:' and 'LENGTH:'. Both numbers should be typed in octal. The area of main store from FROM to FROM+LENGTH is output.

Notes:

1) In all the dumps 'ZEROES' is output if one or more entire lines contain zero.

2) An ISO equivalent is printed on the right hand side of the dump.

I - PRINT THE IMP STACK

The entire IMP stack is printed on the output device (see 'O' below).

Note: the GLA of the program is held at the low address end of the stack; the IMP stack (i.e. the SP stack) starts at the high address end and works towards the low address end.

N - SET A NEW PROGRAM CODE BASE

Prompts 'NEW PROGRAM CODE:'. Reply by giving the new address. This command is useful for programs using external routines. To set a breakpoint in an external routine, the code base is set to that of the external routine, as printed out by the Linker, and the relative address specified. This does not affect any previously set breakpoints.

O - CHANGE THE OUTPUT DEVICE

Prompts 'STREAM:'. The reply 'n' is used as in SELECT OUTPUT(n), for the output from commands R, P, I, A and D. By calling DBUG in the form:

```
DBUG /L     or
DBUG /.LP
```

the output may be directed to the file L or to the line printer.

P - PRINT THE PSECT

The PSECT (<u>own</u> system variables) of the nominated task is printed.

The command does the following :

1) Prints the name of the task.

2) Prints the state of the task.

3) Indicates whether there are messages queued.

4) Calls 'R' to dump the registers out.

5) Prints the contents of the segmentation registers, in the form:

<register no> <address> <length> <access>

R - PRINT REGISTERS

Prints the registers of the nominated task. The Local Name Base (LNB) for the outer level is also printed.

S - STOP DEBUG

Halts DBUG.

Note: all breakpoints are cleared.

W - WAIT DEBUG

Suspends DBUG if it is necessary to input to a program on the same terminal. It is restarted by

(Escape) FREE DBUG

Implied command to examine/change main store addresses

This command accepts the following instructions. 'n' and 'm' represent numbers input in octal.

n	prints contents of (n)
n+C	prints contents of (n+program base)
n+I	prints contents of (n+IMP stack base).
n+Rm	prints contents of (n+register m)
n(+options) = m	puts m into address specified by left hand side.

Examples:

DEBUG:100 (cr)	prints contents of location 100.
DEBUG:100+R5 (cr)	prints contents of 100 on from register 5.
DEBUG:100+R5=200 (cr)	plants 200 in location 100 on from register 5.

Notes

* A '*n' at the end of the command will cause the following 'n' locations to be dumped out ('n' may be negative).

* '+' or '+=m' may be entered as a new command. This takes the last location used and steps it up by 2 (or by -2 if '-' is used instead of '+').

SYSTEM FAULT MESSAGES

There are three main classes of fault message output by DEIMOS:

1) Compiler run time messages

2) Loader error messages

3) System detected error messages

1) Compiler run time messages

These messages are produced when the compiler run time support code detects an error. The program is stopped and the error number is returned to the task which started the program up. This task will normally be the Loader and the error message is produced in the form:

F A U L T n

where 'n' has the following values:

n	Meaning
1	Excess Blocks. N.B. only produced inside string handling routine.
2	Symbol in data (from READ routine).
3	String inside out.
4	Not used.
5	No switch label.
6	Capacity exceeded (strings only).
25	An input file does not exist.
26	Syntax fault on the Input/Output definitions string.
27	A block read or write to the disc failed (e.g. illegal block, aborted transfer).
28	The disc is full or the user's directory is full.

2) Loader error messages

These messages are produced when the Loader attempts to load a file and fails.

The format is:

fault n

where 'n' has the following meanings:

n	Meaning
1	The free main store is insufficient to load the program.
2	The format of the initial block of the program is too short.
3	Ditto - but it is too long.
4	The checksum on a block is wrong. Note: this can also be generated when the end of a file is reached prematurely.
5	Out of range. The program is attempting to load into a non-existent part of its VM.
6	End of file. EOF is reached before the final binary block is seen.
7	System full. The maximum number of tasks is already loaded.

In addition the following messages are also output:

Error Messages

no FRED	- An attempt was made to load the file FRED but it did not exist or was not binary.
TASK?	- A command has been issued to a task which does not exist.

Information messages

STOPPED	- A program, under the control of the Loader, has stopped normally. This message will only appear when there is more than one user task running in the System. When only one task is running and it stops, the system prompt is issued.
F A U L T n	- See the earlier section.
TERM REQ.	- A KILL has been given to a task and it has stopped.
XXXX PURGED	- A PURGE has been issued to a TASK and has been carried out.

3) System Detected Error Messages

These messages are issued when the system notices that a task has misbehaved in some way. The format in each case is the same:

<task name>:<error message>

The following messages are used:

BAD SEGMENT a b

The named task has caused a segmentation trap by writing or reading outside its virtual memory. The two octal numbers 'a' and 'b' are the value of the error register and the PC respectivelyat the time of the trap. This message is fatal; the task cannot run any more and should be investigated (using DEBUG), or PURGEd to remove it.

ADDRESS ERROR

This is the same as BAD SEGMENT, except that an address error was seen by the hardware (e.g. addressing a word on a non-word aligned boundary, or illegal stack pointer). It is also fatal to the task.

ILLEGAL INSTRUCTION

Same as above except the task attempted to execute an illegal instruction. It is also fatal to the task.

ILLEGAL SVC

The task has called a SVC that is not recognised by the system. It will normally be fatal.

BAD SER n

The task has sent a message to a non-existent process. This is not fatal: the task will continue to run but if it was expecting a reply it will never receive it. 'n' is the service number that failed.

TIME FAULT

The task has attempted to request the clock to kick it more than once. It is not fatal but indicates problems with the code.

NOT READY! n

This message is issued when a peripheral wants operator action. When this is done, a FREE or ABORT command should be sent to the task. The octal number 'n' is device dependent, but will normally contain the contents of the peripheral's error register (see the DEC PDP11 Peripheral Handbook).

APPENDIX 1

Tape Analysis, Archive & Restore, and Labelling Programs

Each of these programs is called in the usual way. A prompt is then given on the user's terminal; valid replies are described below. Once the actions requested have been carried out by the program, the prompt is repeated. This continues until the appropriate "stop" command is given by the user.

STAR - Standard labelled Tape Archive and Restore

A tape file is identified in its HDR1 label by its 6 character name, its 11/40 owner number - two octal digits, and its version number - two decimal digits. The latter is used to distinguish identically named and owned files on the same tape. (Foreign tapes should have both the owner and version fields recorded as blanks. On reading, these fields are interpreted as 00.) Also recorded in the label is the disc protect code of the file at the time it was written to tape.

Commands are of the form

COMMAND specification

The syntax used in the specification is as follows:

```
<unit>           := <unit number>. ! null
<unit number>    := 0 ! 1 ! 2 ! 3 ! 4
<fgroup>         := 1-6 alphanumeric, the first being alpha. Any can be
                    replaced by ?, implying any alphanumeric. Only the
                    first trailing ? need be given - thus A?=A?????
<version>        := :<version number> ! null
<version number>:= 2 decimal digits
<owner>          := (<owner number>) ! null
<owner number>   := 2 octal digits
<discdump>       := <discname> ! <discname> / <unit><userlist>
<discname>       := 1-6 alphanumeric
<userlist>       := (<users>) ! null
<users>          := <owner number><rest of users>
<rest of users>  := ,<owner number><rest of users> ! null
```

Unembedded spaces are ignored.

In the following command descriptions, OWNFS is the user's file system, specified via the system command LOGON (see above).

Note also that only the first letter of each command name is significant.

FILES

Lists the files present on the tape in the form:

name(owner number):version number

RESTORE <fgroup><owner><version>

If <owner> is null it defaults to OWNFS.
If <version> is null, any version is implied.

Restores file/s destructively to disc in OWNFS (regardless of <owner> field in tape file identifier). If <version> is unspecified, the earliest is restored first, thus leaving the latest on disc.

e.g. R ? restores the latest versions of all files belonging to ownfs to OWNFS.
R ?(21):00 restores the earliest versions of all files belonging to user 21 to OWNFS.

Note that foreign tape files to be restored must be specified as owner 00. Multiple restore specifications can be given on one line separated by commas; thus

R T?(10),FRED,?(00):00

TYPE <fgroup><owner><version>

As RESTORE, except that the file/s are transferred to the console instead of disc. The implications of a null <version> should be considered. Multiple specifications can be given.

ARCHIVE <unit><fgroup><owner>

If <unit> is null it defaults to 0.
If <owner> is null it defaults to OWNFS.

Archives file/s satisfying <fgroup> in directory <unit><owner>. Files are written to tape from the current end of tape onwards (but see DUMP command below). If no identically named and owned file/s already exist on the tape a version number of 00 is recorded, otherwise the latest version number+1 is recorded. Multiple specifications can be given.

DUMP <discdump>

If <unit> is null it defaults to 0.
If <userlist> is null all users are implied.

Overwrites existing tape contents, writing <discname> and the date into the tape volume label.

E.g. DUMP CDM (or DUMP CDM/) archives all files from the disc on unit 0, writing CDM and the date into the volume label

DUMP CDM/(20,21) archives all files belonging to owners 20 and 21 on unit 0

Once written to by the DUMP command, a tape cannot be written to by ARCHIVE. It can be rewritten by DUMP or re-initialised (by a separate program). It can be read by FILES, RESTORE and TYPE in the normal way and also by UNDUMP.

UNDUMP

This is used to restore all files from a tape written by DUMP. The files are restored to their original owners, as distinct from RESTORE which always restores to OWNFS. (All version numbers on a DUMP tape must be 00 since ARCHIVE cannot extend such a tape, and there should not be identically named files in one disc directory.)

The following checks are performed:

a) OWNFS=00
b) the tape has been produced by DUMP; ie the first character in the discname field in the volume label is non-blank (but beware foreign tapes).

These being satisfied, the following information is requested:

1) the name of the disc from which the tape was DUMPed,
2) the date on which the DUMP was carried out,
3) the unit to which the restore is to be made.

If this information is not given correctly the UNDUMP is abandoned. After 5 UNDUMP failures the tape is released and the program stops.

(If the tape is in fact a DUMP tape the discname and date of the DUMP is included in the claim tape message at the start of the session. It need only be copied exactly to satisfy requirements 1 and 2 above. This is not exactly a stringent test but at least gives pause for thought. It may be better to scramble the information in some way.)

BACKUP <unit><userlist>

If <unit> is null it defaults to 0.
If <userlist> is null all users are implied.

This archives all those files in the specified disc areas which are either not already on the tape or which have a different protect code (ie they have been altered since they were archived). Use of this command archives the minimum number of files to maintain an up to date archive tape. Like ARCHIVE, BACKUP adds files to the end of a tape and cannot write to a tape last written to by DUMP.

STOP

Rewinds and releases the tape and stops.

Limits:

Maximum files/tape : 200
Maximum read blocksize : 4000

The program reads or writes at 800 or 1600 bpi (determined by a successful read of the volume label). It will read fixed or variable format files, unblocked, blocked and/or spanned. However, it disregards the logical structure implied by the format, record length and block attribute fields in the HDR2 label, transferring each physical block to the destination as a stream of 8 bit bytes. The logical control information is therefore preserved. It writes fixed-length, unblocked, 512-byte blocks.

STAN - Standard labelled Tape ANalysis

This is an interactive tape analysis program, which is driven by commands either from the console or from a file (input stream 1). Each byte of data on the tape is defined by its byte, block, and file displacement from the start of the tape. Commands exist to move a notional cursor to any byte, and to output one or more bytes from the current cursor position, in octal, hexadecimal, character (optionally translated), or in a combined form of hexadecimal and character side by side.

Primitive commands consist of an operation, generally specifying a movement of the cursor or an output format, followed by a repetition count specifying, for a cursor-moving operation the number of bytes, blocks or files by which the cursor is to be moved or, for an output operation the number of bytes onward from the current position to be output.

The cursor-moving primitive commands are:

- Un - move the cursor to byte 1, block 1 of the nth file Up the tape from the current file. (Up is towards the end of the tape.)
- Dn - move the cursor to byte 1, block 1 of the nth file Down the tape from the current file.
- Fn - move the cursor to byte 1 of the nth block Forward from the current block. (Forward is towards the end of the file.)
- Bn - move the cursor to byte 1 of the nth block Backwards from the current block.
- Rn - move the cursor to the nth byte Right from the current byte. (Right is towards the end of the block.)
- Ln - move the cursor to the nth byte Left from the current byte.

The output primitives are:

- On - output in Octal, n bytes starting with the current byte.
- Hn - output in Hexadecimal, n bytes starting with the current byte.
- Pn - output as Printing characters, n bytes starting with the current byte.
- Cn - output in Combined hexadecimal and character side by side, n bytes starting with the current byte.

Clearly such primitives can fail; for example if the specified cursor destination does not exist, or if there are insufficient bytes remaining in the current block to output.

If the cursor-moving command R fails, the end of block failure message

```
**EOB** AFTER BYTE n
```

is produced. (If an output command fails, the output up to the failure is

produced.) If the cursor-moving command L fails, the start of block failure message is simply

SOB

since it must have failed at byte 1. Similarly, the F and B commands produce the file failure messages

EOF AFTER BLOCK n

and

SOF

respectively, and the U and D commands produce the tape failure messages

EOT AFTER FILE n

and

SOT

respectively.

The repetition count for a primitive command can be of two forms:

a) explicit - an integer>0, meaning do it that many times (an explicit repetition count of 1 need not be typed)

b) indefinite - 0 or ?, meaning do it until EOT, SOT, EOF, etc., and produce the appropriate failure message.

Two types of output have already been mentioned - that produced by the O, H, P and C commands, and failure messages. A third type of output is produced when an explicit U, D, F or B command succeeds.

Whenever an explicit U or D command succeeds, the file descriptor for the newly current file is output, in the form

FILE n

name [format lrecl/blocksize block-attribute]

(See the section "File descriptor formats", below.) Whenever an explicit F or B command succeeds, the block descriptor for the newly current block is output, in the form

BLOCK n LENGTH=m BYTES

Multiple primitive commands can be input on the same line, forming a compound command whose elements are executed from left to right. Additionally, parentheses may be used to define compound commands which, exactly like primitive commands, are followed by an explicit or indefinite repetition count. (An unparenthesised compound command of one or more primitive commands behaves exactly as if it had outer parentheses and a repetition count of 1, and should be so considered in the description of termination and failure, below.)

An explicit primitive command terminates when its repetition count is exhausted. It terminates and fails at EOT, SOT, EOF, etc. An indefinite primitive command terminates only at EOT, SOT, EOF, etc. It never fails (although it produces a failure message).

An explicit compound command terminates when its repetition count is exhausted (i.e. all its elements have been executed from left to right without failure n times). It terminates and fails when any element of it fails. An indefinite compound command terminates only when an element of it fails; the compound command itself never fails. In view of this, care should be taken to ensure that an indefinite compound command contains a subsequence whose repetition will ultimately give a failure, otherwise the program will loop indefinitely. In this context, note that the output commands do not move the cursor.

A reverse slash immediately following a command (primitive or compound) cancels any failure of that command.

Examples

Some examples follow to clarify these points. At the start of a session the cursor is positioned at byte 1, block 1, file 1 and the file descriptor and first block descriptor are output. From this position the following commands generate the output described.

command	output
U?	EOT failure message only.
(U1)?	file descriptors for files 2 onwards to EOT.
F?	EOF failure message for file 1.
(F?U)?	EOF for file 1, then file descriptors and EOF for all files to EOT.
((F1)?U)?	block descriptors for blocks 2 onward to EOF for file 1, then file and block descriptors for all blocks in all files from 2 onward to EOT.
(H50F1)?	if all blocks in file 1>=50 bytes, then hexadecimal of first 50 bytes in block 1 and block descriptors and hexadecimal 50 for all blocks from 2 onward to EOF, else ditto up to first block with <50, when EOB for that block and terminate.
(H50\F1)?	hexadecimal of 50 or "block length" bytes in block 1, whichever is the less, and block descriptors; ditto for all blocks from 2 onwards to EOF.
(F1B1)?	loops indefinitely without output.
(P10)?	the first 10 characters of block 1 indefinitely.
(F?)?	EOF for file 1 indefinitely.

Now to the question of the cursor position after failure. With three exceptions there is always a defined current file, block and byte. As mentioned above, at the start of a session the cursor is positioned at file 1, block 1, byte 1 unless

a) There are no files on the tape, when the EOT condition is immediately raised; this condition is also raised when a U command is executed and the current file is the last on the tape. Thereafter, the block and byte positions are undefined, commands F, B, L, R and the output commands give an appropriate failure message and terminate and U gives the EOT failure message and terminates. Only D of the commands described so far is effective.

or b) The first file contains no data blocks when the empty file condition is raised; this condition is also raised when a U or D command is executed and the newly current file is empty. The file descriptor and the empty file failure message is output. Thereafter, the commands described so far except U and D produce an appropriate failure message and terminate.

or c) the first block of the first file is a bad block (i.e. one which cannot be successfully read from the tape because of parity errors), when the bad block condition is raised; this condition is also raised when a U, D, F or B command is executed and the newly current block is bad. A warning message is output in the form

```
**BAD BLOCK**   m TRANSFERRED
```

where 0<=m<=block length and is the actual number of bytes transferred to main store. If m=0, the block descriptor takes the form

```
BLOCK n   **BAD BLOCK (ZERO LENGTH)**
```

and the current byte becomes undefined. Thereafter the output and L and R commands give an appropriate failure message and terminate. (Note that this condition is only raised for bad data blocks. A bad block occurring as a label is treated as a fatal error; the tape is then rewound and the program stops.)

Apart from these three cases, the cursor position after failure is as follows:

command	file	block	byte
D	1	1	1
F	current	last	1
B	current	1	1
R	current	current	last
L	current	current	1
output	unchanged	unchanged	unchanged

Output from the execution of an O, H, P or C command begins on a new line. Within one such command, newlines are inserted if required on the basis of the page width, which by default is 132. This width can be reset at any time using the W command:

Wn - set the page width to n; n<12 or n=? produces a failure message and fails.

The output produced by the O and H commands is always an exact representation of the bit pattern on the tape. The character output produced by P and as part of C is a translated form of the bit pattern. The translation is controlled by the current translation mode. If the mode=0 (the default) the tape bytes are assumed to be ISO characters and the bottom 7 bits of each byte are used, non-printing characters being translated to space and newline to ^. If the mode=1 the tape bytes are assumed to be EBCDIC characters and are translated to their ISO equivalents, non-printing to space and newline to ^. The translation mode can be changed at any time using the M command:

Mn - set the translation mode=n; n>1 or n=? produces a failure message and fails.

After some interactive movements within the files on a tape, it may be convenient to have the current cursor position output explicitly. This is done using the T(Tell) command:

Tn - output the current cursor position (current file descriptor, block descriptor and byte number); the repetition count is discarded.

The analysis is stopped and the tape released using the S command:

Sn - end the analysis, rewind and release the tape and stop.

Conclusion

The program produces its output on stream 1, which can be a file or any defined output device. As already mentioned, it takes its input from stream 1, which can be defined to be the console or a file. Thus an arbitrary number of tailor-made analyses can be obtained simply by inserting the appropriate command string in a file (preferably pre-tested for indefinite looping, especially if the output is not to the console). As a final example, one likely candidate for inclusion in such a set of analyses is the command string

```
M1((C?F1)?U1)?S
```

which produces - the file descriptor, the block descriptor, a side by side hex/character dump of all the bytes in the block, for all the blocks, for all the files in an EBCDIC coded tape, the output being suitable for a page width of 132.

A version of the program taking identical commands for analysing unlabelled tapes is available. This cannot cope with the empty file condition, since, lacking the context of the header and trailer labels, it must assume an empty file to indicate EOT, but otherwise its behaviour is identical. Since it treats all blocks as data blocks it can be used to analyse a labelled tape, the label blocks being examined as data. In particular, it can be used to analyse a tape containing bad label blocks.

Further versions for different labelling schemes can be produced relatively simply, and further character translation modes can be included given the appropriate translation tables.

Summary of Primitive Commands

Un - move the cursor up n files.

Dn - move the cursor down n files.

Fn - move the cursor forward n blocks.

Bn - move the cursor backward n blocks.

Rn - move the cursor right n bytes.

Ln - move the cursor left n bytes.

On - output n bytes in octal.

Hn - ouput n bytes in hexadecimal.

Pn - output n bytes as printing characters.

Cn - ouput n bytes in combined hex/character side by side.

Wn - set the page width to n.

Mn - set the translation mode to n.

Tn - tell the current cursor position.

Sn - rewind, release the tape and stop.

File Descriptor Format

```
FILE n
name    [ format lrecl/blocksize block-attribute]
```

where:

name	- 17 character file name padded on right with blanks
format	- record format; F = fixed, V = variable, U = undefined (as read from HDR2 label)
lrecl	- logical record length (as read from HDR2 label)
blocksize	- physical blocksize; maximum if V or U format (as read from HDR2 label)
block attribute	- blank = no blocked or spanned records, B = blocked records, S = spanned records, R = blocked and spanned records (as read from HDR2 label)

UTAN is the tape analysis program for unlabelled tapes mentioned in the description of STAN (above). This program takes identical commands and - with the exceptions noted below - performs identically with respect to output, termination and failure.

The exceptions are

a) the file descriptor reduces to

FILE n

b) a tape which begins with a tape mark is considered to have an empty file in position 1; the empty file condition can arise nowhere else, since two adjacent tape marks signal EOT. As a consequence of this, an empty tape (i.e. one which begins with two tape marks) is considered to contain one empty file.

Typically this program should be used to analyse a tape which is either known to be unlabelled or whose labelling scheme is unknown. If labels are present, they can be examined as data; if a labelling scheme is then recognised for which a specific tape analysis program exists, that program should be used to provide more convenient access to the genuine data and to give access to data beyond an empty file, which this program cannot give.

Both UTAN and STAN attempt to establish the density of a tape by reading the first two blocks at 800 bpi, and if they both give parity errors, trying at 1600 bpi.

If a tape has genuine parity errors in the first two blocks, this will result in UTAN giving up altogether since it will fail to establish the density at all. In order to analyse such a tape, UTANR is available. It is identical to UTAN except that it first requests the density from the console instead of trying to establish it on its own. A tape starting with two bad blocks can thus be analysed.

INSL - INitialise/re-initialise Standard Labelled tapes

INSL is used to re-initialise standard labelled tapes. This is achieved simply by writing a new VOL1 label and two tape marks at the start of the tape.

The tape driver TU16Y must be running to use INSL, which first requests the new name for the tape. This can be up to 6 alphanumeric characters the first being alpha. (More than 6 may be typed but only the first 6 are recorded on the tape.) A null name or one containing illegal characters is discarded, with an appropriate message, and the request repeated.

The recording density is then requested. The reply should be 0 for 800 bpi or 1 for 1600 bpi. Any other reply is discarded and the request repeated.

The name and density specified are then output to the console for verification, together with a reminder that the existing tape contents will be lost. A final confirmation that the re-initialisation should be carried out is requested. A reply of Y (yes) causes the new label to be written. If any other reply is given the tape is released without alteration and the program stops.

APPENDIX 2

Building a New System

A new system is built by running a program (SBLD) that links together a number of files containing the basic building blocks of the supervisor. The program then allows the interrupt vectors to be tailored for the specific machine before writing the entire supervisor to one of a number of fixed IPL sites on a disc.

SBLD is run in the form: SBLD3Y SUPDE0,D/.TT

The '3' denotes the current version of the program and will vary from time to time.

SUPDE0 is a steering file, described below.

D is a dummy file name, to allow SBLD to do further I/O.

.TT is a report stream; any normal output definition may be used here.

The steering file SUPDE0 contains all the instructions for building the supervisor. An example file is given below:

```
                                                                    Line
DEIMOS VSN 8.2 22.FEB.80 (NEW BUILD PROG, SUPER AT 2600)             1
2600                                                                 2
BRUN0Y N 0                                                           3
SUP28 60002 0                                                        4
PERM9Y 60006 0                                                       5
BTT4Y 60012 150                                                      6
BDK5Y 60016 70                                                       7
FSYS1Y 60022 300                                                     8
LOAD6Y 60026 300                                                     9
MOTH6Y 60032 50                                                     10
END                                                                 11
-6 304
-7 310 DQS11 # 1
-20 324 DQS11 #2
-21 330
-22 404 DQS11 #3
-23 410
-24 424 DQS11 #4
-25 430
-26 454 PARR INT #1
-27 444 PARR INT #1 (TX)
-14 350  KMC
-13 354 DITTO
0
S                                                                   25
0                                                                   26
```

The 1st line is a ~~command~~ comment which identifies the particular supervisor.

The 2nd line (2600) contains the address at which the supervisor Kernel (SUP28) was linked (see separate instructions). The actual address is dependent on the register load/unload module. This will normally be a

constant which depends on the number and type of 'pseudo' DMA devices built into the assembler register load/unload module.

The 3rd line (BRUN0Y N 0) is the name of the register load/unload module. The N signifies that there are no linkage pointers dumped for initialisation purposes. The 0 is the size of the stack - in this case non-existent as it is an assembler module.

There are a number of variants of this particular module, the main ones being as follows:

BRUM0Y basic module. The supervisor must be linked at 2600.

DVRUBY basic module + a 'pseudo' DMA handler for a SUP11 working in byte mode with BSC framing. The supervisor must be linked at 3600.

DL11Y basic module + a 'pseudo' DMA handler for a DL11 acting as a processor/processor link. The supervisor must be linked at 3600.

XBM01Y basic module + a 'pseudo' DMA handler for a DUP11/DU11 emulating ICL's "ICL02" protocol.

The 4th line (SUP28 60002 0) contains the name of the Kernel module. The 60002 is the address at which its start address and (at 60004) its GLA address are dumped to allow proper initialisation of the supervisor; all these 'dedloc' addresses are fixed and should not be modified without careful consideration of the effects within the Kernel initialisation module SIN008. (No further reference will be made to these numbers.) A stack size of 0 is again given, as its stack is always placed immediately beneath the load/unload module.

The 5th line (PERM9Y 60006 0) is the name of the cement module (implemented in assembler); this contains the shared I/O function and system routines used by all IMP programs (source: PERMAS).

The 6th line (BTT4Y 60012 150) is the name of the main console handler (source: BTT4S). It should be noted that the variables containing the major parameters (hardware address, interrupt numbers, etc.) are owns, to allow the code to be stored with subsequent console handlers.

The 7th line (BDK5Y 60016 70) is the name of the disc handler for the main system unit (0).

There are a number of variants of this handler, depending on the type of disc in use:

BDK5Y (source: BDK5S) RK05 handler (units 0 and 1).

RL03Y (source: RL03S) RL01 handler (units 0 and 2). N.B. hardware unit 1 is mapped to software unit 2.

BDKH9Y (source: BDKH9S) Amplex handler (units 0 and 3). Floppy disc handler (still to be written).

RX021Y (source: RX021S) Floppy disc (RX02) handler.

PRDL3Y (source: PRDL3S) For systems with no disc and which load programs down a DL11.

The 8th line (FSYS1Y 60022 300) is the name of the file system handler. This handler normally organises the files on all of the disc units available on a system. It should be noted that a slightly different version is required for each of the different disc handlers listed above, because of the differing disc sizes. One further minor variation in file system handlers is the number of buffers allocated for the handlers' use; this parameter is completely specified before the module is compiled. For systems with no disc, a dummy program (FSDUMY) is specified.

The 9th line (LOAD6Y 60026 300) is the name of the loader/command language intepreter.

The 10th line (MOTH6Y 60032 50) is the name of the error handler task (usually called MOTH).

The 11th line (END) terminates this phase of the building and switches to the interrupt number setting phase. The input for this phase is in the form:

```
-6 304 command
```

The '-6' is the interrupt service number, i.e. the number on which a device handler does a 'LINKIN' to claim the device.

The '304' is the address to which the device will interrupt; the program then plants the necessary linkage.

Only the interrupt numbers less than -5 may be set in this way. The others, i.e.:

```
-1 main console O/P
-2 main console I/P
-3 unit 0 disc interrupt
-4 address error interrupt
-5 TU16 interrupt
```

must be changed by editing and re-assembling the load/unload module. In addition, the 'pseudo' DMA device interrupts must be set in this way.

This phase of the building is terminated by a 0.

The 25th line (S) is the termination of a patch phase that is now rarely used. The form of any patch is:

```
1000=300
```

which sets octal '300' into octal address '1000'.

The final (26th) line is a character which defines to which site the completed supervisor is to be written; the normal variants are:

```
0 - site 0 (blocks 1-76) of unit 0.
1 - site 0 (blocks 1-76) of unit 1.
T - site 1 (blocks 4600-4676 of unit 0) (N.B. RK05 only)
R - site 0 (blocks 1-70) of unit 2
```

When it has completed, SBLD prints out

```
NOW IPL
```

and stops.

A sample output, to match the input file above, is given below:

```
⊼SBLD2Y SUPDE1,D/.TT
DEIMOS VSN 8.2 22.FEB.80 (NEW BUILD PROG, SUPER AT 2600)
SUPER CODE BASE?002600
FILE:BRUN0Y   000000   000000   002253   000000
FILE:SUP28    002600   007700   014151   060002
FILE:PERM9Y   014200   000000   017377   060006
FILE:BTT4Y    017400   022500   023065   060012
FILE:BDK5Y    023300   024500   024613   060016
FILE:FSYS1Y   025000   031000   035261   060022
FILE:LOAD6Y   035600   045500   046241   060026
FILE:MOTH6Y   046600   047600   050115   060032
FILE:


RESETTING OF INTERRUPT NUMBERS AND VECTORS

INT: -6 VECTOR:000304(001660)
INT: -7 VECTOR:000310(001670)    DQS11 # 1
INT:-20 VECTOR:000324(002044)    DQS11 #2
INT:-21 BECTOR:000330(002054)
INT:-22 VECTOR:000404(002064)    DQS11 #3
INT:-23 VECTOR:000410(002074)
INT:-24 VECTOR:000424(002104)    DQS11 #4
INT:-25 VECTOR:000430(002114)
INT:-26 VECTOR:000454(002124)    PARR INT #1
INT:-27 VECTOR:000444(002134)    PARR INT #1 (TX)
INT:-14 VECTOR:000350(001764)     KMC
INT:-13 VECTOR:000354(001754)    DITTO
INT:PATCH?
SUPERVISOR LOADS FROM 000000 TO 050115 AND 60000 TO 063707
TOP OF STORE IS DETERMINED AT RUN TIME
DISC?
PUT ON UNIT 0 ON SITE # 2
CORE IMAGE WRITTEN
NOW IPL
⊻
```

Notes

1) The numbers following the file name are as follows:

   START OF CODE - START OF GLA/STACK - LAST BYTE USED - DEDLOC ADDRESS

2) The number in brackets following the vector address in the interrupt section is the address at which the linkage code is placed.