IMP as a Tool for Small Operating Systems Implementation

Brian A. C. Gilmore

Thesis Presented for the Degree of Master of Philosophy

Faculty of Science, University of Edinburgh

October 1976

ABSTRACT

This thesis discusses the desirability of using a high level language for small systems implementation. An examination is made of some of the features required in such a language. A detailed comparison is drawn between a multi-programming system for a PDP 11 written in assembler and a similar system written in the high level language IMP. The factors being compared include the costs of writing and maintaining the two systems and the differences in their physical characteristics, namely the store used and time taken. It is shown that the use of IMP is advantageous in the writing and maintaining of the system written in IMP but that this advantage is offset to a certain extent by extra store requirements and a slower response time.

DECLARATION

I declare that this thesis was composed by myself and that the work described was done by myself with the exception of the following parts:

One of the application programs and

part of the IBM 2780 emulator.

Brian A. C. Gilmore

CONTENTS

CHAPTER 1

INTRODUCTION

STATEMENT OF THE PROBLEM

Over the past years the use and variety of high level languages has grown considerably, especially for applications programs and to a more limited extent for compilers. However, at this time, very few systems have been written in a high level language and most of these have been designed to run in a large system environment. This poses a question as to whether a high level language is suitable for implementing a system on a small computer. This also involves considering what features are desirable in such a language.

MYTHOLOGY AND BELIEFS

The benefits of using a high level language are widely claimed (2,3,5). However very few actual comparative figures have been given. The claims usually advanced in favour of using a high level language are:

1) Initial development.

   The costs of writing and commissioning a piece of software in a high level language are lower than writing it in assembler. Brooks (4) has examined two large projects, one written in assembler and the second in a high level language and came to the conclusion that programmer productivity, in terms of the number of source statements written in a given period, is constant

irrespective of the language used. The implication is that the use of a high level language gives an increase of five to six times in the quantity of machine code produced in a given time.

2) Maintainability.

The maintenance of a large system is extremely expensive and it is claimed that using a high level language to write the system reduces these costs (6). One factor involved in this is the turnover in staff that can be expected in any large project. A new programmer has to familiarise himself with the project. Corbato claims that this is achieved more quickly when a high level language is used (5). The author's experiences in taking over programs in both assembler and in a high level language have confirmed this. One problem associated with a program written in assembler is its sheer bulk. Usually the size of the assembler program will be at least three times the size of the equivalent high level language program, and Corbato in fact states a factor of ten times for PL/1 (5).

A second problem is the lack of structure inherent in a program in assembler, as each programmer tends to structure the program in his own way whereas in a high level language much of the structure is dictated by the language. For example, in assembler there are innumerable ways to pass parameters to a routine - in registers, on a stack, in a particular area, in line after the call, or combinations of these. A high level language will, in general, only have one method of passing parameters,

with the probable additional ability to use global variables or the equivalent of a FORTRAN COMMON.

Writing in a high level language removes part of the drudgery associated with writing in assembler. A simple example is the 'FOR' loop. One statement in the high level language will initialise the loop variable, adjust it on each iteration and check for the loop termination. The same loop in assembler on the PDP 11 will be at least five instructions, depending on the complexity of the loop bounds. The existence of such extra statements increases the chance of errors occurring.

3) Portability.

After sinking a large capital expenditure, in the form of programmers' time, into an assembler program, the advent of a new machine will make the bulk of the code useless. In a high level language, once a compiler exists - a one-off job - the moving over of the code should only involve a reconsideration of different word lengths. Even that should be very small or non-existent in a carefully written program. This was clearly shown by R. B John (18) in the implementation of communications software on several different machines.

The training of a programmer is costly and if this training is in assembler it will be wasted if new hardware is introduced. Much of the skill in writing efficient assembler is intimately connected with the tricks and 'clever instructions' implemented

in that hardware, features which are likely to change with different hardware.

These considerations have meant in practice that high level languages are widely used; indeed it is claimed by Brooks that only 'inertia and sloth' prevent their universal use (4).

Despite this the acceptance of the high level language as a systems programming tool for compilers themselves is by no means complete. On large machines most compilers still seem to be written in assembler. The Edinburgh Multi Access System (EMAS) (12,13) is one exception, with all its compilers, ALGOL, FORTRAN and IMP (11, 22) being written in IMP. Some languages are not very suitable for self implementation, COBOL being an example, but the alternative, that of writing the compiler in another high level language seems to be overlooked or rejected.

On small machines, the situation is very similar. On the whole all compilers, especially the manufacturer's own, are in assembler, the exception being the CORAL66 compilers, which have tended to be implemented in CORAL, for example by Ferranti and Interdata.

When it comes to systems themselves it is not normally accepted that a high level language should be used.

The first large systems in a high level language were the Burroughs MCP (6) in 1961, followed by MULTICS (1) in 1964, and EMAS

in 1966. More recently we have seen the ICL VME/B system (20) on the 2900 series written in S3, a high level language.

On mini computers, the use of high level languages has been even slower, the first well known system being UNIX (17) written in 'C', though this is more for a large scale PDP 11. The Argus 700 (8) series system is also written in Coral (1973). Recently, Purser (3) has published a paper describing a real-time system for a PDP11 also written in CORAL.

At the level of systems implementation it is argued that the disadvantages of using a high level language outweigh the advantages. The most commonly cited disadvantages (2) are

1) Extra store.

A program written in a high level language will normally be larger than the counterpart in assembler. A compiler can usually be made to optimise in one of two directions, it either optimises for the shortest code or for the fastest execution. For systems work the code optimiser will probably be too slow, and a compromise is looked for. Such a compiler is unlikely to spot global optimisations that a programmer can. However, in a large project, this trend is counteracted to a certain extent by the decrease in complexity.

2) Slower execution.

The effect of extra code will normally be a slower execution

time.

For systems implementation in particular, there is a third solution, that of a high-low level language - for example Babbage (23) on the GEC 4080, the PL360 group of languages (24) and HAL (25) developed by the Computer Science department of Edinburgh University. This type of language tends to make assembler programming easier, providing some of the constructs usually found in a high level language, but not all the structure that a high level language provides. In certain circumstances such languages can be beneficial but not necessarily on all types of architecture.

THE EXPERIMENT

The author decided to write a system in IMP, a high level language, and compare that system to a similar system already written in assembler.

The author had implemented a multi-programming system in assembler, for a small configuration DEC PDP 11/20. At that point in time (1971), the only other disc-based system available for the PDP 11 was the DEC 'Disc Operating System' (DOS), a single program system. After a period of time the author felt that the effort of making further additions, for example new device handlers, was disproportionate to the complexity of the device. New hardware had become available, in the form of a PDP 11/40 with memory management and a system was required for it. Having worked for a period of time on EMAS, and particularly on the Front End Processor for its

communications, all in IMP, the author considered that it should be possible to implement a system for the PDP 11/40 in IMP.

The designs of the two systems have a common ancestry but do use some different concepts. The differences, and especially those due to the target machine, mean that it is not possible to make a line-by-line comparison of the two systems. It is however, possible to separate out some of the effects of the differences and make useful comparisons of the respective sizes of the systems. Conclusions may easily be drawn on the implementation aspects of the two systems, in particular on the initial implementation costs, the differences in the ease of adding new components and the general maintainability of both systems. It is more difficult to make comparisons of the system overheads in respect to the time spent in the supervisor. Both the differences in the hardware and the consequently differing designs mean that it is very difficult to draw overall timing comparisons, but similar subsections can usefully be compared.

BRIEF HISTORY

Work on the first system was started in October 1971 on a part time basis. The major part of the kernel and some other parts of the system were written by October 1972. By January 1973, a basic system was able to run programs written in a restricted form of IMP. After this period very little was done to the system until August 1973 when work on the benchmarking of EMAS was started. This provided the first real test of the system and over the next few months the system

became more robust. In March 1974 the system was put into use on the PDP 11/20 in the Faculty of Social Science of Edinburgh University. The machine had 24k words of core, three terminals--one a graphics device, a disc, card reader, line printer and a graph plotter. A link to the local IBM 360, to emulate an IBM 2780 RJE terminal, was established later that year and the system was then used on a full time basis. In the following year two other systems were installed on PDP 11/20s.

A new implementation of IMP (9) became available in March 1976; the second system was started in the middle of that month and the first version of the kernel was being tested at the end of the month. By the middle of April the system was running IMP programs, albeit with rather rudimentary I/O. A loader, together with a partially implemented file system, was operational by the end of April. The system was fully self-supporting by the 11th May. Since then, up to the end of June, at which stage development paused, several device handlers, e. g. to handle magnetic tape, a line printer and a synchronous communication line, had been added, the system made more robust, various extensions made to the system and a number of faults put right.

CHAPTER 2

DESCRIPTION OF SYSTEM ONE

GOALS

This system was designed to run in a small PDP11. A minimal system requires at least 8K words of core and one input/output terminal; a clock is desirable.

The system was designed with four main aims:

Running user programs

The system is designed to run general user programs. The system will support a large number of programs, with each program having full access to all available system resources. Each user program runs in its own environment; it may use all the machine store not used by the system, though normally resources will be shared with other programs. A program does not have to be in any special form; for example, an IMP program looks like an IMP program as run on other machines. A special header is required to inform the system of the program characteristics such as its store requirements, priority etc. This header is set up automatically for IMP programs by the compiler but a task-building program needs to be used for assembler programs. A paper tape reader is used to load programs on a minimal system, a disc can be used on larger systems.

Multiple terminal support

The system should support more than one terminal, where each terminal has equal access to all the system resources through a command language interpreter. One console is nominated as a master console for outputting system error messages.

Peripheral support

The system should support a wide range of peripherals, eg, line printer, card reader, paper tape reader and punch, several types of discs, dectape and communication lines.

Fast system response

The system should be able to respond to events with minimum delay, all uninterruptable paths within the kernel should be short, with a guaranteed initial interrupt response of 400 micro seconds. A priority structure may be used to ensure the necessary response time.

CONSTRAINTS

The most serious constraint in the system is that of store. This constraint was imposed by the first aim of the system; in order to allow as much space as possible for user programs the aim was to restrict the kernel to around 4k words of store. This constraint has affected the user interface, which is very basic.

STRUCTURAL OVERVIEW

The primitive structure of the system was derived from a paper by D. Mills (15,16).

The system consists of a minimal kernel (of 800 words of code and a 750 word data area), which supports virtual machines. Each virtual machine runs a program; the program and its environment are collectively known as a TASK. A number of these tasks, known as system tasks, are used to handle the peripherals and other system functions, e. g., a loader and a garbage collector. When a program is run, the system creates a new task which is destroyed when the program terminates.

The kernel is essentially a collection of routines which are invoked by tasks and run on a private kernel stack. Their functions are to:

1) Control CPU allocation.
2) Handle interrupts.
3) Handle task faults.
4) Implement semaphores (claim and release operations).
5) Deal with clock and timer functions.
6) Allocate and deallocate small I/O buffers.

The kernel was designed so that all uninterruptable paths through it are short; for example, one of the longest, the claim semaphore operation (when the semaphore is held) takes an additional 70 micro seconds on top of the basic context switch time (about 330 micro

seconds). A system function that takes longer than this is dealt with in one of three ways depending on the time it takes and its nature.

1) Within the kernel but at least partially interruptable. The timer queue manipulation is an example of this; it is interruptable by an interrupt at the highest priority level. The path length is about 100 micro seconds, rising by 20 micro seconds for each entry on the queue. By allowing it to be interrupted at the highest priority, it follows that tasks running at that priority cannot perform timer operations.

2) By a system task. The garbage collector is a system task; it needs to run uninterruptably for six instructions each time it needs access to the garbage queue but apart from that runs at a normal level for the rest of the time.

3) By running in the user task but protected, either by running at a particular priority level, or by a semaphore. The file storage handler runs in this manner; the advantage is a saving on context switching overheads; the disadvantage is the extra space required by each user task to accomodate it (about 24 bytes). A second example is the insertion and deletion of characters from small I/O buffers; these operations need to be protected from one another; this is achieved by protecting them with a high priority level; the path length is on average 30 micro seconds within the routines.

The distinction between a function performed interruptably within the kernel and performed within a task environment is important. When the kernel is interrupted, it allows the interrupt to be processed, but it will complete its processing before allowing another task to run. When a function being performed in a task environment is interrupted, it will not continue processing until all higher priority tasks have been satisfied.

All the timings given are based on a PDP 11/40. The instruction time for a basic register move is 0.90 micro seconds on the 11/40. The 11/10 figure is 3.1 micro seconds. These and other figures (14) indicate that the timings for an 11/10 are about double the 11/40 timings.

A task consists of two parts, a task environment and a program. The task environment maintains a virtual machine for the running of the program. The task can be in one of five basic states:

WAIT - the task is awaiting an interrupt.

ACTIVE - the task is running.

BLOCKED - the task is held on a semaphore.

PENDING - the task is ready to run and is on a CPU queue

SUSPENDED - the task is prevented from continuing (eg after an error)

The program is limited in the state transitions it can execute to those from the ACTIVE state to one of the others, for example, to the BLOCKED state by attempting to claim a busy semaphore, or the WAIT state by issuing a supervisor call. The remainder of the state changes are carried out by the kernel directly, eg, PENDING to ACTIVE, or BLOCKED to PENDING when a semaphore becomes free, or by the task environment; for example, the normal response to an interrupt will be to change the task state from WAIT to PENDING. The code that interfaces the task to the system at this level is called the 'task monitor' and it runs, interruptable from a higher level, in supervisor mode on the private kernel stack. There are three kinds of interrupts which the 'task monitor' processes:

1) Hardware-generated interrupts from devices linked to that task.
2) Kernel-generated software interrupts when the task fails.
3) Software-generated interrupts from other tasks.

The 'task failure' interrupts are handled at this level to allow different actions, depending on the type of task, to be taken; for example, if a user program task fails, diagnostic information on the failure is held on the stack. A system task does not have the stack space to hold this information.

Interrupts are vectored directly to the task controlling the device. To process an interrupt involves a context switch giving an absolute upper limit of 2,700 interrupts a second. When an interrupt is particularly time critical, or interrupts occur at a rate

approaching or exceeding the limit - thus using nearly all the available CPU - extra code is used at the interrupt level to construct a new interface between the task and the device. A card reader is handled in this way; on interrupt, the data is stored directly into a buffer, and the task is only interrupted on a 'card done' interrupt. The system does not queue interrupts, for space saving reasons. It is normally the practice for one task to control a single device. To handle multiple devices, the task would need to queue the interrupts internally, which would seem rather wasteful considering that the system is built to multiprogram a large number of tasks, and the marginal cost of a second task, sharing its code with the first, is small, about 70 bytes.

The two basic communication and synchronisation mechanisms are the semaphore (21) and a 'software interrupt'.

Each peripheral has a system task to control it. I/O from a user task is passed to the device handler through one of two interfaces depending on whether the peripheral involved is a block transfer device or a single character device.

The block transfer device - disc, dectape - is buffered a block (256 words) at a time and each block is handled individually; processing (of the task) stops until the transfer is completed.

A single character device - terminal, line printer, paper tape devices etc - is handled by buffering the characters into a chain of small buffers (NIBBLES) which are linked together. There is a BUFFER CONTROL BLOCK that contains the necessary pointers into the chain.

The kernel holds a global free list of these buffers. The user task is able to (at least) double buffer its requests before being suspended on the handler's semaphore. The use of small buffers is more expensive in code and certainly more expensive in execution time but in a minimal configuration machine represents a considerable saving in buffer space as there is no need to hold pools of maximum length buffers.

IMPLEMENTATION

The system was written in PDP11 assembler. At the time that the system was started there was no suitable high level language available and the aim was to write a system to run programs rather than write a compiler. The system is cross-assembled and is not self supporting.

Most applications programs are written in a high level language (IMP). As the machines that the system runs on either do not have enough core to run a compiler or do not have enough disc space, The sources are held on EMAS and compiled there. The binary may be transferred to the target machine in a number of ways, for example, paper tape, cards, or via a communications line if one is available. Recently, a smaller compiler has become available and this will allow some of the systems to compile and run on site.

OPERATION

The system has now been in operation at three sites for between one and two years. The three configurations are 16k, 24k and 28k words of core with a wide variety of peripherals. The system has also been used to interactively benchmark the EMAS system, simulating 32 terminals, running on a PDP 11/45 (10). Other systems, one with only 8k words of core, have been run from time to time.

The system has responded well under varying situations from a fairly fixed system handling analogue devices with interrupt rates of up to 5000 a second to a system providing a conversational RJE terminal handling a graphic terminal, line printer and a graph plotter.

CHAPTER 3

DESCRIPTION OF SYSTEM TWO

GOALS

The second system was designed for operation in a medium-sized PDP 11. At least 16k words of core, a memory management unit, a disc or similar fast mass storage device, a terminal and a clock are required. A fully self-supporting system requires 28k words of core in order to support the compiler.

The system was designed with five main aims:

Running user programs

The system is designed to run general user programs. Normally, about twenty simultaneous programs should be supported, but this figure should be a parameter at system generation. Each program will run 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, should be fully protected from the failure of a user program.

Multiple terminal support

The system should support multiple terminals; each terminal should, optionally, be linked to a command language interpreter which will enable the user to initiate and control programs from the terminal.

Peripheral support

The system should support a wide range of peripherals, eg, line printer, card reader, paper tape reader and punch, various discs, magnetic tape and a synchronous communication line running under a number of protocols. It should be possible to add new peripherals with minimum disturbance to the system.

Swapping

The number and size of user programs should not be limited to the physical store size of the machine; a limited swapping strategy should be implemented to support a virtual store size of two to three times the physical store size.

Minimal resident section

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

CONSTRAINTS

The core constraints which dominated the first system do not dominate this system. There are two main reasons for this:

1) the target machine is larger than that for the first system

2) the system is not entirely core resident.

This means that device handlers that are inactive will not be using valuable core; for example, it is feasible to have a more complex terminal handler, because those that are inactive will be swapped

out. The time required to swap a program will be in the order of 1/5 of a second, and to minimise the effects on response the system tasks will be able to lock themselves down while they are active.

It is still desirable to keep the resident part of the system as small as possible to enable the system to run in smaller core configurations. This constraint affects the overall design, distinguishing it from systems with a large set of facilities like UNIX and RSX11D which require 48k words of core to do useful work.

STRUCTURAL OVERVIEW

The design of this system stems largely from the previous system; however the basic task synchronisation mechanism is now a 'message', rather than the semaphore of the first stystem. The concept of messages comes from EMAS, the GEC 4080 and others.

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 as an otherwise standard system task.

The kernel provides the following services:

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 core allocation control.

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 system task may also request to be held in core, as the main disc task does permanently.

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 segments 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. The number of segments owned by a task is not limited to eight, and a

'segment stack' is used to hold the non-mapped segments. A segment may be mapped in a read only or a read/write mode which allows protection of code areas.

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.

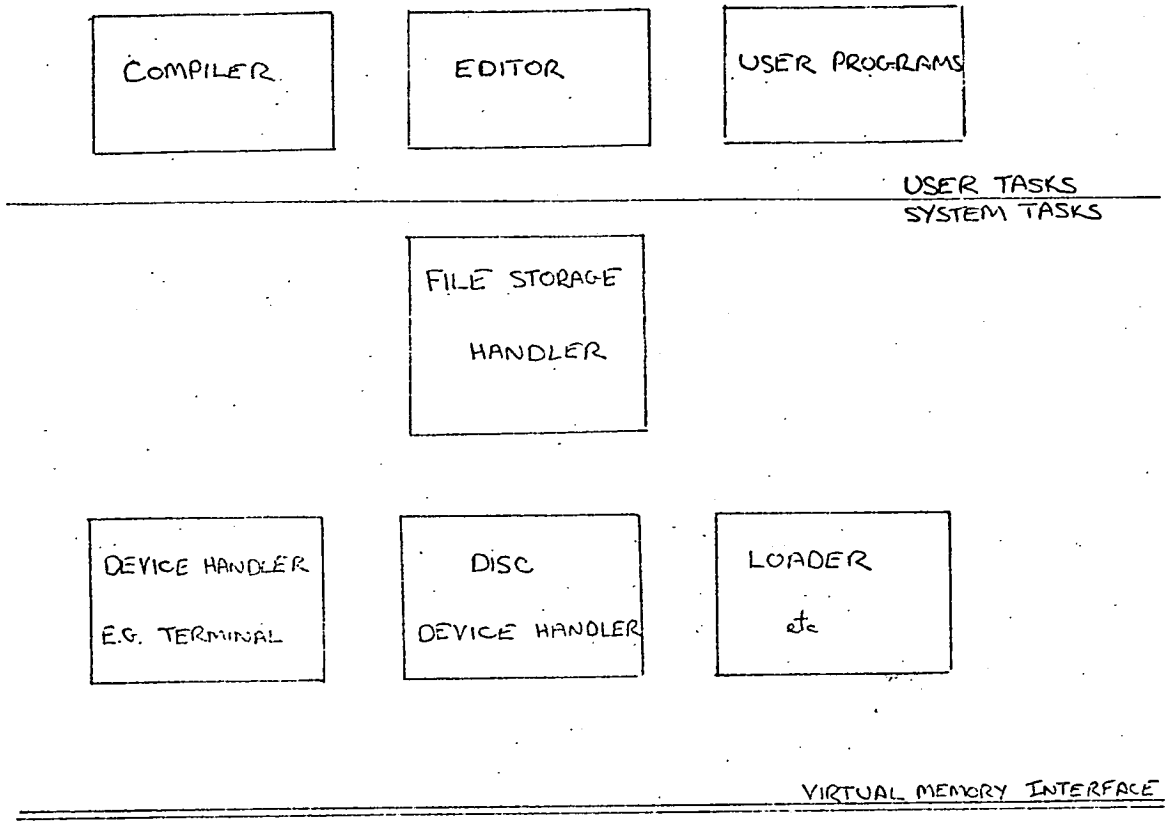
The list of segments used by a task is held in a GLOBAL SEGMENT TABLE within the kernel, with the core or disc address, access permission and the number of tasks using the segment. This table enables the kernel to maintain control over the usage of segments and can easily determine what parts of tasks may be swapped out. The core address, access permission and a pointer into the global segment table are also maintained within the task descriptor to speed up the context switch.

If a task fails with either a hardware fault, eg, an address error or memory protection violation, or with a fault detected by software, e. g., an illegal supervisor call or message, 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 they are requested. Interrupts are handled similarly; the kernel generates and queues 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' thenselves 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 callers VM to the receivers VM. Currently this mechanism is only used to share segments, which are eventually returned to the caller. There is no restriction 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.



STRUCTURE OF THE SECOND SYSTEM

IMPLEMENTATION

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

1) The implementer has had a long experience with IMP, using it as a systems programming language on EMAS and other systems.
2) There was no easy access to any of the other possible languages, with the exception of FORTRAN, which was not considered to be as good as IMP as a systems programming language, either in its structure or its implementation.
3) A new implementation of IMP was available, to a standard that made the project possible.

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 forced 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 disproportionate 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.

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

OPERATION

This system has been in operational use on two machines since May 1976, and is occasionally used on a third machine for a special project, the interactive benchmarking of an ICL 2970. At this stage swopping and the use of more than eight segments have still to be implemented, though most of the necessary kernel features are already present. The multiple terminal support has not been tried, owing to a lack of hardware.

It is still too early at this stage to evaluate this system properly, but it is currently being used by seven other people in widely differing ways and is proving satisfactory.

CHAPTER 4

A QUALITATIVE COMPARISON

A direct comparison between the two systems is difficult for three reasons.

1) The second system was designed to run on a more complex machine than the first system.
2) The implementer was more competent in implementing the second system, although, of course, new mistakes were made.
3) The second system supports a far more comprehensive user interface.

If allowances are made for the extra complexity of the second system, comparisons can be made under the following headings:-

1) Appropriateness of IMP as a system programming language
    A) Structure of IMP, in particular the routine structure
    B) Appropriateness of the data structures and the operators on them
    C) Appropriateness of IMP as implemented

2) Implementation considerations
    A) Initial implementation
    B) Faults
    C) Extensions

3) Physical considerations

A) System size

B) System speed

4) Interfacing costs

1) APPROPRIATENESS OF IMP AS A SYSTEMS PROGRAMMING LANGUAGE.

IMP has shown itself to be an appropriate language for system implementation in large system environments. All of the software of the EMAS system for the ICL 4/75 is written in IMP, as are the compilers. However it does not follow that IMP is equally good for system implementation on small machines, in particular the DEC PDP 11. To judge how useful a language is on a particular machine it is necessary to look at the semantics of the language and the architecture of the machine. A language which closely matches the architecture of one machine may not fit well into a machine with a different architecture. An example of this is a language designed for a multi register machine, unlikely to be efficiently implementable on a single register macine - viz a language which allowed explicit use of the machine registers. IMP is a general purpose language and although its facilities are constrained by what has been efficient to implement on the 4/75, despite this constraint it does not contain many features that are dependent on the 4/75. This has been shown by its implementation on a range of machines, PDP 9,15,10,8, Interdata and the Nova.

A) STRUCTURE OF THE LANGUAGE, IN PARTICULAR ITS ROUTINE STRUCTURE.

A great part of the clarity which can be achieved when writing in IMP is attributable to its overall structure. It is easy for the programmer to structure his program, splitting it into sections and routines. These routines have a clearly defined entry point. The entry to an IMP routine sets up a new environment in which there are three types of variables:-

1) Parameters.

2) Local variables.

3) Global variables.

The parameters to the routine are clearly defined, usually making it unnecessary to refer back to the call of the routine to understand the function of the routine. The local variables, present on the stack only during the execution of the routine, isolates the routine from other sections of code and saves stack space. For complete clarity it would be preferable if the global variables were also defined in the body of the routine, IMP does not do this, which forces the programmer to search around other parts of the program to determine the type and other features of a global variable, or, of course, supply adequate comments in the source.

A result may have to be passed back from a routine but frequently a vector or record result is required, for example, a value and an associated flag is needed. IMP satisfies the first case; a '%FUNCTION' is defined, which returns a single result.

IMP does not allow multiple results to be returned, so the programmer is forced to use a global variable, or pass an additional '%NAME' type parameter to the routine, this being less efficient than the equivalent construct in assembler, which would return the results in two registers.

An operating system does not usually require routines of great complexity, a 'simple' single level non-recursive routine structure being quite sufficient. The IMP routine structure is very powerful, as it allows recursion and multiple lexical depth. The consequence of this is that on some machines it is expensive to enter and exit, as for example on the PDP11, where the stack must be carefully increased to allow for the local name space and parameters, and decreased again on exit; this is especially critical as the hardware also uses the stack and may at any time temporarily plant words on the top. The effect of this can be seen in the kernel where there was a great reluctance to put any routines into the the time critical areas; thus eventually only three routines were used. The entry and exit sequence has since been partially optimised and the entry is seven words and the exit is four words all of which are placed in line. Since this was introduced, the kernel has been slightly rewritten and routines used a little more. In other modules a much heavier use has been made of routines. The residual overhead, however, is still appreciable and further reductions would be welcome but would appear to require the addition of an extra pass to the compiler.

In this instance, CORAL is preferable to IMP, for systems programming at least, because routines are assumed non-recursive unless the compiler is informed otherwise. This, coupled with the static stack, should allow faster routine entry/exit.

When writing in assembler, the 'jump to subroutine' instruction is simple and cheap. The pressure to optimise the code leads to assembler routines in which there is no definable beginning or end, merely various entry and exit points. These same pressures lead to the use of global variables only, usually coupled with a dangerous use of registers. This leads to a very heavy use of 'routines', many with just one or two lines of code, and whole sections of code are just a series of register manipulations and subroutine jumps.

B) APPROPRIATENESS OF THE DATA STRUCTURES AND THE OPERATORS ON THEM

The systems programmer wants to be able to create data structures that are simple to use, but which reflect accurately what he is trying to do. A hierarchical structure is probably most suitable as it will allow different components of the system to operate on separate parts of the structure. For example, a message which is going to be transmitted into a communications network will normally have a hierarchical structure, the actual data at the centre, embedded in, possibly several, layers of protocol. There will be several sections of code each processing only a part of the structure and doing only minimal operations on

other parts. For complete clarity it seems beneficial to have one global definition of the structure, each section then having an expanded definition of its own part.

IMP goes part way towards this goal as it is possible to group collections of data objects, eg, integers, bytes, strings, arrays and pointers into a set called a '%RECORD'. It only partly satisfies the requirement because the possible operations on 'records' are very limited. The definition of IMP only allows 'records' to be assigned, either to one another or a constant. The lack of further operations, for example a properly defined 'compare' and other bit manipulations, lead to inefficiencies in the code. The need for these extensions can be seen in the file storage handler where the name of a file is contained in a six element byte array. At times this must be processed byte by byte but at other times it is much better to consider it as one entity, eg, when comparing it to another name. Without the ability to define a 'record compare' in IMP it must be done in a roundabout way, the result being far less efficient code.

The 'TABLE' feature of CORAL is somewhat different and the author feels, having had experience of both, that the IMP 'RECORDS' are both more powerful and easier for another programmer attempting to understand. The lack of such facilities in both FORTRAN and ALGOL60 reduces their usefulness.

The PDP11 hardware contains operators that the compiler writer

finds great difficulty in using. These are the auto increment and auto decrement operators that assembler programmers tend to make heavy use of. The definition of operations on complex data structures wouuld give much more scope for these operators to be used. An alternative way would be to include these operators directly in the language. There are two problems with this. Firstly, their use would not be efficiently transferrable to many other machines, thus including an essentially machine-dependent feature into the language. Secondly, it would increase the complexity of the coding, making the understanding of it that much more difficult.

In assembler the building blocks are simply integers and bytes, consequently the programmer can create any data structure that he chooses, and can use it very efficiently using the auto increment and decrement instructions mentioned above. However, the heavy use of these operators will produce code that is absolutely dependent on the individual positions of the variables within the data structure.

C) ADEQUACY OF THE IMPLEMENTATION OF IMP

The question is raised as to whether this implementation of IMP is of a high enough standard, or are we just comparing the quality of this implementation to writing a system in assembler? If the output from the IMP compiler is carefully examined, in particular in a comparison with an equivalent assembler section -

as is done in the next chapter - then it is found that the compiler itself is is producing as tight code as can be expected from any high level language and any large improvements would imply language changes.

2) IMPLEMENTATION CONSIDERATIONS

A) INITIAL IMPLEMENTATION

The first system was developed over a period of three years of part time work. The second system was implemented in four months of concentrated work. Although comparisons are difficult the indications are a factor of three or four to one in favour of the second system. The speed with which the second system was written manifests itself to a certain extent in some rough edges. Some futher effort will be required to achieve the same standard as was achieved on the assembler system.

Even with these qualifications the system in IMP involved far less effort than the system in assembler. The author has been able to identify several reasons for this. First, the tools used. The assembler system was initially written and assembled using the DEC operating system DOS for about a year; it was then transferred to EMAS where it has remained. The nature of its structure and language means that whenever it is changed, apart from truly trivial changes, a new listing has to be obtained to keep track of addresses. This involves a printer listing of some

4,800 lines, which usually takes a long time. The binary is then punched on paper tape which is taken to the target machine and tested. If that version fails then a return to EMAS is usually necessary. It should be possible, by working on the assembler (especially in the area of symbol table size) to assemble on the system itself. The problem of up-to-date listings would become even more acute.

The system in IMP, in comparison, is self supporting and is contained in small parts, the largest being 600 IMP statements. To compile this section takes two and a half minutes. Listings are no problem because they are small and take at most twenty minutes on a 30 characters per second terminal, and also because it is not essential in a high level language to have a completely up to date listing of the module. It is sometimes necessary however to decompile the compiler code to check it for faults, but this is usually done to a disc file and then examined in parts using the editor.

The second reason why the first system involved more effort can be seen by the ease with which code was written. It is always easier to use a language that matches the level at which the programmer is thinking. The implementer has found that IMP reflects what he is thinking more closely. For example, take the routine operation of testing a variable and making a decision. In assembler this involves two separate statements, the comparison, followed by a 'branch on condition'. The IMP

solution is one statement, in the form '%IF', closer to the way of thinking.

B) FAULTS

A fault appears in one of two ways; it is either very dramatic in the form of a system crash, or it is insidious, with no hard evidence to pin it down.

By far the more difficult type of fault to find is the second type. This type has tended to be more frequent in the first system, one such fault having taken a week of hard work to uncover the fault was caused by the use of a variable in a complex data structure that didn't in fact exist in one case, and this resulted in the overwriting of the following variables. The nearest equivalent of this type of fault on the second system was a situation where the system ran correctly for a period of time, then suddenly appeared to run out of core. The fault was found fairly rapidly by careful monitoring of core use. Three identifiable reasons why this type of fault occurs more frequently in the first sytem are:

There is far less of the second system, so there is less likelihood of error.

The 'record' structure in IMP protects the programmer from some of the overwriting faults.

The hardware protection given by the memory management unit enables the programmer to shift some faults from the second category into an immediate system crash. Unfortunately this protection is still limited, especially when running in kernel mode.

The 'system crash' type of fault is far more frequent; it is also easier to identify. This type of fault occurred frequently in both systems. The nature of an assembler program, with its symbol table,makes looking through dumps easier than with a dump from an IMP program. In the IMP program it takes experience to calculate where the compiler has allocated the variables. Either way it is still a time consuming process to examine dumps and it has been found very worth while, in both systems, to write a dump analysis program that interprets at least part of the dump. These programs have meant that there is little difference between the two systems in this respect.

Normally, the programmer would expect that a program written in IMP would give a trace back and the values of the variables when it fails, but this is not available yet on the PDP 11. Although this facility would help enormously with application programs, experience has shown, on EMAS at least, that it is of limited value when it comes to supervisors.

After a crash there is usually more information available in the system in IMP as variables are held in store, rather than in

volatile general registers. For example, in situations where core is disappearing, the details of the last transaction are always available in the system in IMP in the variables that are used in core manipulation. The assembler system, however, uses the general registers and all information is lost on exit from the kernel.

In general a section of code will have three main classes of error.

1) Typing faults.

2) 'simple' logical errors.

3) Design fault.

Typing fault

A typing fault is normally shown up best by the strict syntax of the compiler, whereas the very flexible forms of statement used in assembler mean that virtually anything will produce some code.

'simple' logical fault

A 'simple' logical fault, for example the substitution of one variable for another, probably happens with equal frequency in both the high and low level languages, but as there are many more assembler statements than IMP statements, more will occur in a given piece of code. A more serious example of this type of error occurs in conditions; in assembler, even the simplest condition

requires two separate statements, one to set a condition code and the second to determine the branch from the condition code, whilst a complex condition which still requires only one IMP statement could require thirty or forty assembler statements, giving more scope for faults to arise.

Design fault

The system design type of fault is equally likely in both types of system, but it can be much easier to rectify in the second system. Take, for example, the addition (or removal) of new variables from a collection of variables. In IMP this collection of variables would be grouped together into a complex data structure. All accesses to the variables are made in respect of that data structure, so the addition of new elements simply requires a redefinition of the structure, a one line change. In assembler, even where care has been taken to define a data structure by definitions at the beginning, so that a change should only mean the altering of a list of displacements, the actual effects usually spread throughout the program. This is because use has been made of the more efficient forms of access to variables by using registers as pointers, perhaps into the centre of this structure. Use of these facilities makes the initial displacements useless.

When a fault is isolated it must be cured. With the assembler level supervisor there is a temptation to attempt binary patches, which almost inevitably leads to inconsistent systems on different machines and without a lot of care the patch is forgotten, with the result that after the next change is made to the system one repeats the steps to find the previous fault. Binary patching is several times more complex in the second system and would normally require decompiling of the code, thus probably taking as long as a recompilation of the source.

ADDITIONS

Practice has shown that it is much easier to add new system components to the second system. This has been shown in both small and major parts of the system. A handler for multiple teletype multiplexors (DH11) was written and tested in a period of two days on the system in IMP. This handler was used, not to provide access from terminals to the machine, but to simulate the effect of terminals on a mainframe. The equivalent on the assembler system, though using a synchronous communications line running under a simple protocol, took three weeks to write and test. Part of this difference can certainly be explained by the simpler overall structure of the system in IMP, where messages are used, as against the more complex semaphore structure of the assembler system.

A line printer handler provides an example of a small addition

to the system. The system in IMP version took a period of one hour, including the time to think, write and type the code. On the other hand, the addition of a similar handler on the assembler system usually took three or four retries, requiring in total three or four hours of system time, excluding the thinking time required and the extra typing time, probably a factor of three or four times. An explanation of the differences lies within the structure, or lack of it, of the languages. This has already been discussed but an additional point arises. Since, in assembler, a routine is very cheap to call, a section of code is often called, in slightly different ways, from a number of places. A slight change can be made to the code, without realising the effects elsewhere.

When an addition is made to a system, there will usually be faults in it; the considerations of the last section apply to this, partly explaining why it is easier to add to the second system.

A) SYSTEM SIZE

The second system is currently 72% larger than its assembler counterpart; this difference is reflected in 74% extra code and 68% extra stack requirements. Although this seems a very large difference, in actual space it is less than 4k words. The first system occupies over 4k words and the second system about 8k words. The detailed sizes are investigated in the following chapter.

Competence of the coding

It is equally easy to write bad code in both a high level language or assembler. Parts of the system in assembler have now been rewritten several times, their size now reflects the better coding. The second system is still very new; most modules have not been seriously examined since their initial implementation. However, after seeing the sizes of, in particular, the disc handler and the disc directory handler, an attempt was made to reduce them.

The disc handler was reduced by almost 150 bytes or over 20%. The total effort involved was one hour, including testing time. The disc directory handler was reduced by almost 300 bytes or 15%. This was achieved by reconsidering the data structure and defining it in a different way to simplify the operations on it. It took half an hour, but

there is certainly more scope for reducing the overall size of that module further.

The sizes of the second system are not at all constant, not because the code is being changed, but because the compiler is. Since the system was started several versions of the compiler have been used and quite considerable savings have been achieved in the size. This underlines another factor in favour of using a high level language, viz, if the code produced is too big or for that matter too slow, then changes can be made to the compiler without disrupting the logic of the system. The routine entry/exit mechanism provides a good example of this. The next release of the compiler will have a much faster routine entry implemented for routines that it can establish are non-recursive and otherwise well behaved. This modification will reduce the overhead down from 22.5 micro seconds to the assembler figure of 12 micro seconds for a routine without parameters; extra time will be required for parameters.

B) SYSTEM SPEED

The context switch time on the second system is 70 micro seconds slower than the first system. This difference is due to the memory management registers which take the second system 103 micro seconds to load up. The otherwise basic similarity in timing is not surprising as both code sections are in assembler and carry out similar functions.

The basic kernel functions in the second system are between 70% and 80% slower than the first system. Take for example the routine which inserts a task onto one of the CPU queues. The structure of the routine is similar in both the systems, it calls a second routine which is a generalised routine for inserting items on queues, although the queueing strategy is different. The first system takes 64 micro seconds on the longest path, the second system takes 110 micro seconds. This is an increase of 72%, a figure very comparable to the difference in the code sizes. It is interesting to note that the total routine entry/exit overheads in the first system are 12 micro seconds and 45 micro seconds for the second system. Other detailed figures are given in chapter five.

There are some basic differences between the two systems that affect the overall system overheads.

1) On entry to the first system, care is taken not to use more registers than are absolutely necessary until it is certain that a context switch will be necessary. For example, on claiming a non-busy semaphore, one register is needed by the kernel to switch on the type of call, but on inspection of the semaphore, the return to the user consists simply of restoring that one register and returning.

   When the second system is entered, all the registers need to be saved because the programmer no longer has absolute

control over the use of the registers in the IMP section. For a 'simple' call on the kernel this is a significant overhead.

2) A faster interface for I/O from IMP programs has been used in the second system. This has been possible for two reasons. First, the IMP interface is necessary for the system itself, therefore it has been designed in with the system instead of being added later. Secondly, the memory management unit has enabled a more efficient interface. The effect of this is that the output of a single character to a disc buffer is 27 micro seconds in the second system in comparison to 85 micro seconds in the first system.

The overall effect of these factors is that user programs that are not CPU bound actually run faster on the second system.

4) INTERFACING COSTS

The severe core restraints applied to the first system have shown in the user interface, which is, in general, clumsier to use than the interface in the system in IMP. This difference has of course also shown itself in the relative sizes of the two systems. To a certain extent however, it is a function of writing in the two different languages. It has proved easier and less trouble prone to add in IMP

the features which make the system easier to use. In assembler, there is always a tendency to look at the ever growing size and refuse to add a feature that has only a minor effect apparently, but is very expensive in code.

CHAPTER 5

A QUANTITATIVE COMPARISON

SYSTEM SIZE

Table 5.1 below gives the comparative sizes, in bytes, of the basic parts of the two systems.

TABLE 5.1 COMPARATIVE CORE REQUIREMENTS (BYTES)

MODULE NAME	FIRST SYSTEM		SECOND SYSTEM	
	CODE	DATA	CODE	DATA
Kernel	1732	1559	2538	2678
Hardware interface	212		230	
Disc handler	248	64	488	126
File handler	656	1070	1460	1174
Loader	672	120	2112	544
C.L.I.	898	92	1398	(1)
TERMINAL Input	328	74		
Output	248	68		
TOTAL	576	142	1520	344
IMP task support	280			
Mother task			606	258
I/O routines etc	1472		1538	
TOTALS	6848	3047	11890	5122
'%' increase			+74%	+68%
OVERALL total	9895		17012	
'%' increase			+72%	

Note 1. The data areas of the Loader and CLI are currently combined.

The reasons for the size differences are very varied, so each module will be examined independently, with general notes on their size.

The actual running size of the second system is larger as the memory management forces alignment of all sections on a 32 word boundary.

Kernel

Code

A direct comparison is very difficult in the kernel because of their different functions. The size of the first kernel reflects some functions, like the insertion and removal of characters from small I/O buffers that are carried out elsewhere in the second system. The IMP kernel has a considerable amount of code (about 12.5%) that supports the virtual memories.

Data

The data sizes of the two kernels are equally difficult to compare, various data sections have been moved, or were not necessary in the other kernel; for example, the data size of the first kernel includes 800 bytes for the pool of small I/O buffers, whereas the second kernel includes 500 bytes for system task descriptors, 250 bytes for message buffers and 200 bytes for the GLOBAL SEGMENT TABLE.

Hardware interface

These sizes are comparable as they are both in assembler and perform similar functions.

Disc handler

code

As this module carries out a similar function in both systems one would expect the size to be comparable. However on careful analysis of the second system, one finds two elements of the second handler that are not present in the other one - virtual memory mapping and extensions to use a second disc drive. The VM mapping takes 122 bytes and the extensions 78 bytes. This leaves the total at 288 bytes, a figure much closer to the first system. One clear area where words are lost is in the loading of the disc hardware registers, because the assembler module uses the auto increment instructions, where the IMP version uses a more expensive form.

Data

The extra data requirements can be explained by the facts that:-

1) each IMP program has a fixed data overhead of 60 bytes for linkage and checking
2) local variables are used in the second system that are held in registers in the first system.

File handler (directory task)

Code

This module shows the extent to which a section of code very tightly coded in assembler, with almost no local variables exept those held in registers, compares in size to an equivalent section in IMP. This is excellently illustrated by a scrutiny of the number of words used per assembler instruction (a register/register operation uses one word, a single base and displacement two words and a double base and displacement three words). The following table summarises the results.

TABLE 5.2 FILE HANDLER CODE SIZES

	INSTRUCTIONS	WORDS	AVERAGE
ASSEMBLER	245	333	1.36
IMP	417	730	1.75

As can be seen from the table, programming in assembler can create much tighter code. However that is not the full story. The file handler, like the disc handler, has the following additions:

1) VM support. It is necessary to map the parameters from the callers VM to the file support VM. This costs 130 bytes.

2) Extensions. There are three extensions to the code:

a) Support for a second disc drive: 120 bytes

b) An extra call to rename temporary files: 84 bytes

c) Additional file security: 30 bytes

This is a total of 364 bytes, giving revised comparative figures for assembler, 666 bytes; for IMP, 1096 bytes. These figures are much closer and on instruction count terms yields a figure for assembler of 245 instructions and for IMP of 313 instructions.

To give a concrete example, a routine which is fairly typical has been directly compared in its IMP form to the assembler form. The routine is 'EXAMINE' which searches a file directory for a particular file name.

TABLE 5.3 USE OF INSTRUCTIONS IN THE ROUTINE 'EXAMINE'

	IMP	(extras)	ASSEMBLER
Entry sequence	7		1
Disc unit calculation		15 (1)	
File system number	10		7
File system 0 cycle.	5		8
Block loaded check		20	2 (2)
File cycle	10		14
Name comparison	23 (3)		9
Miscellaneous	4		1
%RESULT	6		5
User entry sequence	13	3	2
TOTALS	78	38	44
Instructions	47		35

NOTES:

1. This refers to additions to the IMP code that calculates the disc unit number.
2. In the assembler module, this section of the code is done elsewhere.
3. The latest version of the compiler has reduced this figure to 9 words.

The actual coding of the two routines is given below.

TABLE 5.4 ROUTINE 'EXAMINE' AS CODED IN ASSEMBLER

```
NUMBER OF WORDS          INSTRUCTION          COMMENT

     ; SVC EXAMINE(HEADER)
 2   EXAM1:  JSR     R4,USAVES        ; SAVE USER REGISTERS
     ;       ALSO CLAIMS THE SEMAPHORE TO ENSURE NON-REENTRANT
     ;       AND SETS PRIORITY TO 3 (HIGHEST USER PRIORITY)

     EXAMINE:                                 ; (R0=HEADER, NEXT=R0)
     ;ON EXIT
     ;       R4=POS
     ;       R0=0 IF NO FILE
     ;       R2 --> CODE
     ;
 1           MOV     R0,-(SP)                 ;  SAVE POINTER TO HEADER ON STACK
 2           MOVB    1(R0),R0                 ; GET FSYS NUMBER
 1           BGE     EXCY                     ; USE DEFAULT FILESYS NO.
 2           MOVB    FILSYS(PSP),R0           ; GET IT FROM PSECT
 2   EXCY:   ADD     #DIRBLK,R0               ; AS AN INCREMENT ON DIRBLK

 2           CLR     CODE                     ; WILL HOLD THE FILE PROTECT CODE
     ;
     ;       LOAD(NEXT)
     EXCY0:                                   ; LOOP FOR MULTIPLE DIR. BLOCKS
 2           JSR     PC,LOAD                  ; LOAD THE BLOCK IN R0
 2           MOV     #4,R4
 2           ADD     WRM,R4                   ; R4 -> START OF ACTUAL BLOCK
 2           MOV     #63,R3                   ; NUMBER OF FILES IN BLOCK
     EXCY1:                                   ; CHECK EACH FILE NAME

     ;  COMPARE THE NAME IN THE DIRECTORY BLOCK
     ;   POINTED AT BY R4 TO THE NEW NAME POINTED AT
     ;    BY THE TOP OF THE STACK
 1           MOV     (SP),R1                  ; RESCUE NEW FILE NAME
 1           TST     (R1)+
 1           MOV     R4,R2
 1           CMP     (R1)+,(R2)+
 1           BNE     EX2
 1           CMP     (R1)+,(R2)+
 1           BNE     EX2
 1           CMP     (R1)+,(R2)+
 1           BNE     EX2

     ; MATCHED FILE NAME, SO EXIT

 1           MOV     (R2)+,R0                 ; GET FIRST BLOCK
 1   EXRT:   TST     (SP)+                    ; POP PARAMETER
 1           RTS     PC                       ; AND RETURN
     EX2:
 2          INC     CODE                      ; FILE COUNT
```

```
2           ADD     #12,R4              ; STEP TO NEXT DESCRIPTOR
1           DEC     R3                  ; BOTTOM OF DISC BLOCK?
1           BGT     EXCY1               ; NO, SO TEST NEXT
2           MOV     WRM,R0              ; CURRENT DIRECTORY BUFFER
2           MOV     2(R0),R0            ; CURRENT BLOCK NUMBER
1           INC     R0                  ; LOOK AT NEXT
2           CMP     #DIREND,R0          ; END OF DIRECTORY?
1           BGE     EXCY0               ; NO
    ;
1   EXRT1:  CLR     R0                  ;(NOTE CREATE USES THIS LABEL)
1           CLR     R4                  ; CREATE USES THIS TO DETERMINE
                                       ; THAT DIR ISNT FULL
1           BR      EXRT                ; AND RETURN TO USER
    ;  END OF SVC EXAMINE
```

TABLE 5.5 ROUTINE 'EXAMINE' AS CODED IN IMP

```
NUMBER OF WORDS          STATEMENT                    COMPILER OUTPUT
-------------------------------------------------------------------------

          - FIRST THE RECORD DEFINITIONS
     53     RECORDFORMAT N1F(BYTEINTEGERARRAY NAME(0:5))
     54     RECORDFORMAT N2F(INTEGER A,B,C)
     56   RECORDFORMAT FILEF(RECORD (N1F) N, INTEGER FIRST, PR)
     61   RECORDFORMAT INFF(BYTEINTEGER UNIT, FSYS, RECORD (N1F) N)
     62   RECORDFORMAT INF2F(BYTEINTEGER UNIT, FSYS, RECORD (N2F) N)


    101
          - THE RECORD 'INF' CONTAINS THE FILE DESCRIPTOR
          - PASSED IN BY THE CALLING PROGRAM
    102     RECORD (FILEF) MAP EXAM(RECORD (INFF) NAME INF)
  1                                                  MOV   R0, -(LNB)
  1                                                  MOV   LNB, -(SP)
  1                                                  MOV   SP, LNB
  2                                                  MOV   LNB, 2(LNB)
  2                                                  ADD   #-20, SP
    103       INTEGER N,J,K,HIT, NORD
    104       RECORD (FILE2F) NAME F
    105       RECORD (INF2F) NAME INF2
          - FOR EFFICIENCY, THE USER RECORD HAS TWO FORMATS
          - 1) A BYTE INTEGER ARRAY OF 6 ELEMENTS
          - 2) THREE INTEGER ELEMENTS A, B AND C
    106       INF2 == INF;     ! MAP 'INF' INTO FORMAT OF 3 INTEGERS
  2                                                  MOV   4(LNB), -20(LNB)
          - THE ELEMENT 'INF_UNIT' CONTAINS THE DRIVE NUMBER
          - AS 'INF' AND 'INF2' ARE MAPPED TO THE SAME SPACE, EITHER
          - CAN BE USED
    107       IF INF2_UNIT#0 THEN DRIVE=UNIT1
  2                                                  MOV   -20(LNB), R0
  1                                                  TSTB  (R0)
  1                                                  BEQ   340
  1                                                  MOV   (GLA), R1
  3                                                  MOV   #20000, -66(R1)
          - 'DIRBLK' IS A CONSTANT, IT IS THE START OF THE DIRECTORY AREA
    108       N=DIRBLK+INF_FSYS
  2                                                  MOV   4(LNB), R1
  2                                                  MOVB  1(R1), R2
  2                                                  BIC   #-400, R2
  2                                                  ADD   #150, R2
  2                                                  MOV   R2, -4(LNB)
    109       UNTIL N>=DIRBLK+3 CYCLE;     ! USE 3 BLOCKS AT BOTTOM
  1                                                  BR    376
  2                                                  CMP   #153, -4(LNB)
  1                                                  BLE   614
    110       NORD = N!DRIVE
```

```
2                                                   MOV   -4(LNB), R0
1                                                   MOV   (GLA), R1
2                                                   BIS   -66(R1), R0
2                                                   MOV   R0, -14(LNB)
          - THIS SECTION IS DONE IN 'LOAD' IN THE ASSEMBLER VERSION
          - IT CHECKS TO SEE IF THE DESIRED BLOCK IS IN CORE
          - IF NOT, IT FREES A BLOCK AND READS IT IN
   111        IF D SAVE#NORD START
2                                                   CMP   60(GLA), R0
1                                                   BEQ   464
   112            IF DWRM#0 THEN WRITE D
2                                                   MOV   64(GLA), R2
1                                                   BEQ   434
2                                                   JSR   R0, -272
   113            D SAVE=NORD
3                                                   MOV   -14(LNB), 60(GLA)
   114            DA(N, X_X, DREAD)
2                                                   MOV   -4(LNB), -(SP)
1                                                   MOV   (GLA), R0
2                                                   MOV   -36(R0), R1
1                                                   MOV   R1, -(SP)
1                                                   CLR   -(SP)
2                                                   JSR   R0, -442
   115         FINISH
          - THERE ARE 51 FILE ENTRIES IN EACH DIRECTORY BLOCK
   116         CYCLE J=0, 1, 50
3                                                   MOV   #-1, -6(LNB)
3                                                   CMP   -6(LNB), #62
1                                                   BEQ   606
2                                                   INC   -6(LNB)
          - SAME AS 'CODE' ABOVE, IT IS THE FILE POSITION (GLOBAL)
   117            FNO = J
1                                                   MOV   (GLA), R0
3                                                   MOV   -6(LNB), -72(R0)
          - 'F' POINTS TO AN ENTRY IN THE DIRECTORY ARRAY
   118            F == FA(J)
2                                                   MOV   -6(LNB), R1
2                                                   MUL   R1, #12
1                                                   ADD   GLA, R1
2                                                   ADD   #100, R1
2                                                   MOV   R1, -16(LNB)
          - COMPARES THE NAMES, AN INTEGER AT A TIME
   119            IF F_N_A=INF2_N_A AND F_N_B=INF2_N_B ANDC
2                                                   MOV   -20(LNB), R2
2                                                   CMP   2(R2), (R1)
1                                                   BNE   604
3                                                   CMP   4(R2), 2(R1)
1                                                   BNE   604
3                                                   CMP   6(R2), 4(R1)
1                                                   BNE   604
1                                                   MOV   LNB, SP
1                                                   MOV   (SP)+, LNB
2                                                   ADD   #4, SP
1                                                   MOV   (LNB)+, PC
```

```
   120              F_N_C=INF2_N_C THEN RESULT == F
   121          REPEAT
1                                                    BR    472
   122          N=N+1
2                                                    INC   -4(LNB)
   123          REPEAT
1                                                    BR    366
          - A FAILURE RETURN
   124          RESULT == NULL
1                                                    CLR   R1
1                                                    BR    572
   125        END
   126
   168
          - WHEN A REQUEST IS RECEIVED FROM ANOTHER PROGRAM
          - THE DESIRED SERVICE IS DETERMINED,
          - AND THE RELEVENT SECTION IS ENTERED BY THE SWITCH 'REQUEST'

   169  REQUEST(EXAMINE):          ! P_A2 CONTAINS ADDRESS OF DESC
   170          F==EXAM(INF)
2                                                    MOV   -40(LNB), -(SP)
2                                                    JSR   R0, -1044
2                                                    MOV   R1, -32(LNB)
   171          IF F==NULL THEN NO=0 ELSE NO=F_FIRST!DRIVE
1                                                    BNE   1354
2                                                    CLR   -52(LNB)
1                                                    BR    1370
2                                                    MOV   6(R1), R0
2                                                    BIS   -66(LNB), R0
2                                                    MOV   R0, -52(LNB)

          - THIS IS COMMON TO ALL THE CALLS, THE FINAL 'RETURN'
          - IN THE ASSEMBLER VERSION ENTERS A SIMILAR SECTION OF CODE.
   172  REPLY:  WRITE D IF DWRM#0
2                                                    MOV   64(GLA), R0
1                                                    BEQ   1402
2                                                    JSR   R0, -1240
   173          WRITE B IF BWRM#0;    ! UNTIL CLOCK IS GOING OK
2                                                    MOV   66(GLA), R0
1                                                    BEQ   1414
2                                                    JSR   R0, -1174
   174          MAP VIRT(0, -1, MYSEG)
1                                                    CLR   -(SP)
2                                                    MOV   #-1, -(SP)
2                                                    MOV   #4, -(SP)
2                                                    MOV   46(GLA), R1
1                                                    JSR   GLA, (R1)+
   175          PX_A1=NO
3                                                    MOV   -52(LNB), -26(LNB)
   176          PON(PX)
2                                                    MOV   #-30, -(SP)
1                                                    ADD   LNB, (SP)
2                                                    MOV   40(GLA), R1
1                                                    JSR   GLA, (R1)+
```

The totals indicate that IMP is 77% larger. However the instruction counts are much closer, especially when the 9 instructions for routine entry and exit are taken into consideration.

Data

The bulk of the data in both cases is made up of the two disc buffers used by the code. The slightly larger size of the IMP version again reflects the fixed IMP overhead.

Loader

The loader on the first system is very basic, as it loads the code directly into core and then initiates the task. The second system has to load into a virtual memory, checking that each segment is within its boundaries.

Command Language Interpreter

The size of the IMP version does not appear to be much larger, this being partly due to features in the first system that are in the loader in the second system. The second system version is larger in comparison because of the tidier user interface.

Terminal

The differing buffering strategies make a comparison useless.

I/O routines

It is not surprising that these two modules are comparable in size as they were both written in assembler.

SYSTEM SPEED

It is more difficult to quote accurate timings for the second system than it is to quote the figures for the first system. The main difficulty is that changes to the compiler can make considerable differences to detailed timing figures.

Kernel

The basic context switching time is given in Table 5.6.

TABLE 5.6 BASIC CONTEXT SWITCHING TIME

First system	280 micro seconds
Second system	390 micro seconds

As previously shown, the difference is explained by the

additional loading of the memory management registers necessary on the second system.

The differing functions performed by the two kernels make other comparison figures very difficult. There is, however, one generalisation that can be drawn. A 'simple' entry to the kernel in the first system, viz, one that does not involve another task, has a basic overhead of only 60 micro seconds because a full register save/restore is avoided. The second system requires the full context switch time as there is no control over the use of the registers. This problem would be avoided on the PDP 11/45 since the second set of registers could be used. One fairly simple optimisation concerning the reloading of the memory management registers which would save the 100 micro seconds has not yet been implemented.

A further comparison can be drawn from the minimum time taken to respond to an interrupt (one that is passed back to task level), Table 5.7 gives the figures.

TABLE 5.7 INTERRUPT RESPONSE TIMINGS

First system	303 micro seconds
Second system	693 micro seconds

These figures show a greater variation than expected, the second system being 111% slower than the first. For a clearer understanding of the figures they can be split into the following sections:

A) entry and register save

B) re-schedule of interrupted process

C) checks and other processing

D) schedule of interrupt-owning process

E) CPU dispatch

F) register restore

Table 5.8 shows the timings for each stage

TABLE 5.8 BREAKDOWN OF INTERRUPT RESPONSE TIMINGS

section	first system	second system
A) Entry	35	57
B) Re-schedule	64	112
C) Checks etc	17	128
D) Schedule	61	112
E) CPU dispatch	83	124
F) Register restore	43	160

The table indicates an increase of about 70%-80% in modules which are basically very similar, eg parts B, D and E. Of the three remaining parts, parts A and C are very much greater because of the more generalised interrupt handling of the second system. Part F contains the 103 micro seconds to load the memory management registers.

In Chapter four, where the routine 'schedule' was examined, it was found a large part of the difference between the routines was due to the IMP routine entry/exit overhead. This is also true of the other critical routines in the kernel, for example, the routine to extract an item from a queue is 56% entry/exit overhead.

User Programs

The I/O interface to an IMP program is much more efficient in the second system; the figures have already been given in Chapter four. The compiler is a good example, making heavy use of the I/O facilities, even though it is fairly CPU dependent. Table 5.9 quotes the figures for a first pass compilation of a 580 statement program (22000 characters).

TABLE 5.9 FIRST PASS COMPILATION TIMES (IN SECONDS)

	ENTRY	EXECUTION
First system	7	75
Second system	6	62

A second example is provided by the EDITOR. On the first system it uses a more efficient I/O interface than a standard IMP program but on the second system it uses the standard interface. Table 5.10 gives the figures for entering the editor and then an immediate exit (this copies the file).

TABLE 5.10 EDITING A 43 BLOCK FILE (22000 CHARACTERS)

	ENTRY	EXIT (all in seconds)
First system	2	15
Second system	1	16

The significance of these figures is that they indicate that in these types of program the extra timing overhead in the kernel

of the second system is negligible and is swamped by other overheads, eg, user interface overheads, disc head movement and CPU time spent in the user program.

CHAPTER 6

CONCLUSIONS

It has been shown that the use of IMP for writing a system for a fairly small configuration provides certain advantages and disadvantages.

The advantages were in:-

initial implementation

fault finding

making extensions

Initial Implementation

The first system, written in assembler, took three to four times more effort to write and commission.

Fault Finding

There were fewer 'trivial' faults in the second system and although the number of more serious faults was similar in both systems, they were easier to cure in the second system.

Extensions

In both minor and major additions to the systems, the second system proved itself more easily adaptable than the first system, again by at least a factor of three to one.

The disadvantages of using IMP were found to be:

extra core requirements

slower execution

Extra Core Requirements

The second system required an additional 4k words of store, 70% more than the first system. This is the more important disadvantage although the figure is overstated because of the differences between the two systems.

Slower Execution

The time spent in the supervisor of the second system is greater than that of the first system. This is, however, in most applications a fairly minor proportion of the total CPU time and it was shown in Chapter five that the time spent at the user interface level is as critical as the time spent in the kernel. The slower kernel execution time will be felt most in two particular instances. Firstly, in an application of the system that would normally expect to spend a high proportion of the CPU within the system itself, for example, a message passing application where little processing of the message is done. Secondly, a time critical device could well cause problems on the second system. With the current version of the compiler it might be necessary to recode the central part of the kernel into assembler to handle such a device. However, this measure is a last resort.

APPROPRIATENESS OF IMP

It has been shown that IMP is a suitable language for small systems implementation but that it does have some deficiencies from the systems implementer's point of view.

Routine Entry And Exit

The current (September 1976) overhead incurred in routine entry and exit is too expensive. Either the syntax should be changed to allow the specification of a 'simple' routine or the compiler should determine this for itself.

Vector Or Record Results From Routines

It would be desirable to be able to return a vector (or a '%RECORD') as a result of a routine, the absence of such a facility leads to inefficiencies in the code.

Extension To Record Operators

The ability to 'compare' RECORDS, coupled with other bit by bit manipulations on RECORDS would both be more efficient and lead to better clarity in programmes.

The scale of the disadvantages of using IMP as listed above are heavily dependent on the actual implementation of IMP on the PDP 11. Since the first version of the compiler there has already been a considerable increase in its efficiency and the quoted figures for the space and time used by the system are only true of the version compiled before 1:10:76. The important point to note is that these

figures can be greatly altered by a re-compilation alone without a single change being made to the system.

FUTURE SYSTEMS

It is not possible from these results to claim that every future mini-computer system should be written in IMP, or another high level language, but it is possible to draw guidelines as to where the use of IMP would be beneficial.

Very Small Systems

Where, because of cost considerations or because there are multiple systems, core limitation is a dominant feature the extra core overheads of using IMP would effectively rule its use out. Examples of this type of system could be a small terminal controller or a microprocessor dedicated to a piece of equipment. A possible treatment of this case, however, would be to write the system initially in IMP, test it out in a limited form, or on a larger machine, then hand code the IMP into assembler keeping the basic structure intact. This solution would overcome the core cost of using IMP while retaining most of the initial implementation advantages. Some cost must be attached to the new set of faults that would be introduced at the hand coding stage. Modifications to the system could be handled by a referral back to the IMP version.

Larger Systems

On larger systems, viz over 8k, it is illuminating to compare

the trends of core costs against programmer costs. Table 6.1 below shows the change in core costs over a number of years. Table 6.2 gives some indicative costs of programmer time.

TABLE 6.1 CHANGE IN CORE COSTS

Apr 1972	£7040 for 16k words (DEC)
Mid 1973	£4930 for 16k words (DEC)
Mid 1976	£3000 for 16k words (DEC)
Mid 1976	£3000 for 32k words (OEM)

TABLE 6.2 CHANGE IN PROGRAMMER COSTS (ERCC EXTERNAL RATES)

1972	£28 a day
1974	£34 a day
1976	£68 a day

The tables show very clearly that whereas the cost of core is dropping very rapidly, programmer costs are increasing. The effect of these trends is that measures which reduce programming costs, even though they are at the expense of extra core costs,

will become even more significant. The extra programmer time to balance out the cost of the additional 4k words required by the second system is only 11 days. This holds when a single system is being considered, or even a small group of systems, but the argument changes when a manufacturer's own system is under consideration, where the number of machines will be in the hundreds or thousands. However in this situation there are still advantages in using a high level language. A manufacturer on the scale of DEC has to support a continually changing range of peripherals. Therefore the savings which can be achieved in the ongoing support of systems by the use of a high level language should not be ignored.

APPENDIX 1

USER MANUAL FOR THE FIRST SYSTEM

USER MANUAL FOR MUSS11

B. GILMORE
JUNE 1975

CONTENTS

GENERAL FEATURES OF MUSS11

Muss11 is a system designed to allow multi programming and multi user access to PDP11S without memory management. It is a general purpose system that has the capability of running in a real-time environment. Although it will run on any member of the PDP11 family, it will not take advantage of the memory management/protection features of the larger machines.

A disc, or other form of mass storage, is desirable, but is not a necessity.

The system will multi-program up to 128 separate programs, running on one, or a number of terminals. All programs are core resident and may run at one of eight priorities - corresponding to the eight machine priorities, although for normal programs, only the bottom four are used. As a rule programs are written in IMP, but PAL11F programs can be run with certain restrictions (the restrictions are listed later).

When the system is running, the highest priority program able to run is given the CPU. If there are several programs of equal priority able to run, they are time sliced at not more than 20 milliseconds each.

The priority structure ensures that the system can be used in a real-time environment when very fast responses to interrupts are required. A program may dynamically change its priority up or down in time critical areas to ensure response.

Sections of code on the system may be shared between users, thereby cutting down on the total amount of core required. For example, several users may use the editor, each having his own private storage but sharing one copy of the editor code.

An ASSEMBLER (to stand-alone PAL11F standards) is available for systems with at least 16k words of store, and it is hoped to provide an IMP compiler on systems with 28k words within the next few months. Alternativly programs can be written, compiled and assembled on EMAS, then the binary is loaded on the system.

COMPILING AN IMP PROGRAM

The Imp program should be compiled and assembled on EMAS (for full details of the language and how to run it see documentation by K. Yarwood ERCC (ext. 2636) - for the assembler see B. Gilmore ERCC (ext. 2636).

1) A program requires a short header to inform the system of its starting address, priority, name etc. The format is (as added into the IMP program):-

```
*ENDCO          ; !This is the overall length of the code
*.ASCII /xxxx/; !4 character program name
*0              ; !'subsystem' number (0 is default)
*MAIN+10        ; !starting address (MAIN is a label
                ; !   defined by the compiler)
*40             ; !priority to run at:-
                  ! 0 - Priority level 0
                  ! 40 - Priority level 1
                  ! 100 - Priority level 2
                  ! 140 - Priority level 3
*0              ; !descriptor length (0 is default)
*2000           ; !stack length ( = space for arrays + 1000 octal)
```

e.g.

```
*ENDCO
*.ASCII /SORT/
*0,MAIN+10,40,0,2000
```

2) A control statement is required to force the compiler to
   1) Produce POSITION INDEPENDANT CODE (%CONTROL 8)
   2) Special code for MUSS11 (%CONTROL 8192). This control value effects the code dumped for EXTERNALS, REALS and STRINGS.

   This should be put <u>before</u> the %BEGIN

```
%CONTROL 8200
%BEGIN
```

   The %CONTROL statement may be expressed in either octal or Hex in the standard format, e.g.

   %CONTROL 8200 may be written %CONTROL 0'20010' or
   %CONTROL X'2008'.

   The standard bits for setting checks on or off may also be used, for a full list see Appendix one. The more useful ones are:-
   1) Line number updating (%CONTROL 1)

2) Array bound checking (%CONTROL 4).

Summary:

use %CONTROL 0'20015' for checks ON
%CONTROL 0'20010' for checks OFF

The header for an external routine/fn/map is different and is explained in the section 'using external routines'.

RUNNING THE PROGRAM

The Binary output from the assembler may be 1) punched on tape, 2) punched on cards or 3) sent to the relevant remote.

For tape and cards, the program 'BINY' is used to load the tape/cards to the disc, 'TRANY' is used when a file comes off the line. These two programs are discribed in the section of utility programs.

The file on disc is then 'RUN' - by using the operator command 'RU' in 'OIT' (see command A1 under operator commands). This command instructs the loader to load the file into core, then to set up a system descripter (PSECT) and tell the system to enter it.

The 'sub-system' will then grab space for the programs stack, setting up its 'R1' and its 'SP' and then entering the program at the starting address and priority described in the header.

If the program executes correctly (or has a standard IMP fault) then the message 'STOPPED AT LINE x MIN STK xxxxxx' is output and the program is completely purged from core. (The message 'MIN STK xxxxx' refers to the minum stack the program had while it is executing - this is tied up with the parameter specified on the header).

On the other hand if the program causes an address error or an illegal instruction, then :-

XXXX TASK ERROR!

is output. The program in this case is not purged from core, allow the cause of the fault to be determined, and so must be allowed to go by giving the command 'PU XXXX' (see operators command A7). (XXXX is its program name).

1) PRINT and READF (real read), if required, should be declared as external routine at the top of the program.

```
%EXTERNALROUTINESPEC PRINT(%REAL X, %INTEGER A,B)
%EXTERNALROUTINESPEC READF(%REALNAME X)
```

2) To create a new external routine file.

   1) only one external routine per file is allowed.
   2) compile with normal control options.
   3) use the following header (example uses %EXTERNALROUTINE ABCDEF)

```
*ENDCO-.
*.ASCII /ABCD/
0,ABCDEF,<PRIORITY>,0
                                    ( or 0,16,40,0)
%CONTROL X'2008'
%EXTERNALROUTINE ABCDEF(%INTEGER A,B,C)
      .
      .
      .
      .
%END
%ENDOFFILE
```

3) The routine should be put in a file named ABCDYY (if the name is less than 4 characters then pad out with Y's e.g. AYYYYY).

In general on MUSS11 all input/output is line orientated, ie a program receives an entire line of input at once, and nothing is actually output until the program issues a newline (or SELECT OUTPUT). However as the PDP11 version of the IMP compiler does not cater for PROMPS in the EMAS sense, the general effect can be obtained by using 'PRINTSYMBOL(0)'. PRINTSYMBOL(0) executed by a program will immediately send the current line to the output device - without a terminating newline. The zero character is not printed. For example, if %PRINTTEXT'DATA:'; PRINTSYMBOL(0) is used in a program outputting to the teletype, it will output the 'DATA:', leaving the carriage positioned past it; the user can then type a reply on the same line. However it will not repeat itself if the user types a newline and the program still requires data. The program MUST re-issue the message.

It is perfectly reasonable to have two or more programs outputting to the same teletype. Complete lines from each program will be generated and will appear on the device, the lines alternating between each program. There is a problem however, in having two programs simultaneously inputting from one terminal. It will work, BUT one line of input will go alternately to each program and it will not always be possible to determine which program will get the next line of input.

RUNNING PAL11F PROGRAMS

PAL11F programs may be run on the system if they conform to the following rules:-

1) They must be position independant.
2) The I/O must conform to MUSS11 I/O.

   For example, the TELETYPE must not be accessed directly, it can only be used by issuing the relevant SVC.

3) In most cases, because of the differences in I/O, a program written to run under DOS will not run under MUSS11 without a major re-write (as is the case in switching between DEC systems).

If starting from the beginning, it will usually be easier to imbed the program in an IMP looking header (in assembler). In this manner it will be fairly easy to use the I/O facilities of IMP itself, while still allowing a fairly free use of ASSEMBLER. For further details please contact B. Gilmore.

OPERATOR CONTROL OF THE SYSTEM

An operator can control the system by typing the sequence 'CTRL+R' (denoted as ^R) (which echoes OIT:) and a command.

Notes:

1) only the first two characters of the command word need to be typed.
2) all parameters should be separated by at least one space.
3) the sequence is terminated by a CR.
4) a program's identifier may be used instead of its name.

Commands:-

A) Program control

1)	RU(N) <filename>	run a program
2)	SU(SPEND) <prog name>	suspend a program (when it is next awake)
3)	CO(NTINUE) <prog name>	allow a program to continue
4)	KI(LL) <prog name>	kill a program
5)	LO(AD) <file name>	load a program into core
6)	ST(ART) <prog name>	start up a previously loaded program
7)	PU(RGE) <prog name>	Remove a program from core after it has crashed.

B) System information

1)	PS(ECTS)	list all the programs in the system
2)	DU(MP) <from> <to>	dump out an area of core
3)	EX(AMINE) <prog name>	dump out a program psect
4)	MO(DIFY) <octal>	examine a core location
5)	NE(XT)	examine the next core location
6)	RE(PEAT)	examine the location again
7)	SC <word> <contents>	Set a word in the communication area.

C) Card reader control

1)	CR ASC	tell the card reader to read in ASCII mode
2)	CR BIN	tell the card reader to read in binary mode
3)	CR EOT	marks the end of a pack of cards.

DETAILED DESCRIPTION OF THE COMMANDS.

A1) RUN <filename>

The loader attempts to load the named file, create a descriptor for it (called a PSECT) and start it executing.

1) if the file does not exist (or the file is not in the correct format) then the message 'EOF!' is put out.

2) If it loads correctly, then the program name, address of the file and its system number is output:-

PROG <address> <identifier>

3) If the loader is busy the 'OIT' types NO!

4) If there is no core the 'OIT' types 'NO CORE!'

A2) SUSPEND <progname>

A program may be temporarily halted in its execution to allow

1) the debugging program to be run

2) to wait for an operator to put a special (i.e. non-system) device on line.

3) before a program is killed.

This command does not take effect until the program next has a chance to get the CPU.

A3) CONTINUE <progname>

This command is the converse of the suspend command, allowing the program to continue its execution.

There are two main occasions this is used

1) after a 'SUSPEND' has been done

2) after a program/system test has issued the 'WAIT' SVC to wait for some operator action e.g. after cards have been put in the hopper, 'CO CR' is typed, or if the line printer was off-line - 'CO LP' is typed after it is put on-line.

A4) KILL <prog name>

A program may be aborted by Killing it.

The following sequence must be followed.

1) SU <prog> to get it to the 'wait state'

2) KI <prog> instruct the supervisor to kill it

- the message:-
  n termination req.
  STOPPED AT LINE n MIN STK x
  is output and the program is removed from core

A5) LOAD <file name>

The command does the local phase of the 'Run' command. the loader replies:

PROG <address>

The error messages are as in run.

This command may be used to load an external routine in order to put patches in the file before running the main program.

A6) START <progname>

This command is used to start up a previously loaded program, the loader outputs

<identifier>

A7) PURGE <progname>

This command is used to remove a program from core after it has aborted with an address error or similar fault.

B1) PSECTS

This command lists the programs and system tasks in the format:-

name address state
name address state
.
.
.
etc.

B2) DUMP <from> <to>

Areas of core (or device registers) may be dumped out, dumping

always starts from a 20 byte boundary.

If an illegal address is specified the 'OUCH!' is output. A dump can be stopped by hitting ESC.

B3) EXAMINE <prog name>

The descriptor (PSECT) of a program (its state, prompts system variable, I/O definitions etc.) is dumped out.

B4) MODIFY <address>

A location (given in octal) may be examined and/or modified by using this command e.g. (system response is underlined)

```
MO 1000 (CR)
001000 : 000001 : ^R 123456 . (CR)
                           >
                           ^
```

If the location is not to be modified then the number is left out. Typing a dot will then examine the next location, the '>' will use the contents as the address for examine and ^ will go to the previous address.

B5) NEXT

This examines the next location, responses are the same as for examine.

B6) REPEAT

This repeats the last location.

B7) Set Communication word.

This command sets the word 'word' in the communication area to 'cont'. See EXTERNAL INTEGER MAP COM for details of its use.

Card Reader Commands

C1) CR ASC

This command switches the mode of sending cards to ASCII (which is the default), it will take effect on the next card read.

C2) CR BIN

Is similar to CR ASC but switches the mode to binary.

C3) CR EOF

This command tells the card reader handler to signal an 'end-of-file' to the program reading from it.

1) LINE PRINTER

The system task which handles the line printer will assume it is on-line and ready. If it is not, or the printer stops (out of paper etc.) then the message:-

LP OFFLINE!

is output on the main teletype.

The line printer should be put on-line, or fixed, then:-

^R CO LP (See operator command A3)

is typed on any console.

The message:-

LP TASK ERROR!

will appear if the device registers are removed or changed or if there is an error in the handler (rare!).

2) PAPER TAPE READER

On IPL (initial program load), the tape reader will attempt to start up, the message:-

PR OFFLINE!

will be output if there is no tape (normal condition). This message is also output when the reader reaches the end of a tape.

When a tape is to be read, it is put in the reader and

' ^R CO PR'

is typed on any console. The reader should read a section of tape and will then be ready for use.

'PR TASK ERROR!'

will be output for similar reasons as above.

Note: an 'EOF' is assumed at the end of each tape.

3) CARD READER

This is similar to the paper tape reader with the difference in the End-of-file handling as described in operator commands C1, C2 and C3.

4) DISC AND DEC TAPE

The messages:-
DK OFFLINE!
DF OFFLINE!
DT OFFLINE! - are output if the relevant device is off line,

or if there is a read/write error on a transfer.

A) EDITOR

OBJECT FILE: EDITY

This editor is a PDP11 version of the COMPATIBLE CONTEXT EDITOR which runs on EMAS, PDP15S and PDP8S.

OPERATING PROCEDURE:

When started, the editor prompts with:-

```
EDIT VN.M
#
```

The user responds by typing the input and output file names in the form

```
OUTPUT FILE NAME < INPUT FILE NAME     or
OUTPUT FILE NAME
```

1) If the input file does not exist, or is not specified, then a new file is created.

2) A temporary file is created, so, at the end of the editing session the 'old' output file is destroyed and the temporary file is renamed.

3) To edit a file to itself, the form FILE NAME < FILE NAME is used.

4) Editing commands are as in H. Dewar's 'Compatible Context Editor' with the following differences:-

   A) '%T' will close all the files then restart the EDITOR.

   B) The EDITOR operates with a 'window' of the file held in core, this will normally be transparent to the user, the command 'M-0' is an exception as it can only move to the top of the 'window'.

   C) No secondary stream is implemented.

B) FILE TRANSFER PROGRAM

OBJECT FILE: TRANY

This IMP program is a general file/file and file/peripheral transfer program, including transfers to and from a synchronous communications line in IMP 2780 protocol.

OPERATING PROCEDURES:

When TRAN is ready to accept a command it prompts:-

TRAN:

The format for the response is:-

```
FILE/DEV < FILE/DEV,FILE/DEV or
DIRECTIVE
```

Where

```
FILE/DEV is FILE NAME or
DEVICE NAME
```

A file name is defined as a group of alphanumeric characters with a length of one to six characters.

```
'DEVICE NAME' is defined as:-
        KB: (keyboard input) or
        TT: (teletype output) or
        LP: (line printer) or
        PR: (paper tape reader) or
        CR: (card reader) or
        RJ: (symchronous line)
```

The following are legal directives:

```
ST: - Stop TRAN.
SI: - Force a signon to the remote machine.
NS: - Start without a signon.
EF: - Force an end-of-file to the remote machine.
BI: - Do the transfer in BINARY mode.
```

If a command is typed incorrectly, the message 'FORMAT?' is output and the prompt is re-issued.

Using the form 'FILE/DEV,FILE/DEV' causes a concatenation of the two input streams into the output file or device.

1) TRAN will normally sign-on automatically when the first reference to 'RJ:' is made, the message 'SIGNED ON' is output, then it obeys the request.

2) A number of files may be sent down the line without an end-of-file being sent. An end-of-file is automatically sent on a change of mode to receiving, or may be done explicitly by the directive 'EF:'.

3) In receive mode, non-transparent files (i.e. listing file etc.) are assumed. If a binary file is recieved then TRAN will ignore the current command and request a file name to write the binary file to.

4) A file called 'signof' should be transmitted to close down the link.

C) THE FILE LISTING PROGRAM: FLIST

This program can be used to perform the following functions:

1. List the disc file directory

   a) unordered
   b) ordered alphabetically

2. Destroy files

3. Rename files

4. Search for the existence of specific files or groups of files

COMMANDS

When the program expects a command the symbol '>' is output

E.g. >A

Command		Description
A	:	Give an alphabetic listing of the file directory
F	:	Gives an unordered listing of the file directory
L	:	Look for a file or group of files. Fails if not found
D	:	Destroy a file or group of files. Fails if no file or group of files.
R	:	Rename a file. Fails if new file name exists
B	:	List size of a file or group of files.

To get a group of files we use the 'wild' character '?' to stand for any other character. If it is the last character input then it is propagated right

i.e. 'F?' is expanded to 'F?????'

EXAMPLES

```
>L
MASK:PRT001
```

Checks if file PRT001 exists

```
>L
MASK:P?
```

Lists all files beginning with P

```
>L
MASK:P?T?
                Lists all files beinning with P and  third  letter
is a 'T'

>D
NAME:PRT001
PRT001?:Y       If a 'Y' is input then file is destroyed otherwise
not.

>D
NAME:PRT?
PRT001?N
PRT003?Y        Destroys   file   'PRT003'  but  not  'PRT001'  or
'PRT004'
PRT004?N
>

>R
OLD FILE:TEMP   Re-name file 'TEMP' to 'FLIST'
NEW FILE:FLIST
```

D) ARCHIVE PROGRAM

OBJECT FILE: ARCHY

This program reads and writes files to DECTAPE in a format compatible to the 'local' format used on the ERCC PDP15 and the PHYSICS DEPT. 11/45.

OPERATING PROCEDURE

Archive indicates readyness to accept a command by typing:- 'ARCH?'

The user should mount a DECTAPE on unit 0 before issuing the first command.

The user by typing the appropriate command on the keyboard may:-

a) Zero a DECTAPE.
b) Print out the directory of a tape.
c) Archive a file to DECTAPE
d) Restore a file to disc from DECTAPE.
e) Delete a file on the DECTAPE
f) Stop the program

A command consists of:-

DIRECTIVE (cr) or
DIRECTIVE (space) FILENAMES (cr)

A DIRECTIVE consists of the first two letters of the command verb.
FILENAMES is one or more file names seperated by commas.

COMMAND DESCRIPTION

a) ZERO a dectape
FORMAT: ZE (cr)

This command will delete all the files on a DECTAPE, reclaiming the space. It must be used when a DECTAPE is used for the first time.

b) DIRECTORY
FORMAT: DI (cr)

This command generates a listing of the DECTAPE directory on the teletype in the form:-

```
FILE NAME xx nnnnnn mmmmmm
      .
      .
```

Where 'xx' is the LENGTH of the file, 'nnnnnn' is the first block the file occupies and 'mmmmmm' is the last block.

c) ARCHIVE
FORMAT: AR (space) FILENAMES (cr)

The file(s) is written out to the DECTAPE
1) If the file does not exist (on the disc), the message 'FILE DOES NOT EXIST' is output.
2) If it is already archived on the DECTAPE the message 'FILE ALREADY ARCHIVED' is output.
3) If the DECTAPE is full, then 'DEC TAPE FULL' is output.

d) RESTORE
FORMAT: RE (space) FILENAMES (cr)

The file(s) is written from the DECTAPE on to the disc (with the same name).
1) If the file exists on the disc, it is overwritten.
2) If the file can't be found on the DECTAPE, the message 'FILE NOT ON ARCHIVE' is output.

e) DELETE
FORMAT: DE (space) FILENAMES (cr)

The file on DECTAPE is destroyed
The message 'FILE DOES NOT EXIST' is output if the file can't be found.

Note: This command frees the directory entry, but does not reclaim the file space on the tape unless it was the last file to be put on the tape.

f) STOP
FORMAT: ST (cr)
The program terminates

A %SPEC must be given for each routine.

1) %EXTERNALROUTINESPEC WAIT(%INTEGER ticks)

This routine suspends the program for the given number of ticks (1 tick = 1/50 sec).

If ticks=0 then it is suspended until the operator releases it with 'CO PROG'.

2) %EXTERNALROUTINESPEC NAME(%BYTEINTEGERARRAYNAME KEY)

This routine reads in a file name from the currently selected input stream [up to 6 chars - '0'-'9' and 'A'-'Z'].

The array should be declared in the form

```
%BYTEINTEGERARRAY KEY (-1:9)
```

[an immediate return is made if KEY(1)=99].

3) %EXTERNALROUTINESPEC DEFN(%BYTEINTEGERARRAYNAME BUFFER, NAME, %INTEGER STREAM)

This routine sets up a STREAM or SQFILE definition.

%BYTEINTEGERARRAY BUFFER(0:530) is for the internal use of the stream/SQfile
%BYTEINTEGERARRAY NAME (1:6) is a file name (see %ROUTINE NAME)
STREAM is the required stream.

This routine should be called prior to using a disc file e.g.

```
DEFN(BUFF, KEY, 2)
SELECT OUTPUT(2)
WRITE(I,5) etc.
```

4) %EXTERNALROUTINESPEC WRIT(%INTEGER X)

This routine writes a positive integer on the current output stream with no leading space.

5) %EXTERNAL %INTEGER %MAP %SPEC COM(%INTEGER ELEMENT NO)

Within the supervisor there is a communication area to allow programs to pass information to each other and to control the running of certain 'automatic' programs. The area is 21 words long - designated as 0:24 (octal). The MAP COM allows values to be set into the elements of the area, and be retrieved from it. Although the exact use of each word will be different in each system, the following words are reserved:-

word (octal)
1: - used for cpu timing (by the program CPU)
2: - ditto
3: - holds the day (if set by SETIM)
4: - holds the month (ditto)
5: - holds the year (ditto)

21: - used by ACARD, the card reader spooling program.
22: - used by APLOT, the plotter spooling program
23: - used by ALIST, the line printer spooling program
24: - used by A2780, the 2780-emulator spooling program.

The MAP may be used in a number of ways, for example:-

```
%INTEGER I
I = COM(10);                          ! read the value of
word 10
COM(10) = 2;                          ! set word 10 to 2
```

or

```
%INTEGERNAME COMWORD; %INTEGER I
COMWORD == COM(10);                   ! point COMWORD at
word 10
I = COMWORD; COMWORD = 2;     ! get, then set word 10
```

6) %EXTERNAL %ROUTINE %SPEC DA(%INTEGER MODE, BLOCK, ADDRESS)

This routine is used to read or write blocks to the DISC or DECTAPE.

mode = 0 - READ a block from the DISC.
= 1 - WRITE a block to the DISC.
= 2 - READ a block from DECTAPE.
= 3 - WRITE a block to DECTAPE.

BLOCK - device block number.
ADDRESS - address of 256 word buffer.

The routine may also be called with '%INTEGERNAME ADDRESS', for

example:-

```
%INTEGERARRAY A(0:255)
DA(0, 500, A(0))
```

Which will READ block 500 into A.

6) %EXTERNAL %ROUTINE %SPEC FIDENT(%INTEGER TYPE, %C
%BYTEINTEGERARRAYNAME KEY)

This routine provides information about groups of file names. It is used when it is necessary to automatically allocate file names of a given form, or when groups of file names are being processed.

The first four letters of each DISC file are matched against the four letters provided by the caller in KEY(1), KEY(2), KEY(3) and KEY(4). The final two letters of a matching file are then treated as a number, files in the range 0 to 40 are used, the actual use depending on TYPE.

TYPE = 0 - returns the name of the first existing file
type = 1 - returns the next free file name.

The resultant file name is passed back in KEY(1) - KEY(6). An error return (KEY(0)=-1) is made if

1) there is no file (TYPE=0) or
2) no free file name exists (TYPE=1).

For example - all files of the type 'PRITXX' can be obtained by a program, one at a time by calling FIDENT(0, A) where A(1) to A(4) are set to 'PRIT'.

It should be noted that each file should be 'renamed' or 'destroyed' after use or FIDENT will pick it up again when recalled.

The ARRAY should be declared at least (0:6), or if RENAME is going to be called then (-1:6).

7) %EXTERNAL %INTEGER %FN %SPEC RENAME(%BYTEINTEGERARRAYNAME FROM, TO)

This function renames the file.'from' to the file 'TO'. The result is zero if the rename succeeds, non-zero if 'FROM' doesn't exist or 'TO' does exist.

The ARRAYS should be declared :-

%BYTEINTEGERARRAY <name> (-1:6)

The first two elements are used by 'RENAME', the file name is in elements one to six.

APPENDIX 2

PRELIMINARY USER MANUAL FOR THE SECOND SYSTEM

USER MANUAL FOR DEIMOS

B. GILMORE
AUGUST 1976

CONTENTS

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 '>' and '<' overlaid. If a program is running, the CLI can be invoked by typing <escape>, the command prompt should then appear.

The input to the CLI has two forms, either:-
A) A FILE NAME, followed by 'stream definitions' or
B) A COMMAND VERB, possibly followed by parameters

A) A 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 does not contain a program, the message '*NO FRED', where FRED is the name of the file, is output and the command prompt is re-issued.

The 'stream definitions' are input in the form:-

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

<input 1> etc, each represent a 'stream definition'.

A 'stream definition' is of the form:-

.TT - which is either input or output from a terminal.
.LP - output to a line printer
<any other devices on the system, specified in the same way.>
<a FILE DEFINITION>

A FILE DEFINITION is in the form:-

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

The <unit number>, currently, is either '0' or '1' and refers to the physical disc drive. If neither is specified, '0' is assumed.
The <file name> consists of an alphanumeric string, which must be preceeded by a letter, of up to 6 characters in length
The <file system number> is the file system that the file comes from, or is created in. If none is specified, the users own is used (see the command LOGON) eg

FRED - the file called FRED on unit 0 in the users file system.
0.FRED - the same.
1.FRED - The file FRED is taken from disc unit one.
FRED(0) - the file FRED from file sys. zero (the systems own one)

1.FRED(25)- the file FRED from disc unit one and file system 25 (octal).

.TT - the terminal, input if on the left hand side, output if on the right hand side.

The stream definitions are linked into the input and output streams available to an IMP program, for example, if the program FRED is run with the following command:-

FRED F1,F2/O1,O2,.TT

then input stream one, used by calling 'SELECT INPUT(1)' is mapped to the file 'F1'. A 'SELECT OUTPUT(2)' will send output to the file O2', and 'SELECT OUTPUT(3)' will send the output to the terminal.

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

Input stream zero and output stream zero are always mapped to the users terminal.

FURTHER EXAMPLES

PROG .TT	- uses one input stream '.TT', no output stream (apart from zero) is defined.
PROG FILE	- will read from a file called FILE on stream one.
PROG /FILE	- no input streams defined (apart from zero), will write to a file called FILE after a 'SELECT OUTPUT(1)'.
PROG FILE/FILE	- this 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)', where a file is mapped to that stream, the system will create a temporary file (FILE# in the previous example). When the program stops, or calls 'CLOSE OUTPUT', the original file (FILE) is destroyed and the temporary file (FILE#) RENAMED to that name (FILE).

B) A COMMAND VERB

There are currently six available system commands, they are:-

a) LOGON - used to 'logon' to a file system.
b) TASKS - List the running programs on the system.
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.

A) LOGON

This command is called in the form:

LOGON nn

where 'nn' is a two digit octal number, representing the users 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:

FREE xxxx

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

E) KILL

This command is called in the form:

KILL xxxx

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

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 a program from the system that has crashed with an 'ADDRESS ERROR' or a 'SEG FAULT'.

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

THE EDITOR

The EDITOR is a PDP11 version of the COMPATIBLE CONTEXT EDITOR.

For general information on the Editor, see the USER GUIDE of the COMPATIBLE CONTEXT EDITOR by H Dewar.

The command for calling the Editor has three 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.
```

NOTES

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

2) 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.

The IMP compiler is a three pass compiler to which 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: 'L' can be either 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 the defaults, see the section on the LINKER.

NOTES

1) At present, the program 'IMPX' must be run to avoid the linking phase for %EXTERNALROUTINE files. In this case, the file 'B' above is the compilers third pass output.
2) The test compiler is accessed by running the program 'NIMP'.
3) The first pass automatically uses a file of '%SPECS' called 'PRIMS' which is taken from the system file system.
4) The first pass creates a temporary output file '0' for use with the econd pass. The second pass creates '02' and '03' and the third pass '0Y'.
5) 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.
   If the linker is to be run as a seperate 4th pass, the file '0Y' is used as its input.

The LINKER is normally run automatically as a fourth pass to the IMP compiler. It is run individually by typing one of the two following forms of command to the COMMAND LANGUAGE INTERPRETER.

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

Both of the above commands take the file 'A', which must be the output from the third pass of the compiler and create a runnable file 'B'.

The second form of the command overrides the standard LINKER defaults, the LINKER prompts for the new data as follows (defaults in brackets):-

NAME: - up to 4 characters are typed in as the TASK NAME for the program (the first four characters of the object file name are used).

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

STREAMS: - the max number of input and output streams that the program is going to use, excluding INPUT and OUTPUT (0) (default: 3).

If a program has external references in it, the LINKER will first attempt to satisfy then using 'LIB000' in the users file system. If there are still unsatisfied references, the LINKER will look at 'LIB000' in file system zero (the system file system). For more information on LIBRARIES, see the section on them.

A sample LINKER output map is given below:-

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

The '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 beneath that is the 'external definitions', in this case entry points that 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 that contains an entry point that has already been loaded, the message '*DOUBLE DEF' is output and the LINKER stops.

When the LINKER finds an EXTERNAL REFERENCE in a program file it will attempt to satisfy the reference by:-

1) Searching the library 'LIB000' in the USER file system, loading object files as necessary, then
2) Searching the library 'LIB000' in the SYSTEM file system (zero).

There are three programs currently available to the user to manipulate libraries, they are:-

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 the 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 destroyrd.

2) INSERT

There are two main forms of this command as follows:

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

The first command will add the entries of the file called TESTY into the library called LIB.

The second command will add the contents of the file called TESTY into the library called 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 file 'LIB'

OBJECT: DBUG

This program is used as an aid to debugging programs, it will mormally be 'linked' to a running program on the system using the command 'T' (for details see later), all accesses to locations will then be made in that programs virtual memory.
It may be used to:-

1) SET and CLEAR breakpoints
2) Dump out the PSECT, REGISTERS and/or the IMP STACK.
3) EXAMINE and CHANGE locations in core.
4) DUMP general areas of core.

DEBUG indicates its readiness to accept commands by typing 'DEBUG:'. The following commands may be used.

T - Set Task number of program to be debugged.
B - Set breakpoint.
C - Clear breakpoint.
N - Set a new program code base.
R - Dump the REGISTERS.
P - Dump the PSECT.
I - Dump the IMP stack.
A - Do 'P', 'R', and 'I'.
D - Dump an area of core.
O - Change the output device.
W - WAIT DEBUG.
S - Stop DEBUG.
? - Print options.

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

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

B - SET BREAKPOINT

The prompt 'ADDR:' is output, the reply is the RELATIVE address (wrt the start of the program) of the desired breakpoint. Debug will remember the contents and place a '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' is typed if ALL the breakpoints are to be cleared. If the specific breakpoint has not been set, the message '?' is output. The original contents are replaced.

N - SET A NEW PROGRAM CODE BASE

Prompts 'NEW PROGRAM CODE:'. Reply 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 effect any previously set breakpoints.

R - PRINT REGISTERS.

This command prints the registers of the nominated task. The LOCAL NAME BASE (LNB) for the outer level is also printed.

P - PRINT THE PSECT

The PSECT (own system variables) of the nominated task is printed.
The command :-

1) Prints the name of the task
2) Prints the state of the task.
3) Indicates whether there are messages queued.
4) Calls 'REGISTERS' to dump the registers out.
5) Prints the contents of the segmentation registers in the form:-

<REGISTER NUMBER> <ADDRESS> <LENGTH> <ACCESS>

I - PRINT THE IMP STACK

The entire IMP STACK is printed on the output device.
NOTE: The GLA of the program is held at the low address end of the stack, the IMP stack (ie the SP stack) starts at the high address end and works towards the low address end. is not the same as the start of the stack.

A - PRINT ALL

The command executes a print psect (P), print registers (R) and the IMP stack (I). This command is useful for dumping all the relevant information about a program onto the line printer (see command 'O').

D - DUMP AN AREA OF CORE

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

NOTE: 1) On all the dumps 'ZEROES' is ouput if one or more entire line contains zero.
2) An ISO equivalent is printed on the RHS of the dump.

O - CHANGE THE OUTPUT DEVICE

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

DBUG /L or
DBUG /.LP etc

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

W - WAIT DEBUG

This command is used to suspend DEBUG if it is necessary to input to a program on the same teletype. It is restarted by (ESCAPE) FREE DBUG

S - STOP DEBUG

Debug halts.
Note: all breakpoints are cleared.

IMPLIED COMMAND TO EXAMINE/CHANGE CORE ADDRESSES

This command accepts the following instructions:-
Note '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 STK base).
N+RM	: prints contents of (N+REGISTER M)
N(+ options)=m	: puts M into N

eg: DEBUG:100 (cr) - will print contents of locn 100.
DEBUG:100+R5 (cr) - will print contents of 100 on from reg. 5.
DEBUG:100+R5=200 (cr) - ditto, except plants 200.
NOTES

An '*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 (-2 if '-' is used).

APPENDIX 3

LISTING OF THE KERNEL OF THE SECOND SYSTEM

```
CONTROL K'100001';     ! 'SYSTEM' ROUTINE ENTRY+MUL+TRUSTED

PERMROUTINESPEC SVC
PERMINTEGERMAPSPEC INTEGER(INTEGER X); ! USED IN INIT
RECORDFORMAT DUMMY(INTEGER X)
CONSTRECORD (DUMMY) NAME NULL=0


BEGIN
      CONSTINTEGER TASK LOW LIMIT=30
      CONSTINTEGER TASK LIMIT=50
      CONSTINTEGER FREE CELLS=50
      CONSTINTEGER NO OF SERVICES=50
      CONSTINTEGER NO OF INTS=10
      CONSTINTEGER FRAG NO=30
      CONSTINTEGER PSECT LENGTH=47
      CONSTINTEGER SVC LIMIT=16
      CONSTINTEGER INT LIMIT=-7
      CONSTINTEGER K SEG LIMIT=50

      CONSTINTEGER TTID=30;            ! TASK LO LIMIT
      CONSTINTEGER DKID=31;            !   "   "    " +1
      CONSTINTEGER DIRID=32;           !   "   "    " +2
      CONSTINTEGER LOADID=33;          !   "   "    " +2
      CONSTINTEGER MOTHER=34;          !   "   "    " +3

      CONSTINTEGERNAME PS=K'177776'; ! STATUS WORD
      CONSTINTEGERNAME STACK LIMIT=K'177774'

      RECORDFORMAT EF(RECORD (EF) NAME LINK, INTEGER ID, A1)
      RECORDFORMAT QF(RECORD (EF) NAME E)
      RECORDFORMAT TF(RECORD (TF) NAME LINK, INTEGER ID, T)

      RECORDFORMAT KSEGF(INTEGER USE, DADD, PAR, PDR)
      RECORDFORMAT KSEGLF(RECORD (KSEGLF) NAME L, INTEGER B, C, D)

      RECORDFORMAT UREGSF(INTEGER R0, R1, R2, R3, R4, R5, PC, C
                                  PS, SP)
      RECORDFORMAT SEGF(INTEGER PAR, PDR, RECORD (KSEGF) NAME KSL, C
        INTEGER USE)
      RECORDFORMAT PSECTF(BYTEINTEGER ID, STATE,  C
        BYTEINTEGERARRAY NAME(0:3), C
        BYTEINTEGER PRIO, RECORD (QF) POFFQ, C
        RECORD (UREGSF) URS, INTEGER TRAPV, C
        RECORD (SEGF) ARRAY SEG(0:7))
      RECORDFORMAT PSTF(RECORD (PSECTF) NAME P)

      RECORDFORMAT PF(BYTEINTEGER SERVICE, REPLY, C
        INTEGER A1, A2, A3)
      RECORDFORMAT P2F(INTEGER D, A1, A2, A3)
```

```
RECORDFORMAT MAINPF(RECORD (MAINPF) NAME L, RECORD (P2F) P)

RECORDFORMAT STOREF(INTEGER LEN, BLOCK NO)
RECORDFORMAT ADDRFN(RECORD (ADDRFN) NAME PSECTA, LAST32, COREA)
CONSTRECORD (ADDRFN) NAME ADDS=K'120'

RECORDFORMAT D1F(INTEGER X)
RECORDFORMAT D2F(RECORD (QF) NAME X)



RECORD (EF) NAME E
RECORD (TF) NAME T, T2, TN, TB
RECORD (PSECTF) NAME PSECT, PSECT2, PSECTN, PSECT3
RECORD (SEGF) NAME SEG1, SEG2
RECORD (KSEGF) NAME KS1, KS2
RECORD (KSEGLF) NAME KL
RECORD (KSEGLF) NAME FREE SEGL

RECORD (QF) ARRAY CPUQ(0:7)
!*

RECORD (PF) PX
RECORD (PF) NAME P, Q
RECORD (P2F) NAME P2, Q2
RECORD (MAINPF) NAME MAINP, MP2
RECORD (QF) NAME FREE PARAM
RECORD (QF) TIME Q;              ! HEAD OF TIMER LIST

INTEGER QU, SERVICE, TICKS, LEN, I, PT, L2, BLOCK, S, ID

RECORD (D1F) NAME D1
RECORD (D2F) D2


RECORD (PSTF) ARRAY PSECTA(TASK LOW LIMIT:TASK LIMIT)
RECORD (EF) ARRAY ONQ(TASK LOW LIMIT:TASK LIMIT)
RECORD (TF) ARRAY ONTMQ(TASK LOW LIMIT:TASK LIMIT)

RECORD (MAINPF) ARRAY PARAMS(0:FREE CELLS)

RECORD (STOREF) ARRAY STORE(0:FRAG NO)
RECORD (STOREF) NAME ST1

RECORD (KSEGLF) ARRAY KSEGL(1:K SEG LIMIT)

RECORD (P2F) ARRAY LAST THIRTY2(0:31);  OWNINTEGER LAST=0

OWNBYTEINTEGERARRAY SER MAP(INT LIMIT:NO OF SERVICES)= C
   0, 0, 0, MOTHER, DKID, TTID, TTID, 0,
    TTID, 0, DKID, DIRID, LOADID, 0, MOTHER, 0(43)

CONSTINTEGER FAULT SER=-4
```

```
!!    TU 16 INT = -5
!!    DQS11 TX INT = -6
!!    DQS11 RX INT = -7



EXTERNALINTEGERFNSPEC RUN(RECORD (PSECTF) NAME PSECT)
ROUTINESPEC INITIALISE
ROUTINESPEC FILL SEG(RECORD (SEGF) NAME SEG, C
  RECORD (KSEGF) NAME KS, INTEGER PAR, PDR)
!*
ROUTINESPEC PUSH(RECORD (QF) NAME Q, RECORD (EF) NAME E)
RECORD (EF) MAPSPEC POP(RECORD (QF) NAME Q)
ROUTINESPEC SCHEDULE
ROUTINESPEC DEALLOCATE(RECORD (KSEGF) NAME KS)
ROUTINESPEC FAULT(INTEGER I)

!***********************************************
!*          SUPERVISOR STATES                  *
!***********************************************
CONSTINTEGER IDLE ST=-1
CONSTINTEGER TASK ST=0

!***********************************************
!*                TASK STATES                  *
!***********************************************
CONSTINTEGER T WAIT=1
CONSTINTEGER T POFF=2
CONSTINTEGER T TIME=4
CONSTBYTEINTEGER T CPUQ=8
CONSTBYTEINTEGER T RUN=16
CONSTBYTEINTEGER T SUSP=K'200'


!***********************************************
!*         SVC SERVICES (BY EMT VALUE)         *
!***********************************************
CONSTINTEGER INTERRUPT=-1
CONSTINTEGER WAIT=1
CONSTINTEGER PON R=2
CONSTINTEGER POFF R=3
CONSTINTEGER INSERT=4
CONSTINTEGER DELETE=5
CONSTINTEGER ALLOCATE CORE=6
CONSTINTEGER FREESP=7
CONSTINTEGER SET TIME=8
CONSTINTEGER SCHEDULE T=9
CONSTINTEGER MAP VIRT=10
CONSTINTEGER GET ABS=11
CONSTINTEGER GET ID=12
CONSTINTEGER LINKIN=13
CONSTINTEGER MAP SHARED=14
CONSTINTEGER MAP HREGS=15
```

```
CONSTINTEGER MAP PSECT=16


!***********************************************
!*          STATIC CORE LOCATIONS              *
!***********************************************
CONSTINTEGERNAME INT VALUE=K'40'
CONSTINTEGERNAME SUPER =K'42'
CONSTINTEGERNAME ALARM F=K'44'
CONSTINTEGERNAME S ENTRY=K'46'
CONSTINTEGERNAME PSECT AREA=K'50'
CONSTINTEGERNAME FAULT TYPE=K'52'

!***********************************************
SWITCH SER(-1:SVC LIMIT)

OWNINTEGERARRAY PRESET(0:234)= C
0,M'TT',M'  ',4,0,0(6),K'20210',K'140200',K'120200',0,0(32),
0,M'KD',M'  ',4,0,0(6),K'20210',K'140200',K'120200',0,0(32),
0,M'ID',M'TR',3,0,0(6),K'20210',K'140140',K'120200',0,0(32),
0,M'OL',M'DA',1,0,0(6),K'20032',K'140040',K'120300',0,0(32),
0,M'OM',M'HT',3,0,0(6),K'20032',K'140040',K'120200',0,0(32)

!***************************************************
!*             START OF CODE PROPER                *
!***************************************************

INITIALISE;                ! HELD IN DE-ALLOCATABLE SPACE


!***************************************************
!*             BASIC LOOP IS CPU SCHEDULER         *
!***************************************************

CYCLE
E==NULL
CYCLE QU = 7, -1, 0
   IF NOT CPUQ(QU)_E==NULL START
      E==POP(CPUQ(QU))
      EXIT
   FINISH
REPEAT
IF E==NULL START;          ! IDLE
   SUPER=IDLE ST;            ! MARK SUPERVISOR IN IDLE
   PS=0;                      ! SET PRIO=ZERO
IDLE LP:   *1;                  ! EXECUTE 'WAIT'
   ->IDLE LP
FINISH
!* FOUND PROCESS, SO SCHEDULE
PSECT==PSECTA(E_ID)_P;        ! MAP 'PSECT' TO ACTUAL SPACE
GO:  IF PSECT_STATE&T SUSP#0 THENCONTINUE;  ! DON'T RUN IT
SUPER=TASK ST;                ! GOING INTO 'USER' PROCESS
PSECT_STATE=T RUN
SERVICE=RUN(PSECT);            ! EXTERNAL
```

```
     ->SER(SERVICE) IF SERVICE<=SVC LIMIT
     FAULT TYPE=5
ERROR:
     INT VALUE=FAULT SER

SER(INTERRUPT):              ; ! DEVICE INTERRUPT
     IF INT VALUE#FAULT SER START
        SCHEDULE UNLESS SUPER=IDLE ST
     ELSE
        PX_A2=PSECT_ID
        PX_A3=FAULT TYPE
     FINISH
     ->CLOCKINT IF INT VALUE=0
     ID=SER MAP(INT VALUE)
     PX_SERVICE=INT VALUE
     PX_REPLY=0;                      ! INTERRUPT
     PX_A1=INT VALUE;  ! REMOVE IN DUE COURSE
     P2==PX;  P==P2
     !* AND SEND IT
     !! SEND MESS TO RELEVANT TASK
     ->DO PON

SER(WAIT):
     PSECT_STATE=T WAIT
     CONTINUE;                    ! FIND SOMETHING ELSE

SER(PON R):
     SCHEDULE
     P2==PSECT_URS;            ! MAP PARAM AREA TO HIS REGS
     P==P2
     !* NOW PLANT ON Q
     !* AND SCHEDULE PROCESS IF NECESSARY
     S=P_SERVICE;              ! PICK UP THE ROUTING
     ID=SER MAP(S);            ! AND FIND THE OWNING PROCESS
DO PON:
     PSECT3==PSECTA(ID)_P;         ! PSECT OF RECEIVING MESSAGE
     FAULT TYPE=6 AND ->ERROR IF PSECT3==NULL OR ID=0
     PSECT==PSECT3
     Q==PSECT_URS;  Q2==Q
     IF PSECT_STATE&T POFF#0 START;  ! WAITING FOR POFF
        IF Q_SERVICE=0 OR Q2_D=P2_D START
PON EXECUTE:
           Q2 == PSECT_URS
           Q2 = P2
           LAST THIRTY2(LAST)=P2;  LAST=(LAST+1)&31
           SCHEDULE
           CONTINUE
        FINISH
     FINISH
     MAINP==FREE PARAM;           ! PICK UP NEW PARAM CELL
     FREE PARAM==MAINP_L;         ! RELINK FREE LIST
     MAINP_P = P2
     PUSH(PSECT_POFFQ, MAINP);     ! PUT ON TASK POFF Q
     CONTINUE
```

```
SER(POFF R):;              ! USER POFF
     UNLESS PSECT_POFFQ_E==NULL START; ! Q NON ZERO
        MP2==PSECT_POFFQ_E;     ! GET LAST ENTRY
        Q==PSECT_URS;  Q2==Q
        UNTIL MP2==MAINP CYCLE;        ! CYCLE WHOLE Q
           MAINP==POP(PSECT_POFFQ)
           P==MAINP_P;  P2==P
           IF Q_SERVICE=0 OR Q2_D=P2_D START
              MAINP_L==FREE PARAM;  FREE PARAM==MAINP; ! RELINK ON Q
              ->PON EXECUTE
           FINISH
           PUSH(PSECT_POFFQ, MAINP)
        REPEAT
     FINISH
     PSECT_STATE=T POFF
     CONTINUE


SER(SCHEDULE T):                  ! R0 IS ID OF TASK TO BE SCHEDULED
     SCHEDULE;     ! RE-SCHEDULE CALLER
     PSECT==PSECTA(PSECT_URS_R0)_P
     FAULT(7) IF PSECT==NULL
     SCHEDULE
        CONTINUE

SER(DELETE):                        ! DELETE THE RUNNING TASK
        IF PSECT_ID=LOADID START
           SCHEDULE;     ! RE-SCHEDULE LOADER
           PSECT==PSECTA(PSECT_URS_R0)_P
        FINISH
        CYCLE;                     ! CLEAR OUT THE POFF Q
           MAINP==POP(PSECT_POFFQ)
           EXITIF MAINP==NULL
           MAINP_L==FREE PARAM;  FREE PARAM==MAINP
        REPEAT
       CYCLE I=7, -1, 0;           ! GO DOWN THE SEGS
           KS1==PSECT_SEG(I)_KSL
           UNLESS KS1 == NULL START
              KS1_USE=KS1_USE-1
              DEALLOCATE(KS1) IF KS1_USE=0
           FINISH
        REPEAT
        PSECTA(PSECT_ID)_P==NULL


     REPEAT;                       ! OF MAIN LOOP

CLOCKINT:                          ! CLOCK HAS OVERFLOWED
     !* SEND MESSAGE TO FIRST TASK ON Q
     !* SET CLOCK TO NEXT TIME
     TN==POP(TIME Q)
     UNLESS TIMEQ_E==NULL THEN ALARM F=TIMEQ_E_A1
     ID=TN_ID
     PX_SERVICE=ID;  PX_REPLY=0
```

```
      P2==PX;  TN_T=0
      ->DO PON

SER(SET TIME):                              ! SET TIMER FOR URS_R0 TICKS
      ID=PSECT_ID
      IF ONTMQ(ID)_T#0 THEN FAULT(6); ! ON Q ALREADY
      TN==ONTMQ(ID)
      TICKS=PSECT_URS_R0;           ! NO OF TICKS
      TB==TIMEQ_E;                     ! LAST ENTRY
      ->BOT IF TB==NULL
      IF TB_LINK==TB THEN T==TB ELSE T==TB_LINK
      T_T=ALARM F;        ! ADJUST FOR TIME PAST
      T2==TB
      CYCLE;                            ! CHECK THE LIST
         IF TICKS<T_T START;         ! PUT ON Q HERE
            TN_LINK==T2_LINK;  T2_LINK==TN
            T_T=T_T-TICKS
            EXIT
         FINISH
         TICKS=TICKS-T_T
         IF T==TB START;              ! AT BOTTOM
BOT:        PUSH(TIMEQ, TN);            ! PLANT ON END
            EXIT
         FINISH
         T2==T
         T==T2_LINK
      REPEAT
      TN_T=TICKS
      ALARM F = TICKS;                   ! START THE CLOCK
      ->GO;                             ! IMMEDIATE RESCHEDULE

SER(ALLOCATE CORE):
      PT= -1
      IF PSECT_ID <= LOADID START
      LEN=PSECT_URS_R0;                 ! CORE REQUIRED IN BLOCKS
      PSECT_URS_R0=0;                   ! URS_R1 IS THE NEW SEG
      PT=-1; L2=0
      CYCLE I=FRAG NO, -1, 0
         IF STORE(I)_LEN>LEN AND STORE(I)_LEN>L2 THEN C
           PT=I AND L2=STORE(I)_LEN
      REPEAT
      FINISH
      IF PT=-1 THEN ->GO;               ! NO CORE
         ST1==STORE(PT)
         BLOCK=ST1_BLOCK NO;            ! ADDRESS OF BLOCK (IN BLOCKS)
         IF L2>LEN START;             ! EXCESS, SO TRIM
            ST1_BLOCKNO=ST1_BLOCK NO+LEN
            ST1_LEN=ST1_LEN-LEN
         ELSE E=0
      KL==FREE SEGL
      FAULT(12) IF KL==NULL;            ! NO FREE SEGMENT CELLS
      FREE SEGL==KL_L
      KS1==KL;                            ! MAP THE 'REAL' TYPE ON
      KS1_USE=0;                          ! 'SHARED' WILL MAKE IT '1'
```

```
      KS1_PAR=BLOCK;  KS1_PDR=(LEN-1)<<8!6
      SEG1==PSECTN_SEG(PSECT_URS_R1)
      PSECT_URS_R0=BLOCK
      ->DO SHARED;                          ! FILL HIS SEG ENTRY

SER(MAP VIRT):                              ! MAP USER A TO B
                                             ! R0 = TARGET ID
                                             ! R1 = TARGET SEG
                                             ! R2 = CALLERS SEG
                                             ! R1 = -1 SIGNIFIES DROP SEG
      S=0
      IF PSECT_ID=LOADID THEN S=6
      SEG1==PSECT_SEG(PSECT_URS_R2);    ! GET CALLERS SEG
      IF PSECT_URS_R1<0 START;          ! DROP SEGMENT
         KS1==SEG1_KSL
         IF KS1==NULL THEN FAULT(10);   ! NO SEG
         KS1_USE=KS1_USE-1
         IF KS1_USE=0 THEN DEALLOCATE(KS1)
         SEG1=0;                          ! ZERO CALLERS ENTRY
      ELSE
         !! MAP TO DESIRED SEG
         PSECT2==PSECTA(PSECT_URS_R0)_P
         FAULT(4) IF PSECT2==NULL
         KS1==PSECT2_SEG(PSECT_URS_R1)_KSL
DO SHARED:
         FAULT(13) IF KS1==NULL
         SEG1_PAR=KS1_PAR;  SEG1_PDR=KS1_PDR!S
         SEG1_KSL==KS1
         KS1_USE=KS1_USE+1
      FINISH
      ->GO

SER(GET ABS):                               ! GET ABSOLUTE ADDR OF VIRT SEG
                                            ! R0 = TARGET ID
                                            ! R1 = TARGET SEG
                                            !  R2 = 0 (DROP SEG) = 1 (GET SEG)
      PSECT2==PSECTA(PSECT_URS_R0)_P
      FAULT(5) IF PSECT2==NULL
      SEG1==PSECT2_SEG(PSECT_URS_R1)
      KS1==SEG1_KSL
      IF PSECT_URS_R2#0 START
         PSECT_URS_R0=SEG1_PAR
         PSECT_URS_R1=SEG1_PDR
         KS1_USE=KS1_USE+1
      ELSE
         KS1_USE=KS1_USE-1
         IF KS1_USE=0 THEN DEALLOCATE(KS1)
      FINISH
      ->GO


SER(GET ID):                          ! RETURN ID OF TASK IN R0
         PSECT_URS_R0=PSECT_ID
         ->GO
```

```
SER(LINKIN):                          ! R0 IS REQUIRED SERVICE
     SER MAP(PSECT_URS_R0)=PSECT_ID
     ->GO

SER(MAP SHARED):                      ! R0 IS ID, R1=SEG, R2=SHARED NO
     PSECT2==PSECTA(PSECT_URS_R0)_P
     FAULT(8) IF PSECT2==NULL
     SEG1==PSECT2_SEG(PSECT_URS_R1)
     KS1==PSECT_SEG(1)_KSL;  S=2
     ->DO SHARED

SER(INSERT):                          ! ALLOCATE A NEW PSECT (AND MAP TO R0?)
     CYCLE ID=TASK LOW LIMIT, 1, TASK LIMIT
        EXIT IF PSECTA(ID)_P==NULL
     REPEAT

     D1==D2;                          ! DUMMY FORMATS TO STUFF ADDRESS
     D1_X=PSECT AREA+(ID-MOTHER-1)<<7
     !! SHOULD BE *(PSECT LENGTH*2)
     PSECTA(ID)_P==D2_X
     PSECT2==D2_X;                    ! MAP TO ARRAY AND PSECT2
     PSECT2_ID=ID
     SEG1==PSECT_SEG(PSECT_URS_R0); ! MAP TO LOADER PSECT
     SEG1_PAR=D1_X>>6;  SEG1_PDR=1<<8!6; ! 2 SEGS, READ/WRITE
     SER MAP(ID)=ID;                ! MAP HIS MAIN SERVICE IN
     PSECTN==PSECT2;                ! FOR USE WITH GET CORE
     ->GO;                            ! RESTART LOADER

SER(MAP HREGS):                       ! MAP HARDWARE REGS TO SEG R0
     SEG1==PSECT_SEG(PSECT_URS_R0)
        SEG1_PAR=K'7600';  SEG1_PDR=K'77406';  SEG1_KSL==NULL
        ->GO

SER(MAP PSECT):                     ! MAP PSECT 'R0' TO SEG IN R1
     I=PSECT_URS_R1
     ID=PSECT_URS_R0
     SEG1==PSECT_SEG(I);  SEG1=0
     PT=0
     IF PSECT_ID=LOADID THEN S=2<<8!6 ELSE S=2<<8!2
     D2_X==PSECTA(ID)_P
     UNLESS D2_X==NULL START
        SEG1_PAR=D1_X>>6;       ! MAP TO THE START OF ITS BLOCK
        SEG1_PDR=S;    ! ACCESS DEPENDS ON TASK
        PT=I<<13!(D1_X&K'77');  ! POINT R0 TO ITS BEGINNING
     FINISH
     PSECT_URS_R0=PT
     ->GO


     ROUTINE PUSH(RECORD (QF) NAME Q, RECORD (EF) NAME E)
        IF Q_E==NULL THEN E_LINK==E ELSESTART
           E_LINK==Q_E_LINK
           Q_E_LINK==E
```

```
    FINISH
    Q_E==E
END


RECORD (EF) MAP POP (RECORD (QF) NAME Q)
    RECORD (EF) NAME E
    IF Q_E==NULL THEN RESULT==NULL
    E==Q_E_LINK
    IF NOT E==Q_E START;          ! ONE ITEM ONLY
       Q_E_LINK==E_LINK
    ELSE
       E==Q_E
       Q_E==NULL
    FINISH
    RESULT == E
END
ROUTINE SCHEDULE
    PSECT_STATE=(PSECT_STATE&T SUSP)!T CPUQ
    PUSH(CPUQ(PSECT_PRIO), ONQ(PSECT_ID))
END

ROUTINE DEALLOCATE(RECORD (KSEGF) NAME KS)
    RECORD (STOREF) NAME S, S2, S3
    RECORD (KSEGLF) NAME KSL
    INTEGER I, BOT, BLOCK, LEN

    BLOCK=KS_PAR;  LEN=KS_PDR>>8+1
    BOT=BLOCK+LEN;  S2==NULL
    CYCLE I=FRAG NO, -1, 0
       S==STORE(I)
       IF S_BLOCK NO+S_LEN=BLOCK START
          IF S2==NULL START
             S_LEN=S_LEN+LEN;    ! ADD IT ON THE BOTTOM
             S2==S;                ! REMEMBER IT
          ELSE
             S_LEN=S_LEN+S2_LEN
             S2_BLOCK NO=0;  S2_LEN=0
             EXIT
          FINISH
       ELSE
          IF S_BLOCK NO=BOT START
             IF S2==NULL START; ! NOT FOUND THE UPPER HALF
                S_BLOCK NO=BLOCK;  S_LEN=S_LEN+LEN
                S2==S;          ! MARK FOUND
             ELSE
                 S_BLOCK NO=S2_BLOCK NO
                 S_LEN=S_LEN+S2_LEN
                 S2_BLOCK NO=0;  S2_LEN=0
                EXIT
             FINISH
          FINISH
       FINISH
       IF S_BLOCK NO=0 THEN S3==S; ! REMEMBER EMPTY SLOT
```

```
    REPEAT
    IF S2==NULL START
       S3_BLOCK NO=BLOCK;   S3_LEN=LEN
    FINISH
    KSL==KS
    KSL_L==FREE SEGL
    FREE SEGL==KSL;                  ! MAP SEG ENTRY BACK TO FREE LIST
END

ROUTINE FAULT(INTEGER I)
    *K'016500'; *2; ! MOV 2(LNB),R0
    *0
END

ROUTINE INITIALISE

    RECORDFORMAT SF(INTEGERARRAY SEG(0:7))
    CONSTRECORD (SF) NAME K PAR=K'172340'
    CONSTRECORD (SF) NAME K PDR=K'172300'
    CONSTRECORD (SF) NAME U PAR=K'177640'
    CONSTRECORD (SF) NAME U PDR=K'177600'

    CONSTINTEGERNAME SR0=K'177572'
    CONSTINTEGERNAME SR2=K'177576'

    CONSTINTEGERNAME CLOCK V=K'177546'

    RECORDFORMAT DEDLOC F(INTEGERARRAY D(0:4))
    CONSTRECORD (DEDLOCF) NAME DEDLOC=K'60000'

    INTEGER I, PT, BASE, TOP CORE, ID, TOP, PERM, PERML, STK, TC
    INTEGER KST, STKL
    RECORDFORMAT D1F(INTEGERNAME X)
    RECORDFORMAT D2F(RECORD (QF) NAME X)
    RECORD (D1F) NAME D1
    RECORD (D2F) D2

    K PAR=0; K PDR=0
    CYCLE I=0, 1, 6
       K PAR_SEG(I)=I<<7;  K PDR_SEG(I)=K'77406'
    REPEAT
    K PAR_SEG(7)=K'7600';       ! MAP TO HARDWARE VECTORS
    K PDR_SEG(7)=K'77406'
    UPAR=0;  UPDR=0
    SR0=1;                        ! GET IT GOING
    PS=K'340';                    ! ENSUE UNINTERRUPTABLE
    CLOCKV=K'100'
    !! STACK LIMIT=K'400'


    D1==D2
    KST=2

    CPUQ(I)=0 FOR I=0, 1, 7
```

```
LAST  THIRTY2(I)=0 FOR I=0,1,31
SUPER=IDLE ST
ALARM F=0
FOR I=TASK LOW LIMIT, 1, TASK LIMIT CYCLE
   PSECTA(I)_P==NULL
   ONQ(I)=0;  ONQ(I)_ID=I
   ONTMQ(I)=0;  ONTMQ(I)_ID=I
REPEAT
ID=TASK LOW LIMIT
PERM=DEDLOC_D(3)>>6;    ! PICKUP ADDR OF PERM
TOP=DEDLOC_D(5)>>6
PERML=((TOP-PERM-1)<<8)&K'177400'!2;  ! READ ONLY
TOP CORE=DEDLOC_D(0);  TC=TOP CORE
CYCLE I=TOP CORE, 2, K'137336'
    IF I=K'60000' THEN I=K'60060'
   INTEGER(I)=0
REPEAT
PSECT AREA=TOP CORE
TOP CORE=TOP CORE+K'200'*10; !  ONLY 10 TASKS?????
I=5
CYCLE
   EXITIF DEDLOC_D(I)=TC
   D1_X==PRESET(PT)
   PSECTA(ID)_P==D2_X
   PSECT==D2_X
   PSECT_ID=ID
   PT=PT+PSECT LENGTH
   BASE=DEDLOC_D(I)>>6
   STK=DEDLOC_D(I+1)>>6
   TOP=DEDLOC_D(I+2)>>6
   FILL SEG(PSECT_SEG(2), KSEGL(KST), BASE, (STK-BASE-1)<<8!6)
   KST = KST+1;                ! FILL IN CODE ADDRESS
   FILL SEG(PSECT_SEG(6), KSEGL(KST), STK, (TOP-STK-1)<<8!6)
   KST = KST+1;                ! FILL IN STACK AREA
   PSECT_SEG(7)_PAR=K'7600'
   PSECT_SEG(7)_PDR=K'77406'
   SER MAP(ID)=ID
   PSECT_SEG(1)_PAR=PERM;  PSECT_SEG(1)_PDR=PERML
   PSECT_SEG(1)_KSL==KSEGL(1)
   PSECT_URS_R1=K'140000';    ! VIRT ADD OF TOP OF STK
   PSECT_URS_SP=K'140000'+(TOP-STK)<<6
   IF ID>=LOADID START
      FILL SEG(PSECT_SEG(7), KSEGL(KST), C
         TOP CORE>>6, K'13'<<8!6);      ! FILL IN I/O SEG
      TOP CORE=TOP CORE+K'1400';  KST=KST+1
      PSECT_URS_R2=2;          ! INDICATE LOADER TO PERM
      INTEGER(STK<<6+4)=X'0A41';    ! A,NL
      PSECT_URS_R0=K'140004'; ! POINTS TO ABOVE
   FINISH
   SCHEDULE
   I=I+2;  ID=ID+1
REPEAT
I=TOP CORE>>6
STORE(0)_BLOCK NO=I
```

```
        STORE(0)_LEN=K'2000'-I
        ADDS_PSECTA==PSECTA(TASK LOW LIMIT)
        ADDS_LAST32==LAST THIRTY2(0)
        ADDS_COREA==STORE(0)
        PARAMS(I)_L==PARAMS(I+1) FOR I=0, 1, FREE CELLS-1
        FREE PARAM==PARAMS(0)
        CYCLE I=KST, 1, K SEG LIMIT-1
           KSEGL(I)=0
           KSEGL(I)_L==KSEGL(I+1)
        REPEAT
        FREE SEGL==KSEGL(KST)
        KS1==KSEGL(1)
        KS1_PAR=PERM;  KS1_PDR=PERML
        KS1_USE = 10
        K PDR_SEG(I)=0 FOR I=3, 1, 6
     END
     ROUTINE FILL SEG(RECORD (SEGF) NAME SEG, RECORD (KSEGF) NAME KS, C
       INTEGER PAR, PDR)

        SEG_PAR=PAR;  SEG_PDR=PDR; SEG_KSL==KS
        KS_USE=1
        KS_PAR=PAR;  KS_PDR=PDR
        KS_DADD=0
     END
ENDOFPROGRAM
```

REFERENCES

1 F. J. Corbato, C. T. Clingen, J. H. Salter, 'MULTICS, The First Seven Years', Proceedings of the 1972 AFIPS SJCC.

2 Software Engineering, Ed. P. Naur and B. Randell, 1969.

3 W. F. C. Purser, 'The Design of a Real-time Operating System for a Minicomputer', Software Practice and Experience, 5, No 2, 147-167 (1975)

4 F. P. Brooks, 'Mythical Man Month'

5 F. J. Corbato, 'PL/1 as a Tool for Systems Programming', Datamation 15 No. 6, 68-76 (1969)

6 E. I. Organick, 'Computer System Organisation'

7 C. Adams, 'Architecture and Performance Evaluation of EMAS', (To apear in Seminars on Modelling and Performance Evaluation at I.R.E.I.A. 1975)

8 Ferranti, 'Argus 700 Reference Manual'

9 P. Robertson, 'On the Production of Optimised Code from a Transportable Compiler for High Level Languages', (a Ph.D. Thesis to be published).

10 C. Adams and G. E. Millard, 'Performance Measurment on the Edinburgh Multi Access System', Proceedings International Computing Symposium June 1975.

11 P. D. Stephens, 'The IMP Language and Compiler', Computer Journal Vol 17 No 3.

12 H. Whitfield, A. S. Wight, 'EMAS - The Edinburgh Multi Access System', Computer Journal Vol 16 No 4.

13 D. J. Rees, 'The EMAS Director', Computer Journal Vol 18 No 2.

14 PDP 11/40 Handbook, Digital Equipment Corporation.

15 D. L. Mills, 'Proposal for a Multi Programming System for a PDP 11', (June 1971).

16 D. L. Mills, 'Multi-programming in a Small Systems Environment', University of Michigan, Technical Report 19.

17 D. M. Ritchie, K. Thompson, 'The UNIX Time Sharing System', Comm of the ACM, Symposium on Operating Systems, October 15-17 1973.

18 R. B. John, 'The design of Systems for Telecommunications between Small and Large Computers', Ph. D. thesis, University of Edinburgh 1973.

19 P. M. Woodwardet. al., 'Definition of CORAL 66', H. M. S. O..

20 I. C. L., 'System B Architecture', SID D100

21 E. W. Dijkstra, 'Co-operating Sequential Processes', Programming Languages (ed. F. Genvys) Acedemic Press 1968.

22 M. M. Barritt et. al., 'The IMP Language Manual', Edinburgh Regional Computing Centre, 1970

23 GEC-ELLIOTT Automation Ltd., Babbage User Manual 1972

24 N. Wirth, 'PL360 A Programming Language for the 360 computers', Journal of the ACM Vol 5 No 1 Jan 1968

25 H. Dewar, 'HAL 7502', Department of Computer Science, Edinburgh University, 1975

ACKNOWLEDGEMENTS

There have been too many people who have helped in one way or another and it impossible to mention them all individually. However, specific thanks are due to Dave Mills, Nick Shelness and Professor S. Michaelson who have been very patient and helpful as supervisors, also to Peter Robertson for his IMP compiler, and finally to Dr. J. G. Burns for the encouragement and help he gave to the author.