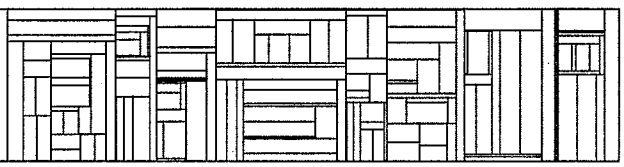
A MULTI-EMULATION SYSTEM

by

Ejvind Lynning

DAIMI PB-62
July 1976

Institute of Mathematics University of Aarhus
DEPARTMENT OF COMPUTER SCIENCE
Ny Munkegade - 8000 Aarhus C - Denmark
Phone 06-12 83 55



PREFACE

The present report is submitted as my master's thesis (speciale-opgave).

The multi-emulation project (the terminology is defined later) grew out of discussions at DAIMI about the situation which arose over the department's experimental microprogrammable computer : RIKKE-1 (and MATHILDA) in 1974: Two emulated machines, OCODE and PCODE, had been built and it was desirable to enable these machines to cooperate. More generally a supportive environment for the development of emulated machines and other microprogrammed implementations was wished for. In the autumn of 1974 the subject was discussed in a study-group where Peter Kornerup, Nigel Derrett, and Mike Manthey (among others) took part. The presently reported project came later and does not directly build on that group's discussions.

For advice on the project, and in particular for valuable criticism of this writing as it developed, thanks are due to Nigel Derrett, my thesis adviser. For careful, yet humorous drawing of the figures in this volume I thank my wife, Kirsten Hays. By typing the manuscript Karen Møller and Eva Sloth have also been of great help.

Documentation of the programs of which the described system actually consists is found in an appendix which will not be published.

Århus, June 1976, Ejvind Lynning

TABLE OF CONTENTS

I. INTRODUCTION

Recent years have witnessed an upsurge of interest in microprogramming, and a number of more or less dynamically microprogrammable computers have seen the light of day, or in many cases only the neon-lit laboratory. Even textbooks have begun to appear on the topic of microprogramming, e.g. [11], indicating that the subject has reached wide recognition as an area of interest. Originally microprogramming was introduced as an orderly way of organising the control section of an automatic computer but the recent growth of interest in the area is founded on technological advances which allow changeable microprogrammed control; this development has facilitated the exploitation of microprogramming for several purposes which go beyond the original aim of orderly hardware organisation.

Judged by a criterion of economic success a most important application of microprogramming has been the realisation of the IBM 360 line of computers on a number of different microprogrammable host processors (or CPUs). These processors were designed with the specific intention that they should serve as hosts for the 360 models. However the flexibility allowed by changeable microprogram was exploited to make these processors support also programs for older IBM products by imitation or emulation, as the technique was named, of the older models [8].

The possibility of configuring a given microprogrammable processor to appear and behave according to a given description has not only been exploited for emulation of already existing physical machines. Clearly there is no reason why the specifications to which an emulator must conform should come from a physically existing machine. Abstract machines may be defined and implemented by emulation. Machines designed to support particular high level languages are important in this category.

With the B1700 the Burroughs Corporation provides a means of emulating abstract or virtual machines in a manner claimed to be very flexible. The microprogrammable processors in the B1700 computers are not designed to support any particular "general purpose" machine (such as, e.g. the 360), but rather to allow efficient emulation of arbitrary machines [9].

Microprogramming is not only being used for emulation but also to support particular applications, e. g. graphics, or to fortify operating systems by the provision of useful operations as microprogrammed primitives [12]. For example semaphores may be supplied to an operating system in this fashion. In a more general situation systems may be provided where microprogramming is not significantly more cumbersome than ordinary "high level" programming. With such a system a programmer may choose to implement any algorithm, or time-consuming part thereof, as a microprogram.

One member of the group of fairly small microprogrammable computers which have appeared in research environments is RIKKE-1 which was designed and built at the Department of Computer Science, Aarhus University (DAIMI) during the years 1972-75. RIKKE-1 was intended to be used as a tool for emulator and processor design research. In particular RIKKE is intended to control MATHILDA (a computer of similar design, but with a 64 bit main data path as compared to RIKKE's 16 bits) in experiments with non-standard arithmetic.

Various design and microprogramming projects have taken place in connection with RIKKE-1: An emulator for PCODE, designed as a suitable target code for compilation of programs in the PASCAL language, has been implemented [17] (this uses both RIKKE and MATHILDA); an i/o-nucleus, for block transfers to and from peripheral devices, and an OCODE emulator [18] are used as the microprogrammed level of the RIKKE BCPL system, a programming environment simular to the Oxford OS-system [5]; and an interpreter, similar to Landin's SECD-machine, has been built for the purpose of evaluating LCODE expressions, which are the output of a compiler-compiler system [19].

One can foresee the development of further such special-purpose interpreters or emulators, e. g. for numerical experiments; indeed being a host for such experimental implementations is RIKKE's raison d'être. In connection with such projects there will be a need for basic support software.

If we look at the design and development of a particular emulated machine we may be influenced by traditional stand-alone machines and include in the design facilities for i/o and a filing system, in general: an operating system. Turning our attention for a moment to the IBM 360 model 30 which supports switching

of microprogram to run programs for the earlier 1401 model, we find that 1401 i/o is done not by microprogram, but by simulation within the 360 [8]. Similarly in the B1700 a master control program which runs on a particular emulated machine (although part of it is directly microprogrammed) manages system resources for all existing virtual machines. We may raise a more general question as to whether programs already developed to run on some well-established virtual machine can be used to serve in the development of new emulators and programs for them.

Consider now experimental applications of microprogramming such as they may be expected on RIKKE, e.g. language machines, list processing, or matrix operations. A number of problems which arise in connection with such applications are of no real interest to the experimenter. Primarily he does not want to have to build an operating system, i.e. filing system and i/o programs, for each microprogramming project. He wants utility programs to be available and easy to call. But it does not suffice to provide service programs once and for all. It may also be very desirable to be able to combine micro- and high level programs in a flexible fashion. For example some applications of the kinds alluded to above may involve both pre- and post-processing of the data structures which are transformed by the central algorithm. In so far as these tasks are not of central interest to the experimenter he will wish to get them done as easily as possible, typically by writing programs in a high level language. By providing tools which allow such flexibility we may help the experimental worker to focus attention on his real problem.

Motivated by these considerations we have established as the general objective of the project described in this report: to allow the creation of sets of co-operating programs, which may run on different virtual processors, or some of which may indeed be written in microcode. The organisation of the report is then as follows.

Chapter II is intended to clarify, define, and delimit the problem.
Chapter III is a description of the system which has been designed. This description has been written with the aim of explaining design choices and of providing a complete presentation of the designed system, leaving no more loose ends than are actually present in the system. A fair number of details are therefore included, although the purely technical details which would

only interest a possible user of the designed system are postponed until Chapter VI, the user manual.

There are not many generally known computers systems which support multiple emulators. As mentioned above, some of the IBM 360 models may be switched to act as older IBM machines, and the B1700 supports user written emulators. In Chapter IV, our proposed system is compared to these systems in a discussion of other possible models for allowing communication between programs running on different machines emulated on the same physical processor. Also a paper by Tafvelin [7], describing a system very similar to the one proposed in this report, is discussed; this paper is also referred to in Chapter III, as some of the facilities described there are strongly inspired by Tafvelin.

The problem of designing suitable basic software to support microprogramming experiments on RIKKE has been discussed at DAIMI since the summer of 1974 by several people, each participant in the discussions having his own understanding of the problem, and each developing his own model for a solution. This report is not a history of that debate, but in Chapter IV, some of the important concepts which the discussions have centered on are mentioned, and possible extensions to the proposed design are suggested in chapter V.

One comment should be made here with respect to general models and possible extensions. Two approaches to problem solution and system design may be contrasted. In one, a concrete problem is taken as a starting point, an effort is made to understand that problem and its implications, and then some solution is devised, incorporating the structures deemed necessary. In the other approach the researcher first sits comfortably back and meditates, trying to identify what may naturally be seen as a more general problem of which the concrete problem at hand may be considered a special case. Then the general problem is attacked, whereby one not only solves the motivating problem, but also creates stronger tools, possibly solving the problems which might pop up three years hence. The latter approach may be called academic; it may result in no concrete achievements if the general problems are too hard to grasp. The effort reported here has taken place under timing constraints, and it has been influenced by a desire to reach concrete results. Therefore the philosophy of the former of the two appraoches has been dominant. However

a striving to provide as flexible and general a design as possible within the limitations of time has hopefully influenced the design and also the present discussion.

The final chapter of the report contains a collection of specifications of the proposed system. The chapter is intended as a handbook for the designer of emulators or other micrpprograms to fit with the system, and for the compiler writer, who must think about building code segments to be executed by such emulators.

II. THE PROBLEM OF MULTI-EMULATION

In order to define more precisely the problem which was introduced in the preceding chapter, and which is attacked throughout this report, it is useful first to define or discuss some relevant terms. The clarification which is hopefully attained by this discussion, will be of value through all the following chapters.

Emulator

In the IBM view as put forward by Tucker [10], an emulator is a combination of hardware and software additions to a given machine which enables it to act as if it were another machine. This view is tied to a concept of machine as a physical entity. A different understanding, taking into consideration the fact that machines may be purely abstract, is reflected in R. Rosin's definition: "We use the term emulator to describe a complete set of microprograms which, when embedded in a control store, define a machine". In this view, the 360 machines are themselves emulated as well as the e.g. 7090 models supported by the same microprogrammed processors. Our use of the term agrees with Rosin's definition; indeed by an emulator we will always understand a microprogram (or collection of microprograms, depending on the unit to which one assigns the name microprogram) whereby some processor is realised.

Micro-procedure

We are also interested in microprograms which do not emulate any processor. Some times we will forget this and merely speak of emulators, because these are the kind of microprograms we expect to see most of. However, when we wish to stress that a given microprogram is not an emulator, in fact that it is a direct implementation of an algorithm which is not interpretative, we use the term "micro-procedure". This term is chosen because it is not used elsewhere.

Virtual processor

A machine, defined in terms of some instruction set and possibly other architectural characteristics, is called a "virtual processor" or sometimes a "virtual machine". These terms may be seen as opposed to "physical" or "host processor", used for the actual hardware, that part of a computer which is not changeable by a process of programming. The design of a virtual machine

will often include a set of registers, the values in which together define the processor's state at a given time. The collection of values in such registers will be referred to as the processor's state vector.

Interpreter

A virtual machine may be realised by emulation. But in general it may also be implemented by means of interpretation or simulation, achieved by programming in a high level language [13]. In fact we will later consider a concept of interpreter which includes an emulator as a special case. An interpreter is any program which by interpretation of code realises a virtual processor.

Transfer of control

In a simple one-level machine the flow of program control is represented by the changing state of the instruction pointer. When a sequence of instructions is carried out, the instruction pointer is merely incremented, jumps cause more drastic changes, and routine call mechanisms may be implemented using a stack, so that instruction pointer values are saved away and later restored. In an emulated machine control moves at two levels. The microinstruction pointer is associated with execution of the emulator microprogram, but there will also be an emulator-accessible register which functions as an instruction pointer in the emulated machine; it points into the interpreted code. From the user or outside observer point of view the latter is the "real" instruction pointer. However, in connection with micro-procedures the micro-instruction pointer is the real instruction pointer. Rather than pursuing a discussion of this distinction further, we merely state that when control at the micro-level passes from one emulator to another (or to a micro-procedure) we will speak of an "external transfer of control". Thus an external control transfer occurs e.g. when a PASCAL program which runs on the PCODE machine calls a BCPL program on the OCODE machine. Notice that there is a connection with the traditional use of the term external subroutine or entry point, namely the problem of linking; it must be possible to identify an external entry point.

Multi-emulation

A system which supports external control transfers (i.e. across emulator boundaries) in a general fashion will be called a "multi-emulation" system, and the activity within it "multi-emulation".

The following sections expand on those aspects of the problem of multi-emulation which have influenced the design of the proposed multi-emulation system for RIKKE-1.

1. Resources

When a virtual (target) machine is emulated on some physical (host) machine, the various constituent parts of the target machine: registers; storage; i/o-devices; and functional units, are mapped onto parts of the host machine. We may view the host machine as a collection of physical resources to be used in this mapping, and indeed the suitability of these resources with respect to the particular mapping determines the success of the emulation. As long as one needs to consider only one emulator the problem of finding such resources remains relatively simple, although possibly not solvable. However when several emulators must co-exist in the physical host a competition begins for resources such as host machine registers, control store (microprogram store), main storage, and perhaps i/o devices; a competition which becomes fiercer with more parallelism in emulator execution.

The problem presents itself in different ways for the different kinds of resources. Peripheral devices need only be multiplexed in a multi-tasking or multi-programming environment, and only if multi-emulation is achieved through some form of multiprogramming do the problems that arise with peripheral devices have to be taken into consideration. We will ignore them here.

The registers of virtual machines, making up their state vectors, may be mapped onto host registers, but can be saved in main storage when a particular emulator is not active, and thus the problem of sharing registers can be reduced to the problem of sharing main storage. A special problem arises with "blind-alley" control registers, i.e. registers which cannot be read on to the main data bus, a type which abounds on RIKKE-1. An emulator which uses these registers must initialize them properly each time it receives control, so that they can be kept out of what is intrinsically the processor state vector; alternatively such registers may, when they are not contested, be granted wholly to a particular emulator.

Control store, if it is not large enough to contain all emulators in the system, and if there are no hardware aids to achieve overlaying by some type of virtual addressing, must be loaded with each emulator before it assumes control.

This again reduces to a problem of sharing main store.

Direct access storage, or main store for short,is the remaining trouble spot. Main store will be used to map virtual processor storage, and to save processor state vectors and emulator control store images as suggested above. Thus each emulator will make demands on physical storage in terms of one or more blocks of contiguous main store words. These vectors may be of various lengths. The problem consists of managing the division of storage into such vectors, protecting one emulator's store from erratic behaviour of other emulators, while at the same time providing orderly access for purposes of sharing information.

Disk storage is not considered a scarce resource in this context. However, it is highly desirable to use common formats if not for the data contents of disk pages then at least for the system information which binds disk pages together, and it is necessary to have a common administration of free disk space. Indeed it is difficult to see how a multi-emulation system could use a disk sensibly without a common filing system.

2. Communication

Two typical uses of external control transfers would be:

1) a call from a routine running on some user-implemented emulator to an i/o-routine in an operating system; and

2) the pre-processing in an emulator-supported high level language for a microprogrammed list-processing algorithm, i.e. the building of a list representation in some data area.

These applications suggest the need for parameter passing mechanisms and a scheme for orderly access from different emulators to a shared storage area.

Conversely, a general facility for passing parameters and results, in particular general addresses, and an access scheme for shared areas of information, makes possible both the general invocation of services across emulator boundaries and the execution of algorithms which are implemented with different passes running on different emulators.

Information which is to be understood by several emulators must necessarily be represented in a format acceptable to all; thus it may be useful to specify conventions for representation of the most important data types. This is not a logical necessity; it suffices that in any particular communication situation, some rule is adhered to by the parties involved, but common conventions provide a solution once and for all and also improve prospects for compatibility, e.g. if a given algorithm is programmed to run in different versions on various emulators then such conventions would allow the same call and return sequence to be used in other programs making use of this algorithm.

3. External References

We have already mentioned the problem of allowing one emulator to have access to data in the store "belonging" to another emulator. Another similar situation is the referencing of external code. What information is necessary within a program running on one emulator to reference and call a program on another? The provision of simple means of identifying and transferring control to external programs is an important part of the design of a multi-emulation system.

Since different emulators will assume different code and data formats, loading and linking programs becomes a problem. It would seem to be very complicated to construct a general loader or linkage editor which would satisfy external references for routines loaded into code areas of greatly varying formats, indeed such a linking program would be dependent on the set of emulators available at any given time. Again some conventions must be imposed to overcome this problem. A common scheme, which can work with all emulators, for loading programs and resolving links must be included in the design, and the demands placed by this scheme on code formats must be respected by all for the sake of compability.

III. THE RIKKE MULTI-EMULATION SYSTEM

This chapter contains a description of a multi-emulation system which has been designed to provide solutions to the problems posed in the previous chapter. The system includes the following features: a segmented virtual memory with mechanisms for dynamic resolution of external code and data references; an integrated disk organisation for files and segments; primitives to perform external control transfers in the form of subroutine calls and returns; and a stack with associated operations to hold saved virtual processor state vectors and parameters for external subroutine calls. The subroutine call and return discipline employed for external control transfers implies that the system is limited to handling one process at a time. This restriction will be discussed later in connection with a treatment of other models for multi-emulation. For the moment our interest centers on a description of the system.

The system consists both of micro-programmed modules and of modules programmed in BCPL. The micro-programmed ones are those which must be callable at the micro-level, or which operate on hardware resources directly. The BCPL modules are the more complicated ones, either algorithmically or with respect to data structures; they constitute by far the largest part of the system. In other words only what had to be micro-programmed was microprogrammed. BCPL was chosen partly because it is well suited for the kind of programming in question, but mainly because it was available.

It should be noted that at the time of this writing the proposed system has been designed in detail, and the program modules both in BCPL and RIKKE microcode have been written. But the programs have not been tested, let alone run in production. Therefore we speak of a proposal rather than of an established, completely implemented system.

1. The Virtual Memory

In order to provide a facility for orderly sharing of storage among emulators, a virtual memory system has been implemented. Instead of physical storage blocks, logical segments are allocated as units of memory space.

A segment is simply a vector of (16 bit) words. Access to a particular word in virtual memory goes through the system microprograms READ and WRITE, both of which work with virtual addresses of the form <segment number, word number>.

Segments reside on disk, but are accessed using copies in core storage units, which are of variable length, so as to fit the segments they hold. The implementation of segments does not employ paging. As the amount of core storage available is fairly large, considering expected demands, the problem is not one of efficient use of a scarce resource, but rather one of organising a common resource. The extra level of indirection incurred on storage references in a paging scheme is deemed to cost much more in execution time than could be gained by a better use of physical memory.

A segment fault, i.e. a reference to a segment which is absent from physical store causes the READ or WRITE micro-program, (whichever is in question) to invoke the segment fault handler, which is programmed in BCPL. Its task is to manage the actual storage devices available: the 32K 16 bit RIKKE main store; and the 32K 64 bit wide memory, which is connected to both RIKKE and MATHILDA. Each word of the wide memory is for the purposes of this scheme treated as four 16 bit words, so that all together, there are 160K words of 16 bit memory available in the system.

A very simple algorithm is used to find room for a faulting segment. It uses two lists of storage unit descriptors (one for each of the physical storage devices) which together completely describe how memory is utilised at any given time. The algorithm is in three stages: first it searches for a sufficiently large unoccupied storage unit; then, if sufficient storage is available but scattered, the data in the relevant store is compacted, using a microprogrammed block move; third, if not enough free storage is available, as many occupied segments as necessary are swapped out to disk, beginning with the oldest one. (We do not swap out the least recently used, but the least recently allocated - the one which has occupied physical storage for the longest period of time.)

This algorithm may not be satisfactory in a situation where storage demands outmatch supply, but it is simple, and a situation of scarcity is not expected. Certainly it would be wasteful to expend much effort on optimising this algorithm before collecting performance data, particularly since it may be tuned at any time. However the implementation of a least recently used or working set algorithm would involve changes to READ and WRITE, in order to record all references, or it would involve ensuring that working sets can be easily recognized from the state vectors of virtual processors, a very difficult requirement in a generally unstructured situation. In the adopted implementation we must impose a limit on segment size which ensures that some small number of segments may be in each physical storage medium simultaneously.

The main advantage of the segmentation scheme is that it provides exactly what emulators need for storage, i.e. long vectors of storage words, in a fashion which neatly separates the stores of different emulators (since READ and WRITE take care of checking the validity of all storage references), while at the same time providing shared access to common data segments. There is nothing new in the idea of segmentation, the Burroughs B5500 implementation of the concept goes back as far as 1961, and Dennis' classic paper [1] appeared in 1965; but we notice a difference between the MULTICS-situation [2] (with which Dennis worked) and the RIKKE multi-emulation scene. In the former, identical processors, executing in parallel, shared physical memory and common copies of data and code areas by means of segmentation; in the latter the same sort of sharing takes place, but by different processors executing serially. The real difference between the two situations is that in the MULTICS case segmentation served multi-programming, in ours it serves multi-emulation.

In order to access a given segment it is necessary to know its segment number. All segments which have segment numbers at some given time are called known segments at that time. The collection of known segments make up the address space of a computation. Segments may be added to an address space in two different ways: either a segment may be created from scratch, or a segment which exists permanently in the fi-

ling system may be retrieved by symbolic name and associated with a segment number. The latter process is called making a segment known; it also involves inserting the pair <symbolic name, segment number> into the Known Segment Table (a concept from [2]), so that any further symbolic references to the segment from the same computation may be translated into the same segment number.

We have now seen how the address space of a computation may grow by the addition of more known segments. Eventually a computation will terminate and control return to the system at the basic level. At this time the address space is re-initialised; all segments except those belonging to the system become unknown, and a new computation may begin to build its address space. The computation associated with one such address space history is called a process; hereby we establish a one-to-one relationship between a process and an address space, similar to that which exists in the MULTICS system.

Assigning a number to a segment is always associated with the creation of a segment descriptor; in fact the purpose of a segment number is to address the segment descriptor, which contains:

1. the base address of the segment in physical store, if it is in physical store.
2. the length of the segment.
3. the address of a disk page containing the disk addresses of the actual information pages of the segment.
4. a segment type word with bits to indicate whether the segment is currently in core (segment fault bit) or only on disk, whether the segment is a main store or a wide memory segment, whether the segment may be overwritten or not (protection of pure code segments), and further information which is not of interest here.

READ and WRITE always check the validity of the supplied segment number against the number of known segments, and of the word number against the segment length; WRITE also makes sure that pure code segments are not overwirtten.

The segmentation scheme provides good protection against errors, but it is inefficient, as two references to physical store are needed for each virtual memory reference, the extra one being a segment descriptor fetch from the wide memory. To satisfy those users (emulators) who prefer speed to security (or they may be safe drivers) it is possible to lock a segment in core, thus guaranteeing that it will not be swapped out until the current process terminates, or until it is unlocked again. By using this facility an emulator may reference the segment directly, without invoking READ or WRITE. It needs to know the physical base address and length, which may be kept in virtual processor registers. Since there are no privileged micro-operations for accessing physical store on the RIKKE-1, it would even seem awkward to exclude this possibility.

There is another reason why it might be desirable for an emulator to do the job of READ and WRITE for itself, which has to do with the use of RIKKE registers. Since no facilities are included in the hardware-design of RIKKE to support segmentation, these routines must make use of the same physical registers as the emulators which call them. In particular, three of the general purpose registers connected to the main data bus are used to hold segment number, word number, and the data word to be read or written from or to virtual memory. It would be ridiculously inefficient to use the external subroutine call mechanisms for each READ or WRITE; thus a set of special conventions are needed to specify exactly which register values may be changed by READ, WRITE, and in fact by the OCODE-machine which interprets the segment fault handler. Basically this means that all values temporarily held in local and shifter registers around the main bus must be saved in working registers by an emulator before it calls READ or WRITE. An emulator which does not satisfy this criterion could still be used within the system, provided somebody locks all segments referenced by it and passes their base addresses along before it assumes control. Incidentally, this comment about READ and WRITE applies in exactly the same sense to the micro-programmed primitives LINKREF, NEXT, OUT, and ENDOF to be introduced later. Exact specification of the conventions for register allocation is a subject for the user manual.

Microprograms which have been written prior to the appearance of these specifications cannot be assumed to conform to them. It will also be very difficult to modify, say, the OCODE emulator to use READ and WRITE for its storage references. Fortunately, this is not necessary.

2. External program calls and parameter passing

In stack machines, information describing the state of a process is organised in a stack. A stack frame, or "activation record", may contain return information, implicit parameters, explicit parameters, and local variables. By imposing a subroutine call and return discipline on external control transfers, we may utilise a similar organisation for the purpose of saving return information and of passing parameters. We call this stack the "super-stack".

This term has been coined to allude to the fact that there may be several other stacks around. In particular emulated stack machines will have their own stacks sitting inside private segments. The super-stack which serves control transfers across emulator boundaries should not be confused with stacks of emulated virtual processors.

The super-stack

The super-stack is best described by a picture (Fig. 1).

Fig. 1 illustrates the situation when program A running on emulator X has been called from program B, running on emulator Y, and has started pushing parameters for a further external call. The letters C, E, P, and S represent (system) registers which point to those locations in the stack indicated by the arrows emanating from the letters. The chosen letters have no particular mnemonic significance.

(Fig. 1 - see next page)

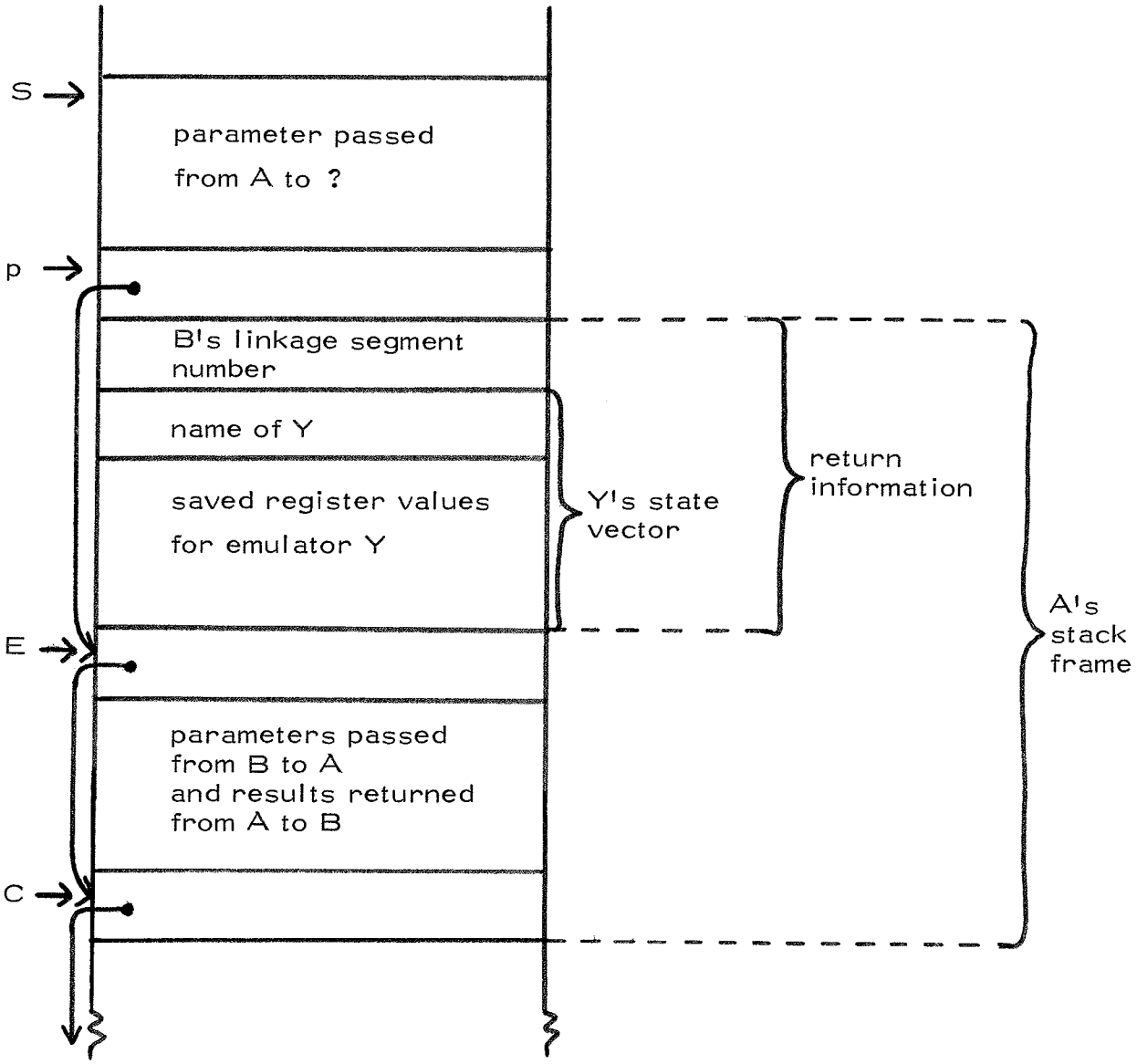


Fig. 1

As is seen, the stack contents are alternately blocks containing parameters and saved virtual processor state vectors. Notice that the last word of such a state vector must be the name of the emulator it belongs to. The area used for passing parameters in a call may be overwritten with re-

sults to be passed back. Thus communication is symmetric for call and return situations.

The primitives which access the super-stack are microprogrammed. They may be considered to be the bottom level microprograms of the system as they do not access the super-stack as a segment, but in terms of absolute addresses. This decision may be thought of as an attempt to reduce microprogram interdependence (between the super-stack primitives on one side and READ and WRITE on the other), since microprograms are complicated enough already. A description of the individual primitives follows (using the BCPL rv-notation). Figures to illustrate the use of the super-stack follow when the calling process has been more completely explained.

GetParam(i): returns rv(C+i). Used to get parameter or result.
GetRegister(i): returns rv(E+i).
Push(x): S:=S+1; rv(S):=x. Push is used to load parameters and register values onto the super-stack.
MarkStackFrame( ): Push(P); E:=S, C:=P. MarkStackFrame is used to place a link in the super-stack and update the system registers accordingly after all parameters for an external call have been pushed, but before the virtual machine registers are saved.
UnmarkStackFrame( ): S:=C; P:=C; E:=rv(C); C:=rv(E). UnmarkStackFrame is used to undo the effect of MarkStackFrame after a return has been made from an external call and the saved register values and returned results have been read from the super-stack.

Notice that all information in the super-stack is accessed as 16 bit words. The accessing primitives have no knowledge of the meaning of information stored in the stack, so it is most natural to use the unit size offered by the hardware. Any further structure to this information must be imposed by convention, as was discussed in section II.2. It would be preposterous to invent formats which have not yet been implemented in connection with projects on RIKKE. As far as systems programming is concerned there are two important data types, i.e. modes of interpreting 16 bit words or collections thereof; these are integer and character string. Conventions are defined for these two types and for booleans and single characters. Note that formats are merely defined by convention, it is up to emula-

tors or compilers to use the super-stack primitives in accordance with such formats.

To justify the choice of integers and character strings we point out that these suffice for any sort of addressing or linking in the system as any references may be reduced to strings or integers: segment numbers and word numbers are integers, symbolic names of segments and files and also of locations within segments consist of character strings. Furthermore, we may reasonably expect that information passed across emulator boundaries will most often, or nearly always, be addressing information: the symbolic name or number of a segment where the called program will find data structures for processing, or the name of a file to be printed, for example. Details of the conventions are found in the user manual.

Enter and Return

This subsection explans the micro-programmed system primitives ENTER and RETURN which handle external program calls and returns.

To describe programs which can be called externally, we use Routine Control Blocks (RCB's). The term is taken from Tafvelin, and used in the same meaning as he uses it. A Routine Control Block is a record with three fields:

1. size of scratch data segment.
2. emulator name.
3. virtual address of program entry point.

The use of the RCB is elaborated below.

The following two situations are intrinsically different:

1. Program A which runs on emulator X is called from some external program; and
2. a return is made to A from some external program which A itself has invoked in a situation such as 1).

In both situations control at the hardware or microprogram level must pass to X, and in both situations X's program counter will be set to point somewhere in A. However, in situation 1), the state of X, i.e. the con-

tents of its registers, must somehow be initialised to establish a processor configuration which will allow the execution of A, while in situation 2) the virtual processor needs to be restored to the state which it was in before the external call originating from A was executed.

Because of this difference we assume that any emulator has two entry points, a "new-entry" point for the former of the above situations, and a "re-entry" point for the latter. In fact more distinction may be needed, as the new-entry may itself represent a sort of re-entry, e.g. programs on different emulators call each other recursively, the question being whether on each external call an entirely new incarnation of the emulator supporting the called program is desired, or whether some version, already lying around, could be used. This question may be seen as an aspect of the wider question of generalized transfer of control, which is very briefly touched upon in section IV.6. Basically, the problem arises when an emulator must obtain information to initialise its state at new-entry. We shall return later to the question of where this information may come from.

The distinction which matters to the system is between the situation where an RCB explicitly handed to the ENTER-microprogram specifies a program to be entered, and the situation where the RETURN microprogram implicitly uses return information in the super-stack to find an emulator to resume control at a point where its state had been saved. Tables are kept of the new-entry and re-entry points of all emulators in the system.

A third module which an emulator must contain if it is to support external program calls is a "calling sequence" part which will save the virtual processor state and then pass control to ENTER.

To clarify the implications of these demands on the structure of emulators we present a standard example of what these parts of an emulator may look like, in an informal notation. Let the state vector consist of n registers.

```
NewEntry(PC):
    1:   for i=1 to n-1 do initialise i^th register
    2:   ProgramCounter:=PC (n^th register)
```

3: move parameters from super-stack to private storage using GetParam

4: goto instruction fetch (or entry point into main algorithm)

CallingSequence (external routine):

5: Get parameters from private storage and push them on the super-stack

6: MarkStackFrame( )

7: for i=1 to n do Push (i^{th} register)

8: Push (emulator's own name)

9: calculate address of RCB and place it in standard location

10: goto ENTER

ReEntry( ):

11: for i=1 to n do i^{th} register := GetRegister(i)

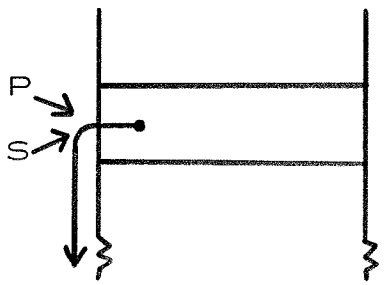
12: move results from super stack to private storage using GetParam

13: UnmarkStackFrame( )

14: goto instruction fetch loop.

We now present some pictures which illustrate the changing state of the super-stack as an external call proceeds. Referring to the CallingSequence above, we have the situation in Fig. 2 before the line labelled 5. Between lines 5 and 6 we get Fig. 3. Fig. 4 depicts the state of the super-stack after line 8. And finally after ENTER, the situation is as shown in Fig. 5; this is the "normal" situation when control resides "inside" some emulator, and in that sense similar to Fig. 2, but now one more stack frame is present than before the call started.

Fig. 2



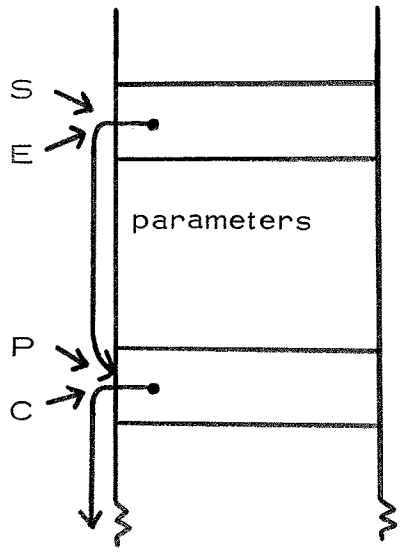


Fig. 3

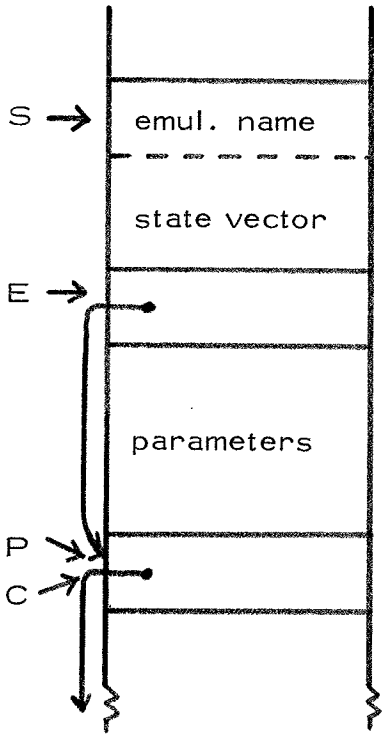


Fig. 4

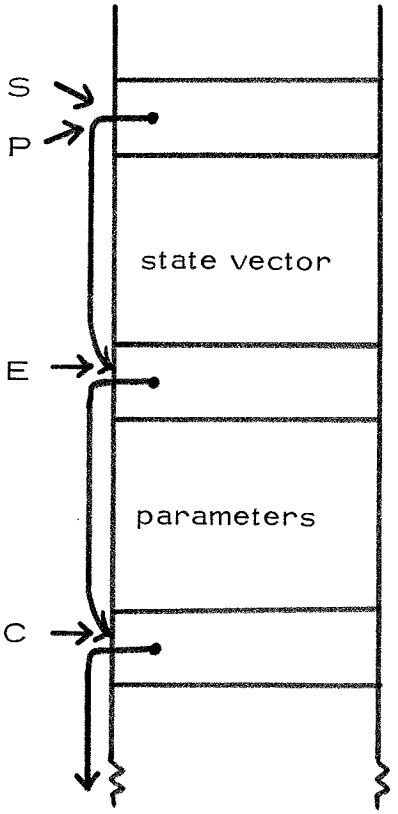


Fig. 5

Three points in connection with this description deserve further comments.

1) The Calling sequence and re-entry parts stand in a natural relation to each other in that every time control goes out of an emulator through its CallingSequence it will eventually return to ReEntry. Similarly, whenever control passes into an emulator at NewEntry it will eventually leave the emulator again and go to the system's RETURN. That part of the emulator which contains the branch to RETURN may also deserve to have been described in some detail. We merely remark that it does not need to manipulate the super-stack pointers (system registers), but that it may involve writing results which are passed back to a calling program into the super-stack.

2) All moving of parameters or results to or from the super-stack may involve conversion between the system data formats discussed above

and the emulator's own data representation. (This takes place in the lines labelled 3, 5, and 12.)

3) Lines 1 and 11 in the above description may hide a situation which is really more complicated than it appears. Suppose an emulator uses working registers to hold decoding tables, and suppose a fair number of such registers are used. It may seem odd to save these registers away as part of the emulator state vector, particularly as the tables are constant. Also at NewEntry it may be difficult to generate the values in such tables. We therefore make it possible to associate a small segment with each emulator, a segment which belongs to the system, and which is therefore known to any process automatically. Hereby we allow all references (i.e. from all processes) to such a segment to use the same segment number, realising that it is difficult to microprogram symbolic references. A segment of this kind, called an emulator segment, may be initialised in the system at the time a new emulator is introduced, and emulator initialisation values may be placed in it.

An emulator segment may also be used to solve the above mentioned dilemma as to whether a new-entry may really be a re-entry. Specifically, it it is desirable in an emulator's CallingSequence to save its state so that it may be restored, not only at a re-entry, but also at a (recursive) new-entry, the state may be saved in the emulator segment. In fact the emulator segment may be thought of as an emulator's own storage, not belonging to any program it interprets but to the emulator itself. The system will re-initialise the contents of all emulator segments at the beginning of each process.

The groundwork has now been done for a description of ENTER and RETURN. We first look af ENTER in detail. Before considering the Routine Control Block handed to it, ENTER does the following so as to preserve the return information in the super stack: (cf. Figs. 4 and 5)

Push (current linkage segment number) (see section 3)
Push(E)
P:=S.

Then ENTER interprets an RCB as follows. The size of an optional scratch segment may be specified to provide local storage to the called program. Unless this size is 0, ENTER must create such a segment. Also the emulator microprogram to receive control must be placed in control store. To perform these two tasks ENTER calls the higher level of the system. Microprograms are kept in ordinary (write-protected) segments, and the equipment for translating emulator name into symbolic name and further into segment number is all found in the parts of the system which are programmed in BCPL. Finally control goes to the NewEntry of the "called" emulator, with the contents of the virtual address field of the RCB and the number of the optional data segment being passed in standard register locations. If the RCB represents a microprocedure and not an interpreted program, its virtual address part is at worst superfluous. The point is that the right microprogram gets control. Thus we may use the same kind of RCB to represent micro-procedures and interpreted programs, and, correspondingly, the same version of ENTER for both.

Dancing on the delicate barrier between micro- and higher level in the system requires a light and steady foot. A couple of non-trivial problems are hidden in the above description of ENTER. How does the microprogram ENTER pass control to system routines at the higher level; by calling ENTER? Can a RETURN be made to ENTER? When ENTERing a BCPL system routine, how is the OCODE emulator loaded? The answers to the first and third questions are connected. Obviously the OCODE emulator and system microprograms must be resident in control store, or no emulator could ever be loaded; for much the same reason system routines do not require the creation of scratch data segments. It follows that when higher level system routines are called, ENTER may pass control directly to the OCODE emulator without the steps described above. If we assume this characteristic to hold for all OCODE programs then ENTER may simply test the emulator name to see if a given RCB represents a system routine. Thus ENTER may in fact invoke the system programs which swap emulators in the standard way, i.e. using ENTER, without running into recursive trouble, provided it knows where the RCB for the system program sits. The remaining question of RETURNing to ENTER is solved by giving the ENTER microprogram an emulator name and an associated entry in the table of re-entry points. The same mechanism is employed for other system microprograms such as READ/WRITE and LINKREF which may need to call higher level system programs for help.

One might think RETURN could be simpler. In fact it involves exactly the same kind of complexity. When control is passed to the RETURN microprogram, it looks in location P-2 in the super-stack for the name of the emulator (X) to return to. Unless X is a resident emulator, RETURN then calls the higher level routine which swaps emulators. Eventually the OCODE emulator will return, to RETURN that is. Taking a look at its own emulator name RETURN will realise that it is itself resident (although it is only aware of the residency characteristic, not that it is looking at itself), so that it may pass control directly to its own ReEntry, where it resumes the real task of returning to X, a task which is now very simple.

RETURN could also usefully destroy a data segment of the kind specified in an RCB, if one had been associated with the program invocation which is returning. This would necessitate storing the data segment number somewhere in the super-stack. However for the initial implementation we rely on a technique where the higher level system keeps track of the disk pages used for such segments, so they may later be recovered for other use. All that is lost, then, is a segment descriptor. This approach simplifies RETURN and super-stack contents. Besides, by not collecting the assumed garbage at return time, we allow a kind of general retention.

The above description of ENTER and RETURN is complete except that it does not take linkage segments into account. These will be introduced in the next section.

Emulators, Compilers, Programmers

At this stage there may be some confusion as to who has which responsibility in connection with external calls. How much must the programmer do, how much the compiler, and how much the emulator? The system design does not answer these questions, nor should it; there are several possibilities, depending on the language and virtual processor involved. In general it seems reasonable to require that the programmer provide specifications of external entry points, including symbolic name (see next section) and type, i.e. the types of parameters and results, This might be the case in, say, a PASCAL or a FORTRAN machine. However, in a

BCPL-oriented machine it is possible to allow the programmer control of individual super-stack primitives. In some other language system, a number of external entry points might be known to the compiler as part of a run-time library, and the programmer might not even know when external calls are brought about.

Also the division of labour between the emulator and the compiler may vary from case to case. The ReEntry, NewEntry, and CallingSequence parts of an emulator, which we have described, need not necessarily be programmed as modules set apart from the rest of an emulator. It may be convenient to create machine instructions which allow a compiler to build them as code to be interpreted for each individual external entry point. In particular this may be useful for parameter conversion.

3. External References and Linking

In section II. 3 the need for a unified representation of external references was argued. After virtual memory has been introduced this amounts to cross-segment references. Let us be more explicit. Assume the code of a program lies in segment A; the processor interpreting this program has in its registers the segment number of A and also of data segment B which contains the processor stack and other data of the computation in progress. If this program makes an external program call, or references data other than that in B, its processor may not know the relevant segment number, and therefore some manner of indirect reference is needed.

A general solution for this problem was found for the MULTICS-project [3], where linkage segments were introduced, and Tafvelin [7] has used MULTICS-like linkage segments in his multi-emulation environment. Our design follows his approach. To any code segment, i.e. compiled program, belongs a linkage segment. For each external reference from the code (which may be referenced from several places in the code segment) the linkage segment contains a link reference (in MULTICS: "ITS") which is basically a virtual address. However, as segment numbers are dynamically assigned, and as the point being referenced in some external segment is likely to change location over time within that external segment, e.g. with recompilation, it is impossible at compile time to find virtual addresses for all points of external reference which will be valid at run

time; therefore symbolic addressing is used. A link reference is always in one of two possible states, either resolved, in which case it contains the virtual address of the point referred to, or unresolved, in which case it contains the virtual address of a location in the code segment to which it belongs, where the symbolic reference may be found in the form <symbolic segment name, symbolic word name>.

The mapping of symbolic word name to word number is performed by looking up the name in a table of externally referenced locations which is found in the beginning of the referenced segment. Any segment which contains targets for external references must begin with such a table. The segments for which this is relevant are linkage segments and segments containing data structures accessed using external references, such as data structures on which several programs running on different emulators perform operations.

The reason linkage segments are included here is that in addition to link references they also contain RCBs for externally available entry points in the code segment to which they belong. Thus an external code reference goes from caller's linkage segment to called program's linkage segment. The segment number of the linkage segment for the program in execution is always in a system register.

An external reference from the program in code segment "PROG1" to the entry point "BLUE" in the code segment "PROG2" is illustrated in Figs. 6 and 7. In Fig. 6 the reference is unresolved; in Fig. 7 it has been resolved. The segment "PROG1". "CODE" has segment number a. In Fig. 6 the virtual address <a, r> is represented by the two letters occurring in the linkage segment. In Fig. 7 all references by virtual address are drawn as arrows. It should be pointed out that for the arrow from "PROG1". "CODE" to the correspronding linkage segment the segment number of the virtual address is implicit.

(Figs. 6 and 7 - see the next two pages)

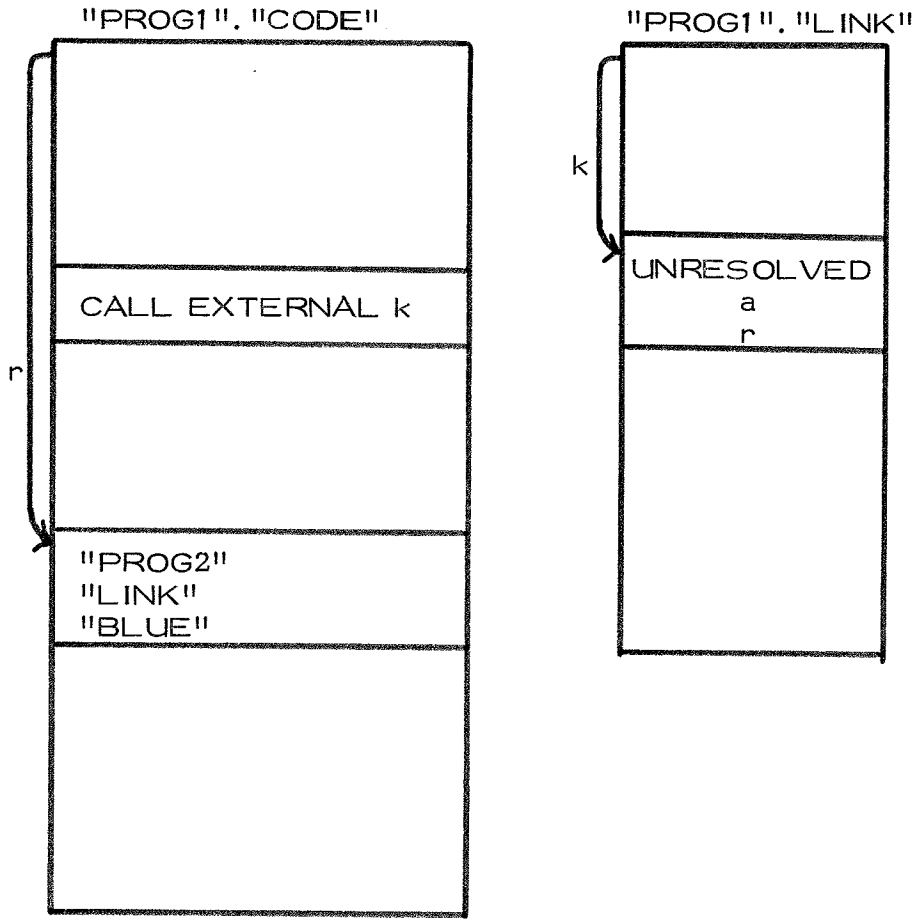

"PROG1". "CODE"
CALL EXTERNAL k
r
"PROG2"
"LINK"
"BLUE"
"PROG1". "LINK"
k
UNRESOLVED
a
r

Fig. 6

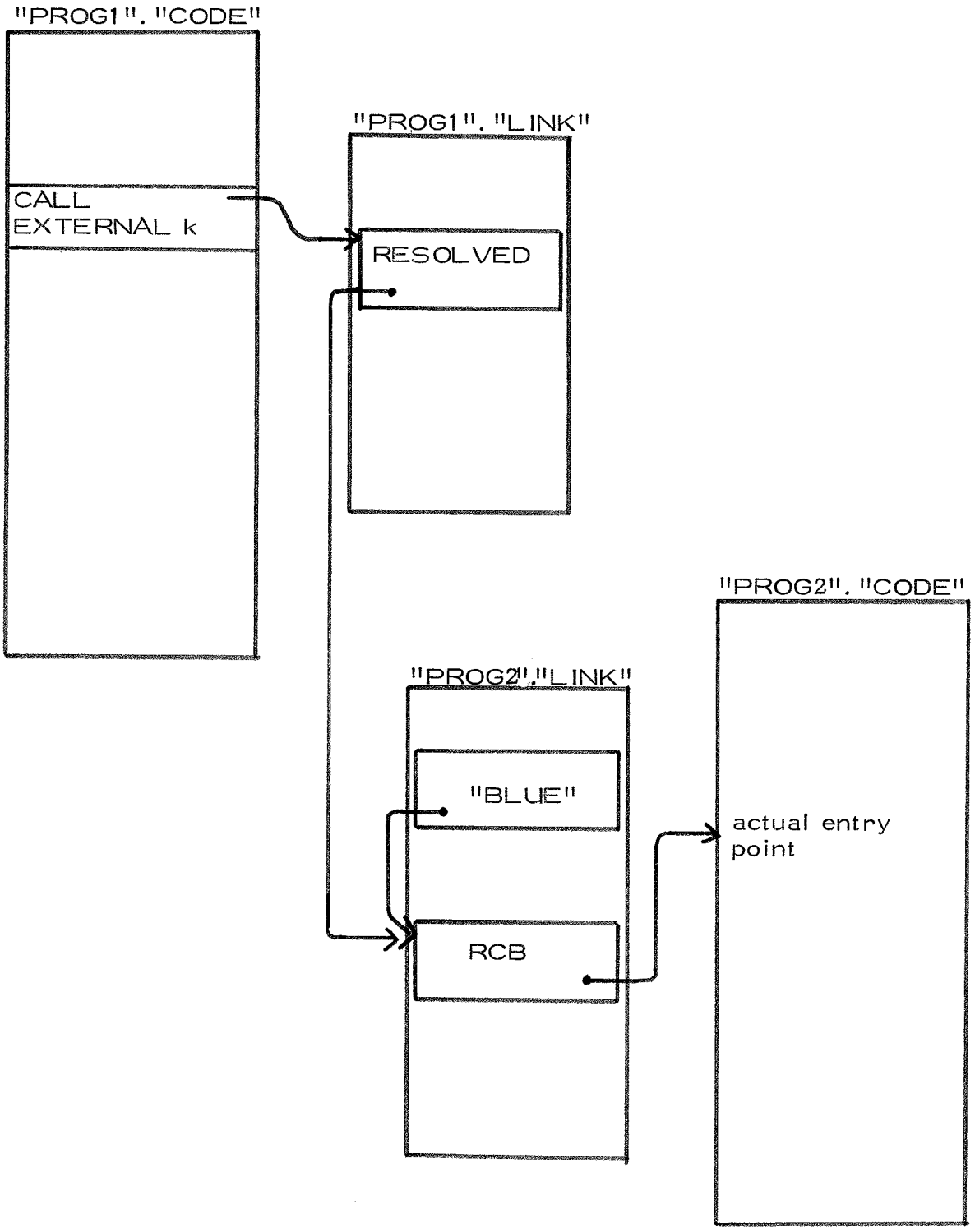

"PROG1". "CODE"
CALL
EXTERNAL k
"PROG1". "LINK"
RESOLVED
"PROG2". "CODE"
"PROG2". "LINK"
"BLUE"
RCB
actual entry
point

Fig. 7

When making an external reference, which we now understand as an indirect reference through the current linkage segment, an emulator must call the system microroutine LINKREF, with the location of the link reference within the linkage segment as an argument (k in Fig. 6). LINKREF can then read the link reference. If it is not resolved a call must be made to a higher-level link resolution routine which uses the following algorithm:

1. look up the symbolic name of the referred segment S in the Known Segment Table; if it is known get its segment number and proceed to step 4.
2. make S known and proceed to step 4 unless S is a linkage segment.
3. make the corresponding code segment known; its symbolic name may be derived from that of S.
4. look up the symbolic word name in S, and finally overwrite the link reference to indicate the virtual address which has been found, and the fact that the reference is now resolved.

A moment's thought reveals that updating the linkage segment register becomes an additional task for ENTER and RETURN. For ENTER the new value in the register is simply the segment-number part of the virtual address of the RCB for the called program. The old value must be saved on the super-stack (see Fig. 1) and later restored by RETURN.

With this scheme an external program call proceeds in two stages, first a call to LINKREF to find the address of an RCB which is then passed to ENTER. The fact that there are two stages is due to LINKREF being general enough to handle both code and data references.

The gains from the use of linkage segments are as follows:

1) External references, which are alien to the inner worlds of emulators, are separated from code and data into their own segments, where they provide a simple and uniform representation of link information, irrespective of emulators' internal formats.

2) Link resolution of code references provides late dynamic binding at no extra cost. It may not seem so important to delay link resolu-

tion in a one-process environment without code-sharing, but consider for example a command interpreter which, according to the commands received, may call on a variety of programs; in such a situation it is desirable not to have to load them all, but only those needed, when they are needed.

3) In a multi-programming environment, linkage segments help to provide re-entrant code segments, and thereby code-sharing among processes, as they allow the variable information related to the code segments to be factored out into small linkage segments, of which processes may have one each. In our situation the corresponding gain is probably less important: when code segments do not need to be modified, they can be write-protected; this serves to protect them, and it is not necessary to move them physically, when they are swapped out of core.

By the phrase "variable information related to a code segment" is meant all information which is unknown at compile time. All addresses occurring in code segments are either symbolic or offsets within segments, and therefore do not require any binding between compilation time and run-time. It is a well-known aspect of segmentation with linkage segments that no explicit loading is needed; this pleasant characteristic is invariant under the presence of several emulators. All compilers in the system are expected to produce code which is ready to be ENTERed.

One load-time problem was skimmed over above: When a linkage segment and the corresponding code segment are made known to a process all references from the linkage segment to the code segment by segment number must be filled in, since this is the time when the segment number is determined for the duration of that process. These references include both RCBs and all the link references, as these are at this time unresolved and therefore point to the code segment. This action takes place in step 3 of the link resolution algorithm. Thus while the contents of a code segment never change once the segment has been compiled, the corresponding linkage segment must be modified to reflect the incorporation of the segment pair in the address space of different processes. Within each process it must again be modified as external references are resolved. It follows that each process needs a separate copy of a given linkage segment. Since a linkage

segment is small it is reasonable to lock it in physical store once the link resolution routine has triggered it to fault. We thereby avoid having to make an explicit segment copy, as the locking ensures against the possibility of having to overwrite the permanent "template" version on disk by swapping a linkage segment out. The advantage of the linkage scheme is still that linkage segments all have a common format so the problems of linking can be handled once and for all by the link resolution routine.

4. Permanent Disk Storage Arrangements

There are three reasons for including a disk filing system in the present design:

1) The disk, as a physical resource, must be shared by emulators; this means that common formats are necessary. (We might make some inflexible division of the disk into sections for each virtual emulator, but this would seem unreasonable in a system containing a varying number of experimental emulators.)

2) One of the reasons for introducing multi-emulation was the saving in systems programming obtained by having a common set of programs for this task, and these were more or less in existence before the project started.

3) By introducing segments which reside on disk we have already gone some way toward administering the disk from the multi-emulation system programs.

The system which administers files and permanently existing segments, and which is described in the following, is an adaptation of the filing system designed for the RIKKE BCPL system [16], and is therefore influenced by the Oxford OS-system.

The Disk

A DRI 200 disk unit will be connected to RIKKE by a controller built by the DAIMI hardware staff. Logically speaking, the disk contains a number of pages, each containing a four word header and a 256 word data block. The disk is accessed by means of micro-programs, integrated with the O-CODE machine, which communicate with the disk controller. A point of in-

terest is that when handling page transfers, these programs separate header and data block, so that these may be read from (or written to) locations quite separate in memory, even residing in separate physical memories. This separation allows all control of page header information to take place in the BCPL part of the system, which has its own private data area, even though the data contents of pages may be transferred to or from any location in physical memory. Page headers are used to validate disk pages in a fashion inspired by Lampson [6]. Page headers identify the file or segment in question, and the serial number of the page, information which is checked on every disk read. Also a checksum computed by the microprogrammed transfer routine is included in the header.

Technologically it should be noted that the disk controller is not a channel, actual transfers take place as programmed i/o at the micro-level. In a slow one-process system the use of interrupt-driven disk transfers is hardly of interest, quite apart from the problem of simulating these interrupts on RIKKE, which has no hardware interrupts. For these reasons, disk cylinder seek operations are simply waited for. Transfers of several pages in a row, i.e. segment swaps, are made efficient by the use of a buffering scheme in the disk controller, which allows overlapping of transfers to or from RIKKE memory with mechanical operations and transfers to or from the actual disk. This scheme also removes any critical timing requirements from the disk-handling micro-programs, and thus serves to simplify them.

Files and Segments

Segments are ordered sets of words of information. So are files. Segments may be created and destroyed at any time during a computation. So may files. A segment may survive the computation which creates it by the establishment of an association between the segment value and a symbolic name and the insertion of this pair in a directory. The same is true for a file. The difference between files and segments is that segments are randomly accessed, while files must be read or written sequentially. Segments provide emulators with linearly organised memories, while files are used for sequentially arranged information, such as texts to be input or output via peripheral devices, or edited. Files are specifically intended

as a spooling medium for input and output.

Technically, this distinction means that a segment is either wholly in or wholly out of physical store, whereas for a file it suffices to have direct access to one page, at the location where reading or writing is currently taking place. (In a paged system, this distinction would be immaterial, and indeed the MULTICS system only has segments.) Apart from this technically founded difference, files and segments are treated alike; they occupy the same type of entries in directories, (i.e. directory searches are the same for files and segments) and they are described by headers in the same format in the system master file. It is also quite easy to convert a segment to a file, and vice versa, preserving the information content.

Segment and File Values, Headings

In accordance with the philosophy of the OS-system, every file or segment has a value, an integer, which serves to identify the file or segment during its whole existence. A segment value should be distinguished from a segment number, which serves to access the segment while it is known to a particular process. The value is used for the following purposes:

1) as control information in the header of every page of the file or segment.

2) as the identification associated with symbolic names in directories.

3) as an index in the master file, which contains a description, called a "heading" of each existing file or segment, sufficient to access its constituent pages; a heading also contains a file or segment title which is a character string.

Directories and Symbolic Names

Directories are files. They are used to permanently associate symbolic names with files and segments. As in Oxford OS and DEC's TOPS 10 system, a symbolic name consists of two character strings.
The first string is the name a user gives to the file or segment in question, whereas the second string or "extension" describes the kind of information contained therein. Thus a number of logically related files or segments,

such as source text, compiled code, and documentation for a given program, may all be given the same first name but will have different second names. We write symbolic file or segment names as two quoted character strings separated by a period (.).

In some instances the extension name is used to inform the system of how to process a given file or segment. Thus the link resolution routine expects code and linkage segments to have second names "CODE" and "LINK" respectively. User-built language systems with compilers may add to these conventions.

A directory contains a list of pairs of the form ⟨symbolic name, value⟩ and nothing else. There are routines in the system to insert and remove directory entries, and to search for or rename entries. The same symbolic name may occur in several directories without causing confusion, regardless of whether all the occurrences refer to the same file or segment.

As a directory is itself a file, it may be associated with a symbolic name and inserted in another directory (or in fact, in itself if one cares to do so). Thus a directory structure, for example a tree, may be built. Some particular directory will at any given time be considered to be the current directory, and many system directory references will go through this directory, specifically those having to do with link resolution.

Overview, Robustness

At this point we may illustrate (by a rather sketchy diagram) how a file is represented in the system. The file in Fig. 8 consists of 7 pages, and is associated with the name "MATRIX". "DATA" in the depicted directory; it has value f and the title "MATRIX". As illustrated, the heading for f contains the addresses of the first and last page in the file; the remaining pages are addressed using a chain which occupies one word of the header of each page in the file. For a segment, one disk page is allocated to hold a table of pointers to all the segment's constituent pages (this information is needed when the segment is swapped out on disk). The address of this page is kept in the heading instead of the first and last page addresses.

(Fig. 8 - see next page)

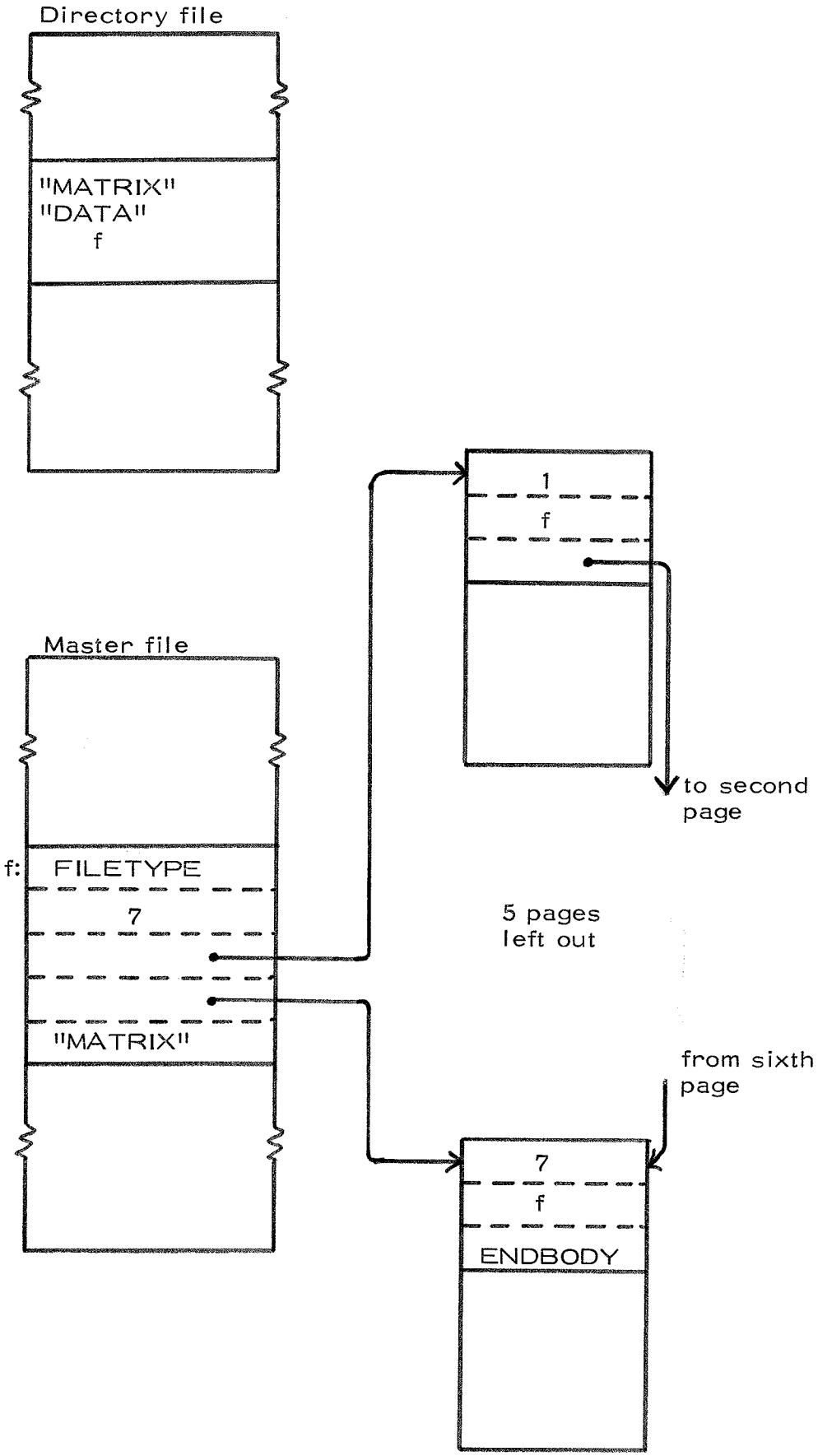

Directory file
"MATRIX"
"DATA"
f
1
f
to second
page
Master file
f:
FILETYPE
7
5 pages
left out
"MATRIX"
from sixth
page
7
f
ENDBODY

Fig. 8

The information which we have chosen to gather in a file heading could also have been directly associated with a symbolic name in a directory entry, thus saving one physical disk access each time access is opened to a file or segment. However this only occurs when a file is set up for reading or writing, or when a segment is made known to a process, and there are two important gains to be obtained from having separate file (segment) headers.

1) There may be any number of directory entries (including none) in any number of directories associated with a given file or segment. This allows sharing of information among users, each having his own private directory.

2) It follows that directory entries are not essential for the well-being of files; a file may exist without any directory entries referring to it. However if this state is permanent, i.e. it lasts beyond the duration of a process, it is likely to be the result of the corruption of one or more directories. Indeed the disk will from time to time be cleaned, implying that any page which does not belong to any file or segment referenced from some directory is made a free disk page, i.e. avialable for allocation. When corruption of directories occurs, the heading information may be used to re-establish symbolic access to "lost" files; this is the reason for having file titles. Ordinarily a file's title will be identical to the first name in the most important directory entry for the file. Thus the redundant information in headings makes the filing system robust in the sense that it can withstand directory corruption.

Similarly, the information in page headers serves to preserve a file's or segment's integrity even if its heading should become corrupted. In this case, however, the type of file or segment is lost as well as the title. We see that corruption of the master file is not fatal for the actual stored information. It should be noted that tolerance to corruption of directories and the master file is an essential aspect of robustness of the filing system as a permanent storage mechanism, because directories and the master file are accessed very often and are therefore vulnerable to random errors.

Access to Files

The RIKKE BCPL system uses streams [5] to read and write file contents. The already existing software for this purpose is also used in the multi-emulation system. This approach means that the routines for creating and handling streams must be made externally available. For the sake of efficiency the stream primitives: NEXT which gets the next word from a stream, i.e. file; ENDOF which tests for end of file; and OUT which adds a word to the end of a file, are made callable at the micro-level. The necessity of starting the somewhat heavy machinery for external calls once for every word transferred to or from a file is thus avoided. Moreover, this approach provides one more kind of equality between files and segments: both may be accessed at the micro-level.

In the initial implementation, all calls on NEXT, ENDOF, and OUT will be referred to the BCPL level, where these functions exist already. However one may later choose to microprogram these functions, or major parts of them, in a tuning process similar to that of the Oxford OS-system, where Next, Endof, and Out have become single instructions of the interpreter used with that system.

5. System Overview

After the features of the RIKKE multi-emulator system have been described, two questions (at least) remain unanswered: How are processes started, and how does the system react to errors? These questions both relate to the framework within which the multi emulation facilities are provided. We have explained that important system modules have been programmed in BCPL; in fact these parts have been added to the existing RIKKE BCPL system, in a sense integrated into that system. It has therefore been natural also to use the basic command interpreter loop of the BCPL system as the place from where multi-emulation processes are initiated. This has been accomplished by adding to the routine which restores the BCPL system to its standard initial state a part which restores the multi-emulation environment to its configuration at process start. We could now redefine a process to be that computation which takes place between two succesive calls to the command interpreter form the basic system loop.

Error conditions in the OCODE machine on which the BCPL system runs cause the OCODE emulator to generate a trap, i.e. a call to a general error handling routine which is in fact a modified command interpreter loop, used to obtain debugging information interactively. The error handler is also called by the BCPL system routines in error situations. It is simple and natural also to use this facility for the multi-emulation system. From the modules which have been integrated into the BCPL system it is easy enough to make an ordinary routine call to the error handler, and for the microprogrammed parts a bit of trickery has been invented in order to bring about an OCODE machine trap whenever an error occurs.

The debugging tools in the existing error handler consist basically of facilities for inspecting relevant parts of the OCODE machine's store, with particular attention given to the stack. Facilities extending these may be created, allowing the operator to inspect the contents of any segment, addressed symbolically or by segment number, and the super-stack.

A third area where the BCPL system becomes useful is i/o. It is expected that programs which run on virtual machines other than the OCODE machine will do their i/o to or from files (spooling). The BCPL system command interpreter makes it easy to transfer files to and from peripheral devices. Finally we note that the BCPL system also includes an interactive text editor.

This discussion of the relationship between the multi-emulation facilities and the RIKKE BCPL system concludes our presentation of the RIKKE multi-emulation system.

IV. ALTERNATIVE MODELS FOR MULTI-EMULATION

In this chapter we consider some models for multi-emulation systems which differ from the system described in the previous chapter. Some of the ideas which are mentioned come from discussions about multi-emulation which have taken place within DAIMI, and some come from the meagre published literature on the subject.

In the latter catagory we find reports on the Burroughs B1700 system and on the compatibility features of the IBM 360 models. There are other (mostly small) computers besides the B1700 which allow user micro-programming. A number of these are mentioned and discussed with respect to their microprogrammability in [11], but a perusal of the descriptions found there does not reveal that multi-emulation in the sense defined in chapter II is supported in a general fashion by any of these systems. We therefore limit our discussion of existing computer systems to the above mentioned two. Since little is known to this author about the real multi-emulation capability of the B1700, the discussion of that system becomes primarily a comparison of the suitability of the B1700 hardware and of the RIKKE-1 hardware with respect to multi-emulation.

We open the chapter with a section on the relation of multi-emulation to multi-programming, a question which seems to be basic also to some of the subsequent discussions. Other sections are devoted to the paper by Tafvelin [7], to the concept of a multi-interpreter system as opposed to a multi-emulator system, and to ideas of generalised transfer of control.

The inclusion of this chapter in the present report is intended to serve a double purpose. One purpose is to put the designed multi-emulation system in perspective compared to other similar implementations and ideas, thereby indicating both its power and its limitations. Another purpose is to justify some of the more important design choices which have been made.

For the sake of brevity we shall refer in the following to our multi-emulation system simply as the ME-system.

1. Multi-programming

The designers of the RIKKE and MATHILDA computers have written [14]

> "A core project in the systems area is to define and implement a micro-monitor, resident in RIKKE control store, to allocate the resources of the system, I/O as well as processors, and which will allow multi-programming among emulators".

In this view the hardware organisation of working registers into groups would be used to allow the state vectors of several emulators to be in physical registers simultaneously, with basically one state vector or "context block" per group. The so-called micro-monitor should be very simple, its task being basically to multiplex emong emulators by controlling the value of the register group pointer and the micro-program counter. In this scheme the active emulators, several of which may be the same if desired, provide a number of virtual machines, executing in parallel. In the resulting model, cooperation between programs on different emulators would take place using process-synchronisation and inter-process communication tools such as semaphores and message queues. These would then have to be implemented as part of the micro-monitor. It was thought important that supervisory functions be confined to the micro-programmed monitor - no emulator should be preferred to any other, there should not be some virtual machine in charge of the others, no master, no slaves. Comparing this approach with the ME-system, there are three major differences.

1) Instead of external subroutine calls this model provides inter-process communication. This difference is a consequence of two different philosophies in viewing a multi-emulation system: n one view, there are a number of virtual processors operating in parallel with tasks distributed properly among them; in the other, there is only one processor, the physical machine, which acquires different "virtual" characteristics, depending on the task it is dealing with. The author contends that the latter view is more natural from a user point of view in the many situations where a user program invokes a subroutine, which just happens to run on a different emulator. In the adopted design this fact is represented by the emulator name field within an RCB. As discussed near the end of section III the

existence of external calls may be more or less hidden from users. In the micro-monitor multi-programming model, explicit communication with some particular virtual machine must take place depending on the subroutine in question. It is admitted , however, that it may be possible to hide the dirt away in compilers.

2) The stumbling block in discussions about the micro-monitor was always the management of core storage. An idea was presented that each virtual processor should have its own private portion of storage to be administered using base and limit registers. As a possible hierarchy of processes developed, these units of storage would be subdivided, to correspond to a tree-structure of processes. These ideas never developed satisfactorily. The fixed amount of physical core storage actually available would be incompatible with variable demands from a variable number of processes. It is an important idea of the ME-system design that a general and flexible core management scheme is a basic necessity for multi-emulation. However, this can hardly be achieved without higher-level programming, and this again implies that some interpreter will be running programs which manage resources for all, and in that sense it becomes a master. Actually though, it is the core management program which is master, but it does mean in the ME-system that the OCODE emulator obtains a special status; in fact it is indispensable, which cannot be said of any other emulator.

It is not clear whether the term i/o devices in the above quotation also covered a disk unit. In the philosophy of the ME-system, a disk is considered as a storage device, the utilisation of which must be integrated with that of core storage in order to provide memory space for several virtual processors. The tasks of administering permanent information on disk is also seen as a necessary function which must be included in any set of basic programs, whether they be called operating system, multi-emulation system, or micro-monitor Again this implies that a set of disk management programs must be placed between the physical disk and "user" programs, which access information stored on disk. In this author's view it is impracticable to perform reasonable disk storage management on RIKKE-1 by micro-program alone; higher-level programs are necessary.

To expand this point: RIKKE microcode is not suited for systems programming, but at most for implementing virtual machines and algorithms of the kind for which it has been designed, such as arithmetic ones. Thus any system providing basic services must make use of higher-level programming. The problem must be faced of interfacing services at the micro-level to higher-level programs running on at least one selected emulator.

3) A micro-monitor for the multi-programming of virtual processes including process-communication mechanisms would provide a general opportunity for work with multi-programming applications. The ME-system seems not to accomplish this. This limitation deserves some further consideration.

Emulation of systems for parallel computation

Emulators defining virtual machines which permit algorithms to be organised into parallel or pseudo-parallel computations are not excluded from the ME-system. In particular, emulators which utilise the two physical processors of the system, RIKKE and MATHILDA, in parallel, are heartily welcomed, along with any designs which merely simulate parallelism, perhaps for the sake of experiment. It is only where such emulators interface to other virtual machines, running under the ME-system, that a stack-oriented discipline is imposed. Drawing a network picture of the logical movement of control over time, we see that the notion of parallelism is subjected to serialism at the outer level as illustrated in Fig. 9, where ⊠ is used to mark transfer of control across emulator boundaries, either external calls , or external returns.

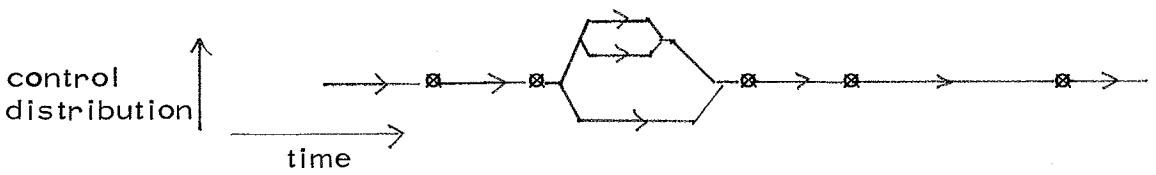


Fig. 9

Multi-tasking at system level

Fig. 9 still shows the limitation that ⊠'s are forced to occur serially. Applications in which parallel computations are each distributed over several virtual processors are excluded from the ME-system. Nevertheless this kind of generality may be desirable with respect to possible future applications. It is therefore of some importance that the system design does not preclude extensions aimed at accommodating such applications. Stated more positively, there should be a natural way to implement suitable extensions.

In section V.1 we suggest a possible (natural) extension similar to that by which multiple tasks are provided in the B6700; this will allow for example coroutines as separate co-operating tasks.

Multi-user service

The discussion of multi-programming has so far been concerned with the demands on system facilities by algorithms which are parallel in nature, realised through one form or another of parallel computation The idea of time-sharing among several users, situated at so many terminals, has not been touched upon. Nor will it really be dealt with here, simply because the RIKKE-1 processor is not deemed to have the capacity to serve more than one user with reasonable performance. There is nothing in the RIKKE configuration, i.e. no hardware interrupts and no channels, which points toward such use. Based on experience with the BCPL system the judgement is passed that RIKKE, when emulating virtual machines, will not be fast enough for physical processor time to be devided so as to serve a number of users simultaneously, unless these users refrain from activities which consume more processor time than does simple editing. Design efforts aimed at producing a time-sharing system would only produce boredom and irritation, according to users' inclination. Moreover, the creation of a multi-user system on RIKKE would not meet any real need.

2. Compatibility Features on the IBM 360 Computers

As mentioned in the introduction (chapter I), the microprogrammed processors which are used to implement most of the models in the IBM 360 line of computers, may be switched to imitate older IBM models, notably models from the 1400 and 7000 series. In fact compatibility features for

second generation machines are quite common on commercial third generation computers. These features are supplied to assist with the tasks of program conversion involved in the installation of new equipment. The following brief comparison is based on the paper of Rosin [8], which provides a good overview of these techniques as well as an interesting discussion of IBM policy with respect to user microprogramming. Several references to more specific IBM documentation are included. Incidentally, Rosin's paper also foresaw some of the important questions pertaining to a more general form of multi-emulation such as the passing of information between emulators, the problems of assuring that one gets back into the right emulator after some type of external call, and the formulation of a common filing system. This is remarkable if only because the paper appeared a relatively long time ago (1969).

Comparing the 360 compability features to the ME-system it is interesting to note that in the IBM 2025 processor (ordinarily used for model 360/25) which supports a compability switch to a mode so that it acts as a 1400 model, a writeable control store is used (for other processors read-only control store is employed). Thus potentially an effect similar to our emulator switches could be realised. Another interesting aspect is that in several cases the i/o instructions of an emulated machine are implemented by means of simulation in the corresponding 360 model, a situation which is reminiscent of the ME-system where the capabilities of the OCODE machine are employed to provide a filing system also for other machines.

The scope of this comparison is, however, limited. The kind of emulator switching provided on the 360 does not support general external calls across emulator boundaries. Emulator switching is accomplished by such means as the reading of a card deck and thus requires operator intervention. Nor are the processors intended for user microprogramming. We therefore have no opportunity to evaluate the more powerful aspects of the ME-system in this comparison.

The point of importance in this discussion of the IBM 360 models (and similar third generation machines) is that they testified to the validity of emu-

lation and of a limited form of multi-emulation as techniques for realising computing machines, even on a large scale. For the ME-system these systems are therefore important as an historical background.

3. The Burroughs B1700 Computer System

The only microprogrammable computer system known to the author which is widely available and supports multiple emulators in a general fashion, is the Burroughs B1700. Only "high level" information on the mechanisms whereby multiple emulators are supported may be found in the generally accessible literature; facilities are described mostly in terms of what they do for the user, whereas little insight is provided into how they are implemented. As information involving implementation details is difficult to obtain the following short discussion is based on two papers by Wilner in the 1972 FJCC proceedings [9].

At the outset we remark that the B1700 system and the RIKKE ME-system show some similarity, at least superficially. Both claim to flexibly support multiple emulators, and both involve a central storage management program to serve as an arbiter of claims for physical resources. The OCODE emulator and the SDL S- machine which interprets the Master Control Program (MCP) are both resident to secure availability of system functions. Similarly, both the MCP and the ME-system contain a microprogrammed resident part.

Hardware aspects

Some of the major differences between the B1700 and the RIKKE ME-system are, of course, due to hardware differences between the physical processors. A very general comment about the RIKKE-1 hardware design [14, 15] is that it lacks structuring elements which support general low-level systems programs, such as the ME-system. In particular, all three physical storage media, main store, wide memory, and control store, are plain random access devices, each having only a simple memory address register. All binding of addresses must be programmed, and therefore any structuring of storage contents must be enforced by layers of program placed between the hardware and user programs. The reasons for this design characteristic probably lie partly in considerations of cost, and partly in the notion that by imposing no structure, the hard-

ware allows any structure, i.e. a generality argument. By making no decision, one does not make the wrong decision. However, if one wants microprogramming experiments to take place in a common supportive environment such as the ME-system, one only postpones the decision until this system or environment is designed, and at that time the system designer wishes it had already been made and implemented in hardware.

Like RIKKE, the B1700 is intended to have no pre-defined processor-structure but rather to be an efficient general emulation vehicle. However, its hardware includes several important structuring features: defined-field memory addressing, (capability of addressing arbitrary bit strings as units of memory); virtual addresses both for ordinary storage and microprogram storage; also microprograms may overflow from M-memory (fast control store) into S-memory (ordinary slower memory), and still be fetchable for execution.

Wilner argues strongly that the defined-field addressability eminently supports the building of efficient emulators. In our context, where relationships between emulators,and problems arising from their coexistence are of greater interest than emulators' internal problems, it should be clear from previous discussions that suitable hardware-supported virtual addressing schemes would have been very helpful as a groundwork for the ME-system.

It should perhaps be clarified that no assertion has been made to the effect that RIKKE lacks hardware structure in general. For example, at register group level, there is (optional) stack structure which may eminently serve particular processors. Our claim is that structure is lacking for the support of an overall system.

A similar problem arises over physical processor registers. While the number of registers available for user microprogramming in the B1700 is large and flexible [11] it is still possible to present these registers in one or two pages of text and tables. Wilner states that emulator switching on the B1700 is hardware assisted, but does not go into detail about the kind of assistance. One may guess that there are particular hardware facilities present for the saving of register values (virtual processor state vectors) in particular memory locations. On RIKKE, such a scheme would first of all be impossible, because so many registers are in blind alleys, and moreover also unreasonable, since most of

the information kept in the totality of registers is not likely to be relevant to a particular virtual processor. Thus, by the inclusion of a multitude of facilities for the individual emulator writer, (facilities which may either assist or confuse him) management across emulators is made difficult. The ME-system approach to emulator switching - that emulators are themselves responsible for saving and initialising their states - in a sense amounts to giving up management of emulator switches.

In conclusion we state that as a general emulation vehicle the B1700 has a number of advantages compared to RIKKE-1 by being structured to run multiple emulators. We shall not comment on the relative merits of the two machines as hosts for individual emulators on their own, other than noting the truism that these must depend on the kind of emulator in question.

External calls

From Wilner's description it appears that the emulator switching mechanism of the B1700 serves two purposes:

1) To provide the right emulator for the MCP upon interrupts which must be handled by the system.

2) To allow different versions of the same emulator (S-machine) to be used interchangably during interpretation of a program in a particular S-language. This provides, for example, switches between a tracing and a non-tracing version of an emulator.

Comparing this to the ME-system, the idea of general external procedure calls across emulator boundaries is conspicuously absent. A restricted form of such calls must be available to allow requests to MCP-procedures. It is difficult to see why the general facility should not be available, when interpreter switching is claimed to be so simple, considering the avowed B1700 philosophy of attaining efficiency by flexible processor reconfiguration according to the task at hand.

4. Supporting User-Microprogram Development

The idea of employing MULTICS-type dynamic linking in a multi-emulation environment is taken from Tafvelin who has described the use of external

subroutine calls in a situation with developing user microprograms [7]. Tafvelin's paper has two concerns : the general linking problem; and the problem of flexible microprogram simulation. A scheme for microprogram development is presented which involves running newly constructed microprograms on a host machine simulator rather than directly on the host machine until they have been tested and debugged. Microprograms are at least as error-prone as any other kind of programs; for this reason special tools for revealing and locating microprogram errors are desirable. Such tools may be included in a simulator. Tafvelin therefore introduces a special kind of RCB which is used (among other purposes) to indicate that a given microprogram which is called (ENTERed) should be simulated. Such a system, of course, must be integrated with a host machine simulator.

True simulation of an apparatus as complex as RIKKE is difficult to realise, in particular with respest to the question of compatibility in a multi-emulation environment. A microprogram may run perfectly well when simulated alose, but in hard life, where other microprograms contest for register resources, it may cause disaster. The problem is that the multi-emulation system adds a layer to the actual hardware which may be very difficult to simulate. Also it is likely to change more often than the hardware. Furthermore, experience with RIKKE indicates that hardware timing problems diminish the value of simulation. This is not to deny that useful results may be obtained from simulation, but complete verification of micro-programs by simulation is unrealistic on RIKKE. Nevertheless the implementation of a good simulator which will fit into the ME-system, appears as a useful next project toward building a convenient environment for microprogramming experiments.

5. Nested Interpreters

Multi-programmed multi-emulation (section 1) is a general notion of which the ME-system may be seen as a special case. Another generalisation, which may be reduced to the ME-system as a special case, is multi-emulation with nested interpreters. The ME-system design started with the problem of organising a system in which a number of microprograms, called emulators, disguise the physical processor to make it appear different at different times. If attention is turned away from the physical machine, and the things emulators do for it, towards the virtual machines they implement, the key notion becomes that of machines implemented through interpretation. Abstracting from the con-

ditioning of microprogramming experience, one sees a program interpreting something understood to be instructions for a processor, thereby realising that processor. To a degree, depending on one's ability for abstractions, it becomes unimportant that emulators reside in control store, which is technically different from other stores, particularly in that its contents actually determine physical processor behaviour; and similarly it becomes unimportant that emulators are written in a particular language, known as micro-code.

In the ensuing generalisation, an interpreter may be written in the code of any machine, and consequently the actual algorithm being executed in such a system may be realised through several layers of interpretation. A code stream is interpreted by an interpreter, which itself is a code stream which is interpreted etc., until in n steps the hardware is reached. It is then seen as a hardwired interpreter, and the fact that it is micro-programmable is logically insignificant.

Derrett and Manthey [13], in a discussion of multi-interpreter systems, state as a requirement to such a system that it must be possible to write emulators in high level languages, and immediately reach the generalised model alluded to above, calling the structure "nested interpreters" for obvious reasons.

This discussion of the relation of Derrett's and Manthey's views to the ME-system must begin by acknowledging that the ME-system design was deeply influenced by discussions of their ideas, a point that will become clear when testing the ME-system against the requirements they formulated for a general multi-interpreter system.

Requirement 1: A program running on one emulator should be able to call a procedure which runs on another.

This is certainly true in the ME-system for the kinds of emulators allowed.

Requirement 2: It should be possible to write a procedure in host machine code (microcode) and call it from a program running on an interpreter.

This requirement is also satisfied. According to taste one may dislike the fact that micro-procedures are forced to look rather like emulators. But they are certainly allowed in the ME-system.

Requirement 3: It should be possible to write an interpreter in a high level language.

This is where our ways part. In section V. 2 we will discuss what extensions to the ME-system are necessary to allow interpreters at any level.

Requirement 4: It should be possible for an interpreter to call a program which runs at any level of interpretation.

The point here is that the procedure call may originate from the interpreter itself, not from the program it interprets. The example given is an emulator calling a garbage collector program externally. In the framework of the ME-system such a call is completely possible, regardless of whether the external procedure is microprogrammed or interpreted. It should be admitted that the inclusion of symbolic references in a micro-program presents a practical problem, the solution of which may require "cheating" the current linkage segment; for example a compiler may include the external references, which really originate from the emulator for which it generates code, at an agreed location in all code/linkage segment pairs.

Requirement 5: The calling sequence for a procedure should be independent of the interpreter nest on which it runs.

If we replace interpreter nest by emulator this requirement is satisfied.

Requirement 6: The representation of a procedure should be, as far as possible, independent of the interpreter nest on which it runs.

The point of this requirement is that the representation of data structures should not be dependent on interpreter nests, and the requirement does not really apply for the restricted kind of interpretation allowed in the ME-system. It will, however, be kept in mind in the development of a suggestion for an extension to the ME-system to allow general interpreter nests in section V. 2.

The reason why the suggested extension towards allowing interpreter nests which is presented in the next chapter has not been implemented is twofold: First, the necessary understanding of the problem was reached relatively late. Second, this author considers interpreter nests to be of very limited value in the context of RIKKE projects. It is evident that if nests of depth greater than one are used for extensive calculations abominable slowness will result. The potentially useful applications of interpreter nests seem to lie where interpre-

tation is used not for raw execution but as an organisational principle in programs which have the common characteristic of essentially consisting of a loop which uses a pointer into some data structure to keep track of how far execution has progressed. We may think of a job control language interpreter or a program which manages a data base by processing transactions. However applications of this kind are a long way from the problems which motivated the presently described project. One interpreter nest of depth two which has our interest is the execution of a host machine program on a simulator written in a high level language (see also section 4).

Addressing Environments

A further difference between the ME-system and the views of Derrett and Manthey has to do with a procedure's addressing environment. They represent a procedure (externally callable program) in a record with three fields:

1. Type of code.
2. Address of code.
3. Environment.

Comparing this to a Routine Control Block it becomes evident that fields 1 and 2 have exact equivalents in the emulator name and entry point address fields of an RCB, whereas the "environment" proposed by Derrett and Manthey has no counterpart in an RCB. The environment is intended as a means (some type of general pointer) of getting at storage which is not local to the procedure. It corresponds to for example a display or static chain in an ALGOL-type run-time system. When an addressing environment is supplied along with the code for a procedure we get a structure which is named a "closure" or in LISP terminology a "funarg".

In a block structured language like ALGOL such a data structure is necessary in order to represent a procedure which may refer to variables at outer block levels, i.e. all procedures except a whole ALGOL program considered as a procedure at block level 0. Similarly in a classical LISP interpreter [21], the recursive EVAL routine works on two arguments, a LISP-expression and an environment. Ruling out the environment component from the data structure describing a procedure restricts the kind of procedure which may be described to a block level 0 program in an ALGOL-type situation or to the initial ex-

pression which is the argument of EVALQUOTE (with an initial NIL environment) in a LISP situation.

The scratch data segments which may be specified in RCBs provide only local storage to procedures. It is therefore clear that the restrictions described above are imposed on the kind of procedures which may be called externally in the ME-system. There are non-trivial problems involved in allowing the more general form ofprocedure proposed by Derrett and Manthey. Consider a call to procedure A declared at block level n in an ALGOL program. In the ordinary execution of the ALGOL program this call cannot occur before control has passed to the enclosing block at level n-1 from where the procedure A may be named. And whether or not A has been passed one or more times by name before being called, an activation record for the enclosing block must be on the stack at the time of call. Now if A is known, say, to a BCPL program, by symbolic name, how do we assure that its addressing environment exists at the time when it is called from the BCPL program? This difficult question which involves general retention must be answered before more powerful RCBs can safely be allowed. Even then, the usefulness of allowing external knowledge of the kind of addressing environment which is essentially an internal feature of a given virtual machine architecture, must be argued. For the initial ME-system it has been deemed sufficient and safe to include symbolic naming as the only means of external addressing.

6. Generalised Transfer of Control

The super-stack mechanism which is employed to represent saved virtual processor state vectors and externally passed parameters in the ME-system has caused a subroutine call and return discipline to be imposed on all external control transfers. Programming applications are conceivable, indeed some are mentioned in the existing literature, in connection with which the availability of more general control transfer primitives is desirable. We may think in particular of coroutines. There are two reasons why such transfers of control are not supported by the ME-system.

1) The super-stack neatly encapsulates all the information associated with external transfers of control; in this respect it functions as a simple and compact record of execution for a process. This simplicity is not merely con-

venient for the implementer, it also serves to lessen the likelihood of errors as compared with a more complicated situation, not only the likelihood of system implementation bugs, but also of user errors arising later because of a scheme which might be difficult to comprehend. Allowing a very general transfer primitive, such as suggested in [22], would certainly force us to adopt a more complex structure for the (system level) record of execution; the simple linear organisation of "activation records" in the super-stack would have to be given up. In this context it is also significant that the applications which motivated the design of the ME-system (see chapter I) are naturally accommodated in the framework of a subroutine discipline.

2) We mentioned in section 1 of this chapter that individual emulated virtual machines may allow multi-programming. Similarly, individual virtual machines may support any conceivable kind of control transfer internally. Only external control transfers, which are assumed to be relatively rare, are forced by the super-stack structure to be subroutine calls or returns.

As far as coroutines are concerned it is suggested that these may naturally be understood as cooperating pseudo-parallel tasks (processes). In [4] it is described how process synchronisation primitives may be used in a natural way to achieve coroutine sequencing. We therefore suggest that an extension to the ME-system allowing multiple parallel tasks at the system level may also be used to support coroutines if at some stage this should be considered desirable. Such an extension is described briefly in section V. 1.

V. SUGGESTED EXTENSIONS TO THE MULTI-EMULATION SYSTEM

In this chapter we suggest possible future extensions to the ME-system in two directions: first an extension allowing multiple tasks to execute in pseudo-parallel at the system level (as opposed to internally in emulated machines) is briefly indicated, and then the data structures which are necessary to allow general interpreter nests are considered at more length.

1. Multi-tasking

There is only one physical processor directly involved in the ME-system, namely RIKKE-1. Coordinating the use of RIKKE and MATHILDA is considered a separate problem, left to individual virtual machines which may utilise MATHILDA as well as RIKKE. Multiple tasks (or processes) cannot therefore be executed in true parallel. Only pseudo-parallel or conceptually parallel tasks may be realised. Such organisation is used in computing applications to provide time-sharing among several users of the same processor or to facilitate efficient batch processing overlapped with i/o taking place via channels. Neither of these uses are relevant in the RIKKE situation (see section IV. 1). However there exist algorithms which may naturally be divided into (conceptually) parallel tasks or organised as co-routines which may also be considered as pseudo-parallel tasks. For this reason we give some hints as to how multiple tasks may be represented in an extension to the ME-system and how task switching may be accomplished.

In the Burroughs B6700 [4] the state of each task is represented and completely defined by the task's stack. Multiple tasks are allowed, the tasks belonging to a given job being organised in a tree structure, reflecting a parent-offspring relationship between tasks. Task switching is accomplished by adjusting a few machine registers which point to the stack. Similarly, in the ME-system, the state of a process is at least to an extent defined by the super-stack. It is therefore suggested that multi-tasking could be achieved by allowing multiple super-stacks, one per task.

Drawing on the analogy to the B6700 situation we further state that switching from task A to task B could be accomplished by

1. Saving the current virtual processor state of A on A's super-stack as if A were making an external call.

2. Switching the system registers which point to A's super-stack to point to B's super-stack.

3. Effecting a RETURN; assuming B's virtual processor state had previously been saved as was done for A in step 1, this would cause B to continue naturally.

For process synchronisation it seems natural to extend the analogy and use a vector of task attributes kept in the bottom of the task stack. These attributes may then be operated on by a suitable set of synchronisation primitives. We offer no details on this point.

So far, essentially the same mechanisms have been recommended as are employed in the B6700. However, the B6700 stacks also contain all information needed to reference data, given the block structured architecture of that system. The same is not necessarily true about the super-stack. In the B6700 two sibling tasks, initiated by the same parent task, share a portion of their stacks, namely some part of the parent's stack. This scheme provides data sharing, and also protection of private data. A similar mechanism could not be had for nothing in an extended ME-system, but would require a way of representing task ownership of segments, segments being the most appropriate counterpart of the activation records in the B6700 stack for purposes of data reference. However the only problem is protection failure in the case where a task requesting a virtual memory access uses a segment number belonging to another task. Correct access is certainly possible within the segmentation framework we have presented in section III. 1.

No further investigation into the problem of address space will be undertaken. Hopefully the point has been demonstrated that the ME-system design is extensible in the direction of multi-tasking, although much detailed work remains to be done before such an extension can be realised.

2. Structures for Nested Interpretation

We now return to the concept of nested interpretation which was introduced in section IV. 5 to discuss what structures are necessary to accommodate

general nests. Two main questions must be answered.

1) How should storage be organised for code and data areas of the necessary layers of interpreters, given that what is data to one level, is really code to the next one up?

2) How can primitives for external entry and return be implemented? In other words, how is the multi-layer interpreter nest structure which is necessary to interpret an actual useful algorithm initialised and set in motion?

Derrett and Manthey state that the second problem may be solved by an iterative mechanism which initialises interpreter states at all levels in a nest from top to bottom before passing actual microprogram control to the emulator which carries the weight of the nest. In fact return is much simpler than entry, and can be achieved exactly as in the ME-system, since the state vectors of all interpreters, except the bottom-level emulator, will have been saved in virtual memory, in exactly those locations where they will be needed. Problem 1 is not treated by Derrett and Mantehy, but since it has always appeared to the author as a significant conceptual obstacle, it will be examined in the following.

Consider a program Fred *) running on the interpreted (hypothetical) XCODE machine. It is not important at what level of an interpreter nest the XCODE interpreter sits; indeed for the sake of the generality of this discussion it should not be. The storage seen by the XCODE machine may naturally be divided into a code area and a data area. The code area contains Fred, in fact those XCODE instructions into which Fred was once compiled. The data area contains a record of execution for Fred and other data structures which Fred is processing. Values held in XCODE machine registers - these are data at the next level down - provide a means of interpreting Fred's data area, possibly they point to a stack which describes the computation state. Among these values there will also be a program counter, pointing into Fred's

---

*) In all the discussions about multi-interpretation, that program for whose sake the whole interpreter nest is constructed, the only program which is not itself an interpreter, has been called Fred. The fact that the name Fred is ordinarily used for humans made it natural to nourish love/hate feelings toward the subject.

code area. The situation is illustrated in Fig. 10. So far, all is well and simple enough.

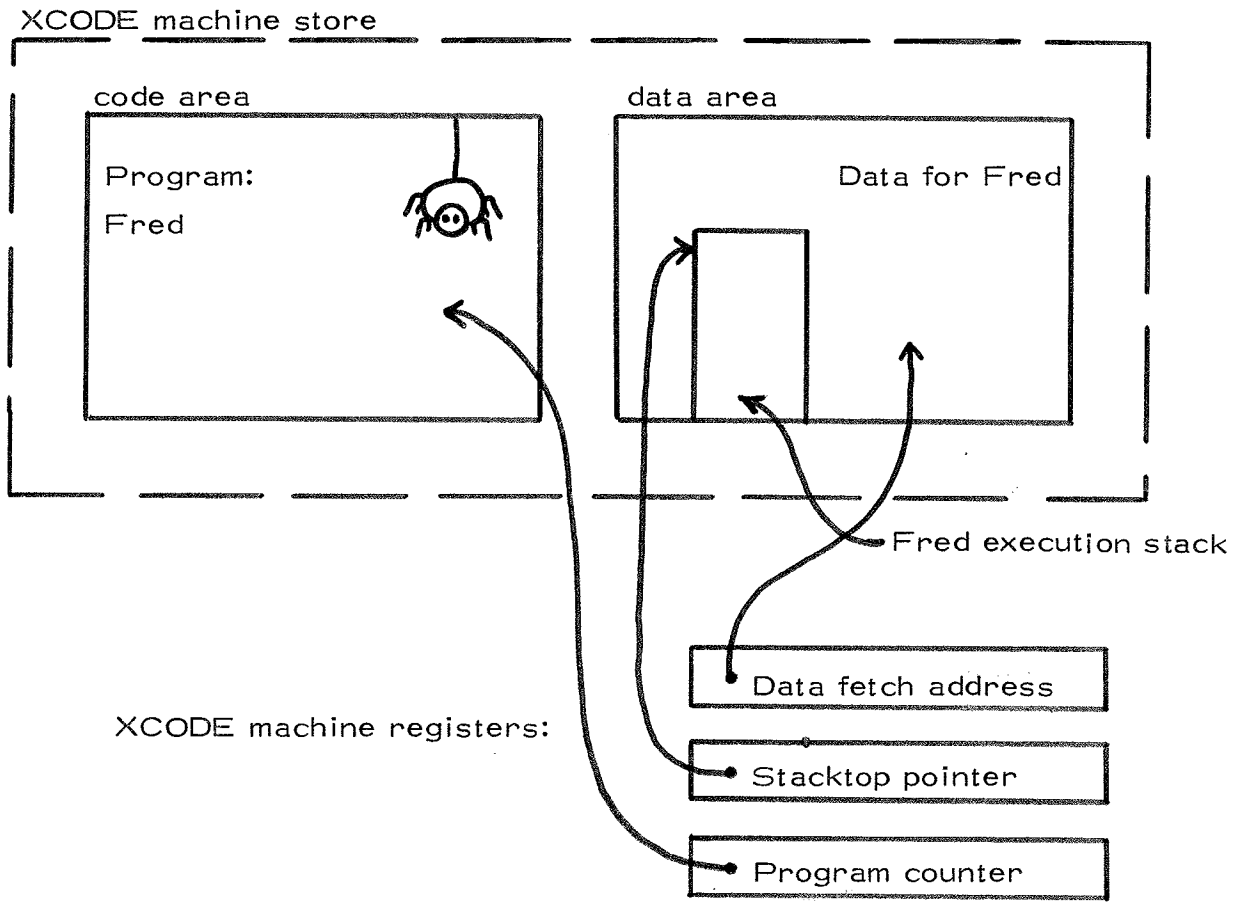


Fig. 10

Now see what happens if we replace Fred by an interpreter, say the YCODE interpreter, written in XCODE. Inside the code area of the XCODE machine store we now have the code of the YCODE interpreter. In this situation we know more about the contents of the data area than we did in the case with Fred, since we have some knowledge about what the YCODE interpreter is doing, whereas we explicitly did not care what Fred was doing. The YCODE interpreter is realising the YCODE machine; let us assume that the architecture of the YCODE machine, like that of the XCODE machine, involves some registers, which hold the usual state vector, some storage organised into a code area and a data area, and some functional units. In the realisation of the YCODE machine by XCODE programming, the functional units, with instruction sequencing foremost among them, are mapped into the XCODE program which we called the YCODE interpreter. The other three components, registers, code, and data storage, must be mapped into the data area of the XCODE machine. This is illustrated in Fig. 11.

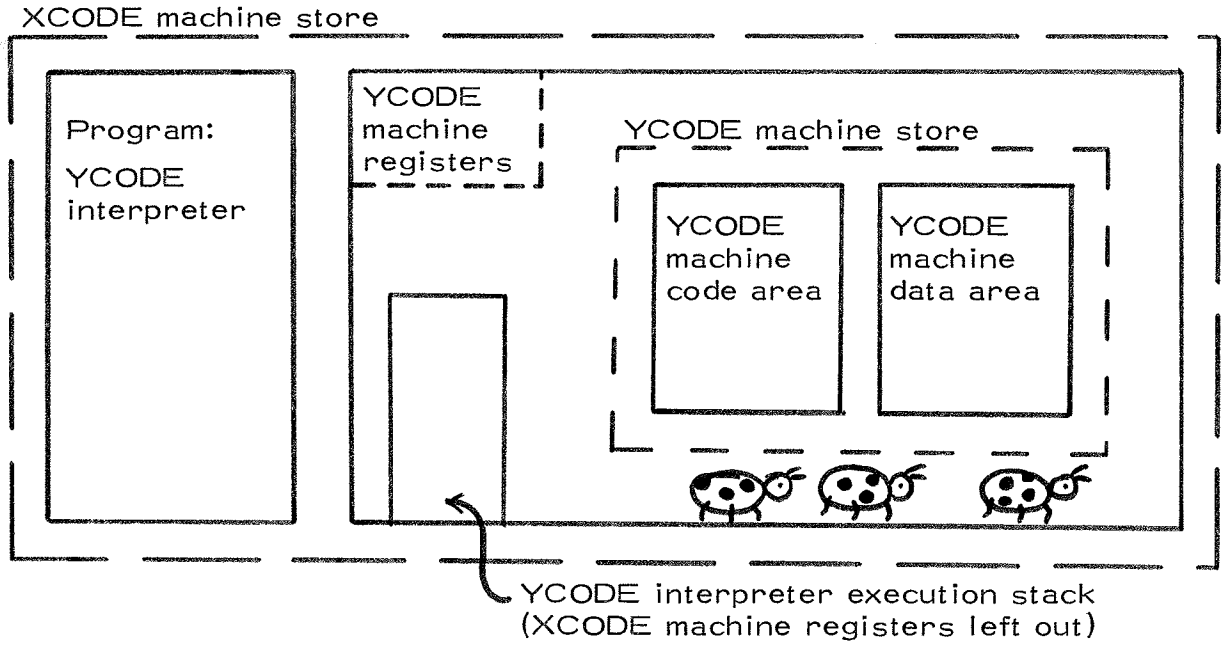


Fig. 11

As our next complication we rewrite Fred in YCODE and draw a picture of Fred running on the YCODE machine which we have just described. So far we have assumed the same things about the YCODE machine as we have about XCODE; let us continue in this path and assume the YCODE machine is also some kind of stack machine; thus part of the YCODE machine's data area holds an execution stack for Fred, while other Fred-data are elsewhere in that area. Fig. 12 is then a special case of Fig. 11.

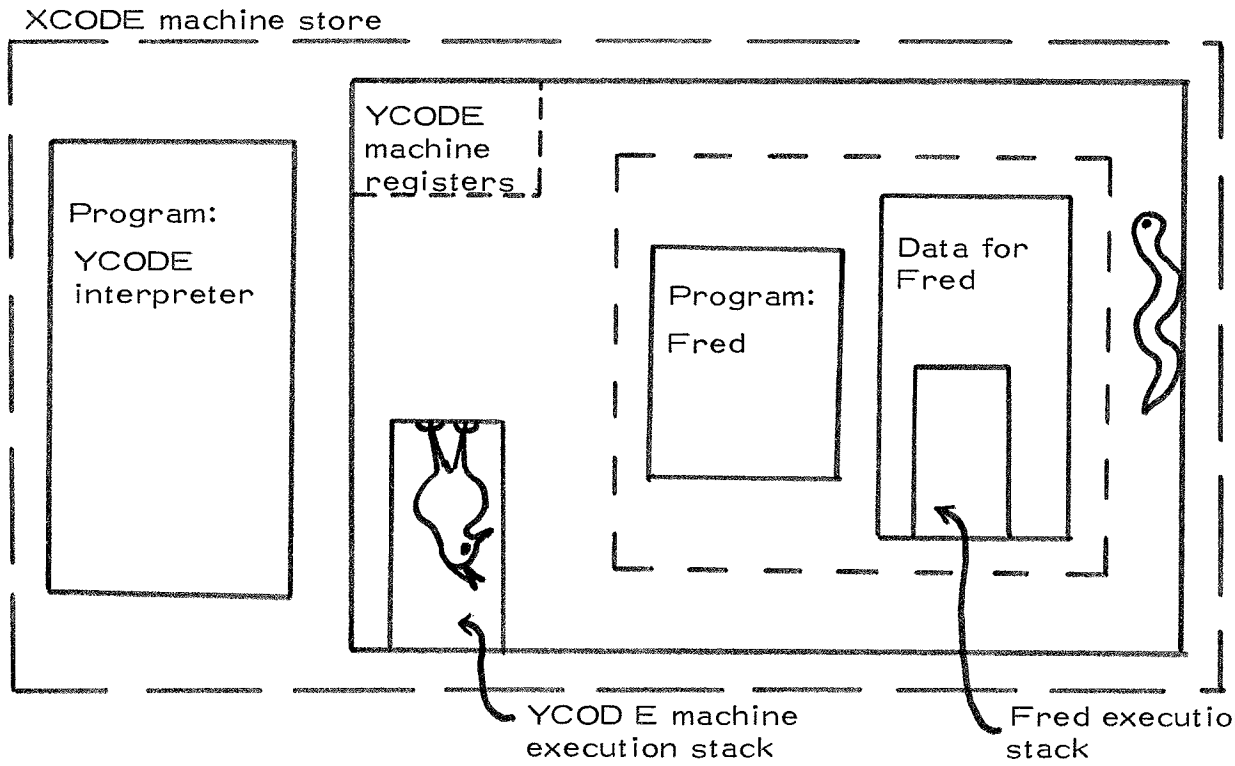


Fig. 12

Instead of now replacing Fred by the ZCODE interpreter to see what happens, we jump to the conclusion that interpreter nests involve a nesting of storage areas, which may be recursively defined by the equation: data for i'th level machine equals (processor state plus data plus code) for i+1'st level machine, the numbering being from hardware level towards Fred.

With this understanding, let us think about the compiler which produces an executable (interpretable) version of Fred from some source text, in the light of requirement 6 of Derrett and Manthey, that the representation of a procedure should be independent of the interpreter nest on which it runs. Such independence would obviously be advantageous for compiler writers and updaters. Comparing Figs. 10 and 12, we see that the requirement is certainly not satisfied for Fred, at least not as long as we interpret the pictures naively and think of boxes depicted within boxes as representing containment. It would be unreasonable to ask a compiler to produce code for a procedure embedded in a set of chinese boxes, which are irrelevant for that procedure per se. One might also consider the task of placing code inside the chinese boxes as belonging to a loader. Special purpose loaders are undesirable in a general system so the situation remains unsatisfactory.

The troubling phenomenon is the recursive embedding. However, with inspiration from the B6700 philosophy, we may choose to replace the physical embedding by descriptors of external storage units. In the B6700 this is achieved by means of specially tagged data descriptors. In the ME-system, the virtual memory with <segment number, word number> addresses fits the problem as a glove does the hand, providing a simple means to represent inside one segment the fact that some other segment is in a sense a part of it, or rather that ownership of and access to the former segment implies access also to the latter.

Turning attention for a moment to the registers of interpreted machines, which have so far been kept in a corner, we realise that there is really no reason to draw them separately or otherwise to think of them as separate from the record of execution of the interpreter program. Referring explicitly to Figs. 11 and 12 we may as well draw the YCODE machine registers within the YCODE interpreter execution stack. To Justify this, we consider an

interpreter written in a block structured language; the registers, being of elementary data types, will then naturally be mapped into local variables at some level, and therefore kept in the stack. Continuing in this line, descriptors for code and data areas are naturally kept in registers of virtual machines; at least they are themselves simple and may be kept in the record of execution. Having thus effectively factored the embedding out, we draw a new picture of Fred running on YCODE on XCODE, Fig. 13. The arrows represent descriptors; in the ME-system they may be implemented simply as segment numbers.

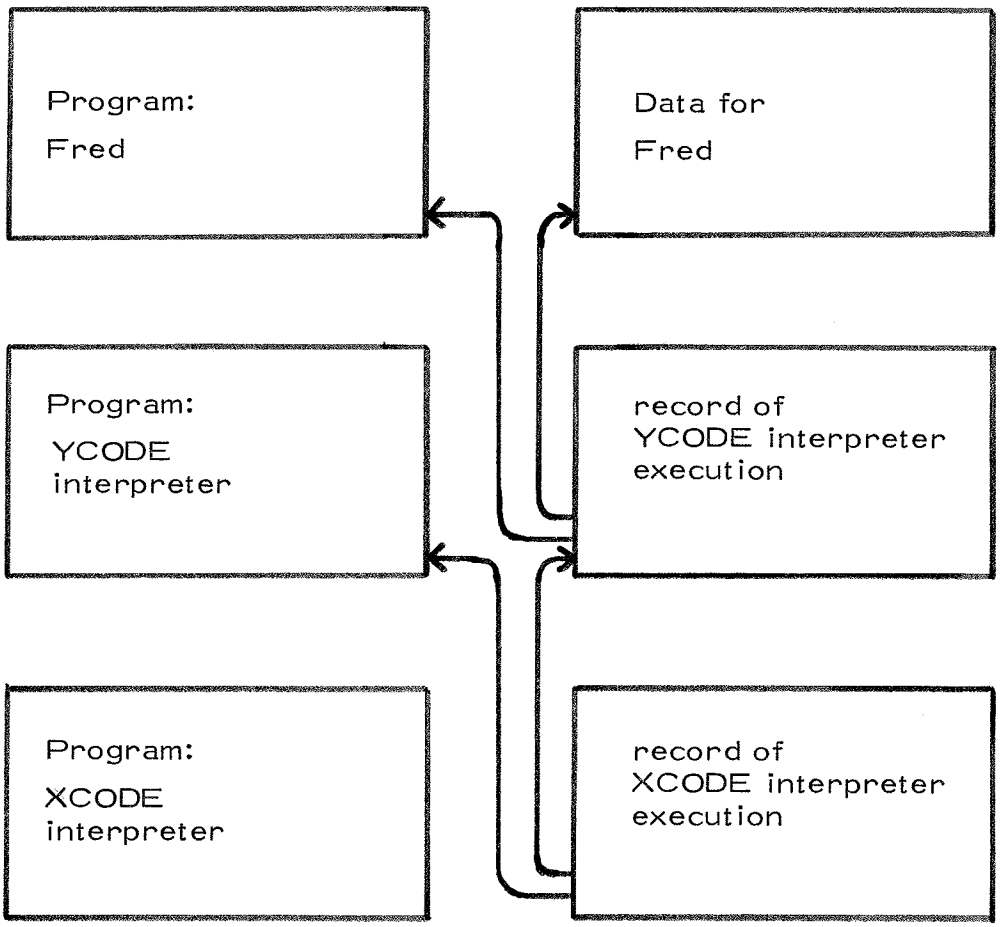


Fig. 13

Fig. 13 is much nicer than Fig. 12. The addition of a new interpreter in the nest no longer renders the picture more difficult to discern, it only makes it taller. It should be possible to devise an implementation which reflects this simplicity. The important characteristic which is apparent from

Fig. 13 is that the representation of Fred, in terms of a code segment and a data segment in no way depends on the nest beneath it with respect to internal structure; excepting of course, that Fred must be in YCODE format. Thus an implementation which reflects Fig. 13 will satisfy requirement 6 of Derrett and Manthey.

Let us consider in more detail how an interpreter as depicted in Fig. 13 may be represented and built. Apparently it consists of two segments placed on the same level in the drawing, namely a code segment and a data segment. These segments must be in a format which is understood by the interpreter immediately beneath. If that interpreter is the hardware, in other words if we are dealing with a microprogrammed interpreter, the code segment will be a microprogram kept in control store, and the data segment is likely to be placed in host registers. For higher level interpreters ordinary segments of the kind introduced in section III.1 may be used. The contents of an interpreter code segment (like those of any other code segment) need not change during the life of the interpreter program; it may be created by compilation of the interpreter program source text. The data segment is in a sense more complicated since it changes with interpretation of a program on the virtual machine in question. We will,however, assume that the state of the data segment as it looks when execution of an interpreted program begins is well defined and independent of the particular interpreted program in question, indeed we assume that the contents of the data segment at such a moment are known when the interpreter is created except for two items:

1. Program counter, a pointer into the interpreted code segment defining where execution is to take place.
2. Data segment pointer, the number of the segment which contains the data area of the interpreted program.

Item 2 and the segment number part of item 1 are represented as upward directed arrows in Fig. 13.

The assumption that these items are the only variable pieces of information needed to initialise the processor state of the machine being realised by an interpreter before it may be started is equivalent to the assumption that a new-entry module of a microprogrammed emulator may be defined which can

initialise the emulator state properly. With higher level interpreters we are in the fortunate situation that the initial state may be represented once and for all in a data segment "template", because data segments in the filing system integrated with virtual memory may have permanent existence as opposed to states represented in host registers.

We now turn to problem 2 given at the beginning of this section, i.e. the problem of ENTERing a routine which runs on an interpreter nest. It is clear that ENTER must set up the whole interpreter nest, working its way from top to bottom. At each step (level) the code segment part of an interpreter is created by making it known unless it is already known, and the data segment is created by copying the template which is stored permanently in the filing system and then inserting the two above mentioned items pointing to code and data segments at the level above. For this scheme to work a table is needed with information about each interpreter, including:

1. Name of the virtual machine (interpreter) on which it runs; if the interpreter is microprogrammed, this is "host machine".
2. Symbolic addressing information for its code and data segments.
3. The relative location within the data segment where items 1 and 2 from above should be inserted at initialisation time.
4. The relative location within the code segment where the interpreter should be entered.

We are now in a position to indicate concisely how ENTER could be implemented in the current framework by presenting a version written in pseudo-BCPL. As always the argument for ENTER is an RCB where we have replaced the emulator name field by an interpreter name; this is merely a change of name and does not imply a more complicated field. The present version of ENTER uses a recursive routine EnterOneLevel which is called once for each level in a nest. Some undefined simple routines which access the information in 1-4 above are called. The program should be self-explanatory.

```
let Enter(RCB) be
$( let DS=MakeScratchSegment(RCB↓SCRATCHSEGSIZE)
   EnterOneLevel(RCB↓ENTRYPOINT, DS, RCB↓INTERPRETERNAME) $)

let EnterOneLevel(EP, DS, Interpreter) be
$( E if RunsOn(Interpreter)="host machine" do // as described
        //in section III.2
else  $(let DS1=InitTemplateCopy(Interpreter, EP, DS) // inserts
        //EP and DS at locations found in the interpreter table
        and Offset=FindEntryPoint(Interpreter)
        and CS=MakeIntrprtrKnown(Interpreter)
        EnterOneLevel(<CS, Offset>, DS1, RunsOn(interpreter)) $)E
```

At this stage it should be pointed out that certain assumptions about the interpreters which may occur in interpreter nests are implicit in the preceding discussion.

1) An interpreter sees its storage as divided into a code area and a data area; both of these are ordinary segments.

2) An interpreter must support instructions to read and write a word referenced by general virtual address; this is necessary in order to pass storage references downward in a nest.

3) Since the concept of a current linkage segment in the ME-system implies a unique linkage segment, which is associated with a code segment being executed, a problem arises when several code segments are being executed in a nest at the same time, namely the target program as well as some number of interpreters.There seems to be a need for a linkage segment per code segment. The fastest way out of this dilemma is to declare that interpreters must live without linkage segments. In so doing we cannot satisfy requirement 4 of Derrett and Manthey that interpreters should have the same rights as any other programs to make external calls. It should be fairly simple to provide an ad hoc solution to this problem by the creation of phony RCBs for the programs interpreters wish to call. It may also be possible to develop a more general model for an instruction pointer which will accomodate procedure calls from interpreters in a completely general fashion. Perhaps an instruction pointer should be considered a vector of instruction pointers at different levels of a nest, somehow similar to an ALGOL-type runtime display,

although this analogy should not be stretched too far. However more study and better understanding is necessary before we can decide whether a sound model can be found and also whether it will really be very useful.

The point of this section has been to provide an understanding of the data structures involved in nested interpretation, and to indicate a way in which the ME-system may be augmented to support applications of nested interpreters. We have not been able to do so without making certain (restrictive) assumptions about interpreters, nor is it claimed that the final word has been said about nested interpreters, or for that matter that they are worth a lot of effort.

VI. USER MANUAL

0. Introduction

This manual is intended for the user of the RIKKE multi-emulation system. By user we mean "immediate user", i.e. a person who undertakes one of the following tasks:

1. Implementation of a virtual processor by microprogrammed interpretation.
2. Construction of other - special purpose - microprograms which must be interfaced to the multi-emulation system.
3. Writing a compiler to translate programs in a high level language into some corresponding virtual machine code.
4. Writing a program distributed across emulator boundaries, i.e. a program consisting of parts running on different virtual machines.

The writer of simple programs which may be run within the boundary of an individual virtual machine may be seen as a user of the multi-emulation system one step removed. It should not be necessary for users of this category to be directly acquainted with this manual. However the material on system services provided via external entry points into the BCPL system should be available to a general user community, probably propagated through manuals for individual language systems. People with tasks of type 4 above may also be sufficiently served by manuals for the individual virtual machines (most likely this means programming languages)in question, particularly the information on how external calls are supported by a given emulator/compiler couple. All users should be familiar with the BCPL system from the console operator's point of view. For this purpose we refer to the BCPL system manual [16].

The material in this manual cannot stand alone. To understand what goes on, the user must also have read chapters II and III. Before undertaking a microprogramming project in the multi-emulation framework it is of course also necessary to be familiar with the actual hardware, microinstructions, etc. of RIKKE-1 [15]. In fact we suggest the following schedule for the learning, thinking, design effort, and actual implementation involved in such a project:

1. Get interested in some problem, e.g. building an ALGOL machine.
2. Get to know and understand that problem.
3. Get the idea of using RIKKE for an implementation (someone comes along and points out its existence).
4. Get to know RIKKE/MATHILDA.
5. Read chapters II and III.
6. Carry out a detailed design.
7. Write microprograms (or other programs) using this manual as a handbook. *)

Roughly speaking the contents of the manual fall in three parts: information useful for the microprogrammer, information primarily intended for the compiler writer, and finally a section (section 9) listing those system routines which may be called externally. The first part consists of sections 1-6. It begins with a discussion of how emulators should provide input and output facilities. Then follow the detailed rules for sharing the microprogram-accessible registers of RIKKE-1. Sections 3 and 4 discuss questions related to the interfacing of user and system microprograms. The problem of organising microprograms too large to fit in the user microprogram part of control store is treated in section 5. Section 6 is useful for the programmer wishing to access segments directly, without using READ and WRITE. Whether section 7 should be seen as belonging to the first or the second part of the manual depends on the user's approach. It defines the common data formats recommended particularly for information in the super-stack. Section 8 is for the compiler writer. It discusses the formats for code and linkage segments.

One convention should be presented at the outset. At various points in the following sections information in terms of actual numbers is shown, e.g. control store addresses of system entry points. Since these numbers are likely to change with program adjustments, they are represented by BCPL-like manifest names, i.e. by strings of upper case alphabetical characters. It is the responsibility of the system upkeeper to maintain a list of manifest values and make this list available to users.

---

*) It is logically ironic that the schedule is presented as part of step 7, but also logically impossible to present it at the head of the schedule.

1. I/O and Files

One problem facing the emulator designer is to provide a mechanism for input and output in the emulated machine. This will often be one of the odd problems, not related to the real purpose of an emulation project, particularly when the purpose is to implement a given high level language. It will often be desirable not to have to think about microprogramming i/o. Nevertheless it is necessary to be able to transfer information between programs running on the emulated machine and external sources or destinations.

The mechanism provided in the multi-emulation system to assist with input and output to and from user programs outside the OCODE machine is the filing system. System routines are available to provide access to symbolically named files (see READFILE and WRITEFILE in section 9), and routines to read or write individual words from/to files may be called at the micro-level (see NEXT, ENDOF, and OUT in section 3, and also RESET and CLOSE in section 9).

Using programs callable from the BCPL system command interpreter, symbolically named files may be read in from or output to peripheral devices.

In the initial design external access has not been provided to those streams in the BCPL system which handle peripheral devices such as the line printer, the paper tape reader and punch, and the console terminal. It will be simple to provide these streams if they are deemed desirable at some later time. In the initial implementation users wishing to do i/o directly with these devices must write their own microprograms (or interfaces to the i/o nucleus [20]). Such practice is not recommended, except for one instance: Emulators for machines intended to accommodate programs which undertake a dialogue with the operator should contain in their repertoires simple instructions for reading and writing one character to or from the operator's console. Such instructions are easy to provide by microprogram. Otherwise, the recommended approach is to spool i/o on files.

2. Conventions for the Use of RIKKE's Registers

RIKKE-1 contains a myriad of registers: working registers, mask registers, shifter registers, pointer registers, register save groups etc. The

programmer intending to fit his microprogram into the multi-emulation system framework needs to know which registers he may use at what time. Given the large number of registers it is unreasonable (as well as impossible) to save the complete register status of the physical RIKKE when transferring control from one microprogram to another. Instead it is left to the individual emulator to save those register values which are important to it.

Normal state

The normal state of RIKKE-1 is defined by a certain set of values which are present in a collection of important registers. When control is transferred from the system to a user microprogram RIKKE will always be in the normal state, and this must also be the case when control goes back into the system. The following list defines the normal state: (throughout the discussion of registers we use ALGOL-type subscripts within square brackets to indicate a given register in a register group)

MA[0] = PA[0] = LA[0] = LB[0] = all ones (no mask)
MA[1] = PA[1] = all zeroes (full mask)
MAP = PAP = LAP = LBP = 0
VS(0)S = AS(0)S = DS(0:1)S = VS(15)S = AS(15)S = DS(14:15)S = 0
(shifter registers set to logical left and right shift)
AS(V)S = 15
WB[0, 1... 15] = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, -1
(it is convenient to have these small constants at hand)
CUALF = A+B (relative addressing used in micro-sequencing)

Register Categories

The remaining conventions for use of registers are explained by means of a principle of categorisation. The registers used by a given microprogram may be divided into three categories:

1. Those which are permanently allocated to that microprogram, i.e. by convention no other microprogram may overwrite them.
2. Those not in category 1 which also have permanent significance for the microprogram: they hold part of its state vector.

3. Those only occupied temporarily by one or more parts of the micro program, parts which do not contain branches to other micro-programs.

We suggest that the emulator writer should have these categories in mind, so that the microprogram state at external branch points always agrees with the chosen categorisation. It should be clear that no overlapping must occur between category 1 of one microprogram and categories 2 and 3 of any other. Similarly it is clear that the registers in category 2 are those which must be saved on the super-stack during external calls. In general only the register values in category 1 can be expected to remain unchanged while another microprogram has control. However it must be possible to call certain system microprograms without first having to save the whole register state away; this is particularly true for the primitive used to save register values on the super-stack (Push), but obviously also for READ and WRITE to name just a couple. Thus there is a need for particular conventions specifying, for each possible call to system microroutines, exactly which registers are in categories 2 and 3 of the relevant part of the system. With such information it suffices for a user microprogram to save its values outside of the registers known to be endangered before making a call to the system. For example, working registers may be used for such a purpose.

The registers permanently allocated exclusively for system use (category 1) are:

WA groups 0, 1, 11, and 12; WB groups 4-7.
MB[12-14], and LA[14-15].

Category 2 may be considered empty for the system; all the registers of permanent significance are in category 1, or are part of the normal state.

Category 3 depends on the routine in question. However all system microroutines except the super-stack primitives may involve calls to BCPL routines, i.e. the OCODE machine, and therefore we may define category 3 once and for all to be the following:

WAU, WAG, WBU, WBG, DS, VS, AS, LR[0-3], ALF,
SA, IA, IB, OA, OB, OC, IBD, OBD, OCD, ODD,
MSA, MBP, KD, KC, LRIP, LROP;
The RA-stack may have its contents changed arbitrarily, RAP may also change;
The RB-stack is used to return to the calling microprogram by the specification RB+1 in the sequencing field of the last microinstruction of each system routine; the RB-stack contents will not be violated by system routines; the idea is that the RA-stack is used locally, the RB-stack externally;
In each of these save register groups: WAUS, WAGS, WBUS, WBGS, CBS, BSSG, PGSG, the register pointed at by the group pointer is used; the remaining registers in the groups remain unchanged, and so do the group pointers.

For the super-stack primitives category 3 is specified in section 3 for each entry point separately.

We remark that only for the system is it of interest to know category 3. For all external calls, except to the system, category 2 registers should be saved on the super-stack. Category 1, however, will be the object of battle between future emulator owners. It is suggested that it will be wrong to include working registers in this category; apart from the chunk already allocated to the system these should always be at the disposal of whatever microprogram possesses control.

3. System Microprogram Entry Points

The purposes and workings of system facilities such as super-stack primitives, access to virtual memory and file streams, link resolution, and external calls and returns are presented and explained in chapter III. To invoke any of these services control must pass from a user microprogram into the system microprograms. A specification of system microprogram entry points is therefore needed. For each of these we give in the following list its name, the parameters which must be passed in RIKKE registers, and a brief description. Except for ENTER and RETURN all the entry points represent subroutines which return by the specification of RB+1 in the sequencing field of the last micro-instruction.

The format used for each entry point is

EPNAME[a_1 in R_1, a_2 in R_2,...]; Description,

where EPNAME is the manifest name of the entry point; a_1, a_2,... are names of parameters, the use of which is explained in Description; and R_1, R_2,... are the RIKKE registers in which the parameters must be placed. Values which are returned to calling microprograms may be specified in a similar format.

The list begins with the super-stack primitives. They have been introduced and explained in section III.2. For each of these, the registers in category 3 (see preceding section) are listed. They all assume LRIP = LROP at entry. The common value is denoted LRP.

PUSH[x in VS]; x is pushed onto the super-stack.
Category 3: WAU, WAG, AS, LR[LRP], ALF, MSA, OA, WBU.

GPAR[i in LR[LRP]]; GetParam[i].
GREG[i in LR[LRP]]; GetRegister[i].
Both of these return a result in IA.
Category 3: WAU, WAG, ALF, AS, MSA.

PRES[i in LR[LRP], x in VS]; PutResult[i, x].
Category 3: WAU, WAG, AS, ALF, LR[LRP, OA, MSA, WBU.

MSF[ ]; MarkStackFrame.
Category 3: WAU, WAG, AS, LR[LRP], ALF, MSA, OA, VS, WBU.

UMSF[ ]; UnmarkStackFrame.
Category 3: WAU, WAG, VS, MSA.

Now follows the remaining entry points. For these category 3 is specified above (in section 2).

READ[SN in AS, WN in VS];
reads the contents of virtual address <SN, WN> and returns it in LR[0]; at exit LRP = 0.

WRITE[SN in AS, WN in VS, x in LR[0]];
assumes LRIP = LROP = 0 at entry; writes x into the location with virtual address <SN, WN>.

LINKREF[WN in VS];
reads the link reference at location WN in the current linkage segment, resolves it if it is unresolved; the virtual address referred to is passed back [SN in AS, WN in VS].

ENTER[SN in AS, WN in VS];
ENTER passes control to the routine described in the RCB at location <SN, WN>. ENTER assumes the calling emulator has done a MarkStackFrame and then pushed some register values including at least its own name on the super-stack, so that the name is now at the top of the stack. Notice that ENTER does not assume that an RCB is necessarily placed in a linkage segment; thus the user may build RCBs and place them anywhere in virtual memory. This possibility is particularly useful for calling micro-procedures which are not automatically associated with RCBs in linkage segments. See also the description of RCB format in section 8, and of information passed to user microprogram new-entry points in section 4.

RETURN; effects an external return.

NEXT[S in AS]; gets the next word from the stream S and returns it in DS. Section 9 tells how to get hold of S.

OUT[S in AS, x in LR[LROP]]; outputs x along stream S.

ENDOF[S in AS]; returns a boolean value in DS, true if the end of stream S has been reached, otherwise false.

4. Interfacing User Microprograms to the System

This topic is discussed extensively in section III. 2. It therefore suffices here to state a few simple facts.

Emulator names and emulator segment numbers, both of which are merely small integers, are assigned for new emulation projects by the system maintenance staff.

At new-entry into an emulator the following values are passed in registers: the virtual entry point address (PC-field) from the RCB handed to ENTER is passed [SN in AS, WN in VS]; LR[0] contains the segment number of the data segment defined in the RCB, or 0 if this option is not used (0 specified).

At re-entry no values are passed to the emulator.

5. Use of Control Store

The resident system microprograms, including the OCODE emulator and i/o-nucleus, occupy approximately 750 words of control store (micro-instructions). We denote the exact number by SYSCSLENGTH. Approximately 1300 words out of the 2K of control store remain for user microprograms. User emulators to be included in the system's table of emulators should always be micro-assembled starting in location SYSCSLENGTH in order for the emulator swap mechanism employed by ENTER and RETURN to work. For users whose microprograms fit in the available 1300 words this is all there is to know about placing the microprogram in control store.

Problems arise only in connection with very large microprograms. As RIKKE hardware (micro-instruction sequencing mechanism) does not allow the building of a control store address space larger than the physical control store, all overlaying must be achieved under explicit program control. The following scheme is suggested for organising large microprograms:

The program is organised into a central module and some overlays. The central module contains all externally visible entry points and all external branches as well as a central part of the algorithm in question which invokes the overlays, for example an instruction fetch and decode loop. The overlays may for example contain rarely executed instructions in a virtual machine, or instructions which require bulky microcode. Now, to conveniently invoke the overlays they may be given status of emulators in the system, thereby enabling the use of the emulator swapping routine built into the system's ENTER and RETURN. Before ENTERing an overlay it is necessary to build a phony RCB containing its emulator name and then go through an ordinary calling sequence, excepting that it should not be necessary to save the processor state. The re-entry point of the central module, of course, should be prepared for the subsequent return. Another way to achieve the

overlaying is to handle control store swaps explicitly within the central module. This, of course, reduces the amount of control store available for each overlay, since some part of the central module must be resident as long as any part of the microprogram is active. Perhaps more importantly, the approach described above also saves the user microprogram from dealing explicitly with segments containing microprograms. On the other hand, if swapping is handled within the central module, there is no bothering with super-stack complications.

6. Segments

Each segment has a number of attributes recorded in its heading and, when it is known, also in its segment descriptor. Those potentially of interest to the user are:

1) Segment size. This is specified when a segment is created. A segment with size s contains words addressed by the integers 0 through s.

2) Segment type. The type word has bits for various purposes:

   a. A segment is either a main store segment or a wide store segment. It will always be loaded in the same physical store. The creator of a segment determines this attribute. Scratch data segments created by ENTER are always wide store segments. The maximum size of main store segments is 10K, the maximum size of a wide store segment is 40K.

   b. Another bit is the linkage segment indicator.

   c. A segment may be write-protected, e.g. a pure code segment.

3) For segments locked in core so that they may be directly accessed by user microprogram the core base address is also of interest.

The system routines described in section 9 may be used to manage segments, including alteration of segment types.

7. Common Data Formats

The need for common data formats was argued in section II.2 and detailed conventions promised in section III.2. These are not complicated. Simplest of all is the type boolean. A boolean value is held in one 16 bit word, in which bit 15 is significant. 1 means true, 0 false. Integers are also simple;

they are represented in 2's complement in single 16 bit words, so that the value range is from -32768 to +32767.

Characters are represented in ASCII code, with the parity bit irrelevant. Thus each character is encoded as an integer in the range 0 through 127.

Since characters occupy 7 bits each they may be packed two per word for character strings, and so they are. A string of length n occupies n/2+1 consecutive words; the first word contains n and the first character, word two contains the second and third character, etc. Odd numbered characters occupy bits 0-6 of the words in which they sit, even numbered characters occupy bits 8-14; whereas n occupies bits 8-15 of the first word. Thus the length of a character string must be in the range 0 through 255. For strings of even length, the unnecessary byte in the last word must contain 0.

Fig. 14 shows the super-stack with 3 parameters pushed: the integer 37, the string "banana", and the boolean false. It is clearly necessary to know the number and types of the parameters in order to interpret them sensibly.

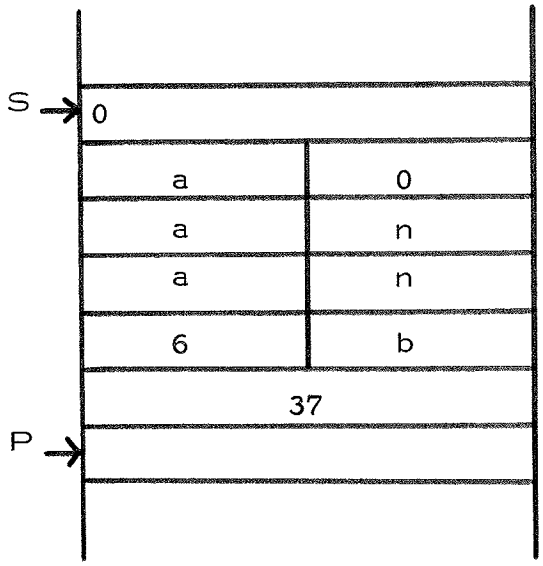


Fig. 14

Character strings and integers are also used extensively in connection with linking. See the next section.

8. Formats of Code and Linkage Segments

This section is intended for the compiler writer who wishes to utilise the system facilities for external calls and data references in a natural way. It describes the formats code and linkage information must satisfy in order to make full use of the system.

We will assume a virtual processor design which separates code and data into different segments, so that the contents of a code segment may be invariable after it has been created by a compiler. In this situation we know from section III.3 that all the variable information associated with loading may be kept in a linkage segment. Code and linkage segments exist in a one-to-one correspondence. A compiler should always produce one of each at a time. They will have the same first name in directory entries, whereas the extension should be "CODE" for code segments and "LINK" for linkage segments.

As the linkage segment is accessed for all external references (calls) to the code segment it must in various places contain the segment number of the code segment. This number is unknown at compile time and must be inserted when the two segments are made known as a result of a symbolic reference being made. To allow this insertion all words in the linkage segment which are to contain the code segment number must be chained together. That is, the compiler should initialise each of these words to contain the address within the linkage segment of the next word in the chain, and the last word in the chain to contain the value CHAINEND (-1).

As explained in section III.3, the contents of linkage segments are of three kinds. The exact forms of these are as follows.

a) The linkage segment must begin with a table of symbolic names of external entry points in the corresponding code segment. This table must begin in location 1. Each entry is a character string - the symbolic name - followed by the address within the linkage segment itself of the RCB for the entry point. These entries must be placed in contiguous locations so the table occupies an unbroken section of the segment. Word 0 of the linkage segment should contain p, the address fo the last word of this table, and word p+1 is head of the chain of references to the code segment introduced above.

b) An RCB describing an external entry point into the code segment occupies four words which contain:

1. An integer giving the size of a scratch data segment to be created when ENTERing the code at the point described by this RCB. If the value is 0 no segment is created.
2. The name of the emulator which interprets the code.
3. Code segment number; this word must be part of the code segment reference chain.
4. Word number of the entry point within the code segment; this value should be chosen so that the emulator which interprets the code may use it to initialise its program counter.

c) A link reference consists of three words. Initially it will be unresolved, and the compiler should therefore initialise it to contain:

1. The boolean false.
2. A word in the chain of code segment references.
3. The address within the code segment where the symbolic reference begins. A symbolic reference consists of three character strings, which must be placed immediately following each other. The first two strings make up the segment name, the third is the symbolic word name.

We are now in a position to give an example of the appearance of a code segment and its associated linkage segment. The code segment contains two external references, a code reference to the entry point "POP" within the code segment whose first name is "PROGX", and a data reference to the word named "ROOT" in the data segment named "OAKTREE". "DATA"; notice the extension of the symbolic segment name for the code reference is "LINK". There are two external entry points in the segment, named "PAP" and "PEP".

9. System Routines

The facilities provided by microprogram entry points into the system do not satisfy all user program needs for system services. Some routines in the BCPL part of the system have therefore been made callable as ordinary external entry points, i.e. via ENTER. The new-entry point into the OCODE machine has been pasted onto the already existing emulator and it does not

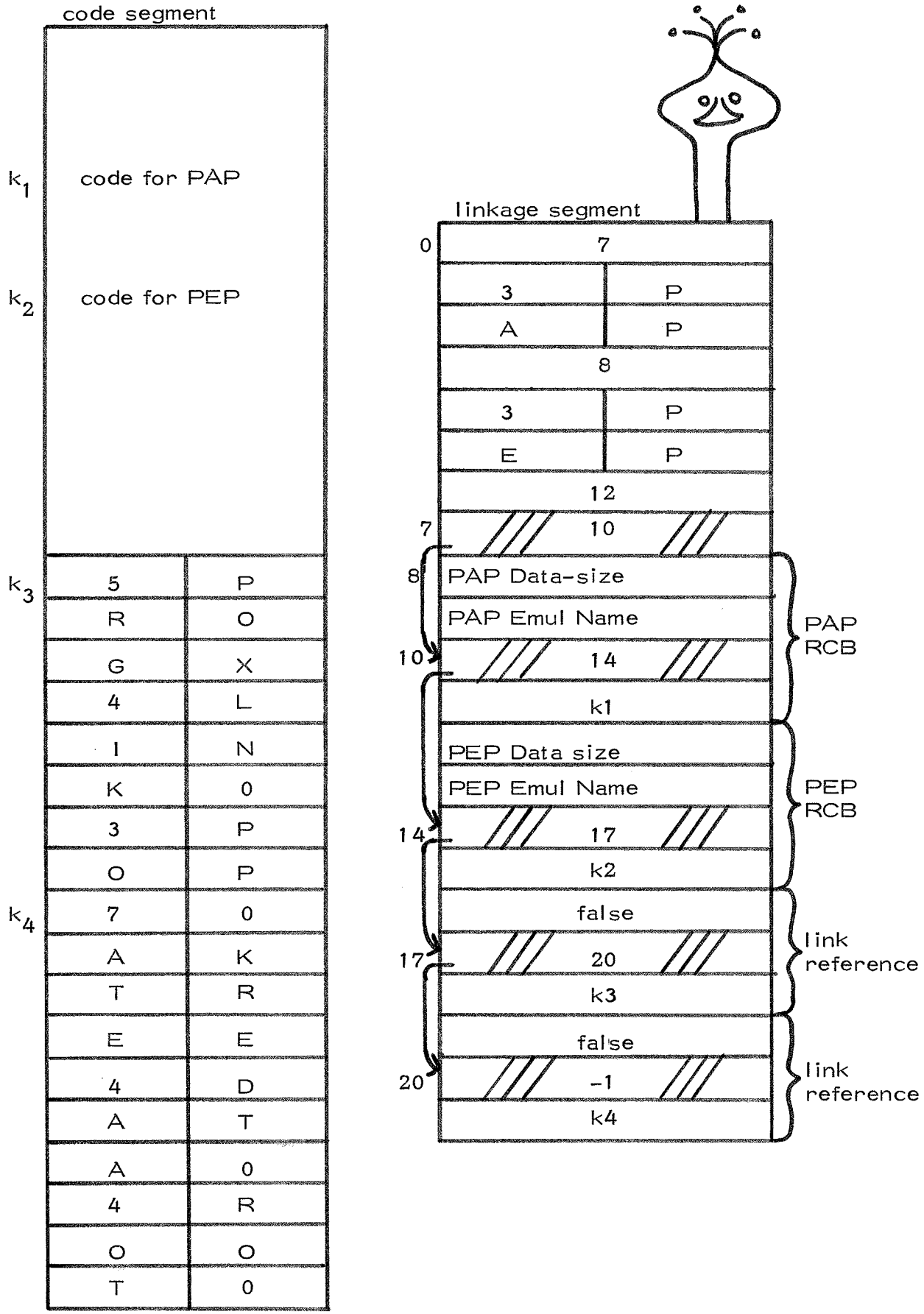

code segment
k1
code for PAP
k2
code for PEP
k3
5 P
R O
G X
4 L
I N
K 0
3 P
O P
k4
7 0
A K
T R
E E
4 D
A T
A 0
4 R
O O
T 0
linkage segment
0
7
3 P
A P
8
3 P
E P
12
7
10
8
PAP Data-size
PAP Emul Name
10
14
k1
PAP RCB
PEP Data size
PEP Emul Name
14
17
k2
PEP RCB
false
17
20
k3
link reference
false
20
-1
k4
link reference

Fig. 15

understand virtual address PCs. Therefore the system really has only one external entry point; the type of service required must be specified as a parameter. Depending on the value of this parameter, additional parameters are also required to further specify the request. Some of the system routines return values in the parameter locations of the super-stack. The RCB providing external entry into the system is always located at virtual address <0,0>.

Most of these routines provide access to the filing system or serve to change the state of a file or segment. More facilities may be added at a later time; the initial set described here is rather rudimentary.

For each routine we give the manifest representing the value of the first (integer) parameter which causes that particular routine to be invoked, then a specification of the types of the remaining necessary parameters, and a description of the service provided.

LOCK (SN:integer); locks the segment with number SN in core and returns its base address in core memory in parameter location 1 and its size in parameter location 2. LOCK may be used when a user program wishes to have direct access to a segment in core.

UNLOCK(SN:integer); unlocks segment.

CLOSE(S:integer); closes the stream which has value S.

RESET(S:integer); resets the stream which has value S.
We refer to the BCPL system manual for the meaning of Reset and Close.

MAKESEG(SegName: string, Extension: string; MSSeg: boolean; Size:integer); creates a segment of size Size, a main store segment if MSSeg is true, otherwise a wide store segment. The segment becomes associated with the name SegName.Extension in the current directory.

SEGNUMB(SegName:string,Extension:string); if the segment named SegName.Extension is not already known it is made known by referring to the current directory. Its segment number is then returned in parameter location 1. This provides a sort of high level link resolution.

MAKESEGNUMB(SegName:string,Extension:string,MSSeg:boolean, Size:integer); equivalent to MAKESEG followed by SEGNUMB.

WRITEFILE(FileName:string,Extension:string,Bytes:boolean); returns in parameter location 1 the value of a stream which may be used to write a file named FileName.Extension in the current directory. The name must not exist in the directory prior to the call. If Bytes is true, the file will be written as a file of bytes, i.e. each word written must contain a zero left byte; bytes will be packed two to a word. This is useful for text files, where it is natural to write one character at a time. Bytes does not affect the type of the file, only the way it is written; thus the same value should be specified when the file is later read, by means of READFILE.

READFILE(FileName:string,Extension:string,Bytes:boolean); returns the value of a stream to read the file named FileName.Extension in the current Directory (in parameter location 1). Bytes is explained above under WRITE-FILE.

SETSEGTYPE(SegName:string,Extension:string,Type:Integer); modifies the type of the segment named SegName.Extension in the current directory, depending on the value of Type. The following values may be used: NULL (= 0) causing no change, PROTECT (= 1), causing the segment to become write protected, and LINK (= 2), causing the segment to achieve status of linkage segment. Further values may be specified later. Caution: the type of a segment may not be changed while it is known.

FILEFROMSEG(SegName:string,Extension:string); the segment named SegName.Extension in the current directory becomes a file with exactly the same contents.

SEGFROMFILE(FileName:string, Extension:string,MSSeg:boolean, Type:integer); the file named FileName.Extension in the current directory becomes a segment, a main store segment if MSSeg is true, otherwise a wide store segment. Type is used exactly as for SETSEGTYPE above.

It is suggested that a compiler may initially write a code segment and the corresponding linkage segments as files, and later change these files to segments, setting the type appropriately.

VII. REFERENCES

[1] Jack B. Dennis: Segmentation and the Design of Multiprogrammed Computer Systems. JACM, vol. 12 (1965), No. 4, pp. 589-602.

[2] A. Bensoussan, C. T. Clingen & R. C. Daley: The MULTICS Virtual Memory. CACM, vol 15 (1972), No. 5, pp. 308-318.

[3] R. C: Daley & J. B. Dennis: Virtual Memory, Processes, and Sharing in MULTICS. CACM, vol. 11 (1968), No. 5, pp. 306-312.

[4] Elliott I. Organick: Computer System Organization, The B5700/B6700 Series. Academic Press, New York 1973.

[5] J. E. Stoy & C. Strachey: An Experimental Operating System for a small Computer. Part 1: General Principles and Structure. Computer Journal, vol. 15 (1972), No. 2, pp. 117-124. Part 2: Input/Output and Filing System. Computer Journal, vol. 15 (1972), No. 3, pp. 195-203.

[6] Butler W. Lampson: An Operating System for a Single-User Machine. Lecture Notes in Computer Science, vol 16 (1974), pp. 208-217. Springer Verlag, Berlin 1974.

[7] Sven Tafvelin: Dynamic Microprogramming and External Subroutine Calls in a MULTICS-Type Environment. BIT, bind 15 (1975) No. 2, pp. 192-202.

[8] Robert F. Rosin: Contemporary Concepts of Microprogramming and Emulation. Computing Surveys, vol. 1 (1969), No. 4, pp. 197-212.

[9] W. T. Wilner: Design of the B1700. Proc. FJCC 1972, pp. 489-497. See also B1700 Memory Utilization, same volume, pp. 579-586.

[10] S. G Tucker: Emulation of Large Systems. CACM, vol. 8, (1965), No. 12, pp. 753-761.

[11] A.K. Agrawala & T.C. Rauscher: Foundation of Microprogramming, Architecture, Software, and Applications. Academic Press, New York 1976.

[12] J.K. Broadbent: Microprogramming and System Architecture. Computer Journal, vol. 17 (1974, No. 1, pp. 2-8.

[13] N. Derrett & M.J. Manthey: Multi-Interpreter Systems. DAIMI PB-55, January 1976.

[14] P. Kornerup & B.D. Shriver: An Overview of the MATHILDA System. DAIMI PB-34, August 1974.

[15] Jørgen Staunstrup: A Description of the RIKKE-1 System. DAIMI PB-29, May 1974.

[16] Ejvind Lynning: The RIKKE BCPL System, A Programmer's Manual. DAIMI MD-22, February 1976.

[17] B.B. Kristensen, O.L. Madsen & B.B. Jensen: A PASCAL Environment Machine (PCODE). DAIMI PB-28, April 1974.

[18] Ole Sørensen: The Emulated OCODE Machine for the Support of BCPL. DAIMI PB-45, April 1975.

[19] S.E. Clausen, B. Madsen & E. Madsen: A Description of the LCODE-Interpreter on RIKKE-1. DAIMI MD-20, January 1976.

[20] E. Kressel & E. Lynning: The I/O-Nucleus on RIKKE-1. DAIMI MD-21, October 1975.

[21] J. McCarthy et al.: LISP 1.5 Programmer's Manual. The MIT Press, Cambridge, Mass. 1965.

[22] B.W. Lampson, J.G. Mitchell & E.H. Satterthwaite: On the Transfer of Control between Contexts. Lecture Notes in Computer Science vol, 19 (1974), pp. 181-203, Springer Verlag, Berlin 1974.