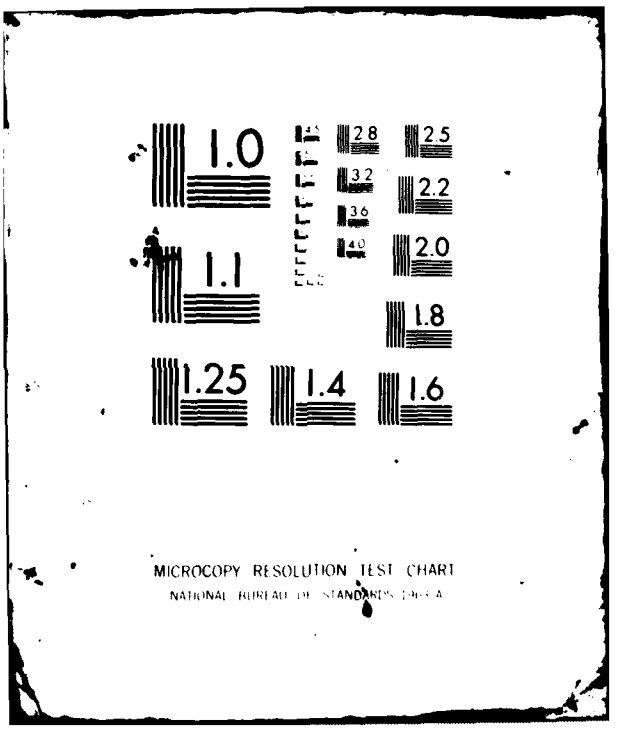
AD-A108 731 ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) F/G 9/2
FLEX FIRMWARE,(U)
SEP 81 I F CURRIE, P W EDWARDS, J M FOSTER
UNCLASSIFIED RSRE-81009 DRIC-BR-81061 NL

END
DATE FILMED
1 82
DTIC


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report No 81009

Title: FLEX FIRMWARE

Authors: I F Currie, P W Edwards and J M Foster

Date: September 1981

SUMMARY

This report describes the instruction set and general firmware architecture of the Flex computer. This architecture was designed by programmers, for programmers and represents a radically new approach in computer design.

Flex firmware

Contents

0 Introduction

This report describes the firmware architecture of the Flex computer. This architecture has been implemented in micro-code on various micro-programmable hardware configurations. The actual hardware used and the details of the micro-programming are outwith the scope of the report.

One might ask why it was considered desirable to invent yet another machine architecture. To answer this, the reader is invited to make a mental survey of programming languages with the concepts they embody and the standard machine architectures onto which they are mapped. Except for minor cosmetic constructions, the degree of mismatch in this mapping is usually astonishingly bad. This implies that compilers and other software are awkward and expensive to write. Even more serious, the limitations imposed by the difficulties of their implementations become embedded in their systems and in the minds of the programmers who use them. The cycle is completed when these same programmers design new languages and build in the same limitations.

Where, then, does Flex improve this situation? One could say that a quantitative improvement in technique has allowed us to make a qualitative improvement. The quantitative improvement arises from the use of micro-programming to implement a fast and efficient storage allocation scheme. This has allowed us the qualitative improvement of allowing the Flex machine to understand procedures, a program construct occuring in virtually every programming language. Further, these procedures are not limited by the artificial restrictions of stack based implementations, allowing their construction in a free and natural manner from within other procedures. The freedom that this endows on the programmer has to be experienced to be believed. The use of procedures to encapsulate actions and data with complete generality is an extremely powerful and easy to use technique. Programmers used to conventional machines experience a considerable culture shock when they learn how simply one can do things which could previously only be done by a mixture of black magic, inside knowledge and a lot of tedious programming.

The other significant advance in Flex is its use of file-store where files of arbitrary structure (and mode) can be constructed. The basis for this is the use of pointers which can exist equally well on backing store as in main store and can be freely handled by the programmer. The data that is pointed to by these pointers can contain other pointers and can have modes analagous to those existing in main-store e.g. one can have procedures existing on backing-store. Thus the programmer is freed from the tyranny of having to structure and name his files in some extremely limited manner prescribed by most conventional systems.

The instruction set devised for Flex is intended to make easy the compilation of high level languages. There is no assembly language for Flex and there never has been. All programs for Flex have, and will be, written in high level languages. In view of the above remarks these languages will tend to be procedure oriented.

The contents of this document describe only that part of the Flex machine which has been micro-coded i.e. the firmware. Clearly a Flex system must also include a considerable amount of software including operating systems, compilers etc. The inner core of this software is called kernel and is the most primitive set of procedures available to the non-priviledged user to allow him to make use of the privileged instructions. Thus the normal user could regard the kernel procedures as an extension of instruction set to replace the privileged instructions which he cannot use directly.

1 Data in Flex

Data in Flex can be handled in words, characters or booleans. Words are either pointers or non-pointers with distinct representations so that arithmetic cannot be performed on pointers and non-pointers cannot be used in indirect addressing operations.

Pointers and non-pointers in main memory are distinguished by the tag bits associated with the data. Each 8-bit byte of memory has one tag bit. A pointer is a 3-byte word with all three of its tag bits set to one. Non-pointers have zero tag bits.

1.1 Non-pointers

The various instructions which operate on non-pointers use the following representations:

BOOL - 1 bit.
CHAR - 8 bits.
INT - 1 word , 24 bits 2's complement.
LONG INT- 2 words, 1st word is most significant and second is positive.
REAL - 2 words, biased 9-bit 2's exponent in ls end of 2nd word, 38-bit 2's complement abscissa, split 24, 14 between 1st and 2nd word, with second word positive. Exact zero is represented by zero in both words and an underflowed number by (0,16r800000).

1.2 Pointers

Pointers always contain the address of a block. Each block has an overhead word which gives the size of the block with the tags of this word giving the type of the block. Existing pointers can be copied freely but one cannot synthesize a pointer to an existing block, and the contents of a block can only be accessed according to rules defined by both the pointer and the type of the block. Even given maximum access, a pointer only allows access within the limits of its block, not including the overhead word. A pointer to a block can exist in two forms - either locked or un-locked - the effect of locking depending on the type of block.

1.2.1 Blocks

The tags of the overhead words distinguish 6 different types :

1 - Work-space block i.e. the locals of some call of a procedure.

The first four words of a work-space block are completely inaccessible to everybody but the micro-code. They contain link information for exiting from the procedure or for re-instating this work-space as the current locals. The instructions which access within a work-space block (e.g.0 or 3) will automatically compensate for these extra invisible words so that, for example, a zero displacement (a-field) will in fact give the 5-th word of the block. A locked pointer to a work-space block means that one cannot use this pointer to assign or store to the block, ie a locked pointer means that the block cannot be altered via this pointer.

2 - Code block i.e. contains the code and constants of a procedure.

Code blocks have two invisible words, which contain information on the size of work-space block required to run this code. The next word (i.e. the first word of constants) gives the start of code relative to the start of the block. A locked pointer means that the code-block cannot be altered via this pointer.

3 - As type 4 , but cannot contain pointers.

4 - Normal data block.

Normal data blocks have no invisible words. A locked pointer means that the block cannot be altered via this pointer.

5 - Closure i.e. contains pointers to code and non-local blocks.

A pointer to a closure is a procedure. The only operations allowed on procedures are calls i.e. the contents of the closure are hidden. When a locked procedure is called, the resulting call will be run in privileged state.

6 - Keyed block; access controlled by knowledge of first word in block.

Access to a keyed block via an unlocked pointer makes it identical to a normal data block; access via a locked pointer is barred. A locked pointer to a keyed block may be unlocked if one knows the contents of the first word of the block using instruction 160. Locked pointers to other types of blocks cannot be unlocked.

1.2.2 Shaky pointers

In addition to the locked/unlocked property of pointers, pointers are either firm or shaky. So long as a block is pointed at by a firm pointer then any shaky pointer to that block remains valid; however, if there are no firm pointers to a block, then the shaky pointers to it will be replaced by 0, a non-pointer, in garbage collection, and hence the block will vanish. This facility is largely intended to provide easy aliasing between disc and main-store; it is also used internally in the micro code in the procedure call mechanism.

1.2.3 References,vectors and arrays

Since pointers only refer to complete blocks, several of the instructions make use of references i.e. (pointer, non-pointer) pairs with the interpretation that the first word gives a block and the second some kind of a displacement within it, subject to the usual access constraints of the pointer. The kind of displacement is defined by the first two bits of the second word:

0X - word displacement from logical start of block,
10 - char displacement from logical start of block,
11 - bit displacement from logical start of block.

A vector is defined as a triple consisting of (non-pointer, reference) where the first word is the upper bound (implicit lower bound = 1). This upper bound expresses the number of elements in the vector; the element size will be defined by the instruction using the vector and the type of its reference.

An N-dimensional array is a tuple of words consisting of N non-pointer triples (lower bound, stride, upper bound) followed by a reference.

2 Program in Flex

While running program, Flex is always obeying the code of some procedure, the current procedure.

2.1 Locals,non-locals,constants

The locals of this procedure,the current locals, are contained in a work-space block (type 1); these locals are directly accesssible using instructions such as load_1_word_local (op code 0). Two other areas are similarly accessible - the non-locals (eg op code 1) and the constants (op code 2) of the current procedure. The non-locals (if any) are in one of the blocks (type 4) pointed at in the closure which forms the current procedure while the constants (together with the code of the procedure) are in the other (type 2). The various access rights posessed by these pointers at the time the closure was formed, are carried over into the operation of instructions which use them implicitly; for example, if the non-locals pointer was locked at closure time, the store_into_non-locals instruction (op code 41) will be in error if it is encountered in the procedure code.

2.2 The local stack

The locals operate as a stack entirely contained within the current work-space block. The next free word on this stack is the stack-front (sf). Clearly sf is always constrained to lie within the limits of the current work-space; any attempt (either explicitly or implicitly) to go outside its bounds will result in an error.

2.3 The U register and tos

There is only one general purpose register in Flex - the universal register U. U may hold any number of words,characters or booleans, a special illegal value transformable (using instruction 165) to a word pair, or void. The instructions which load U (op codes 0 - 37 etc) push the old value in U (provided it is not illegal) onto the local stack (updating sf in multiples of words) before loading the new value.

Most of the arithmetic instructions use a value on top of the stack (tos) together with the value on U to produce results. The tos value is removed and sf reduced by the operation of the instruction. The number of words in tos depends on the particular instruction, and also sometimes on the value in U (eg equality, op code 114). Thus the int_multiply instruction (op code 100) multiplies a 1-non-ptr U to a 1-non-ptr tos, giving the answer as a 1-non-ptr, reducing sf by 1 word. The real_multiply instruction works similarly removing 2 words from the stack while the equality instruction removes the number of words required to hold the value in U.

2.4 Program control

The flow of control instructions only allow jumps within the current procedure code and even then only in a restricted form in that usually only forward jumps are allowed to carry a non-void U. The only ways to escape from the current procedure code is to call another procedure, exit from the current one, fail, or obey the goto instruction (op code 71). The address of the instruction currently being obeyed is held in the program control register pc ; clearly pc is constrained to be within the limits of instructions within the current codeblock.

2.5 Privilege

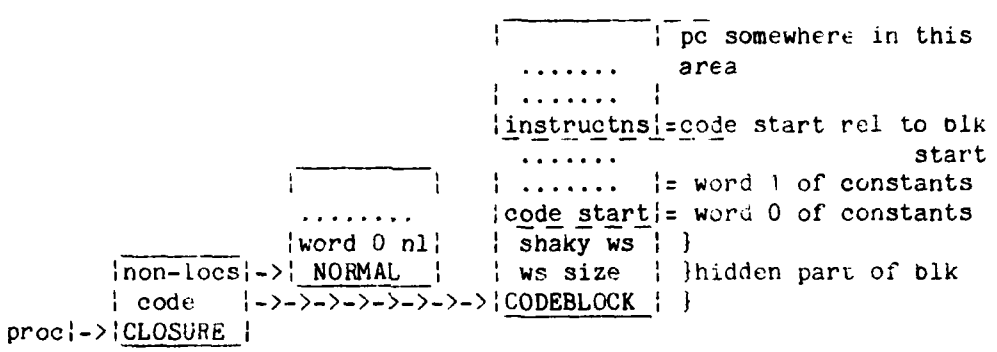
Some of the instructions are only allowed if the procedure code is being run in privileged state. These instructions are mainly concerned with peripheral transfers. This state is a property of the procedure call and in general will follow the nested sequence of procedure calls. Thus, if an unprivileged procedure is called from within a privileged one, on exit from the inner call the outer will remain privileged and similarly in the reverse sense. Clearly the operation which makes a procedure privileged can only be obeyed in privileged state.

2.5 Exception state

The action on a failure due to the illegal operation of an instruction is controlled by another state, either T- or D-state. In this case, the state is set by instructions (op code 94 & 95) and carried into inner procedure calls. On exit from a procedure the TD state is reset to what it was on entry.

3 Procedures

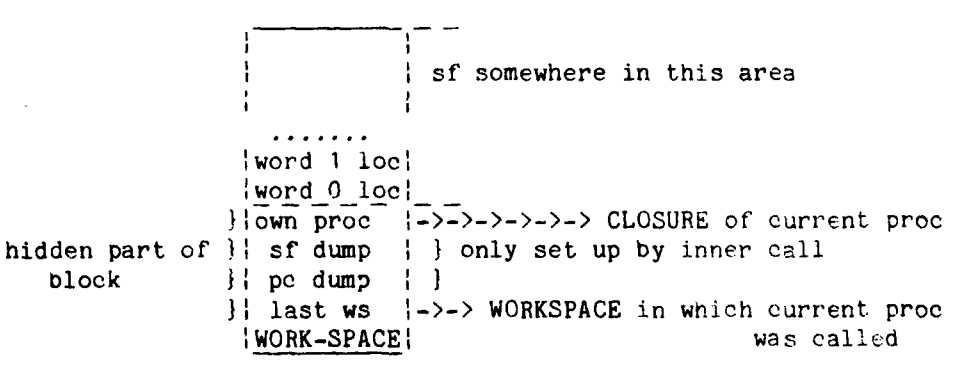
A procedure is a pointer to a closure block:



The constants involved in the procedure are held in the codeblock starting at the third word. The two words below are hidden, being used in the call and exit instructions. Word 0 of the constants is a byte displacement from the block start giving the 1st instruction of the procedure. This word cannot be altered, although it can be read in program.

3.1 Work-spaces

The current work-space block is:



The workspace chain given by the last ws chain in a work-space is terminated by a zero word, in the first work-space of a process.

3.2 Procedure calls

The action of a procedure call (op codes 64-67) is as follows. The current values of pc and sf relative to their respective blocks are stored in the second and third words of the current workspace. The privilege state and the TD state are also stored along with pc. The codeblock derived from the procedure to be called is now examined. It contains, in its first two words, information to produce a work-space block for the procedure being called. If the second word (shaky_ws in diagram) is a pointer, then we know (see exit) that this is shaky pointer to a chain of work-spaces suitable for running this procedure and hence we have the desired work-space by removing it from this chain. If shaky_ws is not a pointer, then a new work-space block is generated given its size in the first word in the standard manner. This may involve one in a garbage collection and so the procedure call instructions are arranged so that they can be restarted after such a garbage collection.

Having produced a work-space suitable for the new procedure, a pointer to the current work-space is put in its first word, and the new procedure in its fourth. This new work-space now becomes current, sf is set to its fifth word (word 0 of locals), pc to the first instruction of the new codeblock, and the internal registers set up so that the current locals, non-locals and constants come from the new locals, non-local block and codeblock respectively. If the procedure being called is locked, the code will run in privileged state; otherwise it is unprivileged.

During all of this, the contents of U remain unchanged so that parameters to a procedure are normally passed in U.

3.3 Exit from procedures

Exit from a procedure (op codes 68, 69) is essentially the reverse process: if a pointer to the current work-space has not been loaded while obeying the current procedure (ie instructions 34 or 168 have not been encountered) then the current work-space is put on the chain shaky_ws in the current codeblock. The previous work-space (in 1st word of current work-space) is now made current, and sf, pc and the two states are reset from their dump positions in this work-space. The locals are made current in this work-space and non-locals and constants are reset from the own_proc dumped within it.

During exit, the contents of U remain unchanged so that results of procedures are normally passed in U.

3.4 Demand loading

A closure block can be somewhat different to that shown above. The closure instruction (72) allows the operand relating to a codeblock to be a keyed block. The firmware will interpolate a call of a system procedure, load_proc, in the operation of a call instruction whose closure contains a keyed block. Load_proc is rather similar to the scavenge procedure, in that it can accept any parameter. It can access the keyed blocks as non-locals, and the intention is that it will use information from the keyed blocks to load the actual code etc, from backing store into mainstore. Load_proc will then exit to repeat the original call instruction that provoked it, ensuring that this kind of closure is only fully loaded when it is actually called.

4 Exceptions

An exception in Flex occurs when some attempt is made to break the rules of the Flex instruction set. All exceptions have a characteristic word-pair associated with them - those raised directly by the micro code consist of zero followed by a small integer.

4.1 Errors and failures

Exceptions arise in two slightly different ways, called, for want of better words, errors and failures.

An error occurs where any attempt to continue with the instruction would compromise the access rules for blocks and pointers. Roughly speaking,one could say that they are the compiler's fault. They include attempts to access outside the limits of a block or applying the wrong type of operands to instructions.

A failure tends to be more data-dependent and more likely to be the program writer's fault. Typical failures are arithmetic overflow and indices out of bounds. Also included are the explicit exceptions raised by the fail and exit_fail instructions (op codes 173, 69) where the characteristic is given by the operand U.

4.2 Exceptions in T-state

The only difference between failures and errors occur when the exception is raised in T-state. In this case, a failure produces an illegal value in U which has an associated word-pair identical to the characteristic of the exception. Any attempt to use an illegal value in instructions other than those explicitly designed to deal with them (op codes 92, 165 etc) will result in an error whose characteristic is the same as that associated with the illegal value. Of course, one can deal with failures like overflow in the current procedure by using these illegal-handling instructions.

In T-state, an error results in the premature exit from the current procedure with an illegal value in U whose word-pair is the characteristic of the error. Since illegal values give errors unless explicitly tested for, the net effect is that an entire chain of procedure calls are exited from until one is encountered which is prepared to accept an illegal result.

4.3 Exceptions in D-state

In D-state (Diagnostic state) all exceptions are treated identically. In essence, a system procedure, fail_proc, is called in place of the current procedure, so that, if an exit was obeyed in fail_proc, it would exit to the same place as the current procedure. The parameters of the call of fail_proc give access to the locals

and codeblock of the failing procedure and the characteristic of the exception is available as a non-local. The (softwared) action of fail_proc is to construct a chain of failing environments. It does this by constructing an element of the chain and then doing an exit_fail with a reference to the element in U. Eventually, some lower procedure will gather up the resulting illegal and use it to construct visible diagnostics for the exception. As far as the firmware is concerned, all that it does at an exception in D-state is to find fail_proc in system_block, dump the characterisic into system_block, and call fail_proc with a four word parameter consisting of a pointer to the current locals, relative values of pc and sf, and a pointer to the local codeblock.

5 Storage allocation

Only the micro-code regards Flex main store as a linear store addressable from end to end. The macro-code which is the Flex instruction set only understands blocks and pointers to them, so that Flex program can only address those disjoint unrelated blocks for which it has pointers of the right sort. Running programs will involve new blocks being created to hold data, for example by using the generate instructions (72-74 etc) or simply by calling a procedure which requires a new work-space block. Thus the micro-code which implements those instructions simply grabs a new empty block from the top of a continually growing stack in the linear store, putting in the appropriate overhead word and delivering a pointer as result.

This linearly growing stack will eventually encompass the entire physical store and at this stage garbage collection occurs. The garbage collector notes all of blocks which are still "live", and compacts all live blocks down to the bottom of store, updating all pointers in them so that they still point to the same data. Thus the space occupied by "dead" blocks is recovered and, hopefully, there will be sufficient room in the linear store for the request for a new block which provoked the garbage collection.

Clearly the address actually held in a pointer can change on each garbage collection. However since all pointers to a given block are changed consistently and since arithmetic is forbidden on pointers, one can regard pointers as immutable objects in Flex programs.

A live block is either a unique block, system_block, or else is pointed to from within another live block. System_block is a block known to the micro code and contains the interrupt procedures and other goodies to keep alive all currently active processes.

The actual sequence of events which happens in the micro code at a garbage collection is as follows. The micro code discovers that a request for the generation of a block cannot be satisfied from the linear store. It then interpolates the call of a procedure, scavenge, before the current instruction. Scavenge (which is found in system block) is a peculiar procedure in that one can guarantee that there will always be a workspace available for it and that it can accept any value in U as a parameter. This last is neccessary since the instruction requiring the block could be a procedure call which is meant to leave the value of U unchanged. The privileged scavenge procedure dumps the value of U, obeys the garbage_collect instruction, and then finds if the current store demand can be satisfied; if it can then the dumped value of U is reinstated and scavenge is exited - to repeat the store grabbing instruction. If it cannot be satisfied, then some process must be failed so that store can be released to continue.

6 Interrupts

Interrupts do not occur when instructions are being obeyed in privileged state.

An interrupt can only occur when U contains void; this sometimes occurs in the middle of an instruction where repeating the instruction would do no harm. This is the case in the load instructions where U is void after it has been pushed but before the new value has been loaded into U; since pushing void is a null operation, the instruction has the same effect whether or not it has been interrupted and restarted.

When an interrupt occurs, the effect is exactly the same as if a parameterless procedure (delivering void) had been called in the code being interrupted. This procedure depends on the type of interrupt and is found in system_block. The call of an interrupt procedure will not invoke garbage collection and is intended to be run in privileged state.

The two main interrupts are the comflex channel and the timer. The Flex hardware contains a milli-second clock, which the firmware uses to make an interval timer. This timer can be set using the privileged set_slot_time instruction (op code 216) and when the interval specified is expired,the timer interrupts.

7 Data in file store

The basic file store operations on Flex are create a new block on disc, and read an existing block. Thus we may write away information to a given file store, receiving back a disc pointer which we can subsequently use to read back the information. The information written to a filestore may include disc pointers to other blocks in the filestore so that the file store can contain trees of arbitrary complexity.

On disc, a disc pointer is represented by four bytes; in main memory a disc pointer is a locked pointer to a keyed block containing these bytes with a key which identifies the file store containing the data pointed at by the disc pointer:

Tags	Byte2	Byte1	Byte0	Word	
XXX				4	Possible alias for disc value
000				3	Address for pointer on disc
000			U | T	2	Tag for pointer on disc.
111				1	File store key.
110			15	0	Overhead word = block size.

Only one such block will exist in main memory for given words 1, 2 and 3; the instructions which handle disc ptrs will identify it uniquely via a hash table in system_block.

The four byte representation on disc comes from the least significant byte of word 2, and the remaining three from word 3. The use of the filestore key in the instructions (op codes 225, 226, 233) which transput disc pointers will ensure that the only disc pointers which can exist on a given filestore are pointers within the filestore.

U in word 2 = Unit number in filestore (0 <= U <= 15);
T in word 2 = Type of value corresponding to disc pointer
(1 <= T <= 15)

Disc pointer types:
T = 1 => Will always be aliased with existing value in main memory.
T = 2 => Block on disc is read in as code-block.
T = 3 => " " " is read in as procedure with no non-locals.
T = 4 => As T=3 but procedure is locked.
T = 5 => Block on disc is read as procedure.
T = 6 => As T=5 but procedure is locked.
T = 7 => Disc pointer is non-writeable disc reference.
T = 8 => Disc pointer is disc reference.
T = 9 => Block on disc is normal block which can contain pointers.
T = 10 => Block on disc is normal block which cannot contain pointers.
T = 11-15 => Unassigned and unused.

The procedure d_to_b takes a disc pointer as a parameter and delivers a main memory pointer as result, the type of block pointed to depending on T value above. The pointer delivered will be such that the corresponding block cannot be written to. Word 4 will then contain a shaky version of the pointer so that as long as a firm version of that pointer exists subsequent calls of d_to_b on the disc pointer will not require an interaction with disc. Type 1 disc pointers will always have an existing alias and so never interact with disc.

Disc pointers with types 9 and 10 may also be read using the procedure from_disc which allows the data read to be scattered into places defined by its parameters. In this case no aliassing takes place.

Disc pointers of types 2 to 6 and 9 and 10 may be created by writing data to a filestore using the appropriate system procedure depending on the type.

Disc pointers of type 8 are called disc references; they are peculiar in that they define the only words in a filestore which can be overwritten. A disc reference is really a reference to one word; usually, this one word will contain a disc pointer to some kind of a dictionary which is updated from time to time. The contents of a disc reference may be read using d_to_b as above; however, updating it is slightly more compilcated. The disc system will update the disc reference as a unitary operation and will only allow it to be updated if the old value contained within it is presented at the same time as the new one. Thus simultaneous updating of the same disc reference can be detected and resolved in a reasonably economical way.

8 Flex instruction set

Instructions are 1, 2 or 3 bytes long, the first defining the operation code. The remaining bytes, if any, are denoted by a & sz (1 byte quantities) or p (two byte quantity). The a-field generally is a data displacement and the sz-field gives data size. The a and sz fields can be effectively extended by using the modify_next instruction (op code 76). A p-field is a byte displacement from the beginning of the current procedure code.

A * in the description means that the instruction is interruptable at that point.

Load 1 word

0 , a : Push U * , U:= a-th word of locals.
1 , a : Push U * , U:= a-th word of non-locals.
2 , a : Push U * , U:= a-th word of constants.
3 , a : U:= a-th word of block pointed at by U.

Load 2 words

4 , a : Push U * , U:= word-pair starting at a-th word of locals.
5 , a : Push U * , U:= word-pair starting at a-th word of non-locals.
6 , a : Push U * , U:= word-pair starting at a-th word of constants.
7 , a : U:= word-pair at a-th word of block pointed at by U.

Load N words

8 , a , sz : Push U * , U:= sz+3 words starting at a-th word of locals.
9 , a , sz : Push U * , U:= sz+3 words starting at a-th word of non-locals.
10, a , sz : Push U * , U:= sz+3 words starting at a-th word of constants.
11, a , sz : U:= sz+3 words at a-th word of block pointed at by U.

Load 1 character

12, a : Push U * , U:= character in ls byte of a-th word of locals.
13, a : Push U * , U:= character in ls byte of a-th word of non-locals.
14, a : Push U * , U:= character in ls byte of a-th word of constants.
15, a : U:= character in ls byte of a-th word of block pointed at by U.

Load 1 boolean

```
16, a       : Push U * , U:= bool in ls bit of a-th word of locals.
17, a       : Push U * , U:= bool in ls bit of a-th word of
              non-locals.
18, a       : Push U * , U:= bool in ls bit of a-th word of
              constants.
19, a       : U:= bool in ls bit of a-th word of block pointed at
              by U.
```

Load N characters

```
20, a , sz  : Push U * , U:= sz+2 chars at a-th word of locals.
21, a , sz  : Push U * , U:= sz+2 chars at a-th word of non-locals.
22, a , sz  : Push U * , U:= sz+2 chars at a-th word of constants.
23, a , sz  : U:= sz+2 chars at a-th word of block pointed at by U.
```

Load N booleans

```
24, a , sz  : Push U * , U:= sz+2 bools at a-th word of locals.
25, a , sz  : Push U * , U:= sz+2 bools at a-th word of non-locals.
24, a , sz  : Push U * , U:= sz+2 bools at a-th word of locals.
27, a , sz  : U:= sz+2 bools at a-th word of block pointed at by U.
```

Load ptr to current areas

```
28          : Push U * , U:= ptr to non-locals block.
29          : Push U * , U:= ptr to constants block.
```

Load literally

```
30, a       : Push U * , U:= a  (CHAR).
31, a       : Push U * , U:= a  (BOOL)
32, a       : Push U * , U:= a  (INT).
33          : Push U * , U:= void.
```

Load ptr to locals

```
34          : Push U * , U:= ptr to locals block.
```

Load times

```
35          : Push U * , U:= time of day ( LONG INT milli-secs).
36          : Push U * , U:= unexpired slot-time ( INT milli-secs).
```

Push and take

```
37, sz      : Push U * , U:= sz words on tos.
```

Push to byte and bit positions

38, a : Push chars in U to byte position sf-a , U:=void * .
39, a : Push bools in U to bit position sf-a , U:=void * .

Store U

40 , a : Stores U (any value) at a-th word of locals , U:=void * .
41 , a : Stores U (any value) at a-th word of non-locals, U:=void * .
42 , a : Stores U (any value) at a-th word of constants, U:=void * .
43 , a : Stores U (any value) at a-th word of block pointed at by [illegible]

Select from U

44 , a , sz : U:= sz words starting at a-th of U.
45 , a , sz : U:= sz chars starting at a-th of U.
46 , a , sz : U:= sz bools starting at a-th of U.

Select ref

47 , a : Select ref i.e. add a to last word of U.

Derefs

48 , a , sz : Deref word vector in U i.e. U:=(UPB vector * a + sz) words pointed at by vector.
49 , sz : Deref word ref in U i.e. U:= sz words pointed at by ref.
50 , a , sz : Deref char vector in U i.e. U:=(UPB vector * a + sz) chars pointed at by vector.
51 , sz : Deref char ref in U i.e. U:= sz chars pointed at by ref.
52 , a , sz : Deref bool vector in U i.e. U:=(UPB vector * a + sz) bools pointed at by vector.
53 , sz : Deref bool ref in U i.e. U:= sz bools pointed at by ref.

Pack & Unpack

54 : Unpack i.e. U:= word contents of block pointed at by U.
55 : Pack i.e. U:= ptr to block (type 4) containing copy of U.

Vector operations

```
56 , sz     : Index vector in U with element size sz by index on
              tos giving ref to element in U;
              Given U = (b,p,d) and tos = i,
              U:=(p,d+(i-1)*sz) where 1 <= i <= b.
57 , sz     : Trim vector on tos with element size sz by int pair
              in U giving trimmed vector in U;
              Given U = (i,j) and  tos = (b,p,d),
              U:= (j-i+1,p,d+(i-1)*sz)  where i>=1 and j<=b.
58          : If UPB vector on tos /= UPB vector in U then
              fail (0,2).
```

Array operations

```
59 , a      : Index a+1 dimensional array in U by a+1 indices on
              tos giving ref to element in U;
              Given U = (lb0,s0,ub0, ...,lba,sa,uba, p,d)
              and tos = (i0,i1,... ia),
              U:=(p,d + s0*(i0-lb0) + ... + sa*(ia-lba))
               where lbn <= in <= ubn  for  0 <= n <= a.
60 , a      : Trim a+1 dimensional array on tos by integer triple
              in U, giving a+1 dimensional array in U;
              Given U = (f,t,nlb)
              and tos = (lb0,s0,ub0, ...,lba,sa,uba,p,d),
              U:=(nlb,s0,nlb+t-f, lb1,s1,ub1,..., lba,sa,uba,
                    p, d+(f-l0)*s0 )  where  f >= lb0 and t <= ub0.
61 , a      : Slice a+1 dimensional array on tos by index in U,
              giving a dimensional array in U.
              Given U = i
              and tos = (lb0,s0,ub0, ...,lba,sa,uba,p,d),
              U:= (lb1,s1,ub1, ..., lba,sa,uba,p, d+ s0*(i0-lb0))
                where  lb0 <= i <= ub0
```

Unite

```
62 , a , sz : Unite U with a and make it a sz word object, i.e.
              U:= (a,U.., 0,...).
```

Assign

```
63          : Assign  value in U to position given by ref on tos,
              and let U := ref;
              Words in U cannot be assigned to a CHAR- or BOOL-ref;
              Chars in U cannot be assigned to a BOOL-ref.
```

Procedure calls & exits

```
64 , a      : Call the procedure given at a-th word of locals.
65 , a      : Call the procedure given at a-th word of non-locals.
66 , a      : Call the procedure given at a-th word of constants.
67          : Call the procedure given on tos.

68          : Exit from current procedure.
69          : Exit from current procedure and fail U .
70          :
```

Goto

```
71          : Goto label given in U, where label is pair (pointer
              to destination workspace, p-displacement in
              corresponding codeblock).
```

Generate new blocks

```
72          : U:= ptr to new closure (type 5) formed from  ptr to
              code (type 2 or 6) in U and ptr to non-locals (type 4)
              or zero on tos; (PTR,WORD) -> PROC.
73          : U:= ptr to new normal array block (type 4) of size in
              words given by U.
74          : U:= ptr to new block (type 3) of size given by U.
75          : Change block (type 4) pointed at by U into codeblock
              (type 2).
```

Modify next

```
76 ,a1,sz1  : Modify the a & sz fields of next instruction
              (if present) by a1*256 & sz1*256.
```

Stack front operations

```
77 , a      : Set sf to start of locals + a words.
78 , a      : If sf /= start of locals + a  then fail (0,5)
```

Discard

```
79          : U:=void * .
```

Operations on pointers

```
80          : U:= shake U;  PTR -> PTR.
81          : U:= firm U;  WORD -> WORD .
82          : U:= U is a ptr; WORD -> BOOL.
83          : U:= block type of ptr in U;  PTR -> INT.
84          : U:= byte block size of ptr in U;  PTR -> INT.
```

Ref changes

```
85          : Make ref in U into char ref;  REF WORD -> REF CHAR.
86          : Make ref in U into bool ref.
              (REF WORD or REF CHAR) -> REF BOOL.
```

Modify next dynamically

```
87          : Modify the a & sz fields of next instruction by
              int pair on tos.
```

Jumps & branches

```
88 , p      : IF U then jump to p FI ,  U:= void * .
89 , p      : IF not U then jump to p FI ,  U:= void * .
90 , p      : IF U then jump forward to p ELSE U:= void * FI.
91 , p      : IF not U then jump forward to p ELSE U:= void * FI.
92 , p      : IF U is illegal then jump to p, U:=void * FI.
93 , p      : Jump to p (if U not void then jump must be forward).
```

Set failure state

```
94          : Set D-state.
95          : Set T-state.
```

FOR instructions

```
96 , p      : For test; (FOR,BY) in U , TO on stack ;
              IF (TO-FOR)*BY < 0
              THEN U:=void, Pull TO , jump to p * FI.
97 , p      : For step; * (FOR,BY) on tos , U:=(FOR+BY,BY),
              jump to p.
```

Switches

```
98 , a      : Case switch; jump to next + ( 1<= U <= a | 3*U | 0).
99          : Associative switch;
              Followed by (b1,p1)or(x1+128,y1,p1)...(bn=0,pn),
                        b1<128 and x1<128;
              FOR i DO
              IF U=bi (single byte) OR  U>=xi AND U<=yi THEN
                 jump to pn,  U:=void
              ELIF bi = 0 THEN jump forward to pn FI
              OD.
```

Integer arithmetic

```
100         : U:=tos+U;  (INT,INT)->INT
              or ((INT,0),LONG INT)->LONG INT.
101         : U:=tos-U;  (INT,INT)->INT
              or ((INT,0),LONG INT)->LONG INT.
102         : U:=tos*U;  (INT,INT)->INT.
103         : U:=(remainder,tos/U); (INT,INT)->(INT,INT).
```

Integer tests

```
104         : U:=tos>=U;  (INT,INT)-> BOOL.
105         : U:=tos<U;   (INT,INT)-> BOOL.
106         : U:=tos<=U;  (INT,INT)-> BOOL.
107         : U:=tos>U;   (INT,INT)-> BOOL.
```

Monadic operations

```
108         : U:= ABS U;  INT->INT.
109         : U:= -U;     INT->INT.
110         : U:= ABS U;  (CHAR or BOOL or WORD) -> INT.
111         : U:= U is illegal;  ANY or illegal -> BOOL.
112         : U:= REPR U;  INT->CHAR.
113         : U:= ODD U;   INT->BOOL.
```

Equality

```
114         : U:= tos=U    (ANY,ANY)->BOOL.
115         : U:= tos/=U   (ANY,ANY)->BOOL.
```

Logical operations

```
116         : U:= tos OR U;   (n-BOOL,n-BOOL) -> n-BOOL
              or (INT,INT)->INT.
117         : U:= tos AND U;   (n-BOOL,n-BOOL) -> n-BOOL
              or (INT,INT)->INT.
118         : U:= tos EXOR U   (n-BOOL,n-BOOL) -> n-BOOL
              or (INT,INT)->INT.
119         : U:= tos EQUIV U;   (n-BOOL,n-BOOL) -> n-BOOL
              or (INT,INT)->INT.
120         : U:= NOT tos AND U;  (n-BOOL,n-BOOL) -> n-BOOL
              or (INT,INT)->INT.
121         : U:= NOT U;   n-BOOL -> n-BOOL   or INT->INT.
```

String equality

```
122         : U:=(string in vector on tos = string in vector in U);
              (VECTOR[]CHAR,VECTOR[]CHAR)->BOOL.
123         : U:=(string in vector on tos /= string in vector in U);
              (VECTOR[]CHAR,VECTOR[]CHAR)->BOOL.
```

Real arithmetic

```
124         : U:= tos+U;  (REAL,REAL)->REAL.
125         : U:= tos-U;  (REAL,REAL)->REAL.
126         : U:= tos*U;  (REAL,REAL)->REAL.
127         : U:= tos/U;  (REAL,REAL)->REAL.
```

Real tests

```
128         : U:= tos>=U; (REAL,REAL)->BOOL.
129         : U:= tos<U; (REAL,REAL)->BOOL.
130         : U:= tos<=U; (REAL,REAL)->BOOL.
131         : U:= tos>U; (REAL,REAL)->BOOL.
```

Real monadic operations

```
132         : U:= ABS U;  REAL -> REAL.
133         : U:= - U;  REAL -> REAL.
134         : U:=ENTIER U;  REAL -> INT.
135         : U:=ROUND U; REAL -> INT.
136         : U:= widen U;     INT -> REAL.
137         : U:= widen U;     LONG INT -> REAL.
138         : U:= ENTIER U;    REAL -> LONG INT.
139         : U:= ROUND U;     REAL -> LONG INT.
```

Long arithmetic

```
140         : U:=tos+U ;    (INT,INT)->LONG INT
              or (INT,LONG INT)->LONG INT.
141         : U:=tos-U ;    (INT,INT)->LONG INT
              or (INT,LONG INT)->LONG INT.
142         : U:=tos*U ;    (INT,INT)->LONG INT.
143         : U:=(remainder,tos/U);  (LONG INT,INT)->(INT,INT).
```

Long to decimal

```
144         : U:=(remainder,U/10);  LONG INT -> (INT,LONG INT).
```

Long monadics

```
145         : U:= LENGTHEN U;    INT -> LONG INT.
146         : U:= SHORTEN U;   LONG INT -> INT.
```

Decimal to long

```
147         : U:= U*10 +( U>=0 ¦ tos ¦ -tos);
              (INT,LONG INT) -> LONG INT.
```

Long tests

```
148         : U:= tos>=U; (LONG INT,LONG INT)->BOOL.
149         : U:= tos<U; (LONG INT,LONG INT)->BOOL.
150         : U:= tos<=U; (LONG INT,LONG INT)->BOOL.
151         : U:= tos>U; (LONG INT,LONG INT)->BOOL.
```

Keyed block operations

```
158          : U:= open ptr to new keyed block of size U words;
               INT -> PTR.
159          : U:= locked version of pointer in U;
               PTR or REF -> PTR or REF.
160          : U:= open ptr to keyed block in U with key on tos;
               (WORD,PTR or REF) -> PTR or REF.
161          : Change normal block pointed to by U into keyed block;
               PTR or REF -> PTR or REF.
```

Load d_to_b

```
162          : Push U * , U:= d_to_b (in 16th word of system block).
```

Decimal exponent conversions

```
163          : U:= (e,1) where 1*10^e = U ;  REAL -> (INT,LONG INT).
164          : U:= U *10^tos;  (INT,LONG INT) -> REAL.
```

Unite with illegal

```
165, sz      : Unite illegal i.e.
               U:=IF  U isnt illegal THEN  (1,U,..,0,...)
                  ELSE (2,characteristic word pair of U,..0..,)
                  Fi;
                (ANY or ILLEGAL ) -> sz-WORD.
```

Vector pack and unpack

```
166          : Pack U into new vector ;
               n-WORDs,n-CHARs or n-BOOLs
                 -> VECTOR[]WORD,CHAR or BOOL.
167          : Unpack vector in U to prduce n-words,n-chars
               or n-bools in U;
               VECTOR[]WORD,CHAR or BOOL
                 -> n-WORDs,n-CHARs or n-BOOL.
```

Load vector of characters

```
168, a , sz : Push U * , U:=vector(sz,locals,a+16r800000).
169, a , sz : Push U * , U:=vector(sz,non-locals,a+16r800000).
170, a , sz : Push U * , U:=vector(sz,constants,a+16r800000).
```

Fail

```
173          : U:= illegal formed from word-pair in U;
               (WORD,WORD) -> illegal.
```

Disable and enable

```
200          : If U = systemblock (PTR or REF) then set privileged
               state else fail (0,11).
201          : Set non-privileged state.
```

Privileged instructions

Append

153 : U := tos Append U ,where tos = ref to block whose 1st word is chain ending in zero.
(REF , WORD or REF) -> REF

Scavenge

204 : Do a garbage collection, delivering the number of bytes recovered in U as an INT.

Dump and reset U

206 : Dump U (in internal form) to the first 5 words of the current work-space,leaving U void.
207 : Reset U from dumped value in first 5 words of the current work-space.

Peripheral processor channel

208 : Write character to peripheral processor, by back-door channel.
209 : Send current pc to peripheral processor and return control of microcode to peripheral processor.
210 : U := character read from peripheral processor.

Timing

215 : Set time (ms) from LONG INT in U; U:=void.
216 : Set interval timer (ms) from INT in U; U:=void.

Load system_block

217 : U:= REF to U'th word of system_block; INT -> REF.

Find

219 : Find PTR in system hash-table whose first 3 words are equal to U; If not there create 4 word keyed block containing (U,0), insert locked pointer in hash table and deliver it;
(PTR,INT,INT) -> PTR.

Comflex output

```
220         : Write comflex header (dest,source,size) given in U
              to comflex, U:=void.
221         : Write INT in U to comflex as 1 byte; U:=void.
222         : Write INT in U to comflex as 2 bytes; U:=void.
223         : Write INT in U to comflex as 3 bytes; U:=void.
224         : Write vector of chars to comflex; U:=void.
225         : Write word in U to comflex as 4 bytes; If U is a
              pointer, then then the first word in its block must be
              identical to the word on tos or else the first word is
              system_block and the second word is 1, and the bytes
              written will be derived from the contents of the
              block; if U is not a pointer, then the word on tos is
              irrelevant and the bytes written are 0 , followed by
              the byte representation of the integer in U.
              (WORD,WORD) -> void.
226         : Write vector of words in U to comflex; Each word is
              treated as in instruction 225.
              (WORD,VECTOR[]WORD) -> void.
227         : Complete packet (by sending U as a byte repeatedly,if
              necessary), delivering comflex status in U (fail if
              status shows error). INT -> INT.
```

Comflex input

```
228         : Push U ; U := comflex header as integer triple.
229         : Push U ; U := 1 byte from comflex as integer.
230         : Push U ; U := 2 bytes from comflex as integer.
231         : Push U ; U := 3 bytes from comflex as integer.
232         : Read bytes from comflex into vector of chars in U.
233         : Let b = next byte from comflex, a = next 3 bytes
              from comflex;
              IF b = 0 THEN  U:= a
              ELIF b = 1 THEN U:= FIND ( system_block,1,a)
              ELSE U := FIND ( U,b,a) FI;  FIND = Op code 219;
              PTR -> WORD.
235         : Push U; Clear comflex buffer by reading bytes if
              necessary delivering status in U (fail if status
              shows error).
```

Comflex control

```
236         : Push U; U:= comflex status as INT.
237         : U as INT is sent to comflex  command channel;
              U:= void.
```

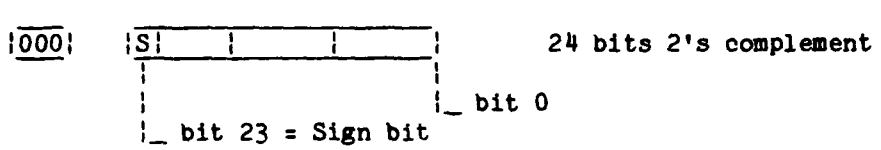
Make and break blocks

```
239         : U := word-pair in first two words of block pointed
              at by U;
              PTR -> WORDPAIR.
240         : Assign word pair in U to block pointed at by tos;
              (PTR,WORDPAIR) -> void.
```

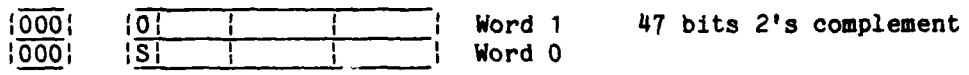
9 Summaries

9.1 Data in store

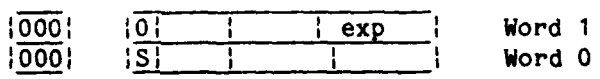
9.1.1 Integers



9.1.2 Long integers



9.1.3 Reals



Exp is a biassed 9-bit binary exponent ie. True exponent is exp-256.

9.2 Pointers

Tags	Byte2	Byte1	Byte0	
111	LS			22 bit byte-address of block.

LS: S (bit22) =0 -> firm pointer; =1 -> shaky pointer.
L (bit23) =0 -> open pointer; =1 -> locked pointer.

9.2.1 References

Tags	Byte2	Byte1	Byte0	
000	PQ			Word 1 22 bit displacement from:
111	LS			Word 0 22 bit byte-address of block

PQ (2 bits) = 0X -> word displacement
= 10 -> byte displacement
= 11 -> bit displacement.

9.2.2 Vectors

Tags	Byte2	Byte1	Byte0	
000	PQ			Word 2 22 bit displacement from:
111	LS			Word 1 22 bit byte-address of block
000				Word 0 Integer upper bound.

9.2.3 Arrays

Tags	Byte2	Byte1	Byte0	
000	PQ			Word 4 22 bit displacement from:
111	LS			Word 3 22 bit byte-address of block
000				Word 2 Integer upper bound.
000				Word 1 Integer stride.
000				Word 0 Integer lower bound.

1-dimensional array.

9.3 Blocks

9.3.1 Work spaces

Tags	Byte2	Byte1	Byte0		
XXX				Word 5	Logical start of block.
111	L			Word 4	Current procedure.
000				Word 3	Sf dump rel to current ws.
000	IT			Word 2	Pc dump rel to current code
111	R			Word 1	Pointer to last ws.
001				Word 0	Overhead word = block size (with overhead)in bytes.

R (bit 23) =1 -> current ws has been loaded.
I (bit 23) =1 -> current code runs in privileged state.
T (bit 22) =1 -> current code runs in T-state.

Word 1 may be zero (with zero tags) for 1st ws of process.

9.3.2 Code blocks

Tags	Byte2	Byte1	Byte0		
XXX					
000				Word 3	Logical start of block
				Word 2	Zero or shaky ptr to ws
000				Word 1	Byte size for work space.
010				Word 0	Overhead word = block size.

Word 3 is zero-th constant, cannot be written to, and is the byte displacement from word 0 of the start of the code within this block

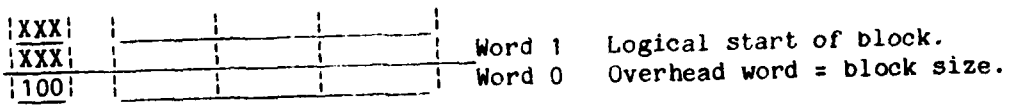
If word 2 is a pointer it is a shaky one to a workspace suitable for this code block; this chain of shaky pointers is continued through word 1 of the workspace.

9.3.3 Type 3 blocks

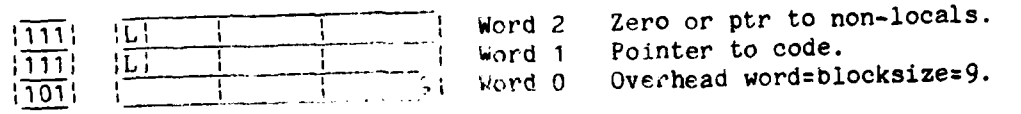
Tags	Byte2	Byte1	Byte0		
000					
000				Word 1	Logical start of block.
011				Word 0	Overhead word = block size.

Pointers will not be preserved in type 3 blocks.

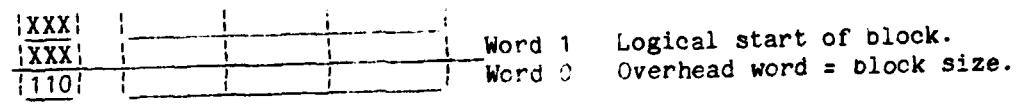
9.3.4 Normal blocks



9.3.5 Closures



9.3.6 Keyed blocks



9.4 Disc pointers

A disc pointer in main memory is a locked pointer to a keyed block (type 6):

Tags	Byte2	Byte1	Byte0		
XXX				Word 4	Possible alias for disc value
000				Word 3	Address for pointer on disc
000			U | T	Word 2	Tag for pointer on disc.
111				Word 1	File store key.
110			15	Word 0	Overhead word = block size.

On disc a disc pointer is represented by 4 bytes, the first coming from the least significant byte of word 2, and the remaining three from word 3.

U in word 2 = Unit number in filestore (0 <= U <= 15);
T in word 2 = Type of value corresponding to disc pointer
(1 <= T <= 15)

Disc pointer types:

- T = 1 => Will always be aliased with existing value in main memory.
- T = 2 => Block on disc is read in as code-block.
- T = 3 => " " " is read in as procedure with no non-locals.
- T = 4 => As T=3 but procedure is locked.
- T = 5 => Block on disc is read as procedure.
- T = 6 => As T=5 but procedure is locked.
- T = 7 => Disc pointer is non-writeable disc reference.
- T = 8 => Disc pointer is disc reference.
- T = 9 => Block on disc is normal block which can contain pointers.
- T = 10 => Block on disc is normal block which cannot contain pointers.
- T = 11-15 => Unassigned and unused.

9.5 System block

Word 1 } Error words at failure; Word 1 also has proc during
Word 2 } load_int and current work_space during scavenge.

Word 3 - Size in bytes of last demand for store during garbage collection.

Word 4 - Procedure invoked by failure;
PROC(4 WORD) VOID fail_proc.

Word 5 - Procedure called when demand for space is not satisfied;
PROC(ANY)ANY scavenge.

Word 6 - Procedure invoked by comflex interrupt;
PROC VOID com_int.

Word 7 - Procedure invoked by sbc data interupt;
PROC VOID.

Word 8 - Procedure invoked by sbc channel free interupt;
PROC VOID.

Word 9 - Procedure invoked by expiry of interval timer;
PROC VOID timer.

Word 10 - Procedure invoked by interrupt 0;
PROC VOID.

Word 11 - Procedure invoked by interrupt 1;
PROC VOID.

Word 12 - Procedure called when exiting from proc with zero link;
PROC VOID endprocess.

Word 13 - Procedure called when calling a proc with disc ptr for code_block
PROC VOID load_proc.

Word 14 } REF to hash table containing all disc ptrs in main
} memory;

Word 15 } REF 256 PAIR where PAIR = (shaky disc ptr, PTR PAIR or zero).

Word 16 - Procedure d_to_b accessible by op code 162.

Word 17 - Word accesible by op code 171.

9.6 Exception values

Error(0,0) - wrong type of value in U ; If the value in U is illegal, then the exception value will come from the illegal.

Fail(0,1) - index out of bounds.

Error(0,2) - vector check fail (op code 58).

Fail(0,3) - integer arithmetic overflow.

Error(0,4) - wrong type of block.

Error(0,5) - a or sz displacements wrong in some way, usually too big

Error(0,6) - stack overflow in current work space.

Error(0,7) - stack underflow in current work space.

Error(0,8) - attempt to access outside a block.

Error(0,9) - attempt to jump outside code block.

Error(0,10) - attempt to use a pointer of the wrong sort in accessing disc.

Error(0,11) - attempt to use a privileged instruction without privilege.

Error(0,12) - operand on tos is of wrong type.

Error(0,13) - attempt to open keyed block with wrong key.

Error(0,14) - attempt to access locked block.

Error(0,15) - attempt to dereference nil.

Fail(0,16) - real arithmetic overflow.

Error(0,17) - illegal op code.

END

DATE
FILMED

1-82

DTIC