Abstract PROLOG machine

= a specification

F.G. McCabe
May 1983

DOC 83/12

Abstract

This document describes an abstract PROLOG machine (APM) suitable as a target machine for a PROLOG compiler. The abstract machine described here is a specification of a family of such machines, it does not define a particular implementation; though a style of implementation does naturally arise from the specification.

The main characteristics of the APM are that it is quite low level (and hence hopefully easy to implement on a particular hardware/software system) and yet it is still quite 'clean'. For example the state of the machine can be regarded as a PROLOG term, this property is very useful for certain system-level components of a complete PROLOG system.

The machine consists of a number of registers, with a suitably large amount of uniformly addressable memory associated with the machine. The memory is assumed to have memory management in the form of a garbage collector of some flavour.

This paper is not intended as an introduction to the concept of the Abstract PROLOG Machine. In particular there is no discussion of PROLOG, compiling it to APM code, the programming environment of the APM and a whole host of other important issues.

The intention is to provide a complete specification of the APM sufficient to enable a programmer to implement a version of it for himself on a new computer system.

Keywords: PROLOG, Logic Programming, Computer Architectures.

Department of Computing
Imperial College
180 Queen's Gate
London SW7 2BZ

Telephone: 01-589-5111 Telex: 261503

Contents

1. The registers of the Abstract PROLOG machine

There are 9 registers in the APM, each of which can hold a single PROLOG object. Each of the various registers is dedicated to a particular purpose. The registers are:

C	Instruction stream context. A tuple of instructions forming the current context.
O	Instruction stream offset. The offset within the context where the current instruction being executed is. The offset is an integer which 'points' to the first word of the current instruction.
T	Accumulator. Holds a term during unification
L	Local Environment. Tuple of variables in the clause
G	Goal register. Remaining calls to be executed on success
S	Stack. Contains backtracking points
R	Reset list. List of variables bound
INTERRUPT	Holds a tuple of interrupt service process descriptors
ERROR	Holds a tuple of error response process desciptors

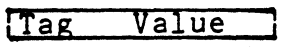
Remark about references to variables

Each of the registers can hold a word which can represent any PROLOG object. HOWEVER NO register can be a variable. A register with a variable as its value is achieved by having the register be a variable link to the actual variable in memory. **In general variable-variable bindings are completely transparent.**

2. The memory architecture of the APM

Memory is addressable in words and in multiples of words. There is no assumed segmentation structure, in particular there is no assumption which identifies addresses with integers. However, a smoother implementation will result if a uniform address space can be used.

A word has a structure consisting of a TAG field, and a value field. Where the tag is a number in the range 0 - 7, and Value is either an integer or APM word address. There may be other system fields, to hold garbage collection information for example, depending on the implementation.



The tag gives the 'type' of the word, individual tag values are discussed below. Since all tags are in the range 0 to 7 only 3bits are required to represent it.

The value field is at least big enough to hold a full memory address or a single precision integer. The actual size of neither the value field nor a complete APM word are defined, nor is this information easily determinable by a running APM program.

2.1 Word types

Unbound variable UNB tag (0)

An unbound variable is signalled by the UNB (0) tag in the word. The value field is also normally zero (or the address of the zeroth APM word). If the value field is not zero then it is the address of a process descriptor. Certain operations on such a variable will cause the process descriptor to be triggered and entered. See Section 5 for a more complete description of process descriptors and other exception conditions.

Link Word LINK tag (1)

A link word is a synonym for another word. The value field of a link word is the address of another word which actually contains the object represented by the link. Thus for almost all purposes it is as though the link word were replaced by the word referenced. Of course the 'target' of the link may also be a link, in general an arbitrary chain of links can be set up. These links have to be followed to access the real value of the word.

Link words are the main mechanism used to make variable-variable bindings. Through the use of links many references to a single logical variable can be constructed. When that variable becomes instantiated then the value is 'transmitted' simultaneously to all links which referenced the variable.

Since logical variables can occur almost anywhere in the system it is important that the system automatically dereferences such links at all times. In particular links can occur within and be part of the executable object code of the PROLOG programs.

The only exceptions to the automatic dereferencing of link words is in the addressing of operands of instructions. See section 3 for a discussion of when links are followed or not in this situation.

NIL word NIL tag (2)

The NIL word is used to represent the empty list, or the end of a non-empty list. The value field of a NIL word is essentially irrelevant, however it is permissable to assume a value such as -1, or 0. This may simplify certain comparison operations which form part of some instructions.

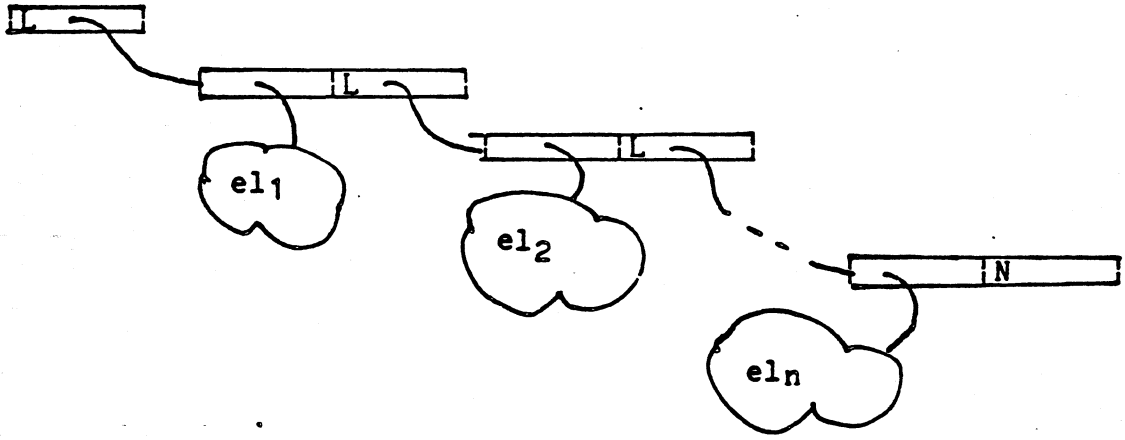
Integer word, INT tag (3)

The integer word is used to represent an integer number. It is guaranteed that at least 16 bits be used to represent integers, allowing all integers in the range -32768..+32767 to be represented by such a word. Of course integers may be larger than this, and it is perfectly possible to represent more numbers in an integer word in a particular implementation. However it is not considered good practice to rely on the range of an integer word to be more than 16 bits since this limits the portability of software.

All instruction function codes are represented by integers in no more than 15 bits. Hence all instruction codes are guaranteed to be positive integers in all implementations of the APM.

List word LST tag (4)

A list word represents a cell in a list. The value field of a list word in the address of a list pair, represented by two consecutive APM words. The address points to the first or zeroth word of the pair. The first is the head of the list, and the second word is the tail word of the list. A list of arbitrary length is constructed by stringing list words together:

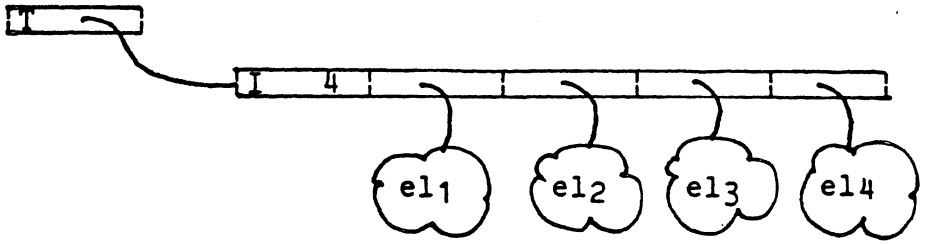


The end of the list is signified by a NIL word in the tail of a list pair.

This method for representing lists uses a **Tagged pointer architecture**, so-called because the tag of each pointer gives the type of object being pointed at. This approach contrasts with the **tagged record architecture** where all pointers are the same but the target (such as the list cell) carry a tag giving the type of the record. In a tagged pointer architecture it is quite possible for different pointers to a given block of words to view the same area of memory as though they were different sorts of objects.

Tuple word TPL tag (5)

A tuple word represents a tuple of objects. The value field of a tuple word contains the start address of a contiguous block of APM words. The 0^{th} word is always an integer and gives the number of entries in the tuple: it is the length of the tuple (not including the length word itself). Tuples are similar to arrays in more conventional systems, in a PROLOG system they principally play the role of fixed length lists.



Real Numbers REAL tag (6)

A complete system often requires the ability to manipulate numbers which are either not whole numbers (such as 3.14) or extremely large (3.0×10^{9}) or extremely small (200×10^{-9}). For this purpose a separate (but compatible) type of number is provided for in the APM.

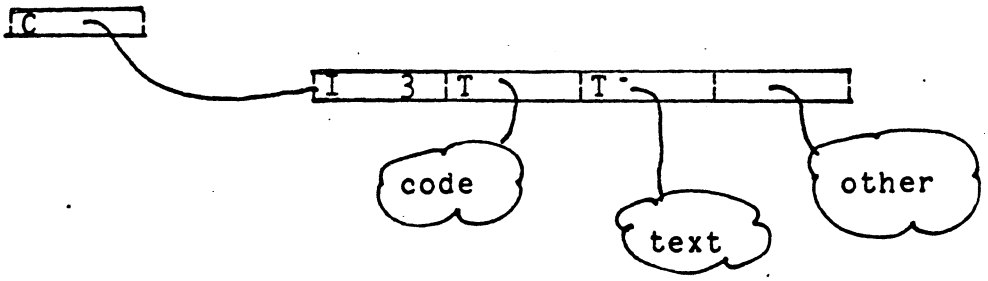
These real (sic) numbers can be represented in any way that is convenient. Usually some kind of floating point numbers will be used but other alternatives exist including rational numbers (the dividend & divisor represented as a pair of integer numbers).

It is possible that a floating point number can be represented inside the value field of an APM word but it is not required. It is perfectly allowable for the value field to be the address of some structure giving the number. Again this could be in a separate array of floating point numbers or some combination of APM words. There are no operations in the APM which look inside such numbers, only operations of the number as a whole are performed. This gives considerable freedom in the representation of this class of number.

A separate type of integer is used because many integers will fit in a value field of an APM word. While this constraint is not strictly necessary it is included as a guide to a reasonably efficient implementation. Integers are so frequently needed (they form the bulk of the object code of PROLOG programs) that it is desirable to optimize them separately.

Constants CON tag (7)

Constants are essentially indivisable symbols as commonly manipulated by PROLOG programs. They are represented in the APM as a unique reference to a special data structure whose main role is to represent various properties of the constant. The value field of the constant word points to this structure which is actually a 3-tuple. Two constants are the same only if they both reference the same APM structure. This is common but not universal practice in symbol manipulation systems such as LISP and PROLOG.



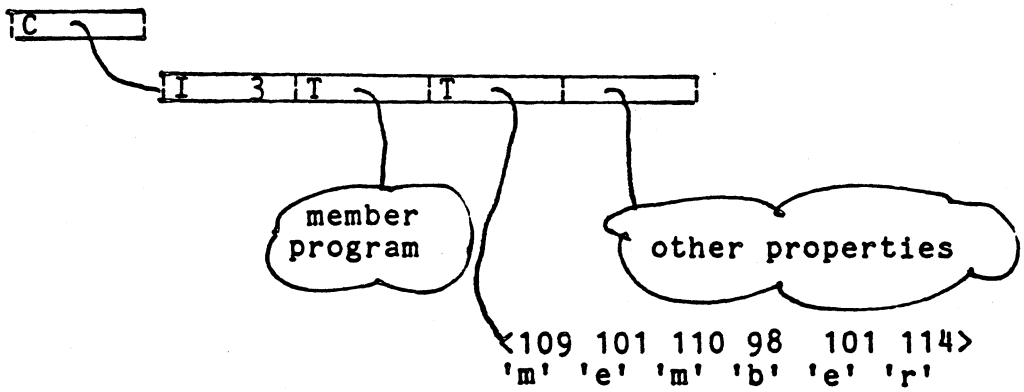
Constants have a number of roles in the APM system. They are used for predicate symbols, file names, names of programs (modules) and many other uses. Hence a given constant can have a number of properties, including the string of characters which form the print name of the constant. Other common properties are object code (for compiled PROLOG programs), source code, module structures, file control structures and so on.

The two main properties of principal concern to the APM are the print name and the object code. Other properties are possible but not directly manipulated by the APM itself. To avoid the cost of a complex property list structure for the essential code and text properties they are represented by special entries in the main control sructure. The first is the code property (a tuple) which occupies the first entry in the constant tuple; and the second is the text of the symbol.

The text is represented by a tuple of integers which form the character codes which make up the print name of the constant. Usually these will be ASCII codes but others are not excluded.

The third element of the constant's structure is left undefined at this stage. It is expected that it will be used as the entry point for a generalized property list structure. No operations in the APM itself make any assumptions about the contents of this field.

A complete constant may be quite a complicated object, for example the constant "member" might look like:



2.2 Garbage collection

The memory is assumed to be self garbage collecting, there are instructions to create new list cells and tuples but none to explicitly de-allocate them. Each word, even if it is part of a list or tuple, is individually garbage collectable. In particular there may be references to components of tuples & list cells as well as references to whole tuples and list cells. This possibility arises from the fact that link words (logical variables) can link two arbitrary data structures together.

The actual garbage collection scheme which should be used is

undefined, though there are a number of possibilities. A mark & collect scheme is probably the most feasable for a small 'soft' implementation of the system. In such a system each word of a list or tuple would be marked and during the collect phase unmarked words collected together into some kind of free list.

Another possibility is maintain a separate bit map in another part of the system memory. This bit map would have 1 bit for each word in the memory, and new cells are allocated by searching the bit map for a contiguous group of free bits. (This means that a garbage collector only needs to mark, it does not need to scan & collect, at the cost of searching for each new cell)

2.3 Correspondence between types

To ensure portability of APM code accross different implementations there are no assumptions about the word length of the APM. Furthermore, there is no assumed mapping from integers to addresses apart from that defined in the addressing modes.

For example, it is possible to 'change the type' of a word using various instructions. If the type of a word is changed from an address of one type to an address of another type then the address is still valid. However, if the type of a word is changed from an integer to an address of some kind then it is **not** guaranteed to be valid. Nor would converting an address to an integer necessarily produce a sensible result.

3. Abstract PROLOG machine instruction & address specification

The instruction stream consists of a tuple of integer opcodes and operands. Each instruction forms a segment of the tuple consisting of one to four elements of the tuple. The first element of the instruction's segment is a 15bit (hence positive) integer. This integer encodes the operation to be performed and a specification of the operands of the instruction. The encoding scheme is discussed in detail below.

The operands of an instruction can evaluate to any legal PROLOG term; i.e. they may be integers, constants or even sub-tuples of instructions to execute. The actual type of operands expected by given instructions will vary with the individual instruction.

Notice that the instruction stream is a valid PROLOG object, albeit one in a particular format. This gives immense flexibility to the system, and also means that a PROLOG compiler in PROLOG can generate the 'machine code' for the APM (without recourse to a special assembler). It also means that a hardware system does not need two languages (the second is the non-PROLOG version of the APM [or Concrete (sic) PROLOG machine {CPM!}]).

3.1 Instruction operands

Instructions can have zero, one, two or three operands. These operands usually respresent data that is being manipulated by the instruction however they can be sub-streams of execution (see for example the CALL instruction).

There are no particular restrictions as to the type of objects pointed to by the operands. Often they can be any legal PROLOG object; though some instructions do make certain assumptions about their operands. For example, the TRY instruction expects its operand to be a tuple in the format of a stream of instructions to execute.

The assumed types of operand an instruction may have are not always checked, if they are checked then a 'Bad operand' error may be signalled.

The operands of an instruction can be computed in a variety of addressing modes. The detailed meaning of an operand depends on whether it is being used as the **destination** address of an instruction or the **source** or **auxilliary** address.

When used as a source or auxilliary address the operand is the contents of the register or structure element, except when it is a variable in which case the operand evaluates to a link to the variable (i.e. the address of the variable).

When used as the destination the operand is nearly always the address of the register or cell described. The available addressing modes are: (using terminology of conventional register architectures)

Literal (address mode 0) - the operand is the next element occurring after the initial opcode word in the instruction stream.

The literal mode is only allowed as the source or auxilliary operand of an instruction; the use of literal mode as a destination address is undefined. In some implementations using a literal as a destination will signal a 'Bad operand' error.

Register - the operand is one of the registers

T	address mode 1
L	address mode 2
G	address mode 3
S	address mode 4
R	address mode 5

Recall that this means the contents (after pursuing any variable references i.e. links) of the register when used as a source, and the register itself when used as the destination.

Offset - where the element following the instruction opcode in the instruction stream is a positive integer which is an offset to an address contained in one of the registers. The normal way the offset mode is written in symbolic APM 'assembler' is:

<reg>[<offset>]

where **<reg>** is either the T register (address mode 6) or the L register (address mode 7), and **<offset>** is a small positive integer whose range is determined by the local limitation on sizes of tuples. For example to specify the 1st element of the L-tuple (i.e. the local variables) use:

L[1]

If the T register (say) contains a list, then the head of the list is accessed by using the form:

T[0]

and its tail is accessed via:

T[1]

If T contains a tuple, then the length of the tuple in T is accessed by:

T[0]

If an instruction uses the offset addressing mode (or literal addressing mode) for one or more of its operands then the offsets (and/or literal data) follow the initial instruction integer in the instruction stream.

In these circumstances the literal data involved in the destination address comes before the literal data needed in the source address, which in turn comes before the offset for the auxilliary address computation. If two operands are specified then the destination comes before the source in the instruction stream; and the source comes before the auxilliary address. For example, the instruction:

L L[1],T[3]

loads the first local variable with the third argument of the T register. This instruction is coded up as the segment of the instruction stream:

< ... 31755 1 3 ... >

Alternatively to add 23.5 to the third local variable putting the result in the fifth the instruction

```
ADD L[5],L[3],23.5
```

could be used. This would be encoded as the sequence:

```
< ... 32277 5 3 23.5 ... >
```

3.2 Instruction encoding

Although instructions are described by a positive integer, for the purposes of encoding the instruction's addresses and function code it is better to see the number as a string of 15 bits lying in the least significant part of the integer value.

The function code of the instruction is a number in the range 0-63 which is coded up in a 6bit field in the least significant part of the number (bits 0 to 5).

Instructions can have zero, one, two or three operands. Each operand is coded as a 3bit address specifier in the instruction's bit string. The three address specifiers occupy bits 6 to 14.

The auxilliary address mode specification occupies bits 6 to 8 in the instruction format, the destination is in bits 12 to 14, and the source address occupies bits 9-11. A complete instruction operation code with its four sub-fields looks like:

```
|ddd|sss|aaa|ffffff|
 dest source aux function code
```

3.3 Links in addresses

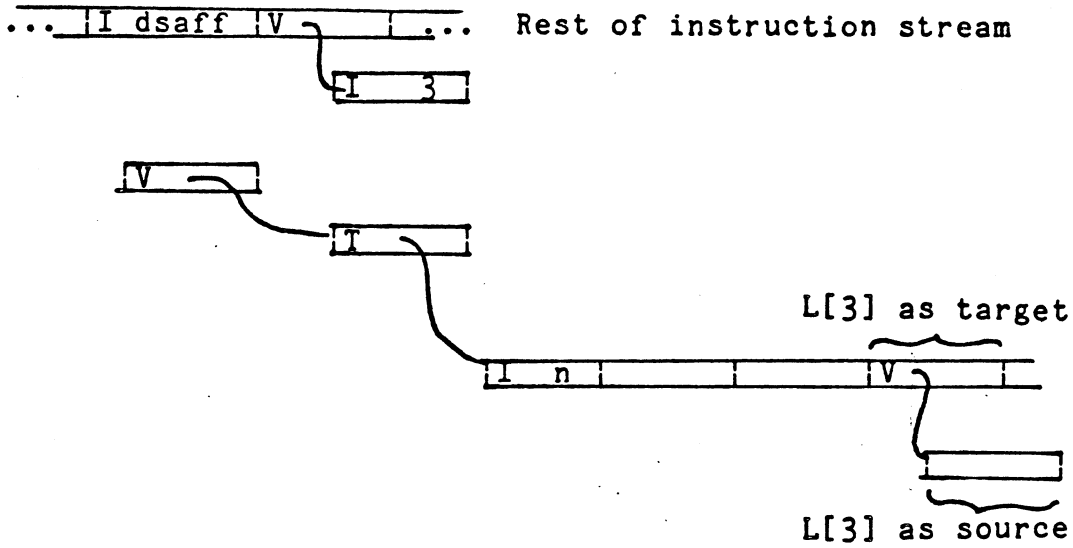
Link words can appear in almost any part of an address computation, up and including the final location specified. For example, suppose the address L[3] is specified as the destination mode for some instruction.

The L register may itself contain a link to another word in memory which points to the tuple. This link is followed when computing the L[3] address.

Similarly the "3", which is part of the address specification, may be reached through one or more link words. In this case the links start from inside the instruction stream itself: since the instruction stream is a normal PROLOG tuple the various members of that tuple may have link words in their data structures.

Finally, the third location in the L tuple may also contain a link word. This link is followed only if the L[3] is a source operand (the contents is being looked for). If L[3] is a target then this final link is not normally followed.

instruction



NOTE Although normally the address contained in the destination field is normally the target location for the instruction this is not always the case. Similarly the address in the source and auxilliary fields are normally sources, but some instructions treat these as targets.

For example, the LIST instruction checks the destination operand to see that it is is list pair. Thus it treats the destination as a kind of source (if the destination turns out to be a variable then it does overwrite it). The source and auxilliary addresses are assigned the contents of the head and tail respectively of the list being matched against, and hence are target rather than source addresses.

4. Abstract PROLOG Machine Instruction set

In this section we give a complete listing of the instruction set of the APM.

In describing each instruction below, the memnonic of the instruction is given together with a bit-wise encoding of the instruction (for bit fanatics only). Also (where appropriate) a register transition diagram is given, followed by a brief English description of the action of the instruction. This diagram displays each register, together with its assumed contents, both before and after the execution of the instruction. The letters stand for the registers, if a value remains unchanged then the same letter is used both before and after.

```
     C' O' T' L' G' S' R'
=>   C  O  T  L  G  S  R
```

Note that the old context is often referred to as just C, rather than repeating the whole argument. Also the new value for the offset is also often expressed as "O+1" or "O+2" etc. This should be read as 'increment the offset by one plus how ever many literal arguments there are to the instruction'. In many cases this will be actually the integer 1, or 2 etc.; however it may be less if the actual operands refer to registers rather than literals or register offsets.

For example, the normal CALL instruction has the transition diagram:

```
     < .. CALL constant .. > O T L          G  S R
=>   constant[1]             1 T L <C O+2 L G> S R where C is old
                                                   context
```

but the instruction "CALL L" has the transition:

```
     < .. CALL L ..> O T L          G  S R
     L[1]            1 T L <C O+1 L G> S R
```

Conventions

The following conventions are used in this document for syntax:

```
( )      list  (X|Y)  element X followed by list Y
< >      tuple
_        undefined/don't care
sss      source address
ddd      destination address
aaa      auxilliary address
???      don't care usually zero
```

Each instruction is written in the form:

<opcode> {{<destination>,}<source>{,<auxilliary>}}

Instructions with their arguments are actually coded up as small (15bit) integers possibly followed by extra integer and other elements in the instruction stream. However we will normally display instructions by using a mnemonic rather than the instruction code itself.

4.1 Control Instructions

The first instruction group implements the main control instructions which are used to compile the top-level problem reduction. There are instructions for calling sub-programs (i.e. for sub-goals in the bodies of clauses) and for trying alternative clauses.

<u>HALT</u> <u>Stop the APM</u> ?????????000000

This is used to halt execution of the APM. In a soft implementation this would initiate a return to the operating system. In a hardware system HALT makes no sense except possibly to turn the hardware off.

<u>CALL Source</u> <u>Execute a program.</u> ???sss???000001

```
     < .. CALL source ..> O T L                G   S R
=>   source[1]            1 T L          <C O+2 L G> S R if const
=>   source[0]            1 T source[1]  <C O+2 L G> S R if PD
```

source should evaluate to a constant **or a process desciptor.** The case where source is a constant is the more usual one, and corresponds to calling a PROLOG program. The first element of the constant's structure taken to be the new instruction stream, typically code that has been compiled, which is inserted into the C (context) register. The O (offset) register is initialised to 1 (the first instruction in the new context).

The old values of the C and O register are saved, together with the L register, in an entry on the G (goal) register. When a subsequent SUCC instruction is executed this information is restored allowing the instructions following the CALL instruction to be executed.

If the source is a process descriptor (i.e. a list pair whose head is a code tuple, and whose tail is a tuple of variables etc.) the head is loaded into the Context register and the tail is loaded into the L register.

This form of call is used to apply lambda expressions; the head of the process descriptor is the compiled body of the lambda expression, and the tail is the vector of values for the **free** variables occurring in the body. The T register holds the values for the **bound** variables of the lambda expression.

<u>SUCC</u> <u>Return successfully</u> ?????????000010

```
     < .. SUCC .. > O' T L' <C O L G> S R
=>   C              O  T L          G S R
```

Pop the goal, collecting the C, O (remaining calls to execute) & L (local env) registers. This instruction is placed after all the calls in the body of a clause; or <u>as the body</u> of an assertion. It is executed when unification is successful, and there are no (more) calls in the body to check.

Because of the existence of the next instruction the main use is actually for unit assertions rather than rule clauses.

LSTCALL Source Call the last subprogram ???sss???000011

```
     <.. LSTCALL source ..> O T L G S R
=>   source[1]              1 T L G S R   if source is a constant
```

This is similar to CALL except that the Goal register is not changed.

The LSTCALL instruction can also 'execute' a process descriptor in a similar way to the CALL instruction:

```
     <.. LSTCALL (code|env) ..> O T L   G S R
=>   code                       1 T env G S R
```

A CALL instruction would be used for every atom on the right hand side of a clause except for the last atom which uses the LSTCALL. This then replaces the SUCC instruction at the end of the clause which is now not necessary.

TRY source Try to use a clause ???sss???000100

```
     < .. TRY source .. > O T L G                  S  R
=>   source               1 T L G <C O+2 T G R S> R
```

This instruction is used to 'try' each clause of a program in turn. The fail return stack is used to store the state of the APM should backtracking become necessary.

The T register is saved by the TRY instruction because it holds the tuple of arguments of the call, the C and O registers are saved to indicate the next clause to try, G is saved as it forms the continuation after the current call has succeeded, and S & R form the current backtracking state of the APM.

The last clause in a program is not normally prefixed by a TRY instruction as there are no more alternative clauses. By not growing the stack for the last clause it avoids the situation where the stack contains entries of the form:

```
<<..FAIL> n ... S>
```

In fact a stack of this form is exactly equivalent to the shorter stack S. A program for a relation is compiled as a sequence of TRY instructions, except for the last:

$$\langle \text{TRY } Cl_1 \text{ TRY } Cl_2 \ldots Cl_{last} \rangle$$

FAIL Backtrack to a previous state ?????????000101

```
     < .. FAIL ..> O' T' L G' <C O T G R S> R'
=>   C             O  T  L G                S  R
```

with variables R'-R reset

The terms pointed to in the reset list R'-R are changed back into unknowns. Note that at this point the old stack frame can be explicitly removed if necessary (as opposed to relying on automatic G/C).

Fail is normally not explicitly executed, but a number of instructions can fail, in this case the FAIL instruction is executed implicitly.

NOP Source No operation/comment ???sss???000110

The comment instruction performs no operation in the APM. However since it potentially has a single argument (which it does nothing to) the NOP instruction can be used to embed comments in the code.

Such comments might be inserted by the compiler to enable the code to be decompiled, or to allow some kind of symbolic debugging of the APM code. The degenerate form of NOP (with the source address specifying no address) is exactly analogous to the NOP instructions of more conventional computers. (Though its application here is not certain.)

e.g. To embed at the head of a compiled clause information describing the variables used in the clause:

```
     < ..NOP (X Y Z Total) .. > 0   T L G S R
=>   C                          0+2 T L G S R
```

Other kinds of comment might include the original text of the clause, or meta-level assertions about the use of compiled clauses.

4.2 Unification instructions

These instructions are used to perform matching and binding operations.

UNIFY left,right Full unification ???lllrrr000111

This instruction implements the full unification algorithm. The UNIFY instruction is used to implement the EQ(x x) test. In other words the UNIFY instruction is typically compiled for the second and subsequent occurrences of variables in the head of a clause.

If during the unification an attempt is made to bind a variable with a non-zero value field, then the instruction is unwound (including undoing any bindings made) and the process descriptor for the variable is activated. On return the UNIFY instruction is re-started.

There may be arbitrarily many bindings of variables made during the course of this instruction. If the UNIFY is succesful then on completion the reset list is grown accordingly.

If the unification fails, then any bindings made are undone, and the next instruction to be executed is "FAIL".

```
     <.. UNIFY L[1],L[3] ..> 0   T L G S R
=>   C                       0+3 T L G S R'
```

where R' - R are all the variables bound during the unification.

The following PROLOG program defines the effect of this instruction (it contains an assignment not normally allowed in PROLOG):

```
UNIFY(left right resets newresets) if
     VAR(left)&
     NOT(VAR (right) & SAMEVAR(left,right))&
     BIND(left right resets newresets)
UNIFY(left right resets resets) if
     SAMEVAR(left right)
UNIFY(left right resets newresets) if
```

```
     VAR(right)&
     NOT(VAR (left) & SAMEVAR(left, right)
     BIND(right left resets newresets)
UNIFY(left right resets newresets) if
     TAG(left tag)&
     TAG(right tag)&
     UNICASES(tag left right resets nresets)

UNICASES(CON left right resets resets) if
     EQSYMBOL(left right)
UNICASES(INT left right resets resets) if
     EQINT(left right)
UNICASES(REAL left right resets resets) if
     EQREALS(left right)
UNICASES(LIST (lefth|leftt) (righth|rightt) resets nresets) if
     UNIFY(lefth righth resets iresets)&
     UNIFY(leftt rightt iresets nresets)
UNICASES(TUPLE left right resets nresets) if
     LENGTH(left length)&              {test the lengths}
     LENGTH(right length)&
     UNITUPLES(left right length resets nresets)

UNITUPLES(left right 0 resets resets) if
UNITUPLES(left right pos resets nresets) if
     NTH(left pos leftel)&
     NTH(right pos rightel)&
     UNIFY(leftel rightel resets iresets)&
     UNITUPLES(left right pos+1 iresets nresets)

BIND(left right resets (left|resets)) if
     left := right
```

TUPLE dest,length Unify a tuple **dddlll???001000**

This instruction tests to see if the Dest is a **length**-tuple or a variable. In the case of it being a variable a new **length**-tuple is created and stored in Dest.

```
     <.. TUPLE dest,length ..> 0    T L G S R
=>   C                         0+3  T L G S R if dest=length-tuple
=>   C                         0+3  T L G S (dest|R)
                                          if var(dest)
=>   <FAIL>                    1    T L G S R otherwise
```

A reset entry is grown on the R register pointing to the word which was bound.

Remarks above concerning non-zero value fields for variables also apply to the TUPLE instruction.

If Dest is a non-variable and not a **length**-tuple then a FAIL instruction is executed.

The TUPLE instruction is roughly analogous to the call:

```
     EQ(dest <_ .. _>)
              ~~~~~
              length
```

MOVE Dest,source,specifier Unload tuple **dddsssaaa001001**

Locations in **dest** are loaded with values from successive locations in **source** as determined by the **specifier**. This instruction is the complement of the FILL instruction, and indeed

is very similar except that the specifier modifies the the destination rather than the source; and no new structures are created.

The locations

source[1] ... **source[n]**

are Loaded into locations

dest[specifier[1]] ... **dest[specifier[n]]**

For example, the instruction

MOVE L,T,<5 6>

is equivalent to the instruction sequence:

```
L    L[5],T[1]
L    L[6],T[2]
```

MOVE is primarily used to 'unload' a tuple during unification. In particular if the unification is matching a tuple of variables all of which are their first occurrence then the MOVE instruction can in one go load up all the values of the tuple or list pair. This is actually quite a common phenomenon with tuple processing programs and by judicious use of extra 'hidden' local variables can be made even more effective.

NOTE that it is possible for this instruction to be used in situations where it makes no sense, for example the source (or destination) might actually not be a tuple, or the specifier may nt be a tuple in the correct form. The implementor may if he chooses to check this, in any case the programmer should make sure that the preconditions are properly fulfilled otherwise anything may happen.

LIST Dest,Head,Tail Unify a list pair **dddhhhttt001010**

The LIST instruction compares the term found in Dest and checks that it is unifyable with a list pair. If Dest is a variable then a new list cell is created and stored in it and a new reset entry is grown on the R register.

If Dest is non-variable and **not** a list pair then the LIST instruction FAILs.

Otherwise the head and tail of the list pair are Loaded into the addresses given by Head and Tail respectiveley (corresponding to Source and Auxilliary modes).

The LIST instruction is anomolous in that it is the only instruction with more than one destination; in fact it can be regarded as having **three** destinations since the source might be overwritten.

As with the TUPLE instruction, the LIST instruction can be regarded as being equivalent to the call:

EQ(dest (head|tail))

4.3 General purpose data movement instructions

This group of instructions shift data about amongst the registers. Most of them may not be used very often by a particular compiler, though some will be very common.

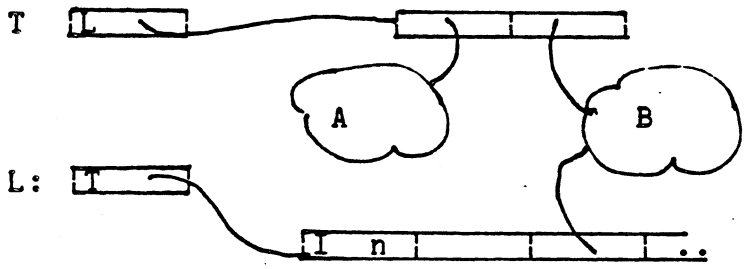
L Dest,source Load register **dddsss???001011**

This instruction loads the destination register with the contents of the source register. If the source was an unbound variable then a link word is left in the destination rather than another variable, otherwise the contents of the source word is stored in the destination.

Note that the destination can have any previous value, it is not restricted to overwriting variables; in fact the L is a form of assignment.

If the T register contains some arbitrary list "(A¦B)", then accessing the tail into the second local variable becomes:

```
     <.. L L[2],T[1] ..> 0   T L             G S R
=>   C                   0+3 T L/[2]:=T[1] G S R
```



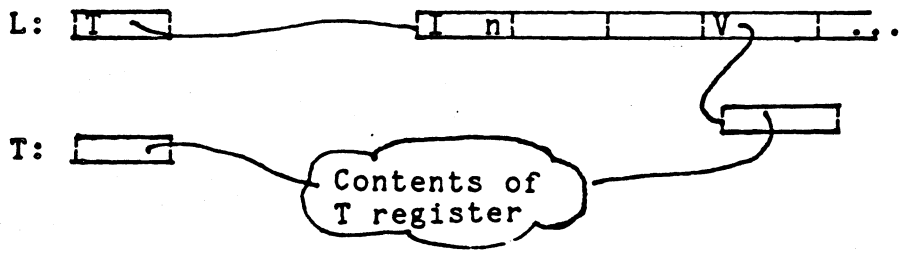
S Dest,source Store register **dddsss???001100**

The S instruction is identical in effect to the L instruction except in one respect. If the destination already contains a variable link then the S instruction uses the value of the link as its destination address rather than the register itself. The L instruction simply overwrites the destination regardless of its prior contents.

The S instruction is one way of binding a non-local variable in a clause: if the variable reference is passed in a register then Storing into that register binds the variable as opposed to re-assigning the register. However it does not grow the reset list by doing so.

For example, to bind the variable referenced by the third local variable to the contents of T:

```
     <.. S L[3],T ..> 0   T L          G S     R
=>   C                0+2 T [L[3]]:=T G S     R
```



If the source and destination are actually **both the same variable** then the S instruction performs no operation. (If the normal mechanisms were to apply a link would be set up in the target word to itself! This is extremely dangerous and could

cause the APM to enter an infinite loop).

LX Dest,source,offset Indexed load **dddsssooo001101**

The indexed load instruction is similar to the L instruction, except that the source address is specified by:

```
source[offset]
```

i.e. it is 'equivalent' to the instruction

```
L    dest,source[offset]
```

This instruction is used to implement the NTH builtin primitive, unify with the n^{th} element of a tuple. The following code would implement the NTH program:

```
NTH: <TUPLE    T,3              ;usage is NTH(tuple, n, element)
     L         L,T              ;no pattern matching needed
     LX        T,L[1],L[2]      ;fetch the nth element
     UNIFY     T,L[3]           ;unify against the element
     SUCC>                      ;all done
```

The auxilliary address is used in this instruction to specify the index location.

4.5 Tag manipulation instructions

These instructions operate on the tag field directly. They are only used by system level programs. (E.g. a constant can be created in this way from a tuple of small integers.)

LTAG Dest,source Load tag **dddsss???001110**

```
     <.. LTAG Dest,source ..> 0   T L G S R
=>   C                        0+3 T L G S R    dest:=tag[source]
```

The destination is replaced by an integer word whose value is the TAG field of the source. For example, the TAG field for a constant is 7 then if the accumulator was a constant, LTAG T,T leaves the integer 7 in the accumulator. (The accumulator should previously have been stored.)

STAG Dest,source Set tag field **dddsss???001111**

Ths instruction is similar to the S(tore) instruction except that only the tag field of the destination is affected. The tag field is taken from the least significant three bits of the value field of the source. This instruction is used for example, to create a new constant having constructed a tuple of the required form:

```
STAG 7,T
```

Type conversions which have been effected using this instruction are not always guaranteed to be valid. So long as an address type is converted to an address of another type then the conversion is valid.

However, for those instances where either an integer is converted to an address, or where an address is converted to an integer the results are not defined.

TAG source,aux Check tag of source ???sssaaa010000

This instruction checks the tag of source is the same as the integer contents of aux. For example, to check that the 2nd argument of T is a constant use:

```
    <.. TAG T[2],7 ..> 0    T L G S R
=>  C                  0+3  T L G S R      if T[2] is a constant
=>  <FAIL>             1    T L G S R      otherwise
```

4.6 Structure building instructions

These build lists and tuples directly. They all leave a pointer to the structure in the destination.

CONS Dest,head,tail Construct a list pair dddhhhttt010001

A new list pair is created and loaded into the destination. The head of the new list pair is Loaded from the source address, and the tail of the list pair is Loaded from the Auxilliary address. For example, to put a new unit list in the accumulator whose head is the 2nd local variable then we have:

```
    <.. CONS T,L[2],() ..> 0    T       L G S R
=>  C                      0+4  (L[2])  L G S R
```

TPL Dest,source Construct a tuple dddsss???010010

The destination is left with a tuple of length defined by the value field of the source register. Each element of the tuple is initialized to unbound. This instruction can be used to create a new local environment on entering a new clause, as in:

```
    <.. TPL L,n ..> 0    T      L  G S R
=>  C               0+3  T <      > G S R
                         n vars
```

EXTEND Dest,source,length extend tuple dddsssaaa010011

The EXTEND instruction is used to extend a tuple with unbound variables. **source** should evaluate to a tuple whose length is less than (or equal to) the **length** operand. A new **length**-tuple is created in **dest**, and the first few locations are filled by copying the source tuple into the new tuple.

The instruction is equivalent to the instruction sequence:

```
TPL  temp,length                ;create a length tuple
L    temp[1],source[1]          ;copy the source tuple into it
.         .
.         .
L    temp[source[0]],source[source[0]]
L    dest,temp
```

where temp is some internal temporary register.

The main use for EXTEND is in the bodies of lambda expressions; it is used to extend the environment containing the free variables of the lambda expression at the time of the call to include the other local variables needed to execute the body of the lambda expression.

FILL Dest,source,specifier fill-in tuple dddsssaaa010100

A new tuple the same size as the specifier is created in dest, and then it is filled by values from source.

Successive locations in **dest** are Loaded with values from the **source** as determined by successive locations in **specifier.**

If the specifier is a **n**-tuple then locations

dest[1] .. dest[n]

are Loaded with values

source[specifier[1]] .. source[specifier[n]]

respectively.

For example, the instruction

FILL T,L,<2 3 5 0>

is equivalent to the instruction sequence:

```
TPL  temp,4
L    temp[1],L[2]
L    temp[2],L[3]
L    temp[3],L[5]
L    temp[4],L[0]
L    T,temp
```

where temp is some internal temporary location.

The specifier must therefore always be a tuple containing only integers that can be used as offsets to the source of the instruction.

4.7 Arithmetic Instructions

These implement the arithmetic operations needed by a PROLOG system. The arithmetic instructions operate on both integer and real number operands. Either type may be used freely with any of the instructions. Any results that are computed will be represented using an integer word if possible, otherwise a real number is used (which may involve constructing it from the heap) to represent the result.

Essentially an integer word will be used if the result is an integer which can fit naturally into an integer word as defined by the word length of the actual implementation (at least 16bits). Because the integer size is guaranteed to be at least 16 bits, then it follows that if the result of an operation is an integer in the range -32767..32767 then an integer word can be (and will be) used.

The arithmetic instructions are all three address instructions.

ADD Dest,source,aux Add two numbers together dddsssaaa010101

```
     <.. ADD Dest,source,aux ..> O    T L G S R
=>   C                           O+4  T L G S R   dest:=source+aux
```

If the result of the addition is an integer in the range defined by the implementation then the result will be represented by an integer word. Otherwise the destination will contain a real number word which may well be a pointer to a real number as described in section 2.

If the source and auxilliary are not both numbers then a 'Bad operand' error will be signalled. If the result of the addition is not containable as a number (either its too small or too large) then an 'Overflow error' or 'Underflow error' is signalled.

SUB Dest,source,aux Subtract two numbers **dddsssaaa010110**

```
     <.. SUB Dest,source,aux ..> 0   T L G S R
=>   C                           0+4 T L G S R   dest:=source-aux
```

The remarks above about the representation of the result of the operation also apply to this instruction: the destination will be represented as an integer if it is possible within the implementation.

MUL Dest,source,aux Multiply two numbers **dddsssaaa010111**

```
     <.. MUL Dest,source,aux ..> 0   T L G S R
=>   C                           0+4 T L G S R   dest:=source*aux
```

MUL multiplies the two numbers in the source and aux and loads the result into the destination.

DIV Dest,source,aux Divide two numbers **dddsssaaa011000**

```
     <.. DIV Dest,source,aux ..> 0   T L G S R
=>   C                           0+4 T L G S R   dest:=source/aux
```

The destination is left with a number which represents the quotient of the division. This quotient will be an integer word if the quotient is an integer in the right range, otherwise a real number will be used.

A note on arithmetic overflows etc

Arithmetic overflows and divide by zero result in an error trap. See below for discussion on the way errors are handled.

INT Dest,source Find the nearest integer **dddsss???011001**

The destination is replaced by an number which the nearest integer (towards zero) of the source. Of course, if the source is already an integer then INT amounts to a L instruction.

Note that the nearest integer to a large real number might still be too large to be represented using an integer word. In this case a real number is used to represent the integer.

```
     <.. INT Dest,source ..> 0   T L G S R
=>   C                       0+3 T L G S R
```

dest:=sign(source)*floor(abs(source))

LESS source,aux Arithmetic inequality ???sssaaa011010

This instruction compares the number in source with the number in aux. If the source number is numerically smaller than the auxilliary number then no operation is performed, otherwise the instruction FAILs.

The numbers involved can be either integers, or real or a mixture.

```
     <.. LESS source,aux ..> 0   T L G S R
=>   C                       0+3 T L G S R      if source<aux
=>   <FAIL>                  1   T L G S R      otherwise
```

GE source,aux Greater than or equal ???sssaaa011011

This instruction is the complement of the LESS instruction, it succeeds if the source number is greater than or equal to the auxilliary number.

```
     <.. GE source,aux ..> 0   T L G S R
=>   C                     0+3 T L G S R      if source≥aux
=>   <FAIL>                1   T L G S R      otherwise
```

BOR Dest,source,aux Bitwise OR dddsssaaa011100

The destination is replaced by the bitwise OR of the two integers in source and aux.

```
     <.. BOR Dest,source,aux ..> 0   T L G S R
=>   C                           0+4 T L G S R
                                     dest:=source OR aux
```

This instruction assumes that the two numbers are integer words and generates an integer word in the destination.

BAND Dest,source,aux Bitwise AND dddsssaaa011101

The destination is replaced with the bitwise AND of the two source integers.

```
     <.. BAND Dest,source,aux ..> 0   T L G S R
=>   C                            0+4 T L G S R
                                     dest:=source AND aux
```

BXOR Dest,source,aux Bitwise XOR dddsssaaa011110

The destination is overwritten by the bitwise XOR of the two integers in source and auxilliary.

```
     <.. BOR Dest,source,aux ..> 0   T L G S R
=>   C                           0+4 T L G S R
                                     dest:=source XOR aux
```

BNOT Dest,source Bitwise one's complement **dddsss???011111**

This instruction performs a one's complement of the source (which should be an integer).

```
     <.. BNOT Dest,source ..> 0   T L G S R
=>   C                        0+3 T L G S R      dest:=NOT source
```

Note If the operands to either the BOR, BNOT, BXOR or the BAND instructions are not integers or have more than 15 significant bits then the result is **not** guaranteed to be portable. In general the explicit use of such numbers is discouraged.

4.8 Miscellaneous instructions

In this group are a miscellaneous collection of instructions which interface with the APM's environment and set up the various interrupt handling facilities.

Because of the way the APM is (deliberately) isolated from the outside world in which it is embedded, it is difficult to communicate 'control' information to and from the APM.

To enable a running PROLOG program to actually perform I/O a special ESCAPE instruction is used. This instruction is used to connect physical files and devices to logical numbers as well as actually performing the I/O.

SERROR Source Load the Error register **???sss???100000**

```
     <.. SERROR source ..> 0   T L G S R
=>   C                     0+2 T L G S R
```

Source should evaluate to a tuple of 11 process descriptors; The tuple of process descriptor is loaded into the ERROR register, and from now on internal errors that arise will activate process descriptors from the tuple. If Error i is signalled then the i^{th} element of the tuple is used for the process descriptor for that error.

SINTRUPT Source Load the Interrupt register **???sss???100001**

```
     <.. SINTRUPT source ..> 0   T L G S R
=>   C                       0+2 T L G S R
```

Source evaluates to a tuple of process descriptors which is loaded into the INTERRUPT register. This process descriptor now forms the interrupt handler; i.e. any external 'hardware' interrupts will cause this process to be activated.

EI Enable interrupts **????????100010**

```
     <.. EI ..> 0   T L G S R
=>   C          0+1 T L G S R        Interrupts are enabled.
```

If an external interrupt is detected by the machine then the relevant interrupt process descriptor is triggered. The APM starts off life with interrupts disabled.

DI Disable interrupts ????????100011

Prevent incoming interrupts from being handled. Any incoming interrupt will be serviced if it is still pending when interrupts are re-enabled.

ESCAPE source Invoke external function ???sss???100100

Source should be an integer in the range 0-**maxescape**, where maxescape is the implementation defined number of external functions supplied with the APM. It is similar to a CALL instruction except that instead of calling an APM program, an assembler or C (or whatever) program is called instead. The accumulator T is used to pass the arguments to and from the external function.

```
    <.. ESCAPE source ..> 0    arguments L G S R
=>  C                     0+2  arguments L G S R
```

No other registers are (normally) affected by the ESCAPE instruction; however an individual ESCAPE code may well as part of its function alter a register.

The number and layout of the various escape functions is inevitably implementation dependant; however it is suggested that a minimum number of them be implemented, and for the sake of portability the initial numbering should be as follows:

```
ESCAPE 0  <directory-name>   mount a directory
ESCAPE 1  <port file>        open file into port
ESCAPE 2  <port file>        create file into port
ESCAPE 3  <port file>        open file for append into port
ESCAPE 4  <port>             close port down
ESCAPE 5  <file>             delete file
ESCAPE 6  <file1 file2>      rename file1 to file2
ESCAPE 7  <port position>    return the position of the port
ESCAPE 8  <port position>    set position of the port
ESCAPE 9  <port char>        read character(integer) from port
ESCAPE 10 <port char>        write character(integer) to port
ESCAPE 11 <port token type>  read token from port return type
ESCAPE 12 <port token>       write token to port
ESCAPE 13 <port term>        read a term in object code form
```

In a hardware implementation there is no other underlying hardware so this escape instruction does not make sense; however an escape instruction might well be useful as a way of communicating with a co-processor. In any case it is assumed that users (as opposed to compiler writers) will not be making direct use of the escape function.

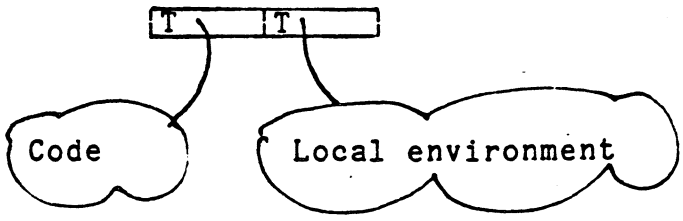
5. Interrupt and error handling

5.1 Process decriptors

Interrupts, errors and unification interrupts are all handled in a uniform manner using an object called the **process descriptor.** The process descriptor is a data structure which describes a suspended goal or process; it is similar to the body of a clause. The suspended program might be responsible for binding a variable, responding to an interrupt, patching an error (such as no clauses) or handling special operating system functions.

The process descriptor is a list cell which contains the compiled PROLOG code to execute in the head cell, and data (usually a tuple of local variables) in the tail cell.

A Process Descriptor (PD) looks like:



When a process is activated it is as though a CALL has been executed. Thus the Goal register is grown as for a normal CALL, and the environment of the process decriptor is installed in the L register, and the code of the process descriptor is installed in the C register, the O register being set to 1.

The only difference between an ordinary CALL and a resumed process is that the latter can occur almost anywhere, in particular in the case of an interrupt it can occur during the unification process. So the process would normally also have to 'save' other registers (in particular the T register) so that when resuming the interrupted process all the registers are left intact.

Thus on an error or interrrupt the state of the APM becomes:-

```
     C       O T L                          G  S R
=>   Service 1 T service-variables <C O L G> S R
```

If the interrupting service code uses other APM registers then their values should be explicitly stored and restored by the service code itself.

The three registers affected by an interrupt are the instruction stream, which will now contain the compiled PROLOG program used to service the interrupt, the local variables register which now contains the local variables of the service code, and the G register which points to where to resume after the interrupt has completed.

5.2 Interrupts and errors

Two special registers allow the processor to service interrupts error conditions.

The INTERRUPT register contains a tuple of process descriptors which describe code which is executed whenever interrupts are serviced. The ERROR register contains a tuple of 11 (the number

of errors detected) process descriptors for servicing error conditions.

Interrupt handling

An interrupt is an event which occurs external to the APM but which must be recognised by some program in the system. Typically this is some kind of I/O event such as a key being pressed, or data becoming available at an input port.

The APM only responds to such events if the EI instruction has been executed, until a DI is executed.

With each possible kind of interrupt is associated a small integer: the interrupt number. The interrupt number is used as an index into the tuple of process decriptors previously set up to handle interrupts. It is the responsibility of the system designer to ensure that a process descriptor exeists for each possible interrupt. The tuple of process descriptors used for servicing interrupts is exactly analogous to an interrupt vector in more conventional machines.

In a software implementation which is embedded in a host operating system it may not actually be possible or desirable to detect many interrupts. The two most useful interrupts in such a system would be a keyboard break (the user types ^C) and possibly somekind of clock interrupt. This latter might be based on a real time clock or simply a count of the APM instructions executed.

Because it can be quite expensive to continuously poll the keyboard between each instruction it is suggested that only certain instructions actually do this. An implementor might, instead, only poll the keyboard during each execution of the CALL instruction (say). When a ^C is detected **then** the interrupt process is triggered.

In a hardware system the low level polling is accomplished by hardware circuits. This therefore means it is quite reasonable to poll for interrupts between each instructions.

Error traps

Various internal conditions give rise to an ERROR trap. These are errors which are detected during the execution of a program, for example a division by zero raises error trap 1.

Error traps result in a process from the ERROR tuple being activated. Since one of the available error conditions is actually the 'no space left' it is important that the handler for this error **does not use any space** until such time as it is guaranteed that there is space to execute.

Usually this means that the process decriptor for the space error must deliberately remove from the system enough data so that a normal return to the top-level supervisor is possible. This might mean, for example, losing the current execution of the user's program and restarting the system with some kind of warm start.

In fact the space error represents a particular problem since the context switch to the error handler itself involves growing the G register. To get around this will involve some fiddling of the space allocation method to keep in reserve a special space for the space error to 'grow into'.

The error codes actually available may vary from system to system, but should include the following: