University of Glasgow
Department of Computing Science

Lilybank Gardens
Glasgow G12 8QQ

University of St Andrews
Department of Computational Science

North Haugh
St. Andrews KY16 8SX

The Persistent Store Machine

Persistent Programming
Research Project 18

The Persistent Store Machine

W.P. Cockshott

Department of Computing Science
University of Glasgow
14 Lilybank Gardens
Glasgow G12 8QQ

Persistent Programming Research Report 18

INDEX

Introduction 1

1. Memory Structure 2
   1.1 PIDS 3
   1.2 Word Alignment 4
   1.3 Memory Management 5
      1.3.1 PIDLAM 5
      1.3.2 Size Relation 6
      1.3.3 Mapping 6

2. Transactions 8
   2.1 Recovery 8
   2.2 Nesting 9

3. Registers and Addressing 11
   3.1 Programmable Registers 11
      3.1.1 A - Accumulator 11
      3.1.2 Segment Registers 11
         3.1.2.1 C - Code Base 12
         3.1.2.2 V - Variable Base 12
         3.1.2.3 D - Display Base 12
      3.1.3 Offset Registers 12
         3.1.3.1 P - Program Counter 12
         3.1.3.2 L - Local Variable Register 12
         3.1.3.3 I - Index Register 12
      3.1.4 Transaction Register 12
   3.2 PSM Store Addressing Modes 13
      3.2.1 Direct 13
      3.2.2 Indirect 13
      3.2.3 Indexing 14
      3.2.4 Global 15
      3.2.5 Global Indirect 15
      3.2.6 Local 16
      3.2.7 Local Indirect 16
      3.2.8 Display Indirect 17
      3.2.9 Immediate 17
      3.2.10 Constant 17

4. Types 18
4.1 Method of Definition 18
4.2 Basic Types 18
4.2.1 DSG definitions of basic types 19
4.2.1.1 Characters 19
4.2.1.2 Integers 19
4.2.1.3 Reals 19
4.2.1.4 Tagged Words 20
4.2.2 Pointers 22
4.3 Command Types 22
4.3.1 Tuple 22
4.3.2 String 23
4.3.2.1 Short Strings 23
4.3.2.2 Long Strings 23
4.3.3 Code Vector 24
4.3.4 Relation 24
4.3.4.1 new 24
4.3.4.2 Delete and Lookup 24
4.3.5 Procedure Closure 25
4.3.6 Range 25
4.3.7 Vector 26
4.3.8 Source or sink 26
4.3.9 Process 27
4.3.10 Box Diagram Conventions 27

5. Maps and Routines 28
5.1 Procedure Call Mechanism 28
5.1.1 An Example 28
5.1.2 The Problem 29
5.1.3 PSM Ordinary Variable Allocation 30
5.1.4 Ordinary Procedure Space Allocation 30
5.1.4.1 Ordinary Procedure Declarations 31
5.1.4.2 Ordinary Procedure Calls 32
5.1.5 Special Procedures 32
5.1.6 Special Variables 33
5.1.7 Procedure Results 34
5.2 Arrays 35
5.3 Relations 35
5.3.1 Narity >= Arguments 35
5.3.2 Narity < Arguments <= 2 (Narity) 36

6. Source or Sink 37
6.1 Format 37
6.1.1 Lazy List 38
6.1.2 Cursor 38
6.1.3 Fifo 40
6.2 Migration of sources and sinks 41

7. Relations 42
7.1 Choice 43
7.2 new 44
7.3 insert 44

8. Instruction Set 46
8.1 Notation 46
8.2 Load and Store Instructions 46
8.3 Arithmetic and Logical Instructions 47
8.4 Control Flow 47
8.5 Mapping Operations 48
8.5.1 System Maps 49
8.6 String Operations 49
8.7 Heap Operations 49
8.8 Relation Operations 49
8.9 Iteration 50

9. Syntax of Assembler 51

10. Appendices 53

The Persistent Store Machine

This report arose out of work done in Edinburgh University during 1983-84 to investigate design of an optimal machine architecture for persistent programming.

It was hoped that it might be possible to implement the machine in micro code on either an ICL 39 series processor or on the Orion microcoded engine. Due to the difficulty in obtaining adequate microcode support for the first alternative, and the politics of research funding in the second case neither of these alternatives have proved possible. Instead some of the key ideas of this paper have been implemented in a simplified form in another machine the "Poppy". This has been designed and built by myself and Peter Balch with hardware funding provided by Acorn computers. This will be the subject of a future report.

The key ideas which have been carried over from the PSM investigation to the Poppy are :

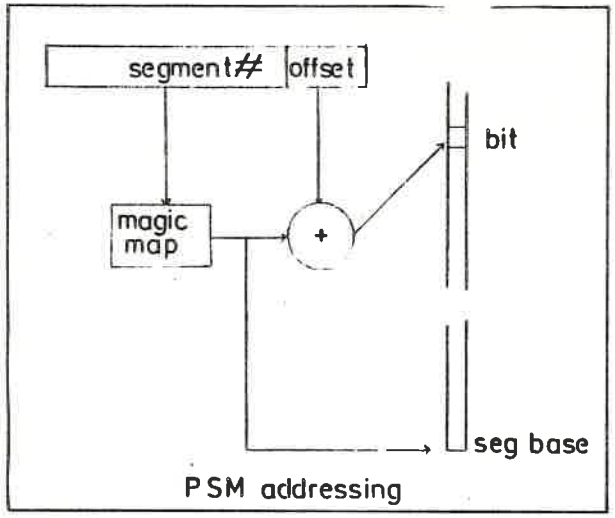
1. The use of long pids as the names of objects rather than short pids as their addresses.
2. The use of a stack based compiler technology rather than the heap frame based compiler technology used in the PS-algol abstract machine.

As is often the case, the first investigation that you make at a computing problem has a sweep that it is later difficult to maintain in later implementations. At first you are concerned with providing a theoretically elegant solution without bothering a great deal about the problems of actually implementing it. This is what happened with the PSM. The machine described here is certainly implementable, but not with the limited time and resources available. So the first prototype has had to make sacrifices. In one sense the "Poppy" is a better machine, it does after all exist, but in another sense it is very much the poor relation to the machine described here.

1. MEMORY STRUCTURE

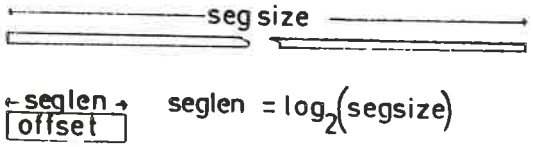
The memory of the PSM is made up of a large sequence of bits.

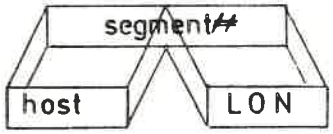
Each bit has a unique address. Its address is made up to two parts: an object (or segment) number and an offset within a segment.



A segment may contain SEGSIZE bits. So the offset part of a bit's address is SEGLEN = log2 SEGSIZE, assuming that the maximum size of a segment is a power of two.

A segment number is itself made up of two parts: a Local Object Number and a network Address. The Network Address uniquely specifies the machine upon which the segment resides. The Local Object Number (LON)





uniquely specifies the object within the machine. Let us define the number of bits in a Network Address to be NETLEN and the number of bits that go to make up a Local Object Number be LONLEN.

Each of these components of a bit's address should be big enough that the machine is not likely to be affected by address space limits at any level.

Suggested values of these parameters are

Field	Length in bits
SEGLEN	32
LONLEN	48
NETLEN	48

1.1. PIDS

Segment numbers are also referred to as Persistent IDentifiers, or PIDs. It is intended that PIDs will uniquely and persistently identify objects. No two objects in the world wide network of linked PSMs can have the same PID. The 48 bit NETLEN allows the use of Ethernet host numbers which are guaranteed unique for each machine.

The 48 bit SEGLEN is enough to allow segments to be allocated by a simple non-repeating counter within each machine. The persistence is ensured by having a one level address space, with objects being mapped between different storage medias on demand. No distinction is made between objects on disk or ram. It is assumed that Ram will be non-volatile, or if not, that the system will make periodic copies of such objects as necessary to non-volatile store.

1.2 Word Alignment

Bits may be addressed freely, but the PSM supports a number of different word types of different lengths: characters, reals, integers, tagged words.

These words can only be addressed with the addresses that are integral multiples of the number of bits in the word.

Multiple bit values have as their address the address of their least significant bit. The most significant bit of any object must have the highest address.

1.3. Memory Management

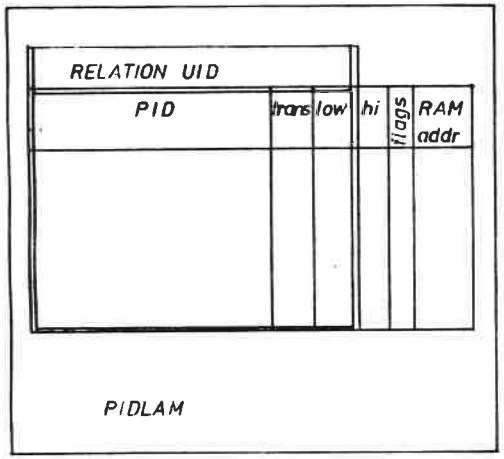
The mechanism used to support the Virtual Memory system is implementation dependent.

It will probably be necessary to support some sort of paging within objects, how this will be done will vary from one implementation to another. A suggested implementation is given here.

1.3.1. PIDLAM

The central feature of the memory management is the Persistent IDentifier to Local Address Map or PIDLAM. This can be thought of as a Relation with the following format:-

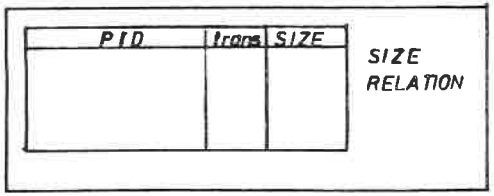
```
PIDLAM  :  RELATION
           object  :  PID ;
           trans   :  0..NO_OF_ACTIVE_TRANSACTIONS-1;
           low     :  INTEGER ; (* page number *)
           hi      :  INTEGER ; (* upper bound *)
           flags   :  SET OF ( WRITE, READ, WPERMIT, LOCK );
           address :  0..MAXADDR;
           END;
```



1.3.2. SIZE RELATION

An additional relation of type;

```
SIZES  :RELATION
          object  :  PID  ;
          trans   :  0..NO_OF_ACTIVE_TANSACTIONS -1 ;
          size    :  INTEGER ;
          END;
```



holds the sizes of objects. This relation could be merged with the PIDLAM to give a single relation in first normal form.

1.3.3. MAPPING

The PIDLAM is used to map accesses to objects onto RAM addresses. A selection is performed to find an entry whose PID and trans match the request:


```
PROCEDURE checkpidlam ( sef: PID;
        transno  : 0..NO_OF_ACTIVE_TRANSACTIONS -1 :
        offer  :INTEGER):TUPLE;
BEGIN
RETURN
   PIDLAM SUCH THAT
      object = seg AND
      trans  = transno AND
      low    = offset DIV PAGESIZE ;
      hi     > offset REM PAGESIZE ;
   END
END

PROCEDURE whereis( seg: PID;
        transno  : 0..NO_OF_ACTIVE_TRANSACTIONS -1 ;
        offset :INTEGER):INTEGER;
VAR T :TUPLE; trans:INTEGER;
BEGIN
        T:=checkpidlam(seg,transno,offset);
        trans:=transno;
        IF T £ nil THEN
                RETURN T PROJECT addr END
        ELSE
           WHILE T=nil AND trans> = 0 DO
               trans:=parentof(trans);
               T:=checkpidlam(seg,trans,offset);
           END;
           IF T=nil THEN
               import(seg,transno,offfset);
               RETURN whereis(seg,transno,offset);
           END;
           T.trans:=transno;
           insert(T,PIDLAM);
           RETURN T PROJECT addr END;
        END
END
```

2. TRANSACTIONS

A transaction is an environment within which a program can execute.

A transaction has associated with it the following attributes:

a) Its parent transaction. Transactions can be arbitrarily nested.

b) The concurrency control algorithm that it uses which may be:

   i. None at all.
   ii. Optimistic concurrency control.
   iii. Locking objects.

c) its access rights.

A transaction is delineated by the operations START TRANSACTION and COMMIT or by the operations START TRANSACTION and ABORT.

2.1. RECOVERY

Consider the sequence of operations:

```
.
x:=5
.
START TRANSACTION
        .
        .
        print x
        .
        x:= 9
        .
        print x
        .
ABORT
.
print x
```

This will result in the output of

```
5
9
5
```


In words, the effect of an ABORT is to return the state of the store to the state it had when the transaction started. On the other hand:

```
x:= 5
.
START TRANSACTION
        .
        .
        print x
        .
        x:= 9
        .
        print x
        .
COMMIT
.
print x
```

will produce the result

```
5
9
9
```

2.2. NESTING

The effect of nesting is a logical extension of the effect of COMMITing or ABORTing a transaction in the global environment.

```
x:= 1
START TRANSACTION
        print x
        x:=2
        START TRANSACTION
                x:=3
                print x
        ABORT
COMMIT
print x
```

will produce results

```
1
2
3
```

whereas

```
x:= 1
START TRANSACTION
        print x
        x:=2
        START TRANSACTION
                x:=3
                print x
        COMMIT
COMMIT
print x
```

will produce as results:

```
1
3
3
```


3. REGISTERS AND ADDRESSING

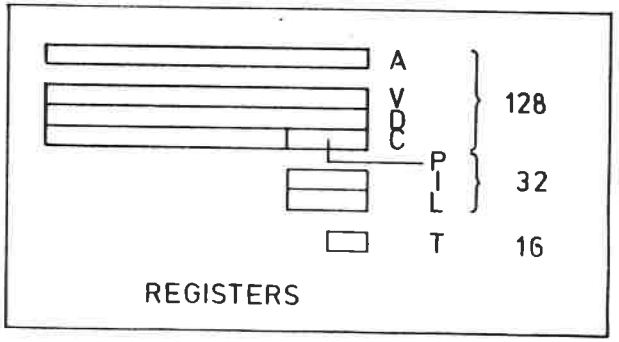
The domain of a process executing on the PSM is a set of segments reachable from its segment registers. All processes have at least two segments : the Code Segment from which all instructions are fetched and the Variables Segment which contains the program variables upon which the instructions operate.

In addition, programs in block structured functional languages may use a Display segment to hold the static context of the currently executing procedure.

3.1. PROGRAMMABLE REGISTERS

The registers can be divided into 4 groups:

1. An accumulator
2. Segment registers
3. Offset registers
4. A transaction register



3.1.1. A - Accumulator

The PSM has an 128 bit accumulator that acts as the destination of arithmetic and load instructions.

3.1.2. Segment Registers

The PSM has three 128 bit segment registers. These can be loaded with the virtual address of segments.

3.1.2.1. C - Code Base

The C register points at the virtual address of the code segment. Since this must always be of type CODEVEC, the type field of the virtual address is free to be used as an offset into the segment. The least significant part of the C register, holding an offset into the code segment is the P register.

3.1.2.2. V - Variable Base

The V register points at the virtual address of the variables segment.

3.1.2.3. D - Display Base

The D register points at the virtual address of the display segment.

3.1.3. Offset Registers

The offset registers are used to provide offsets into the segments. They are all 32 bits long.

3.1.3.1. P - Program Counter

The Program Counter contains the offset into the code segment of the current instruction. It is the least significant part of the C register.

3.3.2. L - Local Variable Register

This register holds upper limit of the space available to the current procedure context. It is always treated as an offset from the V register.

3.1.3.3. I - Index Register

This is used as a general purpose index register. It may be used to index into any of the 4 segments making up the current domain of a process.

3.1.4. TRANSACTION REGISTER

This is a 16 bit register that holds the currently active transaction. Transactions are dealt with in another section. It also holds other status bits.

3.2. PSM STORE ADDRESSING MODES

The addressing modes have been chosen to allow efficient implementation of block structure languages, including languages which support higher order functions.

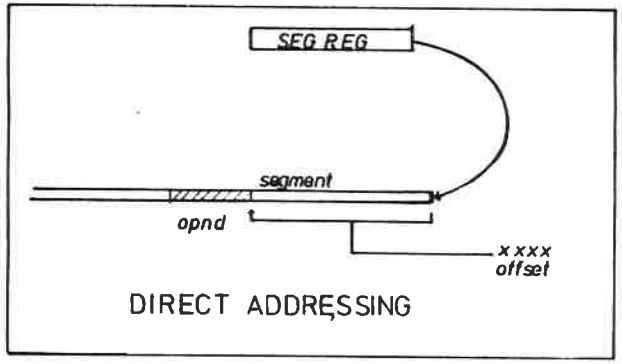
Addressing mode summary

Indicates legal combinations

Basemode	Direct		Indirect	
		Indexed		Indexed
Display	no	no	yes	yes
Global	yes	yes	yes	yes
Local	yes	yes	yes	yes
Literal	yes	yes	yes	yes
Immediate	n/a	n/a	n/a	n/a

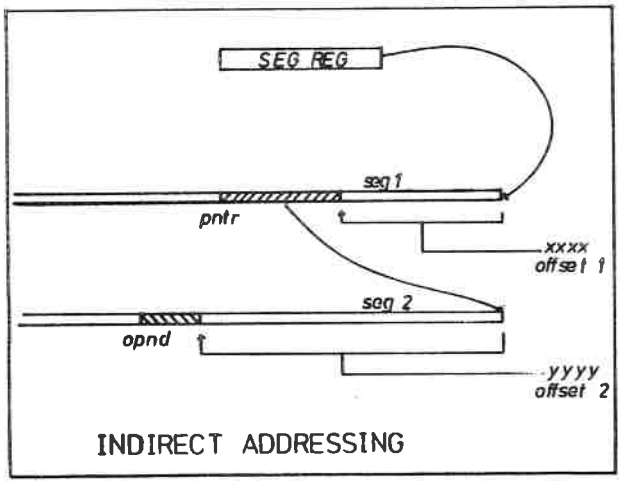
3.2.1. DIRECT

With direct addressing the operand address is always located within one of the segments that make up the domain of a process.



3.2.2. INDIRECT

Indirect addressing proceeds by two stages. First, direct addressing is used to locate a descriptor within the process domain, then the descriptor is dereferenced and an offset added to the base of the resulting segment to obtain the operand address.

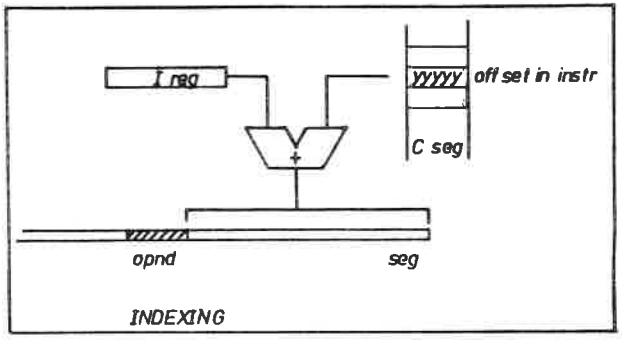


3.2.3. INDEXING

Notation:

Opcode Mode (I)

Both direct and indirect addresses may be indexed. The direct or indirect addressing is applied first and then the I register is added to the resulting within segment address, to obtain the final address of the object.

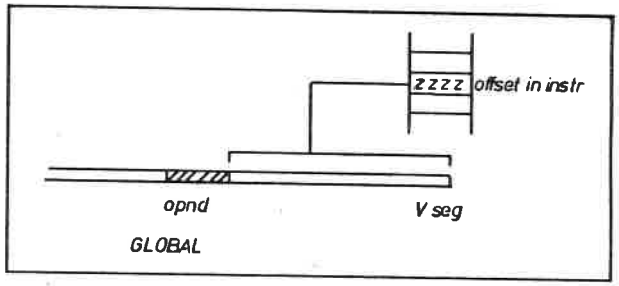


3.2.4. GLOBAL

Notation:

Opcode G(param)

This address mode is designed to allow access to global and own variables in Algol like languages like Fortran. Global address mode instructions take a single parameter which is interpreted as an offset from the start of the variables segment.



3.2.5. GLOBAL INDIRECT

Notation:

Opcode G(param, param)

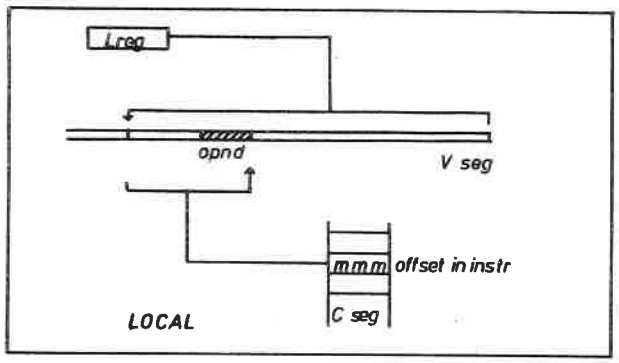
This is designed to allow access to fields of global records. Indirect instructions take two parameters. The first parameter is interpreted as for a global instruction, the descriptor at the resulting address is dereferenced, the second parameter is then used as an offset into this segment.

3.2.6. LOCAL

Notation:

Opcode L(param)

This takes a single parameter treated as a signed integer. It is used to address local variables in block structured languages. The parameter is subtracted from the L register to find the offset into the variables segment.



3.2.7. LOCAL INDIRECT

Notation:

Opcode L(param, param)

This is analogous to global indirect except that the first parameter is subtracted from the L register to find the offset into the variables segment.

3.2.8. DISPLAY INDIRECT

Notation:

Opcode D(param, param)

This is a method of implementing intermediate level variable access in block structure languages. This method can be used by procedures which may be passed out of their scope of declaration. This mode takes two parameters. The D register points at a display, either on the stack (V segment) or in a separate segment. The first parameter is used to index this display to obtain a pointer that is indexed via the second parameter to obtain the location of the desired variable.

3.2.9. IMMEDIATE

Notation:

Opcode integer-literal

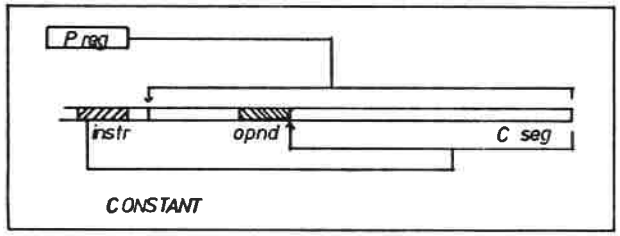
In this mode the parameter to the instruction is the operand to be used. This can only be used with short integer literals.

3.2.10. CONSTANT

Notation:

Opcode C(param)	direct
Opcode C(param, param)	indirect

Longer literals are obtained by using the Constant mode in which the parameter is used as an offset into the code segment to find the operand.



4. TYPES

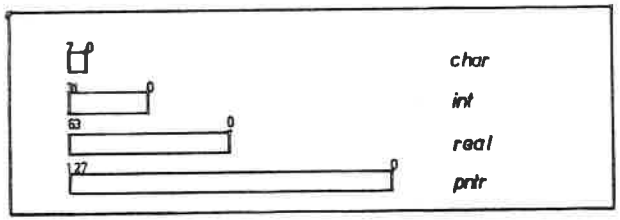
4.1 Method of Definition

Two methods of definition will be used to specify the types used by the PSM. The structural properties of types will be specified in the Data Structure Grammar. A less specific structural specification plus a semantic specification will be provided in Modula 2.

4.2. Basic Types

There exist several types, termed basic, that are assumed to exist in all machines based upon this architecture. These are:

1. Characters
2. Integers
3. Reals
4. Tagged words.



4.2.1. DSG definitions of basic types

The following primitives are used as a base for all formats:

```
bit :
     bit0;
     bit1.
bit ( n ) :
     if ( n / 2 ) * 2 = n then bit0 else bit1 fi.
word (length , value ):
     if length = 1 then bit ( value ) else
          word( length -1 , value / 2 ) ,
          bit ( value )
     fi.
```

4.2.1.1. CHARACTERS

```
CHAR:
      bit^CHARLEN.
```

4.2.1.2. INTEGERS

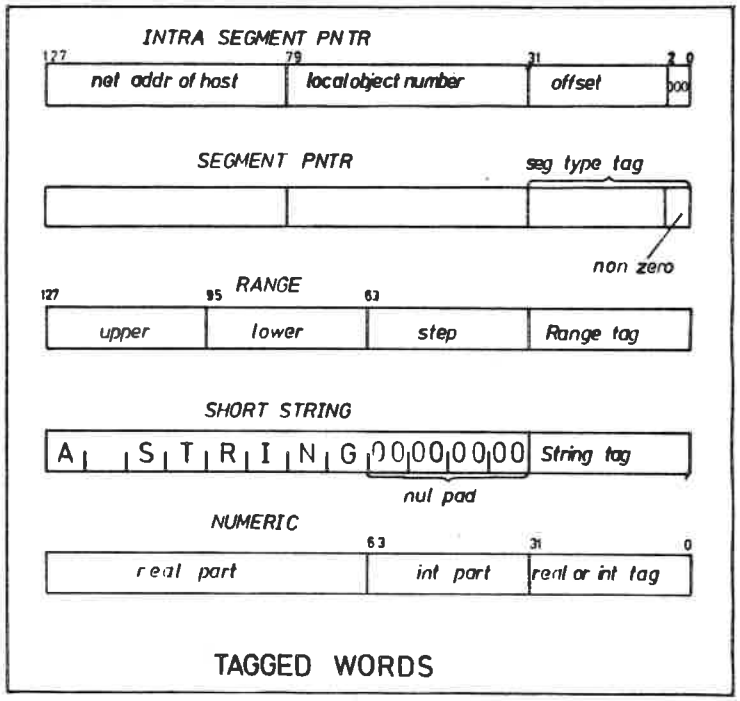
```
INT(n):
     word (SEGLEN,n);
     bit ^ SEGLEN.
```

4.2.1.3. REALS

```
REAL:
     bit^REALLEN
```

4.2.1.4. Tagged Words

Tagged words are a fundamental feature of the architecture. In all implementations these are assumed to be TAGLEN bits long. The internal format is :




```
Tagged-word ( tag, global, local ):

        word (NETLEN, global), word (LONLEN, local ), INT(tag);
        INT(w3), INT(w2), INT(w1), INT(tag);
        bit^TAGLEN.
```

or in Modula 2

```
TaggedWord = RECORD
              CASE  tag : INTEGER OF
                Shortstring :
                    sstr  :ARRAY [0..SSL-1]  OF CHAR ;
                Range : range :RANGE ;
                Integer,
                Real  : Number:RECORD
                            intpart:INTEGER;
                            realpart:REAL;
                        END
              ELSE
                segment : pntr: PID ;
              END
PID         = RECORD
                local, global : CARDINAL;
              END
```

Field	Meaning
tag	This encodes the type of the object either referred to or contained within the tagged work. The type definition mechanism is dealt with in detail elsewhere.
global	This holds the network key of the server at which the object is domiciled if the tagged word is a pointer, and otherwise holds the top NETLEN bits of the tagged word's value
local	This holds the local key of the object within its server if the tagged word is a pointer, otherwise it holds the lower LONLEN bits of the objects value.

Since tagged words have a tag field we may speak of a tagged word of type T, i.e., a tagged word with a tag field of T, where T is a symbol representing a SEGLEN bit type code. These will be the same codes as used by the type manager of the local machine when compiling type definitions.

The significance of a tagged word depends upon its tag field. For a large class of tags, the tagged word is a pointer to an object.

4.2.2. Pointers

Pointer :
    Taggedword.

Tagged words can function as pointers in two ways. In the first case, they can be pointers to segments, in which case the tag encodes the type of the segment.

In the second case the tagged word can be interpreted as a pointer into a segment. In this case, the tag is interpreted as a offset into the segment. Since the smallest basic data item supported is a character, and since all addresses of basic data items must be aligned with the wordlength of the item, we will only ever want pointers that are at least character aligned. If the least significant log2(CHARLEN) bits are non zero, then the tagged word cannot be a pointer into a segment but must be a pointer to a segment. In practice this is likely to mean that one eighth of the types will be reserved as offsets.

4.3. Command Types

Compound types are those which are made up by composing elements of the base types. There are two classes of compound types :

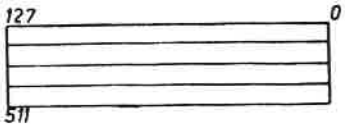
1. Those recognised in the machine specification
2. Those defined by applications programs.

The following compound types are recognised by the machine:

4.3.1. Tuple

implemented as a pointer to a sequence of tagged words on the heap

```
TUPLE ( n ) :
        Tagged-word^n.

TUPLE = RECORD
        seq : ARRAY [0 .. n-1]  OF TaggedWord;
        END
```

A 4 TUPLE

4.3.2. String

Sequences of characters, these occur in two forms:

STRING:
    Short-string;
    Long-string.

Note that

1. ISO "nul" is taken to represent the end of a string.
2. The most significant character in a string has the highest address.
3. Strings are padded out with ISO nul

4.3.2.1. Short Strings

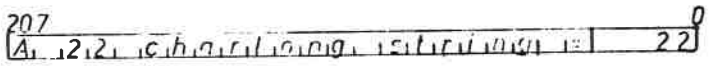
These are packed into a taggedword with tag Shortstring.

Shortstring (n):
        char^n ,
        word (SEGLEN, Shortstring).

4.3.2.2. Long Strings

These are implemented as a pointer to an object on the head with the following format:

Longstring (n) :
    CHAR ^ n, INT (n).



4.3.3. Code Vector

a pointer to a sequence of instructions

4.3.4. RELATION

REALTIONs are an abstract data type used to implement associative store, the implementation is dealt with elsewhere. They are implemented as a tagged pointer of type RELATION to a RELATION data structure supporting the following operations.

```
DEFINITION MODULE RELATIONs;
        EXPORT RELATION, CHOICE, new, insert, delete, lookup;
        TYPE RELATION, CHOICE;
        PROCEDURE new(narity :1..16;
                      choose :CHOICE;
                      ):^RELATION;
        PROCEDURE  insert( tup :^TUPLE; tab :^RELATION;);
        PROCEDURE delete( lo, hi :^TUPLE; tab :^RELATION; );
        PROCEDURE lookup( lo, hi :^TUPLE; tab :^RELATION;):SOURCE
END
```

4.3.4.1. new

This creates a RELATION whose tuples have narity, using the choice vector choose.

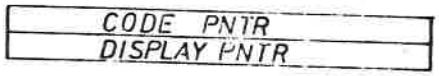
4.3.4.2. Delete and Lookup

These take ranges specified by a pair of tuples and either delete all tuples from the RELATION within that range, or return a SOURCE of all tuples within that range.

4.3.5. PROCEDURE CLOSURE

The system recognises both procedure closures, and references to procedure closures. A procedure closure is a pair of tagged words:

```
CLOSURE:
     displaypointer, codevectorpointer.
displaypointer:
     Taggedword.
codevectorpointer:
     Taggedword.
CLOSURE= RECORD
        codevec, display : Taggedword;
      END
```



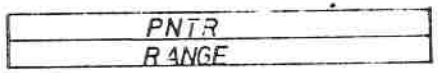
A PROC

4.3.6. Range

A range is a tagged word containing 3 integers being the upper and lower bounds and the step of the range.

```
range (step, lower,upper):
     INT(upper), INT(lower), INT(step), INT(Range).
RANGE = RECORD
         step, lower,upper : INTEGER ;
       END
```

4.3.7. Vector



A VECTOR

Vectors are sequences of items with an associated range. The step of the range specifies the number of bits in each element of the vector.

The advantage of this format is that a vector becomes a particular way of viewing a segment. This allows slicing and remapping of bounds to be trivially accomplished.

Vector :

Range, Pointer .

```
TYPE VECTOR=RECORD
        Pointer:TaggedWord;
        Range  :TaggedWord;
END
```

4.3.8. Source or sink

A source is an abstraction that can be used to view the generation of a sequence of values. The specific forms of sources and sinks are dealt with in the approripate chapter.

```
DEFINITION MODULE SourceAndSink;
        EXPORT SOURCE,SINK, append, next;
        TYPE SOURCE, SINK;
        PROCEDURE append(s :SINK; v:TaggedWord;);
        PROCEDURE next(s :SOURCE;):TaggedWord;
END
```


4.3.9. Process

A process is represented by a tuple that defines its current state.

representation to be decided later.

4.3.10. Box Diagram Conventions

In the box diagrams, assume that

1. If a box occurs below another one, it has a higher address.

2. With a horizontal box, the byte addressing convention depends upon the underlying machine for long items. The preferred model is the Vax model in which byte addresses of long word boxes are represented with the low byte to the right and the high byte to the left.

5. MAPS AND ROUTINES

The PSM provides a single high level abstraction to handle several operations that most systems treat as distinct. Arrays, Relations and functions are all dealt with as special cases of the abstraction MAP.

The application of a function to a list of arguments, the application of an array to a list of arguments and the application of a relation to a partially specified tuple are all handled by the same mechanism. This is intended to allow flexibility in the implementation of maps, so that if efficiency demands it, a procedure can be replaced by an array or relation without having to alter the code that called the procedure.

5.1. PROCEDURE CALL MECHANISM

An objective of the PSM is to provide the means to implement languages in which functions may be treated as persistent data objects. An instance of such a language is PS-algol. In this, procedures may export other procedures, either as a result, or by assigning locally declared procedures to global variables.

5.1.1. AN EXAMPLE

Consider the following rather pointless fraction of PS-algol:

```
Annotation          Program text
Point

                    let v:=vector 0::9  of 3.0
                    let x:=READR : let y =x/5
1 ---               let p:= proc ; nullproc
                    while ~ eof  do
                    begin
                       let a:=readi
                       let b= proc(int  i --> real)
                              if i=0 then v(i) else v(i)+b(i-1)
```



```
2 ---                    v(a):=b(a)
                       if a > 5 do p:=proc(); {x:=a*x+y}
                    end
3 ---                  p()
                    write x,y,v(6)
```

This fragment contains two procedures both declared in the inner block. Of these, 'b' is an ordinary recursive procedure that persists only for the duration of the block in which it is declared. On the other hand,

to it some times round the loop. When 'p' is called at point '3', this must invoke the last local procedure assigned to 'p'. This procedure references 3 identifiers 'a,x,y' of which 'x,y' are globals and the local 'a'.

Therefore both the procedure and the local identifier 'a' must persist after their declaring scope has been exited.

5.1.2. THE PROBLEM

Use of conventional algol storage allocation schemes with space for variables allocated on a stack will not work in this case. With such a system 'a' would occupy a fixed position on the stack. Once the declaring block had been exited, this location's clue would become undefined as the location might be reused for some other purpose. An alternative method, often used in functional languages is to allocate blocks (or procedure contexts) on the heap. This is done with the current PS-algol system. It ensures that identifiers that are still 'live' do not get discarded, so that the above example would run correctly. However, allocation of space on a head is less efiicient than on a stack.

We would expect that even in languages which allow functions to persist beyond their context of declaration, most functions will continue to be used only within their declaring context. It is thus only in exceptional cases that the normal algol method of allocating variables on a stack is likely to prove inadequate. The problem is to devise a procedure call mechanism that deals with most variables in the simple stack fashion, but deals with those that must persist by allocating them on the heap.

5.1.3. PSM ORDINARY VARIABLE ALLOCATION

The primary means by which variables are allocated space on the PSM is the variable segment, addressed by the V register. For block structured languages, this is to be treated as a stack.

For a conventional block structured language like Algol-60 three addressing modes are used:

1. Global :variables and own variables are allocated space at the base of the V segment. These are then accessed via Global address mode which adds an offset to the base of the segment to find the location of the variable. The allocation of space for these variables can be done staticly.

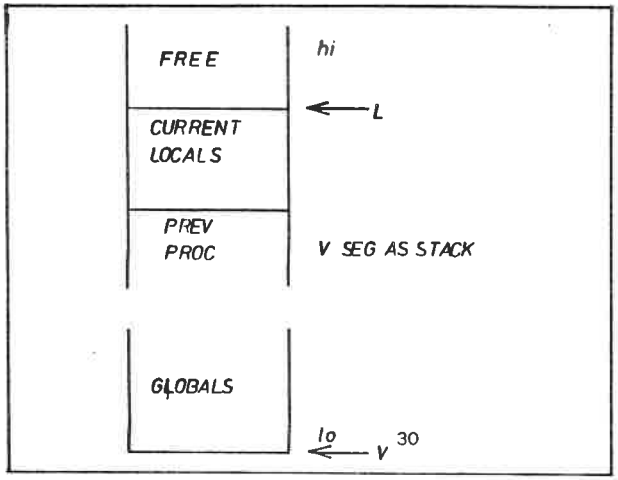
2. Local :variables are accessed via Local address mode, in which a staticly known offset is subtracted from the L register to obtain a displacement from the base of the V segment.

3. Intermediate : variables which are accessed via a display pointed at by the D register.

5.1.4. ORDINARY PROCEDURE SPACE ALLOCATION

It is possible for a two pass compiler to know at compile time how much space is required for local variables and temporary locations used for expression evaluation.

The organisation of space for local variables can be shown as follows:



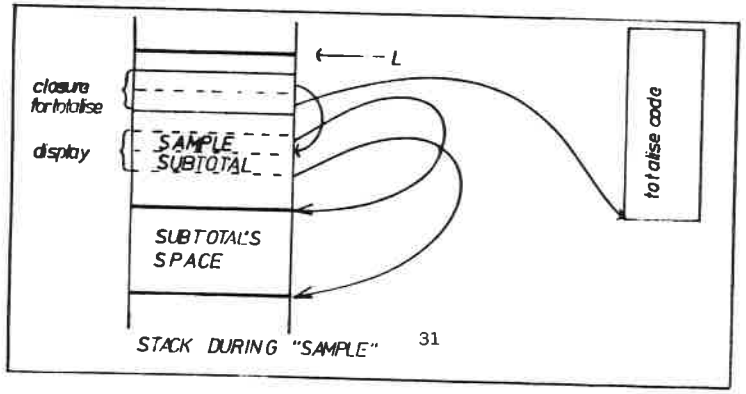
It can be seen that the L register acts as a combination of stack pointer and Frame pointer. It marks the upper bound of the space available to the current procedure, and it is used as the base from which the addresses of local variables are calculated.

5.1.4.1. ORDINARY PROCEDURE DECLARATIONS

Consider the following Salgol fragment.

```
procedure subtotal(int no —) int)
begin
let sequence:=vector 1::no of 0
for i=1 to no do sequence(i):= readi

procedure sample(*int range —) int)
begin
  let low=range(1)
  let hi=range(2)

  procedure totalise( int sofar—) int)
      if sofar  hi then  0 else
      sequence(sofar)+totalise(sofar+1)

  totalise(low)
end

sample(@1 of int   [ readi,  readi])
```

The procedure'totalise' must have access to the variable 'sequence' which is declared in 'subtotal' and to the variable 'hi' declared in sample. At the point of declaration of 'totalise' the compiler must plant code to build a display for the procedure. The display will point at the start of the contexts for the procedures 'subtotal' and 'sample'.



After planting code to construct the display, the compiler generates code to construct a procedure closure for totalise.

5.1.4.2. ORDINARY PROCEDURE CALLS

The Apply instruction (used for procedure calls) therefore takes two parameters: The first operand is a map which is applied to the argument(s) at L(0),L(TAGLEN),L(2 TAGLEN), etc. The second operand specifies the number of arguments to be applied.

The C and D registers are stored immediately above the address pointed to by the L register, and control is transferred to the address pointed to by the code vector part of the closure.

The closure consists of a double word : a pointer to a code vector and a display pointer. The C register is set to point at the start of the code vector and the D register loaded with the display pointer.

On entry to the new procedure, the L register is incremented by an amount sufficient to allow for the space required by the procedure.

Using the display addressing modes the procedure can get access to variables in its surrounding lexical levels.

5.1.5. SPECIAL PROCEDURES

At compile time it is possible to divide procedures into two classes : the ordinary and the special. Special procedures are those that may survive their context of declaration and which refer to intermediate variables.

In this context intermediate means non-Local and non-global and non-Own.

Any procedure that may be

a. returned as results from the procedures within which they were declared

b. assigned to a variable external to the block in which they are declared

c. assigned to a field of a heap object

d. passed as a parameter to a procedure that may assign its procedure parameter to a variable external to the scope of declaration of the first procedure or to the field of a heap object

and which accesses intermediate variables must be considered special.

Special procedures cannot safely use the type of display addressing explained above, as they might be called after the context in which they were declared had ceased to exist. In that case, the closure, the display, and the intermediate variables that the procedure referenced might have had their space in the V segment reallocated.

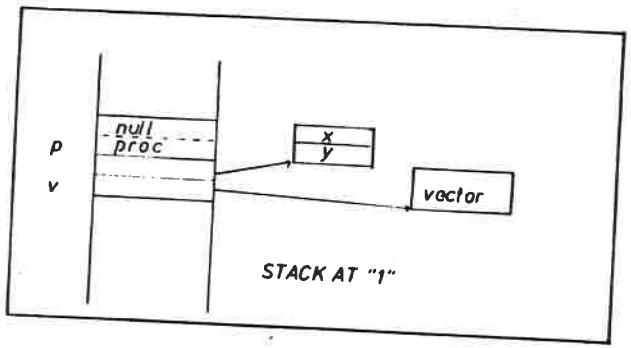
5.1.6. SPECIAL VARIABLES

A variable is special if it may be referred to at an intermediate level by a special procedure. Special variables may have to persist after the flow of control has exited the block in which they were declared. This implies that they must be allocated on the heap. Since it can be known at compile time which variables in each block are special, we can plant code at the start of special blocks to generate a vector on the heap to hold them. This vector would be pointed to by an unnamed location within the stack frame allocated to the block. Consider again this following rather pointless fraction of PS-algol:

```
Annotation          Program text
Point
                    let v:=vector 0::9  of 3.0
                    let x:=readr ; let y =x/5
                    let p:= proc ; nullproc
1 ---
                    while ~eof  do
                    begin
                       let a:=readi
                       let b= proc(int  i-> real)
                              if i=0 the v(i) else v(i)+b(i-1)
2 ---                    v(a):=b(a)
                       if a > 5 do p:=proc(); {x:=a*x+y}
                    end
3 ---
                       p()
                    write x,y,v(6)
```

Suppose we are at point '1' in the program given above, the stack would

look like:



Since x and y are accessed by a special procedure, they are put into a cell on the heap.

Ordinary code (not in a special procedure) will access local and global special variables using Local Indirect of Global Indirect addressing.

The closure of an ordinary procedure holds a pointer to the code vector and a pointer to a display in the variable segment area devoted to the procedure's context of declaration. The closure of a special procedure substitutes for this a pointer to a display on the heap. The display contains pointers to each vector of special variables that the procedure will access.

Special procedures are entered by exactly the same sequence of code as ordinary ones. Once entered, however, Display Indirect addressing will result in variables being accessed on the heap rather than in the variables segment.

5.1.7. PROCEDURE RESULTS

If a procedure is to return a procedure result, then it must copy the procedure closure out of the local context onto the heap. It then returns a pointer to the closure on the heap. This pointer is of type CLOSURE.

5.2. ARRAYS

If the operand of an APPLY instruction is found at run time to be an array, then the array is indexed by the first argument. If the argument count is greater than one, the result is applied to the remaining arguments.

5.3. RELATIONS

If the operand of an apply operation is found at run time to be a relation, then the arguments are treated as pattern to which the relation is applied according to the following rules.

5.3.1. Narity >= Arguments

If the narity of the relation is greater than or equal to the number of arguments, then the result is a stream of all tuples whose corresponding fields match the arguments. Given the relation:

arachnid	8	spider
bird	2	eagle
bird	2	sparrow
insect	6	ant
insect	6	fly
mammal	0	whale
mammal	2	human
mammal	4	horse
reptile	0	cobra

Then applying it to the argument list

bird 2

we would get the stream:

bird	2	eagle
bird	2	sparrow

-applying it to the argument list

OMEGA 2

we would get the stream

bird	2	eagle
bird	2	sparrow
mammal	2	human
mammal	2	human

applying it to the argument list

mammal

we would get the stream

mammal	0	whale
mammal	2	human
mammal	4	horse

applying it to the argument list

mammal 4 horse

we would get the stream

mammal	4	horse

5.3.2. Narity < Arguments <= 2 (Narity)

It the narity of the relation is less than the number of arguments to which it is being applied, then the arguments are taken to be a range query as follows.

Using the same table, the argument list

bird 2 OMEGA mammal 4

would yield the stream of all tuples in the range

bird 2 <= tuple <= mammal 4

i.e

bird	2	eagle
bird	2	sparrow
mammal	2	human
mammal	4	horse

6. Source or Sink

A source is an abstraction that can be used to view the generation of a sequence of values.

Sequences of values occur in a number of contexts in computation:

- They are generated as a result of iteration through data-structures like arrays
- They occur as input or output streams
- They occur due to the temporal sequencing of data transfers in interprocess communication
- They may occur as a result of applying some selection operation to a relation or set of data records

Sources provide an 'input' abstraction for sequences of values. Sinks provide an 'output' abstraction for sequences of values.

These two types provide a single interface to several different mechanisms or representations.

```
DEFINITION MODULE SourceAndSink;
        EXPORT SOURCE, SINK, append, next;
        TYPE SOURCE, SINK;
        PROCEDURE append(s :SINK; v:TaggedWord;);
        PROCEDURE next(VAR fin:BOOLEAN;
                        s :SOURCE;):TaggedWord;
END
```

Sources and sinks may be implemented in the following ways: Lazy lists, Cursors, Fifos and Input Output devices.

6.1. Format

The following Modula 2 definition can be used to specify the layout of a source or sink.

```
TYPE SOURCE = RECORD
        discriminant:(close,cur,fi,input);
        CASE discriminant of
```

```
        close    :      ure :LAZYLIST;
        cur      :      sor :CURSOR;
        fi       :      fo  :FIFO;
        input    :      port:INPORT;
        END;
    END
```

6.1.1. Lazy list

A lazily evaluated list is a procedure closure. The next item is obtained by invoking the procedure closure.

```
    DEFINITION MODULE lazylists;
        EXPORT LAZYLIST,next;
        TYPE LAZYLIST,;
        PROCEDURE next(VAR fin:BOOLEAN;
                       s :POINTER TO LAZYLIST;):TaggedWord;
    END
    IMPLEMENTATION MODULE lazylists;
        TYPE LAZYLIST = PROCEDURE ():TaggedWord;
        PROCEDURE next(VAR fin:BOOLEAN;
                       s :POINTER TO LAZYLIST;):TaggedWord;
        BEGIN
            RETURN  s();
        END;
    END;
```

6.1.2. Cursor

A cursor is a data structure that is used to iterate over the machine recognised data types: Strings, Tuples, Ranges, Vectors and Relations. It acts only as a source not as a sink.

```
    IMPLEMENTATION MODULE Cursors;
        EXPORT CURSOR,  next;
        TYPE CURSOR =RECORD
                Data,Posn:TaggedWord;
            END;
        PROCEDURE next(VAR fin:BOOLEAN;
                       s :CURSOR;):TaggedWord;
```



```
PROCEDURE next(VAR fin:BOOLEAN;
               s :Cursor;):TaggedWord;
Var index:INTEGER;
BEGIN
  WITH  s DO
    CASE Data.tag OF
    String,Tuple:BEGIN
         index:=Posn.Intpart;
         Posn.Intpart:=Posn.Intpart+1;
         fin:=Posn.Intpart⟩upper;
         RETURN
           Data^.Pointer^[index]
           (* assume scaled indexing *)
        END
    Vector:BEGIN
        WITH Data^.Range DO
         index=Posn.Intpart;
         Posn.Intpart:=Posn.Intpart+step
         fin:=Posn.Intpart⟩upper;
         RETURN
           Data^.Pointer^[index..Index+step]
           (* return appropriate slice *)
        END
    END;
    Range:BEGIN
        WITH Data.range DO
         index=Posn.Intpart;
         Posn.Intpart:=Posn.Intpart+step;
         IF step⟩0 THEN
             fin:=Posn.Intpart⟩upper
         ELSE fin:=Posn.Intpart⟨upper END;
         RETURN index
        END
    END;
    Relation:BEGIN
              (* to be dealt with later *)
             END;
  END;
END
END
```

6.1.3. Fifo

Fifos are the fullest implementation of sources and sinks. They can be used to communicate between processes. A fifo can act as either a source or a sink and has the form:

```
DEFINTION MODULE FIFOS;
        EXPORT FIFO, append, next;
        IMPORT TaggedWord;
        TYPE FIFO;
        PROCEDURE append(s  :FIFO;  v:TaggedWord);

      PROCEDURE next (var fin:Boolean;
                      s :FIFO;):TaggedWord
END


IMPLEMENTATION MODULE FIFOS;
      EXPORT FIFO, append, next;
      IMPORT TaggedWord,CurrentProcess,Schedule,
             Reschedule;
      TYPE FIFO=RECORD
             n,N            : INTEGER;
             in,out         : [0..N-1]
             rlist,wlist    : POINTER TO FIFO;
             buff           : BUFFER;
      END;
      TYPE BUFFER = ARRAY  [0..N ]  OF TaggedWord;
      VAR C:BOOLEAN;
      PROCEDURE append(s :FIFO;  v:TaggedWord);
      BEGIN
            WITH s DO
              n:=n+1;
              IF  n>N THEN
                    append(wlist,CurrentProcess);
                    Reschedule()
              END;
              buff [in] :=v;in:=(in+1)MOD N;
              IF n<=0 THEN
                    schedule(next (C, rlist));
                    schedule(CurrentProcess);
                    Reschedule()
              END
```



```
              END
      END
      PROCEDURE next(VAR fin:BOOLEAN;
                     s :FIFO):TaggedWord;
      BEGIN  fin:=FALSE;
             WITH s DO
               n:=n-1;
               IF n<N THEN
                     append(rlist,CurrentProcess);
                     Reschedule()
               END;
               buff [out] :=v;out:=(out+1)MOD N;
               IF n>=0 THEN
                     schedule(next(C,wlist));
                     schedule(CurrentProcess);
                     Reschedule()
               END
             END
      END
END
```

The values are all held in tagged words.

The r and w lists point to Fifos of processes waiting to read from or write to the Fifo

6.2. Migration of sources and sinks

Sources and sinks are not allowed to migrate off the machine on which they are created. This is because they are the basic communication system provided by the machine. Communiction between processes, whether on the same or different machines occurs by the transfer of information through FIFOs. In the event of an attempt being made to append to a remote FIFO, instead of the FIFO being fetched into local memory, as would happen with other types of object, a message is sent to the domicile of the FIFO containing the object to be appended and the actual appending takes place on the remote machine.

The implications of this must be dealt with in a later refinement of the specification.

7. RELATIONs

The RELATION type provides an associative set facility suitable for the implementation of relations etc.

The operations on RELATIONs are specified in the definition module given in the section on types. This section deals with the implementation. RELATIONs are implemented as a hash tree, with the hashing controlled by a "choose" field. The hash code is constructed by performing a bitwise shuffle on selected fields of the relation. This allows any arbitrary ammount of selectivity to be assigned to any field of the relation. Using this shuffled hash key, partial match queries become possible

```
IMPLEMENTATION MODULE RELATIONs;
        TYPE
                RELATION=RECORD
                        choose: CHOICE;
                        node : POINTER TO NODE;
                        elements: CORDINAL;
                        narity: 1..16;
                END;
                NODE = RECORD
                        index :ARRAY 0..MAXENT OF CHAR;
                        store :ARRAY 1..MAXENT+1 OF ENTRY:
                        free : CARDINAL ;
                END;
                ENTRY = RECORD
                        link:TaggedWord ;
                        tup :ARRAY 0..narity OF TaggedWord;
                END
        PROCEDURE new(n :1..16;c :CHOICE ;):POINTER TO RELATION
        PROCEDURE inset( tup : POINTER TO TUPLE;
                         tab : POINTER TO RELATION;);
        PROCEDURE delete( lo, hi : POINTER TO TUPLE;
                         tab : pointer to relation; );
        PROCEDURE lookup( lo, hi : POINTER TO TUPLE;
                         tab : POINTER TO TELATION;):SOURCE;
END
```


7.1. CHOICE

A choice vector is a tagged word with the following format:

```
CHOICE:
   selectros,
   hashbits,
   padding,
   word(SEGLEN,CHOICE).
   selectors:
      select  16

   hashbits:
      hashcoice  16.

   padding:
      bit   (LONLEN+PIDLEN-80).

   select( n ):
      word ( 4 , n ).

   hashchoice:
      tohash ;
      nottohash.

   tohash:
      bit1.

   notohash:
      bit0.
```

The choice vector is used to construct a shuffle over the fields of the tuple to produce a key to be used in indexing the relation. The following algorithm is used:

```
PROCEDURE SHUFFLE( C : CHOICE; T : POINTER TO TUPLE):BITSET;
VAR i,selected :0..15; h : ARRAY   0..15   OF BITSET;
   bc : ARRAY [0..15]  OF 0..127;
   r :BITSET; j :0..127;
BEGIN
 FOR i= 0 TO tuplesize(T) -1 DO
    h[i] := head(T[i]);
    IF C.hashbits[i]=tohash THEN h[i]:=hash(h[i])END;
    bc[i]:=127;
 END;
 FOR j= 0 TO 127 DO
    selected:=C.selector[j MOD 16];
   r[127-j]:=h[selected][ bc[selected]];
   bc[selected]:=bc[selected]-1;
 END;
 RETURN r;
END
```

7.2. new

```
PROCEDURE new(n :1..16;c :CHOICE ;): POINTER TO RELATION;
       VAR T : POINTER TO RELATION;
       BEGIN
              T:=CLM(RELATIONLEN);
              T^.choose:=c;T^.narity:=n;
              T^.elements:=0;
              T^.node:=newnode(n);
              RETURN (T);
       END;
```

7.3. insert

```
(* note that this algortihm must be extended to allow indefinitely
large overflow buckets at the bottom of the tree *)
```



```
PROCEDURE insert( tuple : POINTER TO TUPLE;
                  tab : POINTER TO RELATION;):
      VAR Key:BITSET;
       PROCEDURE ins(noddy: POINTER TO NODE;field: CARDINAL);
        VAR key,sentry:CHAR;
              newn: POINTER TO NODE;
       BEGIN
       key:=Key[field..field+step];
        WITH noddy BEGIN
        key:=
        IF store[index[key]].link is POINTER TO NODE THEN
             ins(store[index key]].link,field+step);
        ELSE
        IF NOT equal(store[index key]],tuple) THEN
             sentry:=store[free];
             free;=store[free].link;
             store[sentry].link:=key;
             store[sentry].tup:=tuple;
             index[key]:=sentry;
        END;
        IF free=NILVAL THEN
             FOR key= 0 to MAXENT DO
             IF index[key]£NILVAL THEN
             BEGIN
               sentry:=index[key];
               newn:=newnode(tab.narity);
               WHILE sentry£NILVAL DO
               BEGIN
                 ins(newn,store[sentry].tup,field+step);
                 sentry;=store[sentry].link;
               END
               index[key].link:=newn;
             END
       END
BEGIN
      Key:=shuffle(tuple,tab^.choose);
      ins(tab^.node,tuple,0);
END
```

8. INSTRUCTION SET

8.1. NOTATION

For definitional purposes the notation used for instructions is

Opcode Type Parm1 , Parm2
Where Type can be :

C	-	CHAR
I	-	INT
R	-	REAL
T	-	TAGGED WORD

8.2. LOAD AND STORE INSTRUCTIONS

	Mnemonic	Comment
L	⟨reg⟩	Load ⟨reg⟩ with operand
S	⟨reg⟩	Store ⟨reg⟩ into operand
LA		Load accumulator with address of operand
M	⟨reg1⟩ ⟨reg2⟩	More ⟨reg1⟩ to ⟨reg2⟩

Allowed move instructions

Source	Destination			
	A	Segment	Offset	Transaction
A	n/a	yes	yes	yes
Segment	yes	yes	no	no
Offset	yes	no	yes	no
Transaction	yes	no	no	no

The acumulator is treated as a typed tagged word. If it is loaded with a real, its tag field gets set to real etc. The machine thus always knows the type of the operand in the accumulator.

8.3. ARITHMETIC AND LOGICAL INSTRUCTIONS

These all use the accumulator as the destination. Dyadic operations take the accumulator as one of the arguments. Since the accumulator is treated as a tagged word, arithmetic is on the appropriate field of the tagged word. Mixed mode arithmetic obeys the following table:

ACCUMULATOR	C	I	R	T
OPND				
C	C or I	I or R	R	type de
I	I or R	I or R	R	type de
R	R	R	R	type de
T	type dependent	type dependent	type dependent	type de

Arithmetic operations on tagged words are legal if the tagged word is a tagged C, I or R. Logical operations are allowed on tagged words.

Mnemonic	Comment
ADD	Add operand to accumulator
SUB	Subtract operand from accumulator
MUL	Multiply accumulator by operand
DIV	Divide accumulator by operand
FLT	Convert integer to floating point
FIX	Convert real to integer
AND	And operand with accumulator
OR	Or operand with accumulator
SH	Shift accumulator operand specifies amount of shift, if negative then shift right else shift left.
RT	Rotate accumulator by operand interpreted as in SH
CP	Compares the accumulator with the operand. Zero flag set if they are equal, minus flag set if A operand

8.4. CONTROL FLOW

Mnemonic	Comment
J	Load P with the operand+P
JZ	Load P with the operand+P if the zero flag set
JM	Load P with the operand+P if the minus flag set

8.5. MAPPING OPERATIONS

The machine supports several sorts of maps:

Arrays : dense maps from the integers to values
RELATIONs : sets of tuples
Procedures : algorithmic maps

From the standpoint of the instructions all maps are treated the same way. Details of implementation are provided in the chapter on types.

APP The first operand is a map which is applied to the argument(s) at L(0),L(TAGLEN), L(2 TAGLEN), etc. The second operand specifies the number of arguments to be applied.
The space in the variable
segment above L must be free for the machine
to save state information, and for use as local
variables in procedural maps.
If the map is a procedure then the following
sequence of register transfer operations occurs
opnd2 → L(- TAGLEN)
C → L(-2 TAGLEN)
D → L(-3 TAGLEN)
opnd1 → C
if indexed then
I→P
else
0→P
fi
Results of maps are returned the A register.

RET Return from a procedural map,
l ← L - opnd
P ← L(- TAGLEN)
C ← L(-2 TAGLEN)
D ← L(-3 TAGLEN)

Apply immediate is used to call various system maps.

8.5.1. SYSTEM MAPS

The following system maps can be applied to arguments using apply immediate.

MAP EFFECT

SIZE The argument r is taken to be a segment Tagged word, the number of words in the segment is returned in A

TYPE The argument r is a segment Tagged word, the operation returns the type of segment.

MKSQ The argument is a Tagged word referring to some aggregate datatype: set, vector, tuple, list, range. The procedure returns a SOURCE allowing the aggregate to be stepped through.

8.6. STRING OPERATIONS

These operate on strings of characters or sequences of words. Result returned in the accumulator.

SUBS A holds a range. The operand specified a segment a new segment is formed containing a copy of the specified range of the original.

8.7. HEAP OPERATIONS

These operations create new heap objects.

CLM Claim an object
The accumulator specifies its size in words
the operand specifies the type of the heap
object to be created. The result is a Tagged word
of a new object in the accumulator.

8.8. RELATION OPERATIONS

These operations are specified in the types chapter sub-section on RELATIONs.

OPCODE	PARAM1	PARAM2	MEANING
NEWTAB	CHOICE		Accumulator contains narity, new RELATION returned in accumulator

INS	tup		RELATION to be inserted into in accumulator
DEL	lo	hi	RELATION in accumulator

8.9. ITERATION

These are operations on sources and sinks, further documented in chapter on types.

NEXT The accumulator holds a source Tagged word, if the operation succeeds the next element from the source is placed in the accumulator. If the operation fails, the operand is taken to be a relative jump address that is added to P.

APPEND The accumulator holds a sink Tagged word, the operand is appended to the sink.

9. SYNTAX OF ASSEMBLER

```
LINES : ;
     | LINES LINE ;

LINE : STAT
    | error

STAT : lab INS TERMINATOR;
    | INS TERMINATOR;

TERMINATOR : Terminator

INS : OP TYPE PARMS ;        |   Move MOVEPAIR       printf("
Move op ") ;  ;   | LDST REG PARM ;

MOVEPAIR : Accumulator REG ;          | REG   Accumulator   ;
      | SEGREG SEGREG ;       | OFFREG OFFREG  ;

OP   : Add ;       | Sub ;       |   Mul   ;         |   Div  ;
     |  Cp  ;        |  Float  ;       |  Fix ;        |  And ;
     | Or ;       | Shift ;      | Rotate ;          | Jump   ;
     |  Jumpz  ;        |  Jumpm  ;       | Apply ;          |
Return ;       | Subs ;       | Clm ;      | Next ;           |
Append ;

LDST : Load ;       | Store ; PARMS : PARM PARM2 ;

PARM2 : PARM ;
     | ;

PARM : OPERAND INDEXED ;
    | OPERAND ;

INDEXED : Bra OFFREG Ket ;

OPERAND : GLOBAL;       | LOCAL ;         | DISPLAY ;         |
IMMEDIATE ;       | CONSTANT ;

GLOBAL : Global ARG ;

LOCAL : Lreg ARG ;

DISPLAY : DisplayBase ARG ;

IMMEDIATE : EXPRESSION ;

EXPRESSION : IntLit ;

CONSTANT : CodeBase ARG ;

TYPE : INT ;
     | REAL ;
     | CHAR ;
     | TAG ; INT : Index ;

REAL : Real ;

CHAR : Char ;

TAG  : Tag ;
```

```
REG : Accumulator;
    | SEGREG ;
    | Treg ;
    | OFFREG ;

OFFREG :ProgramCounter ;          |Leg ;        |Index ;

SEGREG : CodeBase;      | VariableBase              |Display-
Base ;

ARG : Bra EXPRESSION INDIRECT Ket ;

INDIRECT : EXPRESSION ;       | ; tohelp ❮dsg
```


10. Appendix 1

10.1. Data Structure Grammar Notation

10.2. An Example : the IMP string

10.3. Example 2 : Defining PS-algol structures

10.3.1. BASIC VALUES

10.3.2. STRINGS

10.3.3. STRUCTURES

10.3.4. VECTORS

10.3.5. FRAMES

10.3.6. CODE VECTORS

10. Appendix 1

10.1. Data Structure Grammar Notation

The data structure grammar exists in 2 forms, an ascii text string form, and a compiled binary form.

Its function is to provide a means of defining the binary format of computer data-types.

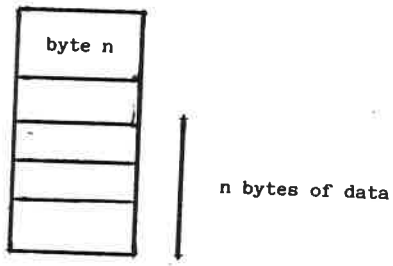
The grammar notation has the following features:

1. The LHS of a production is separated from the RHS by a ":"
2. Alternatives are spearated by ";"
3. Lists are terminated by "."
4. Concatenation is denoted by a ","
5. Productions may take integer parameters , formal integer parameters are in lower case.
6. Conditional productions are definable using "if" "then" "else" "fi".
7. Arithmetic on integers is built in.
8. Repetition is available
   a ^ n
   means a sequence of n a's
9. Indefinite repetition is denoted by the Kleene star.
10. There are only two terminal symbols : bit1 , bit0
11. Productions may take other productions as parameters, written in upper case
12. If a production has no body it is assumed that it uses the body of the immediately following production.

10.2. An Example : the IMP string

We wish to define the IMP string data format. This may be described informally as a sequence of bytes, the first of which describes how many others follow it.

The following diagram gives an idea of what is involved.



The following set of productions enable us to define this:

```
bit :
     bit0;
     bit1.
bit ( n ) :
     if ( n / 2 ) * 2 = n then bit0 else bit1 fi.
word (length , value ):
     if length = 1 then bit (value ) else
          word( length -1 , value / 2 ),
          bit ( value )
     fi.
byte:
     bit^8.
byte ( n ):
     word ( 8 , n ).
IMPSTRING( n ) :
     byte^n , byte ( n ).
```

Note that his defines strings with a Vax convention for the ordering of bytes, i.e., the numbering of bits and the numbering of bytes follows the same convention.

10.3. Example 2 : Defining PS-algol structures

10.3.1. BASIC VALUES

```
IB:
     INTEGER;
     BOOLEAN.
INTEGER:
     bit ^ 32 .
REAL:
     bit ^ 64 .
Pointer-TYPE:
     heap-address;
     PID.
heap-address:
     bit0 , bit ^ 32 .
PID:
     bit1 , instance , dbno , pseuod-class .
instance:
     bit ^ 7 .
dbno:
     byte.
pseudo-class:
     bit ^ 16 .
PROCEDURE:
     static-link-pointer , code-vector-pointer.
static-link-pointer:
     Pointer-TYPE .
code-vector-pointer:
     Pointer-TYPE.
```

10.3.2. STRINGS

```
STRING ( n ) :
     string-body ( n ) , string-header ( n ) .
string-header ( n ) :
     string-tag , bit ^ 7 , flag-bits , bit , bit , word ( 16, n ).
string-body ( n ) :
     pad  ( n ), byte ^ n .
pad ( n ) :
     byte ^ ( 4 - ( n - 4( n / 4 ))).
string-tag :
     tag ( 1 ).
tag ( n ):
     word ( 4 , n ).
fla-bits:
     garbage-mark, written-bit, commit-mark.
garbage-mark :
     bit.
written-bit :
     bit.
commit-mark :
     bit.
```

10.3.3. STRUCTURES

```
STRUCTURE ( n , m ) :
     structure-body( n, m ), class , structure-header ( n, m ).
structure-header (n, m ):
     struct-tag , pntrs( n ), flag-bits , bit, bit, word( 16, n+m+2 ).
struct-tag :
     tag ( 3 ).
pntrs ( n ) :
     word ( 7 , n ).
structure-body ( n, m ) :
     pointer-TYPE ^ ( n - 1 ) , IB ^ m.
class :
     pointer-TYPE.
```


10.3.4. VECTORS

```
VECTOR ( lwb, upb ):
     IB-vector ( lwb, upb);
     Pointer-TYPE-vector ( lwb , upb );
     Procedure-vector (lwb, upb);
     REAL-vector(lwb, upb ).
vector-header(type, lwb , upb ) :
     word (32, upb),
     word(32, lwb),
     tag ( type ), bit^7, flag-bits, bit ^ 18.
vector-body ( TYPE , lwb, upb ) :
     TYPE ^ (upb - lwb + 1 ).
vector-format ( TYPE, type, lwb, upb ) :
     vector-body ( lwb, upb ) , vector-header ( type , lwb, upb ).
IB-vector ( lwb, upb ):
     vector-format ( IB, 6, lwb, upb ).
REAL-vector ( lwb, upb ) :
     vector-format ( REAL, 7, lwb, upb).
Procedure-vector ( lwb, upb ) :
     vector-format (PROCEDURE, 5 , lwb, upb ).
Pointer-TYPE-vector ( lwb, upb ) :
     vector-format ( Pointer-TYPE, 4, lwb, upb ).
```

10.3.5. FRAMES

```
FRAME ( mdepth, pdepth, display-depth, psp, msp ) :
     lexical-level
     word( 32, msp ),
     word (32, psp ),
     line-number,
     return-offset,
     dynamic-link,
     static-link,
     procedure-address,
     Pointer-TYPE ^ display-depth,
     Pointer-TYPE ^ ( psp - 4 - display-depth),
     IB ^ (mdepth + pdepth - psp - msp ),
     IB ^ ( msp ),
     pointer-stack-link,
     tag (8), bit ^ 7, flag-bits, word (18, (mdepth+pdepth+7)*4).
```

```
lexical-level:
return-offset:
pointer-stack-link:
line-number:
     IB.
dynamic-link:
static-link:
procedure-address:
     Pointer-TYPE.

lexical-level:
return-offset:
pointer-stack-link:
line-number:
     IB.
```

10.3.6. CODE VECTORS

```
CODE_VECTOR ( n ) :
     IB ^ n,
     psd,
     msd,
     vs,
     vp,
     tag( 9 ), bit ^ 7, flag-bits , word (18, 4*(n+5) ).
psd:
msd:
     IB.
vs:
vp:
     Pointer-TYPE.
```


Bibliography

Copies of documents in this list may be obtained by writing to The Secretary, Persistent Programming Research Group, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland.

Atkinson, M.P.
'A note on the application of differential files to computer aided design', ACM SIGDA newsletter Summer 1978.

Atkinson, M.P.
'Programming Languages and Databases', Proceedings of the 4th International Conference on Very Large Data Bases, Berlin, (Ed. S.P. Yao), IEEE, Sept. 78, 408-419. (A revised version of this is available from the department as CSR-26-78).

Morrison, R.
S-Algol language reference manual. University of St Andrews CS-79-1, 1979.

Atkinson, M.P.
'Progress in documentation: Database management systems in library automation and information retrieval', Journal of Documentation Vol.35, No.1, March 1979, 49-91. Available as departmental report CSR-43-79.

Atkinson, M.P.
'Data management for interactive graphics', Proceedings of the Infotech State of the Art Conference, October 1979. Available as departmental report CSR-51-80.

Atkinson, M.P. (ed.)
'Data design', Infotech State of the Art Report, Series 7, No.4, May 1980.

Bailey, P.J., Maritz, P. & Morrison, R.
The S-algol abstract machine. University of St Andrews CS-80-2, 1980.

Atkinson, M.P. (ed.)
'Databases', Pergammon Infotech State of the Art Report, Series 9, No.8, January 1982. (535 pages).

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
'Nepal - the New Edinburgh Persistent Algorithmic Language', in Database, Pergammon Infotech State of the Art Report, Series 9, No.8 (January 1982) - also as Departmental Report CSR-90-81.

Morrison, R.
Low cost computer graphics for micro computers. Software Practice and Experience, 12, 1981, 767-776.

Atkinson, M.P., Hepp, P.E., Ivanov, H., McDuff, A., Proctor, R. & Wilson, A.G.
'EDQUSE reference manual', Department of Computer Science, University of Edinburgh, September 1981.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
'PS-algol: An Algol with a Persistent Heap', ACM SIGPLAN Notices Vol.17, No. 7, (July 1981) 24-31. Also as Departmental Report CSR-94-81.

Cole, A.J. & Morrison, R.
An introduction to programming with S-algol. Cambridge University Press, 1982.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
'Algorithms for a Persistent Heap', Software Practice and Experience, Vol.13, No.3 (March 1983). Also as Departmental Report CSR-109-82.

Morrison, R.
The string as a simple data type. Sigplan Notices, Vol.17,3, 1982.

Atkinson, M.P.
'Data management', in Encyclopedia of Computer Science and Engineering 2nd Edition, Ralston & Meek (editors) January 1983. van Nostrand Reinhold.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
'CMS - A chunk management system', Software Practice and Experience, Vol.13, No.3 (March 1983). Also as Departmental Report CSR-110-82.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Progress with Persistent Programming", presented at CREST course UEA, September 1982, revised in Databases - Role and Structure, see PPR-8-84.

Stocker, P.M., Atkinson, M.P. & Grey, P.M.D. (eds.)
Databases - Role and Structure, CUP 1984.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Problems with persistent programming languages", presented at the Workshop on programming languages and database systems, University of Pennsylvania. October 1982. To be published (revised) in the Workshop proceedings 1983, see PPR-2-83.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
'Current progress with persistent programming', presented at the DEC workshop on Programming Languages and Databases, Boston, April 1983.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
'An approach to persistent programming', in The Computer Journal, 1983, Vol.26, No.4 - see PPR-2-83.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"PS-algol a language for persistent programming", 10th Australian Computer Conference, Melbourne, Sept. 1983 - see PPR-2-83.

Morrison, R., Weatherall, M., Podolski, Z. & Bailey, P.J.
High level language support for 3-dimension graphics, Eurographics Conference Zagreb, Sept. 1983.

Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. & Morrison, R.
"POMS : a persistent object management system", Software Practice and Exerience, Vol.14, No.1, January 1984.

Kulkarni, K.G. & Atkinson, M.P.
"Experimenting with the Functional Data Model", in Databases - Role and Structure, CUP 1984.

Atkinson, M.P. & Morrison, R.
"Persistent First Class Procedures are Enough", Foundations of Software Technology and Theoretical Computer Science (ed M. Joseph & R. Shyamasundar) Lecture Notes in Computer Science 181, Springer Verlag, Berlin (1984).

Hepp, P.E. & Atkinson, M.P.
Tools and components for rapid prototyping with persistent data, to be submitted.

Atkinson, M.P., Bocca, J.B., Elsey, T.J., Fiddian, N.J., Flower, M., Gray, P.M.D., Gray, W.A., Hepp, P.E., Johnson, R.G., Milne, W., Norrie, M.C., Omololu, A.O., Oxborrow, E.A., Shave, M.J.R., Smith, A.M., Stocker, P.M. & Walker, J.
The Proteus distributed database system, proceedings of the third British National Conference on Databases, (July 1984).

Kulkarni, K.G. & Atkinson, M.P.
EFDM : Extended Functional Data Model, to be published in The Computer Journal.

Kulkarni, K.G. & Atkinson, M.P.
EFDM : A DBMS based on the functional data model, to be submitted.

Atkinson, M.P. & Buneman, O.P.
Database programming languages design, submitted to ACM Computing Surveys - see PPR-17-85.

Atkinson, M.P. & Morrison, R.
"Procedures as persistent data objects", printed in ACM TOPLAS (Oct. 1985) - see PPR-9-84.

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
"A persistent graphics facility for the ICL PERQ", to be published in Software Practice and Experience, November 1984.

Morrison, R., Dearle, A., Bailey, P., Brown, A. & Atkinson, M.P.
"An integrated graphics programming system", to be presented at EUROGRAPHICS UK, Glasgow University, March 1986.

Morrison, R., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments", presented at 8th International Conference on SE Imperial, (August 1985).

Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", presented at Data Types and Persistence Workshop, Appin, August 1985 - see PPR-16-85.

Davie, A.J.T.
"Conditional declarations and pattern matching", presented at Data Types and Persistence Workshop, Appin, August 1985 - see PPR-16-85.

Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, presented at Data Types and Persistence Workshop, Appin, August 1985 - see PPR-16-85.

Buneman, P.
"Data types for data base programming", presented at Data Types and Persistence Workshop, Appin, August 1985 - see PPR-16-85.

Cockshott, W.P.
"Addressing mechanisms and persistent programming", presented at Data Types and Persistence Workshop, Appin, August 1985 - see PPR-16-85.

Norrie, M.C.
"PS-algol: A user perspective", presented at Data Types and Persistence Workshop, Appin, August 1985 - see PPR-16-85.

Owoso, S.O.O.
"On the need for a Flexible Type System in Persistent Programming Languages", presented at Data Types and Persistence Workshop, Appin, August 1985 - see PPR-16-85.

Hepp P.E. and Norrie, M.C.
"RAQUEL: User Manual" Department of Computer Science Report CSR-188-85, University of Edinburgh.

Norrie, M.C.
"The Edinburgh Node of the Proteus Distributed Database System", Department of Computer Science Report CSR-191-85, University of Edinburgh.

Theses

The following Ph.D. theses have been produced by member of the group and are available from The Secretary, Persistent Programming Group, University of Glasgow, Department of Computing Science, Glasgow G12 8QQ, Scotland.

W.P. Cockshott
Orthogonal Persistent, University of Edinburgh, February 1983.

K.G. Kulkarni
Evaluation of Functional Data Models for Database Design and Use, University of Edinburgh, 1983.

P.E. Hepp
A DBS Architecture Supporting Coexisting Query Languages and Data Models, University of Edinburgh, 1983.

G.D.M. Ross
Virtual Files: A Framework for Experimental Design, University of Edinburgh, 1983.

G.O. Owoso
Data Description and Manipulation in Persistent Programming Languages, University of Edinburgh, 1984.

Persistent Programming Research Reports

This series was started in May 1983. The following list gives those produced and those planned plus their status at 15 November 1985.

Copies of documents in this list may be obtained by writing to The Secretary, The Persistent Programming Research Group, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ.

PPR-1-83	The Persistent Object Management System	[Printed]
PPR-2-83	PS-algol Papers: a collection of related papers on PS-algol	[Printed]
PPR-3-83	The PS-algol implementor's guide	[Withdrawn]
PPR-4-83	The PS-algol reference manual	[Printed]
PPR-5-83	Experimenting with the Functional Data Model	[Printed]
PPR-6-83	A DBS Architecture supporting coexisting user interfaces: Description and Examples	[Printed]
PPR-7-83	EFDM - User Manual	[Printed]
PPR-8-84	Progress with Persistent Programming	[Printed]
PPR-9-84	Procedures as Persistent Data Objects	[Printed]
PPR-10-84	A Persistent Graphics Facility for the ICL PERQ	[Printed]
PPR-11-85	PS-Algol Abstract Machine Manual	[Printed]
PPR-12-85	PS-Algol Reference Manual - second edition	[Printed]
PPR-13-85	CPOMS - A Revised Version of The Persistent Object Management System in C	[Printed]
PPR-14-85	An Integrated Graphics Programming Environment	[Printed]
PPR-15-85	The Persistent Store as an Enabling Technology for Integrated Project Support Environment	[Printed]
PPR-16-85	Proceedings of the Persistence and Data Types Workshop, Appin, August 1985	[Printed]
PPR-17-85	Database Programming Language Design	[Printed]
PPR-18-85	The Persistent Store Machine	[Printed]
PPR-19-85	Integrated Persistent Programming Systems	[Printed]
PPR-20-85	Building a Microcomputer with Associative Virtual Memory	[Printed]
PPR-21-85	A Persistent Information Space Architecture	[In Preparation]
PPR-22-85	Some Applications Programmed in a Persistent Language	[In Preparation]
PPR-23-85	PS/Algol Applications Programs	[In Preparation]
PPR-24-85	A Compilation Technique for a Block Retention Language	[In Preparation]
PPR-25-85	Thoughts on Concurrency	[In Preparation]
PPR-26-85	An Exception Handling Model in a Persistent Programming Language	[In Preparation]
PPR-27-85	Concurrency in Persistent Programming Languages	[In Preparation]
PPR-28-85	A Type Theory for Database Programming Languages	[In Preparation]