MPS Runtime Reference Manual

11 AUG 72

MPS 10.0

James G. Mitchell*

Xerox Palo Alto Research Center*
3180 Porter Drive
Palo Alto, CA 94304
(415) 493-1600

Stanford Research Institute
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 326-6200

This document describes the implementation of the Modular Programming System. It is intended for use by the implementors of MPS, and contains the following sections:.PBS;.LBS=1;

The allocation of registers in the MPL run-time environment: (registers)

The layout of the fixed part of a dseg: (dseg)

The layout of the system dseg transfer vector and the global fixed tables used in MPS: (sysdseg)

The machinery which supports function call/return, both compiler generated code and the run-time support code: (functions)

The machinery which supports ports and port calls (portcalls)

The description of MPL object modules, including the code and symbol table information blocks: (modules)

Format of the object code block in an MPL module: (code)

Structure of the symbol table information in an object module as constructed by the MPL compiler: (symbols)

The implementation of SIGNALs: (signals)

The temporary string facilities provided for bootstrapping MPS: (strings)

The free storage package used for the HEAP; (fsp)

An overview of the process by which the MPS is bootstrapped into existence initially: (bootstrap)

(registers).PBS;.LBS=1; Register Assignments: MPL definitions for register assignments can be found in (mps,mplregs,1:Wn) and can be INCLUDE'd.

0:NULMSG holds the null message for PORT system

1:A1 first scratch ac

2:A2 scratch ac 2

2:RS result stack pointer

3:A3 scratch ac 3

4:A4 scratch ac 4

5:A5 scratch ac 5

6:A6 scratch ac 6

7:RP record pointer (for field variables)

10:M stack mark (frame pointer)

11:S stack pointer

12:P pointer to recent PORT

13:C code segment

14:D data segment

15:LB link base register

16:SJ for system JSP's

17:SD system data segment

(dseg).PBS; Dseg Layout: MPL definitions for the fixed part of a dseg are given both as record declarations and as a set of displacements (compile-time constants) in (mps,processdefs,1:wn) which can be used as an INCLUDE file.

Base of frame

0: FakeExtLoc: ADDR SYSUFL(SD) to catch stack underflow

1: FakeRetWrd: Z RetLoc pointer to RetLoc in this word

2: FakeOldM: WORD 0 fake oldF pointer

3: FakeSysWord: WORD 0 head of enable list, if any

Ties to stack segment, code segment, this data segment itself, and owner's dsg

4: SegNums: XWD dsegnum,csegnum

5: Bases: ZWD dsegbase,csegbase absolute addresses

6: StackSeg: ADDR stackbase segmented address

State

7: StatePC: Z pc value for process

10: StateM: Z base of current stack frame

11: StateS: Z top of stack

```
  Call/Return machinery

    12: LocPtr:   Z   RetLoc  used to provide not-in-memory traps

    13: RetLoc:   XCT  ZSegInCore    /MOVE C,Bases  ; Bases
    absolute/

    14:            MOVS  D,C   load code base register

    15:             JRST @-2(S)   start at ExtLoc word in frame

(sysdseg).PBS;.LBS=0;System Dseg Transfer Vector Area
: at the moment there is no MPL INCLUDE file for these
definitions; however, the vector is declared in mplrun, and a set
of the declarations appears in nucleus.


  %...biggest index used so far -- 156B %


  %return code%

    sysrtn=40B    %simple returns%

    sysdrtn=124B  %deallocation returns%

  %call trap cell%

    fnt= 42B

  %stack overflow%

    sysovr= 44B %overflow on frame allocation%

    sysov2= 46B %overflow from within body of code%

    sysov3= 50B %overflow on AOBJP of local call%

  %allocation%

    xmake= 52B %make array%

    xmkstr= 54B %make string%

  %processes%

    create= 56B %create process%

    destroy= 60B %destroy process%

    run= 62B %run process%
```

```
    stop= 64B %stop process%
    pshenv= 66B %push environment%
    popenv= 70B %pop environment%
  %ports%
    msgtrap= 72B
    pendingfault= 74B
    portcall= 76B
  %signals%
    sig=100B
    sigport=102B
    sigprocess=104B
    err=152B
    errport=154B
    errprocess=156B
    ctunw=106B %catch phrase unwind for exit%
    ctunw=106B %catch phrase unwind for exit%
    resume=110B %"return" from signal%
    propsig=112B %propagate signal%
  %joins%
    joinboth=114B %Join PORT and PORT%
    joinproc=116B
    joinvar=120B
    jointo=122B
    bind=126B
  % number of arguments checking %
    toofewargs=130B  % caller supplied too few arguments %
    toomanyargs=132B % caller supplied too many arguments %
```

```
    unboundfn=134B  % calls on unresolved fn descriptors come
    through here %,

    undefinedsignal=136B  % all signal variables are initially
    bound to this signal code %;

  % binding and other faults associated with control %

    undefinedsignal=136B  % all signal variables are initially
    bound to this code %,

    segfault=140B  % segment out of VM fault %,

    badportcall=142B  % system called code for invalid port call
    %,

    controlfault=144B  % port call on a non-active process %,

    resolutionfault=146B  %  port call on an unconnected port %,

    portretry =150B  %  pc of a process which stopped because of
    a resolution or control fault points here for retry of the
    port call %;

  % fixed tables %

    xwdtab=140B

    argcheck=200B
```

(functions).PBS; MPS Function Calls

Function returns, arguments, and local variables are stored on a stack.

Two machine registers are reserved for addressing the stack, a top-of-stack pointer S and a frame pointer F.

The stack has the following appearance:

```
S>      [temporaries]
        [local variables] - K words
M>      old M
        return link
        saved return (inter-segment calls)
        [arguments] - m words
old S>
```

The stack is associated with the currently executing process (docmds,processes,:)

MPL declarations for a stack frame, both from the point of

view of the frame pointer and from the point of view of a pointer to the saved return word are contained in the MPL INCLUDE file (mps,processdefs,1:wn).

The function call F(a1, ..., aN) results in the following code:

```
Same segment                Other segment
------------                -------------
PUSH S,a1            )
 ...                 ) -- same: push arguments
PUSH S,aN            )
AOBJP S,@SYSOV3(SD)    XCT  @SUBR(D)
PUSHJ S,ENTRY(C)       PUSHJ S,@SUBR+1(D)
SUB S,XWDTAB+N       ) -- same: delete arguments
```

XWDTAB is global and in a fixed location.

XWDTAB[j] contains XWD j+3,j+3.

For inter-segment calls, the cells SUBR(DATA) and SUBR+1(DATA) contain:

```
ADDR RETLOC
ADDR XENTRY(C)
```

The entry (F) PROCEDURE(p1, ..., pM) compiles the following code:

```
XENTRY: HRRZ A1,@0(S)
        XCT ARGCHECK-XWDTAB-M(A1)
        MOVS D,C
ENTRY:  PUSH S,M
        XCT FNT(SD)
        ADD S,XWDTAB+K-3
        JUMPGE S,@SYSOVR(SD)
```

ARGCHECK is global and in a fixed location.

ARGCHECK[0] contains PUSH S,LOCPTR(D).

APGCHECK[-j] contains JSP SJ,@TOOFEWARGS.

ARGCHECK[j] contains JSP SJ,@TOOMANYARGS.

The register SD contains the address of the system data segment.

FNT(SD) contains MOVE M,S or a jump to the system if currently tracing procedure calls.

SYSOVR(SD) contains a jump to system code to handle stack overflow.

K is the size of the stack frame for F's local environment.

The statement RETURN compiles the following instruction:

```
        JRST @SYSRTN(SD)
```

where SYSRTN(SD) points to RTNCOD

```
RTNCOD: MOVE RS,S
        MOVE S,M
        MOVE M,0(M)
        JRST @-1(S)
```

The register RS (currently = A2, see REGISTERS above) is used by the caller to retrieve the results.

Returns with multiple results use the same system code.

The results are pushed in reverse order.

RETURN(r1, r2, ... rn) results in

```
        PUSH S,rn
        ...
        PUSH S,r2
        PUSH S,r1
        JRST @SYSRTN(SD)
```

On the calling side, x ← p(: m1, m2, ... mn) results in

```
        ...
        SUB S,XWDTAB
        POP RS,x
        POP RS,m1
        ...
        POP RS,mn
```

If the lhs's would endanger the results on the stack (e.g. *(f())*), then life becomes more complex. The code for this case is

```
SUB S,RS

MOVEM S,mrcell  % mrcell is a compiler-allocated temporary
%

MOVE S,RS

POP's using S

ADD S,[n,,n]

ADD S,mrcell
```

This scheme protects the results by moving the stack pointer over them and allows the caller to accept any initial segment of the multiple results. (I.e. if the routine returns 4 results and you only want the first 2, then you need only store the first two.)

Multiword scalars of length n are treated like n single word scalars.

In addition, there are some global data structures associated with the function machinery.

Each linkage (static data) segment contains a few words of descriptive information:

```
BASES:  XWD LBASE,CBASE; Linkage and code base addresses
LOCPTR: ADDR RETLOC; Pointer to return routine

RETLOC: XCT ZSegInVM ;absolute address of cell in global
table
        MOVS D,C
        JRST @-2(S); Return through saved actual link
```

There is a global table with one entry for each instance of a code-data pair:

ZSegInVM is a two-word cell containing in its first word either

MOVE C,BASES (BASES an absolute address), or

JSP SJ,@SegFault(SD) if the code segment associated with the data segment is not swapped in.

The second word of each entry is used to save the BASES address when the JSP instruction occupies the first word, and the segment number of the associated code segment (or possibly a link chaining all the entries for one code segment together).

Times for function calls using this scheme:

Total local call/return is about 29 microseconds.

Total external call/return is about 45 microseconds.

These times are for calls with no arguments. For n words of args, add 3.9 * n for PUSH'ing before the call.

(portcalls) .PBS;

MPS port call machinery

Format of a port:

```
port:  ZWD  port,objectport  % pointer to object port to
which connected and to myself (absolute addresses) %

msg:   XWD  400000,0   % message buffer (initially contains
the null message %

dsegptr:  ADDR  dsegaddr  % dsegaddr is the absolute address
of the dseg in which the port resides %

startup: JRST @statePC(LB)   % used by portcall machinery to
resume the process to which this port belongs %
```

MPL declarations for ports are contained in the INCLUDE file (mps,processdefs,1:wn).

Code for *[*lhs '←*]* 'PORT '(portname *['*, message*]* *['*! ctchp*]* ');

In-line code:

The in-line code produced depends on

(a) whether the portname is local, and

(b) whether the PORT call is in an expression.

If the portname is not local, then the address of the port is calculated and saved. If it is local, then it can simply be addressed directly.

If the PORT call is used in an expression, then control must reenter the process over the same port and there must be a non-null message waiting in the port or an appropriate trap is caused.

The following is a detailed description of the in-line code produced.

First, it may be necessary to save certain registers on the stack if the PORT call is part of a complex expression being evaluated.

Message value is loaded into register A1.

If the portname is local then

Load register P with the connection word from the port.

```
JSP SJ,@PORTCALL(SD)
```

If there is a catch phrase, produce code for it here.

Else (portname not local)

Load address of port.

Load register P with the connection word from the port.

If the PORT call used in expression then push port address on stack so that can check later.

Push D register so can restore later.

This is necessary because may be doing PORT call for some other process, which means D not equal to LB. System code for port calls sets D to LB when starts up process, so when control comes back to this process will not have proper value in D.

JSP SJ,@PORTCALL(SD)

If there is a catch phrase, produce code for it here.

Pop stack to D register.

If the PORT call used in expression, then POP stack to register so that can test later to ensure that control has returned over same port. (Register RP is used for this currently).

If the PORT call value was to be loaded into some register other than A1, then move it.

Now restore any registers which were saved initially.

If the PORT call was not used in an expression, then done.

Else, must check port and message.

HRRZ P,P to make P = address of port.

If portname was local then CAIE P,portaddr

Else CAME P,RP compare to address which was saved on stack.

JSP SJ,@PENDINGFAULT(SD) to indicate that control did not return over same port that left.

CAMN ac,NULMSG compare message to null message.

JSP SJ,@MSGTRAP(SD) to indicate that port call returned null message.

If there was a catch phrase, then produce (nonjumping)

JUMP instruction here which addresses the start of the catch phrase code.

Finally, MOVEM NULMSG,PortMessageBuffer to indicate that the message has been "removed" for use in the expression.

code in MPLRUNTIME:

```
portcallX: HLRZ A2,P  % must ensure that port belongs to
process in control %

CAME LB,dsegptr(A2)  % compare controlling dseg address and
dseg addr to which subject port belongs %

JUMPA BadPortCall(SD)  % bomb out if not equal %

MOVEM S,StateS(LB)  save stack pointer of subject process

MOVEM M,StateM(LB)  save frame pointer

MOVEM SJ,StatePC(LB)  save program counter

MOVSM P,(P)  make object port connected to subject port

HRRZ LB,dsegptr(P) begin startup of object process by
getting base of his dseg.

XCT RetLoc(LB)  %move Bases into register C (and check for
code segment not-in-memory)

MOVE D,LB  load D register (=LB contents)

MOVE M,StateM(LB)  load frame pointer

MOVE S,StateS   and stack pointer for object process

JRST  startup(P)    fire up the object process
```

(modules).PBS;

(code) MPS Object Code Format: MPL declarations for the blocks in an object module may be found in the INCLUDE file (mps,moduledefs,l:wn).

An MPS object code file consists of a series of blocks, each carrying length and type information.

The first word of a block is XWD TYPE,LENGTH.

The TYPE is one of the following codes:

0 - empty space

1 - code

2 - hash table

3 - symbol data

4 - name table

5 - structure table

The LENGTH is the number of words in the block, including the header word.

LENGTH=0 is an error, except that TYPE=LENGTH=0 signifies the end of the file.

The rest of the block (LENGTH-1 words) is interpreted according to TYPE.

The first block in the file always contains the code for a module.

The first few words of the code block contain information for the MPS loader.

Word 1 contains a zero if this is a DATA module; otherwise

Word 1 left half contains

the displacement of the first instruction of the code segment after the initialization code (i.e. the procedures).

Word 1 right half contains

the displacement of the first instruction of the process body (i.e. after the procedures).

Word 2 left half contains

the displacement of the first of the literals (i.e. the word following the actual code).

Word 2 right half contains

the size of the data segment.

Word 3 right half contains

hash index for module name.

Word 4 is the entry for the initialization code.

This is a procedure of one argument (the number of words of arguments passed to the CREATE operation). These arguments have been stored into the first words of the data segment. If the count is incorrect then there will be a trap to the system like that for a procedure call with the incorrect number of arguments.

The code itself follows.

The code assumes that the CODE base register is loaded with the origin of the file, i.e. the address of the TYPE,LENGTH word of the code block.

Unlike the situation in L1O, all subsequent words in the code area are genuine instructions.

Literals follow the code.

Every module also has a hash table block, a block of semantic entries, a block of name strings, and a block giving structural information relating the object code to the source statments.

A type 2 block (hash table) has an extra word following the LENGTH word which contains in its left and right halves, respectively, the following:

The left value, x, is the size in words of the index portion of the hash table which follows,

The right value, h, is the number of hash table entries in the block beyond the index portiion of the table.

These two values are required by any routines using syml1O to access the tables because the routine settbl needs to be told x and h as its second and fourth arguments, respectively.

Let m be the value in the right half of the 2B6 header; then m, x, and h are related as

m = 2 + x + 2*h
i.e., m includes the two header words, the number of index words, and the hash entries (which currently require two words each).

All relative links between the hash table and the name and semantic blocks are relative to the first data word of the block; this is word 1 in the semantic and name blocks, and word 2 in the hash table block.

The format of semantic entries is described in (DOCMPS,DOCSYM,).

A type 5 (structure) block contains a sequence of words of the form XWD lc,,bytes.

The lc indicates the displacement in the code corresponding to the start of a source language statement (SLS).

The bytes are six bits wide and are used to encode the structural position of this SLS relative to its predecessor (or to the origin if this is the first SLS).

The first byte is treated specially:

Byte=N, N IN [0,63] -- go up N levels in the structure. In other words, depth in the tree is decreased by N.

All subsequent bytes are treated the same, namely:

Byte=0 -- stop here.

Byte=N, N IN [1,63] -- take Q=N MOD 16 successors, Q IN [1,15], and then go down M=N/16 levels, M IN [0,3]. When go Q successors, increment position at current depth by Q. When go down M levels, M > 0, increment depth by Q and set the position at each new level to 1.

As many words are used as necessary to encode the new position; the lc is simply the same in each word. Most common relationships can be encoded in a single word.

(symbols) Context usage in MPL(A): MPL declarations for context usage in object modules, and symbol type declarations can be found in the INCLUDE file (mps,modulesymdefs,1:wn)..PBS;

CTX pdctx

For module names, procedure descriptors, and directory link strings.

CTX x1ctx

For first word of external or forward procedure calls -- not output.

CTX x2ctx

For second word of external or forward procedure calls -- not output.

CTX lsctx

For literal strings.

CTX metctx

For tokens and special variables used with MPLMETA.

CTX mplctx

For special use in compiler -- not output.

CTX prctx

For outermost scope of module.

Currently pdctx=0, x1ctx=1, x2ctx=2, lsctx=3, metctx=4, mplctx=5, prctx=6.

The higher numbered contexts are used for

INCLUDE'd modules

procedures

Format of semantic data entries: MPL declarations for the symbol type field and its possible values can be found in the INCLUDE file (mps,modulesymdefs,1:wn).

in first word of entry

attributes -- 13 bits at position 19

from right to left in the attribute field

defned -- defined id

linked -- referenced but not yet defined

const -- compile-time constant

noout -- do not output to object file

type[5] -- type of the id (see below)

word 2 of se holds system info

word 3 of se holds *V

word 4 of se is used for constval and dirlink

word 5 of se is used for nwords and contx

word 6 of se is used for numarg

(symtypes) Types of entries

UNDEF=0 undefined

Semantic entries are initialized to zero, so type is automatically UNDEF when entry created.

PROCD=1 local procedure

Under prctx

*V is external entry point

(*V+3 is internal entry point)

@contx holds the context number for local declarations

@numarg = number of WORDS of arguments

If has entry under pdctx

*V is location of procedure descriptor in dseg

This should be initialized by CREATE to hold actual descriptor.

May have had entries under x1ctx and x2ctx if there were calls made on the procedure before it was defined. These entries used to fixup such calls, then are "deleted" (not output).

LAB=2 label

*V is location of first instruction of the statement following the label.

MWS=3 multiword scalar

*V is loc of first word

@nwords = number of words

PORT=4 ports

*V is loc of first word

@nwords = number of words

SIGCOD=5 signal code

*V is loc

CREATE should initialize this cell to hold its own segmented address. All sigcod's are in the dseg, even if declared in a procedure.

@nwords = number of words

VAR=6 normal identifier

if @S const then this is constant and value is in @constval

else *V gives loc of variable

Under metctx -- a special variable for compiler.

*V gives location in dseg.

Name indicates which variable.

'nexttoken' -- pointer to next input token (for META).

'outword' -- pointer to word holding next output (for OUT).

'outline' -- pointer to string holding next output (for SOUT).

FIELD=7 id's declared as FIELD or in RECORD

if @S const then this is a constant field and @constval holds the field descriptor (byte pointer)

else is variable field and *V gives the loc

MODULE=8 names of modules

Under pdctx

if *V # 0 then this module was INCLUDE'd and *V = context number for declarations in that module

else this module name was not INCLUDE'd but has been used in CREATE or other such statement

if @dirlink # 0 then module was INCLUDE'd and @dirlink = hash of string used to access the module.

If the link contained (dir,file,junk), then the string will be <dir>file followed by a zero character.

XPROC=9 external procedure

Under pdctx

if @dirlink # 0 then this name was listed in DIRECTORY and @dirlink = hash of string containing link from directory

The string actually contains everything that was written between the parentheses of the link.

@numarg = number of WORDS of args assumed (or -1 if no calls actually made)

*V = loc of first word of procedure descriptor in dseg.

CREATE should initialize descriptor to trap if used before bound.

Entries under x1ctx and x2ctx used to fixup calls but not output.

ARRAY=10 statically allocated arrays

*V contains addr of first word of array

RECORD=11 for id used as name of record

@nwords contains RECORDSIZE (i.e. the number of words needed to hold an instance of this record).

PROCV=12 variables declared to be procedure

*V is loc of first word of descriptor

Where possible (i.e. in dseg) these should be initialized to trap.

@nwords is size of descriptor

STRING=13

*V is loc of pointer to descriptor

Under 1sctx

literal string -- initialized by CREATE

Under other contexts -- string variables

ARRAYV=14 array variable (holds pointer to array)

REGISTER=15 fixed location scalar

SIGVAR=16 signal variable (holds signal code)

UXPROC=17 unreferenced external procedure -- not output

NODENAME=18 symbol used as tree name in MPLMETA construct

Under metctx.

*V gives location in dseg for name table index when initialize the parser.

TOKEN=19 symbol used as token in MPLMETA construct

Under metctx.

*V gives location in dseg for token when initialize the parser.

Use of father-son linking capabilities

The se for the name of the module being compiled is the root of the tree.

The hash index for the module name is stored in the object file as described elsewhere in this file.

Immediate sons of this se are

1) outref XPROC's and literal strings

2) module arguments

3) directory names

if directory entry is an INCLUDE, then tree for the included declarations is under the se for the name of the module.

4) variables declared in the module

5) procedure names

6) labels in the body of the module

Sons of the procedure name se are

1) formal parameters

2) local declared id's

3) local labels

Records are structured as tree under the se for the record name.

The tree structure reflects the structure used in the record declaration.

(signals).PBS;

Significant features

Continued propagation through dynamic scope until signal terminated.

Pass message as well as signal code.

Return value as result of signal.

Special UNWIND signal to allow cleanup.

Signal frame

A signal frame contains

a flag set in the return word indicating a signal frame

pointer to the frame whose catch phrases are being executed

pointer to the process which originated the signal

global name of signal code (segmented address)

signal message

a flag indicating whether RESUME is legal for this signal

The signal frame is always on the stack of the process whose catch phrases are being executed.

Signal propagation and termination

A signal is said to be terminated when some catch statement executes either

a RESUME, or

a branch out of the catch phrase (such as RETURN or EXIT).

In the first case

the signal frame is deleted

If a RESUME is legal, then

the process which originated the signal is given control

Else a new SIGNAL is generated indicating attempt to resume after "unresumable" signal.

In the other cases

the stack of the process terminating the signal must be unwound back to the point where the signal is being terminated.

Before a frame is deleted, a special signal (UNWIND) is given so that any necessary housekeeping can be performed.

If a signal is not terminated within a process, then it is propagated to the creator of that process. The signal frame is deleted from one stack and recreated on the other.

RESUME statement

RESUME can return a value and thus any signal (SIGNAL, SIGNAL PORT, or SIGNAL PROCESS) can be used in an expression.

If RESUME does not explicitly specify a value, then the special value NULLMESSAGE is used. If a signal is used in an expression and gets back a NULLMESSAGE, then a MESSAGEFAULT trap is caused.

The same trap occurs when a PORT is used in an expression and fails to return a value and when a EMPTY specifies a port containing NULLMESSAGE.

It is possible to attach a catch phrase to the signal to handle this trap or others.

It will often be the case that the signalling program is not willing to be resumed. To make this explicit and to allow the system to intercept illegal attempts to resume, 'ERROR may be used in place of 'SIGNAL. Thus ERROR, ERROR PORT, and ERROR PROCESS all behave like the corresponding SIGNAL statements, except control can not return be means of a RESUME.

Scope of catch phrase and propogation of signals

A catch phrase covering a statement list may occur at the beginning of

the program body,

a procedure body, or

a block.

Such catch phrases are enabled throughout the execution of the statement list. They are automatically disabled when the statement list is completed or left by a branch statement.

A catch phrase which covers a function call or port call is enabled when control leaves and is disabled when control returns.

A catch phrase is never enabled when it is being executed. In other words, the scope of the catch phrase does not include the catch phrase itself.

Implementation

A system routine is called to propagate signals.

This routine "calls" the innermost enabled catch phrase with the signal code, signal message, and pointer to original frame as arguments.

The catch phrase is compiled to use the pointer to access local variables of the origial frame.

Control returns to the system routine if the signal is to be propagated beyond this frame.

Thus the main problem for the system routine is to determine the entry point of the innermost enabled catch phrase.

If control left the frame by a function or port call, then a catch phrase associated with that call would necessarily be the innermost one.

The reactivation location for the frame (stored as the process PC or the return location of the frame above it on the stack) is used to look for the presence of a catch phrase associated with the call.

In the PDP10 implementation, catch phrases always begin with a JUMPA instruction which branches around the body of the catch phrase. The system routine looks for the presence of this instruction following at the reactivation location to determine whether there is a catch phrase with the call.

If there is no catch phrase with the call, then the innermost enabled catch phrase is associated with a statement list containing the reactivation location.

To help the system routine find the entry point in this case, all frames contain a pointer to the innermost enabled catch phrase which is associated with a statement list.

If the pointer is zero, then there are no enabled statement list catch phrases.

A flag in the return word (which is automatically zeroed on calls) indicates whether this pointer has been set. If the flag is zero, then there are no enabled statement list catch phrases.

If the signal is not terminated by the catch phrase, then it must be propagated. If this is the last (i.e. outermost) enabled catch phrase associated with the frame then control is simply returned to the system routine. Otherwise, control is transfered to the innermost enclosing enabled catch phrase (which can be determined at compile time since catch phrase scope is lexical).

Signal codes and signal variables

The signal codes name the signal. They occur as the first argument of the signal statement and at the head of catch phrase cases.

There are system defined codes (UNWIND, PORTFAULT, etc.) and user defined codes.

User defined codes are simply identifiers. There is no special declaration for such codes. An identifier used as a signal cannot be used in other capacities within that context.

Since signals can be passed between processes, it must be possible to indicate that a signal code in one process is to be the same as another signal code in another process. This is simply a name binding problem and is handled by the usual machinery (i.e. signal codes are bound by the same mechanisms that are used to bind external procedure names).

There are SIGNAL variables which can hold signal codes. (SIGNAL variables are to signal codes as PROCEDURE variables are to external procedure references).

Misc

In those cases where a trap can be caused after control is returned, special means are required to determine if there was a catch phrase associated with the call.

Examples

PENDINGFAULT -- control did not come back thru the same port

MESSAGEFAULT -- expecting message but did not get one

In the PDP10 implementation, these traps are initiated by inline code following the call (and following the catch phrase if there is one).

The trap is actually a call to a system routine which generates an appropriate signal. If there is a catch phrase associated with the original call, then it must be given a chance to catch this signal. Since the pointer to the call is no longer available, it is instead stored by the compiler as the address of a JUMP instruction (which is actually a NOP) following the system call to produce the trap. If there is no JUMP instruction following the system call, then there is no catch phrase with the original call.

Since a test for PENDINGFAULT is always followed by a test for MESSAGEFAULT, there is only one JUMP instruction produced which is "shared" by these two. The PENDINGFAULT system routine knows to look past the inline code for the MESSAGEFAULT test for the JUMP instruction.

(strings).PBS;

This section describes a string system for MPS which we will implement in order to get MPS off the ground.

We intend that it be replaced by something closer to the proposal in (DOCSTR,) at some future date.

Language syntax and semantics

A variable of type STRING is meant to hold a pointer to a string descriptor.

String descriptors are allocated from a "heap", either automatically or by system functions accessible to the programmer.

A STRING variable gets a descriptor allocated for it on

procedure entry (or process creation), and deallocated on procedure exit (process destruction).

A dimensioned string, like a dimensioned array, gets its body allocated in the same way.

The automatic allocator actually associates with each frame or process a list of the storage allocated for it, so the right thing happens even if a string variable is subsequently used to hold a pointer to a user-allocated descriptor.

Types of string descriptors:

1) Explicit-string descriptor

points to block of characters

Fields

Front

first character of text block which is contained in this string

End

first character of text block which follows this string

Maxend

maximum value for End before overflow this text block

Pointer

address of text block

Ident

This field is available to hold program-specific information. It can be written and read by user programs and is intended to hold information which will help the program identify the string.

2) Implicit-string descriptor

Fields

ReadFunction

Descriptor of function used to read characters in the string.

WriteFunction

Descriptor of function used to write characters in the string.

LengthFunction

Descriptor of function used to find and set the length of the string.

Ident

Same as above.

On a read access, the system returns the result of

ReadFunction(String, Position).

On a write access to the string, the system calls

WriteFunction(String, Position, Char).

When the length of the string is requested, the system returns

LengthFunction(String, 0).

To set the length of a string, the system performs

LengthFunction(String, 1, Nchars).

The LengthFunction is intended as a catch-all for which additional uses may be found in the future.

There is no special syntax associated with strings.

Assignment for strings is defined as simply copying the pointer to the descriptor.

Mention of a string variable refers to the pointer.

Special action is taken for literal strings.

For the moment, literal strings may appear only in the program, not as initialization or parameter values.

Functions to be provided

Where a string is listed as an argument, a pointer to a string descriptor is actually required.

Functions for setting the fields of string descriptors

MakeStrDesc(String, Front, End, Maxend, Pointer)

The descriptor pointed to by String is made an explicit-string descriptor with fields set to the values passed for the other arguments.

(The Ident field is unchanged by this operation. It is initialized to zero when the descriptor is created.)

The primary use of this function will be to make a descriptor which points to a text body in a private storage area.

(I'm afraid this will require knowledge of how characters are counted in text blocks.)

MakeImpStrDesc(String, ReadFcn, WriteFcn, LengthFcn)

The descriptor pointed to by String is made a implicit-string descriptor with fields set to the values passed for the other arguments.

(Again, the Ident field is unchanged.)

SetDescIdent(String, Ident)

The Ident field of the descriptor is set to the given value.

NumWords ← WordsForBody(Nchars)

Returns count of how many words will be required to hold the specified number of characters.

Functions for getting info from string descriptors

From any string descriptor

STRType(String)

STRIdent(String)

STRLast(String)

Returns the index of the last character in the string.

STRLength(String)

Returns the current length of the string.

The values of STRLast and STRLength are calculated from

the values of the Front and End fields for explict type strings and from the value of LengthFunction for implicit type strings.

The following functions simply provide access to various fields of the descriptors.

From explicit-string descriptors

STRFront(String)

STREnd(String)

STRMaxend(String)

STRPointer(String)

From implicit-string descriptors

STRReadFcn(String)

STRWriteFcn(String)

STRLengthFcn(String)

Functions implementing language features

The following functions are needed to implement string features which will someday be added to the language.

Functions for accessing characters

These two functions are used to implement string[exp] as a left hand side in MPL.

NthChar(String, Position)

Loads the character from the specified position. If Position is not within the bounds of the String a special value EOS (End Of String) is returned.

SetNthChar(String, Position, Char)

Writes the character at the specified position. If Position is beyond the end of String, an error is generated.

Functions for string construction

SetStrNull(String)

Resets the string, i.e. sets End=Front.

SetStrLength(String, Nchars)

Sets the length of the string to Nchars, which must lie in [0, Maxend-Front].

AppendString(To, From)

AppendChar(To, Char)

AppendSubstring(To, From, First, Last)

If First<0, First is taken as 0; if Last>STRLast(To), Last is taken as STRLast(to).

AppendBlanks(To, Count)

Functions for creating and destroying strings

The lifetime of a declared string is limited to the lifetime of the scope in which it is declared. In other words, when a procedure returns or a process is destroyed all strings which were declared in that procedure or process are automatically deleted.

The following functions provide for the creation of strings whose lifetime is explicitly controlled by the programmer.

refstring ← MakeString()

The function MakeString returns a pointer to a descriptor for a null string.

ReleaseString(refstring)

The referenced string is deleted.

Functions for general storage allocation

There are also some procedures for allocating and deallocating storage on the heap, whose use is not limited to strings.

refblock ← MakeBlock(n)

Returns a pointer to a newly created block of n words. The block is guaranteed to contain only zeros.

ReleaseBlock(refblock)

Deletes the referenced block. As usual, the programmer is responsible for ensuring that no pointers to the block remain.

SplitBlock(refblock, n)

The referenced block is split. After the operation, refblock refers to an n-word block consisting of the first n words of the old block; the rest of the old block is deleted. If n is greater than the length of the block, an error occurs.

BlockSize(refblock)

Returns the length of the block in words.

Implementation

The RH of the system word of a procedure frame is the head of a chain of automatically allocated blocks for that procedure.

This includes string descriptors and string and array bodies.

The allocator chains all the blocks it creates through a pointer in *[-1,18:0]* of the block.

[-1,17:18] is used for the block length and *[-1,1:35]* for a free flag.

There is a bit in the return word in a frame which is cleared by PUSHJ and set when the first block is allocated.

The system and return words in the fake frame in a dseg are used in the same way for the automatic storage for the process.

The compiler generates a different return instruction for a procedure return if it is possible that automatic storage hs been allocated.

The deallocator must check the bit, since in the future it may be possible to have automatic storage which does not show up explicitly in the declarations.

The DESTROY procedure must check the bit for process storage.

For the moment, the literal strings appearing in a program are collected together under a new reserved context in the symbol table.

The context number is 3; the "name" is the text of the literal; the "value" is the dseg location for the pointer to the descriptor.

When a process is created from the program, the CREATE

procedure is responsible for allocating descriptors and bodies for the strings, copying their text (which appears in the symbol table) into the body area, and setting up pointers in the dseg.

The storage allocated in this way persists for the lifetime of the process just like static string variables.

(fsp) .PBS;

The free storage package provides a simplified zone type storage allocation system.

The following procedures are found in (MPS, FSP,).

MakeZone(zone,size)

The "size" words of storage starting at location given by the (virtual memory) address "zone" is initialized to be a storage zone. After this has been done the zone can be used as a parameter of the following procedures.

Links used by the free storage package are all maintained relative to the starting location of the zone. Thus the zone may be relocated without disturbing its use.

Generates the following signal:

BadZoneSize -- on PDP10 size must be between 6 and 100000B words.

node ← MakeNode(zone,size);

Returns (virtual memory) address of node of "size" words of user storage in the specified zone.

Generates the following signals:

BadNodeSize -- size <= 0,

NoRoomInZone -- cannot find space for node of size words.

NodeSize(zone,node);

Return the size in words of the node.

FreeNode(zone,node);

Release the storage occupied by the node.

SplitNode(zone,node,size);

Split the specified node into two sections -- the first of which is a node of size words, the second of which is freed.

(names).PBS;

This section describes how processes, as well as variables and procedures in them and the modules from which they were created are named and accessed.

The following rules and their consequences apply:

(a) The context in which a system routine is called, along with complete qualification within that context are both required to name a process.

Process naming syntax:

processname ::= [processid] $('. processid);

processid ::= .

If the optional, leading processid is not present (i.e., if the processname begins with a ".") then the context within which the name match is to be performed is the root of the segment naming tree. Otherwise, the context within which the name is to be matched is the context of the process whose dseg address is in the D register ("whose static data is current" as opposed to "which is in control" - the LB register is the address of the dseg of the process which is in control).

(b) Given the segment number of a dseg (possibly a stack segment) - which may be obtained by coercing a processname, an object in the dseg can be referred to by the syntax

objectname ::= processname $('. .ID);

Since a processname must be completely qualified, there is no ambiguity in the meaning of the .ID's.

Module and process names:

A complete TENEX file name is a module name. A simple identifier used as a module name must be bound to a TENEX file name. The mechanisms for accomplishing this are

(a) at compile time:

the directory of a MPL program contains an entry such as

(ModuleName) (directory,file)

Then any use of ModuleName is equivalent to using the

TENEX file <directory>file with the normal TENEX conventions on file names and completion of incomplete file names.

(b) at execution time:

The run-time segment space is accessed to determine if there exists a process whose name is ModuleName which is accessible by the conventions stated below for segment names. If such a process exists, a match has been found. Otherwise, a stack of directory files is searched to map the module name to a file name. A directory file is a sequential text file with entries of the form

'(.ID ') '(directory',file /',name/')

on separate lines (anything at all can follow the second right parenthesis).

A directory file is searched from line 0 to the last line of the file, in order. If no match is found in the most current directory, the next most current directory is searched until the directory stack is exhausted, at which time a signal (UndefinedModule) is generated.

Files can be pushed onto the directory stack or removed by the MPL procedures NewDirectory(STRING filename) and RemoveDirectory(STRING filename). If the argument to RemoveDirectory is the null string, the directory file on the top of the directory stack is removed; otherwise the named file is deleted from the directory stack, if in it. NewDirectory puts the name of the file on the top of the directory stack after first removing any occurrence of the same name in the stack.

Binding Proedure Names

An external procedure p which is not declared as a procedure variable and is used in a process X is bound at run-time when it is first called from within X. This can be overridden by an explicit BIND statement at any time or by a call on the system-supplied procedure BindProcedures(dsegname).

If there was an entry for p in X's compile-time directory then p will be bound to the procedure in the declared file, if an instance of it exists. If no instance of that file exists, the signal ModuleNotCreated is generated. If p had no directory entry in X, then the system will attempt to bind p by finding an instance of a procedure wth the same name. The name is sought according to the following algorithm:

(NameSearch):

(a) X's sibling processes are searched for an object named p of the same type as p in X. If exactly one such is found, X.p is bound to it; if more than one such is found, a signal, AmbiguousName is generated.

(b) If no match for p is found among X's siblings, then X's parent is searched for a match; if none is found, X's grandparent is searched, etc. If the root of the segment tree is reached without a match, then the signal ResolutionFault is generated.

Whenever an unbound procedure variable is called, the ResolutionFault signal is generated so that the binding may be done by any program willing to catch the signal.

(bootstrap).PBS;

This section gives an overview of the bootstrapping process by which the MPS comes into existence, both initially, and later when MPS exists to bootstrap itself.

The non-MPS beginning

An L10 program (on the PDP-10) maps into the "bottom" of memory the following files:

MPLNUCLEUS

this program will be given control after the L10 program has finished. It completes the job of creating the MPS environment. It is described more fully in the next section.

MPLRUNTIME

this program contains the run-time support code and system transfer vector for MPS. The NUCLEUS will help it to set up the environment.

SEGRUN

This is the segmentation machinery. MPLNUCLEUS will pass on information given it about the whereabouts of MPLNUCLEUS, MPLRUNTIME, and SEGRUN so that SEGRUN can initialize the segmet tables correctly.

The L10 program then allocates space at the "top" of memory for data segments for each of the three above MPL programs. The addresses of each of the programs and their data segments are placed in a fixed place in the MPLNUCLEUS dseg.

A stack segment is also allocated below the data segments at

the top of memory. It belongs to MPLNUCLEUS and its address will be placed in the standard place in MPLNUCLEUS' dseg.

The MPS bootstrapping NUCLEUS

Control is given to the MPLRUNTIME "process" so that it can initialize the system transfer vector, the SD register, and the ARGCHECK and XWDTAB vectors.

All procedure descriptors should either be bound by the NUCLEUS (which means that it must know all the uses of SEGRUN from within MPLRUNTIME and vice versa) or should cause a trap which the NUCLEUS will translate into a call on the BIND routines in MPLRUNTIME (in this case, the NUCLEUS only needs to know about a few procedure descriptors; all others will be bound as they are used).

Initializing the MPL runtime package

Initializes the system vector, the ARGCHECK and XWDTAB vectors.

Initializing the segmentation machinery (SEGRUN)

SEGRUN can use almost all the normal MPLRUNTIME facilities to initialize itself. It is passed the address of a table of pairs of addresses (fileseg, dataseg) where, if fileseg=0, it is ignored, and the associated dataseg address is taken to be the address of a stack segment, and if fileseg=-1, the entry marks the end of the table.

Creating and starting the MPS debugger (the first true MPL process)

Once the MPL environment has been established, the NUCLEUS CREATEs the MPS DEBUGGER, using all the normal MPL facilities.