

Pascal-M

M-code documentation V1.4

Summary

This document describes the M-code machine
How to implement, loader, objectcode, instructions.

Authors

Mark. D. Rustad (initial version), Hans Otten (current maintainer)

Revisions:

V1.0 9 september 1978 (Mark D. Rustad),
V1.3 1986 Hans Otten
V1.4 August 2020 Hans Otten

M-code machine

The Pascal-M compiler produces objectcode for the virtual M-code machine. This is a variant of the P-code machine that is defined in the design of the Px interpreters (see the Pascal P4 source, books, design articles).

The M-code machine can be implemented by an interpreter

At the moment several interpreters of M-code interpreters are available (others have existed but are lost for now):

- V1 KIM-1 6502 code interpreter in assembler
- V1 Pascal interpreter
- V1.3 VAX/VMS Pascal interpreter
- V1.4 Freepascal interpreter

V1.0 of the M-code machine is defined in Preliminary M-code description 1978 .doc

Changes between V1.0 and V1.4 are:

- New instructions C2, SFA – Set file address C3 GFA Get File Address
- New standard procedures OD OE, RSFRWW, OF STT , 10 CLS
- Changes to RDI RLN WRI WLN etc read/write instructions to handle file address.

Data-representation :

character : 1 byte on stack

```
      +-----+
Addr -> ! ascii code char !
      +-----+
```

boolean : 2 bytes on stack

```
      +-----+
      !           0           !
      +-----+
Addr -> ! true or false ! true = 1, false = 0
      +-----+
```

integer : 2 bytes on stack

```
      +-----+
      ! most sign part ! div 256 signed
      +-----+
Addr -> ! least sign part ! mod 256 signed
      +-----+
```

pointer : 2 bytes on stack

```

+-----+
! most sign part    ! div 256  signed
+-----+
Addr -> ! least sign part    ! mod 256  signed
+-----+

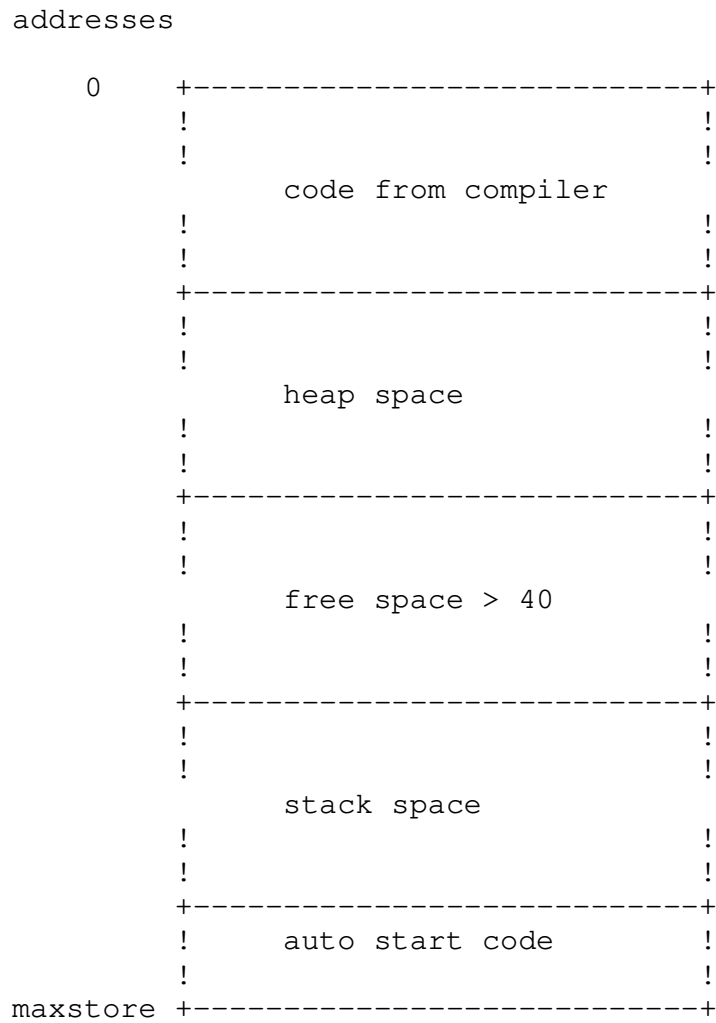
```

Addresses, pointers such as stackpointer, markpointer etc always point to the least significant byte of the data-item.

Examples are given above.

Stack layout as given in the interpreting procedures gives an impression before and after the instruction execution. The stack is actually growing from high addresses to low addresses. For example, reserving space on the stack requires decrementing of the stackpointer.

Memory layout :



M-code machine

The M-code machine is a strict stack computer following the P-code design (see the book by Pemberton for example, chapter 10)

Registers are:

Program counter
Stackpointer
Heap pointer
Markpointer

Memory is 64Kbyte, byte addressable.

At start of the M-code machine:

Heappointer starts after the last byte loaded from objectcode
Program counter := maxstore - 4 ;
Stackpointer = maxstore - 5 ;
markpointer := stackpointer

Main program fixed in upper memory
MST0
CUP1 0
CSP STOP

Stored as:

```
store [maxstore - 4] := 48 ; (* MST0      *)
store [maxstore - 3] := 190 ; (* CUP1      *)
store [maxstore - 2] := 0   ; (* proc nr 0 *)
store [maxstore - 1] := 189 ; (* CSP      *)
store [maxstore   ] := 11  ; (* Standard proc Stop
*)
;
```

Object code

The compiler produces objectcode in Px record format.

The object code produced by the PASCAL-M compiler is composed of lines of standard ASII characters. In most cases, the characters should be interpreted as hexadecimal digits. Though there are several types of records used in M-code, each type starts with an ASCII uppercase “P” and ends with the checksum of the record coded as two hexadecimal digits.

The type of the P-record is determined by the character immediately following the P. The exact format of each of the different record types is given below.

The P1 record

The P1 record is used to load object code into the interpreter’s program area. The two hex digits following the P1 P1 indicate how many bytes of object code are in this record (two hex digits per byte). After the last object code byte is this record’s checksum byte, again, two hex digits. No load address is specified in this record since the M-code loader maintains its own current loading address which is automatically incremented as P1 records are loaded.

A typical P1 record could appear as
P10B09600F9E100902BD0CA159

The P2 record

The P2 record is used for “satisfying” forward references by going back to a previously loaded area in the program and inserting an address. In this case, the four hex digitis appearing after the P2 specify the byte address to be modified (relative to the beginning of the code) and the next four hex digits specify the value to be inserted into that location and the following one (since the planted addresses are always two bytes in length).

A typical P2 record could appear as:
P20001000AFA

The P4 record

The P4 record is used to declare procedure/function entry addresses and specify the name and number of the procedure. The two hex digits immediately following the P4 specify this procedure’s number and the next 8 ASCII characters are the first 8 characters of the procedure’s name (space-filled if necessary). The procedure name specification in this case is an exception to the general rule that m-code consists of hex digits. The checksum of this record does include the ASCII characters added in according to their ASCII values.

A P4 record could appear as:
P401OUT 66

The P9 record

The P9 record signifies the end of loading. At some point, the P9 record should include the number of object records loaded (for error detection purposes) but does not at this time.

A P9 record appears as:

P9

M-code operators

The m-code interpreter actually executes the instruction set of a hypothetical stack computer. Although the design of m-code was based on the p-code produced by the P2 compiler, numerous changes have been made. Following is a description of each of the m-code operators.

0x, LDCIS – Load small integer constant

The LDCIS instruction pushes a small integer constant in the range of [0..15] onto the stack in standard integer form (16 bit representation). The constant is in the lower 4 bits of the instruction byte itself, allowing the entire operation to only require a single byte of M-code.

1X YY, LDAS – Load short address

The LDAS instruction loads the absolute address of the Yyth byte at the Xth level onto the stack. This instruction is only generated for offsets less than 256 bytes from a base address.

2X YY YY, LDA – Load Address

The LDA instruction pushes the absolute address of the YYYth byte at the Xth level onto the stack.

3X, MSTO – Mark stack at Xth level without return bytes

The MSTO instruction marks the stack in preparation for a procedure call. The MSTN instruction is used for a function call.

4X, MSTN YY – Mark stack at Xth level with return bytes

The MSTN instruction marks the stack in preparation for a function call.

5X YY, LOD1 – Load 1-byte item on stack

The LOD1 instruction loads the byte YYth at the Xth level on the stack. Before pushing the item onto the stack, it is expanded to 16 bits.

6X YY, LDO2 – Load 1-byte item on stack

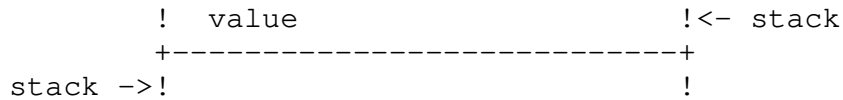
The LOD1 instruction loads the byte YYth and YY=1th bytes at the Xth level on the stack.

7X YY, STR1 – Store 1-byte data into memory

The STR1 instruction pulls one 16-bit item off the stack and stores the least significant byte into the YYth level

pulls word from stack and stores least significant byte at level at offset following opcode

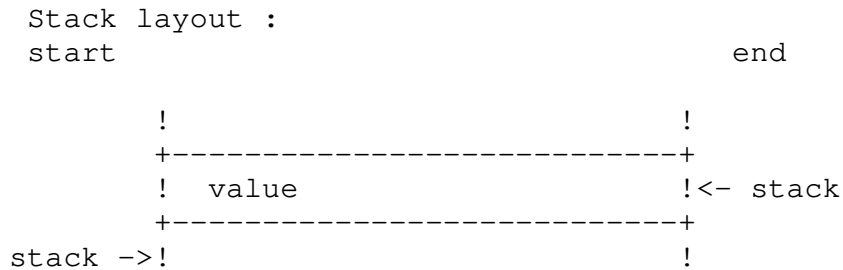
```
Stack layout :
start                                     end
!                                     !
+-----+-----+-----+-----+-----+
```

8X YY. STR2 - Store 2-byte data item into memory

The STR2 instruction pulls one 16-bit item off the stack and stores the bytes into the YYth bytes at the Xth level.

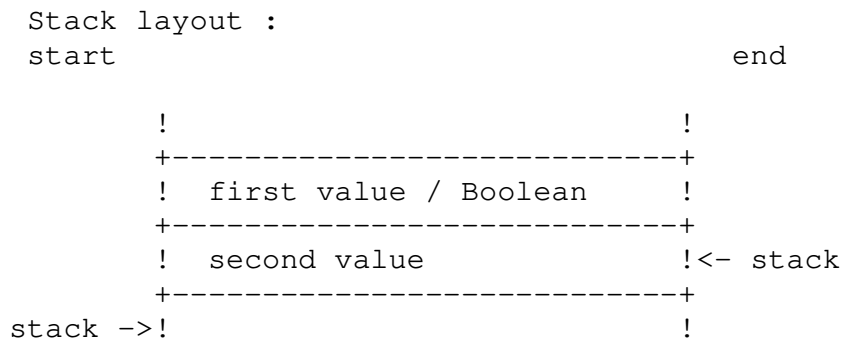
pulls word from stack and stores word
at level at offset following opcode



90, LEQ2 – 2-byte Less than or equal test

The LEQ2 instruction pulls two 2-byte items off the stack and pushes a Boolean item into the stack. The Boolean item will be a one (true) whenever the item next to the top of the stack is less than or equal to the item at the top of the stack is less than or equal to the item at the top of the stack. Otherwise, the Boolean item will be zero (false).

compares for less or equal
condition tested : first value =< second value
result : Boolean 1 if less or equal else Boolean 0



91, MFOR – For loop processing instruction

The FOR instruction is placed at the end of every loop. The instruction both initializes the loop and does the end check dependent upon data on the stack. For this reason, it is necessary that there are no goto in the language and that no extraneous data be left on the stack as a result of going through the loop.

implements the FOR statement.

Controls the loop with information on the stack.

Stored are

- the address of the loop variable
- the beginvalue
- the endvalue
- a flag word

The FOR instruction implements a jump-instruction, the jump is taken as long as the end-condition is not reached.

The stack is cleaned up if the end-condition is reached.

The flag word has the following meaning :

Bit 0 = 0 : the loop variable is a one byte location

Bit 0 = 1 : the loop variable is a two byte location

Bit 2 = 0 : to loop

Bit 2 = 1 : downto loop

Bit 8 = 0 : loop not initialized

Bit 8 = 1 : loop initialized

Stack layout :

```
start                                     end
!                                     !<- stack if end
+-----+
! address of loop variable !
+-----+
! beginvalue               !
+-----+
! endvalue                 !
+-----+
! flag word                !
+-----+
stack ->!                       !<- stack if not
end
```

92 XX XX, LEQM – Less than or equal for arrays and records.

See 95 – LESM for description

93, LES2 – less than test for 2-byte data items

The LES2 instruction is similar to the LQ2 instruction except it does a “less than“ test.

compares for less or equal

condition tested : first value < second value

result : boolean 1 if less or equal else boolean 0

```

Stack layout :
start                                     end

!                                     !
+-----+
! first value / boolean                !
+-----+
! second value                          ! <- stack
+-----+
stack ->!                               !

```

94, LEQ8 – less or equals test for sets

this is the Is contained in test for sets. It pulls 2 8-byte sets off the stack and compares them for at least the common bits.

checks if all elements of first set are present
in second set. Leaves boolean result on the stack.

```

Stack layout
begin                                     end

!                                     !
+-----+
! first set / result boolean!
+-----+
! second set                          ! <- stack
+-----+
stack ->!                               !

```

95 XX XX, LESM - Less than test for arrays and records

The LESM instruction pulls two addresses off the stack and compares XXXXth bytes. If the first area is less in value than the area pointed by the second address then a 1 (true) is pushed onto the stack, otherwise a 0 (false) is pushed.

compares for equality of arrays or records
condition tested : first value < second value
result : boolean 1 if equal else boolean 0
size of structure follows opcode

```

Stack layout :
start                                     end

!                                     !
+-----+
! first address / boolean                !
+-----+
! second address                          ! <--stack
+-----+
stack ->!                               !

```

96, EQU2 – Equal test for 2-byte data items

The EQU2 instruction pulls two 2-byte data items off the stack and pushes a Boolean item onto the stack. The Boolean item will be a 1 (true) when the items compared are equal and a 0 (false) otherwise.

compares for equality

condition tested : first value = second value

result : boolean 1 if equal else boolean 0

```
Stack layout :
start                                     end

!                                     !
+-----+
! first value / boolean                !
+-----+
! second value                         ! <- stack
+-----+
stack ->!                               !
```

97, GEQ8 – Greater than or equal test for sets

This is the Contains test for sets. See 94 instruction. If true it sends a false on the stack and visa versa due to the generation of a AC – Not instruction following this one by the Pascal-M compiler.

checks if all elements of second set are present

in first set. Leaves boolean result on the stack.

```
Stack layout
begin                                     end

!                                     !
+-----+
! first set / result boolean!
+-----+
! second set                         ! <- stack
+-----+
stack ->!                               !
```

98 XX XX, EQU M – Equal test for arrays and records

The EQU M instruction pulls two addresses off the stack and compares XXXX bytes. If the two areas are equal a 1 (true) is pushed onto the stack, otherwise a 0 (false) is pushed on the stack.

compares for equality of arrays or records

condition tested : first value = second value

result : boolean 1 if equal else boolean 0

size of structure follows opcode

```
Stack layout :
start                                     end
```

```

!                                     !
+-----+
! first address / boolean           !
+-----+
! second address                   !<--stack
+-----+
stack ->!                             !

```

99, EQU8 – Equal test for sets

The EQU8 instruction pulls two byte sets off the stack and compares them. If they are equal a 1 (true) is pushed onto the stack, otherwise a 0 (false) is pushed on the stack.

compares for equality of sets

condition tested : first value = second value

result : boolean 1 if equal else boolean 0

```

Stack layout :
start                                     end

!                                     !
+-----+
! first set value / boolean           !
+-----+
! second set value                   !<--stack
+-----+
stack ->!                             !

```

9A, IND1 – indirectly load 1-byte data item

The IND1 instruction pulls an address off the stack and pushes the byte at that location onto the stack after expanding it to two bytes(as with the LOD1 instruction).

loads one byte value on stack from address on stack

```

Stack layout :
start                                     end

!                                     !
+-----+
! address/value in low byte           !
+-----+
stack ->!                             !<- stack

```

9B, IND2 – Indirectly load 2-byte data item

The IND2 instruction pulls an address off the stack and pushes the byte at the location and the following one onto the stack.

loads two byte value on stack from address on stack

```

Stack layout :
start                                     end

!                                     !
+-----+

```

```

                ! address/value                !
                +-----+
stack ->!                                ! <- stack

```

9C, IND8 – indirectly load 8-byte data item

The IND8 instruction pulls an address off the stack and pushes that byte and the following 7 onto the stack.

(* loads two byte value on stack from address on stack

```

Stack layout :
start                                end

                !                            !
                +-----+
                ! address/ set value        !
                +-----+
stack ->! set value                        !
                +-----+
                ! set value                 !
                +-----+
                ! set value                 !
                +-----+
                !                            !<- stack

```

9D, STO1 – Indirectly store 1-byte data item

The STO1 instruction pulls 2-bytes of data off the stack and an address. The least significant byte of data is stored at the address that was pulled off the stack.

indirectly stores 1 byte from stack at address on stack

```

Stack layout :
begin                                end

                !                            ! <- stack
                +-----+
                !           address           !
                +-----+
                !           value low byte    !
                +-----+
stack ->!                                !

```

9E, STO2 Indirectly store 2-byte data item

The STO2 instruction pulls 2-bytes data off the stack and an address. The 2 bytes of data are stored into memory starting at the address that was pulled off the stack.

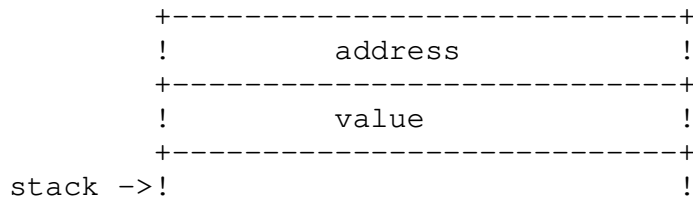
indirectly stores 2 bytes from stack at address on stack

```

Stack layout :
begin                                end

                !                            ! <- stack

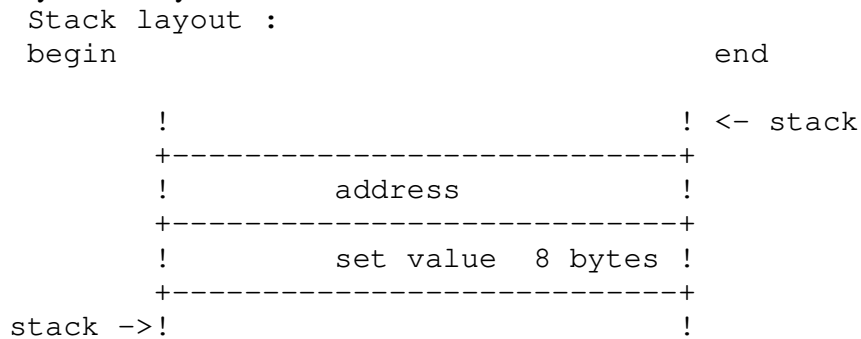
```



9F, STO8 – Indirectly store 8-byte data item.

The STO8 instruction pulls 8-bytes of data off the stack and an address. The 8 bytes of data are stored into memory starting at the address that was pulled off the stack.

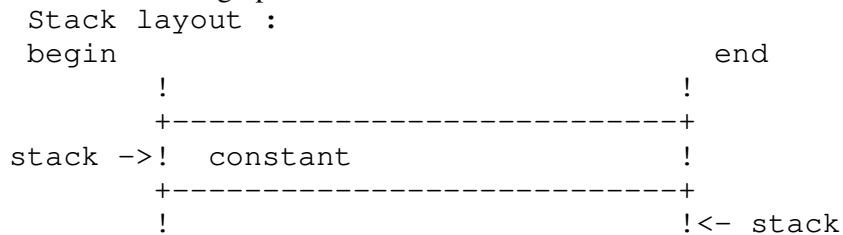
indirectly stores 2 bytes from stack at address on stack



A0 XX XX, LDC – Load 2-byte constant

The LDC instruction pushes the 2-byte constant, XXXX onto the stack.

loads constant following opcode on the stack



A1, RETP – Return from procedure (or function)

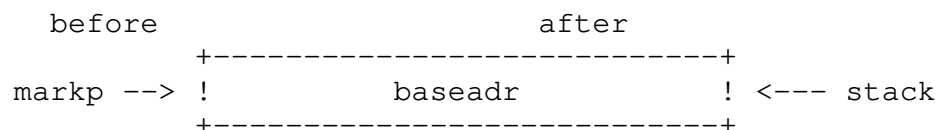
The RETP instruction returns from the current procedure, cleaning up the stack appropriately.

Returns from procedure/function, cleans up

stack by removing local variables and

linkframe

Situation before :



```

!           mpsave           !
+-----+
!           pcsave           !
+-----+
!           local variables   !
!                               !
+-----+
stack --> !

```

A2, ADI – Add integer

The DVI instruction pulls two 2-byte integers off the stack adds them together and pushes the result onto the stack.

adds two integers on stack, leaves integer
result on stack.

evaluates expression :

result := first value / second value

```

Stack layout :
start                                     end
!                                     !
+-----+
! first value / result                 !
+-----+
! second value                         ! <- stack
+-----+
stack ->!

```

A3, MAND – Boolean AND

The MAND instruction pulls two Boolean items off the stack, logical AND-s them together and pushes the result into the stack.

Performs AND on two booleans on the stack,
leaves boolean result on stack.

Booleans placed on stack as two bytes,

low byte contains 0 if false.

evaluates expression :

result := first boolean AND second boolean.

(true is 1 if first and second true)

```

Stack layout :
start                                     end
!                                     !
+-----+
! first boolean/ result               !
+-----+
! second boolean                      ! <- stack
+-----+

```



```
stack ->! !
```

A4, DIF – Set difference

The DIF instruction pulls two 8-byte sets off the stack, finds the difference between the two by logically AND-ing and Exclusive-OR-ing each byte and pushes the result onto the stack.

gives the difference of two sets on the stack,
leaves result on the stack. Collects in result
all elements present in first set and not in second set.

```
Stack layout
begin                               end
!                                   !
+-----+
! first set / result set          !
+-----+
! second set                       ! <- stack
+-----+
stack ->!
```

A5, DVI -- Divide integer

The DVI instruction pulls two 2-byte integers off the stack and divides the top of the stack into the second and pushes the result onto the stack.

divides two integers on stack, leaves integer
result on stack. Signed division.
evaluates expression :
result := first value / second value

```
Stack layout :
start                               end
!                                   !
+-----+
! first value / result            !
+-----+
! second value                     ! <- stack
+-----+
stack ->!
```

A6, INN – Test if element in set

The INN instruction pulls an 8-byte set off the stack and a 2-byte integer. If the set element indicated by the integer is in the set, a 1 (true) will be pushed on the stack, otherwise a 0 (false) will be pushed on the stack.

The is member of test for sets is implemented
with this instruction. Leaves boolean true
on stack if member in set.

```

Stack layout :
start                                     end
      !                                     !
      +-----+
      ! number of member/ result !
      +-----+
      ! set value                   ! <- stack
      +-----+
stack ->!                                 !

```

A7, INT – Set interaction

The INT instruction pulls two 8-byte sets off the stack and finds the intersection by logically AND-ing each byte together. The intersection is then pushed onto the stack.

gives the intersection of two sets on the stack,
 leaves result on the stack. Collects in result
 all elements present in first and in second set.

```

Stack layout
begin                                     end
      !                                     !
      +-----+
      ! first set / result set      !
      +-----+
      ! second set                   ! <- stack
      +-----+
stack ->!                                 !

```

A8, IOR – Inclusive OR

The IOR instruction pulls two Boolean items off the stack, logically OR-s them together and pushes the result on the stack.

Performs OR on two booleans on the stack,
 leaves boolean result on stack.
 Booleans placed on stack as two bytes,
 low byte contains 0 if false.
 evaluates expression :
 result := first boolean OR second boolean.
 (true is 1 if first or second true)

```

Stack layout :
start                                     end
      !                                     !
      +-----+
      ! first boolean/ result      !
      +-----+
      ! second boolean             ! <- stack
      +-----+
stack ->!                                 !

```

A9, MMOD – Modulus function

The MMOD instruction pulls two integers off the stack, and finds the modulus by taking the remainder of dividing the top of the stack into the second. The resulting remainder is then pushed onto the stack.

modulus of two integers on stack, leaves integer

result on stack. Signed division.

evaluates expression :

result := first value mod second value

```
Stack layout :
start                end
      !                !
      +-----+
      ! first value / result !
      +-----+
      ! second value         ! <- stack
      +-----+
stack ->!                !
```

AA, MPI – Multiply integer

The MPI instruction pulls two integers off the stack, multiplies them and pushes the result onto the stack.

multiplies two integers on stack, leaves integer

result on stack. Signed division.

evaluates expression :

result := first value * second value

```
Stack layout :
start                end
      !                !
      +-----+
      ! first value / result !
      +-----+
      ! second value         ! <- stack
      +-----+
stack ->!                !
```

AB, NGI – Negate integer

The NGI instruction pulls the integer off the stack, changes its sign and pushes it back onto the stack.

changes sign of integer on stack

evaluates expression :

result := - result

```
Stack layout :
start                end
      !                !
```

```

+-----+
!      result      !
+-----+
stack ->!                               !<- stack

```

AC, MNOT – negate Boolean

The MNOT instruction pulls one Boolean item of the stack, logically complements it and pushes it back onto the stack.

* complements boolean value on stack

evaluates expression :

result := not result

```

Stack layout :
start                               end
!                                   !
+-----+
!      result      !
+-----+
stack ->!                               !<- stack

```

AD, SBI – Subtract Integer

The DVI instruction pulls two 2-byte integers off the stack and subtracts the top of the stack from the next and pushes the difference onto the stack.

subtracts two integers on stack, leaves integer

result on stack.

evaluates expression :

result := first value - second value

```

Stack layout :
start                               end
!                                   !
+-----+
! first value / result !
+-----+
! second value         ! <- stack
+-----+
stack ->!                               !

```

AE, SGS – Generate singleton set

The SGS instruction pulls an integer off the stack, generates a SET with that integer as its only element and pushes the generated set onto the stack.

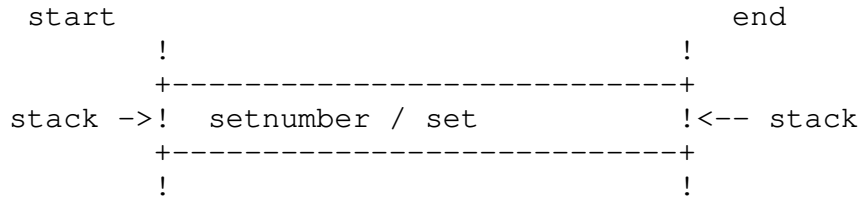
Generate singleton set on stack.

On stack number is present as integer

of set member to be set, other set members

not present.

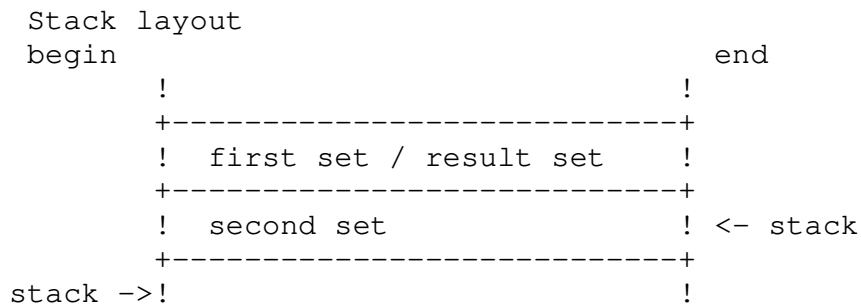
Stack layout :



AF, UNI – Set union

The UNI instruction pulls two sets of the stack, logically O-s them together and pushes the resulting set onto the stack.

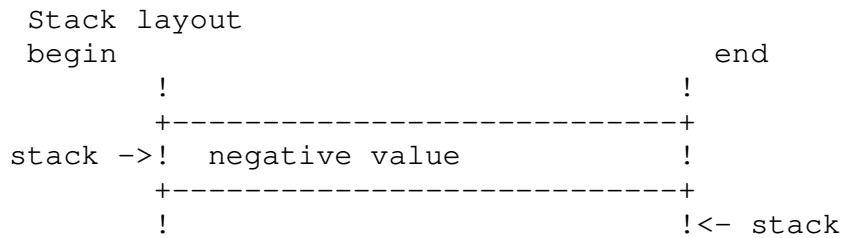
gives the union of two sets on the stack,
 leaves result on the stack. Collects in result
 all elements present in either input set.



B0 X X, LNC – Load negative constant

The LNC instruction loads the negative of XXXX onto the stack.

loads negative value on stack with value after opcode



B1 X X, FJP – False jump

The FJP instruction pulls one Boolean off the stack. If the item is zero (false), control will transfer to the XXXXth byte of the current procedure, otherwise control passes to the next instruction in sequence.

false jump, taken if value on stack is false, zero
 jump performed with UJP



```

                ! boolean                                ! <- stack
                +-----+
stack ->!
                !

```

B2 X X, UJP – unconditional jump

The UJP instruction always transfers control to the XXXXth byte of the procedure,

B3 X X, DEC – Decrement

The INC instruction pulls one 2-byte integer off the stack, detracts XX from it and pushes the result back onto the stack.

decrement value on stack with value after opcode and
leave result in same stack position

```

Stack layout
begin                                end
                !                                !
                +-----+
                ! value /value decremented !
                +-----+
stack ->!                                !<- stack

```

B4 X X, INC – Increment

The INC instruction pulls one 2-byte integer off the stack, adds XX to it and pushes the result back onto the stack.

increment value on stack with value after opcode and
leave result in same stack position

```

Stack layout
begin                                end
                !                                !
                +-----+
                ! value /value incremented !
                +-----+
stack ->!                                !<- stack

```

B5 X X, ENT – Enter block

The ENT instruction is always generated as the first instruction of the procedure. It reserves stack space for all variables local to the procedure.

Reserve stack space for all variables local
to procedure.

PC (high), PC + 1 (low) contain nr of bytes to reserve.

```

Stack layout
begin                                end
                !                                !

```

```

+-----+
stack ->!  room for local variables  !
      .
      .
      !                               !
+-----+
!                                     !<- stack

```

B6 -----, CAS – Case statement procedure

The CAS instruction does most of the processing for the case statement. It occupies a variable amount of memory because the instruction includes the complete jumtable for the case statement.

processes case - statement.

Following the opcode the complete jump-table is present including the otherwise part and the beginning and endvalue of the label:

```

Stack layout
begin                                     end
      !                                     !<- stack
+-----+
stack ->!  current label value          !
+-----+
      !                                     !

```

jumtable (from low to high addresses):

```

+-----+
!  start value label in list  !
+-----+
!  end   value label in list  !
+-----+
!  address of otherwise part  !      (if not
used zero)
+-----+
!  address of first label     !
!  routine                    !
+-----+
!                               !
      .
ordered next label routine addresses
      (if not used zero)
      .
      !                               !
+-----+
!  address of last label     !
!  routine                    !
+-----+

```

B7 X X, MOV

C4 X X X X MOVMB – Move storage

The MOV instruction is used for moving arrays and records around in memory. The address of the sending field is pulled off the stack and the address of the receiving field is pulled off next, XXXX bytes are then transferred from the sending to the receiving field.

moves structures in memory

Size to move follows opcode as word.

If MVB then blanks to pad follows size as word.

Source and destination address are on the stack.

```
Stack layout :
start                                     end
      !                                     !
      +-----+
      ! destination address             ! <- stack
      +-----+
      ! source address                 !
      +-----+
stack ->!                                 !
```

B8, DEC1 – decrement by 1

The DEC1 instruction pulls one 2-byte integer off the stack, subtracts 1 from it and pushes the result back onto the stack.

decrement first value on stack with one and
leave result in same stack position

```
Stack layout
begin                                     end
      !                                     !
      +-----+
      ! value /value - 1                 !
      +-----+
stack ->!                                 !<- stack
```

B9, INC1 – increment by 1

The INC1 instruction pulls one 2-byte integer off the stack, adds 1 to it and pushes the result back onto the stack.

increment first value on stack with one and
leave result in same stack position

```
Stack layout
begin                                     end
      !                                     !
      +-----+
      ! value /value + 1                 !
      +-----+
```



```
stack ->!                                     !<- stack
```

BA X X X X X X X X, LDCS – Load set constant

The LDCS instruction loads the 8-byte set, X X X X X X X X, onto the stack.

loads constant set following opcode on the stack.

```
Stack layout
begin                                     end
      !                                     !
      +-----+
stack ->!  set value                       !
      +-----+
      !                                     !<- stack
```

BB X X, CAP – Call assembly procedure

The CAP instruction is used to call an assembly-language routine previously loaded into a specified address. The actual action of this instruction may be considered to be system-dependent. Local parameters are stored at the stack with LDCI

```
Stack layout
begin                                     end
      !                                     ! <- stack
      +-----+
      ->!  External address                 !
      +-----+
Stack ->!                                     !
```

See the chapter CAP for more information.

BC X ---===, LCA – Load constant address

The LCA instruction pushes the address of the string starting 2 bytes after the instruction code onto the stack. The length of the string is specified by XX. For this reason, all addresses used within the Pascal-M system, whether pointer or array bases, must refer to the actual hardware address of the item.

Load address of string following opcode on stack
pc incremented up to opcode following string

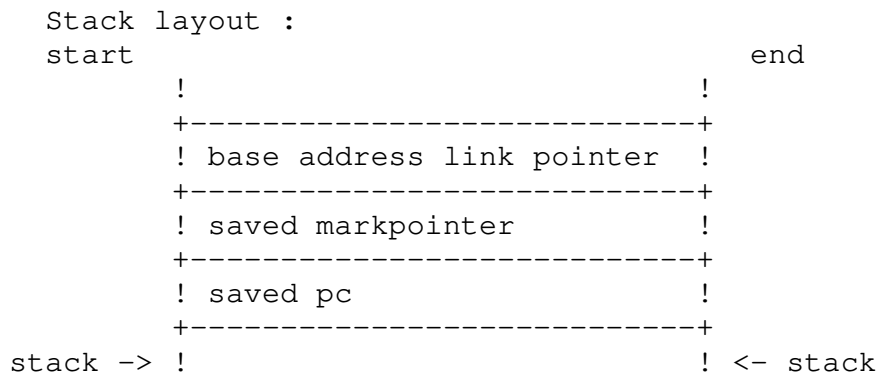
```
Stack layout
begin                                     end
      !                                     !
      +-----+
stack ->!  address of string                 !
      +-----+
      !                                     !<- stack
```

BD X, CSP – Call standard procedure X

The CSP instruction is used for calling standard procedures that exist within the interpreter itself. XX is a numeric index that determines which procedure is to be executed. Since a large number of possible instruction codes have not been used, it may be desirable at some point to eliminate this instruction and incorporate all standard procedures as instructions.

BE XX, CUP1 –simple call user procedure

The CUP1 instruction is used to call procedures whenever the parameter list of the call does itself not include a function call. XX is a numeric index that determines the procedure to be called. The compiler assigns a number to each procedure at compile time. The number is then associated with an address at load time.



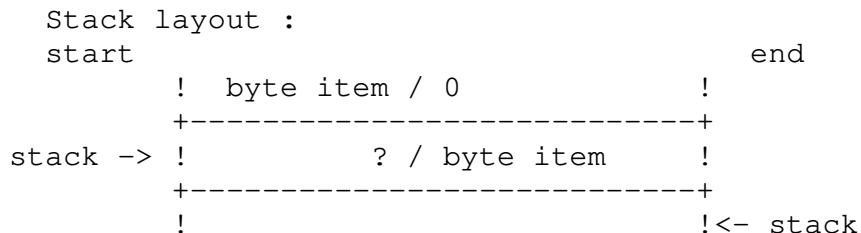
BF X, CUP2 – Complex Call user procedure

The CUP2 instruction is used to call procedures whenever the parameter list of the call does include a call to a function. One 2-byte integer is pulled off the stack, which indicates how many bytes of parameters are being passed. In all other respects, CUP2 functions the same as CUP1.

C0, FIX21 – cleanup stack after call to single-byte function

The FIX21 instruction pushes a single byte of zero onto the stack. This operation is used to standardize the return value of single-byte-valued functions (such as Boolean functions). This action then makes the single byte value occupy two bytes on the stack as is usual.

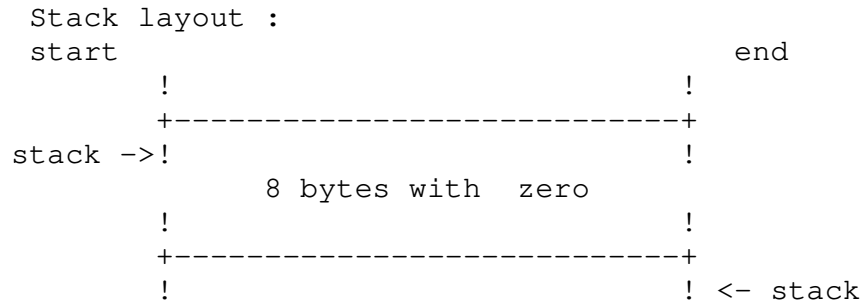
Transfers one byte item on stack into two byte item,
byte item moved to low byte of two byte item,
high byte filled with zero



C1, LNS – Load null set

The LNS instruction pushes 8 bytes of zero onto the stack for use as a null set.

Load null set on stack : 8 bytes of zero

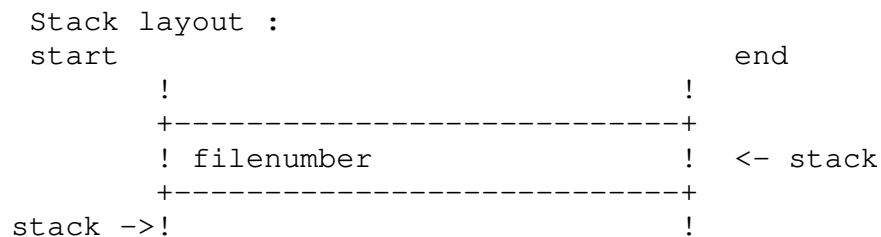


Additions for V1.1 and up:

C2, SFA – Set file address

The SFA instruction pulls one 2-byte address of the stack and places it in a location internal to the interpreter for use by all subsequent I/O operations. The address pulled off the stack is the address of a pointer to the file buffer.

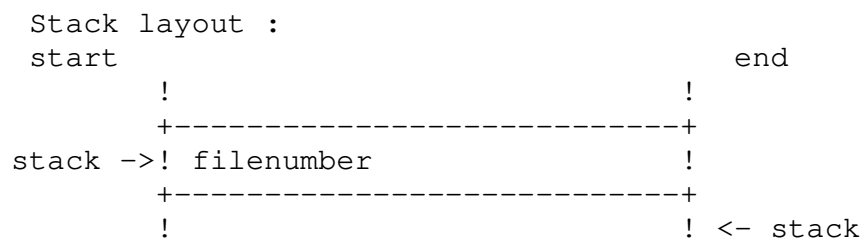
Set file address for all file operations to follow
filename in least sign byte on stack



C3, GFA – Get file address

The GFA instruction pushes the value of the internal file address on the stack.

Get file address for all file operations to follow
filename in least sign byte on stack



Standard procedures

00, WRI – Write integer

The WRI procedure writes one integer onto output.

writes integer on stack in specified field width.

```
Stack layout
begin                               end
!                                   !
+-----+
! char code                          !<- stack
+-----+
! field width                         !
+-----+
stack ->!                             !
```

01, WRC – Write character to output

The WRC procedure writes one character to output.

writes character on stack in specified field width.

```
Stack layout
begin                               end
!                                   !
+-----+
! char code                          ! <- stack
+-----+
! field width                         !
+-----+
stack ->!                             !
```

02, WRS – Write string to output

The WRS procedure writes a packed array of characters onto output.

prints string to output,

```
Stack layout
begin                               end
!                                   !
+-----+
! address                            !<- stack
! of string                          !
+-----+
! specified                          !
! field width                        !
+-----+
! actual                             !
```

```

! field width      !
+-----+
stack ->!          !

```

03, RDI – Read integer

The RDI procedure reads an integer from input

reads integer

```

if terminal then
  if CR entered then eofline becomes true
    and a space returned
  if CTRL-Z entered then endoffile and eofline
    become true
  at least the value zero is returned,
  also for illegal characters

```

```

Stack layout :
start                                     end
!                                       !
+-----+
! address for integer                   !<- stack
+-----+
stack ->!                               !

```

04, RLN – Read to end of line on input

The RLN reads skips to end of line on input.

05, RDC – Read character from input

The RDC procedure reads a single character from input.

reads character from input file

```

Stack layout :
start                                     end
!                                       !
+-----+
! address for char                       !<- stack
+-----+
stack ->!                               !

```

06, WLN – Write end of line

The WLN procedure writes an end of line

07, NEW – Allocate space on the heap

The NEW procedure pulls a 2-byte integer off the stack which specifies the size of the area to be allocated and a 2-byte address which specifies the pointer variable to be set. Space is then allocated on the heap and the pointer variable set to the address of that storage.

(* Reserves space in the heap for variable
 pointer address and size to reserve on the stack
 heap pointer updated

```

Stack layout :
start                                     end
      !                                     !
      +-----+
      ! address of pointer                 !<- stack
      +-----+
      ! size to reserve on heap           !
      +-----+
stack ->!                                     !

```

08, EOF – End of file test

The EOF procedure pushes a Boolean item on the stack which is true if an end of file condition exists.

enters boolean end of file on stack

```

Stack layout :
start                                     end
      !                                     !
      +-----+
stack ->! boolean (1 if endoffile)         !
      +-----+
      !                                     !<- stack

```

09, RST – Reset heap pointer

The RST procedure sets the heap pointer to the value pulled off the stack.

Restores heap pointer to value in pointer

address of pointer on the stack

```

Stack layout :
start                                     end
      !                                     !
      +-----+
      ! address of pointer                 !<- stack
      +-----+
stack ->!                                     !

```

0A, ELN – Test for end of line

The ELN procedure pushes a Boolean item on the stack which is true if an end of line condition exists.

(* enters boolean end of line on stack

```

Stack layout :

```

```

start                                     end
!                                     !
+-----+
stack ->!  boolean (1 if endofline)  !
+-----+
!                                     !<- stack

```

OB, STP – Stop

The STP procedure halts execution.

OC, ODD – Test integer for odd.

The ODD procedure pulls a 2-byte integer off the stack and pushes a Boolean item onto the stack which is true if the integer is odd.

enters boolean true on stack if value on stack is odd

```

Stack layout :
start                                     end
!                                     !
+-----+
stack ->!  value/ result (1 if odd)  !<-stack
+-----+
!                                     !

```

OD : RSFRWW ;

OE: RSFRWW reset rewrite file

resets or rewrites file

filename as string on stack

```

Stack layout :
start                                     end
!                                     !
+-----+
! address of strng          !<-- stack
+-----+
! length of strng          !
+-----+
stack ->!                                     !

```

OF : STT status I/O

enters status of file on stack for current filename

```

Stack layout :
start                                     end
!                                     !
+-----+
stack ->!  value/ result (1 if odd)  !
+-----+

```

!

!<-stack

10 :CLS closefile

CAP Call Assembly Procedure

A way to call any system dependent and hardware dependent routine.

Syntax

```
PROCEDURE <name> {< parameter list>} ; extern = <address>
```

Where

- name is the internal name of the procedure in the Pascal[program
- address is the location of the routine in memory, to be supplied outside of the Pascal program. As Pascal-M is a 16 bit only compiler, this limits addressing space to 64K
-

At entry the routine is free to perform any action.

The optional parameter list allows to pass data and variables to the called routine. The routine is responsible for the interpreter stack handling, see the M-code instructions what to expect on the stack and how to return values.

On return the routine must remove the 2 bytes of the interpreter stack containing the called address.

Examples for V1 of the Pascal-M compiler KIM-1 interpreter.

```
                .org    $00C0    ; because the compiler knows this
                ;
proc1   lda      #$00    ; procedure ttyin ;extern = $00C0
                ;
                sta      setin
                jmp      procex
                nop
proc2   lda      #$01    ; procedure sethsr ;extern =
$00C8 ;
                sta      setin
                jmp      procex
                nop
proc3   lda      #$00    ; procedure ttyout ;extern =
$00D0 ;
                sta      setout
                jmp      procex
                nop
proc4   lda      #$01    ; procedure sethsp ;extern =
$0058
                sta      setout
procex  jsr      pull2
                jmp      loop
```

Implementation of CAP in KIM-1 6502 interpreter

```

;
; BB CAP
;
cap   ldx    #$00
      lda    (pc,x)
      sta    tmp212 + 1
      ldx    #pc
      jsr    incr           ; pc := pc + 1
      lda    (pc,x)
      sta    tmp212
      ldx    #pc
      jsr    incr
      jmp    (tmp212)      ; jump to new address
;
```

Compiler error messages

- 2 Syntax: identifier expected
- 3 Syntax: Program expected
- 4 Syntax: ")" expected
- 5 Syntax: ":" expected
- 6 Syntax: illegal symbol
- 7 Syntax: actual parameter list
- 8 Syntax: OF expected
- 9 Syntax: "(" expected
- 10 Syntax: type specification expected
- 11 Syntax: "[" expected
- 12 Syntax: "]" expected
- 13 Syntax: END expected
- 14 Syntax: ";" expected
- 15 Syntax: integer expected
- 16 Syntax: "-" expected
- 17 Syntax: BEGIN expected
- 18 Syntax: error in declaration part
- 19 Syntax: error in field list
- 20 Syntax: "," expected
- 21 Syntax: "*" expected

- 50 Syntax: "error in constant
- 51 Syntax: ":=" expected
- 52 Syntax: THEN expected
- 53 Syntax: UNTIL expected
- 54 Syntax: DO expected
- 55 Syntax: TO/DOWNTO expected
- 56 Syntax: IF expected
- 58 Syntax: ill-formed expression
- 59 Syntax: error in variable

- 101 Identifier declared twice
- 102 Low bound exceeds high-bound
- 103 Identifier is not a type identifier
- 104 Identifier not declared
- 105 Sign not allowed
- 106 Number expected
- 107 Incompatible subrange types
- 110 Tag type must be an ordinal type
- 111 Incompatible with tag type
- 113 Index type must be an ordinal type
- 115 Base type must be scalar or subrange
- 116 Error in type of procedure parameter
- 117 Unsatisfied forward reference
- 118 Forward reference type identifier
- 119 Forward declared : repetition par. list
- 120 Function result: scalar,subrange,pointer' ;

122 Forward declared: repetition result type' ;
123 Missing result type in function declar.
125 Error in type of standard function par.
126 Number of parameters disagrees with decl' ;
129 Incompatible operands
130 Expression is not of SET type
131 Test on equality allowed only
132 Inclusion not allowed in set comparisons' ;
134 Illegal type of operands
135 Boolean operands required
136 Set element must be scalar or subrange
137 Set element types not compatible
138 Type must be array
139 Index type is not compatible with decl.
140 Type must be record
141 Type must be pointer
142 Illegal parameter substitution
143 Illegal type of loop control variable
144 Illegal type of expression
145 Type conflict
147 Case label and case expression not comp.' ;
148 Subrange bounds must be scalar
149 Index type must not be an integer
150 Assignment to standard function illegal
152 No such field in this record
154 Actual parameter must be a variable
155 Control variable declared interm. level
156 Value already as a label in CASE
157 Too many cases in CASE statement
160 Previous declaration was not forward
161 Again forward declared
169 SET element not in range 0 .. 63
170 String constant must not exceed one line' ;
171 Integer constant exceeds range (32767)
172 Too many nested scopes of identifiers
173 Too many nested procedures/functions
174 Index expression out of bounds
175 Internal compiler error : standard funct' ;
176 Illegal character found
177 Error in type
178 Illegal reference to variable
179 Internal error : wrong size variable
180 Maximum number of files exceeded