THE VIRTUAL FORTH COMPUTER

A major shortcoming of my book Systems Guide to fig-FORTH was that it did not fully explain the functions of the low level FORTH codes. It is thus very difficult to convince the readers many of the inherent advantages of FORTH. I tried to explain some of the most important low level codes which are related to the inner interpreters in FORTH. Since it was very difficult to use the 6502 machine codes as published in the fig-FORTH Model and Installation Manual, I chose to describe these words in the PDP-11 codes, which are much more descriptive as to the functions of these low level words. To explain fully how the FORTH virtual computer operates, one really has to go through the entire nucleus and understand all the low level words. I thought the best way was to rewrite the Installation Manual in PDP-11 codes and put a 'Guide' on top of it.

SYSTEMS GUIDE TO THE FORTH NUCLEUS

I took John James' PDP-11 fig-FORTH and translated it into the form of the Installation Manual. In this chapter, I will go through the nucleus part of it and try to comment on all the low level words. I think this will be useful for those who are still looking for a map to travel through this maze of codes.

In order to follow me through this advanture, it is assumed that the reader has a basic understanding of the PDP-11 machine architecture and its instruction set, particularly the strange addressing modes in the handling of operands. One has to put up with the reverse Polish style of the assembly codes. What we will gain is the clarity in expressing the functions of FORTH words in terms of the PDP-11 codes.

PROGRAMMING MODEL OF THE FORTH VIRTUAL COMPUTER

The FORTH virtual computer consists of a bank of memory, a set of registers, and some I/O devices. The most crucial part is the registers and their functions. There are five registers:

SP The data stack pointer which manages the flow of data through the FORTH computer. Since all the computational functions occur on top of the stack, one might visualize the data stack as the ALU in the FORTH computer.

RP The return stack pointer which keeps track on the nesting and unnesting of high level words. It is identical to the stack in conventional computers for subroutine calls.

IP The interpreter pointer which always points to the next word to be executes by the inner interpreter NEXT.

W The current word pointer through which the inner interpreter jumps indirectly to the routine executing this word.

120 LIST

121 LIST

```
( COLD AND WARM ENTRY, USER PARAMETERS )
VOCABULARY NEWFORTH IMMEDIATE VARIABLE ORIGIN  ASSEMBLER
HERE ORIGIN !   NEWFORTH DEFINITIONS   ASSEMBLER
HERE JMP, ( COLD )  HERE JMP,    ( WORD ALIGNED TO WARM )
11 ,   0 ,    ( CPU AND REVISION PARAMETERS )
0 ,    ( TASK-10, TOPMOST WORD IN FORTH VOCABULARY )
10 ,    ( BACKSPACE CHARACTER )
0 ,    ( XUP, POINTER TO USER AREA )
0 ,    ( XS0, POINTER TO INITIAL TOP OF STACK )
0 ,    ( XR0, POINTER TO INITIAL TOP OF RETURN STACK )
0 ,    ( TIB, TERMINAL INPUT BUFFER )
37 ,   ( MAXIMUN NAME FIELD WIDTH )
1 ,    ( WARNING MODE, WITH DISC )
0 ,    ( XDP, FENCE TO PROTECT AGAINST 'FORGET' THE SYSTEM )
0 ,    ( XDP, INTIAL VALUE OF DP )
0 ,    ( XXVOC, POINTER TO INITIAL VOCABULARY LINK )   ;S
```

122 LIST

```
( START OF NUCLEUS, NEXT, LIT, EXECUTE    )
0 ,    ( DSKBUF, INITIALIZE 'FIRST'   )
0 ,    ( ENDBUF, INITIALIZE 'LIMIT'   )
0 ,   0 ,    ( AVAILABLE FOR USER )

FORTH DEFINITIONS     ( DEFINED 'NEXT,' AS A MACRO )

: NEXT,    IP )+ W MOV,    W @)+ JMP,    ;

NEWFORTH DEFINITIONS   ASSEMBLER
```

UP The user area pointer holding the offset address to the data table where all the system variables are kept.

THE CODE INTERPRETER--- NEXT,

NEXT, is an assembly macro which is appended to every low level word in the nucleus. It is equivalent to the RTS code in the conventional machine codes. What it does is to pick up the address in the cell pointed to by the interpreter pointer IP and execute it. Because FORTH codes are indirectly threaded through their code fields, NEXT, puts this code field address in current word pointer W and make an indirect jump to the code routine whose address is store in the code field. In standard PDP-11 assembly format, NEXT, appears as following:

```
NEXT:    MOV   (IP)+,W
         JMP   @(W)+
```

Meanwhile, IP is incremented to point to the next word to be executed and W is incremented to point to the parameter field of the word currently under execution.

LIT This word is compiled before an in-line literal so that the literal will be pushed on the data stack instead of being interpreted as an address. The literal pointed by IP is pushed on stack and IP is incremented

EXECUTE It executes the word whose code field address is on top of the data stack. This address is poped into W and an indirect jump does the execution.

THE BRANCHES

The FORTH computer uses two words to implement unconditional and conditional jumps. They are compiled into high level word to construct appropriate structures and are not to be used elsewhere or for any other purposes.

BRANCH The next cell following always has an offset to the address of the next word to be executed. Add this offset to IP and NEXT, will pick up the execution sequence at that point.

0BRANCH Do the branch only if top of data stack is zero. Otherwise, skip the offset cell and continue on the normal execution sequence.

123 LIST

124 LIST

```
( LIT, EXECUTE, BRANCH, 0BRANCH  )

CODE LIT   IP )+ S -) MOV,   NEXT,   C;

CODE EXECUTE   S )+ W MOV,   W @)+ JMP,   C;


CODE BRANCH   ( ADJUST IP BY IN-LINE LITERAL )
      IP () IP ADD,   NEXT,   C;

CODE 0BRANCH   ( IF TOS IS ZERO, BRANCH FROM LITERAL )
      S )+ TST,   EQ IF,   IP () IP ADD,
      ELSE,   IP )+ TST,   THEN,   NEXT,   C;

;S
```

125 LIST

```
( LOOP CONTROL, (LOOP  , (+LOOP   )

CODE (LOOP)  ( INCREMENT LOOP INDEX, LOOP UNTIL REACHING LIMIT )
      RP () INC,   RP () 2 RP I) CMP,
      LT IF,   IP () IP ADD,
      ELSE,   IP )+ RP )+ CMP,   RP )+ TST,
      THEN,   NEXT,   C;

CODE (+LOOP)   ( INCREMENT INDEX BY TOS, LOOP TO LIMIT )
      S () RP () ADD,   S )+ TST,
      LT IF,   2 RP I) RP () CMP,   ( NEGATIVE TOS )
      ELSE,   RP () 2 RP I) CMP,   ( POSITIVE TOS )
      THEN,
      LT IF,   IP () IP ADD,
      ELSE,   IP )+ RP )+ CMP,   RP )+ TST,   THEN, NEXT,   C;
;S
```

THE LOOPS

Controlled or finite loops end with one of two loop endings which returns the execution to the starting point of the loop or exits the loop, depending upon the index and limit values on the return stack left by DO. Before exiting the loop, return stack has to be restored.

(LOOP) Increment the loop index on top of the return stack. If the index is equal or greater than the limit under it, restore return stack and exit the loop. Otherwise return to DO, whose offset is in the next cell.

(+LOOP) Increment index by the amount on the data stack. It is similar to (LOOP), except it will have to take care the cases in which the increment is negative. In this case, reverse index and limit before comparison.

(DO) Move the limit and index values from the data stack to the return stack, thus starting the DO-LOOP .

I Copy the current loop index on top of the return stack to the top of the data stack, to be used by other words inside the DO-LOOP.

DIGIT Given an Ascii code and the current base on the data stack, convert the code to its corresponding numeric value according to the current base. If the conversion is successful, leave the value and a true flag on the stack. Otherwise, leave only a false flag.
This is the basic numeric input routine which convert Ascii codes to numbers. The fact that BASE is used in this routine gives FORTH the capability to switch from one number system to the other, which can be matched by few computer systems, mainframe or not.

DICTIONARY SEARCH

(FIND) This is the basic routine to do dictionary search. Given the address of a string in memory and the name field address of a word in the dictionary, this (FIND) will search through the dictionary linked to the given word for the instruction whose name matches the given string.
The strategy used here is to compare the first two bytes, the length bytes and first character, as a number. If this comparison failed, the search goes on to the next word in the linked chain. If the first two bytes match, the strings are compared to their ends.
If a word is found, its parameter field address, its length and a true flag are on the stack. Otherwise, only a false flag is left on the stack.

```
  126 LIST

( (DO-    )

CODE (DO)    ( MOVE TWO STACK ITEMS TO RETURN STACK )
      S )+ R0 MOV,   S )+ RP -) MOV,   R0 RP -) MOV,
      NEXT,   C;

CODE I    ( COPY CURRENT LOOP INDEX TO STACK )
      RP () S -) MOV,   NEXT,   C;

;S
```

```
  127 LIST

( DIGIT  )
CODE DIGIT    ( CONVERT ASCII CHAR-2, WITH BASE-1 )
              ( IF OK RETURN DIGIT-2, TRUE-1; OTHERWISE FALSE-1 )
      60 # 2 S I) SUB,   2 S I) 11 # CMP,  ( IF > 9, BRANCH )
      GT IF,   7 # 2 S I) SUB,   2 S I) 12 # CMP,
          LT IF,   HERE ( 2$ )   S )+ TST,   S () CLR,   NEXT,
          THEN,
      THEN, ( 1$ )   2 S I) TST,
          GE IF,   2 S I) S () CMP,
              LT IF,   1 # S () MOV,   NEXT,   ( VALID DIGIT )
              THEN,
          THEN,
      ( 2$ ) JMP,   ( ERROR RETURN )   C;

;S
```

```
  128 LIST

( FIND FOR VARIABLE LENGTH NAMES )
CODE (FIND)   ( HERE NFA --- PFA LENGTH TRUE; ELSE FALSE )
      S )+ R0 MOV,  S )+ R1 MOV,  R5 RP -) MOV,  R4 RP -) MOV,
      R3 RP -) MOV, RP -) CLR, R1 () R2 MOV,  100200 # R2 BIC,
BEGIN, ( FCOMP )   R0 () R3 MOV,  100300 # R3 BIC,   R2 R3 CMP,
NE WHILE,   BEGIN, HERE ( XMATCH )  R0 )+ TST,   MI UNTIL,
  R0 () TST,   EQ IF, ( FAILED)   RP )+ TST,   RP )+ R3 MOV,
      RP )+ R4 MOV,   RP )+ R5 MOV,   S -) CLR,   NEXT,   THEN,
R0 () R0 MOV,   REPEAT,
( NOFAST )   R0 () RP () MOV,   R1 R5 MOV,
BEGIN,  100000 # R0 )+ BIT,  EQ WHILE, ( MLOOP )  R5 )+ TST,
  R5 () R4 MOV,  R0 () R3 MOV,  100000 # R3 BIC,  R3 R4 CMP,
( XMATCH ) BNE,   REPEAT,
( FOUND )  RP )+ R2 MOV,  RP )+ R3 MOV,  RP )+ R4 MOV,
RP )+ R5 MOV,  4 # R0 ADD,  R0 S -) MOV,  177400 # R2 BIC,
R2 S -) MOV,  1 # S -) MOV,  NEXT,   C;   ;S
```

THE PARSING ROUTINE --- ENCLOSE

ENCLOSE This is the tool used by the text interpreter to isolate words from the input stream of characters. The input on data stack are the address of the character stream and the Ascii character serving as the delimiter. It skips the leading delimiters in the stream, leaves the address of the stream, the offset to the first non-delimiting character, the offset to the end of the found string before the trailing delimiters, and the offset to the first trailing delimiter. A word is thus found in the input stream.

Ascii NUL is the absolute delimiter. The offsets returned by ENCLOSE will never pass a NUL character.

TERMINAL I/O

EMIT Print an Ascii character to the console CRT terminal.

KEY Wait until a key is pressed on the console keyboard. Return the corresponding Ascii code on the stack.

?TERMINAL Return a false flag on the stack if no key was pressed on the keyboard. Otherwise, returns a true flag, which is the Ascii code in this particular implementation.

CR Output a carriage return and a line feed to terminal.

BLOCK MOVE --- CMOVE

CMOVE This word copies a range or memory, byte by byte, to another memory area. The starting address, the distination address and the byte count are given on the data stack.

If the byte count is zero, nothing will be copied.

```
  129 LIST

( ENCLOSE  )
CODE ENCLOSE   ( ADDR DELIM --- ADDR OFFSET END NEXT )
      S () R0 MOV,   2 S I) R1 MOV,   4 # S SUB,
      BEGIN, ( ENC1 )   R1 )+ R0 CMPB,   NE UNTIL,
      1 # R1 SUB,   R1 4 S I) MOV,
      HERE ( ENC2 )   R1 () TSTB,   NE IF,   ( NOT NULL )
          R1 )+ R0 CMPB,   BNE, ( TO ENC2 )
          R1 S () MOV,   1 # R1 SUB,
      ELSE, ( ENC4, NULL CASE )   R1 S () MOV,   R1 4 S I) CMP,
          EQ IF,   1 # R1 ADD,   THEN,
      THEN, ( ENC3 )   R1 2 S I) MOV,   6 S I) R1 MOV,
      R1 S () SUB,   R1 2 S I) SUB,   R1 4 S I) SUB,
      NEXT,   C;
;S



  130 LIST

( TERMINAL I/O   EMIT, KEY,  ?TERMINAL, CR   )
CODE EMIT   ( PRINT ASCII VALUE ON TOS, INCREMENT OUT  )
      42 U I) INC,   BEGIN,   177564 @# TST,   NE UNTIL,
      S )+ 177566 @# MOV,   NEXT,   C;
CODE KEY   ( ACCEPT ONE CHAR FROM TERMINAL TO TOS )
      BEGIN,   177560 @# TSTB,   NE UNTIL,   177560 @# CLR,
      177562 @# R1 MOVB,   177600 # R1 BIC,   177 # R1 CMP,
      EQ IF,   10 # R1 MOV,   THEN,   R1 S -) MOV,   NEXT,   C;
CODE ?TERMINAL   ( 'BREAK' LEAVES 1 ON STACK; OTHERWISE 0 )
      177560 @# TSTB,   EQ IF,   S -) CLR,
      ELSE,   177562 @# S -) MOV,   THEN,
      177560 @# CLR,   NEXT,   C;
CODE CR   ( OUTPUT CR/LF TO TERMINAL )
      BEGIN,   177564 @# TST,   NE UNTIL,   15 # 177566 @# MOV,
      BEGIN,   177564 @# TST,   NE UNTIL,   12 # 177566 @# MOV,
      NEXT,   C;   ;S


  131 LIST

( CMOVE  )
CODE CMOVE   ( FROM-3, TO-2, COUNT-1 --- )
      S () TST,   NE IF,
          2 S I) R0 MOV,   4 S I) R1 MOV,
          BEGIN,   R1 )+ R0 )+ MOVB,   S () DEC,   EQ UNTIL,
      THEN,   6 # S ADD,   NEXT,   C;

;S
```

UNSIGNED MULTIPLICATION

UMULT — This multiplication subroutine takes the top two unsigned numbers on the data stack, multiply them and returns the 32-bit unsigned double product on the stack.
The reason why this routine is coded as a subroutine is that one might easily substitute it with a hardware multiplication instruction if the PDP-11 CPU supports such an instruction.

U* — This is the actual FORTH word for unsigned multiplication. It calls UMULT as a subroutine.

Multiplications and divisions are fundamental to any high level computer. They must be implemented in the nucleus.

UNSIGNED DIVISION

UDIV — An unsigned 32-bit divisor and a 16-bit unsigned divider are on the stack. This subroutine returns the 16-bit quotient on the top of stack and the remainder under it.

U/ — The actual FORTH unsigned 32-bit division word. All the other variants of dividing words are derived from U/.

LOGICAL OPERATORS

AND — It is strange that PDP-11 instruction set does not include the AND code. It is synthesized from a complement and a bit clear instructions.

OR — Bitwise OR of the top two numbers on the top of the data stack.

XOR — Exclusive OR code is provided in the Extended Instruction Set (EIS) of PDP-11. Covering the low end PDP machines which may not have this provision, the XOR is again synthesized.

```
  132 LIST

( U* , UNSIGNED MULTIPLICATION FOR 16 BIT NUMBERS )
CODE UMULT
      S )+ R2 MOV,   20 # RP -) MOV,
      R0 CLR,   R1 CLR,   ( ACCUMULATOR )
      BEGIN, ( 2$ )   R1 ROL,   R0 ROL,   R2 ROL,
          CS IF,   S () R1 ADD,   R0 ADC,   THEN,
      RP () DEC,   EQ UNTIL,
      R1 S () MOV,   R0 S -) MOV,   RP )+ TST,   PC RTS,
      C;
CODE U*   PC ' UMULT JSR,   NEXT,   C;

;S
```

```
  133 LIST

( U/ , UNSIGNED DIVIDE FOR 31 BIT NUMBER )
CODE UDIV
      S )+ R2 MOV, ( DIVISOR )   S )+ R0 MOV,
      S )+ R1 MOV,   20 # S -) MOV, ( LOOP COUNT )
      BEGIN, ( 1$ )   R1 ASL,   R0 ROL,
          NE IF,   R2 R0 SUB,   R1 INC,
              CS IF,   R2 R0 ADD,   R1 DEC,   THEN,
          THEN,
      S () DEC,   EQ UNTIL,
      S )+ TST,   R0 S -) MOV,   R1 S -) MOV,   PC RTS,
      C;
CODE U/   PC ' UDIV JSR,   NEXT,   C;

;S
```

```
  134 LIST

( LOGICALS   AND, OR, XOR  )

CODE AND   ( LOGICAL BITWISE AND OF TOP TWO ITEMS )
      S () COM,   S )+ S () BIC,   NEXT,   C;

CODE OR    ( BITWISE OR OF TOP TWO ITEMS )
      S )+ S () BIS,   NEXT,   C;

CODE XOR   ( EXCLUSIVE-OR OF TOP TWO ITEMS )
      S () RP -) MOV,   2 S I) RP () BIC,   S )+ S () BIC,
      RP )+ S () BIS,   NEXT,   C;

;S
```

MISCELLANIOUS STACK OPERATORS

SP@ Fetch the current data stack pointer to the top of the data stack. For safty, the stack pointer is fetched through a scratch register.

SP! Load the initial value into the data stack pointer. The value is stored as the third item in user area.

RP! Initialize the return stack pointer from user area.

;S High level return. It must be executed as the last word in any colon definition. It unnests the high level word by one level, undoing what DOCOL accomplished.
It pops an address from the return stack back into the interpretive pointer IP and calls NEXT, to execute it. This restore the execution sequence left by the execution of a high level colon word.

RETURN STACK OPERATORS

LEAVE It simply copies the index value on top of the return stack to the limit just below it. This action ensures that the next time (LOOP) executes, the loop will be terminated unconditionally.

>R Pop the top of data stack and push it onto the return stack.

R> Pop the top of the return stack and push it onto the data stack.

R, R@ Copy the top of the return stack and push it onto the data stack. Functionally the same as I, but may differ in other implementations.

NUMERIC TESTS

0= Return a true flag if top of stack is zero. Otherwise, return a false flag.

0< Retrun a true flag if top of stack is negative. Otherwise, return a false flag.

These two tests are the most fundamental ones. All other numeric tests, like > , < , = , etc., can be derived from these two tests. The other tests are left in the high level section. Only these two are implemented in the nucleus.

```
  135 LIST

( STACK INITIATION,  SP@, SP!, RP!, ;S  )

CODE SP@    ( FETCH STACK POINTER TO TOS )
      S R1 MOV,    R1 S -) MOV,    NEXT,    C;

CODE SP!    ( LOAD SP FROM S0 IN USER AREA )
      6 U I) S MOV,    NEXT,    C;

CODE RP!    ( LOAD RP FROM R0 IN USER AREA )
      10 U I) RP MOV,    NEXT,    C;

CODE ;S    ( RESTORE IP FROM RETURN STACK )
      RP )+ IP MOV,    NEXT,    C;

;S


  136 LIST

( RETURN STACK WORDS    LEAVE, >R, R>, R  )

CODE LEAVE    ( FORCE EXIT OF DO-LOOP BY SETTING LIMIT TO INDEX )
      RP () 2 RP I) MOV,    NEXT,    C;

CODE >R    ( MOVE FROM DATA STACK TO RETURN STACK )
      S )+ RP -) MOV,    NEXT,    C;

CODE R>    ( MOVE FROM RETURN STACK TO DATA STACK )
      RP )+ S -) MOV,    NEXT,    C;

CODE R     ( COPY TOP OF RETURN STACK TO TOS )
      RP () S -) MOV,    NEXT,    C;
: R@ R ;
;S


  137 LIST

( TESTS    0=, 0<    )

CODE 0=    ( REVERSE LOGICAL STATE OF TOS  )
      S () TST,    NE IF,    S () CLR,    ELSE,    1 # S () MOV,
      THEN,    NEXT,    C;

CODE 0<    ( LEAVE TRUE IF NEGATIVE, OTHERWISE FALSE )
      S () TST,    MI IF,    1 # S () MOV,    ELSE,    S () CLR,
      THEN,    NEXT,    C;

;S
```

MATH OPERATORS

+	Return the sum of the top two stack items.
D+	Add two double integers on the stack and leave the double integer sum on the stack.
MINUS	Negate the top of stack.
DMINUS	Negate the double integer on top of the stack.

These are the most fundamental math operators which has to be coded in machine codes. Other math operators can be derived from them.

STACK OPERATORS

OVER	Duplicate the second item on the data stack.
DROP	Pop the top item off the data stack.
SWAP	Exchange the top two items on the data stack.
DUP	Duplicate the top item on the data stack.

Since most words use only the topmost items on the data stack, these stack operators are quite adequate. If you have to dig deeper into the data stack to find things, it is a good time to think again on your program design.

DIRECT OPERATIONS IN MEMORY

+!	Add the second number on the stack to the contents of memory addressed by the top stack item.
TOGGLE	Toggle bits in a memory byte addressed by the second item on the stack. Bits to be toggled are those set in the top item on the stack.

These two operators allow us to change memory contents without having to get the memory contents first on the data stack.

```
  138 LIST

( MATH    + , D+ , MINUS, DMINUS    )

CODE +     ( LEAVE SUM OF TWO TOP ITEMS )
      S )+ S () ADD,    NEXT,    C;

CODE D+    ( ADD TWO DOUBLE INTEGERS, LEAVE DOUBLE NUMBER )
      2 S I) 6 S I) ADD,    4 S I) ADC,    S () 4 S I) ADD,
      4 # S ADD,    NEXT,    C;

CODE MINUS    ( TWO'S COMPLEMENT OF TOS )
      S () NEG,    NEXT,    C;
: NEGATE MINUS ;
CODE DMINUS    ( TWO'S COMPLEMENT OF TOS DOUBLE NUMBER )
      S () NEG,    2 S I) NEG,     S () SBC,    NEXT,    C;
: DNEGATE DMINUS ;
;S


  139 LIST

( STACK MANIPULATION    OVER, DROP, SWAP, DUP    )

CODE OVER    ( DUPLICATE SECOND ITEM AS NEW TOS )
      2 S I) S -) MOV,    NEXT,    C;

CODE DROP    ( DROP TOS )
      2 # S ADD,    NEXT,    C;

CODE SWAP    ( EXCHANGE TWO TOS ITEMS )
      2 S I) R1 MOV,    S () 2 S I) MOV,    R1 S () MOV,
      NEXT,    C;

CODE DUP    ( DUPLICATE TOS )
      S () S -) MOV,    NEXT,    C;

;S


  140 LIST

( MEMORY INCREMENT    +! , TOGGLE    )

CODE +!    ( ADD SECOND TO MEMORY ADDRESSED BY TOS )
      2 S I) 0 S @I) ADD,    4 # S ADD,    NEXT,    C;

CODE TOGGLE    ( BYTE AT ADDR-2, BIT PATTERN-1 --- )
      2 S I) S -) MOV,    0 S @I) S () MOVB,    S () RP -) MOV,
      2 S I) RP () BIC,    S )+ S () BIC,    RP )+ S () BIS,
      2 S I) S -) MOV,    2 S I) 0 S @I) MOVB,    6 # S ADD,
      NEXT,    C;

;S
```

MEMORY OPERATIONS

@	Replace the address on top of the stack with its contents.
C@	Replace the address on top of the stack with the byte value it addresses.
!	Store the second stack item in the memory addressed by the top item on the stack.
C!	Store the second byte value on stack into the memory addressed by the top item.

COLON DEFINITION

:	Create a header for a high level colon definition and start the compilation process.
DOCOL	This is the address interpreter which executes a list of addresses like a sequence of subroutines. What it has to do is: save the address in IP on the return stack, to be returned to by ;S , and copy W register, which points to the parameter field of the currently executed colon word, to IP, and call NEXT, to do the work. IP points now to the list of addresses to be executed in sequence. It is equivalent to SUBROUTINE in FORTRAN, without all the complications.
;	Compile a ;S at the end of a colon definition and return to the interpretive or execution mode.

OTHER INNER INTERPRETERS

CONSTANT	Defining word to create named constants.
DOCON	The constant interpreter which copies the contents in the parameter field onto the data stack. This routine interrprets the constant word defined by CONSTANT.
VARIABLE	Defining word to create named memory addresses.
DOVAR	The variable interpreter which pushes the parameter field address of a variable onto the data stack, allowing the contents to be accessed or modified.
USER	Defining word to create user variables.
DOUSE	The user variable interpreter which returns the memory address of the user varaible defined by USER.

DOCON, DOVAR, and DOUSE are interpreters which are invoked by referencing named constants, variables, or user variables.

```
  141 LIST

( MEMORY OPERATIONS    @ , C@ , ! , C!    )

CODE @    ( REPLACE STACK ADDRESS WITH CONTENTS )
      0 S @I) S () MOV,    NEXT,    C;

CODE C@    ( REPLACE STACK ADDRESS WITH POINTED BYTE VALUE )
      0 S @I) R1 MOVB,    177400 # R1 BIC,    R1 S () MOV,
      NEXT,    C;

CODE !    ( STORE SECON AT LOC ADDRESSED BY TOS )
      2 S I) 0 S @I) MOV,    4 # S ADD,    NEXT,    C;

CODE C!    ( STORE SECOND AT BYTE ADDRESSED BY TOS )
      2 S I) 0 S @I) MOVB,    4 # S ADD,    NEXT,    C;

;S


  142 LIST

( COLON DEFINITION    : ,  ;    )

: :    ( CREATE NEW COLON DEFINITION UNTIL  ;  )
      ?EXEC    !CSP    CURRENT @ CONTEXT !
      CREATE    ]    ;CODE    IMMEDIATE
      ( DOCOL: )    IP RP -) MOV,    W IP MOV,    NEXT,    C;

: ;    ( TERMINATE COLON DEFINITION    )
      ?CSP    COMPILE ;S    SMUDGE    [COMPILE] [    ;    IMMEDIATE

;S




  143 LIST

( CONSTANT, VARIABLE, USER    )

: CONSTANT     ( WORD WHICH LATER CREATES CONSTANTS )
      CREATE    SMUDGE    ,    ;CODE
      ( DOCON: )    W () S -) MOV,    NEXT,    C;

: VARIABLE    ( WORD WHICH LATER CREATES VARIABLES  )
      CONSTANT    ;CODE
      ( DOVAR: )    W S -) MOV,    NEXT,    C;

: USER       ( CREATE USER VARIABLE  )
      CONSTANT    ;CODE
      ( DOUSE: )    W () S -) MOV,    U S () ADD,    NEXT,    C;

;S
```