LIBRARY OF FORTH ROUTINES AND UTILITIES

PLUME

TEXT-PROCESSING ROUTINES

A MINI-DATABASE MANAGER

GRAPHICS ROUTINES

ASSEMBLY LANGUAGE ROUTINES

APPLICATION FRONT ENDS

BY JAMES D. TERRY

FOREWORD BY MIKE EDELHART

A SHADOW LAWN PRESS BOOK

1

Introduction

Programming languages are like human languages: their main purpose is communication. Programming languages are how we communicate with computers, how we instruct them to carry out the actions we want them to perform. Just as the language we speak and write affects how well we can communicate with our fellow human beings, so too does our choice of a programming language control how effectively we can communicate with our computers. Forth is one of the many programming languages you can use to communicate with your IBM-PC. In this book we try to increase your Forth vocabulary, and make it easier for you to communicate with your IBM-PC. *The Library of Forth Routines* is not an introduction. It assumes that you have at least a working knowledge of Forth. It will, however, take you from your working knowledge to working Forth programs. The ready-to-use "toolkits" provided here should enable you to increase your Forth programming speed and efficiency. The Forth words presented in this book can be used without restriction in any private or commercial program.

FORTH DIALECTS

Each version of Forth available can be thought of as a dialect of the language. The Forth words contained in this book have all been implemented and tested in actual programs, hence they are written in a specific version of Forth. That version is, naturally enough, one published by the author's company, and is known as Atila. You can order Atila and the source for all the words in this book using the coupon you will find in the back of the book. If you are using Atila, you will run into no dialect or version problems.

The words in this book have been written to be universal to almost all Forth dialects. You should also be able to use these programs with other versions of Forth. No unusual or esoteric words specific to Atila have been used whenever possible. Appendix B includes sources for any Atila words that

might possibly not be in your Forth. Additionally, all words have been defined in uppercase, and with the first three letters and length unique, to avoid problems with Forths that have these restrictions. Every effort has been made to present you with Forth code that you can use, whatever version of Forth you have.

THE IBM-PC

While this book is directed toward the IBM-PC and compatibles, most of the words presented could be used in any Forth system on any computer. Only Chapters 4, 5, 6, and 9 are truly dependent on the IBM-PC. Chapter 5 could be applied to any 8088/8086 system, and Chapter 6 to any computer that uses the 8087. This leaves 9 of the book's 13 chapters that can be implemented on any Forth system not run on a PC.

The version of Forth used in this book, and most other Forths available for the IBM-PC, is a 16-bit or small memory model Forth: This means that it uses address data that are 16 bits wide. While the IBM-PC can have up to 640K of memory, 16 bits only allows 64K to be addressed. To utilize the extra memory, most small memory model Forths come with a set of words to access the extra memory using 32-bit pointers, which take up two stack entries. There is, as yet, no standardization in the Forth community for these words. Presented below are the Atila words for accessing the extra memory on the IBM-PC. If you are not using Atila, you will need to refer to the documentation provided with your version of Forth to find equivalent words. (*Note:* The stack notation used in this book is described in Appendix A).

>X (– A)	Leave the segment address Atila is executing in, used primarily to convert Atila addresses to 32-bit format.
X! (N A1 A2 –)	Store N in the cell at segment A2, offset A1.
X@ (A1 A2 – N)	Leave N, the cell at segment A2, offset A1.
X <CMOVE (A1 A2 A3 A4 N –)	Move N bytes from segment A2, offset A1, to segment A4, offset A3. Move backwards in memory.
XC! (N A1 A2 –)	Store N in the byte at segment A2, offset A1.
XC@ (A1 A2 – N)	Leave N, the byte at segment A2, offset A1.

XCMOVE (A1 A2 A3 A4 N –)	Move N bytes from segment A2, offset A1, to segment A4, offset A3. Move forward in memory.
XFILL (A1 A2 N1 N2 –)	Fill N1 bytes of memory with byte N2. Start at segment A2, offset A1.

These words are used only in Chapters 4 and 9.

CONCLUSION

The toolkits provided in this book are building blocks you can use in your Forth programs. Forth is an extensible language, and the words in this book can also be extended. Throughout the text suggestions for extensions have been provided. It is the author's hope that, as you use the words in this book, you find many interesting and challenging ways to extend the building blocks that have been provided.

2

CASE Statements

Words Defined in This Chapter:

CASE:	Vectored case defining word.
CASE	Branching case word.
=OF	Equal condition in a case statement.
>OF	Greater than condition in a case statement.
<OF	Less than condition in a case statement.
RNG−OF	Range of number condition in a case statement.
END−OF	End of condition in a case statement.
ENDCASE	End of branching case.

CASE STATEMENTS

In this chapter we present a set of words that implement case statements in Forth. Standard Forth contains almost all other standard branching and iteration constructs. Case statements are not included, but this chapter corrects that omission. Two types of case statements make up this chapter. The first is a simple vectored case word. This word will be called CASE:, or "case-colon". It enables us to define a single word that expects an integer on the stack. It will execute the word corresponding to that number in its list, then exit itself. Here is an example:

```
ATILA OK    : ONE . " Word One " ; [return]
ATILA OK    : TWO . " Word Two " ; [return]
ATILA OK    : THREE . " Word Three " ; [return]
```

```
ATILA OK    CASE: EXAMPLE ONE TWO THREE ; [return]
```

We now have created a word, **EXAMPLE**, that is a vectored case statement. If we pass it a "one," it will execute the first word in its list.

```
ATILA OK 1 EXAMPLE [return]
Word One ATILA OK
```

Passing it a two would execute the word **TWO**, passing a three would cause **THREE** to be executed. Passing a number not in the range of one to three would result in unpredictable behavior. There is no limit to how many words may be in any single vectored case word. No other code can be in such a word, however. The list of words in a vectored case word is just used as a list in this case.

The next kind of case statement we wish to present is more general; it allows different codes to be executed, depending on the value of an integer at the start of the construct. A typical example might look like this:

(Number of eyes monster has is on the stack.)

```
CASE
  0 =OF ." A blind monster smells you." END-OF
  1 =OF ." A Cyclops stares at you." END-OF
  2 =OF ." It looks normal enough." END-OF
  NOT-OF ." A multi-eyed creature is eying you." END-OF
ENDCASE
```

As you can see, this construct provides a simple way to express what would be awkward with nested IF statements, although anything we do with a case statement we could also do with an IF. First let's present the words, then we'll get to a complete description.

The additional words <OF, >OF and RNG-OF enable us to check for less than, greater than, and range of numbers, respectively, in a case statement. This short example should demonstrate:

```
#IN CASE
  1 =OF ." You typed a one." END-OF
  5 <OF ." Less than 5! " END-OF
  6 12 RNG-OF ." Between 6 and 12 " END-OF
  NOT-OF ." Greater then 12! " END-OF
ENDCASE
```

CASE:

Define a vectored case word.

Stack on Entry: (Compile Time) Empty.
(Run Time) Number of word in list to execute.

Stack on Exit: (Compile Time) Empty.
(Run Time) Undetermined, depends on word executed.

Example of Use: See previous text.

Algorithm: Put the language in compile mode. This will cause all words to have their addresses enclosed in the dictionary. At run time, multiply number passed by two (two bytes for each address). Add this to the start address and fetch the proper address to execute.

Suggested Extensions: Define an ENDCASE word for this construct that will store the number of words in the list. Use this number to make sure undefined words don't get executed.

Definition:

```
:CASE: <BUILDS [COMPILE] ] DOES> 2* + @ EXECUTE ;
```

CASE

Define a general case word.

Stack on Entry: (Compile Time) Empty.
(Run Time) Number to be used in comparisons.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See previous text.

Algorithm: Place the number to be compared against on the return stack. Then store the depth of the data stack in #OFS. This will be used by ENDCASE to determine how many ENDIFs to compile.

Suggested Extensions: Define new case statements that use strings or float-

ing points after we have introduced them in the following chapters.

Definition:

```
0 VARIABLE #OFS
: CASE COMPILE >R SP@ #OFS ! ; IMMEDIATE
```

=OF

Start a branch in a case statement, when an equal condition is met.

Stack on Entry: (Compile Time) Empty.
(Run Time) Number to be compared against case value.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See previous text.

Algorithm: The word CRS will get a copy of the number being held on the return stack. Compare it, using the equal word, to the number on the stack. Compile an IF statement that will handle the branching based on the comparison.

Suggested Extensions: None.

Definition:

```
: CRS R> DUP >R ;
: /=OF CRS = ;
: =OF COMPILE /=OF [COMPILE] IF ; IMMEDIATE
```

<OF

Start a branch in a case statement, when a "less than" condition is met.

Stack on Entry: (Compile Time) Empty.
(Run Time) Number to be compared against case value.

Stack on Exit: (Compile Time) Empty.

(Run Time) Empty.

Definition:

```
: /<OF CRS > ;
: <OF COMPILE /<OF [COMPILE] IF ; IMMEDIATE
```

>OF

Start a branch in a case statement, when a "greater than" condition is met.

Stack on Entry: (Compile Time) Empty.
(Run Time) Number to be compared against case value.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Definition:

```
: />OF CRS < ;
: >OF COMPILE />OF [COMPILE] IF ; IMMEDIATE
```

RNG-OF

Start a branch in a case statement, when a range condition is met.

Stack on Entry: (Compile Time) Empty.
(Run Time) Lower limit of range.
Upper limit of range.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Definition:

```
: /RNG-OF CRS SWAP OVER >= LROT <= AND ;
: RNG-OF COMPILE /RNG-OF [COMPILE] IF ; IMMEDIATE
```

NOT–OF

Start an unconditional branch in a case statement.

Stack on Entry: (Compile Time) Empty.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Definition:

```
: NOT-OF -1 LITERAL [COMPILE] IF ; IMMEDIATE
```

ENDCASE

End a case statement.

Stack on Entry: (Compile Time) One entry for each OF .. END–OF pair.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See previous text.

Algorithm: At run time clear the return stack. At compile time, compile an ENDIF for each IF compiled by an OF word. Use the variable #OFS to check against the data stack, which should hold an address for each OF statement.

Suggested Extensions: None.

Definition:

```
: ENDCASE COMPILE R> COMPILE DROP BEGIN
SP@ #OFS @ <> WHILE
[COMPILE] ENDIF
REPEAT ; IMMEDIATE
```

END–OF

End an OF branch of a CASE statement.

Stack on Entry: (Compile Time) Empty.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See previous text.

Algorithm: At compile time, compile an ELSE that will be executed if the IF compiled by the OF statement fails.

Suggested Extensions: None.

Definition:

```
: END–OF [COMPILE] ELSE ; IMMEDIATE.
```

3

A Programmer's Calculator

In this chapter, we write a simple programmer's calculator. It is provided as an example of a complete Forth program. A programmer's calculator is a useful "stand-alone" tool. It will enable us to input numbers in decimal, binary, hex, and octal bases, to perform the normal arithmetic calculations on them (addition, subtraction, division, and multiplication), and to do some logical operations (like AND, OR and NOR). Our calculator will also enable us to convert between numbers and ASCII.

STEP 1: THE DESIGN

Before we start to write any program, we should design it completely. The design includes both how the program will look to the user and how the program itself will work internally. In this case, let's start with how the user will see our calculator. Our programmer's calculator will use postfix math, just like Forth itself. The user will be able to enter numbers or operations. The available operations will include: the math operators addition, subtraction, multiplication, and division; the logical operators AND, OR, and Exclusive-OR; the ability to display the current number in bases 2, 8, 10, and 16; and the ability to display the ASCII equivalent of a number or vice versa. Additionally, we'll give our calculator a memory function that enables it to hold and recall a single number. Of course, the user can at any time clear the display or the memory. Our display on the screen will look like this the following illustration (see Figure 3-1).

Internally our program will use the Forth data stack as the stack for our calculator. Obviously, we will need words to perform each of the operations allowed in the above description. What else will we need? A routine that takes the input from the user and determines if it is a number or if an operation will be needed. So will a word that displays the top number on the stack on the first

line of our calculator. And, we'll want a word that will write out our menu on the display.

CURRENT NUMBER: _____ BASE: _____ STACK DEPTH: _____			
OPERATIONS: (Precede with a /)			
+ – ADD – – SUBTRACT	* – MULTIPLY / – DIVIDE	& – AND | – OR	X – Exclusive–OR N – NOT
B – Binary Display D – Decimal Display	O – Octal Display H – HEX Display	U – Unsigned S – Signed	A – ASCII Display
! – Store Memory	@ – Fetch Memory	M – Clear Memory	
C – Clear Display	E – END Program		
ENTER NUMBER OR OPERATION => _____			

Figure 3-1

STEP 2: START CODING

Let's start by trying to save the user of our calculator some problems. Many of the operations listed above will require a specific number of arguments. Before our calculator tries to add two numbers we should make sure that it has two numbers on the stack to add. Some operations, like memory store, for example, will only require a single number on the stack. Here is a word that will check to make sure there are enough numbers on the stack. It will take as an argument the number of stack entries needed and will return a flag. The flag will be true if there are enough numbers on the stack, false if there are too few. This word, STACK_CHECK, will have to make sure not to count its arguments when it looks at the stack. Here goes:

```
: STACK_CHECK DEPTH 1-<= ;
```

Whenever there are too few arguments we'll want to tell the user. Instead of repeating the message after each call to STACK_CHECK, it would be easier to make it part of the word itself. It did look too easy, didn't it? Here is a new version:

```
( Make sure there are enough entries )
( N – F )
( N – Number of entries requires )
```

```
( F -- True or False )
: STACK_CHECK DEPTH 1- <= DUP NOT IF
    2 VTAB 0 HTAB
    ." Not enough data for operation."
  ENDIF ;
```

The VTAB and HTAB words position the cursor on line two of the display. These are not standard Forth words; no standard words exist for this purpose. Consult your manual to see how to do this in your version of Forth. With error checking in hand we can proceed to write some of the operators. Let's start with the math operators.

```
( Add two numbers )
: ADD 2 STACK_CHECK IF + ENDIF ;
```

ADD makes sure we have enough data using our error-checking word; if there is, it proceeds. The same will hold true for all the following.

```
( Subtract two numbers )
: SUB 2 STACK_CHECK IF - ENDIF ;

( Multiply two numbers )
: MULT 2 STACK_CHECK IF * ENDIF ;

( Divide two numbers )
: DIVIDE 2 STACK_CHECK IF / ENDIF ;

( AND two numbers )
: CAND 2 STACK_CHECK IF AND ENDIF ;
```

Notice how we had to call our AND something else to avoid redefining the Forth word AND.

```
( OR two numbers )
: COR 2 STACK_CHECK IF OR ENDIF ;

( Exclusive-OR two numbers )
: CXOR 2 STACK_CHECK IF XOR ENDIF ;
```

Now, to break the monotony, we'll write an operation on a single number. The logical inverse of a number can be found by Exclusive-ORing it with all ones.

```
( NOT, or logical inverse a number. )
: CNOT 1 STACK_CHECK IF -1 XOR ENDIF ;
```

Next let's deal with the memory function of our calculator. First, we need a variable that holds the actual memory value.

```
0 VARIABLE MEMORY
```

Our calculator can perform three operations: store to memory, fetch from memory, and clear memory. Here are the words that will accomplish that.

```
( Store top number in memory. )
: !MEM 1 STACK_CHECK IF DUP MEMORY ! ENDIF ;

( Fetch number from memory. )
: @MEM MEMORY @ ;

( Clear memory. )
: 0MEM MEMORY 0SET ;
```

Now we can start to deal with the display operations. These operations don't really affect the numbers on the stack, but rather how the top number is displayed. Forth has a built in variable, BASE, that determines what base that the numbers printed by DOT (.) will be displayed in. This makes our job a lot easier. A few variables will hold whether or not we want the output displayed signed or unsigned, numeric or ASCII. Here is the code:

```
( Signed or unsigned variable )
( Hold true if signed output )
0 VARIABLE SIGNED?

( Numeric or ASCII variable )
(Hold true if numeric output )
0 VARIABLE NUMERIC?

( Display Numeric ) : YES_NUMERIC -1 NUMERIC? ! ;

( Make display binary )
: 2BASE 2 BASE ! YES_NUMERIC ;

( Make display octal )
: 8BASE 8 BASE ! YES_NUMERIC ;

( Make display decimal )
( An already existing Forth word. )
: 10BASE DECIMAL YES_NUMERIC ;

( Make display hex )
( An already existing Forth word. )
: 16BASE HEX YES_NUMERIC ;
```

```
( Display Signed )
: YES_SIGNED -1 SIGNED? ! ;

( Display Unsigned)
: NOT_SIGNED SIGNED? 0SET ;

( Display ASCII )
: NOT_NUMERIC NUMERIC? 0SET ;
```

The last two operations clear the display (which for our purpose means empty the stack) and end the program.

```
( Drop numbers until the stack is empty )
: CLEAR_STACK BEGIN DEPTH 0 <> WHILE DROP REPEAT ;

( Clear the screen, leave the program. )
: END_CALCULATOR HOME ." Calculator complete. " CR ABORT ;
```

HOME is another nonstandard Forth word that clears the display screen. See the documentation that came with your Forth to determine the appropriate word in your version of Forth.

We need a number of words to draw the top line of the display. The first prints the top number on the stack. It must see if the stack is empty, in which case it will print seven blanks, or if numeric or ASCII output is desired. If the current output is numeric it must check SIGNED? to see how to print the number. It ends up looking like this:

```
: NUMBER_DISPLAY
  0 VTAB 15 HTAB DEPTH 0= IF
    7 0 DO SPACE LOOP
  ELSE
    NUMERIC? IF
      SIGNED? IF
        DUP .
      ELSE
        DUP U.
      ENDIF
    ELSE
      EMIT
    ENDIF
  ENDIF ;
```

The second display word must print out what the current base is.

```
: BASE_DISPLAY
  0 VTAB 25 HTAB BASE @ DUP 2 = IF
    ." Binary "
```

```
ELSE
   DUP 8 = IF
      ." Octal "
   ELSE
      10 = IF
         ." Decimal "
      ELSE
         ." Hex "
ENDIF ENDIF ENDIF ;
```

The final display word will print out how many numbers are on the stack.

```
: DEPTH_DISPLAY 0 VTAB 35 HTAB DEPTH . ;
```

We join all three into a single word for convenience:

```
: DISPLAY NUMBER_DISPLAY BASE_DISPLAY DEPTH_DISPLAY ;
```

When we start the program we want to draw the screen display. Here is that start-up word:

```
: MENU HOME
." CURRENT NUMBER:   BASE:   STACK DEPTH:" CR CR
." OPERATIONS: (Precede with a /)" CR CR
```

." + – ADD	* – MULTIPLY	& – AND	X – Exclusive–OR" CR
." – – SUBTRACT	/– DIVIDE	| – OR	N – NOT" CR CR
." B – Binary Display	O – Octal Display	U – Unsigned	A – ASCII Display" CR
." D – Decimal Display	H – Hex Display	S – Signed" CR CR	
." ! – Store Memory	@ – Fetch Memory	M – Clear Memory" CR CR	
." C – Clear Display	E – END Program" CR CR		

```
." ENTER NUMBER OR OPERATION => " CR ;
```

STEP 3: PUTTING THE PIECES TOGETHER

Now we have almost all the words that our calculator program will use. We make a list in memory of all the possible operations, one character representing each possible operation. A word will search this list when an operation is entered, and then execute the word that implements that operation. First, we must make the list. The list is a one-dimensional array. The first entry is how many elements are in the list.

```
( Convert ASCII to integer and enclose it in dictionary )
: AI BL WORD 1+ C@ C, ;

( Array of commands; there are 20)
20 CVARIABLE COMS AI + AI * AI & AI X AI – AI / AI|   AI N
AI B AI O AI U AI A AI D AI H AI S AI ! AI @ AI M AI C AI E 0 C,
```

We'll use the vectored case word to execute the commands, (hold your breath):

```
CASE: DO__COMS ADD MULT CAND CXOR SUB DIVIDE COR CNOT
2BASE 8BASE NOT__SIGNED NOT__NUMERIC 10BASE 16BASE
YES__SIGNED !MEM @MEM CLEAR__MEM CLEAR__STACK END__
PROGRAM ;
```

Here is the word that will search the list for the command passed on the stack:

```
: SEARCH__COMS 0 COMS C@ 1 DO
     OVER COMS I + C@ = IF
        DROP I LEAVE
     LOOP
     SWAP DROP ;
```

SEARCH__COMS will leave a zero on the stack if the command passed is not found in the list; it will leave the number of the command if it is found. We'll use SEARCH__COMS in this word to actually implement command execution.

```
: EXECUTE__A__COMMAND SEARCH__COMS ?DUP IF
     DO__COMS
   ELSE
     1 VTAB 0 HTAB ." Invalid command"
   ENDIF ;
```

Before we can go any further, we need a word that will handle input. This word must input a number or an operation. If the string input starts with a

slash, this word will leave a true flag and the character following the slash. This should be an operation. If no slash is found, it will convert the input to a number. If the NUMERIC? variable is false, it will take that number as the ASCII value of the first character input. If NUMERIC? is true, it will convert the string input to a number, in the current base. In either case, it will push that number on the stack. Here is the code:

```
77 CCONSTANT SLASH

: INPUT 12 VTAB 18 HTAB 20 0 DO SPACE LOOP 12 VTAB 18 HTAB
   QUERY >IN 0SET BL WORD DUP 1+ C@ SLASH = IF
      2+ C@ EXECUTE_A_COMMAND
   ELSE
      NUMERIC? IF
         >R 0, R> >BINARY DROP
      ELSE
         1+ C@
      ENDIF
   ENDIF ;
```

Step 4: THE FINAL PROGRAM

All the words we need for our final word are now ready. Our final word will be an endless loop. The END_PROGRAM word will be the only exit, by means of an ABORT. Without further hesitation:

```
: CALCULATOR MENU BEGIN
      INPUT
      DISPLAY
   0 UNTIL ;
```

This is an example of a complete, though simple, Forth program. To run the program we just type CALCULATOR at our display.

4

Full-Screen Editor

Forth can be programmed directly by simply entering word definitions at the OK prompt. After a few typos that require you to retype entire lines, you might wish there was some other way. This chapter will present another way: a full-screen editor for Forth blocks. It will enable us to easily enter and modify source text. The word EDIT will invoke the editor. EDIT will expect a screen or block number to edit on the stack.

The editor will place the text onto the screen and we will be able to use the arrow keys to move the cursor around and enter text. Screens will be 24 lines of 40 characters. If an 80-column screen is available, the right-hand side will hold some instructions for the user. Control keys will offer more commands. Here is a summary:

Key	Effect
Up Arrow	*Move the cursor up a single line.*
Down Arrow	*Move the cursor down a single line.*
Left Arrow	*Move the cursor left a single character.*
Right Arrow	*Move the cursor right a single character.*
Del	*Delete the character at the current cursor position.*
Ins	*Place the editor in insert mode.*
CTRL–O	*Insert a line on the screen.*
CTRL–F	*Delete a line.*
CTRL–K	*Erase the screen.*
CTRL–X	*Exit the editor, saving the current screen.*
CTRL–Q	*Exit the editor without saving the current screen.*

Home	*Place the cursor in the upper left-hand corner of the screen.*
End	*Place the cursor in the middle of the screen.*
Enter	*If in insert mode, take editor out of insert mode. Otherwise, move cursor to start of next line.*

The editor will work by directly manipulating screen memory. The variable S_START holds the starting segment address. Because screen memory is outside of Forth's 64K segment, the extra memory words discussed in Chapter 1 are used.

Suggested Extensions: The editor presented in this chapter, while sufficient for productive use, does have a number of possible extensions. One of the most useful would be the ability to move blocks of text, especially between screens. Another might be the ability to move to the next or previous screen while remaining in the editor.

Extending the editor to edit more than a screen at a time would be a useful, but quite major revision.

S_START (– N)

A constant holding the starting segment of screen memory.

Stack on Entry: Empty.

Stack on Exit: (N) – The starting segment of screen memory.

Example of Use: See words defined below.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
( For a Monochrome Display )
HEX B000 CONSTANT S_START DECIMAL

( For a Color Display )
HEX B800 CONSTANT S_START DECIMAL
```

XDL (– N)

A constant holding the number of bytes used to store each line on the display.

Stack on Entry: Empty.

Stack on Exit: (N) – The number of bytes used for each line.

Example of Use: See words defined below.

Algorithm: The IBM display uses to two bytes for each character. See your *Technical Reference Manual* for details.

Suggest Extensions: None.

Definition:

```
( For Forty-Column Display )
40 CCONSTANT XDL

( For a Color Display)
80 CCONSTANT XDL
```

TABLE–SEARCH (A N1 – (N2) F)

Search a table of integers for a specific entry.

Stack on Entry: (A) – The address of the table to search. The first entry of the table must be its length.
(N1) – The integer to search for.

Stack on Exit: (N2) – The position of the integer, if it was found.
(F) – A Boolean flag, true if the integer was found in the table, false otherwise.

Example of Use:

If A_TABLE was a table of integers and 99 was not in that table, then

```
A_TABLE 99 TABLE–SEARCH
```

would leave a false flag on the stack.

Algorithm: Place a false flag on the stack. Obtain the length of the table and start a loop through the table's entries. If a matching entry is found, change the false flag on the stack to true and leave the loop index on the stack, then exit the loop. If the loop falls through, the false flag will be left on the stack.

Suggested Extensions: None.

Definition:

```
: TABLE-SEARCH 0 LROT DUP C@ 1 DO
    DUP I + C@ 3 PICK = IF
      >R >R NOT J SWAP R> R> LEAVE
    ENDIF
  LOOP DROP DROP ;
```

XCUR (– A)

A variable used to hold the x position of the cursor.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of XCUR.

Example of Use: See words defined below.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE XCUR
```

YCUR (– A)

A variable used to hold the y position of the cursor.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of YCUR.

Example of Use: See words defined below.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE YCUR
```

PUTON (N –)

Move text from disk to screen memory.

Stack on Entry: (N) – The block number to move.

Stack on Exit: Empty.

Example of Use:

```
19 PUTON
```

This would move the contents of block 19 into screen memory.

Algorithm: Each Forth screen (at least in this editor) is made up of 24 lines of 40 characters. This word uses two loops. The outer loop of 24 is executed once for each line, the inner of 40 moves each individual character. S_START and XDL control where in memory the characters are moved.

Suggested Extensions: None.

Definition:

```
: PUTON BLOCK 24 0 DO
    40 0 DO
      DUP C@ J XDL * I 2* + S_START XC! 1+
    LOOP
  LOOP DROP ;
```

PUTBACK (N –)

Move text from screen memory to disk.

Stack on Entry: (N) The disk block to move text to.

Stack on Exit: Empty.

Example of Use:

```
19 PUTBACK
```

This would move the text in screen memory to block 19.

Algorithm: First clear the disk block. Two loops are used, as in PUTON. The outer is executed once for each line, the inner once for each character in a line. Characters are moved one at a time from screen memory to the disk block.

Suggested Extensions: None.

Definition:

```
: PUTBACK DUP BLOCK 960 + 64 ERASE
  BLOCK 24 0 DO
    40 0 DO
      J XDL * I 2* + S_START XC@ OVER C! 1+
    LOOP
  LOOP DROP ;
```

EKEY (- C)

Wait for a keypress and return its value.

Stack on Entry: Empty.

Stack on Exit: (C) A character representing the keypress.

Example of Use:

```
: WAITA BEGIN EKEY 65 = UNTIL ;
```

This word will wait until an uppercase A is pressed.

Algorithm: The Atila word KEY normally returns a cell value with the lower byte being zero to indicate an extended key code. This word converts that into

a single byte. This was done so SEARCH–TABLE could be used with byte-length integers.

Suggested Extensions: None.

Definition:

```
: EKEY KEY DUP 128 > IF
    256 / 128 +
  ENDIF ;
```

CHECK (–)

Make sure that XCUR and YCUR hold valid values.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: This word does four distinct checks: XCUR is first checked to see if it has gone past the left edge of the screen. If it has, it is set to 39, the right edge of the screen, and YCUR is decremented. XCUR is then checked to see if it has moved off the right edge of the screen. YCUR is checked to see if it has moved off the top or bottom of the screen.

Suggested Extensions: None.

Definition:

```
: CHECK XCUR C@ 255 = IF
    39 XCUR C! -1 YCUR C+!
  ENDIF
  XCUR C@ 40 = IF
    XCUR COSET 1 YCUR C+!
  ENDIF
  YCUR C@ 255 = IF
    23 YCUR C!
  ENDIF
  YCUR C@ 24 = IF
    YCUR COSET
  ENDIF ;
```

Q-ESC (– F)

Handle a control Q.

Stack on Entry: Empty.

Stack on Exit: (F) - A false flag, which causes the editor to terminate.

Example of Use: See words defined below.

Algorithm: Control Q is used to leave the editor without saving the screen being edited. It leaves a false flag on the stack, causing SCREENEDIT (defined below) to fall through and exit.

Suggested Extensions: None.

Definition:

```
: Q-ESC 0 ;
```

X-ESC (- F)

Handle a control X.

Stack on Entry: Empty.

Stack on Exit: (F) - A false flag, which causes the editor to terminate.

Example of Use: See words defined below.

Algorithm: Control X is used to exit the editor normally. An UPDATE is done to mark the screen being worked on for writing to disk. It leaves a false flag on the stack, causing SCREENEDIT (defined below) to fall through and exit.

Suggested Extensions: None.

Definition:

```
: X-ESC UPDATE 0 ;
```

UP (– F)

Handle the up arrow key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Algorithm: Decrement the value held in YCUR. Use CHECK to ensure that YCUR still holds a valid value. Leave a true flag on the stack so SCREEN-EDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: UP -1 YCUR C+! CHECK -1 ;
```

DOWN (- F)

Handle the down arrow key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: Increment the value held in YCUR. Use CHECK to ensure that YCUR still holds a valid value. Leave a true flag on the stack so SCREEN-EDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: DOWN 1 YCUR C+! CHECK -1 ;
```

LEFT (– F)

Handle the left arrow key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: Decrement the value held in XCUR. Use CHECK to ensure that XCUR still holds a valid value. Leave a true flag on the stack so SCREEN-EDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: LEFT -1 XCUR C+! CHECK -1 ;
```

RIGHT (– F)

Handle the right arrow key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: Increment the value held in XCUR. Use CHECK to ensure that XCUR still holds a valid value. Leave a true flag on the stack so SCREEN-EDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: RIGHT 1 XCUR C+! CHECK -1 ;
```

EATLEFT (– F)

Handle the backspace key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: Emit a block to erase the character at the current character position. Decrement the value held in XCUR. Use CHECK to ensure that XCUR still holds a valid value. Leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: EATLEFT 32 EMIT -1 XCUR C+! CHECK -1 ;
```

ONIM (–)

Update the screen to show that insert mode is on.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: If the screen is 80 characters wide, then print "ON" at x position 65, y position 16.

Suggested Extensions: None.

Definition:

```
: ONIM XDL 80 > IF
    16 VTAB 65 HTAB ." ON "
  ENDIF ;
```

OFFIM (–)

Update the screen to show that insert mode is off.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: If the screen is 80 characters wide, then print "OFF" at x position 65, y position 16.

Suggested Extensions: None.

Definition:

```
: OFFIM XDL 80 > IF
      16 VTAB 65 HTAB ." OFF"
   ENDIF ;
```

F–ESC (– F)

Handle the control F key. Control F is used to delete a line.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: If the cursor is not on the bottom line, move each line below the current line up one full line. Erase the bottom line. Leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: F–ESC YCUR C@ 23 <> IF
      24 YCUR C@ 1+ DO
         I XDL * S_START
```

```
      I 1- XDL * S_START
      80 XCMOVE
    LOOP
    3680 80 0 DO
      I OVER + 32 SWAP S_START XC!
    2 +LOOP DROP
  ENDIF -1 ;
```

O-ESC (- F)

Handle the control O key. Control O is used to insert a line.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: If the cursor is not on the bottom line, move each line below the current line down one full line. Erase the current line. Leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: O-ESC YCUR C@ 23 <> IF
    YCUR C@ 1- 22 DO
      I XDL * S_START
      I 1+ XDL * S_START
      80 XCMOVE
    -1 +LOOP
    YCUR C@ XDL * 80 0 DO
      I OVER + 32 SWAP S_START XC!
    2 +LOOP DROP
  ENDIF -1 ;
```

INSERT (- A)

A Boolean variable used to hold whether or not the editor is in insert mode.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of insert.

Example of Use: See words defined below.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE INSERT
```

RETURN (– F)

Handle the return key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: When return is pressed, and the editor is not in insert mode, it causes the cursor to be moved to the start of the next line. XCUR is set to zero and YCUR is incremented. CHECK is used to make sure that XCUR and YCUR still hold valid values. If the editor was in insert mode, take it out of insert mode. Leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggest Extensions: None.

Definition:

```
: RETURN INSERT C@ IF
    INSERT C0SET OFFIM
  ELSE
    XCUR C0SET 1 YCUR C+! CHECK
  ENDIF -1 ;
```

I–ESC (– F)

Handle the insert key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: Invert the value of insert, and print the current insert mode on the screen using either OFFIM or ONIM. Leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: I–ESC INSERT DUP C@ IF
    C0SET OFFIM
  ELSE
    C1SET ONIM
  ENDIF –1 ;
```

D–ESC (– F)

Handle the delete key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: Move all the characters to the right of the cursor on the current line left on positon. Blank the rightmost character on the line. Leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: D-ESC YCUR C@ XDL * XCUR C@ 2* + DUP
  2+ S_START ROT S_START 80 XCUR C@
  2* - XCMOVE 32 YCUR C@ XDL * 78 +
  S_START XC! -1 ;
```

PLUG (C – F)

Place a character on the screen.

Stack on Entry: (C) – The character to place on the screen.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: If insert mode is on, move all the characters to the right of the cursor on the current line one position to the right. Emit the character passed to PLUG onto the screen and use RIGHT to move the cursor to the next location. RIGHT will also leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: PLUG INSERT C@ IF
    YCUR C@ XDL * XCUR C@ 2* + DUP 2+
    SWAP S_START ROT S_START 78 XCUR C@
    2* - X<CMOVE
  ENDIF EMIT RIGHT ;
```

HOMEKEY (– F)

Handle the home key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: Move the cursor to the upper left-hand corner of the screen by setting XCUR and YCUR to zero. Leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: HOMEKEY YCUR C0SET XCUR C0SET -1 ;
```

ENDKEY (– F)

Handle the end key.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: Move the cursor to the middle of the screen by setting YCUR to 12 and XCUR to zero. Leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: ENDKEY 12 YCUR C! XCUR C0SET -1 ;
```

HSBE (– A)

A variable used to hold the block number being edited.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of HSBE.

Example of Use: See words defined below.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 VARIABLE HSBE
```

SMESS (–)

Print out editor information.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Clear the screen. If the screen is 80 characters wide, print out some helpful information.

Suggested Extensions: Mode information could be printed if desired, help screens could be added to appear in this portion of the screen.

Definition:

```
: SMESS HOME XDL 80 > IF
    DUP HSBE !
    8 VTAB 50 HTAB ." Screen -> ".
    10 VTAB 50 HTAB ." CTRL-X - Save"
    11 VTAB 50 HTAB ." CTRL-Q - Quit"
    12 VTAB 50 HTAB ." CTRL-O - Insert Line"
    13 VTAB 50 HTAB ." CTRL-F - Delete Line"
    14 VTAB 50 HTAB ." CTRL-K - Clear Screen"
    16 VTAB 50 HTAB ." Insert Mode => OFF"
    ELSE
    DROP
ENDIF INSERT COSET ;
```

K-ESC (– F)

Handle the control K key. Control K will clear the screen.

Stack on Entry: Empty.

Stack on Exit: (F) – A true flag, used to signal that the editor should not terminate.

Example of Use: See words defined below.

Algorithm: Use HOME to clear the screen. Set XCUR and YCUR to zero and use SMESS to print out some screen information. Leave a true flag on the stack so SCREENEDIT (defined below) will not fall through.

Suggested Extensions: None.

Definition:

```
: K-ESC HOME XCUR C0SET YCUR C0SET
  HSBE @ SMESS -1 ;
```

DO-ESC (N – F)

Execute an editor command.

Stack on Entry: (N) – The number of the command to execute.

Stack on Exit: (F) – A true flag, used to signal that the editor whether or not it should terminate.

Example of Use: See words defined below.

Algorithm: DO-ESC is a vectored case word. It will use the number on the stack to find the proper word to execute. All editor words leave a flag on the stack that will be used by SCREENEDIT (defined below) to determine whether or not to continue.

Suggested Extensions: Any commands you wish to add to the editor would have to be added here and in the ESC-TABLE defined below.

Definition:

```
CASE: DO-ESC Q-ESC X-ESC K-ESC RETURN
      UP DOWN LEFT RIGHT EATLEFT
      D-ESC F-ESC I-ESC O-ESC HOMEKEY
      ENDKEY ;
```

ESC–TABLE (– A)

The list of key codes for the editor commands.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of the table.

Example of Use: See words defined below.

Algorithm: None.

Suggested Extensions: Any commands you wish to add to the editor would have to be added here and to DO–ESC defined above.

Definition:

```
16 CVARIABLE ESC–TABLE 17 C, 24 C, 11 C,
   13 C, 200 C, 208 C, 203 C, 205 C, 8 C,
   211 C, 6 C, 210 C, 15 C, 199 C, 207 C,
   0 ,
```

?ESC (C – F)

If a character is an editor command, execute it.

Stack on Entry: (C) – The character to CHECK.

Stack on Exit: (F) – Flag, true if an editor command was not found.

Example of Use: See words defined below.

Algorithm: Search the ESC–TABLE, if an entry is found, use DO–ESC to execute it.

Suggested Extensions: None.

Definition:

```
: ?ESC ESC–TABLE TABLE–SEARCH DUP IF
     DROP DO–ESC –1
   ENDIF NOT ;
```

SCREENEDIT (–)

Edit the text on the screen.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Place the cursor at the appropriate place using VTAB and HTAB. Get a key. If it is an editor command, ?ESC will handle it. If it is not, pass it to PLUG. Continue until a false flag is left on the stack by an editor command.

Suggested Extensions: None.

Definition:

```
: SCREENEDIT BEGIN
    XCUR C@ HTAB YCUR C@
    VTAB EKEY DUP ?ESC IF
      PLUG
    ELSE
      SWAP DROP
    ENDIF NOT
  UNTIL ;
```

EDIT (N –)

Edit a block.

Stack on Entry: (N) – The block to edit.

Stack on Exit: Empty.

Example of Use:

```
19 EDIT
```

This would invoke the editor for screen 19.

Algorithm: Print the starting messages. Use PUTON to place the text onto the screen. Use SCREENEDIT to edit the text. PUTBACK will place the text back to the disk buffer. Clear the screen.

Suggested Extensions: None.

Definition:

```
: EDIT DUP SMESS DUP PUTON XCUR COSET
   YCUR COSET SCREENEDIT PUTBACK HOME ;
```

5

8088 Macro Assembler

Words Defined in This Chapter:

ASSEMBLER	Define a vocabulary for the assembler words.
1byte	Define single-byte opcodes.
AAA	Assemble the 8088 adjust result of ASCII addition instruction.
AAS	Assemble the 8088 adjust result of ASCII subtraction instruction.
CBW	Assemble the 8088 convert byte to word instruction.
CLC	Assemble the 8088 clear carry flag instruction.
CLD	Assemble the 8088 clear direction flag instruction.
CLI	Assemble the 8088 clear interrupt flag instruction.
CMC	Assemble the 8088 complement the carry flag instruction.
CWD	Assemble the 8088 convert word to double word instruction.
DAA	Assemble the 8088 decimal adjust accumulator after addition instruction.
DAS	Assemble the 8088 decimal adjust accumulator after subtraction instruction.
HLT	Assemble the 8088 halt the processor instruction.
INTO	Assemble the 8088 interrupt if overflow instruction.

IRET	Assemble the 8088 return from interrupt instruction.
LAHF	Assemble the 8088 load 8080 flags into AH register instruction.
LOCK	Assemble the 8088 bus lock prefix instruction.
NOP	Assemble the 8088 no operation instruction.
POPF	Assemble the 8088 pop into the flag register instruction.
PUSHF	Assemble the 8088 push from the flag register instruction.
SAHF	Assemble the 8088 store the AH register into the 8080 flags instruction.
STC	Assemble the 8088 set the carry flag instruction.
STD	Assemble the 8088 set the direction flag instruction.
STI	Assemble the 8088 set the interrupt flag instruction.
WAIT	Assemble the 8088 wait for signal on test instruction.
XLAT	Assemble the 8088 table lookup instruction.
REPE	Assemble the 8088 repeat string instruction until zero flag is not set prefix instruction.
REPZ	Assemble the 8088 repeat string instruction until zero flag is not set prefix instruction.
REP	Assemble the 8088 repeat string instruction until zero flag is not set prefix instruction.
REPNE	Assemble the 8088 repeat string instruction until zero flag is set prefix instruction.
REPNZ	Assemble the 8088 repeat string instruction until zero flag is set prefix instruction.
AAD	Assemble the 8088 adjust AX register for division instruction.
AAM	Assemble the 8088 adjust AX register for multiplication instruction.
b/w	A variable that holds the size data the instruction being assembled will operate on, byte or word.
byte	Cause the next instruction to use a byte-length operand.
cell	Cause the next instruction to use a word-length operand.
tipe	A two-cell variable that holds the addressing mode for each possible operand of an instruction.

value	A two-cell variable holding address values.
#	Mark an operand as having a immediate addressing mode.
]	Mark an operand as having a indirect addressing mode.
reset	Reset "tipe" for the start of a new instruction.
00mod	Define a word that assembles as a 00 mod in the addressing mode byte.
[BX+SI]	Set "tipe" to the addressing mode indirect, employing the BX and SI registers.
(BX+DI)	Set "tipe" to the addressing mode indirect, employing the BX and DI registers.
[BP+SI]	Set "tipe" to the addressing mode indirect, employing the BP and SI registers.
(BP+DI)	Set "tipe" to the addressing mode indirect, employing the BP and DI registers.
[SI]	Set "tipe" to the addressing mode indirect, employing the SI register.
(DI)	Set "tipe" to the addressing mode indirect, employing the DI register.
[BX]	Set "tipe" to the addressing mode indirect, employing the BX register.
ES	Set "tipe" to the addressing mode implied, employing the ES register.
CS	Set "tipe" to the addressing mode implied, employing the CS register.
SS	Set "tipe" to the addressing mode implied, employing the SS register.
DS	Set "tipe" to the addressing mode implied, employing the DS register.
01mod	Define a word that assembles as a 01 mod in the addressing mode byte.
[BX+SI+#]	Set "tipe" to the addressing mode indirect, employing the BX and SI registers and an offset.
(BX+DI+#)	Set "tipe" to the addressing mode indirect, employing the BX and DI registers and an offset.
[BP+SI+#]	Set "tipe" to the addressing mode indirect, employing the BP and SI registers and an offset.
(BP+DI+#)	Set "tipe" to the addressing mode indirect, employing the BP and DI registers and an offset.
[SI+#]	Set "tipe" to the addressing mode indirect, employing the SI register and an offset.

(DI+#)	Set "tipe" to the addressing mode indirect, employing the DI register and an offset.
[BP+#]	Set "tipe" to the addressing mode indirect, employing the BP register and an offset.
[BX+#]	Set "tipe" to the addressing mode indirect, employing the BX register and an offset.
11mod	Define a word that assembles as a 11 mod in the addressing mode byte.
AL	Set "tipe" to the addressing mode implied, employing the AL register.
AX	Set "tipe" to the addressing mode implied, employing the AX register.
CL	Set "tipe" to the addressing mode implied, employing the CL register.
CX	Set "tipe" to the addressing mode implied, employing the CX register.
DL	Set "tipe" to the addressing mode implied, employing the DL register.
DX	Set "tipe" to the addressing mode implied, employing the DX register.
BL	Set "tipe" to the addressing mode implied, employing the BL register.
BX	Set "tipe" to the addressing mode implied, employing the BX register.
AH	Set "tipe" to the addressing mode implied, employing the AH register.
SP	Set "tipe" to the addressing mode implied, employing the SP register.
CH	Set "tipe" to the addressing mode implied, employing the CH register.
BP	Set "tipe" to the addressing mode implied, employing the BP register.
DH	Set "tipe" to the addressing mode implied, employing the DH register.
SI	Set "tipe" to the addressing mode implied, employing the SI register.
BH	Set "tipe" to the addressing mode implied, employing the BH register.
DI	Set "tipe" to the addressing mode implied, employing the DI register.
direction	Set the direction bit in the opcode being formed.

size	Set the size bit in the opcode being formed.
r/m	Set the r/m field in the addressing mode byte being formed.
md	Set the mod field in the addressing mode byte being formed.
,value	Store address of offset for the current instruction being assembled.
1reg@	Get the register value stored in the first position of "tipe".
reg@	Get the register value stored in the current position of "tipe".
1reg	Set the register field in the opcode being formed.
reg	Set the register field in the addressing mode byte being formed.
a-mode	Form and store an addressing mode byte. Store the instructions address values.
16-bit-reg?	Is a 16-bit register being used?
seg-reg?	Is a segmentation register being used?
DEC	Assemble the 8088 decrement instruction.
INC	Assemble the 8088 increment instruction.
POP	Assemble the 8088 move data from the stack instruction.
PUSH	Assemble the 8088 move data too the stack instruction.
ac,data?	Check if the operands for the current instruction are a move of data into an accumulator.
immediate?	Check if there is an immediate operand for this instruction.
eb/w	Store a displacement or address in memory.
,data	Store the displacement or address for an instruction.
s-bit	Set the sign bit in an opcode.
e-mode	From a complete addressing mode byte.
swap-dir	Mark an instruction's operands as moving data in the opposite direction.
dir?	Check the direction an instruction's operands are moving data. Swap the directions if appropriate.
1kind	Define a word to assemble the 8088 arithmetic and logical instructions which use two operands.
ADD	Assemble the 8088 add instruction.
ADC	Assemble the 8088 add with carry instruction.

And	Assemble the 8088 logical AND instruction.
CMP	Assemble the 8088 compare instruction.
Or	Assemble the 8088 logical OR instruction.
SBB	Assemble the 8088 subtract with borrow instruction.
SUB	Assemble the 8088 subtract instruction.
XOR	Assemble the 8088 Exclusive–OR instruction.
TEST	Assemble the 8088 test of data instruction.
c–bit	Set the c bit for shift and rotate instructions.
srkind	Define a word to assemble the 8088 shift and rotate instructions.
RCL	Assemble the 8088 rotate through carry-left instruction.
RCR	Assemble the 8088 rotate through carry-right instruction.
ROL	Assemble the 8088 rotate-left instruction.
ROR	Assemble the 8088 rotate-right instruction.
SAR	Assemble the 8088 shift arithmetic right instruction.
SHL	Assemble the 8088 shift-left instruction.
SHR	Assemble the 8088 shift-right instruction.
LDS	Assemble the 8088 load register and DS instruction.
LEA	Assemble the 8088 load effective address instruction.
LES	Assemble the 8088 load register and ES instruction.
mkind	Define a word to assemble the 8088 mathematical instructions.
DIV	Assemble the 8088 divide instruction.
IDIV	Assemble the 8088 unsigned divide instruction.
IMUL	Assemble the 8088 unsigned multiply instruction.
MUL	Assemble the 8088 multiply instruction.
NEG	Assemble the 8088 negate instruction.
NOT	Assemble the 8088 ones complement instruction.
2kind	Define a word to assemble the 8088 string instructions.
CMPS	Assemble the 8088 string compare instruction.
LODS	Assemble the 8088 load accumulator from memory instruction.
MOVS	Assemble the 8088 string move instruction.

SCAS	Assemble the 8088 compare against memory instruction.
STOS	Assemble the 8088 store accumulator to memory instruction.
IN	Assemble the 8088 transfer data from a port to accumulator instruction.
OUT	Assemble the 8088 transfer data from an accumulator to a port instruction.
INT	Assemble the 8088 software interrupt instruction.
SEG	Assemble the 8088 segment override instruction.
XCHG	Assemble the 8088 exchange data instruction.
LONG-CALL	Assemble the 8088 long (intersegment) call instruction.
CALL	Assemble the 8088 short (intrasegment) call instruction.
RET	Assemble the 8088 short (intrasegment) return instruction.
LONG–RET	Assemble the 8088 long (intersegment) return instruction.
LONG–BRANCH	Assemble the 8088 long (intersegment) jump instruction.
BRANCH	Assemble the 8088 short (intrasegment) jump instruction.
MOV	Assemble the 8088 data move instruction.
bopc	Define words to assemble 8088 branching opcodes.
JA	Assemble the 8088 jump on above instruction.
JNBE	Assemble the 8088 jump on not below or equal instruction.
JAE	Assemble the 8088 jump on above or equal instruction.
JNB	Assemble the 8088 jump on not below instruction.
JB	Assemble the 8088 jump on below instruction.
JNAE	Assemble the 8088 jump on not above or equal instruction.
JBE	Assemble the 8088 jump on below or equal instruction.
JNA	Assemble the 8088 jump on not above instruction.

JCXZ	Assemble the 8088 jump if CX equals zero instruction.
JE	Assemble the 8088 jump on equals instruction.
JZ	Assemble the 8088 jump on zero instruction.
JG	Assemble the 8088 jump on greater than instruction.
JNLE	Assemble the 8088 jump on not less or equal instruction.
JGE	Assemble the 8088 jump on greater or equal instruction.
JNL	Assemble the 8088 jump on not less instruction.
JL	Assemble the 8088 jump on less instruction.
JNGE	Assemble the 8088 jump on not greater or equal instruction.
JLE	Assemble the 8088 jump on less than or equal instruction.
JNG	Assemble the 8088 jump on not greater than instruction.
JNE	Assemble the 8088 jump on not equal to instruction.
JNZ	Assemble the 8088 jump on not zero instruction.
JNO	Assemble the 8088 jump on not overflow instruction.
JNP	Assemble the 8088 jump on no parity instruction.
JPO	Assemble the 8088 jump on odd parity instruction.
JNS	Assemble the 8088 jump on not sign instruction.
JO	Assemble the 8088 jump on overflow instruction.
JP	Assemble the 8088 jump on parity even instruction.
JPE	Assemble the 8088 jump on parity even instruction.
JS	Assemble the 8088 jump on sign instruction.
LOOP	Assemble the 8088 loop if CX is not equal to zero instruction.
LOOPE	Assemble the 8088 loop if CX is equal to zero and the zero flag is set instruction.
LOOPZ	Assemble the 8088 loop if CX is equal to zero and the zero flag is set instruction.
LOOPNE	Assemble the 8088 loop if CX is not equal to zero and the zero flag is not set.

LOOPNZ	Assemble the 8088 loop if CX is not equal to zero and the zero flag is not set.
J	Assemble the 8088 jump always instruction.
FLABEL	Complete a forward branch instruction.
ZIF	Start an /IF construct, check if the zero flag is set.
NZIF	Start an /IF construct, check if the zero flag is not set.
SIF	Start an /IF construct, check if the sign flag is set.
NSIF	Start an /IF construct, check if the sign flag is not set.
CIF	Start an /IF construct, check if the carry flag is set.
NCIF	Start an /IF construct, check if the carry flag is set.
/BEGIN	Mark the beginning of a structured loop construct.
/REPEAT	Mark the end of a /BEGIN .../IF .../REPEAT construct.
NEXT	Mark the end of a word defined in assembler.
END–SUB	Mark the end of a subroutine defined with the word SUBROUTINE.
MEND	Mark the end of a macro definition.
/ENDIF	Mark the end of a conditional branching construct.
/ELSE	Allow an alternative branch in a conditional branching construct.
CODE	Start the definition of a word in assembler.
SUBROUTINE	Start the definition of an assembly language subroutine.
MACRO	Start the definition of a macro.

This chapter presents a complete 8088 structured macro assembler for Forth. An assembler is an integral part of any language system, a part neglected by most other languages. In Forth, the assembler is used to define individual words in machine language. Because words are used, it is easy to intermix high-level Forth and assembler in the same program.

Why would we want any part of our programs to be in assembly (or machine) language? As its name implies, machine language is *the* language of our computer. In the case of the IBM-PC, the 8088 CPU is what all programs execute on. Anything the computer can do is controlled by this very central processing unit. There may be times when even Forth is unable to access a specific machine feature. In these cases, we must have an assembler to accomplish our job at all.

Some cases need an assembler for efficiency. Whenever the 8088 CPU executes a Forth program there are the intermediate steps of stepping through all the words, until the lowest level machine language words are found. Most of the time, the IBM seems to have enough time to process our programs with no delay, but on occasion we need it to be faster. Rewriting a high-level Forth word in machine language using an assembler will speed up the execution of that word considerably. With a Forth assembler you can fine tune the performance of your programs.

In order to program in assembly language we must know a lot about the 8088 CPU. We must know how many registers it has. We must be able to see how it accesses memory. We must understand its instruction set. We need to know the function of each instruction the 8088 is capable of executing. But to actually write an assembler we need to know more. We need to know how the 8088 itself looks into memory and decodes its instructions. Learning assembly language can seem difficult; it is an involved subject. There are many

8086 ADDRESSING MODE BYTE

M	M	R	R	R	r/m	r/m	r/m
MOD		REG			R/M		

Figure 5-1

8086 EFFECTIVE ADDRESS DETERMINATION

R/M	MOD 00	MOD 01	MOD 10	MOD 11	MOD 11
000	[BX+SI]	[BX+SI+#]	[BX+SI+#]	AL	AX
001	(BX+DI)	(BX+DI+#)	(BX+DI+#)	CL	CX
010	[BP+SI]	[BX+SI+#]	[BX+SI+#]	DL	DX
011	(BP+DI)	(BX+DI+#)	(BX+DI+#)	BL	BX
100	[SI]	[SI+#]	[SI+#]	AH	SP
101	(DI)	(DI+#)	(DI+#)	CH	BP
110	Memory	[BP+#]	[BP+#]	DH	SI
111	[BX]	[BX+#]	[BX+#]	BH	DI
				w=0	w=1

Figure 5-2

books available on this subject, but since this book does not try to teach assembly-language programming, from this point on in this chapter we will assume a basic familiarity with the 8088 and assembly language.

Our Forth assembler will be used to define words. As in Forth itself, we'll have numerous structured constructs so we can write structured code. Because our scope is only a single word, we won't need a symbol table or many of the other features of a stand-alone assembler. We will have macros, though, mostly because they are so easy to implement in Forth.

The assembler will assemble each instruction into memory at the current dictionary pointer. A thorough study of the instruction set of the 8088 must be done before an assembler is written. A number of instructions are one- or two-byte constants, which are simple to write. They will use comma or c-comma to store their values in the dictionary. More complex instructions must deal with

the addressing modes of the 8088. The effective address calculation of an 8088 instruction employs an addressing mode byte that is of the form in Figure 5-1.

The REG field determines which register is used, or if no register is required by the instruction, it can be an extension of the opcode itself. The MOD and R/M bit fields of an addressing byte can be cross referenced to determine how the effective address of an instruction is determined.

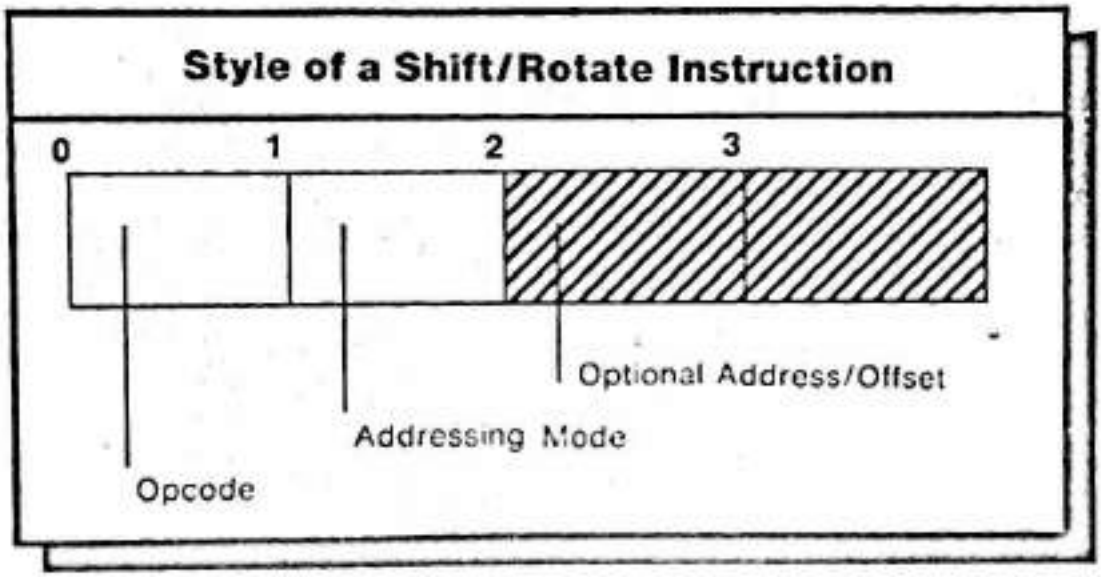
The W or size bit is held in the opcode itself.

We will use a two-celled variable called "tipe" to hold the type of addressing being used. As each operand is specified, "tipe" will be filled in appropriately.

A careful examination of the 8088 instruction set will find instructions that can be grouped together. Examples are the arithmetic and logical instructions, the shift and rotate instructions, and the branching opcodes. Each of these groups of opcodes has a very similar construction at the bit level. This information will allow us to write defining words that can cover the groups of instructions.

Let's take a look at the shift and rotate instructions, one of the simplest groups. Each shift and rotate instruction's base opcode has a unique seven bits, followed by an addressing mode byte, and by address or offset information, if it is required. (See Figure 5-3.) In each opcode there is a c bit, which the 8088 uses to determine how many bits are to be shifted. Since all the shift and rotate words only differ by the first seven bits, we can write one word that will assemble them all.

Other information we can derive from looking at the 8088 instruction set at the bit level will be used in our assembler. Every time a register is specified, for example, three bits are used. Throughout the instruction set they retain the same meaning. This enables us to use data from "tipe" for inclusion in the opcodes and addressing mode bytes.



Coding Instructions

When you write an instruction for a normal assembler, you specify the addressing mode of the instruction symbolically. MOV AX,BX and MOV AX, [BX] are different instructions because of the square brackets that specify indirect, instead of direct, addressing. Some instructions have no addressing mode, like PUSHF or NOP. Our 8088 assembler will use a similar method to specify addressing modes. Instructions that require no addressing mode in a normal assembler, like LAHF or CBW, will require none in our assembler. Every instruction that uses an address will, however, require an addressing mode indicator. Even those that, in a normal 8088 assembler, assume a default mode require an addressing mode indicator in our 8088 assembler. Here is a summary of the addressing modes used:

Forth Indicator	Normal	Forth
Register (Implied)	MOV AL,DL	AL DL MOV
# – Immediate	ADD AX,10	AX 10 # ADD
% – Direct	MOV AX,OFFSET JUNK	AX JUNK % MOV
[BX+SI] – Indirect	MOV AX, [BX+SI]	AX [BX+SI] MOV
(BX+DI) – Indirect	MOV DH, [BX+DI]	DH (BX+DI) MOV
[BP+SI] – Indirect	SUB AX, [BP+SI]	AX [BP+SI] SUB
(BP+DI) – Indirect	MOV AX,(BP+DI)	AX (BP+DI) MOV
[SI] – Indirect	MOV BX, [SI]	BX [SI] MOV
(DI) – Indirect	MOV AX, [DI]	AX (DI) MOV
[BX] – Indirect	MOV AL, [BX]	AL [BX] MOV
[BX+SI+#] – Ind. w/ offset	ADD AH, [BX+SI+3]	AH 3 [BX+SI+#] ADD
(BX+DI+#) – Ind. w/ offset	MOV AX, [BX+SI+9]	AX 9 (BX+DI+#) MOV
[BP+SI+#] – Ind. w/ offset	ADD DL, [BP+SI+1]	DL 1 [BP+SI+#] ADD
(BP+DI+#) – Ind. w/ offset	ADD DL,(BP+DI+22)	DL 22 (BP+DI+#) ADD
[SI+#] – Ind. w/ offset	MOV BX, [SI+325]	BX 325 [SI+#] MOV
(DI+#) – Ind. w/ offset	ADD AX,(DI+2)	AX 2 (DI+#) ADD
[BP+#] – Ind. w/ offset	ADD AX, [BP+12]	AX 12 [BP+#] ADD
[BX+#] – Ind. w/ offset	MOV CL, [BX+6]	CL 6 [BX+#] MOV

Note that all indirect instructions that use DI use parentheses instead of square brackets. This is because of the 3 character and length method of naming words found in many Forths.

Using the Assembler to Define Words

CODE and NEXT are the assembler defining words. For example, to code a word to ADD 10 to the top number on the stack (let's call it ADD10):

```
CODE ADD10 AX POP AX 10 # ADD AX PUSH NEXT
```

CODE will create a word with the name that follows in the input stream (ADD10 in this case) and set the context vocabulary to ASSEMBLER. NEXT will reset the context vocabulary to Forth. Two other defining words are available: SUBROUTINE and END–SUB. SUBROUTINE can be used to create a word that can be called with an assembler CALL or the high–level word CALL. Words created with CODE and NEXT cannot be called in this fashion—they are to be used in high-level defining words, that is, within words created by colon. For example, here is a subroutine that will transfer the AX register to the BX, CX, and DX registers.

```
SUBROUTINE TAX BX AX MOV CX AX MOV DX AX MOVE END–SUB
```

Remember that in subroutines the return address is the top value on the stack. The word SUBROUTINE creates a subroutine with the name next in the input stream and sets the context vocabulary to ASSEMBLER. END–SUB resets the context vocabulary to Forth.

Structured Assembler Words

Our Forth 8088 assembler includes a number of structured constructs to make coding in assembler easier. For looping, the /BEGIN ... cond structure is provided with the following conditional branching keywords: JA, JNBE, JAE, JNB, JB, JNAE, JBE, JNA, JCXZ, JE, JZ, JG, JNLE, JGE, JNL, JL, JNGE, JLE, JNG, JNE, JNZ, JNO, JNP, JPO, JNS, JO, JP, JPE, JS, LOOP, LOOPE, LOOPZ, LOOPNE, LOOPNZ, and J. The code segment:

```
AH 0 MOV AL 8 # MOV /BEGIN [BX] AH MOV BX INC AL DEC JNE
```

would zero out eight bytes pointed to by the BX register. /BEGIN ... cond loops can be nested to any desired level, but all jumps must be contained with the −126 to +129 byte limit of the 8088 conditional branching opcodes.

Conditional branching constructs are also provided in the form of IF ... / ELSE ... /ENDIF. The forms of IF provided include: ZIF, NZIF, SIF, and NSIF. They can be used in the following fashion:

```
CODE TEST NZIF AX PUSH /ENDIF NEXT
```

TEST will push the AX register on the stack if the zero flag is not set. /ELSE is

also available. For example:

```
CODE TEST NZIF AX -1 # MOV /ELSE AX 0 # MOV /ENDIF NEXT
```

This version of TEST will push a true flag if the zero flag is not set and a false flag if it is. The /ELSE branch is also limited to jumps of −126 to +129 bytes.

The /BEGIN ... WHILE ... /REPEAT construct is also provided. The WHILE is replaced by an IF−type word. This code segment will move a string that is pointed to by the BX register to the memory that is pointed to by the SI register, assuming that the string is terminated by a zero byte.

```
/BEGIN AX [BX] MOV NZIF [SI] AX MOV SI INC BX INC /REPEAT
```

MACROS

Our 8088 assembler contains the word MACRO, which will begin the definition of a macro. Macros are essentially a sort of text substitution used during the definition of assembler words. In Forth, macros are more powerful, since they are words themselves and can do processing that is useful. Most often, however, macros are used when the same portion of code must be utilized at many places during the definition of words in assembler. A macro can be defined once for a sequence of instructions, and then the single macro can be used in the places where the entire sequence was desired. For instance, let's say that while writing some words in assembler you often need to take the top number off the stack, add a base address to it, and place it back on the stack. We could define a macro like this:

```
MACRO ADD−BASE AX POP AX BASE # ADD AX PUSH MEND
```

The word MEND ends macro definitions. ADD−BASE could be used anyplace in an assembler word in place of the code in the macro. For example:

```
CODE EXAMP ADD−BASE NEXT
```

would be exactly equivalent to

```
CODE EXAMP AX POP AX BASE # ADD AX PUSH NEXT
```

Macros are more powerful than just shorthand. For example, let's say we often take the top number off the stack and store it in memory. This macro would handle it:

```
MACRO −>MEM AX POP [] AX MOV MEND
```

This macro will expect a number on the stack when it is used and will generate code to store the top number on the stack at that address. If we wished to store the top number on the stack at memory location 800 (in the data segment)

```
CODE STOREAT800 800 ->MEM NEXT
```

would be exactly equivalent to

```
CODE STOREAT800 AX POP 800 % AX MOV NEXT
```

Not only do macros result in less text, but their proper use can result in much more readable assembler code.

NONSTRUCTURED CONSTRUCTS

The word FLABEL can be used to create nonstructured forward jumps by putting a zero on the stack before a branching conditional and FLABEL at the point of the desired forward branch. For example:

```
0 JZ AX POP AX POP FLABEL
```

This code segment would jump over the pulling of the top two words if the zero flag was set.

ASSEMBLER NOTES

Important: In Atila the following registers must be preserved by all assembler words: BP, DI, SI, CS, DS, ES, SS. The CS, DS, and ES registers all have the same value. Each version of Forth will have different requirements for which registers must be preserved and which may be destroyed. Please check the manual for the version of Forth you are using for this information.

In Atila the processor stack is used as the data stack. However, in some Forths the normal processor stack may not be the data stack. In this case, instructions like AX POP will not be able to be used for accessing the data stack. Again, check the documentation for your particular version of Forth.

When assembling a word, Forth is in interpretive mode and the context vocabulary is ASSEMBLER. Stack manipulation words, variables, and constants can all be used for address calculations. For instance, if you desired to access the third byte of the PAD in assembler, you could use the phrase PAD 3 + to obtain the desired address. Being in interpretive mode also means that if an error occurs, the word being assembled will not automatically be forgotten as it is when an error occurs in compile mode. You will find it necessary to

use the word FORGET to remove the definition from the dictionary.

In Atila, all numbers are stored on the stack as words. Double words have the most significant word most accessible.

Example:

This word will zero a byte in memory. It uses the top two words on the stack as a segment and offset, allowing access to all possible 8088 memory.

```
CODE 0SET-X
  DX ES MOV ( Save the ES register)
  ES POP ( Get the segment value)
  BX POP ( Get the address in the segment)
  ES SEG BX 0 # byte MOV ( Use extra segment, and zero the byte)
  ES DX MOV ( Restore the ES register)
NEXT
```

Suggested Extensions: The most useful extension of this assembler would be to include more thorough error checking for illegal or invalid instructions.

ASSEMBLER

Define a vocabulary for the assembler words.

```
( Define the vocabulary )
VOCABULARY ASSEMBLER

( Cause definitions to be entered into it)
ASSEMBLER DEFINITIONS HEX
```

1byte (N –) (–)

Define single-byte opcodes.

Stack on Entry: (Compile Time) N – Opcode value for instruction.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of use:

90 1byte NOP

Create the no operation instruction; when used, a 90 will be enclosed in the dictionary.

Algorithm: Store the value in the definition of the word. At run time, fetch that value and store it in the dictionary.

Suggested Extensions: None.

Definition:

```
: 1byte <BUILDS C, DOES> C@ C ,;

( Define all the single byte opcodes.)
37 1byte AAA 3F 1byte AAS 98 1byte CBW
F8 1byte CLC FC 1byte CLD FA 1byte CLI
F5 1byte CMC 99 1byte CWD 27 1byte DAA
2F 1byte DAS F4 1byte HLT CE 1byte INTO
CF 1byte IRET 9F 1byte LAHF
F0 1byte LOCK 90 1byte NOP
9D 1byte POPF 9C 1byte PUSHF
9E 1byte SAHF F9 1byte STC FD 1byte STD
FB 1byte STI 9B 1byte WAIT
D7 1byte XLAT F3 1byte REPE
F3 1byte REPZ F3 1byte REP
F2 1byte REPNE F2 1byte REPNZ
```

CODE EXAMPLE AX POP AAD CL 3 # MOV CL DIV AAM AX PUSH NEXT

EXAMPLE will divide a two-digit BCD number on the stack by three and return the result in BCD on the top of the stack.

Algorithm: This is a two-byte constant opcode; use comma to enclose its opcode in the dictionary.

Suggested Extensions: None.

Definition:

```
AAD AD5 , ;
```

AAD (–)

Assemble the adjust AX register for division (AAD) instruction.

Stack unaffected.

Example of Use:

```
CODE EXAM:E AX POP AAD CL 3 # MOV CL DIV AAM AX PUSH NEXT
```

EXAMPLE will divide a two-digit BCD number on the stack by three and return the result in BCD on the top of the stack.

Algorithm: This is a two-byte constant opcode; use comma to enclose its opcode in the dictionary.

Suggested Extensions: None.

Definition:

```
AAD AD5 , ;
```

AAM (–)

Assemble the adjust result of BCD multiplication (AAM) instruction.

Stack unaffected.

Definition:

```
: AAM AD4 , ;
```

byte (–)

Cause the next instruction to use a byte-length operand.

Stack unaffected.

Example of Use:

... byte [BX] 0400 MOV ...

This would assemble into a move of the byte from address 400 to the address held in the BX register. Without "byte" the MOV instruction would not be able to determine if a byte or word length move should be encoded.

Algorithm: Set a variable (b/w) that will be checked when the opcode is stored in memory. This allows byte to work with all the instructions.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE b/w
: byte b/w C0SET ;
```

cell (–)

Cause the next instruction to use a word-length operand.

Stack unaffected.

Definition:

```
: cell b/w C1SET ;
```

tipe (– A)

A two-cell array holding the addressing mode.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of "tipe".

Example of Use: See following definitions.

Algorithm: The variable will hold the addressing mode that words we will define shortly need to store. This information will be accessed by the actual opcode words to determine the final value to store in memory. This is a two-cell array, because there can be values for both the source and destination operand of an instruction.

Suggested Extensions: None.

Definition:

```
0 VARIABLE tipe 0 ,
```

value (– A)

A two-cell array holding address values.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of "value".

Example of Use: See following definitions.

Algorithm: The variable will hold the address values that words we will define shortly need to store. This information will be accessed by the actual opcode words to determine the final value to store in memory. This is a two-cell array, because there can be values for both the source and destination operand of an instruction.

Suggested Extensions: None.

Definition:

```
0 VARIABLE value 0 ,
```

f/s+ (N1 – N2)

Point storage of operands to the correct field; source or destination.

Stack on Entry: (N1) – The address of "tipe" or "value".

Stack on Exit: (N2) – The proper value for storing data in "tipe" or "value".

Example of Use: See following words.

Algorithm: When an instruction is being assembled, there is a source and a destination operand. This word uses the variable f/s to keep track of which is currently being addressed. It will increment the value on the stack by two if

the second operand is being addressed. This enables the storing of the proper values in the variables "tipe" and "value," which are described below.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE f/s
: f/s+ f/s C@ IF 2+ ENDIF ;
```

next+ (–)

Increment f/s to allow addressing of the next operand in an instruction.

Stack unaffected.

Definition:

```
: next+ f/s C@ NOT f/s C! ;
```

(N –)

Cause an immediate operand to be assembled.

Stack on Entry: (N) – The immediate value to be used.

Stack on Exit: Empty.

Example of Use:

```
... AL 255 # MOV ...
```

This code would assemble an instruction to move the number 255 into the AL register.

Algorithm: Store the number in "value"; store a 45 as the type of addressing in "tipe".

Suggested Extensions: None.

Definition:

```
: # value f/s+ ! 45 tipe f/s+ ! next+ ;
```

](N -)

Cause an indirect memory addressing operand to be assembled.

Stack on Entry: (N) – The address value to be used.

Stack on Exit: Empty

Example of Use:

```
... cell 800 ] 0 # MOV ...
```

This code would assemble an instruction that would move a zero word into memory address 800 (as referenced by the data segment).

Algorithm: Store the address in "value"; store a 6 as the type of addressing in "tipe".

Suggested Extensions: None.

Definition:

```
: ]6 tipe f/s+ ! value f/s+ ! next+ ;
```

reset (–)

Reset "tipe" for the start of a new instruction.

Stack unaffected.

Example of Use: See words defined below.

Algorithm: A 99 is stored in both places in "tipe" to mark it as not in use. Zero was not used because it is a legal value in the r/m field of the addressing mode byte.

Suggested Extensions: None.

Definition:

```
: reset 99 tipe ! 99 tipe 2+ ! f/s COSET ;
```

00mod (N –) (–)

Define a word that assembles as a 00 mod in the addressing mode byte.

Stack on Entry: (Compile Time) (N) The value to store in "tipe".
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: The values to be stored in "tipe" are equivalent to the values that will have to be stored in the R/M field of the addressing mode byte. At compile time, store the value to be fetched at run time. At run time, store that value in "tipe".

Suggested Extensions: None.

Definition:

```
: 00mod <BUILDS C, DOES> C@ tipe f/s+ !
   next+ ;

( Define all 00 mod addressing modes)
0 00mod [BX+SI]          1 00mod (BX+DI)
2 00mod [BP+SI]          3 00mod (BP+DI)
4 00mod [SI]             5 00mod (DI)
                         7 00mod [BX]
40 00mod ES              41 00mod CS
42 00mod SS              43 00mod DS
```

01mod (N –) (–)

Define a word that assembles as a 01 mod in the addressing mode byte.

Stack on Entry: (Compile Time) (N) The value to store in "tipe".
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: The values to be stored in "tipe" are equivalent to the values that will have to be stored in the r/m field of the addressing mode byte. At compile time, store the value to be fetched at run time. At run time, fetch that value. Since all 00 mod operands require an offset, check to see if that offset is word or byte in length. If it is a word, add eight to the value to be stored in "tipe". Store the value in "tipe," then store the offset in "value".

Suggested Extensions: None.

Definition:

```
: 01mod <BUILDS C, DOES> C@ OVER 256 U>
  IF 8 + ENDIF tipe f/s+ !
  value f/s+ ! next+ ;

  8 01mod [BX+SI+#]          9 01mod (BX+DI+#)
  10 01mod [BP+SI+#]         11 01mod (BP+DI+#)
  12 01mod [SI+#]            13 01mod (DI+#)
  14 01mod [BP+#]            15 01mod [BX+#]
```

11mod (N1 N2 -) (-)

Define a word that assembles as a 11 mod in the addressing mode byte.

Stack on Entry: (Compile Time) (N1) The byte or word value.
The value to store in "tipe".
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: Again the values to be stored in "tipe" are equivalent to the values that will have to be stored in the R/M field of the addressing mode byte. At compile time, we store both the byte or word value and the "tipe" value. At run time, these values are fetched and stored in "tipe" and "b/w".

Suggested Extensions: None.

Definition:

```
: 11mod <BUILDS C, C, DOES> DUP 1+ C@
  b/w C! C@ tipe f/s+ ! next+ ;
```

```
( Define the 11 mod instructions.)

0 24 11mod AL     1 24 11mod AX
0 25 11mod CL     1 25 11mod CX
0 26 11mod DL     1 26 11mod DX
0 27 11mod BL     1 27 11mod BX
0 28 11mod AH     1 28 11mod SP
0 29 11mod CH     1 29 11mod BP
0 30 11mod DH     1 30 11mod SI
0 31 11mod BH     1 31 11mod DI
```

direction (N1 – N2)

Set the direction bit in the opcode being formed.

Stack on Entry: (N1) The opcode.

Stack on Exit: (N2) The opcode with the direction bit set properly.

Example of Use: See words defined below.

Algorithm: Use the variable "dir" to hold the direction. Fetch the value from the dir variable and use "asl" to move it into the proper place in the instruction. (Bit one in this case).

Suggested Extensions: None.

Definition:

```
: asl 0 DO 2* LOOP ;
0 CVARIABLE dir
: direction dir C@ 1 asl OR ;
```

Size (N1 – N2)

Set the size bit in the opcode being formed.

Stack on Entry: (N1) The opcode.

Stack on Exit: (N2) The opcode with the size bit set properly.

Example of Use: See words defined below.

Algorithm: Use the variable "b/w", which holds the size of the current operand, byte or word. The size bit is bit zero in the operand.

Suggested Extensions: None.

Definition:

```
: size b/w C@ OR ;
```

r/m (N1 – N2)

Set the r/m field in the addressing mode byte being formed.

Stack on Entry: (N1) The addressing mode byte.

Stack on Exit: (N2) The addressing mode byte with the r/m field set properly.

Example of Use: See words defined below.

Algorithm: The variable "tipe" holds both the r/m and mod field information. Extract the r/m data from "tipe" and "OR" it into the addressing mode byte. The r/m field is the first three bits of the addressing mode byte, so no shifting is required.

Suggested Extensions: None.

Definition:

```
: r/m@ tipe @ 8 MOD ;
: r/m r/m@ OR ;
```

md (N1 – N2)

Set the mod field in the addressing mode byte being formed.

Stack on Entry: (N1) The addressing mode byte.

Stack on Exit: (N2) The addressing mode byte with the mod field set properly.

Example of Use: See words defined below.

Algorithm: The variable "tipe" holds both the r/m and mod field information. Extract the mod data from "tipe" and "OR" it into the addressing mode byte. Since the mod field is the two high bits of the addressing mode byte, it will be neccessary to shift the data 6 bits before "ORing" it.

Suggested Extensions: None.

Definition:

```
: md@ tipe @ 8 / ;
: md md@ 6 asl OR ;
```

,value (–)

Set the mod field in the addressing mode byte being formed.

Stack unaffected.

Example of Use: See words defined below.

Algorithm: Check the mod field to see if we are accessing a byte or word register. Also check for a memory access. Store the proper address from the variable "value" as a byte or word in the dictionary using the comma words.

Suggested Extensions: None.

Definition:

```
: ,value md@ 1 = IF
    value C@ C,
  ELSE
    md@ 2 = tipe @ 6 = OR IF
      value @ ,
    ENDIF
  ENDIF ;
```

reg@ (– N)

Get the register value stored in the first position of "tipe".

Stack on Entry: Empty.

Stack on Exit: (N) The register value stored in "tipe".

Example of Use: See words defined below.

Algorithm: Fetch the proper value and extract the first three bits.

Suggested Extensions: None.

Definition:

```
: 1reg@ tipe @ 8 MOD ;
```

reg@ (– N)

Get the register value stored in the current position of "tipe".

Stack on Entry: Empty.

Stack on Exit: (N) The register value stored in "tipe".

Example of Use: See words defined below.

Algorithm: Fetch the proper value and extract the first three bits.

Suggested Extensions: None.

Definition:

```
: reg@ tipe 2+ @ 8 MOD ;
```

1reg (N1 – N2)

Set the register field in the opcode being formed.

Stack on Entry: (N1) The opcode.

Stack on Exit: (N2) The opcode with the mod field set properly.

Example of Use: See words defined below.

Algorithm: Fetch the proper value and extract the first three bits. Shift the value three places to the left and "OR" it into the opcode.

Suggested Extensions: None.

Definition:

```
: 1reg 1reg@ 3 asl OR ;
```

reg (N1 – N2)

Set the register field in the addressing mode byte being formed.

Stack on Entry: (N1) The addressing mode byte.

Stack on Exit: (N2) The addressing mode byte with the mod field set properly.

Example of Use: See words defined below.

Algorithm: Fetch the proper value and extract the first three bits. Shift the value three places to the left and "OR" it into the addressing mode byte.

Suggested Extensions: None.

Definition:

```
: reg reg@ 3 asl OR ;
```

a–mode (N –)

Form and store an addressing mode byte. Store the instruction's address values.

Stack on Entry: (N) – The addressing mode byte base.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Set the mod and r/m fields, then store the byte. Use ",value" to store the address information.

Suggested Extensions: None.

Definition:

```
: a-mode md r/m C, ,value ;
```

16-bit-reg? (– B)

Is a 16 bit register being used?

Stack on Entry: Empty.

Stack on Exit: (B) Boolean value determining whether or not a 16-bit register is being used in this instruction.

Example of Use: See words defined below.

Algorithm: Check the value held in "tipe".

Suggested Extensions: None.

Definition:

```
: 16-bit-reg? tipe @ DUP 23 > SWAP 32 <
   AND b/w C@ AND ;
```

seg-reg? (– B)

Is a segment register being used?

Stack on Entry: Empty.

Stack on Exit: (B) Boolean value determining whether or not a segment register is being used in this instruction.

Example of Use: See words defined below.

Algorithm: Check the value held in "tipe".

Suggested Extensions: None.

Definition:

```
: seg-reg? tipe @ DUP 39 > SWAP 44 <
   AND ;
```

DEC

Assemble a decrement instruction.

Stack unaffected.

Example of Use:

```
... AX DEC ...
```

This code would assemble and decrement the AX register instruction.

Algorithm: Form the instruction by setting the base value and then using the words we have defined that fill in each field of an opcode. Reset "tipe" for the next instruction.

Suggested Extensions: None.

Definition:

```
: DEC 16-bit-reg? IF
      1reg@ 72 OR C,
   ELSE
      254 size C,
      8 a-mode
   ENDIF reset ;
```

INC

Assemble an increment instruction.

Stack unaffected.

Example of Use:

```
.. BX INC ...
```

This code would assemble and increment the BX register instruction.

Algorithm: Form the instruction by setting the base value and then using the words we have defined that fill in each field of an opcode. Reset "tipe" for the next instruction.

Suggested Extensions: None.

Definition:

```
: INC 16-bit-reg? IF
      1reg@ 64 OR C,
   ELSE
      254 size C,
      0 a-mode
   ENDIF reset ;
```

POP

Assemble a pop-the-stack instruction.

Stack unaffected.

Example of Use:

```
... DX POP ...
```

This code would assemble a pop of the DX register instruction.

Algorithm: Form the instruction by setting the base value and then using the words we have defined that fill in each field of an opcode. Reset "tipe" for the next instruction. The base value will depend on the type of register or memory being popped.

Suggested Extensions: None.

Definition:

```
: POP seg-reg? IF
```

```
    7 tipe @ 40 - 3 asl OR C,
  ELSE
    16-bit-reg? IF
      1reg@ 88 OR C,
    ELSE
      143 C, 0 a-mode
    ENDIF
  ENDIF reset ;
```

PUSH

Assemble a stack push instruction.

Stack unaffected.

Example of Use:

```
... AX PUSH ...
```

This code would assemble a push of the AX register instruction.

Algorithm: Form the instruction by setting the base value and then using the words we have defined that fill in each field of an opcode. Reset "tipe" for the next instruction. The base value will depend on the type of register or memory being popped.

Suggested Extensions: None.

Definition:

```
: PUSH seg-reg? IF
      6 tipe @ 40 - 3 asl OR C,
    ELSE
      16-bit-reg? IF
        1reg@ 80 OR C,
      ELSE
        255 C, 48 a-mode
      ENDIF
    ENDIF reset ;
```

ac,data? (- B)

Check if the operands for this instruction are a move of data into an accumulator.

Stack on Entry: Empty.

Stack on Exit: (B) – Flag, true if this is a move of data into an accumulator.

Example of Use: See words defined below.

Algorithm: Check the values stored in "tipe" by the operands.

Suggested Extensions: None.

Definition:

```
: ac,data? tipe @ 24 = tipe 2+ @ 45 =
  AND ;
```

immediate? (– B)

Check if there is an immediate operand for this instruction.

Stack on Entry: Empty.

Stack on Exit: (B) – Flag, true if this instruction has an immediate operand.

Example of Use: See words defined below.

Algorithm: Check the values stored in "tipe" by the second operand.

Suggested Extensions: None.

Definition:

```
: immediate? tipe 2+ @ 45 = ;
```

eb/w (N –)

Store a displacement or address into memory.

Stack on Entry: (N) The displacement or address to store.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Check the variable b/w to see if a byte or word should be stored in memory.

Suggested Extensions: None.

Definition:

```
: eb/w b/w C@ IF
    .
  ELSE
    C,
  ENDIF ;
```

,data (–)

Store the displacement or address for an instruction.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Use the second entry in the variable "value".

Suggested Extensions: None.

Definition:

```
: ,data value 2+ @ eb/w ;
```

s–bit (N1 – N2)

Set the s or sign Extension bit in an opcode.

Stack on Entry: (N1) The opcode.

Stack on Exit: (N2) The opcode with the s bit set.

Example of Use: See words defined below.

Algorithm: Determine if the immediate data can be represented in a single, signed byte. If it can, set the s bit, which is bit 1 in the opcode.

Suggested Extensions: None.

Definition:

```
: s-bit value 2+ @ DUP 128 < SWAP -128 > AND IF
    2 OR
  ENDIF ;
```

e-mode (-)

Form a complete addressing mode byte.

Stack unaffected.

Example of Use: See words defined below.

Algorithm: This word is used for opcodes that utilize a complete addressing mode byte. Place a zero on the stack, then use the md, reg, and r/m words to form the addressing mode byte. Store the addressing mode byte in memory and then store immediate or address data with ,value.

Suggested Extensions: None.

Definition:

```
: e-mode 0 md reg r/m C, ,value ;
```

swap-dir (-)

Mark an instruction's operands as moving data in the opposite direction.

Stack unaffected.

Example of Use: See words defined below.

Algorithm: Swap the data in the two positions of "tipe" and "value". Set the direction flag.

Suggested Extensions: None.

Definition:

```
: swap-dir
        tipe DUP @ SWAP 2+ @ tipe !
        tipe 2+ !
        value DUP @ SWAP 2+ @ value !
        value 2+ !
        dir C1SET ;
```

dir? (–)

Check the direction an instruction's operands are moving data. Swap the direction, if appropriate.

Stack unaffected.

Example of Use: See words defined below.

Algorithm: Moves to a register or from a segmentation register are valid; all others are suspect and should be switched. Check "tipe" to determine the course of action to be taken.

Suggested Extensions: Extend this word to check for illegal combinations of opcodes.

Definition:

```
: dir? tipe @ 23 > tipe 2+ @ DUP 39 >
   SWAP 44 < AND NOT AND IF
      swap-dir dir C1SET
   ELSE
      dir C0SET
   ENDIF ;
```

1kind (N1 N2 –) (–)

Assemble the arithmetic and logical instructions that use two operands.

Stack on Entry: (Compile Time) (N1) – The base opcode for an operation that will involve immediate data and the accumulator.

(Compile Time) (N2) – The base opcode for other kinds of operations.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: At compile time, store the base operands in the dictionary. At run time, determine the nature of the instruction and form the opcodes and addressing mode bytes accordingly. These type of instructions have a full range of addressing options and have the leftover 8080 opcode equivalents.

Suggested Extensions: Extend this word to check for illegal combinations of opcodes.

Definition:

```
: 1kind <BUILDS C, C, DOES>
  ac.data? IF
    C@ size C, ,data
  ELSE
    immediate? IF
      128 s-bit size C,
      1+ C@ a-mode ,data
    ELSE
      dir?
      1+ C@ direction size C, e-mode
    ENDIF
  ENDIF reset ;

  ( Define the logic and arithmetic opcodes.)

  0 4 1kind ADD                16 20 1kind ADC
  32 36 1kind AND              56 60 1kind CMP
  8 12 1kind OR                24 28 1kind SBB
  40 44 1kind SUB              48 52 1kind XOR
```

TEST (–)

Assemble the 8088 test instruction.

Stack unaffected.

Example of Use:

... byte 1 [BX+#] 4 TEST ...

This sequence would assemble a test of the byte pointed to by the BX register plus one.

Algorithm: Determine what operands are being used for this instruction and form the opcodes and addressing mode bytes accordingly. This instruction has a full range of addressing options and has a leftover 8080 opcode equivalent.

Suggested Extensions: Extend this word to check for illegal combinations of opcodes.

Definition:

```
: TEST ac,data? IF
      168 size C, ,data
   ELSE
      immediate? IF
         246 size C,
         0 a-mode ,data
      ELSE
         dir? 132 size C, e-mode
      ENDIF
   ENDIF reset ;
```

c-bit (N1 – N2)

Set the c bit for shift and rotate instructions.

Stack on Entry: (N1) The opcode.

Stack on Exit: (N2) The opcode with the c bit set appropriately.

Example of Use: See words defined below.

Algorithm: See if the CL register was specified as the second operand. If it was not, assume a single-bit shift or rotate.

Suggested Extensions: Extend this word to check for values in "tipe" for other than the CL register.

Definition:

```
: c-bit tipe 2+ @ 25 = IF 2 OR ENDIF ;
```

srkind (N –) (–)

Assemble the rotate and shift instructions.

Stack on Entry: (Compile Time) (N) – The base opcode for the instruction being defined.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: At compile time, store the base opcode in the dictionary. At run time, form the instruction by checking to see if the CL register specifies the number of bits to rotate.

Suggested Extensions: Extend this word to check for illegal combinations of opcodes.

Definition:

```
: srkind <BUILDS C, DOES>
  208 c-bit size C, C@ a-mode reset ;

  ( Define the shift and rotate instructions.)
  16 srkind RCL          24 srkind RCR
  0 srkind ROL           8 srkind ROR
  56 srkind SAR          32 srkind SHL
  32 srkind SHL          40 srkind SHR
```

LDS (–)

Assemble the 8088 load of a register and DS from memory instruction.

Stack unaffected.

Example of Use:

... AX [BX] LDS ...

This sequence would assemble a load of the AX and DS registers from the four bytes pointed to by the BX register.

Algorithm: Enclose the opcode constant in the dictionary, then form the addressing mode byte using e–mode.

Suggested Extensions: None.

Definition:

```
: LDS 197 C, dir? e–mode reset ;
```

LEA (–)

Assemble the 8088 load of a register with the effective address of an instruction.

Stack unaffected.

Example of Use:

... AX 6 [BX+SI+#] LEA ...

This sequence would assemble a load of the AX register with the sum of BX, SI, and six.

Algorithm: Enclose the opcode constant in the dictionary, then form the addressing mode byte using e–mode.

Suggested Extensions: None.

Definition:

```
: LEA 141 C, dir? e–mode reset ;
```

LES (–)

Assemble the 8088 load of a register and ES from memory instruction.

Stack unaffected.

Example of Use:

... DX 125 [BX+#] LES ...

This sequence would assemble a load of the DX and ES registers from the four bytes pointed to by the BX register plus 125.

Algorithm: Enclose the opcode constant in the dictionary, then form the addressing mode byte using e−mode.

Suggested Extensions: None.

Definition:

```
: LES 196 C, dir? e−mode reset ;
```

mkind (N −) (−)

Assemble the mathematical instructions.

Stack on Entry: (Compile Time) (N) − The base opcode for the instruction being defined.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: At compile time, store the base opcode in the dictionary. At run time, form the instruction by checking "tipe" to determine the addressing mode being used by the instruction.

Suggested Extensions: Extend this word to check for illegal combinations of opcodes.

Definition:

```
: mkind <BUILDS C, DOES>
  246 size C, C@ a−mode reset ;

  ( Define the mathematical opcodes)
```

```
48 mkind DIV          56 mkind IDIV
40 mkind IMUL         32 mkind MUL
24 mkind NEG          16 mkind Not
```

2kind (N –) (–)

Assemble the mathematical instructions.

Stack on Entry: (Compile Time) (N) – The base opcode for the instruction being defined.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: At compile time, store the base opcode in the dictionary. At run time, set the size bit in the opcode and store it.

Suggested Extensions: Extend this word to check for illegal combinations of opcodes.

Definition:

```
: 2kind <BUILDS C, DOES>
  C@ size C, reset ;

  ( Define the string opcodes.)

  166 2kind CMPS        172 2kind LODS
  164 2kind MOVS        174 2kind SCAS
  170 2kind STOS
```

IN (–)

Assemble the 8088 transfer of data from a port to accumulator instruction.

Stack unaffected.

Example of Use:

```
... AL 8 # IN ...
```

This sequence would assemble an instruction to transfer data from I/O port eight to the AL register.

Algorithm: Check the operands by looking at "tipe". Only accumulator and DX addressing are valid. Store the opcode in the dictionary.

Suggested Extensions: Extend this word to check for illegal combinations of operands.

Definition:

```
: IN tipe @ 26 = IF
     236 size C,
  ELSE
     228 size C, value 2+ @ C,
  ENDIF reset ;
```

OUT (–)

Assemble the 8088 transfer of data from the accumulator to a port instruction.

Stack unaffected.

Example of Use:

```
... DX AX OUT ...
```

This sequence would assemble an instruction to transfer data from the AX register to the I/O port specified by the value in the DX register.

Algorithm: Check the operands by looking at "tipe". Only accumulator and DX addressing are valid. Store the opcode in the dictionary.

Suggested Extensions: Extend this word to check for illegal combinations of operands.

Definition:

```
: OUT tipe @ 26 = IF
     238 size C,
  ELSE
     230 size C, value @ C,
  ENDIF reset ;
```

INT (–)

Assemble the 8088 software interrupt instruction.

Stack unaffected.

Example of Use:

```
... 10 # INT ...
```

This sequence would assemble a number ten interrupt.

Algorithm: Check the operands by looking at "tipe". Only immediate mode or implicit addressing is valid. Store the opcode in the dictionary.

Suggested Extensions: Extend this word to check for illegal combinations of operands.

Definition:

```
: INT tipe @ 45 = IF
     205 C, value @ C,
  ELSE
     204 C,
  ENDIF reset ;
```

SEG (–)

Assemble the 8088 segment override instruction.

Stack unaffected.

Example of Use:

```
... ES SEG AX [BX] MOV ...
```

This sequence would assemble an extra segment override instruction that would cause the effective address of the following MOV instruction to reference data in the extra segment instead of the data segment.

Algorithm: Check the operands by looking at "tipe". The segmentation registers (ES, CS, SS, and DS) are the only valid operands. Store the opcode in the dictionary.

Suggested Extensions: Extend this word to check for illegal operands.

Definition:

```
: SEG 38 tipe @ 40 - 3 asl OR C, reset ;
```

XCHG (-)

Assemble the 8088 exchange data instruction.

Stack unaffected.

Example of Use:

```
... AX BX XCHG ...
```

This sequence would assemble an exchange of the AX and BX registers instruction.

Algorithm: Determine what operands are being used for this instruction and form the opcodes and addressing mode bytes accordingly.

Suggested Extensions: Extend this word to check for illegal operands.

Definition:

```
: XCHG tipe @ DUP 23 > SWAP 32 < AND
  b/w C@ AND IF
     144 tipe @ 24 - OR C,
  ELSE
     swap-dir 134 size C, e-mode
  ENDIF reset ;
```

LONG-CALL (-)

Assemble the 8088 long (intersegment) subroutine call instruction.

Stack unaffected.

Example of Use:

```
... 50 0 LONG-CALL ...
```

This sequence would assemble a call to offset 50 in segment zero.

Algorithm: Determine what operands are being used for this instruction and form the opcodes and addressing mode bytes accordingly. If "tipe" is not set, assume the long address is on the stack.

Suggested Extensions: Extend this word to check for illegal operands.

Definition:

```
: LONG-CALL tipe @ 45 = IF
      255 C, 24 a-mode
   ELSE
      154 C, SWAP , ,
   ENDIF reset ;
```

CALL (–)

Assemble the 8088 short (intrasegment) subroutine call instruction.

Stack unaffected.

Example of Use:

```
... [BX] CALL ...
```

This sequence would assemble a call to the address in the BX register.

Algorithm: Determine what operands are being used for this instruction and form the opcodes and addressing mode bytes accordingly. If an immediate address is being called, determine the offset from the current address for inclusion in the assembled code.

Suggested Extensions: Extend this word to check for illegal operands.

Definition:

```
: CALL tipe @ 99 <> IF
      255 C, 16 a-mode
   ELSE
      HERE 232 C, - 3 - ,
   ENDIF reset ;
```

RET (–)

Assemble the 8088 short (intrasegment) subroutine return instruction.

Stack unaffected.

Example of Use:

```
... RET ...
```

This sequence would assemble a return instruction.

Algorithm: Check "tipe" for the addressing mode being used. Only immediate and implicit modes are valid. The immediate mode causes the stack pointer to be adjusted upon return from the subroutine.

Suggested Extensions: Extend this word to check for illegal operands.

Definition:

```
: RET tipe @ 45 = IF
      194 C, value @ ,
  ELSE
      195 C,
  ENDIF reset ;
```

LONG–RET (–)

Assemble the 8088 long (intrasegment) subroutine return instruction.

Stack unaffected.

Example of Use:

```
... 6 LONG–RET ...
```

This sequence would assemble a long return instruction that would also add 6 to the stack pointer.

Algorithm: Check "tipe" for the addressing mode being used. Only immediate and implicit modes are valid. The immediate mode causes the stack pointer to be adjusted upon return from the subroutine.

Suggested Extensions: Extend this word to check for illegal operands.

Definition:

```
: LONG-RET tipe @ 45 = IF
    202 C, value @ ,
  ELSE
    203 C ,
  ENDIF reset ;
```

LONG-BRANCH (-)

Assemble the 8088 long (intersegment) jump instruction.

Stack unaffected.

Example of Use:

```
... 500 0 LONG-BRANCH ...
```

This sequence would assemble a call to offset 500 in segment zero.

Algorithm: Determine what operands are being used for this instruction and form the opcodes and addressing mode bytes accordingly. If "tipe" is not set, assume the long address is on the stack.

Suggested Extensions: Extend this word to check for illegal operands.

Definition:

```
: LONG-BRANCH tipe @ 99 <> IF
    255 C, 40 a-mode
  ELSE
    234 C, SWAP ,,
  ENDIF ;
```

BRANCH (-)

Assemble the 8088 short (intrasegment) jump instruction.

Stack unaffected.

Example of Use:

... [BX] BRANCH ...

This sequence would assemble a jump to the address in the BX register.

Algorithm: Determine what operands are being used for this instruction and form the opcodes and addressing mode bytes accordingly. If an immediate address is being called, determine the offset from the current address for inclusion in the assembled code. Note that immediate addressing can have a byte or word offset.

Suggested Extensions: Extend this word to check for illegal operands.

Definition:

```
: BRANCH tipe @ 99 <> IF

    255 C, 32 a-mode
  ELSE
    HERE - DUP DUP 127 < SWAP -127 >
    AND IF
      235 C, 3 - C,
    ELSE
      233 C, 3 - ,
    ENDIF
  ENDIF reset ;
```

MOV (-)

Assemble the 8088 data move instruction.

Stack unaffected.

Example of Use:

... DX 14 [BX+SI+#] MOV ...

This sequence would assemble a move of the word found at the address specified by the sum of the BX register, SI register, and 14 to the DX register. The address is in the data statement.

Algorithm: Determine what operands are being used for this instruction and form the opcodes and addressing mode bytes accordingly. Note that MOV encompasses all the different addressing modes.

Suggested Extensions: Extend this word to check for illegal operands.

Definition:

```
: MOV dir? tipe @ 24 = tipe 2+ @ 6 = AND IF
    160 direction size C, value 2+ @ ,
  ELSE
    tipe 2+ @ DUP 40 >= SWAP 44 <= AND IF
      140 direction C, e-mode
    ELSE
      tipe @ 45 = tipe 2+ @ DUP 23 > SWAP 32 < AND AND IF
        176 tipe 2+ @ 24 - OR b/w C@ 3 asl
        OR C, value @ eb/w
      ELSE
        tipe 2+ @ 45 = IF
          198 size C, 0 a-mode ,data
        ELSE
          136 direction size C, e-mode
        ENDIF
      ENDIF
    ENDIF
  ENDIF reset ;
```

bopc (N –) (N –)

Assemble the 8088 branching opcodes.

Stack on Entry: (Compile Time) (N) – The base opcode for the instruction.
(Run Time) (N) The address to jump to or zero if a forward jump is being assembled.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: At compile time, store the base operands in the dictionary. At run time, check the stack for the address to jump to. If the address is zero, leave the address of the current instruction on the stack and set the length of the jump

to zero. If the address on the stack is not zero, determine the offset to that address and enclose it in the dictionary.

Suggested Extensions: Extend this word to check for branches out of range.

Definition:

```
HEX
: bopc <BUILDS C, DOES> C@ C, DUP 0= IF
      C, HERE
   ELSE
      HERE - 1 - C,
   ENDIF reset ;

   ( Assemble the branching opcodes )

   77 bopc JA        77 bopc JNBE      73 bopc JAE
   73 bopc JNB       72 bopc JB        72 bopc JNAE
   76 bopc JBE       76 bopc JNA       E3 bopc JCXZ
   74 bopc JE        74 bopc JZ        7F bopc JG
   7F bopc JNLE      7D bopc JGE       7D bopc JNL
   7C bopc JL        7C bopc JNGE      7E bopc JLE
   7E bopc JNG       75 bopc JNE       75 bopc JNZ
   71 bopc JNO       7B bopc JNP       7B bopc JPO
   79 bopc JNS       70 bopc JO        7A bopc JP
   7A bopc JPE       78 bopc JS        E2 bopc LOOP

E1 bopc LOOPE E1 bopc LOOPZ
E0 bopc LOOPNE E0 bopc LOOPNZ
EB bopc J
DECIMAL
```

FLABEL (A –)

Complete a forward branch instruction.

Stack on Entry: (A) The address the branch instruction is at.

Stack on Exit: Empty.

Example of Use:

```
... 0 JZ AH INC FLABEL ...
```

This sequence would assemble a jump around the increment AH instruction if the zero flag was set.

Algorithm: Determine the offset from the current position to the address of the branch on the stack. Fill in the length of the jump in the branch instruction.

Suggested Extensions: Extend this word to check for jumps out of range.

Definition:

```
: FLABEL 1- DUP HERE - 1+ ABS SWAP C! ;
```

ZIF (–)

Start an /IF construct, checking whether the zero flag is set.

Stack unaffected.

Example of Use:

```
... ZIF SKIPIT CALL /ENDIF ...
```

This sequence would assemble the instructions to call the subroutine SKIPIT if the zero flag is set.

Algorithm: Assemble a forward jump so that if the zero flag is not set (JNZ), /ELSE or /ENDIF will fill in.

Suggested Extensions: None.

Definition:

```
: ZIF 0 JNZ ;
```

NZIF (–)

Start an /IF construct, checking whether the zero flag is not set.

Stack unaffected.

Example of Use:

... NZIF SKIPIT CALL /ENDIF ...

This sequence would assemble the instructions to call the subroutine SKIPIT if the zero flag is not set.

Algorithm: Assemble a forward jump so that if the zero flag is set (JZ), /ELSE or /ENDIF will fill in.

Suggested Extensions: None.

Definition:

: NZIF 0 JZ ;

SIF (–)

Start an /IF construct, checking whether the sign flag is set.

Stack unaffected.

Example of Use:

... SIF SKIPIT CALL /ENDIF ...

This sequence would assemble the instructions to call the subroutine SKIPIT if the sign flag is set.

Algorithm: Assemble a forward jump so that if the sign flag is not set (JNS), /ELSE or /ENDIF will fill in.

Suggested Extensions: None.

Definition:

: SIF 0 JNS ;

NSIF (–)

Start an /IF construct, checking whether the sign flag is not set.

Stack unaffected.

Example of Use:

... NSIF SKIPIT CALL /ENDIF ...

This sequence would assemble the instructions to call the subroutine SKIPIT if the sign flag is not set.

Algorithm: Assemble a forward jump so that if the sign flag is set (JS), /ELSE or /ENDIF will fill in.

Suggested Extensions: None.

Definition:

: NSIF 0 JS ;

CIF (–)

Start an /IF construct, checking whether the carry flag is set.

Stack unaffected.

Example of Use:

... CIF SKIPIT CALL /ENDIF ...

This sequence would assemble the instructions to call the subroutine SKIPIT if the carry flag is set.

Algorithm: Assemble a forward jump so that if the carry flag is not set (JNB), /ELSE or /ENDIF will fill in.

Suggested Extensions: None.

Definition:

: CIF 0 JNB ;

NCIF (–)

Start an /IF construct, checking whether the carry flag is not set.

Stack unaffected.

Example of Use:

... NCIF SKIPIT CALL /ENDIF ...

This sequence would assemble the instructions to call the subroutine SKIPIT if the carry flag is not set.

Algorithm: Assemble a forward jump so that if the carry flag is set (JB), /ELSE or /ENDIF will fill in.

Suggested Extensions: None.

Definition:

: NCIF 0 JB ;

/BEGIN (–)

Mark the begining of a structured loop construct.

Stack unaffected.

Example of Use:

... /BEGIN READY? CALL JNZ ...

This sequence would assemble a loop of calling the subroutine READY? until the zero flag was set.

Algorithm: Leave the current dictionary pointer on the stack. This will be the address that the branching words will want to assemble a jump to.

Suggested Extensions: None.

Definition:

: /BEGIN HERE ;

/REPEAT (A1 A2 –)

Mark the end of a /BEGIN .. /IF .. /REPEAT construct.

Stack on Entry: (A1) – The address the /IF instruction's branch is at.
(A2) – The address the /BEGIN left on the stack.

Stack on Exit: Empty.

Example of Use:

```
... /BEGIN READY? CALL NZIF AL 0 # MOV 8 # AL OUT /REPEAT ...
```

This sequence would assemble the instructions to call the subroutine READY? and keep sending a zero out to port eight until READY? returns with the zero flag set.

Algorithm: First assemble the absolute jump back to the /BEGIN, then fill in the length byte of the branch instruction that can exit the loop.

Suggested Extensions: None.

Definition:

```
: /REPEAT SWAP BRANCH FLABEL ;
```

NEXT (–)

Mark the end of a word defined in assembler.

Stack unaffected.

Example of Use:

```
... CODE OUT AX POP DX POP DX AX OUT NEXT ...
```

This is a complete word that would cause a specific value to be sent out a specific port. NEXT ends the definition and can be thought of as the equivalent of ";" (semi).

Algorithm: Assemble a jump to the address in the BP register, which in Atila holds the address of the inner interpreter NEXT routine. Set the dictionary to be searched back to FORTH.

Suggested Extensions: None.

Definition:

```
: NEXT BP BRANCH FORTH ;
```

END–SUB (–)

Mark the end of a subroutine defined with the word SUBROUTINE.

Stack unaffected.

Example of Use:

```
... SUBROUTINE INC–BIG–NUM AX INC ZIF BX INC /ENDIF
END–SUB ...
```

This is a complete subroutine that increments a 32-bit number held in the AX and BX registers. END–SUB assembles a return instruction and ends the definition and can be thought of as the equivalent of NEXT or ";" (semi).

Algorithm: Assemble a short return instruction, then set the dictionary back to the normal FORTH for future definitions.

Suggested Extensions: None.

Definition:

```
: END–SUB RET FORTH DEFINITIONS ;
```

MEND (–)

Mark the end of a macro definition.

Stack unaffected.

Example of Use:

```
... MACRO 32INC DUP INC ZIF 2+ INC /ENDIF MEND ...
```

This is a complete macro that increments a 32-bit number at a specific memory address. MEND marks the end of the macro definition.

Algorithm: Compile the word semi and make FORTH both the search and define dictionaries.

Suggested Extensions: None.

Definition:

```
: MEND COMPILE ; FORTH DEFINITIONS ;
  IMMEDIATE
```

/ENDIF (A –)

Mark the end of a conditional branching construct.

Stack on Entry: (A) The address the branch instruction is at.

Stack on Exit: Empty.

Example of Use:

```
... NZIF AH INC /ENDIF ...
```

This sequence would assemble a jump around the increment AH instruction if the zero flag was set.

Algorithm: This word is the same as FLABEL and is used to improve readability of code.

Suggested Extensions: None.

Definition:

```
: /ENDIF FLABEL ;
```

/ELSE (A –)

Allow an alternative branch in a conditional branching construct.

Stack on Entry: (A) The address of the branch assembled by the /IF instruction.

Stack on Exit: Empty.

Example of Use:

```
... NZIF AH INC /ELSE AH 0 # MOV /ENDIF ...
```

This sequence would assemble an increment of the AH register if the zero flag is not set and a load of the AH register with zero if it is set.

Algorithm: Assemble an absolute forward jump for /ENDIF to fill in. Then fill in the length of the branch assembled by the /IF word.

Suggested Extensions: None.

Definition:

```
: /ELSE 0 J SWAP FLABEL ;
```

CODE (–)

Start the definition of a word coded in assembler.

Stack unaffected.

Example of Use:

```
... CODE OUT AX POP DX POP DX AX OUT NEXT ...
```

This is a complete word that would cause a specific value to be sent out a specific port.

Algorithm: Use CREATE to enter the name of the word in the dictionary. Set the vocabulary to be searched to ASSEMBLER.

Suggested Extensions: None.

Definition:

```
ATILA DEFINITIONS
: CODE CREATE ASSEMBLER ;
```

SUBROUTINE (–)

Start the definition of an assembly language subroutine.

Stack unaffected.

Example of Use:

```
... SUBROUTINE INC-BIG-NUM AX INC ZIF BX INC /ENDIF
END-SUB ...
```

This is a complete subroutine that increments a 32-bit number held in the AX and BX registers.

Algorithm: Make ASSEMBLER both the search and define dictionaries. Use <BUILDS DOES> to define a word that leaves its address on the stack when it is executed.

Suggested Extensions: None.

Definition:

```
: SUBROUTINE ASSEMBLER DEFINITIONS
  <BUILDS DOES> ;
```

MACRO (–)

Start the defintion of a macro.

Stack unaffected.

Example of use:

```
... MACRO 32INC DUP INC ZIF 2+ INC /ENDIF MEND ...
```

This is a complete macro that increments a 32-bit number at a specific memory address. MEND marks the end of the macro definition.

Algorithm: Use ":" (colon) to define a word. Make ASSEMBLER both the search and define dictionaries.

Suggested Extensions: None.

Definition:

```
: MACRO ASSEMBLER DEFINITIONS : ;
```

6

8087 Numerical Coprocessor

Words Defined in This Chapter:

mftype	Define a word to store a value in b/w.
SHORT_REAL	Define the memory format of the next instruction to be 8087 short real.
SHORT_INTEGER	Define the memory format of the next instruction to be 8087 short integer.
LONG_REAL	Define the memory format of the next instruction to be 8087 long real.
WORD_INTEGER	Define the memory format of the next instruction to be 8087 word integer.
TEMP_REAL	Define the memory format of the next instruction to be 8087 temporary real.
LONG_INTEGER	Define the memory format of the next instruction to be 8087 long integer.
BCD	Define the memory format of the next instruction to be 8087 BCD.
ST	Store a value in "tipe" indicating a 8087 register is being used.
set-st	Set the register number field in the addressing mode byte being formed, and store the addressing mode byte.

st?	Determine whether a 8087 register has been specified.
1cell	Define single-word 8087 opcodes.
FCOMPP	Assemble the 8087 compare and pop twice instruction.
FTST	Assemble the 8087 test top stack value instruction.
FXAM	Assemble the 8087 examine the top stack value instruction.
ZFLD	Assemble the 8087 load-zero instruction.
1FLD	Assemble the 8087 load-one instruction.
PIFLD	Assemble the 8087 load-pi instruction.
L2TFLD	Assemble the 8087 log 2 (10) instruction.
L2EFLD	Assemble the 8087 log 2 (e) instruction.
LG2FLD	Assemble the 8087 log 10 (2) instruction.
LN2FLD	Assemble the 8087 log e (2) instruction.
FSQRT	Assemble the 8087 square root instruction.
FSCALE	Assemble the 8087 scale instruction.
FPREM	Assemble the 8087 remainder instruction.
FRNDINT	Assemble the 8087 round to integer instruction.
FXTRACT	Assemble the 8087 extract mantissa and exponent instruction.
FABS	Assemble the 8087 absolute value instruction.
FCHS	Assemble the 8087 change sign instruction.
FPTAN	Assemble the 8087 tangent instruction.
FPATAN	Assemble the 8087 arctangent instruction.
F2XM1	Assemble the 8087 2∧ST(0)−1 instruction.
FYL2X	Assemble the 8087 ST(1)*log 2(St(0)) instruction.
FYL2XPI	Assemble the 8087 ST(1)*Log 2(ST(0)+1) instruction.
FINIT	Assemble the 8087 initialize instruction.
FENI	Assemble the 8087 enable interrupts instruction.
FDISI	Assemble the 8087 disable interrupts instruction.
FCLEX	Assemble the 8087 clear exceptions instruction.
FINCSTP	Assemble the 8087 increment stack pointer instruction.
FDECSTP	Assemble the 8087 decrement stack pointer instruction.
FNOP	Assemble the 8087 no operation instruction.
START8087	Initialize the 8087 and clear all interrupts.

F@	Fetch a floating-point value.
F!	Store a floating-point value.
FVARIABLE	Define and initialize a floating-point variable.
FCONSTANT	Define a floating-point constant.
F+	Add two floating-point numbers.
F-	Subtract two floating-point numbers.
F*	Multiply two floating-point numbers.
F/	Divide two floating-point numbers.
FNEGATE	Reverse the sign of a floating-point number.
FP->INT	Convert a floating-point number to an integer.
INT->FP	Convert an integer to a floating-point number.
FABS	Find the absolute value of a floating-point number.
F*10	Multiply a floating-point number by 10.
F/10	Divide a floating-point number by 10.
F=	Compare two floating-point numbers, checking for equality.
F0=	Compare a floating-point number to zero.
F<	Compare two floating-point numbers, checking for a less than condition.
F2DUP	Duplicate the top two floating-point numbers on the stack.
F>	Compare two floating-point numbers, checking for a greater than condition.
F<=	Compare two floating-point numbers, checking for a less than or equal condition.
F<>	Compare two floating-point numbers, checking for inequality.
FP->DINT	Convert a floating-point number to a double-length integer.
DINT->FP	Convert a double-length integer to a floating-point number.
FTRUNC	Truncate a floating-point number.
F.R	Print a floating-point number in normal form within a specified field.
F.	Print a floating-point number in normal form.
FE.R	Print a floating-point number in scientific notation, within a specified field.
FE.	Print a floating-point number in scientific notation.
R.	Print a floating-point number.

FNUM	Convert a string to a floating-point number.
SQRT	Calculate the square root of a floating-point number.
LN	Calculate the natural logarithm of a floating-point number.
LOG	Calculate the base 10 logarithm of a floating-point number.
2∧X	An assembler subroutine to calculate two to an arbitrary power.
EXP	Calculate the value of e raised to a floating-point number.
∧	Calculate the exponentiation of one floating-point number to another.
[FPTAN]	An assembler subroutine to execute 8087 FPTAN instruction on an arbitrary number.
TAN	Calculate the tangent of a floating-point number.
2FLD	A macro to push a two onto the 8087 stack.
F/2	A macro to divide the top 8087 stack entry by two.
SIN	Calculate the sine of a floating-point number.
COS	Calculate the cosine of a floating-point number.
1/X	Calculate the multiplicative identity of a floating-point number.
COT	Calculate the cotangent of a floating-point number.
CSC	Calculate the cosecant of a floating-point number.
SEC	Calculate the secant of a floating-point number.
[FPATAN]	An assembler subroutine to execute the 8087 FPATAN instruction on an arbitrary number.
ATAN	Calculate the arctangent of a floating-point number.
ACOTN	Calculate the arccotangent of a floating-point number.
ASIN	Calculate the arcsine of a floating-point number.
ACOS	Calculate the arccosine of a floating-point number.
ASEC	Calculate the arcsecant of a floating-point number.
ACSC	Calculate the arccosecant of a floating-point number.

FSIGN	Calculate the sign of a floating-point number.
SINH	Calculate the hyperbolic sine of a floating-point number.
COSH	Calculate the hyperbolic cosine of a floating-point number.
TANH	Calculate the hyperbolic tangent of a floating-point number.
SECH	Calculate the hyperbolic secant of a floating-point number.
CSCH	Calculate the hyperbolic cosecant of a floating-point number.
COTNH	Calculate the hyperbolic cotangent of a floating-point number.
ASINH	Calculate the inverse hyperbolic sine of a floating-point number.
ACOSH	Calculate the inverse hyperbolic cosine of a floating-point number.
ATANH	Calculate the inverse hyperbolic tangent of a floating-point number.
ASECH	Calculate the inverse hyperbolic secant of a floating-point number.
ACSCH	Calculate the inverse hyperbolic cosecant of a floating-point number.
ACOTNH	Calculate the inverse hyperbolic cotangent of a floating-point number.

FLOATING-POINT NUMBERS

This chapter will present a complete set of Forth words to make use of the 8087 numerical coprocessor. This chip must be available for the floating-point routines described in this book to function. The words presented include almost every possible function, from simple addition to the esoteric inverse hyperbolic cosecant. The power of the 8087 will be made part of our Forth by these words.

The 8087 numeric coprocessor is an extremely powerful and complicated chip. It is designed to function with the 8088 CPU of our IBM-PC and extend its instruction set to deal with floating-point numbers. Because the 8087 can

only be accessed from the machine level, we will have to use the assembler we defined in the previous chapter for much of our work with the 8087. Our first step will be to extend the assembler defined in this chapter to include the 8087 instructions. From this base we will be able to construct our elementary floating-point functions, like addition and subtraction.

Once we have written the basic arithmetic instructions, we will be able to write floating-point input and output words. These will enable us to deal with floating-point numbers as part of the language. At this point, the rest of the floating-point package could be considered optional. The next set of words will be mathematical functions, including logarithms, exponentials, and hyperbolic and transcendental functions. These can be used as the needs of your particular application demand.

INSIDE THE 8087

The words in this chapter can be used without any knowledge of the 8087 chip. They are totally self-contained and handle all of the interaction with the numeric coprocessor. But knowing about the machine you are using is always beneficial. The 8087 has eight internal floating-point registers, each 80 bits long. The registers are organized into a stack, and most operations function by pushing and popping from the stack of registers. In Intel nomenclature, the top of the stack is known as ST(0) or just ST. The next number on the stack is referenced by ST(1), and so on up to ST(7).

The 8087 can handle seven types of numbers, or data formats. They are:

Name	Number of Bits	Range
Word Integer	16	−32768 −>32767
Short Integer	32	−2,147,483,648 −> 2,147,483,647
Long Integer	64	−9,223,372,032,759,841,344 −> 9,223,372,032,759,841,343
Packed Decimal	80	−999,999,999,999,999,999 −> 999,999,999,999,999,999
Short Real	32	−8.43 E−37 −> 3.37e38
Long Real	64	−4.19 E−307 −> 1.67 E308
Temporary Real	80	−3.4 E−4932 −> 1.2 e4932

As can be seen from the table, the 8087 can use a wide range of numbers.

All operations that take place on the chip itself are in temporary real format.

The 8087's instruction set is also wide ranging. It includes pops and pushes to the 8087 stack, basic math instructions like addition and subtraction, specialized math instructions for calculating common functions, and any number of processor control instructions. The 8087 has a control register that specifies how it deals with rounding, error conditions, and interrupts. The 8087 also has a status word, equivalent to the flag register on the 8088, that will hold the result of logical operations on numbers.

The 8088 and 8087 must operate in unison to be useful. The 8088 has a number of instructions that facilitate this interaction. The escape or ESC instruction is used to control the 8087. The 8088 WAIT instruction is used to make sure the two chips are synchronized. The ESC instruction has three unused bits. These bits, along with fields in the addressing mode byte, are used by the 8087 to decode its instructions. When the 8088 executes an ESC instruction, it does nothing. The 8087 is usually watching the 8088 execute its instructions. When the 8087 sees an ESC instruction, it takes over and decodes the instruction. After the 8087 decodes the instruction, it starts processing. While executing an instruction, the 8087 is not watching what the 8088 is doing. If another ESC instruction was encountered by the 8088 during this time, the 8087 would miss it. The 8088 WAIT instruction will make sure no instructions are missed, by causing the 8088 to stop processing until the 8087 is ready to perform another operation and is back watching the 8088 instruction fetch. The WAIT instruction is also necessary when the 8088 and 8087 are accessing the same memory location. The 8088 should WAIT until the 8087 has completed an instruction before using shared memory locations. Otherwise, the 8088 has no way of knowing when the 8087 is finished with the memory.

Additions to the 8088 Macro Assembler

The 8087 extensions to the assembler have the following new addressing modes:

Addressing Mode	Normal	Forth
Stack	FADD ST ST(3)	0 ST 3 ST FADD
Stack and pop	FADDP ST(3),ST	3 ST 0 ST FADD ANDPOP
Stack, reversed	FSUBR ST,ST(1)	0 ST 1 ST FSUB REVERSE

The following data identifiers are used in addressing memory:

Format	Identifier	Example
Short real	SHORT_REAL	SHORT_REAL [BX] FLD
Short integer	SHORT_INTEGER	SHORT_INTEGER 2 [BX+#] FADD
Long real	LONG_REAL	LONG_REAL 0 [BP+#] FSTP
Word integer	WORD_INTEGER	WORD_INTEGER [BX] FDIV REVERSE
Temporary real	TEMP_REAL	TEMP_REAL Holdme] FSTP
Long integer BCD	LONG_INTEGER BCD	LONG_INTEGER [SI] FLD BCD FinalTotal] FSTP

Suggested Extensions: The words in this chapter store all numbers in short real format. This gives six or seven digits of accuracy. If increased accuracy is desired, the words could be modified to use the long real or even temporary real formats for more precision. These words also have no provision for using the 8087 exceptions for overflow, underflow, divide by zero, etc. If these conditions are important in your application, the 8087 control register would need to be set to notify the 8088 of these conditions, and exception handling words would need to be written.

mftype (N –) (–)

Define a word to store a value in b/w.

Stack on Entry: (Compile Time) (N) – Value to store in b/w.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: At compile time, store the value in the dictionary. At run time, fetch the value and store it in b/w.

Suggested Extensions: None.

Definition:

ASSEMBLER DEFINITIONS

```
: mftype <BUILDS C, DOES> C@ b/w C! ;

( Define the types we need.)
2 mftype SHORT_REAL
3 mftype SHORT_INTEGER
4 mftype LONG_REAL
5 mftype WORD_INTEGER
6 mftype TEMP_REAL
7 mftype LONG_INTEGER
8 mftype BCD
```

mf (N –)

Build the memory format field in the opcode being assembled, and store the opcode.

Stack on Entry: (N) The opcode being assembled.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Take the memory format from b/w. Subtract two to get to the desired range. Shift it one bit to the left to place it in the proper position and OR it into the opcode. Store the opcode.

Suggested Extensions: None.

Definition:

```
: mf b/w C@ 2- 1 asl OR C, ;
```

ST (N –)

Store a value in "tipe," indicating an 8087 register is being used.

Stack on Entry: (N) The register being used.

Stack on Exit: Empty.

Example of Use:

... 1 ST FXCH ...

Assemble the 8087 instruction that exchanges the top two registers on the 8087 register stack.

Algorithm: Add 50 to the value and store it in the current position of "tipe"; increment the current position.

Suggested Extensions: Add error checking to make sure only values of zero to seven are specified.

Definition:

```
: ST 50 + tipe f/s+ I next+ ;
```

set–st (N –)

Set the register number field in the addressing mode byte being formed, and store the addressing mode byte.

Stack on Entry: (N) The addressing mode byte being formed.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Fetch the value from "tipe" and subtract 50, OR the number into the addressing mode byte, and store it in the dictionary.

Suggested Extensions: None.

Definition:

```
: set–st tipe @ 50 – OR C, ;
```

st? (– B)

Determine whether an 8087 register has been specified.

Stack on Entry: Empty.

Stack on Exit: (B) Boolean flag, true if an 8087 register has been specified.

Example of Use: See words defined below.

Algorithm: Fetch the value from "tipe," and see if it lies in the range of valid registers (0–7).

Suggested Extensions: None.

Definition:

```
: st? tipe @ DUP 50 >= SWAP 57 <= AND ;
```

1cell (N –) (–)

Define a word to assemble one word opcodes.

Stack on Entry: (Compile Time) (N) – The opcode to assemble.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: At compile time, store the value in the dictionary. At run time, fetch the value and store it in the dictionary. Assemble a WAIT instruction before every opcode.

Suggested Extensions: None.

Definition:

```
: 1cell <BUILDS , DOES> WAIT @ , reset ;

( Define all the 8087 one word opcodes)

-9762 1cell FCOMPP        -6951 1cell FTST
-6695 1cell FXAM          -4391 1cell ZFLD
-5927 1cell 1FLD          -5159 1cell PIFLD
-5671 1cell L2TFLD        -5415 1cell L2EFLD
-4903 1cell LG2FLD        -4647 1cell LN2FLD
-1319 1cell FSQRT         -551 1cell FSCALE
-1831 1cell FPREM         -807 1cell FRNDINT
```

```
-2855 1cell FXTRACT
-7975 1cell FCHS
-3111 1cell FPATAN
-3623 1cell FYL2X
-7205 1cell FINIT
-7717 1cell FDISI
-2087 1cell FINCSTP
-12071 1cell FNOP
-7719 1cell FABS
-3367 1cell FPTAN
-3879 1cell F2XM1
-1575 1cell FYL2XPI
-7973 1cell FENI
-7461 1cell FCLEX
-2343 1cell FDECSTP
```

esc-only (N1 N2 -) (-)

Define a word to define words that will assemble opcodes that have normal addressing mode bytes.

Stack on Entry: (Compile Time) (N1) – The portion of the opcode in the ESC instruction.
(N2) – The portion of the opcode in the addressing mode byte.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: At compile time, store the values in the dictionary. At run time, fetch the values and form the ESC instruction. Form the addressing mode byte normally, adding in the portion of the opcode that belongs in it. Assemble a WAIT instruction before every opcode.

Suggested Extensions: None.

Definition:

```
216 CCONSTANT esc

: esc-only <BUILDS SWAP C, C, DOES>
   WAIT DUP C@ esc OR C,
   1+ C@ a-mode reset ;

1 40 esc-only FLDCW
1 56 esc-only FSTCW
5 56 esc-only FSTSTATUSW
1 48 esc-only FSTENV
```

```
1 32 esc-only FLDENV
5 48 esc-only FSAVE
5 32 esc-only FRSTOR
```

dset (N1 – N2)

Set the destination bit in the opcode being assembled.

Stack on Entry: (N1) The opcode being assembled.

Stack on Exit: (N2) The opcode with the destination bit set.

Example of Use: See words defined below.

Algorithm: Fetch the value from "tipe," and see if the destination is 8087 register zero. If it is, swap the values in "tipe". If it is not, set the direction bit, indicating a move to a register other than zero.

Suggested extensions: None.

Definition:

```
: dset tipe @ 50 = IF
     swap-dir
  ELSE
     4 OR
  ENDIF ;
```

ANDPOP (–)

Set the pop bit in the instruction last assembled.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
... 1 ST 0 ST FADD ANDPOP
```

This would assemble an 8087 instruction to add the top two 8087 registers, store the result in ST(1) and discard in ST(0).

Algorithm: Get the address of the last instruction from the variable rev-ad. Set bit two in the byte at that address, the pop bit in an 8087 instruction.

Suggested Extensions: None.

Definition:

```
0 VARIABLE rev-ad
: ANDPOP rev-ad @ DUP C@ 2 OR SWAP CI ;
```

REVERSE (-)

Set the reverse bit in the instruction last assembled.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
... 1 ST 0 ST FSUB ANDPOP REVERSE ...
```

This would assemble an 8087 instruction to subtract register zero from register one, store the result in ST(1) and discard in ST(0).

Algorithm: Get the address of the last instruction from the variable rev-ad. Set bit four in the byte at that address plus one, the reverse bit in an 8087 instruction.

Suggested Extensions: None.

Definition:

```
: REVERSE rev-ad @ 1+ DUP C@ 8 OR
   SWAP C! ;
```

FXCH (-)

Assemble the 8087 register exchange instruction.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

... 3 ST FXCH ...

This would assemble an 8087 instruction to exchange 8087 registers ST(3) and ST(0).

Algorithm: Set the opcode and then fill in the register field, store the result in the dictionary. Assemble a WAIT instruction before the opcode.

Suggested Extensions: Add error checking for invalid operands.

Definition:

```
: FXCH
   WAIT 217 C, 200 set-st reset ;
```

FST (–)

Assemble the 8087 store instruction.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

... SHORT_REAL TOTAL] FST ...

This would assemble an 8087 instruction to move the value in ST(0) to the 32 bits at the memory address TOTAL, in short real format. The 8087 stack is undisturbed.

Algorithm: Determine if the instruction references the stack or memory and assemble accordingly. Assemble a WAIT instruction before the opcode.

Suggested Extensions: Add error checking for invalid operands.

Definition:

```
: FST
   WAIT st? IF
      221 C, 208 set-st
   ELSE
      217 mf 24 a-mode
   ENDIF reset ;
```

fkind (N1 N2 N3 –) (–)

Define a word to define words that will assemble the primary floating-point instructions.

Stack on Entry: (Compile Time) (N1) – One if the reverse flag is part of the instruction.
(N2) – The portion of the opcode in the ESC instruction.
(N3) – The portion of the opcode in the addressing mode byte.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) Empty.

Example of Use: See words defined below.

Algorithm: At compile time, store the values in the dictionary. At run time, fetch the values and form the ESC instruction. Form the addressing mode byte normally, adding in the portion of the opcode that belongs in it. If the reverse bit can be set, call dset. Assemble a WAIT instruction before every opcode.

Suggested Extensions: Add error checking for invalid operands.

Definition:

```
: fkind <BUILDS LROT SWAP C, C, C, DOES>
      WAIT HERE rev-ad ! st? IF
      216 OVER C@ IF
         dset
      ENDIF
      C, 1+ C@ set-st
   ELSE
      216 mf 2+ C@ a-mode
   ENDIF reset ;

( Define the primary 8087 instructions)
0 208 16 fkind FCOM
0 216 24 fkind FCOMP
1 192 0 fkind FADD
1 224 32 fkind FSUB
1 200 8 fkind FMUL
1 240 48 fkind FDIV
```

FSTP (–)

Assemble the 8087 store and pop instruction.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

... SHORT_REAL TOTAL] FST ...

This would assemble an 8087 instruction to move the value in ST(0) to the 32 bits at the memory address TOTAL, in short real format. ST(0) would then be removed from the 8087 stack.

Algorithm: Determine if the instruction references the stack or memory and assemble accordingly. The special cases of the long integer, temporary real, and BCD formats must be handled. Assemble a WAIT instruction before the opcode.

Suggested Extensions: Add error checking for invalid operands.

Definition:

```
: FSTP
  WAIT st? IF
    221 C, 216 set-st
  ELSE
    b/w C@ 6 < IF
      217 mf 24 a-mode
    ELSE
      b/w C@ 7 = IF
        223 C, 56 a-mode
      ELSE
        b/w C@ 6 = IF
          219 C, 40 a-mode
        ELSE
          223 C, 48 a-mode
  ENDIF ENDIF ENDIF ENDIF reset ;
```

FLD (–)

Assemble the 8087 load instruction.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
... SHORT_REAL TOTAL ] FLD ...
```

This would assemble an 8087 instruction to push the value at the 32 bits at the memory address TOTAL, in short real format, to the 8087 stack.

Algorithm: Determine if the instruction references the stack or memory and assemble accordingly. The special cases of the long integer, temporary real, and BCD formats must be handled. Assemble a WAIT instruction before the opcode.

Suggested Extensions: Add error checking for invalid operands.

Definition:

```
: FLD
  WAIT st? IF
    217 C, 192 set-st
  ELSE
    b/w C@ 6 < IF
      217 mf 0 a-mode
    ELSE
      b/w C@ 7 = IF
        223 C, 40 a-mode
      ELSE
        b/w C@ 6 = IF
          219 C, 32 a-mode
        ELSE
          223 C, 32 a-mode
  ENDIF ENDIF ENDIF ENDIF reset ;
```

START8087 (-)

Initialize the 8087.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

CODE PROGRAM START8087 ...

This would initialize the 8087 and clear its stack. Interrupts are disabled.

Algorithm: Assemble the FINIT, and FDISI instructions.

Suggested Extensions: None.

Definition:

```
CODE START8087 FINIT FDISI NEXT
```

F@ (A – F)

Fetch a floating-point variable.

Stack on Entry: (A) The address of the floating-point value.

Stack on Exit: (F) The floating-point number at A.

Example of Use:

```
... balance F@ R. ...
```

This code fragment would fetch the value of the variable balance and print it on the display.

Algorithm: floating-point numbers are 32 bits long. Use two 16-bit fetches to place the number on the stack.

Suggested Extensions: None.

Definition:

```
: F@ DUP @ SWAP 2+ @ SWAP ;
```

F! (F A –)

Store a floating-point variable.

Stack on Entry: (F) A floating-point number.
(A) The address of the floating-point value.

Stack on Exit: (F) The floating-point number at A.

Example of Use:

```
... Sum F@ 1.5 Sum F! ...
```

This code fragment would add 1.5 to the value held in Sum.

Algorithm: Floating-point numbers are 32-bits long. Use two 16-bit stores to place the number in memory.

Suggested Extensions: None.

Definition:

```
: F! DUP LROT ! 2+ ! ;
```

FVARIABLE (F –) (F – A)

Define a floating-point variable.

Stack on Entry: (Define Time) (F) The initial value of the variable.
(Run Time) Empty.

Stack on Exit: (Define Time) Empty.
(A) The address of the variable.

Example of Use:

```
... Sum F@ 1.5 F+ Sum F! ...
```

This code fragment would add 1.5 to the value held in Sum.

Algorithm: Allocate 4 bytes at define time and store the value found on the stack. At run time, leave the address of the variable.

Suggested Extensions: None.

Definition:

```
: FVARIABLE <BUILDS HERE 4 ALLOT F!
   DOES> ;
```

FCONSTANT (F –) (– F)

Define a floating-point constant.

Stack on Entry: (Define Time) (F) The value of the constant.
(Run Time) Empty.

Stack on Exit: (Define Time) Empty.
(F) The value of the constant.

Example of Use:

```
3.14157 FCONSTANT PI
```

The constant pi.

Algorithm: Allocate 4 bytes at define time and store the value found on the stack. At run time, fetch the value stored in memory.

Suggested Extensions: None.

Definition:

```
: FCONSTANT <BUILDS HERE 4 ALLOT F!
  DOES> F@ ;
```

F+ (F1 F2 – F3)

Leave F3, the floating-point sum of F1 and F2.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (F3) The sum of F1 and F2.

Example of Use:

```
... Sum F@ 1.5 F+ Sum F! ...
```

This code fragment would add 1.5 to the value held in Sum.

Algorithm: Move both numbers from the 8088 data stack to the 8087. Decrement the stack with the pops. Add the numbers on the 8087, then move the result back to the 8088 data stack.

Suggested Extensions: None.

Definition:

```
CODE F+
   AX BP MOV
   BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   SHORT_REAL 4 [BP+#] FADD
   BX POP BX POP
   SHORT_REAL 4 [BP+#] FSTP
   WAIT BP AX MOV
NEXT
```

F− (F1 F2 − F3)

Leave F3, the floating-point difference of F1 minus F2.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (F3) The difference of F1 minus F2.

Example of Use:

```
... Sum F@ PI F− Sum FI ...
```

This code fragment would subtract pi from the value held in Sum.

Algorithm: Move both numbers from the 8088 data stack to the 8087. Decrement the stack with the pops. Subtract the numbers on the 8087, then move the result back to the 8088 data stack.

Suggested Extensions: None.

Definition:

```
CODE F−
   AX BP MOV
   BP SP MOV
   SHORT_REAL 4 [BP+#] FLD
   SHORT_REAL 0 [BP+#] FADD
   BX POP BX POP
   SHORT_REAL 4 [BP+#] FSTP
   WAIT BP AX MOV
NEXT
```

F* (F1 F2 − F3)

Leave F3, the floating-point product of F1 times F2.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (F3) The product of F1 times F2.

Example of Use:

... Sum F@ 10. F * Sum F! ...

This code fragment would multiply the value held in Sum by ten.

Algorithm: Move both numbers from the 8088 data stack to the 8087. Decrement the stack with the pops. Multiply the numbers on the 8087, then move the result back to the 8088 data stack.

Suggested Extensions: None.

Definition:

```
CODE F*
   AX BP MOV
   BP SP MOV
   SHORT_REAL 4 [BP+#] FLD
   SHORT_REAL 0 [BP+#] FADD
   BX POP BX POP
   SHORT_REAL 4 [BP+#] FSTP
   WAIT BP AX MOV
NEXT
```

F/ (F1 F2 – F3)

Leave F3, the floating-point quotient of F1 divided by F2.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (F3) The quotient of F1 divided by F2.

Example of Use:

... Sum F@ 10. F/ Sum F! ...

This code fragment would divide the value held in Sum by ten.

Algorithm: Move both numbers from the 8088 data stack to the 8087. Decre-

ment the stack with the pops. Divide the numbers on the 8087, then move the result back to the 8088 data stack.

Suggested Extensions: None.

Definition:

```
CODE F/
   AX BP MOV
   BP SP MOV
   SHORT_REAL 4 [BP+#] FLD
   SHORT_REAL 0 [BP+#] FDIV
   BX POP BX POP
   SHORT_REAL 4 [BP+#] FSTP
   WAIT BP AX MOV
NEXT
```

FNEGATE (F1 – F2)

Leave F2, F1 with the opposite sign.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) F1 with it's sign reversed.

Example of Use:

```
... Sum F@ PI FNEGATE F* ...
```

This code fragment would multiply Sum by negative pi.

Algorithm: Move the number from the 8088 data stack to the 8087. Negate the number on the 8087, then move the result back to the 8088 data stack.

Suggested Extensions: None.

Definition:

```
CODE FNEGATE
   AX BP MOV
   BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   FCHS
   SHORT_REAL 0 [BP+#] FSTP
```

```
  WAIT BP AX MOV
NEXT
```

FP->INT (F - N)

Leave N, the integer equivalent of F.

Stack on Entry: (F) A floating-point number.

Stack on Exit: (N) The integer equivalent of F.

Example of Use:

```
... PI INT->FP . ...
```

This code fragment would print a three on the display.

Algorithm: Move the number from the 8088 data stack to the 8087. Decrement the stack with a pop. Move the result back to the 8088 data stack, using a 8087 integer store.

Suggested Extensions: None.

Definition:

```
CODE FP->INT
  AX BP MOV
  BP SP MOV
  SHORT_REAL 0 [BP+#] FLD
  BX POP
  WORD_INTEGER 2 [BP+#] FSTP
  WAIT BP AX MOV
NEXT
```

INT->FP (N - F)

Leave F, the floating-point equivalent of N.

Stack on Entry: (N) An integer number.

Stack on Exit: (F) The floating-point equivalent of N.

Example of Use:

```
... 1 INT->FP 3 INT->FP F/ R. ...
```

This code fragment would print a the floating-point representation of one-third on the display.

Algorithm: Move the number from the 8088 data stack to the 8087 with an integer load. Increment the stack with a push. Move the result back to the 8088 data stack, using a 8087 real store.

Suggested Extensions: None.

Definition:

```
CODE INT->FP
   AX BP MOV
   BX PUSH
   BP SP MOV
   WORD_INTEGER 2 [BP+#] FLD
   SHORT_REAL 0 [BP+#] FSTP
   WAIT BP AX MOV
NEXT
```

FABS (F1 – F2)

Leave F2, the absolute value of F1.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The absolute value of F1.

Example of Use:

```
... TEMPERATURE F@ FABS R. ...
```

This code fragment would print the absolute value of the variable TEMPERATURE on the display.

Algorithm: Move the number from the 8088 data stack to the 8087. Determine the absolute value of the number on the 8087, then move the result back to the 8088 data stack.

Suggested Extensions: None.

Definition:

```
CODE FABS
   AX BP MOV
```

```
  BP SP MOV
  SHORT_REAL 0 [BP+#] FLD
  FABS
  SHORT_REAL 0 [BP+#] FSTP
  WAIT BP AX MOV
NEXT
```

F*10 (F1 – F2)

Leave F2, F1 multiplied by ten. Used in floating-point I/O.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) F1 times ten.

Example of Use:

```
... PI F*10 R. ...
```

This code fragment would print 31.4157 on the display. Algorithm: Use the word F* , place ten on the stack by using the double length equivalent.

Suggested Extensions: None.

Definition:

```
: F*10 16672, F* ;
```

F/10 (F1 – F2)

Leave F2, F1 divided by ten. Used in floating-point I/O.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) F1 divided by ten.

Example of Use:

```
... PI F/10 R. ...
```

This code fragment would print .314157 on the display.

Algorithm: Use the word F/, place ten on the stack by using the double-length equivalent.

Suggested Extensions: None.

Definition:

```
: F/10 16672, F/ ;
```

F= (F1 F2 – B)

Compare two floating-point numbers, checking for equality.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (B) Boolean flag, true if F1 is equal to F2.

Example of Use:

```
... Sum F@ TOTAL F@ F= ...
```

This code fragment would leave a true flag on the stack if the values in Sum and TOTAL were equal.

Algorithm: Move the arguments to the 8087 registers. Pop them off the 8088 stack. Compare the numbers on the 8087, popping them off its stack. Move the flag word from the 8087 to the 8088, and check it.

Suggested Extensions: None.

Definition:

```
0 VARIABLE f8087

CODE F=
  CX BP MOV
  BP SP MOV
  SHORT_REAL 0 [BP+#] FLD
  SHORT_REAL 4 [BP+#] FCOMP
  BX POP BX POP BX POP
  f8087 ] FSTSTATUSW DX 0 # MOV
  WAIT AH f8087 1+ ] MOV SAHF ZIF
    DX DEC
```

```
  /ENDIF
  6 [BP+#] DX MOV
  BP CX MOV
NEXT
```

F0= (F – B)

Compare a floating-point number to zero.

Stack on Entry: (F) A floating-point number.

Stack on Exit: (B) Boolean flag, true if F1 is equal to zero.

Example of Use:

```
... Sum F@ F0= ...
```

This code fragment would leave a true flag on the stack if the value in sum was zero.

Algorithm: Move the argument to the 8087 registers. Pop it off the 8088 stack. Compare it to zero on the 8087. Move the flag word from the 8087 to the 8088, and check it.

Suggested Extensions: None.

Definition:

```
CODE F0=
  CX BP MOV
  BP SP MOV
  SHORT_REAL 0 [BP+#] FLD
  FTST
  BX POP
  f8087 ] FSTSTATUSW DX 0 # MOV
  WAIT AH f8087 1+ ] MOV SAHF ZIF
    DX DEC
  /ENDIF
  2 [BP+#] DX MOV
  BP CX MOV
  0 ST FSTP
NEXT
```

F< (F1 F2 – B)

Compare two floating-point numbers, checking for a less than condition.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (B) Boolean flag, true if F1 is less than F2.

Example of Use:

```
... Sum F@ TOTAL F@ F< ...
```

This code fragment would leave a true flag on the stack if the value in Sum was less than the value in TOTAL.

Algorithm: Move the arguments to the 8087 registers. Pop them off the 8088 stack. Compare the numbers on the 8087, popping them off its stack. Move the flag word from the 8087 to the 8088, and check it.

Suggested Extensions: None.

Definition:

```
CODE F<
  CX BP MOV
  BP SP MOV
  SHORT_REAL 4 [BP+#] FLD
  SHORT_REAL 0 [BP+#] FCOMP
  BX POP BX POP BX POP
  f8087 ] FSTSTATUSW DX 0 # MOV
  WAIT AH f8087 1+ ] MOV SAHF CIF
    DX DEC
  /ENDIF
  6 [BP+#] DX MOV
  BP CX MOV
NEXT
```

F2DUP (F1 F2 – F1 F2 F1 F2)

Duplicate the top two floating-point numbers.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (F1) A copy of F1.
(F2) A copy of F2.
(F1) A copy of F1.
(F2) A copy of F2.

Example of Use: See the words defined below.

Algorithm: Loop through the top four words on the stack, rolling them each to the top.

Suggested Extensions: None.

Definition:

```
: F2DUP 4 0 DO 4 PICK LOOP ;
```

F> (F1 F2 – B)

Compare two floating-point numbers, checking for a greater than condition.

Stack on Entry: (F1) A floating-point number.
(F1(F2) A floating-point number.

Stack on Exit: (B) Boolean flag, true if F1 is greater than F2.

Example of Use:

```
... Sum F@ TOTAL F@ F> ...
```

This code fragment would leave a true flag on the stack if the value in Sum was greater than the value in TOTAL.

Algorithm: Swap the arguments and call F<.

Suggested Extensions: None.

Definition:

```
: F> 2SWAP F< ;
```

F<= (F1 F2 – B)

Compare two floating-point numbers, checking for a less than or equal condition.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (B) Boolean flag, true if F1 is less than or equal F2.

Example of Use:

```
... Sum F@ TOTAL F@ F<= ...
```

This code fragment would leave a true flag on the stack if the value in Sum was less than or equal the value in TOTAL.

Algorithm: Duplicate the arguments and use F< and F=. OR their results.

Suggested Extensions: None.

Definition:

```
: F<= F2DUP F< >R F= R> OR ;
```

F>= (F1 F2 – B)

Compare two floating-point numbers, checking for a greater than or equal condition.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (B) Boolean flag, true if F1 is greater than or equal to F2.

Example of Use:

```
... Sum F@ TOTAL F@ F>= ...
```

This code fragment would leave a true flag on the stack if the value in Sum was greater than or equal to the value in TOTAL.

Algorithm: Duplicate the arguments and use F> and F=. OR their results.

Suggested Extensions: None.

Definition:

```
: F>= F2DUP F> >R F= R> OR ;
```

F<> (F1 F2 – B)

Compare two floating-point numbers, checking for equality.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (B) Boolean flag, true if F1 is not equal to F2.

Example of Use:

```
... Sum F@ TOTAL F@ F<> ...
```

This code fragment would leave a true flag on the stack if the values in Sum and TOTAL were not equal.

Algorithm: Not the result of F=.

Suggested Extensions: None.

Definition:

```
: F<> F= NOT ;
```

FP->DINT (F – D)

Leave D, the double length integer equivalent of F.

Stack on Entry: (F) A floating-point number.

Stack on Exit: (D) The double length integer equivalent of F.

Example of Use:

```
... PI FP->DINT D. ...
```

This code fragment would print a three on the display.

Algorithm: Move the number from the 8088 data stack to the 8087. Move the result back to the 8088 data stack, using an 8087 integer store.

Suggested Extensions: None.

Definition:

```
CODE FP->DINT
   AX BP MOV
   BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   SHORT_INTEGER 0 [BP+#] FSTP
   BP AX MOV WAIT
   CX POP BX POP CX PUSH BX PUSH
NEXT
```

DINT->FP (D – F)

Leave F, the floating-point equivalent of D.

Stack on Entry: (D) A doublelength integer.

Stack on Exit: (F) The floating-point equivalent of D.

Example of Use:

```
... 23, DINT->FP R. ...
```

This code fragment would print a 23 on the display.

Algorithm: Move the number from the 8088 data stack to the 8087 using an integer load. Move the result back to the 8088 data stack, using an 8087 real store.

Suggested Extensions: None.

Definition:

```
CODE DINT->FP
   AX BP MOV
   BP SP MOV
   CX POP BX POP CX PUSH BX PUSH
   SHORT_INTEGER 0 [BP+#] FLD
   SHORT_REAL 0 [BP+#] FSTP
   BP AX MOV WAIT
NEXT
```

FTRUNC (F1 – F2)

Leave F2, the integer portion of F1.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The whole number portion of F1.

Example of Use:

```
... 22.75 FTRUNC R. ...
```

This code fragment would print a 22 on the display.

Algorithm: Move the number from the 8088 data stack to the 8087.Change the 8087 rounding mode to truncate the number when it is rounded to zero. Restore the 8087 rounding mode and move the result back to the 8088 data stack, using a 8087 real store.

Suggested Extensions: None.

Definition:

```
: 0. 0. ;

CODE FTRUNC
   f8087 ] FSTCW WAIT
   BX f8087 ] MOV AX BX MOV
   AX 3072 # Or f8087 ] AX MOV
   f8087 ] FLDCW
   CX BP MOV BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   FRNDINT f8087 ] BX MOV
   SHORT_REAL 0 [BP+#] FSTP
   f8087 ] FLDCW WAIT
   BP CX MOV
NEXT
```

F.R (F N –)

Print F on the display, with N digits after the decimal point.

Stack on Entry: (F) A floating-point number.
(N) The number of digits to print after the decimal point.

Stack on Exit: Empty.

Example of Use:

```
... Sum 8 F.R ...
```

This code fragment would print the value of sum with eight digits after the decimal point.

Algorithm: First, print the integer portion of the number. Divide by ten, building the output string in the pad. Print it when zero is reached. Then loop on the trailing digits, print one at a time until the count provided is exhausted or the number is zero.

Suggested Extensions: None.

Definition:

```
-219857153. FCONSTANT .49->
0 VARIABLE PPOS
: F.R >R 2DUP 0. F< IF
    FNEGATE ." -"
  ENDIF
  PAD 30 + PPOS ! 2DUP BEGIN
    FTRUNC 2DUP F/10 FTRUNC 2DUP >R >R
    F*10 F-
    FP->INT 48 + PPOS @ C!
    -1 PPOS +! R> R> 2DUP FP->INT
  0= UNTIL PPOS @ 1+ DUP PAD 30 + -
  NEGATE 1+ TYPE ." ." 2DROP
  2DUP FTRUNC F- R> 0 DO
    F*10 2DUP FTRUNC FP->INT 48 + EMIT
    2DUP FTRUNC F- 2DUP .49-> F+ FP->INT
    0= IF
      LEAVE
    ENDIF
  LOOP 2DROP SPACE ;
```

F. (F –)

Print F on the display, with six digits after the decimal point.

Stack on Entry: (F) A floating-point number.

Stack on Exit: Empty.

Example of Use:

```
... Sum F. ...
```

This code fragment would print the value of sum with six digits after the decimal point.

Algorithm: Call F.R with a six.

Suggested Extensions: None.

Definition:

```
: F. 6 F.R ;
```

FE.R (F N –)

Print F on the display in scientific notation, with N digits after the decimal point.

Stack on Entry: (F) A floating-point number.
(N) The number of digits to print after the decimal point.

Stack on Exit: Empty.

Example of Use:

```
... Sum 8 FE.R ...
```

This code fragment would print the value of Sum in scientific notation with eight digits after the decimal point.

Algorithm: Normalize the number between zero and one by dividing by 10∧6 or 10∧−6 until it is within range of 10∧6 > x > 10∧6. Then, continue the normalization by tens. When it is complete, print out the number, and then the exponent.

Suggested Extensions: None.

Definition:

```
0 VARIABLE TX
16256. FCONSTANT 1.
-858964532. FCONSTANT .1
603998580. FCONSTANT #E6
935146886. FCONSTANT #E-6
: F/#E6 #E6 F/ ;
: F*#E6 #E6 F* ;
: FE.R TX 0SET >R 2DUP 0. F< IF
      ." -" FNEGATE
   ENDIF 2DUP F0= NOT IF
      2DUP #E6 F>= IF BEGIN
```

```
        F/#E6 6 TX +! 2DUP #E6 F<
    UNTIL ELSE
      2DUP #E-6 F< IF BEGIN
          F*#E6 -6 TX +! 2DUP #E-6 F>
    UNTIL ENDIF ENDIF
    2DUP 1. F>= IF BEGIN
      F/10 1 TX +! 2DUP 1. F<
    UNTIL ELSE
    2DUP .1 F<= IF BEGIN
      F*10 -1 TX +! 2DUP .1 F>=
  UNTIL ENDIF ENDIF
  R> F.R ." e" TX ?
ELSE
  R> DROP 2DROP ." 0.0e0 "
ENDIF ;
```

FE. (F –)

Print F on the display in scientific notation, with six digits after the decimal point.

Stack on Entry: (F) A floating-point number.

Stack on Exit: Empty.

Example of Use:

```
... Sum FE. ...
```

This code fragment would print the value of Sum in scientific notation with six digits after the decimal point.

Algorithm: Use FE.R with a value of six.

Suggested Extensions: None.

Definition:

```
: FE. 6 FE.R ;
```

R. (F –)

Print F on the display, in an appropriate form.

Stack on Entry: (F) A floating-point number.

Stack on Exit: Empty.

Example of Use:

```
... Sum F@ R. ...
```

This code fragment would print the value of Sum on the display.

Algorithm: Use FE. if F is a very large or very small number; otherwise, use F. Check for a possible zero to avoid zero in scientific notation.

Suggested Extensions: None.

Definition:

```
: R. 2DUP 0= SWAP 0= AND IF
     2DROP ." 0.0 " EXIT
   ENDIF
   2DUP FABS
   2DUP #E6 F> >R #E-6 F< R> OR IF
      FE.
   ELSE
      F.
   ENDIF ;
```

FNUM (A – (F) B)

Attempt to convert the text string at A to a floating-point number.

Stack on Entry: (A) A text string as from the word WORD.

Stack on Exit: (B) A Boolean flag, true if a number could be converted.
(F) The converted number if the flag is true.

Example of Use:

```
: GET-A-REAL QUERY >IN 0SET BL WORD FNUM ;
```

GET-A-REAL would input a line of text and them attempt to convert the first part of the line to a real number.

Algorithm: Strip away a negative sign if it is present. Then attempt to convert the integer portion of the number. Convert it to floating-point. Look for a

decimal point. If found, convert the numbers after it to floating-point by dividing by ten. Add this to the whole number portion. Look for an 'E' that would indicate an exponent. If an 'E' is found, strip away a negative sign if present. Convert the exponent to integer and apply it to the number obtained. Leave a true flag if all these steps complete successsfully; false otherwise.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE INS
0 CVARIABLE INXS
0. FVARIABLE DIVAMT
16672. FCONSTANT 10.
: 0-9? DUP 48 >= SWAP 57 <= AND ;
: EXPMOD DROP INXS C@ IF
      0 DO F/10 LOOP
   ELSE
      0 DO F*10 LOOP
   ENDIF ;
0 CVARIABLE .GOT
: FNUM
   10. DIVAMT F! INS C0SET INXS C0SET
   .GOT C0SET DUP 1+ C@ 45 = IF
      1+ INS C1SET
   ENDIF 0. ROT >BINARY >R DINT->FP R>
   DUP C@ 46 = IF
      .GOT C1SET BEGIN
         1+ DUP C@ 0-9?
      WHILE
         DUP C@ 48 - INT->FP DIVAMT F@ F/
         ROT >R F+ R>
         DIVAMT DUP >R F@ F*10 R> F!
      REPEAT
   ENDIF
   DUP C@ 69 = IF
      DUP 1+ C@ 45 = IF
         1+ INXS C1SET
      ENDIF 0. ROT >BINARY >R EXPMOD R>
   ENDIF
   INS C@ IF >R FNEGATE R> ENDIF
   C@ BL <> .GOT C@ 0= OR
   IF 2DROP 0 ELSE -1 ENDIF ; ->

( Make floating-point input possible in Atila)
: FINP BL WORD FNUM ;
```

```
: NNUM DUP FNUM IF ROT DROP
      ELSE [ V.NUM @ ] LITERAL
      EXECUTE ENDIF ;
' NNUM V.NUM !
```

SQRT (F1 – F2)

Leave F2,square root of F1.

Argument Range: 0 <= F1 <= Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The square root of F1.

Example of Use:

```
... TEMPERATURE F @ SQRT R. ...
```

This code fragment would print the square root of the variable TEMPERATURE on the display.

Algorithm: Move the number from the 8088 data stack to the 8087. Determine the square root of the number on the 8087, then move the result back to the 8088 data stack.

Suggested Extensions: None.

Definition:

```
CODE SQRT
   AX BP MOV
   BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   FSQRT
   SHORT_REAL 0 [BP+#] FSTP
   BP AX MOV WAIT
NEXT
```

LN (F1 – F2)

Leave F2, the natural log of F1.

Argument Range: –Infinity <= Q F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The natural log of F1.

Example of Use:

```
... TEMPERATURE F@ LN R. ...
```

This code fragment would print the natural log of the variable TEMPERATURE on the display.

Algorithm: Move the number from the 8088 data stack to the 8087. Determine the natural log of the number on the 8087, then move the result back to the 8088 data stack.

Suggested Extensions: None.

```
Definition: CODE LN
   AX BP MOV BP SP MOV
   LN2FLD
   SHORT_REAL 0 [BP+#] FLD
   FYL2X
   SHORT_REAL 0 [BP+#] FSTP
   BP AX MOV WAIT
NEXT
```

LOG (F1 – F2)

Leave F2, the base 10 log of F1.

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The base 10 log of F1.

Example of Use:

```
... TEMPERATURE F@ LOG R. ...
```

This code fragment would print the base 10 log of the variable TEMPERATURE on the display.

Algorithm: Move the number from the 8088 data stack to the 8087. Deter-

mine the base 10 log of the number on the 8087, then move the result back to the 8088 data stack.

Suggested Extensions: None.

Definition:

```
CODE LOG
   AX BP MOV BP SP MOV
   LG2FLD
   SHORT_REAL 0 [BP+#] FLD
   FYL2X
   SHORT_REAL 0 [BP+#] FSTP
   BP AX MOV WAIT
NEXT
```

2∧X

Assembler subroutine to calculate 2 to an arbitray power.

Algorithm: Decompose 2∧X to 2∧X = (2∧I) * (2∧F), where I is the integer portion of X and F is the fractional portion. If f is nonnegative, use the F2XM1 instruction to calculate its value. If it is negative, use the identity 2∧f − 1 = −(2∧−f − 1/(2∧−F)) to calculate its value. When finished scale, that result by I to obtain the final value. 2∧X operates on the top 8087 stack entry.

Suggested Extensions: None.

Definition:

```
SUBROUTINE 2∧X

   0 ST 0 ST FLD FRNDINT 0 ST 0 ST FLD
   2 ST 0 ST FSUB ANDPOP REVERSE
   1 ST FXCH FTST [8087 ] FSTSTATUSW
   WAIT AH [8087 1+ ] MOV SAHF CIF
      FCHS F2XM1 0 ST 0 ST FLD
      1FLD 1 ST 0 ST FADD ANDPOP
      1 ST 0 ST FDIV ANDPOP REVERSE FCHS
   /ELSE
      F2XM1
   /ENDIF
   1FLD 1 ST 0 ST FADD ANDPOP
   FSCALE
```

```
  1 ST FXCH 0 ST 0 ST FSTP
  WAIT
END-SUB
```

EXP (F1 – F2)

Leave F2, e raised to the F1 power.

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) e raised to the F1 power.

Example of Use:

... TEMPERATURE F@ EXP R. ...

This code fragment would print e raised to the value of the variable TEMPERATURE on the display.

Algorithm: Use the identity Y∧X = 2(X*LN(Y)).

Suggested Extensions: None.

Definition:

```
CODE EXP
   DX BP MOV BP SP MOV
   L2EFLD
   SHORT_REAL 0 [BP+#] FMUL
   2∧X CALL
   SHORT_REAL 0 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

∧(F1 F2 – F3)

Leave F3, F1 raised to the F2 power.

Argument Range: –Infinity <= F1 <= +Infinity.
–Infinity <= F2 <= +Infinity.

Stack on Entry: (F1) A floating-point number.
(F2) A floating-point number.

Stack on Exit: (F3) F1 raised to the F2 power.

Example of Use:

```
... TEMPERATURE F@ 3.∧R. ...
```

This code fragment would print the value of the variable TEMPERATURE raised to the third power on the display.

Algorithm: Use the identity Y∧X = 2(X*LN(Y)).

Suggested Extensions: None.

Definition:

```
CODE ∧
   DX BP MOV BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   SHORT_REAL 4 [BP+#] FLD
   FYL2X
   BX POP BX POP 2∧X CALL
   SHORT_REAL 4 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

[FPTAN]

Assembler subroutine that will execute the 8087 FPTAN instruction on any number.

Algorithm: The 8087 instruction FPTAN is used in all the trigonometric calculations. FPTAN only operates on values greater than zero and less than pi/4. This routine reduces the number found on the top of the 8087 stack to this range.
First, the identity tan(x+n Pi) = tan(x) is applied using the 8087 remainder instruction FPREM. FPREM is then used again, this time dividing by pi/4, to reduce the argument to the range required by FPTAN. After this second reduction, the 8087 status word will contain the last three bits of the quotient. This number will be used to determine the octant the final value falls in. Octant one or three requires that we call FPTAN using pi/4 minus the value obtained from FPREM. If the octant is one, five, two, or six, we invert the results of FPTAN. If the octant is two, six, three, or seven we reverse the sign of the result. These rules are derived from the identities tan(x+pi/2)=−cot(x) and tan(Pi/2−x)=cot(x). The returned values, X and Y in most 8087 literature, will be used for all of the trigonometric calculations.

Suggested Extensions: None.

Definition:

```
49152, FVARIABLE F-2
SUBROUTINE [FPTAN]
  CH 0 # MOV FTST f8087 ] FSTSTATUSW
  WAIT AH f8087 1+ ] MOV SAHF CIF
    CH DEC FCHS
  /ENDIF
  PIFLD 1 ST FXCH /BEGIN
    FPREM f8087 ] FSTSTATUSW WAIT
    AH f8087 1+ ] MOV SAHF
  JP 1 ST 0 ST FSTP SHORT_REAL F-2 ] FLD
  PIFLD FSCALE 1 ST 0 ST FSTP 1 ST FXCH
  /BEGIN
    FPREM f8087 ] FSTSTATUSW WAIT
    AH f8087 1+ ] MOV SAHF
  JP FTST f8087 ] FSTSTATUSW WAIT
  CL 0 # MOV byte f8087 1+ ] 65 # And
  byte f8087 1+ ] 64 # CMP ZIF CL DEC
  /ENDIF AH 2 # TEST NZIF
      CL 0 # CMP NZIF
        0 ST 0 ST FSTP 0 ST 0 ST FSTP
        1FLD ZFLD
      /ELSE
        1 ST 0 ST FSUB ANDPOP REVERSE
        FPTAN
      /ENDIF
   /ELSE
      1 ST 0 ST FSTP CL 0 # CMP NZIF
        0 ST 0 ST FSTP 1FLD 1FLD
      /ELSE
        FPTAN
      /ENDIF
   /ENDIF
   2AH 66 # And AH 2 # CMP ZIF
      1 ST FXCH
   /ENDIF AH 64 # CMP ZIF
      1 ST FXCH
   /ENDIF AH 64 # TEST NZIF
      FCHS
   /ENDIF CH 0 # CMP NZIF
      FCHS
   /ENDIF
END-SUB
```

TAN (F1 – F2)

Leave F2, The tangent of F1 (expressed in radians).

Argument Range: −Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The tangent of F1 radians.

Example of Use:

```
... ANANGLE F@ TAN R. ...
```

This code fragment would print the tangent of the variable ANANGLE on the display.

Algorithm: Call FPTAN and return Y divided by X.

Suggested Extensions: None.

Definition:

```
CODE TAN
   DX BP MOV BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   [FPTAN] CALL
   1 ST 0 ST FDIV ANDPOP REVERSE
   SHORT_REAL 0 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

2FLD

A macro to push a two onto the 8087 stack.

Example of Use: See words defined below.

This macro will load a two onto the 8087 stack. It can be used just like any other 8087 instruction in our assembler code. In effect, we can use macros to extend the 8087 instruction set.

Algorithm: Push a one onto the 8087 stack and add it to itself.

Suggested Extensions: None.

Definition:

```
MACRO 2FLD
  1FLD
  0 ST 0 ST FADD
MEND
```

F/2

A macro to divide the top 8087 stack entry by two.

Example of Use: See words defined below.

This macro will load a one onto the 8087 stack and then divide the number ST(1) by it. It can be used just like any other 8087 instruction in our assembler code. In effect, we can use macros to extend the 8087 instruction set.

Algorithm: Push a one onto the 8087 stack and use FDIC to divide ST(1) by it.

Suggested Extensions: None.

Definition:

```
MACRO F/2
  2FLD
  1 ST 0 ST FDIV ANDPOP REVERSE
MEND
```

SIN (F1 – F2)

Leave F2, the sine of F1 (expressed in radians).

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The sine of F1 (radians).

Example of Use:

```
... AN_ANGLE F@ SIN R. ...
```

This code fragment would print the sine of the variable AN_ANGLE on the display.

Algorithm: Call [FPTAN] and return 2*(Y/X) / 1+(Y/X)**2.

Suggested Extensions: None.

Definition:

```
CODE SIN
   DX BP MOV BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   FTST f8087 ] FSTSTATUSW WAIT
   AH f8087 1+ ] MOV SAHF ZIF
      0 ST 0 ST FSTP ZFLD
   /ELSE
      F/2 [FPTAN] CALL
      1 ST 0 ST FDIV ANDPOP REVERSE
      0 ST 0 ST FLD
      2FLD 1 ST 0 ST FMUL ANDPOP
      1 ST FXCH 0 ST 0 ST FLD
      1 ST 0 ST FMUL ANDPOP REVERSE
      1FLD 1 ST 0 ST FADD ANDPOP
      1 ST 0 ST FDIV ANDPOP REVERSE
   /ENDIF
   SHORT_REAL 0 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

COS (F1 – F2)

Leave F2, the cosine of F1 (expressed in radians).

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The cosine of F1 (radians).

Example of Use:

```
... AN_ANGLE F@ COS R. ...
```

This code fragment would print the cosine of the variable AN_ANGLE on the display.

Algorithm: Call [FPTAN] and return 1−(Y/X)**2 / 1+(Y/X)**2.

Suggested Extensions: None.

Definition:

```
CODE COS
  DX BP MOV BP SP MOV
  SHORT_REAL 0 [BP+#] FLD
  FTST [8087 ] FSTSTATUSW WAIT
  AH [8087 1+ ] MOV SAHF ZIF
    0 ST 0 ST FSTP 1FLD
  /ELSE
    F/2 [FPTAN] CALL
    1 ST 0 ST FDIV ANDPOP REVERSE
    0 ST 0 ST FMUL 0 ST 0 ST FLD
    1FLD 1 ST 0 ST FSUB ANDPOP
    1 ST FXCH
    1FLD 1 ST 0 ST FADD ANDPOP
    1 ST 0 ST FDIV ANDPOP REVERSE
  /ENDIF
  SHORT_REAL 0 [BP+#] FSTP
  BP DX MOV WAIT
NEXT
```

1/X (F1 − F2)

Leave F2, the multiplicative identity of F1.

Argument Range: −Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The multiplicative identity of F1.

Example of Use:

```
... AN_ANGLE F@ 1/X R. ...
```

This code fragment would print the multiplicative identity of the variable AN_ANGLE on the display.

Algorithm: Move F1 onto the 8087 stack and them push a 1 on the 8087 stack. Divide the two and return the result to the 8088 data stack.

Suggested Extensions: None.

Definition:

```
CODE 1/X
   DX BP MOV BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   1FLD 1 ST 0 ST FDIV ANDPOP
   SHORT_REAL 0 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

COT (F1 – F2)

Leave F2, the cotangent of F1 (expressed in radians).

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The cotangent of F1 (radians).

Example of Use:

```
... AN_ANGLE F@ COT R. ...
```

This code fragment would print the cotangent of the variable AN_ANGLE on the display.

Algorithm: Call tangent and inverse the result. (COT(X) = 1/TAN(X).

Suggested Extensions: None.

Definition:

```
: COT TAN 1/X ;
```

CSC (F1 – F2)

Leave F2, the cosecant of F1 (expressed in radians).

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The cosecant of F1 (radians).

Example of Use:

```
... ANANGLE F@ CSC R. ...
```

This code fragment would print the cosecant of the variable AN_ANGLE on the display.

Algorithm: Call sine and inverse the result. (CSC(X) = 1/SIN(X).

Suggested Extensions: None.

Definition:

```
: CSC SIN 1/X ;
```

SEC (F1 – F2)

Leave F2, the secant of F1 (expressed in radians).

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The secant of F1 (radians).

Example of Use:

```
... AN_ANGLE F@ SEC R. ...
```

This code fragment would print the secant of the variable AN_ANGLE on the display.

Algorithm: Call cosine and inverse the result. (SEC(X) = 1/COS(X)

Suggested Extensions: None.

Definition:

```
: SEC COS 1/X ;
```

Assembler subroutine that will execute the 8087 FPATAN instruction on any number.

Algorithm: The 8087 instruction FPATAN is used in all the inverse trigonometric calculations. FAPTAN requires two values, known as X and Y, on the 8087 stack. 0 < Y < X must hold true. The identities ArcTan(Z) = −ArcTan(−Z) and ArcTAn(Z) = Pi/2 − ArcTan(1/Z) are used to bring the two arguments into the proper range.

Definition:

```
SUBROUTINE [FPATAN]
   1 ST 0 ST FLD FTST
   CH 0 # MOV f8087 ] FSTSTATUSW
   0 ST 0 ST FSTP
   WAIT AH f8087 1+ ] MOV SAHF CIF
      CH DEC 1 ST FXCH FABS 1 ST FXCH
   /ENDIF
   1 ST FCOM f8087 ] FSTSTATUSW CL 0 #
   MOV WAIT AH f8087 1+ ] MOV SAHF CIF
      CL DEC 1 ST FXCH
   /ENDIF FPATAN CL 0 # CMP NZIF
      FCHS
      1FLD FCHS PIFLD FSCALE
      1 ST FSTP
      1 ST 0 ST FADD ANDPOP
   /ENDIF CH 0 # CMP NZIF
      FCHS
   /ENDIF
END−SUB
```

ATAN (F1 − F2)

Leave F2, the angle (in radians) whose tangent is F1.

Argument Range: −Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The arctangent of F1.

Example of Use:

```
... SOME_RADIANS F@ ATAN R. ...
```

This code fragment would print the angle whose tangent was the value held in the variable SOME_RADIANS.

Algorithm: Use F1 as Y and set X to one, then call [FPATAN].

Suggested Extensions: None.

Definition:

```
CODE ATAN
   DX BP MOV BP SP MOV
   SHORT_REAL 0 [BP+#] FLD 1FLD
   [FPATAN] CALL
   SHORT_REAL 0 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

ACOTN (F1 – F2)

Leave F2, the angle (in radians) whose cotangent is F1.

Argument Range: −Infinity <= F1 <= −1 or 1 <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The arccotangent of F1.

Example of Use:

```
... SOME_RADIANS F@ ACOTN R. ...
```

This code fragment would print the angle whose cotangent was the value held in the variable SOME_RADIANS.

Algorithm: Use F1 as X and set Y to one, then call [FPATAN].

Suggested Extensions: None.

Definition:

```
CODE ACOTN
   DX BP MOV BP SP MOV
   SHORT_REAL 0 [BP+#] FLD FTST
```

```
    18087 ] FSTSTATUSW WAIT
    AH 18087 1+ ] MOV SAHF CIF
      FCHS 1FLD FCHS
    /ELSE
      1FLD
    /ENDIF 1 ST FXCH FPATAN CALL
    SHORT_REAL 0 [BP+#] FSTP
    BP DX MOV WAIT
NEXT
```

ASIN (F1 – F2)

Leave F2, the angle (in radians) whose sine is F1.

Argument Range: −1 <= F1 <= 1.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The arcsine of F1.

Example of Use:

```
... SOME_RADIANS F@ ASIN R. ...
```

This code fragment would print the angle whose sine was the value held in the variable SOME_RADIANS.

Algorithm: Use F1 as Y and set X to SQR((1−F1)*(1+F1)), then call [FPATAN].

Suggested Extensions: None.

Definition:

```
CODE ASIN
  DX BP MOV BP SP MOV
  SHORT_REAL 0 [BP+#] FLD
  0 ST FLD 0 ST FLD 1FLD
  1 ST 0 ST FADD ANDPOP
  1 ST FXCH 1FLD
  1 ST 0 ST FSUB ANDPOP
  1 ST 0 ST FMUL ANDPOP
  FABS FSQRT
```

```
  [FPATAN] CALL
  SHORT_REAL 0 [BP+#] FSTP
  BP DX MOV WAIT
NEXT
```

ACOS (F1 – F2)

Leave F2, the angle (in radians) whose cosine is F1.

Argument Range: −1 <= F1 <= 1.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The arccosine of F1.

Example of Use:

```
... SOME_RADIANS F@ ACOS R. ...
```

This code fragment would print the angle whose cosine was the value held in the variable SOME_RADIANS.

Algorithm: Set Y equal to SQR(1 − F1), set X equal to SQR(1 + F1), then call [FPATAN]. Multiply the result [FPATAN] returned by 2.

Suggested Extensions: None.

Definition:

```
CODE ACOS
  DX BP MOV BP SP MOV
  SHORT_REAL 0 [BP+#] FLD
  0 ST FLD 1FLD
  1 ST 0 ST FSUB ANDPOP FSQRT
  1 ST FXCH 1FLD
  1 ST 0 ST FADD ANDPOP FSQRT
  [FPATAN] CALL
  2FLD 1 ST 0 ST FMUL ANDPOP
  SHORT_REAL 0 [BP+#] FSTP
  BP DX MOV WAIT
NEXT
```

ASEC (F1 – F2)

Leave F2, the angle (in radians) whose secant is F1.

Argument Range: –Infinity <= F1 <= –1 or 1 <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The arcsecant of F1.

Example of Use:

```
... SOME_RADIANS F@ ASEC R. ...
```

This code fragment would print the angle whose secant was the value held in the variable SOME_RADIANS.

Algorithm: Set Y equal to SQR(F1 – 1), set X equal to SQR(F1 + 1), then call [FPATAN]. Multiply the result [FPATAN] returned by 2.

Suggested Extensions: None.

Definition:

```
CODE ASEC
   DX BP MOV BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   0 ST FLD 1FLD
   1 ST 0 ST FSUB ANDPOP REVERSE
   FABS FSQRT 1 ST FXCH 1FLD
   1 ST 0 ST FADD ANDPOP
   FABS FSQRT [FPATAN] CALL
   2FLD 1 ST 0 ST FMUL ANDPOP
   SHORT_REAL 0 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

ACSC (F1 – F2)

Leave F2, the angle (in radians) whose cosecant is F1.

Argument Range: –Infinity <= F1 <= –1 or 1 <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The arccosecant of F1.

Example of Use:

```
... SOME_RADIANS F@ ACSC R. ...
```

This code fragment would print the angle whose cosecant was the value held in the variable SOME_RADIANS.

Algorithm: Set Y equal to the sign of F1 (that is, 1 or −1), set X equal to SQR((F1+1)*(F1−1)), then call [FPATAN].

Suggested Extensions: None.

Definition:

```
CODE ACSC
  DX BP MOV BP SP MOV
  SHORT_REAL 0 [BP+#] FLD
  FTST 18087 ] FSTSTATUSW
  1FLD
  WAIT AH 18087 1+ ] MOV SAHF CIF
    FCHS
  /ENDIF 1 ST FXCH
  0 ST FLD 1FLD
  1 ST 0 ST FADD ANDPOP
  1 ST FXCH 1FLD
  1 ST 0 ST FSUB ANDPOP REVERSE
  1 ST 0 ST FMUL ANDPOP
  FABS FSQRT
  [FPATAN] CALL
  SHORT_REAL 0 [BP+#] FSTP
  BP DX MOV WAIT
NEXT
```

FSIGN (F1 – F2)

Leave F2, −1 if F1 is negative, 0 if F1 is zero, 1 if F1 is positive.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The sign of F1 as above.

Example of Use:

... SOME_RADIANS F@ ACSC R. ...

This code fragment would print the angle whose cosecant was the value held in the variable SOME_RADIANS.

Algorithm: Use the 8087 instruction FTST to test F1, then move the proper value to the 8087 stack.

Suggested Extensions: None.

Definition:

```
CODE FSIGN
   DX BP MOV BP SP MOV
   SHORT_REAL 0 [BP+#] FLD
   FTST [8087 ] FSTSTATUSW
   1FLD
   WAIT AH [8087 1+ ] MOV SAHF CIF
      FCHS
   /ENDIF
   ZIF
      0 ST 0 ST FSTP ZFLD
   /ENDIF 1 ST FXCH 0 ST 0 ST FSTP
   SHORT_REAL 0 [BP+#] FSTP
   BP DX MOV WAIT
NEXT
```

SINH (F1 – F2)

Leave F2, the hyperbolic sine of F1.

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The hyperbolic sine of F1.

Example of Use:

... CABLE F@ SINH R. ...

This code fragment would print the hyperbolic sine of the variable CABLE on the display.

Algorithm: Calculate the identity SINH(X) = (e∧X − e∧X)/2

Suggested Extensions: None.

Definition:

```
16384, FCONSTANT 2.0

: SINH C( F1 – F2)
   2DUP EXP 2SWAP FNEGATE EXP F–
   2.0 F/ ;
```

COSH (F1 – F2)

Leave F2, the hyperbolic cosine of F1.

Argument Range: −Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The hyperbolic cosine of F1.

Example of Use:

```
... CABLE F@ COSH R. ...
```

This code fragment would print the hyperbolic cosine of the variable CABLE on the display.

Algorithm: Calculate the identity COSH(X) = (e∧X + e∧−X)/2

Suggested Extensions: None.

Definition:

```
: COSH C( F1 – F2)
   2DUP EXP 2SWAP FNEGATE EXP F+
   2.0 F/ ;
```

TANH (F1 – F2)

Leave F2, the hyperbolic tangent of F1.

Argument Range: −Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The hyperbolic tangent of F1.

Example of Use:

```
... CABLE F@ TANH R. ...
```

This code fragment would print the hyperbolic tangent of the variable CABLE on the display.

Algorithm: Calculate the identity TANH(X) = (−e∧−X/(e∧X + e−X))*2 + 1

Suggested Extensions: None.

Definition:

```
16256, FCONSTANT 1.0

: TANH 2.0 F* EXP 1.0 F+ 2.0 2SWAP
   F/ 1.0 2SWAP F− ;
```

SECH (F1 − F2)

Leave F2, the hyperbolic secant of F1.

Argument Range: −Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The hyperbolic secant of F1.

Example of Use:

```
... CABLE F@ SECH R. ...
```

This code fragment would print the hyperbolic secant of the variable CABLE on the display.

Algorithm: Inverse the result of COSH, SECH(X) = 1 / COSH(X).

Suggested Extensions: None.

Definition:

```
: SECH COSH 1/X ;
```

CSCH (F1 – F2)

Leave F2, the hyperbolic cosecant of F1.

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The hyperbolic cosecant of F1.

Example of Use:

```
... CABLE F@ CSCH R. ...
```

This code fragment would print the hyperbolic cosecant of the variable CABLE on the display.

Algorithm: Inverse the result of SINH, CSCH(X) = 1 / SINH(X).

Suggested Extensions: None.

Definition:

```
: CSCH SINH 1/X ;
```

COTNH (F1 – F2)

Leave F2, the hyperbolic cotangent of F1.

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The hyperbolic cotangent of F1.

Example of Use:

```
... CABLE F@ COTNH R. ...
```

This code fragment would print the hyperbolic cotangent of the variable CABLE on the display.

Algorithm: Inverse the result of TANH, COTNH(X) = 1 / TANH(X).

Suggested Extensions: None.

Definition:

```
: COTNH TANH 1/X ;
```

ASINH (F1 – F2)

Leave F2, the value whose hyperbolic sine is F1.

Argument Range: –Infinity <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The inverse hyperbolic sine of F1.

Example of Use:

```
... H–VALUE F@ ASINH R. ...
```

This code fragment would print the inverse hyperbolic sine of the variable H–VALUE on the display.

Algorithm: Calculate the identity, ASINH(X) = LN(X + SQR(X^2 + 1)).

Suggested Extensions: None.

Definition:

```
: ASINH
  2DUP 2DUP F* 1.0 F+ SQRT F+ LN ;
```

ACOSH (F1 – F2)

Leave F2, the value whose hyperbolic cosine is F1.

Argument Range: 1 <= F1 <= +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The inverse hyperbolic cosine of F1.

Example of Use:

```
... H-VALUE F@ ACOSH R. ...
```

This code fragment would print the inverse hyperbolic cosine of the variable H-VALUE on the display.

Algorithm: Calculate the identity, ASINH(X) = LN(X + SQR(X∧2 − 1)).

Suggested Extensions: None.

Definition:

```
: ACOSH
  2DUP 2DUP F* 1.0 F- SQRT F+ LN ;
```

ATANH (F1 – F2)

Leave F2, the value whose hyperbolic tangent is F1.

Argument Range: −Infinity < F1 < −1 or 1 < F1 < Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The inverse hyperbolic tangent of F1.

Example of Use:

```
... H-VALUE F@ ATANH R. ...
```

This code fragment would print the inverse hyperbolic cosine of the variable H-VALUE on the display.

Algorithm: Calculate the identity, ASINH(X) = LN(X + SQR(X∧2 − 1)).

Suggested Extensions: None.

Definition:

```
: ATANH
  2DUP 1.0 F+ 2SWAP 1.0 2SWAP F- F/ LN
  2.0 F/ ;
```

ASECH (F1 – F2)

Leave F2, the value whose hyperbolic secant is F1.

Argument Range: 0 < F1 <= 1.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The inverse hyperbolic secant of F1.

Example of Use:

```
... H-VALUE F@ ASECH R. ...
```

This code fragment would print the inverse hyperbolic secant of the variable H-VALUE on the display.

Algorithm: Calculate the identity, ASECH(X) = LN((1 + SQR(1 – X∧2))/X).

Suggested Extensions: None.

Definition:

```
: ASECH
  2DUP 2DUP FNEGATE F* 1.0 F+ SQRT
  1.0 F+ LN 2.0 F/ ;
```

ACSCH (F1 – F2)

Leave F2, the value whose hyperbolic cosecant is F1.

Argument Range: –Infinity < F1 < 0 or 0 < F1 < +Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The inverse hyperbolic cosecant of F1.

Example of Use:

```
... H-VALUE F@ ASECH R. ...
```

This code fragment would print the inverse hyperbolic cosecant of the variable H-VALUE on the display.

Algorithm: Calculate the identity, ACSCH(X) = LN(1/X + (SQR(1+ X∧2)/ABS(X))).

Suggested Extensions: None.

Definition:

```
: ACSCH
  2DUP 2DUP 2DUP F* 1.0 F+ SQRT 1.0 F+
  2SWAP FSIGN F* LN 2SWAP F/ ;
```

ACOTNH (F1 – F2)

Leave F2, the value whose hyperbolic cotangent is F1.

Argument Range: −Infinity < F1 < −1 or 1 < F1 < Infinity.

Stack on Entry: (F1) A floating-point number.

Stack on Exit: (F2) The inverse hyperbolic cotangent of F1.

Example of Use:

```
... H-VALUE F@ ACOTNH R. ...
```

This code fragment would print the inverse hyperbolic cotangent of the variable H-VALUE on the display.

Algorithm: Calculate the identity, COTNH(X)=LN((X+1)/(X−1))/2

Suggested Extensions: None.

Definition:

```
: ACOTNH
  2DUP 2DUP 1.0 F+ 2SWAP 1.0 F- F/ LN
  2.0 F/ ;
```

7

Strings

Words Defined in This Chapter:

$VARIABLE	Define a string variable.
$CONSTANT	Define a string constant.
$ARRAY	Define a one-dimensional string array.
$@	Fetch a string.
$!	Store a string.
$.	Print a string.
$?	Fetch and print a string.
LEN	Determine the length of a string.
LEFT$	Return the left-hand portion of a string.
RIGHT$	Return the right-hand portion of a string.
MID$	Return a section of a string.
ASC	Return the ASCII value of a string.
CHR$	Form a string from a specified ASCII character.
$+!	Concatenate two strings.
+ $=	Compare two strings for equality.
$<	String compare, less than.
$>	String compare, greater than.
$<=	String compare, less than or equal.
$>=	String compare, greater than or equal.
$<>	String compare, not equal.

FIND$	Search for a substring in a string.
NUM$	Convert a single-length number to string.
DNUM$	Convert a double-length number to a string.
DVAL	Convert a string to a double-length number.
VAL	Convert a string to a single-length number.
SORT	Sort a string array.

This chapter contains a complete string-handling package for Forth. The ability to manipulate and process text is, to some degree, called upon in almost every computer application. The need may range from the simple prompts of an engineering program to the complex string manipulation of a text editor. Despite the fact that basic Forth does not come with a predefined string package, Forth easily adapts itself to the manipulation of strings. We'll borrow some terminology from BASIC and use the $ as shorthand for the word string.

Each string will have a length byte, and each string variable will use a byte to hold the maximum size string the variable can hold. Since bytes are being used the maximum size of any single string will be 255 characters.

The first three words, $VARIABLE, $CONSTANT, and $ARRAY will enable us to allocate space for strings. These three words are defining words that have both a compile-time and run-time behavior. The words $! (String Store), $@ (String Fetch), and $+! (String Plus Store) will allow us to store values into string variables and fetch them for further use.
They are analogous to the words @, !, and +!.

LEFT$, RIGHT$, and MID$ enable us to break up strings into smaller strings. $. (String Dot) allows us to print out strings. The words $= (String Equal), $< (String Less Than), $> (String Greater Than) ,$<= (String Less Than or Equal), $>= (String Greater Than or Equal), and $<> (String Not Equal) can be used to compare string lexically. The ASCII code used on the IBM-PC determines the lexical ordering.

VAL, DVAL, NUM$, and DNUM$ are used to convert between numeric and string format. ASC returns the ASCII value of the first character of a string, and CHR$ will make a one-character string consisting of the ASCII character passed to it. FIND$ can be used for substring searches.

The final word, SORT, uses a quicksort algorithm to sort a string array. This is an example of a generic word; we provide SORT with compare and exchange routines (using vectors) and it handles the sort from then on.

Possible Enhancements

The strings in this package are limited to 255 bytes in length. If you

encounter an application that requires larger strings, this package could easily be modified to use larger strings. Because they use the core Forth string words that return byte length strings, NUM$ and VAL$ would be the most difficult to redefine.

A more subtle limitation of this string package is the way in which the string-handling words affect the string variables directly. This is unlike number manipulation in Forth in which numbers on the stack are manipulated. One possible solution would be to have a string stack. A disadvantage would be the large amount of memory required by a string stack; however, if your Forth has the ability to access memory outside the normal 64K Forth limit, a string stack could be an extremely useful enhancement.

$VARIABLE (N –) (– A)

Define a string and allocate space for it.

Stack on Entry: (Compile Time) (N) – Maximum Size of String.
(Run Time) Empty

Stack on Exit: (Compile Time) Empty
(Run Time) (A) – Address of the string variable.

Example of Use:

```
64 $VARIABLE DISK–NAME
```

This will allot a string that can hold up to 64 characters, with the name DISK–NAME.

Algorithm: Enclose the maximim length, then an initial length of zero in the dictionary. Allot space for the string.

Suggested Extensions: Allow the created variable to have an initial value.

Definition:

```
: $VARIABLE <BUILDS DUP C,
  0 C, ALLOT DOES> ;
```

$CONSTANT (–) (– A)

Define and set a string constant.

Stack on Entry: (Compile Time) Empty.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of string constant.

Example of Use:

```
$CONSTANT MY__NAME JAMES"
```

This defines a string constant, MY__NAME, with a value "JAMES". Note that only the trailing quotation mark is used in the definition.

Algorithm: The word WORD returns a string in memory; simply enclose what it returns in the dictionary.

Suggested Extensions: Allow the delimiter to be any character, not just the quote used now.

Definition:

```
34 CCONSTANT ASCII"

: $CONSTANT <BUILDS ASCII" WORD
   C@ 1+ ALLOT DOES> ;
```

$ARRAY (N1 N2 –) (N – A)

Create and allocate space for a string array.

Stack on Entry: (Compile Time) (N1) – Number of strings the array holds.
(N2) – The length of each string.
(Run Time) (N) – Index number of string.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of string variable.

Example of Use:

```
6 40 $ARRAY PHYLUM
```

This would allocate a 16-string array with the name PHYLUM. Each string in the array could hold a maximum of 40 characters. Individual strings in the array could be accessed like this:

This would leave the address of the fourth string in the array. The first string would be referenced as zero.

Algorithm: (Compile Time) For each string, store the maximum size and an initial length of zero, then allocate space for the string.

(Run Time): Calculate the position of the requested string by multiplying the index on the stack by the space for each string and adding the starting address of the array. The space each string occupies is the maximum length plus two (the maximum length byte and the length byte).

Suggested Extensions: Allow the definition of multidimensional arrays.

Definition:

```
: $ARRAY <BUILDS
   SWAP 0 DO
      DUP C, 0 C, DUP ALLOT
   LOOP DROP DOES>
   DUP C@ 2+ ROT * + ;
```

$@ (A1 – A2)

Fetch the address of a string from a string variable.

Stack on Entry: (A1) – Address of string variable.

Stack on Exit: (A2) – Address of the string the variable holds.

Example of Use:

```
DISK-NAME $@
```

This would leave the address of the string held by DISK–NAME on the stack.

Algorithm: Simply add one to the address on the stack to skip the maximum length byte.

Suggested Extensions: None.

Defintition:

```
: $@ 1+ ;
```

$! (A1 A2 –)

Store a string in a string variable.

Stack on Entry: (A1) – Address of a string.
(A2) – Address of a string variable.

Stack on Exit: Empty.

Example of Use:

```
DISK-NAME $@ 2 PHYLUM $!
```

This would store the string held in DISK–NAME into the third string of the array PHYLUM.

Algorithm: First, check to make sure there is sufficient room in the string variable to hold the string; if there is not, abort with an error message. If sufficient room is available, use CMOVE to move the string and length (since they are contiguous) into the string variable.

Suggested Extensions: None.

Definition:

```
: $LEN-CHECK
  <
  ABORT" String Past Storage Allocated."
  ;

: $! 2DUP C@ SWAP C@ $LEN-CHECK
  1+ OVER C@ 1+ CMOVE ;
```

$. (A –)

Print a string on the display.

Stack on Entry: (A) – Address of a string.

Stack on Exit: Empty.

Example of Use:

```
MY-NAME $.
```

Assuming MY_NAME was defined as under $CONSTANT, this code would print "JAMES" on the display.

Algorithm: If the length of the string is nonzero, convert the string address to an address and count suitable for TYPE.

Suggested Extensions: None.

Definition:

```
: $. DUP C@ ?DUP IF
     SWAP 1+ SWAP TYPE
  ENDIF ;
```

$? (A –)

Fetch and print a string.

Stack on Entry: (A) – Address of a String Variable.

Stack on Exit: Empty.

Example of Use:

```
DISK-NAME $?
```

This code would print the string held in DISK–NAME on the display.

Algorithm: Fetch the string, the print it.

Suggested Extensions: None.

Definition:

```
: $? $@ $. ;
```

LEN (A – N)

Return the length of a string.

Stack on Entry: (A) – Address of a string.

Stack on Exit: (N) – The length of that string.

Example of Use:

```
MY-NAME LEN .
```

Again, assuming MY-NAME is defined as under $CONSTANT, this code would print a 5 on the display.

Algorithm: Since the current length is kept as part of the string, simply fetch the length.

Suggested Extension: None.

Definition:

```
: LEN C@ ;
```

LEFT$ (A N -)

Take the leftmost characters of a string.

Stack on Entry: (A) – Address of a string.
(N) – Number of characters to take.

Stack on Exit: Empty.

Example of Use:

```
MY_NAME DISK-NAME $! DISK-NAME 3 LEFT$ DISK-NAME $?
```

This would print "JAM" on the display. If the number of characters the left string is passed is greater than the length of the string, the string will not be changed.

Algorithm: First, check the current length of the string against the desired new length, choose the smaller of the two. Store this value in the length byte of the string.

Suggested extensions: None.

Definition:

```
: LEFT$ SWAP DUP C@ ROT MIN SWAP C! ;
```

RIGHT$ (A N -)

Take the rightmost characters of a string.

Stack on Entry: (A) – Address of a string.
(N) – Number of characters to take.

Stack on Exit: Empty.

Example of Use:

```
MY_NAME DISK-NAME $! DISK-NAME 2 RIGHT$ DISK-NAME $?
```

This would print "ES" on the display. If the number of characters that right string is passed is greater than the length of the string, the string will not be changed.

Algorithm: If the length of the string is less than the number of desired characters, exit the word. Otherwise, determine the start address that must be moved from by calculating START ADDRESS OF STRING + LENGTH OF STRING – NUMBER OF CHARACTERS DESIRED. Move from this address to the start of the string. Finally, store the new length of the string in the length byte.

Suggested Extensions: None.

Definition:

```
: 3PICK 3 PICK ;

: RIGHT$ 2DUP SWAP C@ >= IF
     2DROP EXIT
  ENDIF
  2DUP
  SWAP DUP DUP C@ + 1+ 3PICK -
  SWAP 1+ ROT CMOVE
  SWAP C! ;
```

MID$ (A N1 N2 –)

Return the middle portion of a string.

Stack on Entry: (A) – Address of the string.
(N1) – Starting position in the string.
(N2) – Number of characters to take.

Stack on Exit: Empty.

Example of Use:

```
MY_NAME DISK-NAME $! DISK-NAME 2 2 RIGHT$ DISK-NAME
$?
```

This would print "AM" on the display. If the starting position that MID$ is passed is greater than the length of the string, the string will become an empty string. If there are insufficient characters after the start position, only those available would be returned.

Algorithm: First, determine if the string contains the start position; if not convert it to the empty string and exit the word. Next, determine the number of characters left in the string after the start position, and use this as the length if it is smaller than the number of characters requested. Move the characters from the middle of the string to the start, and, finally, store the new length.

Suggested Extensions: None.

Definition:

```
: MID$ >R 2DUP SWAP C@ > R> SWAP IF
    2DROP DROP EXIT
  ENDIF
  3PICK C@ 3PICK - MIN
  DUP 4 PICK C!
  SWAP 3PICK + 1+
  LROT SWAP 1+ SWAP CMOVE ;
```

ASC (A - C)

Return the ASCII value of the first character of a string.

Stack on Entry: (A) – Address of a string.

Stack on Exit: (C) – ASCII value of the first character.

Example of Use:

```
MY_NAME ASC .
```

This code would print a 99 on the display, since the ASCII value for the character "J" is 99. If ASC is passed the empty string, it will return a zero.

Algorithm: If the string has a length of zero, return a zero. Otherwise, fetch the value of the first character of the string.

Suggested Extensions: Implement a flag for the string package that would enable a user of the package to specify how to handle the ASC of an empty string, either returning a zero (as the code does now), or aborting with an error message.

Definition:

```
: ASC DUP C@ IF
     1+ C@
  ELSE
     DROP 0
  ENDIF ;
```

CHR$ (C – A)

Return a string that consists of a specific ASCII character.

Stack on Entry: (C) – ASCII value.

Stack on Exit: (A) – Address of a string that consists of the ASCII value.

Example of Use:

```
77 CHR$ $.
```

This code would print an asterisk on the display.

Algorithm: /CHR$/ is a string variable that will be used to hold the string CHR$ will create. When CHR$ is invoked, store the ASCII value on the stack into /CHR$/ itself, then fetch its address.

Suggested Extensions: None.

Definition:

```
1 $VARIABLE /CHR$/
1 /CHR$/ 1+ C!
: CHR$ /CHR$/ 2+ C! /CHR$/ $@ ;
```

$+! (A1 A2 –)

Concatenate a string into a string variable.

Stack on Entry: (A1) – Address of a string.
(A2) – Address of a string variable.

Stack on Exit: Empty.

Example of Use:

```
MY_NAME DISK-NAME $! 102 CHR$ DISK-NAME $+! DISK-
NAME $?
```

This code would print "JAMES!" on the display.

Algorithm: First, determine if there is room for the new string that will be created; if not, exit with an error message. If there is room, move the string onto the end of the string variable. Add the length of the string to the current length of the string variable.

Suggested Extensions: Define a word to concatenate onto the left side of a string.

Definition:

```
: $+! 2DUP DUP C@ SWAP 1+ C@ ROT
  C@ + $LEN-CHECK
  2DUP
  1+ DUP C@ + 1+
  SWAP DUP C@ >R 1+ SWAP R> CMOVE
  SWAP C@ SWAP 1+ C+! ;
```

$= (A1 A2 – F)

Compare two strings. (Equal)

Stack on Entry: (A1) – Address of a string.
(A2) – Address of a string.

Stack on Exit: (F) – A Boolean flag.

Example of Use:

```
MY_NAME DISK-NAME $@ $= .
```

This code would print a Boolean flag on the display, 0 (or false) if the two strings were not equal, –1 (or true) if they were.

Algorithm: First, determine if the lengths of the two strings are the same; if not, exit the word with a false flag on the stack. If they are of the same length, use –TEXT, which will compare the strings.

Suggested Extensions: None.

Definition:

```
: $= 2DUP C@ SWAP C@ = IF
     1+ SWAP DUP C@ SWAP 1+ -TEXT NOT
  ELSE
     2DROP 0
  ENDIF ;
```

$< (A1 A2 – F)

Compare two strings. (Greater Than).

Stack on Entry: (A1) – Address of a string.
(A2) – Address of a string.

Stack on Exit: (F) – A Boolean flag.

Example of Use:

```
MY_NAME DISK-NAME $@ $< .
```

This code would print a Boolean flag on the display, –1 (or true) if the string DISK–NAME holds is greater lexically than MY_NAME, 0 (or false) if it is less than or equal to MY_NAME.

Algorithm: Compare the strings using the length of the smaller string. If a less than condition is found, exit the word with a true flag. If a greater than condition is found, exit with a true flag. If the two strings were found to be equal, compare the lengths and return that flag.

Suggested Extensions: None.

Definition:

```
: $< 2DUP 2DUP C@ SWAP C@ MIN >R
  1+ SWAP 1+ R> SWAP -TEXT DUP 0< IF
     2DROP DROP 0
  ELSE
     0> IF
        2DROP -1
     ELSE
        C@ SWAP C@ >
     ENDIF
  ENDIF ;
```

$> (A1 A2 – F)

Compare two strings (Less Than)

Stack on Entry: (A1) – Address of a string.
(A2) – Address of a string.

Stack on Exit: (F) – A Boolean flag.

Definition:

```
: $> SWAP $< ;
```

$<=

Compare two strings (Greater Than or Equal)

Stack on Entry: (A1) – Address of a string.
(A2) – Address of a string.

Stack on Exit: (F) – A Boolean flag.

Definition:

```
: $>= $< NOT ;
```

$>= (A1 A2 – F)

Compare two strings (Less Than or Equal)

Stack on Entry: (A1) – Address of a string.
(A2) – Address of a string.

Stack on Exit: (F) – A Boolean flag.

Definition:

```
: $<= $> NOT ;
```

$<>

Compare two strings (Not Equal)

Stack on Entry: (A1) – Address of a string.
(A2) – Address of a string.

Stack on Exit: (F) – A Boolean flag.

Definition:

```
: $<> $= NOT ;
```

FIND$ (A1 A2 – N)

Determine if one string is a substring of another.

Stack on Entry: (A1) – Address of a string. (String to look for.)
(A2) – Address of a string. (String to look in.)

Stack on Exit: (N) – Position substring found at. (Zero, if not found.)

Example of Use:

```
$CONSTANT ME ME"
ME MY__NAME FIND$ .
```

This code would print 3 on the display, the position of "ME" in "JAMES".

Algorithm: Loop through the string being searched, calling –TEXT once for each possible starting position. The length –TEXT uses will be the length of the string being searched for. If –TEXT returns a match, exit the loop and return the loop index.

Suggested Extensions: None.

Definition:

```
: FIND$ 0 LROT DUP C@ 0 DO
     DUP C@ I – 3PICK C@ < IF
        LEAVE
   ELSE
     OVER DUP C@ SWAP 1+ SWAP
     ?DUP IF
        3PICK I 1+ + –TEXT NOT
     ELSE
        DROP 0
     ENDIF
     IF
```

```
      2DROP DROP I 1+ 0, LEAVE
    ENDIF
  ENDIF
LOOP 2DROP ;
```

NUM$ (N – A)

Convert a number to a string.

Stack on Entry: (N) – Integer

Stack on Exit: (A) – Address of a string.

Example of Use:

```
456 NUM$ 1 LEFT$ $.
```

This code would print a 4, the leftmost character of the string "456".

Algorithm: Use the built–in Forth conversions words.

Suggested Extensions: Define a word that produces a string corresponding to the currency form of a number, such as "$456.00" for the above 456.

Definition:

```
: INUM$ <# #S SIGN #> OVER 1- C! 1- ;

: NUM$ DUP ABS 0 INUM$ ;
```

DNUM$ (D – A)

Convert a double-length number to a string.

Stack on Entry: (D) – A double–length number.

Stack on Exit: (A) – Address of a string.

Definition:

```
: DNUM$ SWAP OVER DABS INUM$ ;
```

DVAL (A – D)

Convert a string to a double-length number.

Stack on Entry: (A) – Address of a string.

Stack on Exit: (D) – Double-length number.

Example of Use:

```
452 NUM$ DVAL D.
```

This code would print the number "452" on the display.

Algorithm: Check the string for a leading negative sign, then use the Forth word >BINARY to do the conversion. If a negative sign was found, negate the result that >BINARY leaves on the stack.

Suggested Extensions: None.

Definition:

```
SCONSTANT "–" –"

: DVAL DUP >R "–" SWAP FIND$ 1 = IF
    R> 1+ –1 >R
  ELSE
    R> 0 >R
  ENDIF
  0, ROT >BINARY DROP R> IF
    DNEGATE
  ENDIF ;
```

VAL (A – N)

Convert a string to a single-length number.

Stack on Entry: (A) – Address of a string.

Stack on Exit: (N) – A single-length number.

Definition:

```
: VAL DVAL DROP ;
```

SORT (N1 N2 –)

Sort a string array.

Stack on Entry: (N1) – Index of first array position.
(N2) – Index of second array position.

Stack on Exit: Empty.

Example of Use:

```
0 9 SORT
```

This code would sort the array NAMES.

Algorithm: The quicksort Algorithm can be found in Wirth[76]. We must provide the SORT word with compare and exchange operations for strings. The words .EXCHANGE, .<COMPARE, .>COMPARE do this.

Suggested Extensions: Implement another vector to allow SORT to sort any string array. Moving each string in the exchange word is a time-consuming process, SORT could be extended to use an index array.

Definition:

```
0 VARIABLE [<COMPARE]
0 VARIABLE [>COMPARE]
0 VARIABLE [EXCHANGE]
0 VARIABLE [X!]

: <X-COMPARE [<COMPARE] @ EXECUTE ;
: >X-COMPARE [>COMPARE] @ EXECUTE ;
: EXCHANGE [EXCHANGE] @ EXECUTE ;
: X! [X!] @ EXECUTE ;

: LEFTSWEEP
   SWAP BEGIN
      DUP <X-COMPARE
   WHILE
      1+
   REPEAT SWAP ;
: RIGHTSWEEP
   BEGIN
      DUP >X-COMPARE
   WHILE
      1-
   REPEAT ;

: SORT
   2DUP 2DUP + 2/ X! BEGIN
```

```
    LEFTSWEEP RIGHTSWEEP
    2DUP <= IF
      2DUP EXCHANGE
      1- SWAP 1+ SWAP
    ENDIF
    2DUP >
  UNTIL
  >R ROT R>
  2DUP < IF
    SORT
  ELSE
    2DROP
  ENDIF
  SWAP 2DUP < IF
    SORT
  ELSE
    2DROP
  ENDIF ;

64 10 $ARRAY NAMES
64 $VARIABLE TEMP
64 $VARIABLE EXCH

: .X! NAMES $@ TEMP $! ;

: .<COMPARE
NAMES $@ TEMP $@ $< ;

: .>COMPARE
  NAMES $@ TEMP $@ > ;

: .EXCHANGE
  DUP NAMES $@ EXCH $!
  OVER NAMES $@ SWAP NAMES $!
  EXCH $@ SWAP NAMES $! ;

' .<COMPARE [<COMPARE] !
' .>COMPARE [>COMPARE] !
' .EXCHANGE [EXCHANGE] !
' .X! X! !
```

8

Input Formatting

Words Defined in This Chapter:

MININ	A byte variable used to specify the minimum number of characters allowed by GET–INPUT.
MAXIN	A byte variable used to specify the maximum number of characters allowed by GET–INPUT.
LEGAL–CHARS	A string variable used to specify the characters GET–INPUT will allow to be entered.
INPUT	A string variable that holds the string gotten by GET–INPUT.
OK–TO–BEEP	A Boolean variable, true if GET–INPUT should beep on an error condition, false otherwise.
KILL–CHAR	A byte constant for the keyboard character that will erase a line in GET–INPUT.
RETURN–CHAR	A byte constant for the keyboard character that will cause GET–INPUT to exit.
BACKSPACE	A byte constant for the keyword character that will cause GET–INPUT to back up one space.
ASCII–CHAR?	Determine if a character is a printable ASCII character.
LEGAL?	Determine if a character will be valid for GET–INPUT.
NULL$SET	Set a string variable to the empty string.
CHOP1	Remove the rightmost character from a string.

DESTRUCT	Erase the character at the current cursor position and place the cursor one to the left of the current cursor position.
RETURN-OK?	Leave a true flag if a return is legal in GET-INPUT.
BACKSPACE-OK?	Leave a true flag if a backspace is legal in GET-INPUT.
ANY?	Leave a true flag if any character is legal in GET-INPUT.
/KILL/	/Erase the current input field and move the cursor to the beginning of the input field.
/OK/	Store a valid character in the string INPUT.
/BACKSPACE/	Handle a backspace for GET-INPUT.
/INIT/	Initialize values for GET-INPUT.
?BEEP	Beep if the variable OK-TO-BEEP holds a true flag.
GET-INPUT	Get input from the keyboard.
INT-INP	Input an integer.
DINT-INP	Input a double-length integer.
FP-INP	Input a floating-point number.
MINVAL	A variable used to specify the minimum allowed integer for INT-BOUNDED-INP.
MAXVAL	A variable used to specify the maximum allowed integer for INT-BOUNDED-INP.
INT-BOUNDED-INP	Input an integer that falls within a specified range.
MONTH	Prompt the user for, and input, a month.
DAY	Prompt the user for, and input, a day.
YEAR	Prompt the user for, and input, a year.
MDY-INPUT	Input a date.
AM/PM	Ask for the strings "AM" or "PM".
HOUR	Prompt the user for, and input, an hour.
MINUTE	Prompt the user for, and input, a minute.
TIME-INP	Input a time in military format.
(Y/N)	Ask for a "Y" or "N" key.
SINGLE-KEY	Ask for a specific set of single keys.
$->UPPER	Convert a string to uppercase.

This chapter presents a set of words to deal with input from the keyboard. There is a infamous acronym in computer science, GIGO, that stands for "Garbage In, Garbage Out." The words in this chapter will stop garbage from ever getting into your programs.

The words in this chapter will function by restricting the user. When we want a number to be input, we will only allow the number keys to be active. If we want a month to be input, we'll only allow the integers between 1 and 12. The word GET–INPUT will be our workhorse. It will be able to place a minimum and maximum on the number of characters to be entered. It will be able to restrict the keys active on the keyboard to any set we specify. By using it as a base we will be able to make sure we get exactly what we want from the user.

The words in this chapter make use of the string words presented in the previous chapter.

Suggested Extensions: This chapter's set of words is quite complete. One possible extension would be to implement a keyboard macro facility into the input routine. This could be accomplished by having GET–INPUT look up characters in an array of strings, and replacing the character with the string.

MININ (– A)

A byte variable used to specify the minimum number of characters allowed by GET–INPUT. Valid range: 0 to 255.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of MININ.

Example of Use:

```
2 MININ C!
```

Storing a two in MININ will cause GET–INPUT to require that at least two characters are entered.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE MININ
```

MAXIN (– A)

A byte variable used to specify the maximum number of characters allowed by GET–INPUT. Valid range: 1 to 255.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of MAXIN.

Example of Use:

```
12 MAXIN C!
```

Storing a twelve in MAXIN will cause GET–INPUT to limit the number of characters entered to twelve.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
255 CVARIABLE MAXIN
```

LEGAL–CHARS (– A)

A string variable that will specify the characters GET–INPUT will allow to be entered.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of LEGAL–CHARS.

Example of Use:

```
DIGITS LEGAL–CHARS $!
```

The string DIGITS is defined as "0123456789–". Storing the string DIGITS in LEGAL–CHARS will cause GET–INPUT to allow only numeric input.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
128 $VARIABLE LEGALCHARS
```

INPUT (– A)

A string variable that will hold the string gotten by GET–INPUT.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of INPUT.

Example of Use:

```
GET–INPUT INPUT $?
```

This code would print the value obtained by GET–INPUT on the display.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
255 $VARIABLE INPUT
```

OK–TO–BEEP (– A)

A Boolean variable, true if GET–INPUT should beep on an error condition, false if it should not.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of OK–TO–BEEP.

Example of Use:

```
OK–TO–BEEP C1SET
```

Setting OK–TO–BEEP to true will cause GET–INPUT to beep if an illegal character is entered, or if too many or too few characters are entered.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE OK-TO-BEEP
```

KILL-CHAR (- N)

A byte constant for the keyboard character that will erase a line in GET-INPUT.

Stack on Entry: Empty.

Stack on Exit: (A) – The value of KILL-CHAR.

Example of Use: See GET-INPUT.

Algorithm: The KILL-CHAR specified by this constant will cause GET-INPUT to erase all the characters typed in so far, and place the cursor back to the start of the input field. KILL-CHAR has been set for the escape key on the IBM-PC keyboard.

Suggested Extensions: KILL-CHAR could be changed to use another key if desired.

Definition:

```
27 CCONSTANT KILL-CHAR
```

RETURN-CHAR (- N)

A byte constant for the keyboard character that will cause GET-INPUT to accept the characters typed in.

Stack on Entry: Empty.

Stack on Exit: (A) – The value of RETURN-CHAR.

Example of Use: See GET-INPUT.

Algorithm: GET-INPUT accepts the RETURN-CHAR to end input.

Suggested Extensions: RETURN-CHAR could be changed to another value for special situations. For example, the plus (+) key could be used on the IBM-PC for input using the numeric keypad.

Definition:

```
13 CCONSTANT RETURN-CHAR
```

BACKSPACE (– B)

A byte constant for the keyboard character that will cause GET–INPUT to back up one space.

Stack on Entry: Empty.

Stack on Exit: (A) – The value of BACKSPACE.

Example of Use: See GET–INPUT.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
8 CCONSTANT BACKSPACE
```

ASCII–CHAR? (C – F)

Determine if a character is a printable ASCII character.

Stack on Entry: (C) The character to check.

Stack on Exit: (F) – A flag, true if the character was a printable ASCII character, false otherwise.

Example of Use:

```
KEY ASCII-CHAR?
```

This code would print a –1 if the value returned by KEY was a legal ASCII value, otherwise it would print a zero.

Algorithm: See if the value of the character lies in the range 32–128.

Suggested Extensions: None.

Definition:

```
: ASCII-CHAR? C(C-F)
   DUP 31 > SWAP 129 < AND;
```

LEGAL? (C – F)

Determine if a character will be valid for GET-INPUT.

Stack on Entry: (C) The character to check.

Stack on Exit: (F) A flag, true if the character is valid for GET-INPUT, false otherwise.

Example of Use:

```
KEY LEGAL?
```

This code would print a −1 if the value returned by KEY was valid for GET-INPUT; otherwise, it would print a zero.

Algorithm: If the string LEGAL-CHARS is empty, all characters are valid. If the string is not empty, check to see if the character is contained in the string.

Suggested Extensions: None.

Definition:

```
: LEGAL? C( C – F)
LEGAL-CHARS $@ LEN 0= IF
   DROP -1
ELSE
   CHR$ LEGAL-CHARS FIND$ NOT NOT
ENDIF ;
```

NULL$SET (A –)

Set a string variable to the empty string.

Stack on Entry: (A) The address of the string variable.

Stack on Exit: Empty.

Example of Use:

```
LEGAL-CHARS NULL$SET
```

This would set the string LEGAL-CHARS to the empty string. This would cause GET-INPUT to allow any ASCII character to be input.

Algorithm: Store a zero in the string's length byte.

Suggested Extensions: None.

Definition:

```
: NULL$SET 1+ C0SET ;
```

CHOP1 (A -)

Remove the rightmost character from a string.

Stack on Entry: (A) The address of the string.

Stack on Exit: Empty.

Example of Use:

```
MY-NAME $@ CHOP1 MY-NAME $?
```

If the string MY-NAME contained the value "JAMES" before the above code was executed, "JAME" would be printed on the display by the above code.

Algorithm: Take the left string using the length minus one.

Suggested Extensions: None.

Definition:

```
: CHOP1 $@ DUP LEN 1- LEFTS ;
```

DESTRUCT (-)

Erase a character at the current cursor position and place the cursor one space to the left of the current position.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Emit a blank to wipe out the current character. Then emit two backspaces to place the cursor one to the left of the former current character.

Suggested Extensions: None.

Definition:

```
: DESTRUCT BL EMIT BACKSPACE EMIT
  BACKSPACE EMIT ;
```

RETURN-OK? (- F)

Leave a true flag if a return is legal in GET-INPUT.

Stack on Entry: Empty.

Stack on Exit: (F) A flag, true if return is legal, false otherwise.

Example of Use: See words defined below.

Algorithm: If the length of the input string INPUT is greater than or equal to the minimum number of characters allowed, as specified by MININ, then a return is valid.

Suggested Extensions: None.

Definition:

```
: RETURN-OK?
  INPUT S@ LEN MININ C@ >= ;
```

BACKSPACE-OK? (- F)

Leave a true flag if a backspace is legal in GET-INPUT.

Stack on Entry: Empty.

Stack on Exit: (F) A flag, true if a backspace is legal, false otherwise.

Example of Use: See words defined below.

Algorithm: If we are not at the left hand side of the input field, then a backspace is legal. INPUT will be an empty string, with a length of zero, if we are at the left–hand side of the input field.

Suggested Extensions: None.

Definition:

```
: BACKSPACE-OK?
  INPUT $@ LEN 0 <> ;
```

ANY? (– F)

Leave a true flag if any characters are legal in GET–INPUT.

Stack on Entry: Empty.

Stack on Exit: (F) A flag, true if any characters are legal, false otherwise.

Example of Use: See words defined below.

Algorithm: If we are at the right-hand side of the input field, no more characters can be allowed. This condition exists when the length of INPUT is equal to the value in MAXIN.

Suggested Extensions: None.

Definition:

```
: ANY?
  INPUT $@ LEN MAXIN C@ <> ;
```

/KILL/ (–)

Erase the current input line and move the cursor to the start of the input field.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: If INPUT is not a null string, loop through its length, wiping out each character on the display. If INPUT is a null string, the line is already empty, so don't do anything.

Suggested Extensions: None.

Definition:

```
: /KILL/
  INPUT $@ LEN ?DUP IF
    0 DO DESTRUCT LOOP
    BL EMIT BACKSPACE EMIT
  ENDIF
  INPUT NULL$SET ;
```

/OK/ (C –)

Store a valid character in the string INPUT.

Stack on Entry: (C) The character to store in INPUT.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Use the string concatenation word to append the character onto INPUT.

Suggested Extensions: None.

Definition:

```
: /OK/ C( C – )
  DUP EMIT CHR$ INPUT $+! ;
```

/BACKPSACE/ (–)

Handle a backspace for GET–INPUT.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Wipe out a character on the screen, then remove it from INPUT.

Suggested Extensions: None.

Definition:

```
: /BACKSPACE/
   DESTRUCT INPUT CHOP1 ;
```

/INIT/ (–)

Initialize values for GET–INPUT.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: The way GET–INPUT is currently defined, the only initialization necessary is to set the string INPUT to the empty string.

Suggested Extensions: None.

Definition:

```
: /INIT/
   INPUT NULL$SET ;
```

?BEEP (–)

Beep if the variable OK–TO–BEEP holds a true flag.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Check the variable, and emit an ASCII bell character if it is true.

Suggested Extensions: None.

Definition:

```
: ?BEEP OK-TO-BEEP C@ IF 7 EMIT ENDIF ;
```

GET-INPUT (-)

Get input from the keyboard.

Minimum length, maximum length, and restricted characters are supported.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
1 MININ C! 1 MAXIN C! DIGITS LEGAL-CHARS $! GET-INPUT
```

This code would get input from the keyboard. It would require one character, and only allow digits to be input. The string input, in this case only a single character in length, would be returned in INPUT.

Algorithm: Begin by initializing the necessary values. Then, start an endless loop of inputting a character from the keyboard. First, check if the key hit was the KILL-CHAR; if it was, call /KILL/ to wipe out the current line. Next, check for the RETURN-CHAR. If the RETURN-CHAR was hit and a return is valid, exit the word. This is the only exit from the endless loop. Check for the BACKSPACE character and execute /BACKSPACE/ if it was found and a backspace is valid. Next, see if any character can be entered. If not continue the loop. If characters can be input, make sure we have a valid ASCII character. If the character is valid, store it in the string INPUT and continue the loop.

Suggested Extensions: The keyboard macros mentioned in the Introduction are one possible extension to this word. Another extension might be to allow the user to escape from the input without hitting return. A function key could

be used. This would give programs that use GET–INPUT the ability to allow users to exit from input screens easily.

Definition:

```
: GET-INPUT
  /INIT/ BEGIN
    KEY DUP KILL-CHAR = IF
      DROP /KILL/
    ELSE
      DUP RETURN-CHAR = IF
        DROP RETURN-OK? IF
          EXIT
        ELSE
          ?BEEP
        ENDIF
      ELSE
        DUP BACKSPACE = IF
          DROP BACKSPACE-OK? IF
            /BACKSPACE/
          ELSE
            ?BEEP
          ENDIF
        ELSE
          ANY? NOT IF
            DROP ?BEEP
          ELSE
            DUP ASCII-CHAR? NOT IF
              DROP ?BEEP
            ELSE
              DUP LEGAL? NOT IF
              01DROP ?BEEP
              ELSE
              1/OK/
              ENDIF
            ENDIF
          ENDIF
        ENDIF
      ENDIF
    ENDIF
  0 UNTIL ;
```

0$ (A –)

Zero the unused portions of a string variable.

Stack on Entry: (A) The address of the string variable.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Use the length of the string and how much space is allocated to it to determine how many bytes to zero out. This is done so that the word >BINARY will terminate properly when passed a string.

Suggested Extensions: None.

Definition:

```
: 0$ DUP C@ OVER 1+ C@ - SWAP
  DUP 1+ C@ + 2+ SWAP ERASE ;
```

INT-INP (- N)

Input an integer.

Stack on Entry: Empty.

Stack on Exit: (N) The integer input.

Example of Use:

```
1 MININ C! 20 MAXIN C! INT-INP
```

This would get an integer from the keyboard; at least one, and a maximum of twenty characters would be input.

Algorithm: Set the legal characters to the digits and the negative sign. Start a loop calling GET-INPUT. Take the string returned and see if it has a leading negative sign. If it does, place a negative one on the return stack. If it does not place a one there. Call >BINARY to convert the string to an integer. If >BINARY has stopped on a zero byte (not an ASCII zero), then the conversion was successful. If the conversion was successful, multiply the result by the one or negative one placed on the return stack, and leave a true on the stack to terminate the loop. If the conversion was unsuccessful, clear the stacks and call ?BEEP and /KILL/ to move the cursor back to the start of the input field. Leave a zero on the stack. This will cause the loop to repeat and another GET-INPUT will be executed.

Suggested Extensions: None.

Definition:

```
$CONSTANT DIGITS -0123456789"
45 CONSTANT .KEY

: INT-INP
  DIGITS LEGAL-CHARS $! BEGIN
    GET-INPUT INPUT 0$
    INPUT $@ DUP 1+ C@ .KEY = IF
      -1 >R 1+
    ELSE
      1 >R
    ENDIF
    0, ROT >BINARY C@ .KEY = IF
      R> DROP 2DROP ?BEEP /KILL/ 0
    ELSE
      DROP R> * -1
    ENDIF
  UNTIL ;
```

DINT-INP (- D)

Input a double-length integer.

Stack on Entry: Empty.

Stack on Exit: (D) The double-length integer input.

Example of Use:

```
1 MININ C! 20 MAXIN C! DINT-INP D
```

This would get a double-length integer from the keyboard; at least one, and a maximum of twenty characters would be input. The result would then be printed on the display.

Algorithm: Set the legal characters to the digits and the negative sign. Start a loop calling GET-INPUT. Take the string returned and see if it has a leading negative sign. If it does, place a negative one on the return stack. If it does not, place a one there. Call >BINARY to convert the string to an integer. If >BINARY has stopped on a zero byte (not an ASCII zero), then the conversion was successful. If the conversion was successful multiply the result by the one or negative one placed on the return stack and leave a true on the stack to terminate the loop: If the conversion was unsuccessful clear the stacks and

call ?BEEP and /KILL/ to move the cursor back to the start of the input field. Leave a zero on the stack. This will cause the loop to repeat and another GET–INPUT will be executed.

Suggested Extensions: None.

Definition:

```
: DINT–INP
  DIGITS LEGAL–CHARS $! BEGIN
    GET–INPUT INPUT 0$
  INPUT $@ DUP 1+ C@ .KEY = IF
    –1 >R 1+
  ELSE
    1 >R
  ENDIF
  0, ROT >BINARY C@ .KEY = IF
    R> DROP 2DROP ?BEEP /KILL/ 0
  ELSE
    R> 1 M*/ –1
  ENDIF
UNTIL ;
```

FP–INP (– R)

Input a floating–point number.

Stack on Entry: Empty.

Stack on Exit: (R) The floating-point number input.

Example of Use:

```
1 MININ C! 8 MAXIN C! FP–INP R.
```

This would get a floating-point number from the keyboard, at least one, and a maximum of twenty characters would be input. The result would then be printed on the display.

Algorithm: This word requires the floating-point number words to have been loaded. Set the legal characters to the digits, the negative sign and the decimal point. Start a loop calling GET–INPUT. Pass the INPUT string to FNUM. FNUM leaves a flag on the stack indicating whether or not it was able to convert the string to a floating point number. If the conversion was successful, leave a true on the stack to terminate the loop. If the conversion was unsuc-

cessful, clear the stacks and call ?BEEP and /KILL/ to move the cursor back to the start of the input field. Leave a zero on the stack. This will cause the loop to repeat and another GET–INPUT will be executed.

Suggested Extensions: None.

Definition:

```
$CONSTANT FDIGITS 0123456789-."

    : FP-INP

    FDIGITS LEGAL-CHARS $I BEGIN
       GET-INPUT INPUT 0$
       INPUT $@ FNUM IF
          -1
       ELSE
          ?BEEP /KILL/ 0
       ENDIF
  UNTIL ;
```

MINVAL (– A)

A variable used to specify the minimum allowed integer for INT–BOUNDED–INP.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of MINVAL.

Example of Use:

```
1 MINVAL !
```

Storing a one in MINVAL will cause INT–BOUNDED–INPUT to only allow positive numbers to be input.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
-32767 VARIABLE MINVAL
```

MAXVAL (– A)

A variable used to specify the maximum allowed integer for INT–BOUNDED–INP.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of MAXVAL.

Example of Use:

```
0 MAXVAL !
```

Storing a zero in MAXVAL will cause INT–BOUNDED–INPUT to only allow negative numbers to be input.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
32767 VARIABLE MAXVAL
```

INT–BOUNDED–INP (– N)

Input an integer that falls within a specified range.

Stack on Entry: Empty.

Stack on Exit: (N) The integer input.

Example of Use:

```
7 MINVAL ! 8 MAXVAL ! 1 MININ C! 1 MAXIN C! INT–BOUNDED–INP
```

This code would force a seven or an eight to be input by setting the minimum value to seven and the maximum value to eight.

Algorithm: Start a loop calling INT–INP. Check the returned value against MINVAL and MAXVAL. If the result does not lie between them, clear the stack and call ?BEEP and /KILL/ to move the cursor back to the start of the input field. Continue the loop. If the result was within the proper range, exit the loop.

Suggested Extensions: Clone this word for double-length or floating-point numbers as needed.

Definition:

```
: INT-BOUNDED-INP
  BEGIN
    INT-INP DUP DUP MINVAL @ >=
    SWAP MAXVAL @ <= AND NOT
  WHILE
    ?BEEP /KILL/ DROP
  REPEAT ;
```

MONTH (– N)

Prompt the user for, and input, a month.

Stack on Entry: Empty.

Stack on Exit: (N) The integer input, in the range 1 to 12.

Example of Use: See MDY–INPUT below.

Algorithm: Set the minimum field length to one, the maximum to two. Set the minimum input value to one, the maximum to twelve. Print out the prompt and call INT–BOUNDED–INP.

Suggested Extensions: None.

Definition:

```
: MONTH
  1 MININ C! 2 MAXIN C!
  1 MINVAL ! 12 MAXVAL !
  ." MONTH: " INT-BOUNDED-INP ;
```

DAY (N1 – N2)

Prompt the user for, and input, the day.

Stack on Entry: (N1) The month as returned by MONTH.

Stack on Exit: (N2) The day input.

Example of Use: See MDY–INPUT below.

Algorithm: Set the minimum field length to one, the maximum to two. Set the minimum input value to one. Look up the maximum value in the array DPM (days per month). Print out the prompt and call INT–BOUNDED–INP.

Suggested Extensions: None.

Definition:

```
31 CVARIABLE DPM 29 C, 31 C, 30 C,
31 C, 30 C, 31 C, 31 C, 30 C,
31 C, 30 C, 31 C,

: DAY
   1– DPM + C@ MAXVAL !
   ." DAY: " INT–BOUNDED–INP ;
```

YEAR (– N)

Prompt the user for, and input, the year.

Stack on Entry: Empty.

Stack on Exit: (N) The year input.

Example of Use: See MDY–INPUT below.

Algorithm: Set the minimum field length to one, the maximum to five. Call INT–INP to get an integer.

Suggested Extensions: None.

Definition:

```
: YEAR
   1 MININ C! 5 MAXIN C!
   ." YEAR: " INT–INP ;
```

MDY–INP (– N1 N2 N3)

Input a date.

Stack on Entry: Empty.

Stack on Exit: (N1) The year input.

(N2) The day input.
(N3) The month input.

Example of Use: None.

Algorithm: First call MONTH. Then call DAY with a copy of the month. Finally, call YEAR.

Suggested Extensions: None.

Definition:

```
: MDY-INP
  MONTH DUP DAY YEAR ;
```

AM/PM (N1 – N2)

Ask for the strings AM or PM, and adjust the time on the stack accordingly.

Stack on Entry: (N1) A military time in the range zero to 1159.

Stack on Exit: (N2) The time adjusted for AM or PM.

Example of Use: See TIME–INP below.

Algorithm: Loop until either "AM" or "PM" is input by the user. If "PM" is input, add 1200 to the time on the stack.

Suggested Extensions: None.

Definition:

```
SCONSTANT AMPMS APM"
SCONSTANT AMS AM"
SCONSTANT PMS PM"

:AM/PM?
  AMPMS LEGAL-CHARS $!
  2 MININ C! 2 MAXIN C!
  ." AM OR PM: " BEGIN
    GET-INPUT
    INPUT $@ DUP AMS $= SWAP
    PMS $= OR NOT
  WHILE
```

```
   ?BEEP /KILL/
REPEAT
INPUT $@ PM$ $= IF
   1200 +
ENDIF ;
```

HOUR (– N)

Prompt for, and input, the hour from the user.

Stack on Entry: Empty.

Stack on Exit: (N) A military time for the hour input.

Example of Use: See TIME–INP below.

Algorithm: Allow only values from one to twelve to be input. If a twelve is input, convert it to zero. Multiply the value by 100.

Suggested Extensions: None.

Definition:

```
: HOUR
   1 MININ C! 2 MAXIN C!
   20 MINVAL ! 12 MAXVAL !
   ." HOUR: " INT–BOUNDED–INP
   DUP 12 = IF 12 – ENDIF
   100 * ;
```

MINUTE (N1 – N2)

Prompt for, and input, the minute from the user.

Stack on Entry: (N1) A military time for the hour just input.

Stack on Exit: (N2) The minutes added to the time.

Example of Use: See TIME–INP below.

Algorithm: Allow only values from zero to 59 to be input.

Suggested Extensions: None.

Definition:

```
: MINUTE
  59 MAXVAL !
  ." MINUTE: " INT-BOUNDED-INP
  + ;
```

TIME-INP (- N)

Input a time in military format.

Stack on Entry: Empty.

Stack on Exit: (N) The time in military format.

Example of Use: None.

Algorithm: Call HOUR, MINUTE, and AM/PM?.

Suggested Extensions: None.

Definition:

```
: TIME-INP
  HOUR MINUTE AM/PM? ;
```

(Y/N) (- F)

Give the user the prompt "(Y/N)" and input a Y or N key. Leave a true flag if a "Y" was hit, false if a "N" was hit.

Stack on Entry: Empty.

Stack on Exit: (F) A flag, true if "Y" was hit, false if "N" was hit.

Example of Use: (Y/N)

This code would print a −1 if "Y" is hit, zero if "N" is hit after the prompt "(Y/N)" is printed on the display.

Algorithm: Print the prompt. Start a loop of inputting a key.

Convert the key hit to uppercase. If it is a "Y" or an "N," exit the loop. EMIT

the character onto the display. Compare the character to "Y" and leave the flag on the stack.

Suggested Extensions: None.

Definition:

```
: (Y/N) ." (Y/N) " 1 BEGIN
    DROP KEY DUP 95 > IF 32 - ENDIF
    DUP DUP 89 = SWAP 78 = OR
  UNTIL
  DUP EMIT 89 = ;
```

SINGLE-KEY (- C)

Allow a user to hit a single key, specified by the characters in the string LEGAL-CHAR.

Stack on Entry: Empty.

Stack on Exit: (C) The character hit.

Example of Use:

```
ABC LEGAL-CHAR $!
." RATE MEAL (A/B/C) " SINGLE-KEY
```

The string ABC is "ABC". This code would allow the user to press either "A", "B", or "C" after the prompt "RATE MEAL (A/B/C)" was printed on the display.

Algorithm: Start a loop of inputting a key. Convert the key to uppercase. See if the key is in the string LEGAL-CHARS. If it is not, continue the loop. If it is, exit the loop and EMIT the character on the display. Leave the key on the stack.

Suggested Extensions: None.

Definition:

```
: SINGLE-KEY 1 BEGIN
    DROP KEY DUP 95 > IF 32 - ENDIF
    DUP CHRS LEGAL-CHARS FINDS
  UNTIL DUP EMIT ;
```

$->UPPER (A -)

Convert a string to all uppercase.

Stack on Entry: (A) The address of the string to convert.

Stack on Exit: Empty.

Example of Use:

```
CITY $->UPPER CITY $?
```

If the string CITY contained "New York" prior to the above code being executed, then the value "NEW YORK" would be printed on the display by the above code.

Algorithm: Loop through the string. If a character has an ASCII code above 95, it is lowercase. Subtract 32 to convert it to uppercase and store it back in the string.

Suggested Extensions: None.

Definition:

```
: $->UPPER
  DUP C@ 0 DO
    1+ DUP C@ 95 > IF -32 OVER C+! ENDIF
  LOOP DROP ;
```

Displays and Output Formating

Words Defined in This Chapter:

ATTRIBUTE	A byte variable holding the current attribute byte.
WT []	Access window table array.
FREE_TABLE	Find a free entry in window table.
CREATE_WINDOW	Create a window.
W_TABLE_ROTATE	Rearrange the order of the window table.
SAVE_WINDOW	Save the contents of a window.
RESTORE_WINDOW	Restore the contents of a window.
SCROLL	Scroll the current window.
WEMIT	Emit a character onto the current window.
WTYPE	Type a string of characters onto the current window.

W"	Enclose a literal to be printed on the current window.
WQUERY	Input a line on the current window.
NORMAL	Display normal characters.
REVERSE	Display reverse or inverse characters.
BLINKING	Display blinking characters.
BLUE	Display blue characters.
GREEN	Display green characters.
RED	Display red characters.
CYAN	Display cyan characters.
BROWN	Display brown characters.
MAGENTA	Display magenta characters.
GRAY	Display gray characters.
L_BLUE	Display light blue characters.
L_GREEN	Display light green characters.
L_CYAN	Display light cyan characters.
L_RED	Display light red characters.
L_MAGENTA	Display light magenta characters.
YELLOW	Display yellow characters.
BRIGHT_WHITE	Display bright white characters.

In this chapter we present a complete set of words to manage display windows on your IBM-PC. Normally the screen of your IBM-PC consists of 25 lines of 80 characters. A window is a rectangular portion of that screen in which all character input and output will take place. A window could be as large as the normal screen, or as small as one line of one character. The windows we will define can overlay each other and be stacked to any depth. The use of windows has become popular in computers such as Apple's Macintosh. Windows have proved to be an intuitive model of how people work, and with these words, the IBM-PC can exploit their full power.

We will make extensive use of the extra memory available on the IBM-PC for these window words. We will use a segment outside the normal 64K address space of Forth to hold saved windows. We will use another segment to hold text literals to be displayed on windows. The actual addresses these segments should reside at are highly dependent on your particular version of Forth. The code presented uses the word >X, which returns the segment address of the basic Forth system in Atila. They then use the memory after the base 64K of the language as the extra segments needed. Most IBM-PC Forth's should have a word similar to >X; consult the manual that came with your version.

The windows we define will be able to overlay each other without restriction. In order to make this possible, we will save the complete contents of all

windows that are not currently displayed. The following values will be kept for each window:

Upper Left X Coordinate (0–79)
Upper Left Y Coordinate (0–24)
X Cursor Position (0–79)
Y Cursor Position (0–24)
Current Width of window (1–80)
Current Height of window (1–25)
Maximum Width of window (1–80)
Maximum Height of window (1–25)
Offset screen stored at in extra segment.
Integer identifier for window.

These values will be kept in an array, W_DATA. This array will be accessed by a number of words we define below. The current window being used for character input and output will be kept in the first portion of the array. The identifier given to each window will enable them to be accessed by name.

Suggested extensions: The window word set presented is fairly complete. One extension that might be aesthetically pleasing would be to automatically box all created windows with the double-line characters provided in the IBM character set. This could be done by changing created windows to reduce all dimensions by two, and adding the box drawing to the MAKE_CURRENT word.

Variables Used by Window Words:

```
( Physical screen segment )
( Use B800 for color display )
HEX B000 VARIABLE SCREEN_SEG DECIMAL

( Segment used to save windows )
HEX > X 2000 + VARIABLE SAVE_SEG DECIMAL

( Segment Offset used to save windows )
1 VARIABLE OFFSET

( Physical width of screen )
( Set to 40 on certain displays )
80 CVARIABLE PHYS_WIDTH

( Current attribute byte )
3 CVARIABLE ATTRIBUTE
```

```
( Maximum number of windows )
( Use any value, constrained only by memory)
7 CCONSTANT MAX__WIND

( Window table width )
11 CCONSTANT W__T__W

( Window data table )
0 VARIABLE W__DATA MAX__WIND W__T__W * DUP
ALLOT W__DATA SWAP ERASE
```

Array Accessing Words:

```
( An offset is defined for each table entry)
0 CCONSTANT X__UP__LEFT__OFS
1 CCONSTANT Y__UP__LEFT__OFS
2 CCONSTANT X__CUR__OFS
3 CCONSTANT Y__CUR__OFS
4 CCONSTANT X__CUR__LEN__OFS
5 CCONSTANT Y__CUR__LEN__OFS
6 CCONSTANT X__MAX__LEN__OFS
7 CCONSTANT Y__MAX__LEN__OFS
8 CCONSTANT OFS__OFS
10 CCONSTANT W__NAME__OFS

( These words allow access to the first array entry)
: FIELD <BUILDS C, DOES> C@ W__DATA + ;
X__UP__LEFT__OFS FIELD X__UP__LEFT
Y__UP__LEFT__OFS FIELD Y__UP__LEFT
XCUR__OFS FIELD XCUR
YCUR__OFS FIELD YCUR
X__CUR__LEN__OFS FIELD X__CUR__LEN
Y__CUR__LEN__OFS FIELD Y__CUR__LEN
X__MAX__LEN__OFS FIELD X__MAX__LEN
Y__MAX__LEN__OFS FIELD Y__MAX__LEN
OFS__OFS FIELD OFS
W__NAME__OFS FIELD W__NAME
```

WT [] (N – A)

Find the address of an entry in the window data array.

Stack on Entry: (N) Index to window table.

Stack on Exit: (A) Address of Nth entry in array.

Example of Use:

This would leave the address of the third window's data on the stack.

Algorithm: Multiply value by size of each array entry, then add the start of the array.

Suggested extensions: None.

Definition:

```
: WT [] W__T__W * W__DATA + ;
```

FREE__TABLE (– N)

Find a free entry in the window data array.

Stack on Entry: Empty.

Stack on Exit: (N) Index number of a free entry.

Example of Use:

```
FREE__TABLE .
```

This code would print the index number of a free entry in the window data array.

Algorithm: Place a 9999 on the stack. Loop through the array, looking for an empty name field; this signifies a free entry. If one is found, drop the 9999, leave its number on the stack, and exit the loop. At the end of the loop, see if the top of the stack is 9999; if it is, no free entries were found.

Suggested Extensions: None.

Definition:

```
: FREE__TABLE
  9999 MAX__WIND 0 DO
    I WT [] W__NAME__OFS +
    C@ 0= IF
      DROP I LEAVE
    ENDIF
```

```
LOOP
DUP 9999 = ABORT" Too many windows." ;
```

CREATE_WINDOW (N1 N2 N3 N4 N5 –)

Create an entry in the window data array for a new window.

Stack on Entry: (N1) – Upper left X coordinate of window.
(N2) – Upper left Y coordinate of window.
(N3) – Maximum width of window.
(N4) – Maximum height of window.
(N5) – Integer identifier for window.

Stack on Exit: Empty.

Example of Use:

```
5 5 40 15 5 CREATE_WINDOW
```

This code would create a window of 15 lines of 40 characters. The window would start 5 characters from the left edge of the physical screen, 5 lines from the top of the physical screen. The window would be referenced by the number five.

Algorithm: Search for a free table entry. Erase it when found, and then fill it in with the passed values.

Suggested Extensions: Check for duplicate identifiers.

Definition:

```
: CREATE_WINDOW
  FREE_TABLE WT []
  DUP W_T_W ERASE
  DUP >R W_NAME_OFS + C!
  R> DUP >R Y_MAX_LEN_OFS + C!
  R> DUP >R X_MAX_LEN_OFS + C!
  R> DUP >R Y_UP_LEFT_OFS + C!
  R> X_UP_LEFT_OFS + C! ;
```

W_TABLE_ROTATE (N –)

Rotate a window to the top of the window data array.

Stack on Entry: (N) Entry number to be rotated to the top.

Stack on Exit: Empty.

Example of Use:

2 W_TABLE_ROTATE

Rotate the third entry to the top of the window data array.

Algorithm: If the top is being rotated to itself, do nothing. Otherwise, save the entry to be rotated to the top in the PAD. Then, move the entries under that entry up by one entry each. When this is complete, move the saved entry from the PAD to the top of the array.

Suggested extensions: None.

Definition:

```
: W_TABLE_ROTATE
  ?DUP IF
     DUP WT [] PAD W_T_W CMOVE
     W_DATA DUP W_T_W + ROT W_T_W *
     <CMOVE
     PAD W_DATA W_T_W CMOVE
  ENDIF ;
```

SAVE_WINDOW

Save the contents of the current window to the save segment area.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

SAVE_WINDOW

This code would save the contents of the current window.

Algorithm: Loop on the current height of the window. Determine the start of each line using the upper left coordinates and then move the data to the extra segment using XCMOVE.

Suggested Extensions: None.

Definition:

```
: SAVE_WINDOW
  Y_CUR_LEN C@ 0 DO
    Y_UP_LEFT C@ I + PHYS_WIDTH C@ 2* *
    2X_UP_LEFT C@ 2* + SCREEN_SEG @
    OFS @ X_CUR_LEN C@ 2* I * +
    SAVE_SEG @
    X_CUR_LEN C@ 2* XCMOVE
  LOOP ;
```

RESTORE_WINDOW

Restore the contents of the current window from the save segment area.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RESTORE_WINDOW
```

This code would restore the contents of the current window.

Algorithm: Loop on the current height of the window. Determine the start of each line using the upper left coordinates and then move the data from the extra segment using XCMOVE.

Suggested Extensions: None.

Definition:

```
: RESTORE_WINDOW
  Y_CUR_LEN C@ 0 DO
    OFS @ X_CUR_LEN C@ 2* I * +
    SAVE_SEG @
    Y_UP_LEFT C@ I + PHY_SWIDTH C@ 2* *
    X_UP_LEFT C@ 2* + SCREEN_SEG @
    X_CUR_LEN C@ 2* XCMOVE
  LOOP ;
```

SCROLL

Scroll the current window.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
SCROLL
```

This code would scroll the current window.

Algorithm: Move each line up one by using the screen segment and X-CMOVE. Blank the bottom line using the constant BLANK, which is a blank character and a normal attribute.

Suggested Extensions: None.

Definition:

```
HEX 0320 CONSTANT BLANK DECIMAL

: LINE-ADDR
   Y_UP_LEFT C@ +
   PHYS_WIDTH C@ 2* *
   X_UP_LEFT C@ 2* +
   SCREEN_SEG @ ;

: SCROLL
   Y_CUR_LEN C@ 1 DO
      I LINE-ADDR
      I 1- LINE-ADDR
      X_CUR_LEN C@ 2* XCMOVE
   LOOP
   Y_CUR_LEN C@ 1- LINE-ADDR DROP
   X_CUR_LEN C@ 0 DO
      BLANK OVER
      SCREEN-SEG @ X! 2+
   LOOP DROP ;
```

WEMIT (C -)

Print a character on the current window.

Stack on Entry: (C) Character to print.

Stack on Exit: Empty.

Example of Use:

```
65 WEMIT
```

This code would print an 'A' on the current window at the current cursor position.

Algorithm: Check to see if the character is a carriage return. If it is, check to see if we must scroll. Set the X cursor position to zero. If the character was not a carriage return, place it on the screen. Increment the X cursor position. If it reaches the end of a line on the window, set it to zero and check for a possible scroll. If scroll is not needed, increment the Y cursor position.

Suggested Extensions: Add support for the backspace character.

Definition:

```
: WEMIT
  DUP 13 = IF
    Y_CUR_LEN C@ 1- YCUR C@ = IF
      SCROLL XCUR C0SET
    ELSE
      1 YCUR C+! XCUR C0SET
    ENDIF
  ELSE
    YCUR C@ LINE-ADDR DROP XCUR C@ 2* +
    DUP >R SCREEN-SEG @ XC!
    ATTRIBUTE C@ R> 1+ SCREEN-SEG @ XC!
    1 XCUR C+! XCUR C@ X_CUR_LEN C@ = IF
      Y_CUR_LEN C@ 1- YCUR C@ = IF
        SCROLL XCUR C0SET
      ELSE
        1 YCUR C+! XCUR C0SET
      ENDIF
    ENDIF
  ENDIF ;
```

WTYPE (A N -)

Print a string of characters on the current window.

Stack on Entry: (A) Address of characters to print.

(N) Number of bytes to print.

Stack on Exit: Empty.

Example of Use:

```
DISK_NAME 15 WTYPE
```

This code would print the 15 characters at DISK_NAME on the current window.

Algorithm: Make sure the count is greater than zero. If it is, loop through the characters calling WEMIT to print them on the display.

Suggested Extensions: None.

Definition:

```
: WTYPE
   DUP 0_ IF EXIT ENDIF
   0 DO I OVER + C@ WEMIT LOOP DROP ;
```

W"

Print a literal on the current window.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
: FINISH W" This is the end! " ;
```

This word would print "This is the end! " on the current window, when it is executed. W" can only be used inside a word definition.

Algorithm: Use WORD to parse the string. Store it in the extra memory pointed to by OFS_LITERAL and SEG_LITERAL. Cause the offset the word was saved at to be placed on the stack when the word is executed. W"EX will use this address and move the literal to the PAD. From the PAD the literal will be printed.

Suggested Extensions: None.

Definition:

```
HEX >X 2200 + VARIABLE SEG__LITERAL
0 VARIABLE OFS__LITERAL
DECIMAL

: W"EX SEG__LITERAL @ PAD >X 64 XCMOVE
   PAD COUNT WTYPE ;

34 CCONSTANT L"
: W" L" WORD DUP C@ 1+ >R
   >X OFS__LITERAL @ SEG__LITERAL @
   R> DUP >R XCMOVE
   OFS__LITERAL @ [COMPILE] LITERAL
   R> OFS__LITERAL +!
   COMPILE W"EX ;
IMMEDIATE
```

CLEAR_WINDOW

Fill the current window with blanks.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
CLEAR__WINDOW
```

This code would clear the current window.

Algorithm: Loop through the screen height and width storing the constant BLANK.

Suggested Extensions: Make a faster version by filling PAD with BLANK and usung XCMOVE.

Definition:

```
: CLEAR__WINDOW
     Y__CUR__LEN C@ 0 DO
       I LINE-ADDR DROP
```

```
    X_CUR_LEN C@ 0 DO
      BLANK OVER
      SCREEN-SEG @ X! 2+
    LOOP DROP
  LOOP ;
```

MAKE_CURRENT (N -)

Cause a specific window to become the current window.

Stack on Entry: (N) Identifier of window to be made current.

Stack on Exit: Empty.

Example of Use:

```
5 MAKE_CURRENT
```

This code would cause the window with the identifier five to become current.

Algorithm: Find the window in the table. Once found, determine whether this window has ever been used before by checking the offset field. If the offset is empty, this is the first time for this window. If this is the case, set the height and width to the their maximum sizes and clear the window. Then, allocate space in the save segment for the window. If the window had been displayed previously, restore the data from the save area to the screen.

Suggested Extensions: None.

Definition:

```
: MAKE_CURRENT
  9999 SWAP MAX_WIND 0 DO
    I WT [] W_NAME_OFS +
    C@ OVER = IF
      2DROP I I LEAVE
    ENDIF
  LOOP
  SWAP 0099 = ABORT" Window not found."
  SAVE_WINDOW
  W_TABLE_ROTATE
  OFS @ 0= IF
    Y_MAX_LEN C@ DUP Y_CUR_LEN C!
```

```
    X_MAX_LEN C@ DUP X_CUR_LEN C!
    XCUR C0SET YCUR C0SET
    2* * OFFSET @ OFS ! OFFSET +!
    CLEAR_WINDOW
  ELSE
    RESTORE_WINDOW
  ENDIF ;
```

WQUERY (–)

Input a string of characters from the keyboard and display it on the current window. Store it in the terminal input buffer.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
WQUERY
```

This code would allow a string of characters to be input.

Algorithm: Erase the input buffers and store a zero in the buffer position variable. Input characters until encountering a carriage return. If the end of the buffer is reached, force a carriage return.

Suggested Extensions: Implement a backspace.

Definition:

```
64 CCONSTANT MAX_CHAR
0 CVARIABLE TIB_POS

: WQUERY
  TIB MAX_CHAR ERASE TIB_POS C0SET BEGIN
    KEY DUP 13 <> TIB_POS C@ MAX_CHAR <> AND
  WHILE
    DUP WEMIT TIB TIB_POS C@ + C!
    1 TIB_POS C+!
  REPEAT DROP ;
```

Various Display Control Words:

These words modify the display attribute byte to allow various types of text output. Most are self-explanatory.

```
( Normal display, white on black )
: NORMAL 7 ATTRIBUTE C! ;

( Reverse, black on white)
: REVERSE 112 ATTRIBUTE C! ;

( Blinking, added to current )
: BLINKING ATTRIBUTE C@ 128 AND ATTRIBUTE C! ;

( Various colors )
: BLUE 1 ATTRIBUTE C! ;
: GREEN 2 ATTRIBUTE C! ;
: RED 4 ATTRIBUTE C! ;
: CYAN 3 ATTRIBUTE C! ;
: BROWN 6 ATTRIBUTE C! ;
: MAGENTA 5 ATTRIBUTE C! ;
: GRAY 8 ATTRIBUTE C! ;
: L_BLUE 9 ATTRIBUTE C! ;
: L_GREEN 10 ATTRIBUTE C! ;
: L_CYAN 11 ATTRIBUTE C! ;
: L_RED 12 ATTRIBUTE C! ;
: L_MAGENTA 13 ATTRIBUTE C! ;
: YELLOW 14 ATTRIBUTE C! ;
: BRIGHT_WHITE 15 ATTRIBUTE C! ;
```

10

Natural Language Processing

Words Defined in This Chapter:

VERB	A constant for the part-of-speech verb.
NOUN	A constant for the part-of-speech noun.
DET	A constant for the part-of-speech determinant.
ADJ	A constant for the part-of-speech adjective.
ADVERB	A constant for the part-of-speech adverb.
CONJ.	A constant for the part-of-speech conjunction.
WORD#	A variable used to keep track of the number of words in the vocabulary table.
VDATA	An array used to hold vocabulary data.
VTABLE	A variable that points to the next free position in VDATA.
SYNONYM	Cause the word entered next into the vocabulary table to be a synonym of the word previously entered in the vocabulary table.
VERB–FOUND	Word number of the verb found by the ATN.
NOUN–FOUND	Word number of the noun found by the ATN.
DET–FOUND	Word number of the determinant found by the ATN.
ADJ–FOUND	Word number of the adjective found by the ATN.
ADVERB–FOUND	Word number of the adverb found by the ATN.

!VOCABULARY	Store a word in the vocabulary table.
W#–LIST	An array of word numbers in the input sentence.
WPS–LIST	An array of the parts of speech of the words in the input sentence.
S–V–T	Search the vocabulary table.
POINTER	A pointer to the array's W#–LIST and WPS–LIST.
ADVANCE	Advance POINTER to the next word in the input sentence.
P.O.S.	Find the part of speech of the current word for the ATN.
#–OF–W	Find the word number of the current word for the ATN.
?VERB	Is the current word a verb?
?NOUN	Is the current word a noun?
?DET	Is the current word a determinant?
?ADJ	Is the current word an adjective?
?ADVERB	Is the current word an adverb?
FAKE–ADJ	A fake vocabulary entry for a numeric adjective.
#ADJ	The value of a numeric adjective.
#?	Is the string on the stack a number?
0–FOUNDS	Zero all the registers used by the ATN.
/INIT/	Initialize the arrays used by INPUT$LOOKUP.
INPUT$	Input a sentence and look up the words in it.
LOOKUP	
GET–GOOD–INPUT	Get a good sentence from the keyboard.
(N2)	Node N2 of the ATN.
(N1)	Node N1 of the ATN.
(NP)	Node NP of the ATN.
(3)	Node 3 of the ATN.
(2)	Node 2 of the ATN.
(1)	Node 1 of the ATN.
(S)	Node S of the ATN.
PARSE	Attempt to apply the ATN to an input sentence.
XPOS	The x position of the robot.
YPOS	The y position of the robot.
FACING	The compass facing of the robot.
MOVE	Move the robot.
TURN	Turn the robot.
MOVE–FORWARD?	Examine the ATN registers for MOVE FORWARD.
MOVE–BACK?	Examine the ATN registers for MOVE BACK.

TURN-LEFT?	Examine the ATN registers for TURN LEFT.
TURN-RIGHT?	Examine the ATN registers for TURN RIGHT.
BACKUP?	Examine the ATN registers for BACKUP.
ROBOT-POS	Print out the robot's position.
ROBOT-FACING	Print out the robot's facing.
HANDLE-INPUT	Attempt to make sense of an input sentence.
RUN-ROBOT	Demonstrate natural language parsing.

Computers can be very difficult for people to use. For the average person, direct use of a computer is impossible. As we have seen in this book, to use a computer effectively we must learn to speak one of its languages. Forth is one such language. However, in order for more people to be able to make use of computers, people with no specialized training, computers will have to learn how to understand human, or natural language. This chapter will present a set of words and techniques that make this possible, at least in part.

The scope of programming a computer to comprehend the entire English language is obviously too large for this chapter, so we'll limit our attempt to a small subset of English. Our small subset will be commands for directing the movement of a robot. We'll be able to understand sentences like this:

```
GO FORWARD 10 FEET.
BACKUP.
TURN LEFT.
MOVE 4 FEET BACKWARDS.
```

The techniques presented here are the same that would be used if we we're going to extend our program to understand a larger part of English. This is probably one of the most interesting chapters in the book to extend.

The problem of programming a computer to understand English can be divided into two parts, syntax and meaning. The method we will use to dissect the syntax of the sentences presented to us is known as "Augmented Transition Networks" or ATNs. They look somewhat like the sentence diagrams one learns in grammar school. ATNs are directed graphs whose arcs can involve other graphs and/or processing. Figure 10-1 is a diagram of the particular ATN we will be using.

As the words in the sentence are examined we move along the nodes and arcs of the ATN. Nodes (the circles) that are marked with a small f are those that processing can validly terminate. They are known as terminal nodes. As the graph is traversed, we can fill in various variables with information, such as what verb was just found. In the parlance of ATN's these variables are known as registers.

When the graph has been traversed successfully we have a syntactically valid sentence. This doesn't necessarily mean we have one that makes sense.

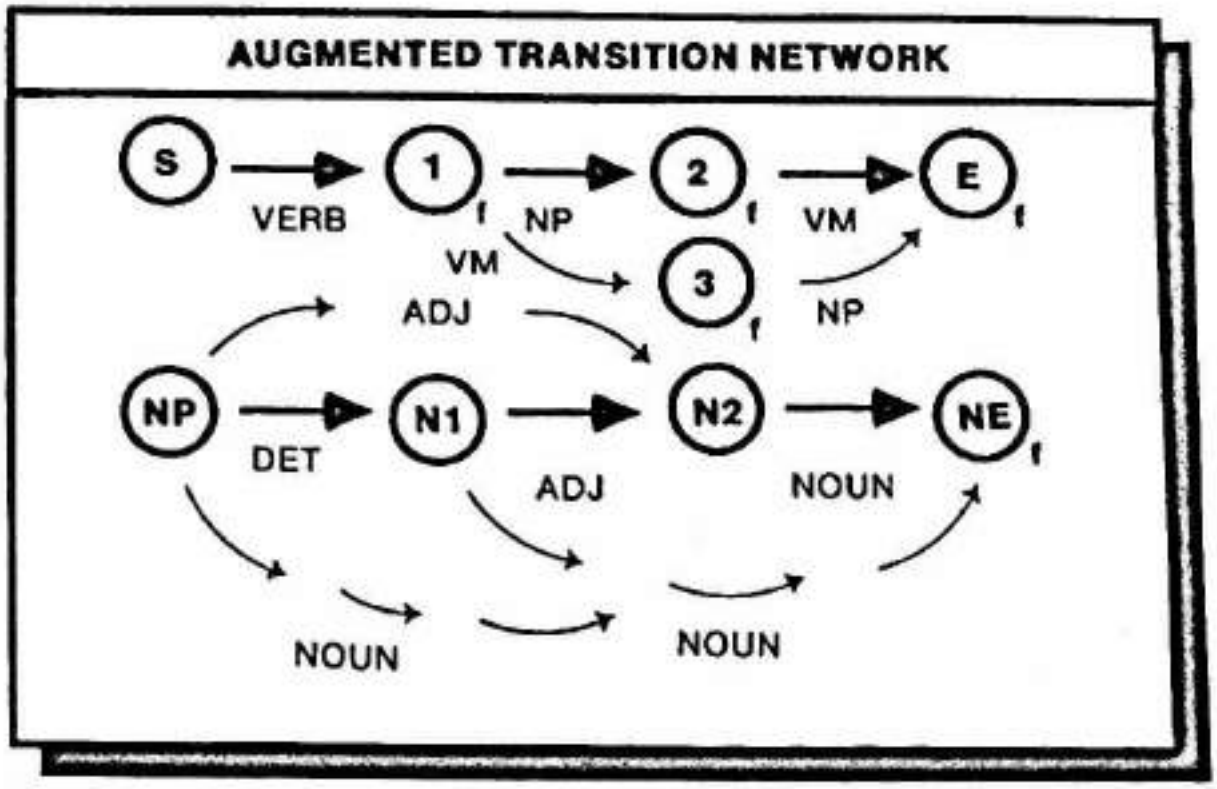


Figure 10-1

Here are some of the sentences that would be considered syntactically valid by our ATN:

DROP THE FEET FORWARD
BACKUP 10 BALLS

Obviously, these won't mean anything to our robot. The second part of our natural language processing will examine the variables or registers filled in by the ATN and try to make sense of them. There is nothing fancy about this second part, it just looks for particular combinations of registers.

We will use a vocabulary table in our processing. Each entry in the table will have a part of speech, a word number, and the word text. We can use synonyms by assigning different words the same number.

Suggested Extensions: This chapter presents a great opportunity for experimentation and extension. The techniques used could be expanded on to give our robot some more intelligence, or could be developed for an entirely different problem domain.

VERB (– N)

A constant for the part-of–speech verb.

Stack on Entry: Empty.

Stack on Exit: (N) The value used to represent a verb.

Example of Use:

P.O.S. VERB =

The word P.O.S. will return the part of speech of a word being examined. This code would leave a true flag on the stack if P.O.S. returned a verb as the current part of speech.

Algorithm: None.

Suggested Extensions: None.

Definition:

1 CCONSTANT VERB

NOUN (– N)

A constant for the part-of-speech noun.

Stack on Entry: Empty.

Stack on Exit: (N) The value used to represent a noun.

Example of Use:

P.O.S. NOUN =

The word P.O.S. will return the part of speech of a word being examined. This code would leave a true flag on the stack if P.O.S. returned a noun as the current part of speech.

Algorithm: None.

Suggested Extensions: None.

Definition:

2 CCONSTANT NOUN

DET (– N)

A constant for the part-of-speech determinant.

Stack on Entry: Empty.

Stack on Exit: (N) The value used to represent a determinant.

Example of Use:

P.O.S. DET =

The word P.O.S. will return the part of speech of a word being examined. This code would leave a true flag on the stack if P.O.S. returned a determinant as the current part of speech.

Algorithm: None.

Suggested Extensions: None.

Definition:

3 CCONSTANT DET

ADJ (– N)

A constant for the part-of-speech adjective.

Stack on Entry: Empty.

Stack on Exit: (N) The value used to represent an adjective.

Example of Use:

P.O.S. ADJ =

The word P.O.S. will return the part of speech of a word being examined. This code would leave a true flag on the stack if P.O.S. returned an adjective as the current part of speech.

Algorithm: None.

Suggested Extensions: None.

Definition:

4 CCONSTANT ADJ

ADVERB (– N)

A constant for the part-of-speech adverb.

Stack on Entry: Empty.

Stack on Exit: (N) The value used to represent an adverb.

Example of Use:

```
P.O.S. ADVERB =
```

The word P.O.S. will return the part of speech of a word being examined. This code would leave a true flag on the stack if P.O.S. returned an adverb as the current part of speech.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
5 CCONSTANT ADVERB
```

CONJ. (– N)

A constant for the part-of-speech conjunction.

Stack on Entry: Empty.

Stack on Exit: (N) The value used to represent a conjunction.

Example of Use:

```
P.O.S. CONJ. =
```

The word P.O.S. will return the part of speech of a word being examined. This code would leave a true flag on the stack if P.O.S. returned a conjunction as the current part of speech.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
6 CCONSTANT CONJ.
```

WORD# (– A)

A variable used to keep track of the number of words in the vocabulary table.

Stack on Entry: Empty.

Stack on Exit: (A) The address of WORD#.

Example of Use:

```
... WORD# @ ...
```

This code would leave the current number of words in the vocabulary table on the stack.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
1 CVARIABLE WORD#
```

VDATA (– A)

An array used to hold vocabulary data.

Stack on Entry: Empty.

Stack on Exit: (A) The address of VDATA.

Example of Use:

```
... 64 VDATA + ...
```

Each entry in the vocabulary table will be 32 bytes long. This piece of code would leave the address of the third entry in the vocabulary table on the stack.

Algorithm: Allocate 802 bytes and then erase them.

Suggested Extensions: None.

Definition:

```
0 VARIABLE VDATA 800 ALLOT
VDATA 802 ERASE
```

VTABLE (– A)

A variable which points to the next free position in VDATA.

Stack on Entry: Empty.

Stack on Exit: (A) The address of VTABLE.

Example of Use:

```
... VTABLE @ VDATA + ...
```

This piece of code would leave the address of the next free entry in VDATA on the stack.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
VDATA VARIABLE VTABLE
```

SYNONYM (–)

Cause the word entered next into the vocabulary table to be a synonym of the word previously entered in the vocabulary table.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
VERB !VOCABULARY SCREAM
```

```
VERB SYNONYM !VOCABULARY YELL
```

This use of SYNONYM would cause SCREAM and YELL to be regarded as synonyms by our natural language processor.

Algorithm: Decrement the value held in WORD#. This will cause both word to have the same "word number" in the vocabulary table.

Suggested Extensions: None.

Definition:

```
: SYNONYM -1 WORD# C+! ;
```

VERB-FOUND (- A)

A byte variable used as a register by the ATN. It will hold the word number of the verb found.

Stack on Entry: Empty.

Stack on Exit: (A) The address of VERB-FOUND.

Example of Use:

```
VERB-FOUND C@ 5 =
```

This code would leave a true flag on the stack if the verb found by the ATN was word number five.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE VERB-FOUND
```

NOUN-FOUND (- A)

A byte variable used as a register by the ATN. It will hold the word number of the noun found.

Stack on Entry: Empty.

Stack on Exit: (A) The address of NOUN–FOUND.

Example of Use:

```
NOUN–FOUND C@ 2 =
```

This code would leave a true flag on the stack if the noun found by the ATN was word number two.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE NOUN-FOUND
```

DET–FOUND (– A)

A byte variable used as a register by the ATN. It will hold the word number of the determinant found.

Stack on Entry: Empty.

Stack on Exit: (A) The address of DET–FOUND.

Example of Use:

```
DET–FOUND C@ 1 =
```

This code would leave a true flag on the stack if the determinant found by the ATN was word number one.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE DET–FOUND
```

ADJ–FOUND (– A)

A byte variable used as a register by the ATN. It will hold the word number of the adjective found.

Stack on Entry: Empty.

Stack on Exit: (A) The address of ADJ–FOUND.

Example of Use:

```
ADJ–FOUND C@ 4 =
```

This code would leave a true flag on the stack if the adjective found by the ATN was word number four.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE ADJ–FOUND
```

ADVERB–FOUND (– A)

A byte variable used as a register by the ATN. It will hold the word number of the adverb found.

Stack on Entry: Empty.

Stack on Exit: (A) The address of ADVERB–FOUND.

Example of Use:

```
ADVERB–FOUND C@ 4 =
```

This code would leave a true flag on the stack if the adjective adverb by the ATN was word number four.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE ADVERB–FOUND
```

!VOCABULARY (N –)

Place a word in the vocabulary table. The word is next in the input stream.

Stack on Entry: (N) The part of speech this word will be.

Stack on Exit: Empty.

Example of Use:

```
NOUN !VOCABULARY FIRETRUCK
```

This would place the noun FIRETRUCK in our vocabulary list.

Algorithm: The entries in the vocabulary table look like this:

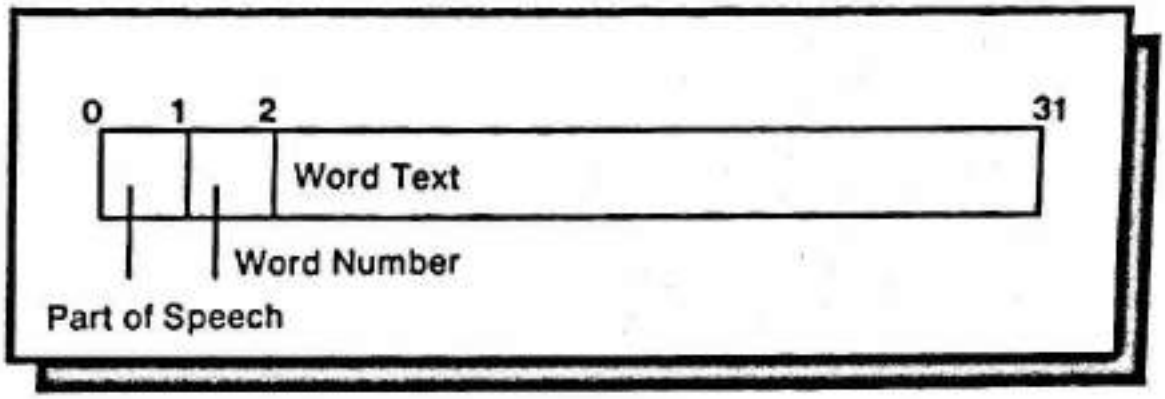


Figure 10-2

VTABLE points to the current free entry in the vocabulary table. The part of speech is stored in the first byte. WORD# is then stored in the second byte. The actual vocabulary word is then taken from the input stream using a blank as the delimiter. The word is then stored in the vocabulary table. WORD# is then incremented and 32 is added to VTABLE to point to the next entry.

Suggested Extensions: None.

Definition:

```
: !VOCABULARY VTABLE @ C!
  WORD# C@ VTABLE @ 1+ C!
```

```
BL WORD VTABLE @ 2+ 30 CMOVE
32 VTABLE +! 1 WORD# C+! ;

( Now store the words we'll use)
VERB !VOCABULARY GO ( 1)
VERB SYNONYM !VOCABULARY MOVE ( 1)
VERB !VOCABULARY TURN ( 2)
VERB !VOCABULARY BACKUP ( 3)
VERB !VOCABULARY PICK ( 4)
VERB !VOCABULARY DROP ( 5)
VERB !VOCABULARY KICK ( 6)
ADVERB !VOCABULARY FORWARD ( 7)
ADVERB !VOCABULARY BACK ( 8)
ADVERB SYNONYM !VOCABULARY BACKWARDS ( 8)
ADVERB !VOCABULARY LEFT ( 9)
ADVERB !VOCABULARY RIGHT ( 10)
DET !VOCABULARY THE ( 11)
NOUN !VOCABULARY FEET ( 12)
NOUN !VOCABULARY BALL ( 13)
ADJ !VOCABULARY RED ( 14)
ADJ !VOCABULARY BLUE ( 15) ->
CONJ. !VOCABULARY AND ( 16)
CONJ. SYNONYM !VOCABULARY THEN ( 16)
```

W#-LIST (- A)

An array used to hold the word numbers of the input sentence.

Stack on Entry: Empty.

Stack on Exit: (A) The address of W#-LIST.

Example of Use:

```
4 W#-LIST + C@
```

This code would leave word number of the fifth word in the input sentence.

Algorithm: Allocate 32 bytes for the array.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE W#-LIST 31 ALLOT
```

WPS-LIST (- A)

An array used to hold the part of speech of each word in the input sentence.

Stack on Entry: Empty.

Stack on Exit: (A) The address of WPS-LIST.

Example of Use:

```
4 W#-LIST + C@
```

This code would leave the part of speech of the fifth word in the input sentence.

Algorithm: Allocate 32 bytes for the array.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE WPS-LIST 31 ALLOT
```

S-V-T (A1 - (A2) F)

Search the vocabulary table for the string at the address on the stack.

Stack on Entry: (A1) – The address of the string to search for.

Stack on Exit: (A2) – The string's entry in the vocabulary table if found.
(F) – A Boolean flag, true if the string was found.

Example of Use:

```
: TEST QUERY >IN 0SET BL WORD S-V-T ;
```

This word would search for an input word in the vocabulary table.

Algorithm: Loop through the entire vocabulary list. Exit when a match is found or the end is reached. A match is found by first checking the lengths of the strings being compared. If they are unequal, skip to the next compare. The next compare uses -TEXT to actually compare the strings. If this compare is true, a match has been found.

Suggested Extensions: None.

Definition:

```
: S-V-T
  VDATA BEGIN
    OVER C@ OVER 2+ C@ = IF
      OVER 1+ OVER 2+ DUP C@ SWAP 1+
      -TEXT 0= IF
        SWAP DROP -1 EXIT
      ENDIF
    ENDIF
    32 + DUP C@ 0=
  UNTIL 2DROP 0 ;
```

POINTER (- A)

A variable used by the ATN to point to the current word being examined. Points to values in W#-LIST and WPS-LIST.

Stack on Entry: Empty.

Stack on Exit: (A) The address of POINTER.

Example of Use:

```
POINTER W#-LIST + C@
```

This code would leave the part of speech of the current word in the input sentence.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE POINTER
```

ADVANCE (–)

Increment POINTER to point to the next word in the input sentence.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Increment POINTER using C+!

Suggested Extensions: None.

Definition:

```
: ADVANCE 1 POINTER C+! ;
```

P.O.S. (– N)

Find the part of speech of the current word for the ATN.

Stack on Entry: Empty.

Stack on Exit: (N) – The part of speech.

Example of Use:

```
P.O.S. VERB =
```

This code would leave a true flag on the stack if the current word the ATN is looking at is a verb.

Algorithm: Access the array WPS–LIST using POINTER.

Suggested Extensions: None.

Definition:

```
: P.O.S.
  POINTER C@ WPS–LIST + C@ ;
```

#-OF-W (- N)

Find the number of the current word the ATN is processing.

Stack on Entry: Empty.

Stack on Exit: (N) – The word number.

Example of Use:

```
#-OF-W 7 =
```

This code would leave a true flag on the stack if the number of the current word was seven.

Algorithm: Access the array W#-LIST using POINTER.

Suggested Extensions: None.

Definition:

```
: #-OF-W
  POINTER C@ W#-LIST + C@ ;
```

?VERB (- F)

Is the current word a verb?

Stack on Entry: Empty.

Stack on Exit: (F) – Boolean flag, true if the current word is a verb.

Example of Use: See words defined below.

Algorithm: If the part of speech of the current word is verb, then store the word number of the current word in the variable VERB-FOUND.

Suggested Extensions: None.

Definition:

```
: ?VERB
  P.O.S. VERB = DUP IF
    #-OF-W VERB-FOUND C!
  ENDIF ;
```

?NOUN (– F)

Is the current word a noun?

Stack on Entry: Empty.

Stack on Exit: (F) – Boolean flag, true if the current word is a noun.

Example of Use: See words defined below.

Algorithm: If the part of speech of the current word is noun, then store the word number of the current word in the variable NOUN–FOUND.

Suggested Extensions: None.

Definition:

```
: ?NOUN
  P.O.S. NOUN = DUP IF
    #–OF–W NOUN–FOUND C!
  ENDIF ;
```

?DET (– F)

Is the current word a determinant?

Stack on Entry: Empty.

Stack on Exit: (F) – Boolean flag, true if the current word is a determinant.

Example of Use: See words defined below.

Algorithm: If the part of speech of the current word is a determinant, then store the word number of the current word in the variable DET–FOUND.

Suggested Extensions: None.

Definition:

```
: ?DET
  P.O.S. DET = DUP IF
    #–OF–W DET–FOUND C!
  ENDIF ;
```

?ADJ (– F)

Is the current word an adjective?

Stack on Entry: Empty.

Stack on Exit: (F) – Boolean flag, true if the current word is an adjective.

Example of Use: See words defined below.

Algorithm: If the part of speech of the current word is an adjective, then store the word number of the current word in the variable ADJ–FOUND.

Suggested Extensions: None.

Definition:

```
: ?ADJ
  P.O.S. ADJ = DUP IF
    #–OF–W ADJ–FOUND C!
  ENDIF ;
```

?ADVERB (– F)

Is the current word an adverb?

Stack on Entry: Empty.

Stack on Exit: (F) – Boolean flag, true if the current word is an adverb.

Example of Use: See words defined below.

Algorithm: If the part of speech of the current word is an adverb, then store the word number of the current word in the variable ADVERB–FOUND.

Suggested Extensions: None.

Definition:

```
: ?ADVERB
  P.O.S. ADVERB = DUP IF
    #–OF–W ADVERB–FOUND C!
  ENDIF ;
```

FAKE-ADJ (– A)

Leave the address of a fake vocabulary table entry for adjective number 255.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of the fake entry.

Example of Use: See words defined below.

Algorithm: This entry will be used for handling numbers in the input sentence.

Suggested Extensions: None.

Definition:

```
4 CVARIABLE FAKE-ADJ 255 C,
```

#ADJ (– A)

A variable which will hold a number found in the input sentence.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of #ADJ.

Example of Use:

```
#ADJ ?
```

This would print the number found in the input sentence. Since it would be extremely inefficient to store all the possible numbers in the vocabulary table, we will use this variable and return an adjective number of 255 to signify that a number was found in the input sentence.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 VARIABLE #ADJ
```

#? (A1 – (A2) F)

Is the string on the stack a number?

Stack on Entry: (A1) – The string to check.

Stack on Exit: (A2) – The address of the vocabulary table entry for a numeric adjective.
(F) – A Boolean flag, true if the string on the stack could be converted to a number.

Example of Use: See words defined below.

Algorithm: Use >BINARY to try to convert the string to a number. If it was successful, store the number in #ADJ and leave the FAKE–ADJ entry on the stack.

Suggested Extensions: None.

Definition:

```
: #?
  0. ROT >BINARY C@ DUP BL = SWAP 0=
  OR IF
    DROP #ADJ ! FAKE-ADJ -1
  ELSE
    2DROP 0
  ENDIF ;
```

0–FOUNDS (–)

Zero all the registers used by the ATN.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Use C0SET on all the byte variables.

Suggested Extensions: None.

Definition:

```
: 0-FOUNDS
  VERB-FOUND C0SET NOUN-FOUND C0SET
  DET-FOUND C0SET ADJ-FOUND C0SET
  ADVERB-FOUND C0SET ;
```

/INIT/ (–)

Init the arrays for INPUT$LOOKUP.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Erase all 32 bytes of WPS-LIST and W#-LIST.

Suggested Extensions: None.

Definition:

```
: /INIT/
   W#-LIST 32 ERASE WPS-LIST 32 ERASE ;
```

INPUT$LOOKUP (– F)

Input a line of text from the keyboard and attempt to look up all the words in the sentence.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if all the words in the sentence could be found in the vocabulary table.

Example of Use: See words defined below.

Algorithm: Use /INIT/ to erase W#-LIST and WPS-LIST. Input a string from the keyboard and loop on the input words until there are no more. For each word, first check to see if it can be converted to a number. If it cannot be converted to a number, try to look it up in the vocabulary table. If it cannot be found in the vocabulary table, exit the loop. If a word was found, store the word number and part of speech in W#-LIST and WPS-LIST.

Suggested Extensions: None.

Definition:

```
: INPUT$LOOKUP
  /INIT/ QUERY >IN 0SET 0 BEGIN
    BL WORD DUP C@
  WHILE
    DUP #? NOT IF
      DUP S-V-T NOT IF
        TYPE2 ." is not in the vocabulary list.
        " DROP 0 EXIT
    ENDIF ENDIF
    SWAP DROP DUP C@ 3 PICK WPS-LIST + C!
    1+ C@ OVER W#-LIST + C! 1+
  REPEAT 2DROP -1 ;
```

GET-GOOD-INPUT (-)

Get a good sentence from the keyboard.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Loop calling INPUT$LOOKUP until it finds a good sentence.

Suggested Extensions: None.

Definition:

```
: GET-GOOD-INPUT BEGIN INPUT$LOOKUP UNTIL ;
```

(N2) (- F)

Node N2 of the ATN.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if N2 found a transition to another node.

Example of Use: See words defined below.

Algorithm: Node N2 needs to find a noun to be successful. If N2 finds a noun, it should advance past the noun in the input list. NE is a node that does nothing and is always successful so for efficiency, is is not actually coded.

Suggested Extensions: None.

Definition:

```
: (N2) ." N2 " ?NOUN DUP IF
     ADVANCE
   ENDIF ;
```

(N1) (– F)

Node N1 of the ATN.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if N1 found a transition to another node.

Example of Use: See words defined below.

Algorithm: Node N1 can move to N2 if an adjective is found or to NE if a noun is found. NE is a node that does nothing and is always successful so for efficiency, is is not actually coded.

Suggested Extensions: None.

Definition:

```
: (N1) ." N1 " ?ADJ IF
     ADVANCE (N2)
   ELSE
     ?NOUN DUP IF
       ADVANCE
     ENDIF
   ENDIF ;
```

(NP) (– F)

Node NP of the ATN.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if NP found a transition to another node.

Example of Use: See words defined below.

Algorithm: NP can advance if it finds a determinant (to N1), an adjective (to N2), or a noun (to NE).

Suggested Extensions: None.

Definition:

```
:(NP) ." NP " ?DET IF
    ADVANCE (N1)
  ELSE
    ?ADJ IF
      ADVANCE (N2)
    ELSE
      ?NOUN DUP IF
        ADVANCE
  ENDIF ENDIF ENDIF ;
```

(3) (– F)

Node 3 of the ATN.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if 3 found a transition to another node.

Example of Use: See words defined below.

Algorithm: 3 can advance if an NP is found. But it is also successful if it cannot advance, because it is a terminal node. Note that when an arc is another graph, as NP is, no advance is done to the next node.

Suggested Extensions: None.

Definition:

```
: (3) ." 3 " (NP) DROP -1 ;
```

(2) (– F)

Node 2 of the ATN.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if 2 found a transition to another node.

Example of Use: See words defined below.

Algorithm: 2 can advance if an adverb is found. But it is also successful if it cannot advance because it is a terminal node.

Suggested Extensions: None.

Definition:

```
: (2) ." 2 " ?ADVERB IF
      ADVANCE
   ENDIF -1 ;
```

(1) (– F)

Node 1 of the ATN.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if 1 found a transition to another node.

Example of Use: See words defined below.

Algorithm: 1 can advance if a noun phrase or an adverb is found. But it is also successful if it cannot advance because it is a terminal node.

Suggested Extensions: None.

Definition:

```
: (1) ." 1 " ?ADVERB IF
      ADVANCE (3)
   ELSE
      (NP) IF
         (2)
      ELSE
      -1
      ENDIF
   ENDIF ;
```

(S) (– F)

Node S of the ATN.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if S found a transition to another node.

Example of Use: See words defined below.

Algorithm: S can advance if a verb is found.

Suggested Extensions: None.

Definition:

```
: (S) ." S " ?VERB IF
      ADVANCE (1)
   ENDIF ;
```

PARSE (– F)

Attempt to apply the ATN to an input sentence.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if the sentence was parsed correctly.

Example of Use: See words defined below.

Algorithm: Zero all the ATN registers, zero POINTER, and start the ATN. If the ATN completes successfully and the sentence is complete, or POSITION points to a conjuction, the parse is successful.

Suggested Extensions: None.

Definition:

```
: PARSE POINTER COSET
   0-FOUNDS (S)
   P.O S. DUP 0= SWAP 6. = OR
   AND ;
```

XPOS (– A)

A variable that will hold our robot's X coordinate position.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of XPOS.

Example of Use:

```
XPOS ?
```

This would print the X coordinate location of our robot.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 VARIABLE XPOS
```

YPOS (– A)

A variable that will hold our robot's Y coordinate position.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of YPOS.

Example of Use:

```
YPOS ?
```

This would print the Y coordinate location of our robot.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 VARIABLE YPOS
```

FACING (– A)

A byte variable that will hold the direction our robot is facing.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of facing.

Example of Use:

```
FACING C?
```

This would print the direction our robot is facing (0–North / 1–East / 2–South / 3–West).

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE FACING
```

MOVE (N –)

Move the robot N feet in its current direction. (This word uses the case words from Chapter 2.)

Stack on Entry: (N) – How many feet to move the robot.

Stack on Exit: Empty.

Example of Use:

```
10 MOVE
```

If the robot was facing north, this would increment its Y coordinate by 10.

Algorithm: Use a case statement to add the value to the proper variable.

Suggested Extensions: None.

Definition:

```
: MOVE FACING C@ CASE
    0 =OF YPOS +! END-OF
    1 =OF XPOS +! END-OF
    2 =OF NEGATE YPOS +! END-OF
    3 =OF NEGATE XPOS +! END-OF
  ENDCASE ;
```

TURN (N –)

Turn the robot one compass direction. (This word uses the case words from Chapter 2.)

Stack on Entry: (N) – Positive one if the robot is turning to its right; negative one if it is turning to its left.

Stack on Exit: Empty.

Example of Use:

```
1 TURN
```

If the robot was facing north before the above code was executed, it would be facing east following its execution.

Algorithm: Add the value on the stack to the variable FACING. Check to see if it has overflowed the range 0–3 and adjust it accordingly.

Suggested Extensions: None.

Definition:

```
: TURN
  FACING C+! FACING C@ CASE
    4 =OF 0 FACING C! END-OF
    255 =OF FACING C0SET END-OF
  ENDCASE ;
```

MOVE-FORWARD? (– F)

Move the robot forward if the proper sentence was entered.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if the robot was moved forward.

Example of Use: See words defined below.

Algorithm: Check the registers from the ATN for the verb/adverb phrase MOVE FORWARD. If they are found, check the adjective and noun to see if they specify how many feet to move forward. If they are not specified, default to one foot. Call MOVE to move the robot.

Suggested Extensions: None.

Definition:

```
: MOVE-FORWARD?
  NOUN-FOUND C@ DUP 0=
  SWAP 12 = OR
  VERB-FOUND C@ 1 = AND
  ADVERB-FOUND C@ 7 = AND IF
    ADJ-FOUND C@ DUP 255 <>
    SWAP 0 <> AND IF
      0 EXIT
    ENDIF
    ADJ-FOUND C@ IF
      #ADJ @
    ELSE
      1
    ENDIF
    MOVE -1 EXIT
  ENDIF 0 ;
```

MOVE-BACK? (– F)

Move the robot backwards if the proper sentence was entered.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if the robot was moved backwards.

Example of Use: See words defined below.

Algorithm: Check the registers from the ATN for the verb/adverb phrase MOVE BACKWARDS. If they are found, check the adjective and noun to see if they specify how many feet to move backwards. If they are not specified, default to one foot. Call MOVE to move the robot.

Suggested Extensions: None.

Definition:

```
: MOVE-BACK?
   NOUN-FOUND C@ DUP 0=
   SWAP 12 = OR
   VERB-FOUND C@ 1 = AND
   ADVERB-FOUND C@ 8 = AND IF
      ADJ-FOUND C@ DUP 255 <>
      SWAP 0 <> AND IF
         0 EXIT
      ENDIF
      ADJ-FOUND C@ IF
         #ADJ @
      ELSE
         1
      ENDIF
      NEGATE MOVE -1 EXIT
   ENDIF 0 ;
```

TURN-LEFT? (- F)

Turn the robot to the left if the proper sentence was entered.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if the robot was turned left.

Example of Use: See words defined below.

Algorithm: Check the registers from the ATN for the verb/adverb phrase TURN LEFT. If they are found, turn the robot to the left with TURN.

Suggested Extensions: None.

Definition:

```
: TURN-LEFT?
   NOUN-FOUND C@ 0=
   2VERB-FOUND C@ 2 = AND
   ADVERB-FOUND C@ 9 = AND IF
      -1 TURN -1
   ELSE
      0
   ENDIF ;
```

TURN-RIGHT? (- F)

Turn the robot to the right if the proper sentence was entered.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if the robot was turned right.

Example of Use: See words defined below.

Algorithm: Check the registers from the ATN for the verb/adverb phrase TURN RIGHT. If they are found, turn the robot to the right with TURN.

Suggested Extensions: None.

Definition:

```
: TURN-RIGHT?
  NOUN-FOUND C@ 0=
  VERB-FOUND C@ 2 = AND
  ADVERB-FOUND C@ 10 = AND IF
    1 TURN -1
  ELSE
    0
  ENDIF ;
```

BACKUP? (- F)

Back up the robot if the proper sentence was entered.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if the robot was moved backwards.

Example of Use: See words defined below.

Algorithm: Check the registers from the ATN for the verb BACKUP. If it is found, check the adjective and noun to see if they specify how many feet to move back. If they are not specified, default to one foot. Call MOVE to move the robot.

Suggested Extensions: None.

Definition:

```
: BACKUP?
   NOUN-FOUND C@ DUP 0=
   SWAP 12 = OR
   VERB-FOUND C@ 3 = AND IF
   ADJ-FOUND C@ DUP 255 <>
   SWAP 0 <> AND IF
      0 EXIT
   ENDIF
   ADJ-FOUND C@ IF
      #ADJ @
   ELSE
      1
   ENDIF
   NEGATE MOVE -1 EXIT
ENDIF 0 ;
```

ROBOT-POS (-)

Print the position of the robot.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
ROBOT-POS
```

Would print:

```
Robot Position:
X => 0
Y => 0
```

If the robot's X and Y position was zero.

Algorithm: Print out the values of XPOS and YPOS.

Suggested Extensions: None.

Definition:

```
: ROBOT-POS
   CR ." Robot Position: " CR
   ." X => " XPOS ? CR
   ." Y => " YPOS ? ;
```

ROBOT-FACING (-)

Print the facing direction of the robot. (This word uses the case words from Chapter 2.)

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

ROBOT-FACING
Would print:
"Facing North."

If the robot was facing north.

Algorithm: Branch on the value held in facing.

Suggested Extensions: None.

Definition:

```
: ROBOT-FACING CR." Facing"
  FACING C@ CASE
    0 =OF ." North." END-OF
    1 =OF ." East." END-OF
    2 =OF ." South." END-OF
    3 =OF ." West." END-OF
  ENDCASE;
```

HANDLE-INPUT (-)

Attempt to make sense of an input sentence.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Call each of the words defined previously that attempt to process input. If any of them complete successfully, the word will exit. If none are able to deal with the input, print out an error message.

Suggested Extensions: To extend how much our robot understands, add more processing phrases to this word.

Definition:

```
: HANDLE-INPUT
  MOVE-FORWARD? IF EXIT ENDIF
  MOVE-BACK? IF EXIT ENDIF
  TURN-LEFT? IF EXIT ENDIF
  TURN-RIGHT? IF EXIT ENDIF
  BACKUP? IF EXIT ENDIF
  CR ." Sorry, I don't understand you."
  CR ;
```

RUN-ROBOT (-)

Demonstrate natural language processing.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: None.

Algorithm: Print out the position and facing direction of the robot. If the input pointer does not point to a conjunction, get more input. Attempt to parse the input. If the parse is unsuccessful print an error message; otherwise, allow our input handler to try to make sense of it.

Suggested Extensions: None.

Definition:

```
: RUN-ROBOT 3 0 DO
    ROBOT-POS ROBOT-FACING
    P.O.S. CONJ. = IF
      ADVANCE
    ELSE
      CR GET-GOOD-INPUT
    ENDIF
    PARSE IF
      HANDLE-INPUT
    ELSE
      CR ." That wasn't very well said."
    ENDIF
  LOOP ;
```

11

Data Structures

Words Defined in This Chapter:

1ARRAY	Define a one-dimensional cell array.
1CARRAY	Define a one-dimensional byte array.
1ARRAY–RNG	Define a one-dimensional cell array, with specified lower and upper bounds.
1CARRAY–RNG	Define a one-dimensional byte array, with specified lower and upper bounds.
OBM	Print the array out-of-bounds error message.
1ARRAY–BC	Define a one-dimensional cell array with bounds checking.
1CARRAY–BC	Define a one-dimensional byte array with bounds checking.
1ARRAY–RNG–BC	Define a one-dimensional cell array, with specified lower and upper bounds. Include bounds checking.
1CARRAY–RNG–BC	Define a one-dimensional byte array, with specified lower and upper bounds. Include bounds checking.
2ARRAY	Define a two-dimensional cell array.
2CARRAY	Define a two-dimensional byte array.

2ARRAY-RNG	Define a two-dimensional cell array, with specified lower and upper bounds.
2CARRAY-RNG	Define a two-dimensional byte array, with specified lower and upper bounds.
2ARRAY-BC	Define a two-dimensional cell array with bounds checking.
2CARRAY-BC	Define a two-dimensional byte array with bounds checking.
2ARRAY-RNG-BC	Define a two-dimensional cell array, with specified lower and upper bounds. Include bounds checking.
2CARRAY-RNG-BC	Define a two-dimensional byte array, with specified lower and upper bounds. Include bounds checking.
1?ARRAY	Define a one-dimensional array with a specified element size.
1?ARRAY-RNG	Define a one-dimensional array with a specified element size. The lower and upper bounds will also be specified.
1?ARRAY-BC	Define a one-dimensional array with a specified element size. Include bounds checking.
1?ARRAY-RNG-BC	Define a one-dimensional array with a specified element size. The lower and upper bounds will also be specified. Include bounds checking.
ARRAY	Define an array of any dimensionality with a specified element size.
DISK-2-MEM	Move data from disk to memory.
MEM-2-DISK	Move data from memory to disk.
DISK-ARRAY	Define a disk array of any dimensionality with a specified element size.
RECORD	Start a record definition.
FIELD	Define a field in a record.
VARIANT	Start a variant in a record definition.
START-VARIANTS	Start a set of variants in a record definition.
END-VARIANT	End the definition of a variant portion of a record.
END-ALL-VARIANTS	Complete the definition of a set of variants in a record.
INSERT	Insert a record as a sub-record in a record being defined.
END-RECORD	Complete the definition of a record.

1AFIELD	Define an array field in a record.
INSTANCE	Create an instance of a record.
M–INSTANCE	Create an array of instances of a record.
TO	Cause a "to" type variable to perform a store.
TVARIABLE	Define a cell sized "to" type variable.
TCVARIABLE	Define a byte sized "to" type variable.
FIFO	Create a queue.
(#FI)	Fetch the start and end pointers of a queue.
?FIFO	Determine if data are stored in a queue.
CNO	Abort with an error message if a queue is empty.
@FIFO	Remove data from a queue.
!FIFO	Store data in a queue.
FIFO–RESET	Clear a queue.

This chapter presents a set of words for managing data, both in memory and on disk. Arrays of every type and size are presented, as is a record structure similar to Pascal's. We also throw in a new kind of variable, "to" type variables, and some words for storing and manipulating queues.

RECORD STRUCTURES

The record structure presented in this chapter will enable us to define records with multiple fields, variant fields, and subrecords. Here is an example of a definition of a simple record:

```
RECORD PERSON
   16 FIELD NAME
      8 FIELD SS-#
END-RECORD
```

PERSON is now the name of a record with two fields. NAME is a field in PERSON that is 16 bytes in length. SS-# is an 8-byte field. RECORD defines a template. To create an actual instance of a record, we use the word INSTANCE. It works like this:

```
INSTANCE JAMES OF PERSON
```

An instance of the record type PERSON called JAMES is created. Now the fields of the record JAMES can be accessed like this:

```
JAMES NAME

JAMES SS-#
```

If we want to create a one-dimensional array of instances of a record, we use the word M-INSTANCE like this:

```
10 M-INSTANCE COMPANY OF PERSON
```

This creates a set of 10 instances of the record PERSON, numbered zero to nine. Here is how we would print out the name of the seventh person:

```
7 COMPANY NAME 16 TYPE
```

Records can have variants; these are identical portions of the record mapped in different ways. This saves space by not wasting any bytes. Our record could be expanded like this:

```
RECORD PERSON
   16 FIELD NAME
   8 FIELD SS-#
   1 FIELD SEX
   START-VARIANTS
      VARIANT
         16 FIELD WIFE-NAME
      END-VARIANT
      VARIANT
         16 FIELD MAIDEN-NAME
         2 FIELD #CHILDREN
         1 FIELD PREG?
      END-VARIANT
   END-ALL-VARIANTS
END-RECORD
```

We save 16 bytes by using variant records in the above example. Any number of variants can exist in our records. There can also be multiple sets of variants in a single record.

At times we may wish to include certain records as part of another record. We can accomplish this with the following code:

```
RECORD CAR
   2 FIELD MAKE
   2 FIELD MODEL
   INSERT OWNER OF PERSON
   6 FIELD PLATE
END-RECORD
```

If ZAQ is an instance of the record CAR, this would print the name of its owner:

```
ZAQ OWNER NAME 16 TYPE
```

Records can be nested to any depth using this technique.

DISK ADDRESS

Words to move data between disk and memory are presented in this chapter. They make use of a double-length number called a *disk address*. This representation just assigns each byte in mass storage a unique number. The bytes on block zero are numbered 0-1023, the bytes on block one 1024-2047, etc. DISK-2-MEM and MEM-2-DISK enable us to deal with disk storage without being concerned with block boundaries.

"To" Type Variables

"To" type variables are an attempt to get rid of some of the strange syntax that store and fetch introduce into Forth. They remove some flexibility in trade for some clarity. The replacement is shown in the chart that follows. "To" type variables don't allow operations like +!, and OSET, so they are somewhat less powerful than normal variables.

Suggested Extensions: If you like "to" type variables, most of the data structure words in this chapter could be modified to use the "to" technique. Only one-dimensional arrays of records are presented. The instance words could be expanded just like the array words.

Operation	Normal Variables	"To" Type Variables
Define	0 VARIABLE STUFF	0 TVARIABLE STUFF
Store	99 STUFF !	99 TO STUFF
Fetch	STUFF @	STUFF

1ARRAY (N –) (N – A)

Define and allocate space for a one-dimensional cell array.

Stack on Entry: (Compile Time) (N) – Number of entries in the array.
(Run Time) (N) – Position in the array to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of the array element.

Example of Use:

```
10 1ARRAY HI-TMT
```

This would define a 10-cell array, called HI-TMT. This would print the value of the second element of HI-TMT:

```
1 HI-TMT ?
```

Note that the array elements are numbered 0-9, so 1 is the second element of the array.

Algorithm: At compile time, allocate space for the array, two bytes for each element. At run time, multiply the element number by two and add it to the base address.

Suggested Extensions: None.

Definition:

```
: 1ARRAY <BUILDS
    HERE SWAP 2* DUP ALLOT ERASE
  DOES>
    SWAP 2* + ;
```

1CARRAY (N -) (N - A)

Define and allocate space for a one-dimensional byte array.

Stack on Entry: (Compile Time) (N) - Number of entries in the array.
(Run Time) (N) - Position in the array to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) - Address of the array element.

Example of Use:

```
50 1CARRAY HI-MOM
```

This would define a 50 cell array, called HI-MOM. This word would sum all the elements of HI-MOM:

```
: SUM-HI-MOM 0 HI-MOM C@ 50 1 DO
    I HI-MOM C@ +
  LOOP ;
```

Note that the array elements are numbered 0-49.

Algorithm: At compile time, allocate space for the array, one byte for each element. At run time, add the element number to the base address.

Suggested Extensions: None.

Definition:

```
: 1CARRAY <BUILDS
  HERE SWAP DUP ALLOT ERASE
DOES>
  + ;
```

1ARRAY-RNG (N1 N2 -) (N - A)

Define and allocate space for a one-dimensional cell array, with specified upper and lower bounds.

Stack on Entry: (Compile Time) (N1) - Lower bound of the array.
(N2) - Upper bound of the array.
(Run Time) (N) - Position in the array to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) - Address of the array element.

Example of Use:

```
1940 1986 1ARRAY-RNG ROYS/YEAR
```

This would define a 47-cell array, called ROYS/YEAR. The lower bound of the array is 1940, the upper bound is 1985. This would access the twenty-second element of the array:

```
1962 ROYS/YEAR
```

Algorithm: At compile time, store the lower bound. Allocate space for the array, two bytes for each element. At run time, subtract the lower bound from the passed element. Multiply by two and add it to the base address.

Suggested Extensions: None.

Definition:

```
: 1ARRAY-RNG <BUILDS
    SWAP DUP , - 1+
    HERE SWAP 2* DUP ALLOT ERASE
  DOES>
    DUP @ ROT SWAP - 2* 2+ + ;
```

1CARRAY-RNG (N1 N2 -) (N - A)

Define and allocate space for a one-dimensional byte array, with specified upper and lower bounds.

Stack on Entry: (Compile Time) (N1) – Lower bound of the array.
(N2) – Upper bound of the array.
(Run Time) (N) – Position in the array to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of the array element.

Example of Use:

```
1964 1986 1CARRAY-RNG SJT/TICKETS/YEAR
```

This would define a 23-cell array, called SJT/TICKETS/YEAR. The lower bound of the array is 1964; the upper bound is 1986. This would print out the twenty-second element of the array.

```
1985 SJT/TICKETS/YEAR C?
```

Algorithm: At compile time, store the lower bound. Allocate space for the array, one byte for each element. At run time, subtract the lower bound from the passed element and add it to the base address.

Suggested Extensions: None.

Definition:

```
: 1CARRAY-RNG <BUILDS
    SWAP DUP , - 1+
    HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP @ ROT SWAP - + 2+ ;
```

OBM (N -)

Print the array out-of-bounds error message.

Stack on Entry: (N) – The offending array element.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Print the message, print the offending element, and abort out.

Suggested Extensions: None.

Definition:

```
: OBM ." Array Out of Bounds ". ABORT;
```

1ARRAY-BC (N -) (N - A)

Define and allocate space for a one-dimensional cell array with bounds checking.

Stack on Entry: (Compile Time) (N) – Number of entries in the array.
(Run Time) (N) – Position in the array to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of the array element.

Example of Use:

```
5 1ARRAY-BC SU+KEV
```

This would define a 5-cell array, called SU+KEV. If this array is passed an element out of range, it will abort out, like this:

```
7 SU+KEV ?

Array Out of Bounds 7
```

Note that the array elements are numbered 0–4.

Algorithm: At compile time, allocate space for the array, two bytes for each element. Store the upper bound of the array. At run time, check the passed array element to see if it falls in the proper range. If the array element is in range, multiply the element number by two and add it to the base address plus two (to skip the upper bound).

Suggested Extensions: None.

Definition:

```
: 1ARRAY-BC <BUILDS
```

```
    DUP ,
    HERE SWAP 2* DUP ALLOT ERASE
  DOES>
    DUP 2+ >R @ OVER > OVER -1 > AND IF
      2* R> +
    ELSE
      OBM
    ENDIF ;
```

1CARRAY-BC (N -) (N - A)

Define and allocate space for a one-dimensional byte array with bounds checking.

Stack on Entry: (Compile Time) (N) – Number of entries in the array.
(Run Time) (N) – Position in the array to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of the array element.

Example of Use:

```
100 1CARRAY-BC DAVE []
```

This would define a 100-byte array, called DAVE []. If this array is passed an element out of range, it will abort out, like this:

```
307 DAVE [] C?

Array Out of Bounds 307
```

Note that the array elements are numbered 0–99.

Algorithm: At compile time, allocate space for the array, one byte for each element. Store the upper bound of the array. At run time, check the passed array element to see if it falls in the proper range. If the array element is in range, add it to the base address plus two (to skip the upper bound).

Suggested Extensions: None.

Definition:

```
: 1CARRAY-BC <BUILDS
    DUP ,
    HERE SWAP DUP ALLOT ERASE
```

```
DOES>
  DUP 2+ >R @ OVER > OVER -1 > AND IF
    R> +
  ELSE
    OBM
  ENDIF ;
```

1ARRAY-RNG-BC (N1 N2 -) (N - A)

Define and allocate space for a one-dimensional cell array, with specified upper and lower bounds. Also include bounds checking.

Stack on Entry: (Compile Time) (N1) – Lower bound of the array.
(N2) – Upper bound of the array.
(Run Time) (N) – Position in the array to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of the array element.

Example of Use:

```
-10 10 1ARRAY-RNG-BC >DAN []
```

This would define a 21-cell array, called >DAN []. The lower bound of the array is –10, the upper bound is 10. This would print out the second element of the array:

```
-9 >DAN []?
```

Since bounds checking is also included in this array, the following code would abort with an array bounds error:

```
-15 >DAN []?

Array out of bounds -15
```

Algorithm: At compile time, allocate space for the array, two bytes for each element. Store the lower and upper bounds. At run time, see if the passed element falls in the range specified by the lower and upper bounds. If the element is within the proper range, subtract the lower bound from the passed element. Multiply by two and add it to the base address plus four (to skip the lower and upper bounds).

Suggested Extensions: None.

Definition:

```
: 1CARRAY-RNG-BC <BUILDS
    SWAP DUP , OVER , - 1+
    HERE SWAP 2* DUP ALLOT ERASE
  DOES>
    DUP >R OVER >R 2DUP @ >= >R
    2+ @ <= R> AND IF
      R> R> DUP @ ROT SWAP - 2* 4 + +
    ELSE
      R> OBM
    ENDIF ;
```

1CARRAY-RNG-BC (N1 N2 -) (N - A)

Define and allocate space for a one-dimensional byte array, with specified upper and lower bounds. Also include bounds checking.

Stack on Entry: (Compile Time) (N1) – Lower bound of the array.
(N2) – Upper bound of the array.
(Run Time) (N) – Position in the array to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of the array element.

Example of Use:

```
1983 1986 1CARRAY-RNG-BC LISA-PER-YEAR()
```

This would define a 4-byte array, called LISA-PER-YEAR(). The lower bound of the array is 1983, the upper bound is 1986. This would print out the second element of the array:

```
1984 LISA-PER-YEAR() C?
```

Since bounds checking is also included in this array, the following code would abort with an array bounds error:

```
1982 LISA-PER-YEAR() C?

Array out of bounds 1982
```

Algorithm: At compile time, allocate space for the array, one byte for each element. Store the lower and upper bounds. At run time, see if the passed element falls in the range specified by the lower and upper bounds. If the element

is within the proper range, subtract the lower bound from the passed element and add it to the base address plus four (to skip the lower and upper bounds).

Suggested Extensions: None.

Definition:

```
: 1CARRAY-RNG-BC <BUILDS
    SWAP DUP , OVER , - 1+
    HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP >R OVER >R 2DUP @ >= >R
    2+ @ <= R> AND IF
      R> R> DUP @ ROT SWAP - + 4 +
    ELSE
      R> OBM
    ENDIF ;
```

2ARRAY (N1 N2 -) (N1 N2 - A)

Define and allocate space for a two-dimensional cell array.

Stack on Entry: (Compile Time) (N1) – Number of rows in the array.
(N2) – Number of columns in the array.
(Run Time) (N1) – Row to be accessed.
(N2) – Column to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of the array element.

Example of Use:

```
5 10 2ARRAY D+ANN
```

This would define a 5-by-10 cell array, called D+ANN. This would print the value of the second row and third column of D+ANN:

```
1 2 D+ANN ?
```

Note that the rows are numbered 0–4, and the columns are numbered 0–9.

Algorithm: At compile time, allocate space for the array, two bytes for each element. The number of array elements if found by multiplying the number of

rows by the number of columns. Store the size of the rows. At run time, multiply the row number by the row size and add the column number times two. Add this to the base address plus two (to skip the row size).

Suggested Extensions: None.

Definition:

```
: 2ARRAY <BUILDS
    DUP 2* , *
    HERE SWAP 2* DUP ALLOT ERASE
  DOES>
    DUP @ 4 ROLL * + SWAP 2* + 2+ ;
```

2CARRAY (N1 N2 –) (N1 N2 – A)

Define and allocate space for a two-dimensional byte array.

Stack on Entry: (Compile Time) (N1) – Number of rows in the array.
(N2) – Number of columns in the array.
(Run Time) (N1) – Row to be accessed.
(N2) – Column to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of the array element.

Example of Use:

```
20 3 2CARRAY I+Y
```

This would define a 20-by-3 byte array, called I+Y. This would print the value of the seventh row and third column of I+Y:

```
6 2 I+Y C?
```

Note that the rows are numbered 0–19, and the columns are numbered 0–2.

Algorithm: At compile time, allocate space for the array, one byte for each element. The number of array elements is found by multiplying the number of rows by the number of columns. Store ths size of the rows. At run time, multiply the row number by the row size and add the column number. Add this to the base address plus two (to skip the row size).

Suggested Extensions: None.

Definition:

```
: 2CARRAY <BUILDS
    DUP , *
    HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP @ 4 ROLL * + + 2+ ;
```

2ARRAY-RNG (N1 N2 N3 N4 -) (N1 N2 - A)

Define and allocate space for a two-dimensional cell array, with specified upper and lower bounds.

Stack on Entry: (Compile Time) (N1) – Lower bound of rows in the array.
(N2) – Upper bound of rows in the array.
(N3) – Lower bound of columns in the array.
(N4) – Upper bound of columns in the array.
(Run Time) (N1) – Row to be accessed.
(N2) – Column to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of the array element.

Example of Use:

```
1929 1939 1 365 STOCK-PRICES
```

This would define an 11-by-386 cell array, called STOCK-PRICES, which would print the value of the second row and seventy-third column of STOCK-PRICES:

```
1930 73 STOCK-PRICES ?
```

Note that the rows are numbered 1929–1939, and the columns are numbered 1–365.

Algorithm: At compile time, allocate space for the array; two bytes for each element. The number of array elements is found by multiplying the number of rows by the number of columns. Store the size of the rows. Store the lower bounds of the rows and columns. At run time, find the absolute row and column number by subtracting the lower bounds from the passed elements. Multiply the row number by the row size and add the column number times two. Add this to the base address plus six (to skip the lower bounds and the row size).

Suggested Extensions: None.

Definition:

```
: 2ARRAY-RNG <BUILDS
    OVER , 4 PICK , SWAP - 1+ >R
    SWAP - 1+ R> DUP 2* , *
    HERE SWAP 2* DUP ALLOT ERASE
  DOES>
    DUP >R DUP LROT @ - LROT 2+ @ -
    SWAP R> DUP 4 + @ 4 ROLL * +
    SWAP 2* + 6 + ;
```

2CARRAY-RNG (N1 N2 N3 N4 -) (N1 N2 - A)

Define and allocate space for a two-dimensional byte array, with specified upper and lower bounds.

Stack on Entry: (Compile Time) (N1) – Lower bound of rows in the array.
(N2) – Upper bound of rows in the array.
(N3) – Lower bound of columns in the array.
(N4) – Upper bound of columns in the array.
(Run Time) (N1) – Row to be accessed.
(N2) – Column to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of the array element.

Example of Use:

```
1 8 1 8 2CARRAY-RNG CHESS-BOARD
```

This would define an 8-by-8 byte array, called CHESS-BOARD. This would print the value of the third row and third column of CHESS-BOARD:

```
3 3 CHESS-BOARD C?
```

Note that the rows are numbered 1–8, and the columns are numbered 1–8.

Algorithm: At compile time, allocate space for the array, one byte for each element. The number of array elements is found by multiplying the number of rows by the number of columns. Store ths size of the rows. Store the lower

bounds of the rows and columns. At run time, find the absolute row and column number by subtracting the lower bounds from the passed elements. Multiply the row number by the row size and add the column number. Add this to the base address plus six (to skip the lower bounds and the row size).

Suggested Extensions: None.

Definition:

```
: 2CARRAY-RNG <BUILDS
    OVER , 4 PICK , SWAP - 1+ >R
    SWAP - 1+ R> DUP , *
    HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP >R DUP LROT @ - LROT 2+ @ -
    SWAP R> DUP 4 + @ 4 ROLL * +
    + 6 + ;
```

2ARRAY-BC (N1 N2 -) (N1 N2 - A)

Define and allocate space for a two-dimensional cell array, with bounds checking.

Stack on Entry: (Compile Time) (N1) - Number of rows in the array.
(N2) - Number of columns in the array.
(Run Time) (N1) - Row to be accessed.
(N2) - Column to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) - Address of the array element.

Example of Use:

```
100 2 2ARRAY-BC GALLONS
```

This would define a 100-by-2 cell array, called GALLONS, which would print the value of the eighty-second row and first column of GALLONS:

```
81 0 GALLONS ?
```

Because this array has bounds checking, passing it a value out of range will result in an abort, like this:

```
8 22 GALLONS ?
```

Array out of bounds 22

Algorithm: At compile time, allocate space for the array, two bytes for each element. The number of array elements is found by multiplying the number of rows by the number of columns. Store the size of the rows. Store the upper bounds of the row and columns. At run time, determine if the passed elements lie within the valid range specified by the row and column sizes. If they do, multiply the row number by the row size and add the column number times two. Add this to the base address plus six (to skip the row and column bounds and the row size).

Suggested Extensions: None.

Definition:

```
: 2ARRAY-BC <BUILDS
    2DUP , , DUP 2* , *
    HERE SWAP 2* DUP ALLOT ERASE
  DOES>
    DUP >R @ OVER > OVER -1 > AND NOT IF
      OBM
    ELSE
      SWAP R> DUP >R 2+ @ OVER >
      OVER -1 > AND NOT IF
        OBM
      ELSE
        SWAP R> DUP 4 + @ 4 ROLL * +
        SWAP 2* + 6 +
    ENDIF ENDIF ;
```

2CARRAY-BC (N1 N2 -) (N1 N2 - A)

Define and allocate space for a two-dimensional byte array, with bounds checking.

Stack on Entry: (Compile Time) (N1) – Number of rows in the array.
(N2) – Number of columns in the array.
(Run Time) (N1) – Row to be accessed.
(N2) – Column to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of the array element.

Example of Use:

```
3 3 2CARRAY-BC TTT-BOARD
```

This would define a 3-by-3 byte array, called TTT-BOARD. This would print the value of the first row and first column of TTT-BOARD:

```
0 0 TTT-BOARD C?
```

Because this array has bounds checking, passing it a value out of range will result in an abort, like this:

```
0 3 TTT-BOARD C?
Array out of bounds 3
```

Algorithm: At compile time, allocate space for the array, one byte for each element. The number of array elements is found by multiplying the number of rows by the number of columns. Store the size of the rows. Store the upper bounds of the row and columns. At run time, determine if the passed elements lie within the valid range specified by the row and column sizes. If they do, multiply the row number by the row size and add the column number. Add this to the base address plus six (to skip the row and column bounds and the row size).

Suggested Extensions: None.

Definition:

```
: 2CARRAY-BC <BUILDS
    2DUP , , DUP , *
    HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP >R @ OVER > OVER -1 > AND NOT IF
      OBM
    ELSE
      SWAP R> DUP >R 2+ @ OVER >
       OVER -1 > AND NOT IF
         OBM
       ELSE
         SWAP R> DUP 4 + @ 4 ROLL * +
         + 6 +
       ENDIF
     ENDIF ;
```

2ARRAY-RNG-BC (N1 N2 N3 N4 -) (N1 N2 - A)

Define and allocate space for a two-dimensional cell array, with specified upper and lower bounds. Also include bounds checking.

Stack on Entry: (Compile Time) (N1) – Lower bound of rows in the array.
(N2) – Upper bound of rows in the array.
(N3) – Lower bound of columns in the array.
(N4) – Upper bound of columns in the array.
(Run Time) (N1) – Row to be accessed.
(N2) – Column to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of the array element.

Example of Use:

```
1 9 0 2 2ARRAY-RNG-BC RUNS
```

This would define a 9-by-3 cell array, called RUNS. This would print the value of the ninth row and third column of RUNS:

```
9 2 RUNS ?
```

Because this array has bounds checking, it will abort when it is passed an element not within its defined bounds. For example:

```
5 -1 RUNS ?

Array out of bounds -1
```

Algorithm: At compile time, allocate space for the array, two bytes for each element. The number of array elements is found by multiplying the number of rows by the number of columns. Store the size of the rows. Store the lower and upper bounds of the rows and columns. At run time, determine if the passed row and column numbers lie within the range defined at compile time. If they do, find the absolute row and column number by subtracting the lower bounds from the passed elements. Multiply the row number by the row size and add the column number times two. Add this to the base address plus ten (to skip the lower and upper bounds and the row size).

Suggested Extensions: None.

Definition:

```
: 2ARRAY-RNG-BC <BUILDS
    5 1 DO I PICK , LOOP SWAP
    - 1+ >R SWAP - 1+ R> DUP 2* , *
    HERE SWAP 2* DUP ALLOT ERASE
  DOES>
    DUP >R OVER SWAP 2DUP 2+ @ > =
```

```
>R @ <= R> AND NOT IF
  OBM
ELSE
  SWAP R> DUP >R OVER SWAP 2DUP 6 +
  @ >= >R 4 + @ <= R> AND NOT IF
    OBM
  ELSE
    SWAP R> DUP >R 2+ @ -
    SWAP R> DUP >R 6 + @ -
    SWAP R> DUP 8 + @ 4 ROLL * +
    SWAP 2* + 10 +
  ENDIF
ENDIF ;
```

2CARRAY-RNG-BC (N1 N2 N3 N4 -)

(N1 N2 - A)

Define and allocate space for a two-dimensional byte array, with specified upper and lower bounds. Also include bounds checking.

Stack on Entry: (Compile Time) (N1) - Lower bound of rows in the array.
(N2) - Upper bound of rows in the array.
(N3) - Lower bound of columns in the array.
(N4) - Upper bound of columns in the array.
(Run Time) (N1) - Row to be accessed.
(N2) - Column to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) - Address of the array element.

Example of Use:

```
75 80 -10 0 2CARRAY-RNG-BC #BROKEN
```

This would define a 6-by-11 byte array, called #BROKEN, which would print the value of the sixth row and eleventh column of #BROKEN:

```
80 0 #BROKEN C?
```

Because this array has bounds checking, it will abort when it is passed an element not within its defined bounds. For example:

```
80 1 #BROKEN C?

Array out of bounds 1
```

Algorithm: At compile time, allocate space for the array, one byte for each element. The number of array elements is found by multiplying the number of rows by the number of columns. Store the size of the rows. Store the lower and upper bounds of the rows and columns. At run time, determine if the passed row and column numbers lie within the range defined at compile time. If they do, find the absolute row and column number by subtracting the lower bounds from the passed elements. Multiply the row number by the row size and add the column number. Add this to the base address plus ten (to skip the lower and upper bounds and the row size).

Suggested Extensions: None.

Definition:

```
: 2CARRAY-RNG-BC <BUILDS
    5 1 DO I PICK , LOOP SWAP
    - 1+ >R SWAP - 1+ R> DUP , *
    HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP >R OVER SWAP 2DUP 2+ @ >=
    >R @ <= R> AND NOT IF
      OBM
    ELSE
      SWAP R> DUP >R OVER SWAP 2DUP 6 +
      @ >= >R 4 + @ <= R> AND NOT IF
        OBM
      ELSE
        SWAP R> DUP >R 2+ @ -
        SWAP R> DUP >R 6 + @ -
        SWAP R> DUP 8 + @ 4 ROLL * +
        + 10 +
      ENDIF
    ENDIF ;
```

1?ARRAY (N1 N2 -) (N - A)

Define and allocate space for a one-dimensional array with a specified element size.

Stack on Entry: (Compile Time) (N1) - Number of entries in the array.
(N2) - Size, in bytes, of each array entry.
(Run Time) (N) - Position in the array to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) - Address of the array element.

Example of Use:

```
20 4 1?ARRAY BIG#S
```

This would define a 20-element array, each element being four bytes in length, called BIG#S. The following would dump the four bytes that make up the second element of BIG#S:

```
1 BIG#S 4 DUMP
```

Note that the array elements are numbered 0–19.

Algorithm: At compile time, allocate space for the array. The number of bytes needed is the number of elements times the size of each element. Store the size of each element. At run time, find the address of the element by multiplying the passed element by the element size. Add this to the base address plus two (to skip the element size).

Suggested Extensions: None.

Definition:

```
: 1?ARRAY <BUILDS
    DUP , *
    HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP @ ROT * + 2+ ;
```

1?ARRAY-BC (N1 N2 –) (N – A)

Define and allocate space for a one-dimensional array with a specified element size. Include array bounds checking.

Stack on Entry: (Compile Time) (N1) – Number of entries in the array.
(N2) – Size, in bytes, of each array entry.
(Run Time) (N) – Position in the array to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of the array element.

Example of Use:

```
5 10 1?ARRAY TREALS []
```

This would define a 5-element array, each element being ten bytes in length, called TREALS []. The following would dump the ten bytes that make up the fifth element of TREALS []:

```
4 TREALS [] 10 DUMP
```

Array bounds checking will catch any attempts to access array elements not in the specified range. For example:

```
10 TREALS [] 10 DUMP
```

```
Array out of bounds 10
```

Algorithm: At compile time, allocate space for the array. The number of bytes needed is the number of elements times the size of each element. Store the size of each element. Store the upper bound of the array. At run time, see if the passed element number is within the range specified at compile time. If it is, find the address of the element by multiplying the passed element by the element size. Add this to the base address plus four (to skip the element size and the upper bound).

Suggested Extensions: None.

Definition:

```
: 1?ARRAY-BC <BUILDS
    DUP , OVER , *
    HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP >R 2+ @ OVER > OVER -1 >
    AND NOT IF
      OBM
    ELSE
      R> DUP @ ROT * + 4 +
    ENDIF ;
```

1?ARRAY-RNG (N1 N2 N3 -) (N - A)

Define and allocate space for a one-dimensional array with a specified element size. The lower and upper bounds of the array will also be specified.

Stack on Entry: (Compile Time) (N1) – The lower bound of the array.
(N2) – The upper bound of the array.
(N3) – Size, in bytes, of each array entry.
(Run Time) (N) – Position in the array to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of the array element.

Example of Use:

```
1900 1999 4 1?ARRAY POPULATION
```

This would define a 100-element array, each element being four bytes in length, called POPULATION. If each entry was a double-length number, this would print out the value of the eighty-seventh element of the array:

```
1986 POPULATION 2@ D.
```

Note that the array elements are numbered 1900–1999.

Algorithm: At compile time, allocate space for the array. The number of bytes needed is the number of elements times the size of each element. Store the size of each element. Store the lower bounds of the array. At run time, convert the passed element to an absolute number by subtracting the lower bound. Find the address of the element by multiplying the absolute element by the element size. Add this to the base address plus four (to skip the element size and the lower bound).

Suggested Extensions: None.

Definition:

```
: 1?ARRAY-RNG <BUILDS
    DUP . 3 PICK . LROT SWAP - 1+ *
    HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP >R 2+ @ - R> DUP @ ROT * + 4 + ;
```

1?ARRAY-RNG-BC (N1 N2 N3 –) (N – A)

Define and allocate space for a one-dimensional array with a specified element size. The lower and upper bounds of the array will also be specified. Bounds checking will be included.

Stack on Entry: (Compile Time) (N1) – The lower bound of the array.
(N2) – The upper bound of the array.
(N3) – Size, in bytes, of each array entry.
(Run Time) (N) – Position in the array to be accessed.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of the array element.

Example of Use:

```
42 51 2 1?ARRAY BUILD/ST
```

This would define a 10-element array, each element being two bytes in length, called BUILD/ST. The following would print the value of the first entry of BUILD/ST:

```
42 BUILD/ST ?
```

Since this array has bounds checking, an error will occur if it is passed an invalid element, like this:

```
66 BUILD/ST ?
Array out of bounds 66
```

Algorithm: At compile time, store the lower and upper bounds. Store the size of the array entries. Allocate space for the array. The number of bytes needed is the number of elements times the size of each element. At run time, see if the passed element is within the range specified at compile time. If it is, convert the passed element to an absolute number by subtracting the lower bound. Find the address of the element by multiplying the absolute element by the element size. Add this to the base address plus six (to skip the element size and the upper and lower bound).

Suggested Extensions: None.

Definition:

```
: 1?ARRAY-RNG-BC <BUILDS
    DUP , 3 PICK , OVER , LROT SWAP - 1+
    * HERE SWAP DUP ALLOT ERASE
  DOES>
    DUP >R OVER SWAP 2DUP 2+ @ >= >R
    4 + @ <= R> AND NOT IF
      OBM
    ELSE
      R> DUP >R 2+ @ -
      R> DUP @ ROT * + 6 +
    ENDIF ;
```

ARRAY (Nx N1 N2 –) (Nx – A)

Define and allocate space for an array of any dimensionality with each entry a specified size.

Stack on Entry: (Compile Time) (Nx) – One size for each dimension.
(N1) – The number of dimensions.
(N2) – Size, in bytes, of each array entry.
(Run Time) (Nx) – One position for each dimension of the array.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – Address of the array element.

Example of Use:

```
5 4 3 2 4 2 ARRAY EIGEN-VS
```

This would define a 5-by-4-by-3-by-2 four-dimensional array called EIGEN-VS. This word can handle arrays of any size, since it is limited only by memory. Each element is two bytes in length. This array has 120 elements. The code below compares the first and last elements of the array:

```
0 0 0 0 EIGEN-VS @ 4 3 2 1 EIGEN-VS @ =
```

Note that the array elements are numbered 0–5, 0–4, 0–3, and 0–2.

Algorithm: At compile time, allocate space for the array. The number of bytes needed is the number of elements times the size of each element. Store the size of each element. At run time multiply, each array element by the size of the array indices below it. Multiply that number by the entry size. Add it to the base address. Add 4 plus 2 times the number of array elements minus one, to skip the element sizes.

Suggested Extensions: Extend this word to use memory outside the 64K address space.

Definition:

```
0 VARIABLE ADR
0 VARIABLE ECOUNT
0 VARIABLE COUNTER
0 VARIABLE #ROLL

: ARRAY <BUILDS
    DUP , >R DUP , 1- BEGIN
      DUP 0 <>
    WHILE
      >R DUP , *
      R> 1-
    REPEAT DROP
```

```
  R> *
  HERE SWAP DUP ALLOT ERASE
DOES>
  ECOUNT 0SET DUP ADR ! 2 + @ DUP #ROLL !
  DUP 1- 2* ADR @ + 2+ COUNTER !
    1- BEGIN
    DUP 0 <>
  WHILE
    >R COUNTER @ @ -2 COUNTER +!
    #ROLL @ 1+ ROLL * ECOUNT +!
    -1 #ROLL +! R> 1-
  REPEAT DROP
  ECOUNT +! ECOUNT @ ADR @ @ *
  ADR @ 2+ @ 1- 2* 4 + + ADR @ + ;
```

DISK-2-MEM (D N1 N2 -)

Move data from disk to memory.

Stack on Entry: (D) – The disk address to move from.
(N1) – The memory address to move to.
(N2) – Number of bytes to move.

Stack on Exit: Empty.

Example of Use:

```
0. BOOT-SECTOR 512 DISK-2-MEM
```

This would move the first 512 bytes from a disk to the array of memory pointed to by BOOT-SECTOR.

Algorithm: Store the number of bytes and memory address in variables. Convert the disk address to a block and offset by dividing by 1024. Start a loop and fetch the block. Add the offset. Move the number of bytes left in the block or the number of bytes left to move, whichever is smaller. Decrement the number of bytes left to move by the number of bytes moved. Add one to the block being moved and zero the offset. Continue the loop if there are any bytes left to move.

Suggested Extensions: None.

Definition:

```
0 VARIABLE BYTES     0 VARIABLE MEM
```

```
0 VARIABLE #BLOCK      0 VARIABLE OA

: DISK-2-MEM
  BYTES ! MEM ! 1024 U/MOD #BLOCK !
  OA ! BEGIN
    #BLOCK @ BLOCK OA @ + MEM @
    1024 OA @ - BYTES @ 2DUP 2DUP U> IF
      SWAP
    ENDIF DROP DUP >R CMOVE R> DUP
    NEGATE BYTES +! MEM +! 1 #BLOCK +!
    OA 0SET BYTES @ 0=
  UNTIL ;
```

MEM-2-DISK (N1 D N2 -)

Move data from disk to memory.

Stack on Entry: (N1) – The memory address to move from.
(D) – The disk address to move to.
(N2) – Number of bytes to move.

Stack on Exit: Empty.

Example of Use:

```
BOOT-SECTOR 0, 512 MEM-2-DISK
```

This would move 512 bytes to the first block of a disk from the array of memory pointed to by BOOT-SECTOR.

Algorithm: Store the number of bytes and memory address in variables. Convert the disk address to a block and offset by dividing by 1024. Start a loop. Move the number of bytes left in the block (after adding the offset) or the number of bytes left to move, whichever is smaller. Decrement the number of bytes left to move by the number of bytes moved. Add one to the block being moved and zero the offset. Continue the loop if there are any bytes left to move.

Suggested Extensions: None.

Definition:

```
: MEM-2-DISK
  BYTES ! 1024 U/MOD #BLOCK ! OA !
  MEM ! BEGIN
```

```
    MEM @ #BLOCK @ BLOCK UPDATE OA @ +
    1024 OA @ - BYTES @ 2DUP 2DUP U> IF
      SWAP
    ENDIF DROP DUP >R CMOVE R> DUP
    NEGATE BYTES +! MEM +! 1 #BLOCK +!
    OA 0SET BYTES @ 0=
  UNTIL ;
```

(This version flushes all buffers before it begins, but is better for larger moves.)

```
: MEM-2-DISK
  FLUSH
  BYTES ! 1024 U/MOD #BLOCK ! OA !
  MEM ! BEGIN
    MEM @
    1024 OA @ - BYTES @ 2DUP 2DUP U> IF
      SWAP
    ENDIF DROP DUP >R CMOVE R> DUP
    DUP 1024 = IF
      #BLOCK @ BUFFER SWAP CMOVE UPDATE
    ELSE
      #BLOCK @ BLOCK OA @ + SWAP CMOVE
      UPDATE
    ENDIF
    R> DUP
    NEGATE BYTES +! MEM +! 1 #BLOCK +!
    OA 0SET BYTES @ 0=
  UNTIL ;
```

DISK-ARRAY (Nx N1 N2 D -) (Nx - D)

Define and allocate space for an array of any dimensionality with each entry a specified size. The array will reside on disk at a specified address.

Stack on Entry: (Compile Time) (Nx) - One size for each dimension.
(N1) - The number of dimensions.
(N2) - Size, in bytes, of each array entry.
(D) - The starting disk position for the entry.
(Run Time) (Nx) - One position for each dimension of the array.

Stack on Exit: (Compile Time) Empty.
(Run Time) (D) – The disk address of the array element.

Example of Use:

```
100 2 16 100,000 DISK-ARRAY QAZ
```

This would define a 100-by-2 disk array named QAZ. Each element of QAZ is sixteen bytes long. This would move an element of QAZ into a sixteen-byte buffer called 16BB.

```
8 1 QAZ 16BB 16 DISK-2-MEM
```

Algorithm: At compile time, store the size of each element. Store the starting disk address. At run time, multiply each array element by the size of the array indices below it. Multiply that number by the entry size. Add it to the base disk address.

Suggested Extensions: None.

Definition:

```
0 VARIABLE ADR
0 VARIABLE ECOUNT
0 VARIABLE COUNTER
0 VARIABLE #ROLL

: DISK-ARRAY <BUILDS
    ,,
    , DUP , 1- BEGIN
      DUP 0 <>
    WHILE
      >R DUP , *
      R> 1-
    REPEAT
    2DROP
  DOES> [ ->
    4 +
    ECOUNT 0SET DUP ADR ! 2 + @ DUP #ROLL !
    DUP 1- 2* ADR @ + 2+ COUNTER !
    1- BEGIN
      DUP 0 <>
    WHILE
      >R COUNTER @ @ -2 COUNTER +!
      #ROLL @ 1+ ROLL * ECOUNT +!
      -1 #ROLL +! R> 1-
```

```
REPEAT DROP
ECOUNT +! ECOUNT @ ADR @ @ *
ADR @ 4 - 2 @ ROT M+ ;
```

RECORD (-) (- N)

Start a record definition.

Stack on Entry: (Compile Time) Empty.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) (N) - The number of bytes in the record.

Example of Use:

```
RECORD PERSON
```

This would start the definition of a record called PERSON.

Algorithm: At compile time, store a zero in the dictionary. This cell will be filled in by END-RECORD with the size of the record. Store the address of the cell in FILL-ADDR for END-RECORD.

Suggested Extensions: None.

Definition:

```
0 VARIABLE CUR-SIZE
0 VARIABLE FILL-ADDR
0 VARIABLE VAR-START
0 VARIABLE %VAR-END

: RECORD <BUILDS
    HERE FILL-ADDR !
    CUR-SIZE 0SET 0 ,
  DOES>
    @ ;
```

FIELD (N -) (A1 - A2)

Define a field in a record.

Stack on Entry: (Compile Time) (N) - The number of bytes to allocate to the field.

(Run Time) (A1) – The base address of the record.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A2) – The address of the field in the record.

Example of Use:

```
RECORD PERSON
  16 FIELD NAME
```

This would define NAME, a field of 16 bytes in the record PERSON. If JAMES was an instance of the record PERSON, this code would leave the address of the name field of JAMES on the stack:

```
JAMES NAME
```

Algorithm: Store the current size of the record at compile time. Then add the size of the current field to CUR–SIZE. At run time, add the number stored to the address on the stack.

Suggested Extensions: None.

Definition:

```
: FIELD <BUILDS
    CUR-SIZE DUP @ , +!
  DOES>
    @ + ;
```

VARIANT (–)

Start a variant in a record definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD PERSON
  16 FIELD NAME
  1 FIELD SEX
  START-VARIANTS
  VARIANT
    2 FIELD #WIVES
```

```
  END-VARIANT
  VARIANT
    2 FIELD #HUSBANDS
    2 FIELD #CHILDREN
  END-VARIANT
  END-ALL-VARIANTS
END-RECORD
```

VARIANT starts two variants in this record.

Algorithm: Store the current size of the record in the variant start. This will be used by END-VARIANT to reset CUR-SIZE.

Suggested Extensions: None.

Definition:

```
: VARIANT
  CUR-SIZE @ VAR-START ! ;
```

START-VARIANTS (-)

Start a set of variants in a record definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD PART
  2 FIELD LOCATION
  1 FIELD TYPE
  START-VARIANTS
  VARIANT
    2 FIELD ALENGTH
  END-VARIANT
  VARIANT
    2 FIELD BLENGTH
    2 FIELD WEIGHT
  END-VARIANT
  END-ALL-VARIANTS
END-RECORD
```

START-VARIANTS must be used before any variants are defined.

Algorithm: Zero %VAR–END. This will be used to hold the largest variant defined, so the CUR–SIZE can be correctly set when the variants are complete.

Suggested Extensions: None.

Definition:

```
: START–VARIANTS
  %VAR–END 0SET ;
```

END-VARIANT (–)

End the definition of a variant portion of a record.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD QUASAR
   16 FIELD NAME
   1 FIELD RAD–TYPE
   START–VARIANTS
   VARIANT
      1 FIELD ALPHA
      1 FIELD BETA
   END–VARIANT
   VARIANT
      1 FIELD GAMMA
   END–VARIANT
   END–ALL–VARIANTS
END–RECORD
```

END–VARIANT ends two variants in this record.

Algorithm: Store the larger of CUR–SIZE and %VAR–END in %VAR–END. Store the VAR–START in CUR–SIZE.

Suggested Extensions: None.

Definition:

```
: END–VARIANT
```

```
  CUR-SIZE DUP >R @
 % VAR-END DUP >R @
  MAX R> !
  VAR-START @ R> ! ;
```

END-ALL-VARIANTS (-)

Complete the definition of a set of variants in a record.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD PITCH
   4 FIELD NAME
   2 FIELD ROTATION
   START-VARIANTS
   VARIANT
      1 FIELD XROT
   END-VARIANT
   VARIANT
      1 FIELD YROT
      1 FIELD ZROT
   END-VARIANT
   END-ALL-VARIANTS
END-RECORD
```

END-ALL-VARIANTS must be used after a set of variants has been defined.

Algorithm: Set CUR-SIZE to the end of the largest variant.

Suggested Extensions: None.

Definition:

```
: END-ALL-VARIANTS
   %VAR-END@ CUR-SIZE ! ;
```

INSERT (–) (A1 – A2)

Insert a record as a subrecord in a record being defined.
Used as: INSERT <name> OF <RECORD–NAME>

Stack on Entry: (Compile Time) Empty.
(Run Time) (A1) – The base address of the record.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A2) – The address of the sub-record in the record.

Example of Use:

```
RECORD PLAYER
  INSERT PINFO OF PERSON
  1 FIELD POSITION
  2 FIELD RATING
END–RECORD
```

If BOUTON was an instance of PLAYER, the following code would leave the address of the name field, from the subrecord, on the stack:

```
BOUTON PINFO NAME
```

Algorithm: At compile time, skip the word AS. Look up the record word with "tick" and execute it. Store the size of the record. At run time, add the start of the sub-record to the address on the stack.

Suggested Extensions: None.

Definition:

```
: INSERT <BUILDS
    BL WORD DROP ' EXECUTE
    CUR–SIZE DUP @ , + !
  DOES>
    @ + ;
```

END–RECORD (–)

Complete the definition of a record.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD SALE
   3 FIELD SALESPERSON
   2 FIELD CLIENT
   4 FIELD AMOUNT
END-RECORD
```

END-RECORD must be used for each record.

Algorithm: Fill in the size of the record.

Suggested Extensions: None.

Definition:

```
: END-RECORD
   CUR-SIZE @ FILL-ADDR @ ! ;
```

1AFIELD (N1 N2 -) (A1 N - A2)

Define an array field in a record.

Stack on Entry: (Compile Time) (N1) - The number of entries.
(N2) - The number of bytes to allocate to each entry.
(Run Time) (N1) - The base address of the record.
(N) - The entry to access.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A2) - The address of the field in the record.

Example of Use:

```
RECORD BBALL-SCORE
   4 2 FIELD HOME-SCORE
   4 2 FIELD AWAY-SCORE
END-RECORD
```

HOME-SCORE and AWAY-SCORE will de defined as array field with

four entries each. The code below would compare two scores stored in an instance of this record called 1GAME.

```
1GAME 2 HOME-SCORE @ 1GAME 2 AWAY-SCORE @ =
```

Algorithm: At compile time, store the size of each entry. Add the size of the total field to CUR-SIZE. At run time, multiply the requested entry number by the entry size and add it to the base address on the stack.

Suggested Extensions: Bounds checking, range arrays, and multiple-dimension array fields could be defined if desired.

Definition:

```
: 1AFIELD <BUILDS
    DUP, CUR-SIZE DUP @ ,
    >R * R> +!
  DOES>
    DUP @ ROT * SWAP 2+ @ + + ;
```

INSTANCE (-) (- A)

Define an instance of a record.
Used as: INSTANCE <name> OF <RECORD-NAME>

Stack on Entry: (Compile Time) Empty.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – The base address of the record.

Example of Use:

```
INSTANCE RUTH OF PLAYER
```

This would create an instance of the record PLAYER, called RUTH.

Algorithm: At compile time, skip the OF. "Tick" the record word and determine how many bytes long it is. Allocate space for the record. At run time, leave the address on the stack.

Suggested Extensions: None.

Definition:

```
: INSTANCE <BUILDS
    BL WORD DROP ' EXECUTE ALLOT
  DOES> ;
```

M-INSTANCE (N -) (N - A)

Define a multiple instance of a record.
Used as: <N> M-INSTANCE <name> OF <RECORD-NAME>

Stack on Entry: (Compile Time) (N) The number of instances to define.
(Run Time) (N) The number of the instance desired.

Stack on Exit: (Compile Time) Empty.
(Run Time) (A) – The base address of the record.

Example of Use:

```
25 M-INSTANCE YANKEES OF PLAYER
```

This would create an 25 element array of the record PLAYER, called YANKEES. This would print the name of the fourth player:

```
3 YANKEES NAME 16 TYPE
```

Algorithm: At compile time, skip the OF. "Tick" the record word and determine how many bytes long it is. Store the record length. Allocate space for all of the records. At run time, multiply the requested position by the length of each record. Leave the address on the stack.

Suggested Extensions: Bounds checking, ranges, and multiple dimensions can be defined, if needed.

Definition:

```
: M-INSTANCE <BUILDS
    BL WORD DROP
    'EXECUTE DUP , * ALLOT
  DOES>
    DUP @ ROT * + 2 + ;
```

RECORD (–) (– N)

(Disk Version)
Start a record definition.

Stack on Entry: (Compile Time) Empty.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) (N) – The number of bytes in the record.

Example of Use:

```
RECORD PERSON
```

This would start the definition of a record called PERSON.

Algorithm: At compile time store a zero in the dictionary. This cell will be filled in by END–RECORD with the size of the record. Store the address of the cell in FILL–ADDR for END–RECORD.

Suggested Extensions: None.

Definition:

```
0 VARIABLE CUR-SIZE
0 VARIABLE FILL-ADDR
0 VARIABLE VAR-START
0 VARIABLE %VAR-END

: RECORD <BUILDS
    HERE FILL-ADDR !
    CUR-SIZE 0SET 0 ,
  DOES>
    @ ;
```

FIELD (N –) (D1 – D2)

(Disk Version)
Define a field in a record.

Stack on Entry: (Compile Time) (N) – The number of bytes to allocate to the field.
(Run Time) (D1) – The base disk address of the record.

Stack on Exit: (Compile Time) Empty.
(Run Time) (D2) – The disk address of the field in the record.

Example of Use:

```
RECORD PERSON
  16 FIELD NAME
```

This would define NAME, a field of 16 bytes in the record PERSON. If JAMES was an instance of the record PERSON, this code would leave the disk address of the name field of JAMES on the stack:

```
JAMES NAME
```

Algorithm: Store the current size of the record at compile time. Then add the size of the current field to CUR–SIZE. At run time, add the number stored to the disk address on the stack.

Suggested Extensions: None.

Definition:

```
: FIELD <BUILDS
  CUR-SIZE DUP @ , + !
DOES>
  @ M+ ;
```

VARIANT (–)

(Disk Version)
Start a variant in a record definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD PERSON
  16 FIELD NAME
  1 FIELD SEX
  START–VARIANTS
  VARIANT
    2 FIELD #WIVES
  END–VARIANT
  VARIANT
    2 FIELD #HUSBANDS
```

```
    2 FIELD #CHILDREN
  END-VARIANT
  END-ALL-VARIANTS
END-RECORD
```

VARIANT starts two variants in this record.

Algorithm: Store the current size of the record in the variant start. This will be used by END-VARIANT to reset CUR-SIZE.

Suggested Extensions: None.

Definition:

```
: VARIANT
  CUR-SIZE @ VAR-START ! ;
```

START-VARIANTS (-)

(Disk Version.)
Start a set of variants in a record definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD PART
  2 FIELD LOCATION
  1 FIELD TYPE
  START-VARIANTS
  VARIANT
    2 FIELD LENGTH
  END-VARIANT
  VARIANT
    2 FIELD LENGTH
    2 FIELD WEIGHT
  END-VARIANT
  END-ALL-VARIANTS
END-RECORD
```

START-VARIANTS must be used before any variants are defined.

Algorithm: Zero %VAR-END. This will be used to hold the largest variant

defined, so the CUR-SIZE can be correctly set when the variants are complete.

Suggested Extensions: None.

Definition:

```
: START-VARIANTS
  %VAR-END 0SET ;
```

END-VARIANT (-)

(Disk version.)
End the definition of a variant portion of a record.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD QUASAR
   16 FIELD NAME
   1 FIELD RAD-TYPE
   START-VARIANTS
   VARIANT
      1 FIELD ALPHA
      1 FIELD BETA
   END-VARIANT
   VARIANT
      1 FIELD GAMMA
   END-VARIANT
   END-ALL-VARIANTS
END-RECORD
```

END-VARIANT ends two variants in this record.

Algorithm: Store the larger of CUR-SIZE and %VAR-END in %VAR-END. Store the VAR-START in CUR-SIZE.

Suggested Extensions: None.

Definition:

```
: END-VARIANT
```

```
  CUR-SIZE DUP >R @
  %VAR-END DUP >R @
  MAX R> !
  VAR-START @ R> ! ;
```

END-ALL-VARIANTS (-)

(Disk version.)
Complete the definition of a set of variants in a record.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD PITCH
  4 FIELD NAME
  2 FIELD ROTATION
  START-VARIANTS
  VARIANT
    1 FIELD XROT
  END-VARIANT
  VARIANT
    1 FIELD YROT
    1 FIELD ZROT
  END-VARIANT
  END-ALL-VARIANTS
END-RECORD
```

END-ALL-VARIANTS must be used after a set of variants has been defined.

Algorithm: Set CUR-SIZE to the end of the largest variant.

Suggested Extensions: None.

Definition:

```
: END-ALL-VARIANTS
  %VAR-END@ CUR-SIZE ! ;
```

INSERT (–) (D1 – D2)

(Disk version.)
Insert a record as a subrecord in a record being defined.
Used as: INSERT <name> OF <RECORD–NAME>

Stack on Entry: (Compile Time) Empty.
(Run Time) (D1) – The base disk address of the record.

Stack on Exit: (Compile Time) Empty.
(Run Time) (D2) – The disk address of the sub-record in the record.

Example of Use:

```
RECORD PLAYER
  INSERT PINFO OF PERSON
  1 FIELD POSITION
  2 FIELD RATING
END–RECORD
```

If BOUTON was an instance of PLAYER, the following code would leave the disk address of the name field, from the subrecord, on the stack:

```
BOUTON PINFO NAME
```

Algorithm: At compile time, skip the word AS. Look up the record word with "tick" and execute it. Store the size of the record. At run time, add the start of the sub-record to the disk address on the stack.

Suggested Extensions: None.

Definition:

```
: INSERT <BUILDS
    BL WORD DROP ' EXECUTE
    CUR–SIZE DUP @ . +!
  DOES>
    @ M+ ;
```

END–RECORD (–)

(Disk version.)
Complete the definition of a record.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RECORD SALE
  3 FIELD SALESPERSON
  2 FIELD CLIENT
  4 FIELD AMOUNT
END-RECORD
```

END-RECORD must used for each record.

Algorithm: Fill in the size of the record.

Suggested Extensions: None.

Definition:

```
: END-RECORD
  CUR-SIZE @ FILL-ADDR @ ! ;
```

1AFIELD (N1 N2 –) (D1 N – D2)

(Disk version.)
Define an array field in a record.

Stack on Entry: (Compile Time) (N1) – The number of entries.
(N2) – The number of bytes to allocate to each entry.
(Run Time) (D1) – The base disk address of the record.
(N) – The entry to access.

Stack on Exit: (Compile Time) Empty.
(Run Time) (D2) – The disk address of the field in the record.

Example of Use:

```
RECORD BBALL-SCORE
  4 2 FIELD HOME-SCORE
  4 2 FIELD AWAY-SCORE
END-RECORD
```

HOME-SCORE and AWAY-SCORE will be defined as array fields with

four entries each. The code below would compare two scores stored in an instance of this record called 1GAME.

```
1GAME 2 HOME-SCORE 1B 2 DISK-2-MEM 1B @
1GAME 2 AWAY-SCORE 1B 2 DISK-2-MEM 1B @ =
```

Algorithm: At compile time, store the size of each entry. Add the size of the total field to CUR-SIZE. At run time, multiply the requested entry number by the entry size and add it to the base address on the stack.

Suggested Extensions: Bounds checking, range arrays, and multiple dimension array fields could be defined if desired.

Definition:

```
: 1AFIELD <BUILDS
    DUP , CUR-SIZE DUP @ ,
    >R * R> +!
  DOES>
    DUP @ ROT * SWAP 2+ @ + M+ ;
```

INSTANCE (D –) (– D)

(Disk version.)
Define an instance of a record.
Used as: <D> INSTANCE <name> OF <RECORD-NAME>

Stack on Entry: (Compile Time) (D) – The starting disk address of the record.
(Run Time) Empty.

Stack on Exit: (Compile Time) Empty.
(Run Time) (D) – The base disk address of the record.

Example of Use:

```
45,000 INSTANCE RUTH OF PLAYER
```

This would create an instance of the record PLAYER, called RUTH. It would start at position 45,000 in disk storage.

Algorithm: At compile time, store the start position. Skip the OF. "Tick" the record word and determine how many bytes long it is. At run time, fetch the start and leave the disk address on the stack.

Suggested Extensions: None.

Definition:

```
: INSTANCE <BUILDS
    ''
    BL WORD DROP ' EXECUTE DROP
  DOES> DUP @ SWAP 2+ @ SWAP ;
```

M-INSTANCE (D N -) (N - D)

(Disk version.)
Define a multiple instance of a record.
Used as: <N> <D> M-INSTANCE <name> OF <RECORD-NAME>

Stack on Entry: (Compile Time) (N) – The number of instances to define.
(D) – The starting disk address of the records.
(Run Time) (N) – The number of the instance desired.

Stack on Exit: (Compile Time) Empty.
(Run Time) (D) – The base disk address of the record.

Example of Use:

```
25 250,000 M-INSTANCE YANKEES OF PLAYER
```

This would create an 25-element array of the record PLAYER, called YANKEES. It would be stored starting at disk location 250,000. This would print the name of the fourth player:

```
3 YANKEES NAME 16B 16 DISK-2-MEM 16B 16 TYPE
```

Algorithm: At compile time, store the start disk address. Skip the OF. "Tick" the record word and determine how many bytes long it is. Store the record length. At run time, multiply the requested position by the length of each record. Add it to the disk start address. Leave the address on the stack.

Suggested Extensions: Bounds checking, ranges, and multiple dimensions can be defined if needed.

Definition:

```
: M-INSTANCE <BUILDS
```

```
  ,, BL WORD DROP DROP
  ' EXECUTE ,
DOES>
  DUP 4 + @ ROT * >R DUP @ SWAP 2+ @ SWAP R> M+ ;
```

TO (–)

Cause a "to" type variable to execute a store.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
0 TO TIME
```

If TIME has been defined as a "to" type variable, the above code will store a zero in TIME.

Algorithm: Set the variable &TO to true.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE &TO: TO &TO C1SET ;
```

TVARIABLE (N –) ((N1) – (N2))

Define and initialize a cell-sized "to" type variable.

Stack on Entry: (Compile Time) (N) – The initial value of the variable.
(Run Time) (N1) – The value to store in the variable, if TO is being used.

Stack on Exit: (Compile Time) Empty.
(Run Time) (N2) – The value of the variable, if TO is not used.

Example of Use:

```
0 TVARIABLE TIME
```

TIME would be defined as a cell sized "to" type variable.

Algorithm: At compile time, store the initial value. At run time, check the value of &TO. If it is true, store the value on the stack in the variable and reset &TO to false. If &TO was false, fetch the value of the variable.

Suggested Extensions: None.

Definition:

```
: TVARIABLE <BUILDS , DOES> &TO C@
  IF &TO C0SET ! ELSE @ ENDIF ;
```

TCVARIABLE (N –) ((N1) – (N2))

Define and initialize a byte sized "to" type variable.

Stack on Entry: (Compile Time) (N) – The initial value of the variable.
(Run Time) (N1) – The value to store in the variable, if TO is being used.

Stack on Exit: (Compile Time) Empty.
(Run Time) (N2) – The value of the variable, if TO is not used.

Example of Use:

```
0 TVARIABLE BONZO
```

BONZO would be defined as a byte sized "to" type variable.

Algorithm: At compile time, store the initial value. At run time, check the value of &TO. If it is true, store the value on the stack in the variable and reset &TO to false. If &TO was false, fetch the value of the variable.

Suggested Extensions: None.

Definition:

```
: TCVARIABLE <BUILDS C, DOES> &TO C@
  IF &TO C0SET C! ELSE C@ ENDIF ;
```

FIFO (N –) (– A)

Create a queue or fifo stack.

Stack on Entry: (Compile time) (N) – The number of entries in the queue.
(Run Time) Empty.

Stack on Exit: (Compile time) Empty.
(Run time) (A) – The address of the queue.

Example of Use:

```
10 FIFO TASKS
```

TASKS would be defined as a queue or fifo stack with room for 10 cell-sized entries.

Algorithm: At compile time, store space for the start and end pointers for the queue. Store the queue size. Initialize the current start and end to 3. At run time, just leave the address.

Suggested Extensions: Words for different–sized elements would be a useful extension.

Definition:

```
: FIFO <BUILDS 1+ DUP 2 + HERE ! HERE
  SWAP 3 + 2* ALLOT 2+ DUP 3 SWAP ! 2+ 3
  SWAP ! DOES> ;
```

(#FI) (A – A N1 N2)

Fetch the start and end pointers of a queue.

Stack on Entry: (A) – The address of the queue.

Stack on Exit: (A) – The address of the queue.
(N1) – The start pointer of the queue.
(N2) – The end pointer of the queue.

Example of Use: See words defined below.

Algorithm: The start is stored at the queue address plus two. The end is at the

queue address plus four. Fetch the two values.

Suggested Extensions: None.

Definition:

```
: (#FI) DUP DUP 2+ @ SWAP 4 + @ ;
```

?FIFO (A – F)

Determine if anything is stored in a queue.

Stack on Entry: (A) – The address of the queue.

Stack on Exit: (F) – A Boolean flag, true if any data are being held in the queue.

Example of Use:

```
TASKS ?FIFO .
```

This would print out a true flag if any data were being held in the queue TASKS.

Algorithm: If the start and the end pointers are equal, the queue is empty.

Suggested Extensions: None.

Definition:

```
: ?FIFO (#FI) <> SWAP DROP ;
```

CNO (A – F)

Abort with an error message if a queue is empty.

Stack on Entry: (A) – The address of the queue.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: If the start and the end pointers are equal, the queue is empty. If this is the case, abort with an error message.

Suggested Extensions: None.

Definition:

```
: CNO (#FI) = ABORT" Fifo Error" ;
```

@FIFO (A – N)

Remove data from a queue.

Stack on Entry: (A) – The address of the queue.

Stack on Exit: (N) – The last cell entered in the queue not yet fetched.

Example of Use:

```
TASKS @FIFO
```

Algorithm: Check to make sure the queue is not empty. If it is not, fetch the cell pointed to by the start pointer. Increment the start pointer. Wrap around to three if it passes the end of the queue.

Suggested Extensions: None.

Definition:

```
: @FIFO C@ DUP DUP 2+ @ 2* + @ SWAP DUP
  2+ 1 SWAP +! DUP DUP @ SWAP 2+ @ <
  IF 3 SWAP 2+ ! ELSE DROP ENDIF ;
```

!FIFO (N A –)

Store data in a queue.

Stack on Entry: (N) – The cell to store in the queue.
(A) – The address of the queue.

Stack on Exit: Empty.

Example of Use:

```
99 TASKS !FIFO
```

This would store a 99 in the queue TASKS.

Algorithm: Store the number in the position pointed to by the end pointer. Increment the pointer and make sure it does not pass the the start pointer. If it does, the queue has overflowed.

Suggested Extensions: None.

Definition:

```
: !FIFO DUP DUP 4 + @ 2* + ROT SWAP !
  DUP 4 + 1 SWAP +! DUP DUP @ SWAP 4 +
  @ < IF DUP 3 SWAP 4 + ! ENDIF CNO
  DROP ;
```

FIFO-RESET (A –)

Empty a queue.

Stack on Entry: (A) – The address of the queue.

Stack on Exit: Empty.

Example of Use:

```
TASKS FIFO-RESET
```

This would set the queue TASKS at empty.

Algorithm: Store a 3 in the start and end pointers.

Suggested Extensions: None.

Definition:

```
: FIFO-RESET DUP 2 + 3 SWAP! 4 + 3
  SWAP! ;
```

12

Expert Systems

Words Defined in This Chapter:

LTEXT	A string array that holds the text for conditions in the expert system.
LT–STATE	A byte array that holds the state of each string condition in the expert system.
RULES	A cell array that holds the address of each rule defined.
R	A byte array that holds the logical value of the level being analyzed.
R#LU	A variable holding the number of rules defined.
LT#LU	A variable holding the number of strings defined.
RULE	Start the definition of a rule.
END–RULE	Complete the definition of a rule.
GET$	Store a string, delimited by braces, in the variable STEMP.
EXIST$	Determine if a string is already stored in the expert system.
!LT	Add a string to the expert system.
SCOMPILE	Compile a string in a rule definition.
WHEN	Compile a WHEN condition in a rule definition.
&	Compile an AND condition in a rule definition.
|	Compile an OR condition in a rule definition.
HYPOTH	Compile an hypothesis in a rule definition.
EXPLAIN	Compile an explanation in a rule definition.
T/F	Prompt the user for and return a Boolean value.
F–S	Convert a Boolean flag to a literal state.

ASK-USER	Determine the state of a string in the expert system by asking the user.
[FIND-RULE]	Find a rule with a specific string as an hypothesis.
RULE-EVAL	A forward definition of the word RULE-EVAL.
UNKNOWN$	Attempt to set the state of an unknown string.
S-F	Convert a literal state to a Boolean flag.
$-EVAL	Leave the state of a string in the expert system.
COND-EVAL	Determine the truth of a condition.
+WHEN	Evaluate a WHEN condition in a rule.
-WHEN	Evaluate a WHEN NOT condition in a rule.
+AND	Evaluate an AND condition in a rule.
-AND	Evaluate an AND NOT condition in a rule.
+OR	Evaluate an OR condition in a rule.
-OR	Evaluate an OR NOT condition in a rule.
C/RULE-EVAL	Evaluate the conditional statements of a rule.
DO-EXPLAIN	Cause the expert system rule interpreter to explain its conclusions.
DON'T-EXPLAIN	Cause the expert system rule interpreter to suppress explanations.
.EXPLAIN	Execute the EXPLAIN clause of a rule definition.
DO-MON	Cause the expert system rule interpreter to print out all hypotheses reached.
DON'T-MON	Cause the expert system rule interpreter only to print the final conclusion reached.
.MON	Print the result of an HYPOTH statement.
TELL?	Should a Forth word hypothesis print out what it is doing?
$APPLY-RULE	Set the string hypothesis of a rule.
WORD-APPLY-RULE	Set the word hypothesis of a rule.
RULE-EVAL	Evaluate a rule.
RESET-SYSTEM	Clear all variables used by the expert system.
RT?	Is a rule's hypothesis known?
APPLY	Apply the rules of an expert system.

Expert systems are computer programs that attempt to duplicate the ability of human experts in a particular area of knowledge. Some of the most interesting work in computer science is taking place in Artificial Intelligence, or AI, and expert systems are among the most intriguing aspects of AI. Expert systems exist for medical diagnosis, searching for oil, and even configuring computer systems.

This chapter presents a set of words that will enable you to design your own expert system. Included in this chapter is a simple sample expert system, one that advises a baseball manager when to attempt a sacrifice bunt. The words presented in this chapter could be used to design an expert system on any subject.

Conceptually, an expert system can be viewed as two distinct parts. The first part is what is known as the rules. This is the format in which the expert knowledge we are trying to put to use is encoded. Here is a simple rule in the format of this chapter's expert system:

```
WHEN { A PERSON OWNS A ROLLS ROYCE }
HYPOTH { THE PERSON IS RICH }
```

Our rule says if a person owns a Rolls Royce, he or she must be rich. As we can see from this example, an expert system can only be as accurate as its rules. Obviously, a person could own a Rolls Royce and not be rich. If the hypothesis of this rule could be retracted later when other evidence was examined, then this rule might be more acceptable. A more sophisticated expert system than the one presented in this chapter would allow this back-tracking. In this expert system, once a conclusion is reached, it cannot be changed. Since this is the case, our rule should be more complete. Our one rule probably should be expanded.

```
WHEN { A PERSON OWNS AN EXPENSIVE CAR }
& { A PERSON OWNS AN EXPENSIVE HOUSE }
HYPOTH { THE PERSON IS RICH }

WHEN CASH>1,000,000
HYPOTH { THE PERSON IS RICH }

WHEN { A PERSON OWNS A ROLLS ROYCE }
|  { A PERSON OWNS A BENTLY }
|  { A PERSON OWNS A LOTUS }
HYPOTH { A PERSON OWNS AN EXPENSIVE CAR }
```

This set of rules has a few new wrinkles thrown in. Our conditions now include the logical operators AND (represented by &) and OR (represented by |). This allows the first rule, which is: if a person owns an expensive car, and a person owns an expensive house, then the person must be rich.

The second rule in the above example includes something not in between braces, which our expert system uses for literals. This is a normal Forth word. Forth words will be available in our expert system so that computation can be performed. All Forth words used in rules should leave a Boolean flag on the stack. Here is how CASH>1,000,000 might be written:

```
: CASH>1,000,000 CASH 2@ 1.000000 D> ;
```

A NOT operator (represented by --) is also available. Here is how it might be used:

WHEN -- { A PERSON OWNS AN EXPENSIVE CAR }
& -- { A PERSON OWNS AN EXPENSIVE HOUSE }
HYPOTH { THE PERSON DOES NOT HAVE AN OPULENT LIFE STYLE }

It will be important in our expert system to represent opposite conditions with the same literal, and to use the negation operator. If this is not done, our rule interpreter, which we'll get to in a moment, will not know we are talking about the same thing. For example:

WHEN { A PERSON IS MALE }
HYPOTH { THE PERSON HAS AN XY CHROMOSOME PAIR }

WHEN { A PERSON IS FEMALE }
HYPOTH { THE PERSON HAS AN XX CHROMOSOME PAIR }

These rules would get the computer to first ask if the person was male, and then to ask if the person was female. A better way to encode these rules would be:

WHEN { A PERSON IS MALE }
HYPOTH { THE PERSON HAS AN XY CHROMOSOME PAIR }

WHEN -- { A PERSON IS FEMALE }
HYPOTH { THE PERSON HAS AN XX CHROMOSOME PAIR }

In this way, the expert system can take advantage of knowledge it has already accumulated.

The second part of our expert system is the rule compiler and the rule interpreter. The rule compiler will read in rules as described previously and store them in memory. Our rule interpreter, the part of an expert system sometimes known as an inference engine, will try to use the rules to reach some conclusion. Here is how a rule will appear in memory:

0–2 Hypothesis Condition: 9 – Hypothesis True
10 – Hypothesis False
3–5 Hypothesis: Word/String Entry.
6–8 Explanation: Word/String Entry.

Then any number of the following:

9–11 Condition: 0 – End of List.
3 – WHEN
4 – WHEN NOT
5 – AND
6 – AND NOT
7 – OR
8 – OR NOT

12–15 Word/String Entry

A Word/String Entry looks like this:

Byte 0: 1 – String
2 – Word

Bytes 1–2: String Number or Word Address

Strings are held in a string array, and the state of each string (true, false, or unknown) is also held in an array.

Suggested Extensions: This chapter has numerous possibilities for extension. Among the most useful extension would be the inclusion of variables in the string rules. This would allow rules like:

```
WHEN { *A IS RICH }
HYPOTH { *A COULD OWN AN EXPENSIVE CAR }
```

This type of rule has much more power than the constant literals used in the expert system presented in this chapter.

Another useful extension would be to allow the retraction of hypotheses, assigning probabilities to conclusions, and parenthesized conditions in the rules. All in all, one could spend a lot of time with this chapter.

Please note: The words in this chapter make use of case statements from Chapter 2, strings from Chapter 7, and array words from Chapter 11.

LTEXT (N – A)

A string array which holds the text for conditions in the expert system.

Stack on Entry: (N) – The array member to access.

Stack on Exit: (A) – The address of the member.

Example of Use:

```
8 LTEXT $?
```

This would print the value of string number eight in the expert system.

Algorithm: Define a string array that allows each string to be up to 64 characters in length.

Suggested Extensions: None.

Definition:

```
100 64 SARRAY LTEXT
```

LT-STATE (N – A)

A byte array that holds the state of each string literal in the expert system.

Stack on Entry: (N) – The array member to access.

Stack on Exit: (A) – The address of the member.

Example of Use:

```
8 LT-STATE C?
```

This would print the state of string eight in our expert system.

Algorithm: Use 1CARRAY. The following values are used:

0 – Literal Unknown.
1 – Literal Known to be True.
2 – Literal Known to be False.

Suggested Extensions: None.

Definition:

```
100 1CARRAY LT-STATE
```

RULES (N – A)

A cell array that holds the address of each rule defined.

Stack on Entry: (N) – The array member to access.

Stack on Exit: (A) – The address of the member.

Example of Use:

```
8 RULES @ 20 DUMP
```

This would dump the first 20 bytes of rule eight.

Algorithm: Use 1ARRAY.

Suggested Extensions: None.

Definition:

```
100 1ARRAY RULES
```

R [] (N – A)

A byte array that holds the logical value of the level being analyzed.

Stack on Entry: (N) – The array member to access.

Stack on Exit: (A) – The address of the member.

Example of Use:

```
1 R [] C?
```

This would print the Boolean value at level one.

Algorithm: Use 1CARRAY.

Suggested Extensions: Thirty-two levels is probably enough to handle even quite complex rule systems. More could be added if desired by enlarging the array.

Definition:

```
32 1CARRAY R []
```

R#LU (– A)

A variable holding the number of rules defined.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of R#LU.

Example of Use:

```
R#LU ?
```

This would print the number of rules defined in the expert system.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 VARIABLE R#LU
```

LT#LU (– A)

A variable holding the number of strings defined.

Stack on Entry: Empty.

Stack on Exit: (A) – The address of LT#LU.

Example of Use:

```
1 LT#LU +!
```

This would increment the number of strings stored.

Algorithm: None.

Suggested Extensions: None.

Definition:

```
0 VARIABLE LT#LU
```

RULE (–)

Start the definition of a rule.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RULE
   WHEN { CLOUDS ARE IN THE SKY }
   HYPOTH { IT IS A BAD DAY TO TAN }
END-RULE
```

RULE must be used to start each rule definition.

Algorithm: Print a message. Store the current dictionary pointer in the array RULES. This is the start address of the rule being defined. Allot 9 bytes at this address to hold the hypothesis and the explanation.

Suggested Extensions: Have this word abort if too many rules are defined.

Definition:

```
: RULE
   CR ." Defining Rule ."
   R#LU DUP ? @
   HERE SWAP RULES !
   HERE 9 ERASE 9 ALLOT ;
```

ENDRULE (-)

Complete the definition of a rule.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
RULE
   WHEN { THE EYES ARE CLOSED }
      & { BREATHING IS REGULAR }
   HYPOTH { THE PATIENT IS ASLEEP }
END-RULE
```

END-RULE must be used to finish each rule definition.

Algorithm: Store zero in the dictionary. This terminates the list of conditions. Increment the number of rules.

Suggested Extensions: None.

Definition:

```
: END-RULE
   0 C, 1 R#LU +! ;
```

GET$ (–)

Store a string, delimited by braces, in the variable $TEMP.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Use WORD.

Suggested Extensions: None.

Definition:

```
125 CCONSTANT    | KEY

64 $VARIABLE $TEMP

: GET$
   | KEY WORD $TEMP $! ;
```

EXIST$ (– (N) F)

Determine if a string is already stored in the expert system.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if the string is found, false if it is not.
(N) – The number of the string if it does not already exist.

Example of Use: See words defined below.

Algorithm: If no strings are defined yet, exit the word with a false flag. Loop through all the strings defined. Compare the strings with $=. Exit the loop if a match is found.

Suggested Extensions: None.

Definition:

```
: $EXIST?
   LT#LU @ ?DUP IF
      0 SWAP 0 DO
         I LTEXT $@ $TEMP $@ $= IF
            DROP I -1 LEAVE
         ENDIF
      LOOP
   ELSE
      0
   ENDIF ;
```

!LT (-)

Add a string to the expert system.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
... EXITSS NOT IF !LT ENDIF ...
```

This would add the string in $TEMP to the expert system if it does not yet exist in the system.

Algorithm: Print a message. Store the string in the LTEXT array. Increment the number of strings in the system.

Suggested Extensions: Have this word abort if too many strings are defined.

Definition:

```
: !LT
  CR ." Compiling " $TEMP $?
  ." as string # " LT#LU ?
  $TEMP $@ LT#LU @ LTEXT $!
  1 LT#LU +! ;
```

$COMPILE (–)

Compile a string in a rule definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: If the string exists, just store its number in the dictionary. If it does not, add the string to the system, then store its number in the dictionary.

Suggested Extensions: None.

Definition:

```
: SCOMPILE
  SEXIST? IF

  ELSE
    LT#LU @ , !LT
  ENDIF ; –>
```

WHEN (–)

Compile a WHEN condition in a rule definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
WHEN -- { THE WALL IS GREEN }
```

This would compile a "while not string" condition in the expert system.

Algorithm: Search for a negation. If one is found, compile a WHEN NOT condition and parse the next token in the input stream. If no negation is found, compile a WHEN condition. If the next token is a left brace, compile a string. If it is not a brace, compile the word found.

Suggested Extensions: When the next token is compiled, make sure it is not

&, | , HYPOTH, or END–RULE. These are common error conditions.

Definition:

```
: -- ;

: { ;

: WHEN
  FIND [ ' -- ] LITERAL OVER = IF
    4 C, DROP FIND
  ELSE
    3 C,
  ENDIF 0 ,
  [ ' { ] LITERAL OVER = IF
    DROP 1 C, GET$ $COMPILE
  ELSE
    2 C, ,
  ENDIF ;
```

& (–)

Compile an AND condition in a rule definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
& 10%–INTEREST–RATE
```

This would compile an AND WORD condition in the expert system.

Algorithm: Search for a negation. If one is found, compile an AND NOT condition and parse the next token in the input stream. If no negation is found, compile an AND condition. If the next token is a left brace, compile a string. If it is not a brace, compile the word found.

Suggested Extensions: When the next token is compiled make sure it is not &, | , HYPOTH, or END–RULE. These are common error conditions.

Definition:

```
: &
```

```
FIND [ ' -- ] LITERAL OVER = IF
   6 C, DROP FIND
ELSE
   5 C,
ENDIF 0 ,
[ ' { ] LITERAL OVER = IF
   DROP 1 C, GET$ $COMPILE
ELSE
   2 C, ,
ENDIF ;
```

| (-)

Compile an OR condition in a rule definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
|   -- { PATIENT IS BLEEDING }
```

This would compile an OR NOT string condition in the expert system.

Algorithm: Search for a negation. If one is found, compile an OR NOT condition and parse the next token in the input stream. If no negation is found, compile an OR condition. If the next token is a left brace, compile a string. If it is not a brace, compile the word found.

Suggested Extensions: When the next token is compiled, make sure it is not &, | , HYPOTH, or END-RULE. These are common error conditions.

Definition:

```
:|
  FIND [ ' -- ] LITERAL OVER = IF
     8 C, DROP FIND
  ELSE
     7 C,
  ENDIF 0 ,
  [' { ] LITERAL OVER = IF
     DROP 1 C, GET$ $COMPILE
  ELSE
```

```
  2 C, ,
ENDIF ;
```

HYPOTH (–)

Compile an hypothesis in a rule definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
HYPOTH -- { RECESSION EXISTS }
```

This would compile an "hypothesis false string" condition in the expert system.

Algorithm: Search for a negation. If one is found, compile an "hypothesis false" condition and parse the next token in the input stream. If no negation is found, compile an "hypothesis true" condition. If the next token is a left brace, compile a string. If it is not a brace, compile the word found. Store all these values in the space allocated when the rule was defined.

Suggested Extensions: When the next token is compiled, make sure it is not &, | , HYPOTH, or END–RULE. These are common error conditions.

Definition:

```
: RAD R#LU @ RULES @ ;

: HYPOTH
  FIND [' -- ] LITERAL OVER = IF
    10 RAD C! DROP FIND
  ELSE
    9 RAD C!
  ENDIF
  [ ' { ] LITERAL OVER = IF
    DROP 1 RAD 3 + C!
    GET$ $EXIST? IF
      RAD 4 + !
    ELSE
      LT#LU @ RAD 4 + ! !LT
    ENDIF
  ELSE
```

```
    RAD 4 + ! 2 RAD 3 + C!
  ENDIF ;
```

EXPLAIN (–)

Compile an explanation in a rule definition.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

EXPLAIN { THE MOON IS RISING }

This would compile an explain statement in the expert system.

Algorithm: Find the next token in the input stream. If is a left brace, compile a string. If it is not a brace, compile the word found. Store all these values in the space allocated when the rule was defined.

Suggested Extensions: When the next token is compiled, make sure it is not &, |, HYPOTH, or END–RULE. These are common error conditions.

Definition:

```
: EXPLAIN
  FIND [ ' { ] LITERAL OVER = IF
    DROP 1 RAD 6 + C!
    GET$ $EXIST? IF
      RAD 7 + !
    ELSE
      LT#LU @ RAD 7 + ! !LT
    ENDIF
  ELSE
    RAD 7 + ! 2 RAD 6 + C!
  ENDIF ;
```

T/F (– F)

Prompt the user for and return a Boolean value.

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if a T is entered, false if an F is entered.

Example of Use: T/F .

This would prompt the user for a Boolean value, input it, and print it on the display.

Algorithm: Print the prompt "(T/F)". Input a keystroke. Continue until a "T" or "F" is input.

Suggested Extensions: None.

Definition:

```
: T/F
  CR ." (T/F) " 1 BEGIN
    DROP KEY DUP 95 > IF 32 – ENDIF
    DUP DUP 84 = SWAP 70 = OR
  UNTIL
  DUP EMIT 84 = ;
```

F–S (F – N)

Convert a Boolean flag to a literal state.

Stack on Entry: (F) – The Boolean flag to convert.

Stack on Exit: (N) – The state, 1 = True, 2 = False.

Example of Use:

```
T/F F–S 8 LT–STATE C!
```

This would set string eight's state to the value input by the user in T/F.

Algorithm: If the flag is true, leave a one; if it is false, leave a two.

Suggested Extensions: None.

Definition:

```
: F–S IF 1 ELSE 2 ENDIF ;
```

ASK-USER (N -)

Determine the state of a string in the expert system by asking the user.

Stack on Entry: (N) – The string to ask about.

Stack on Exit: Empty.

Example of Use:

```
8 ASK-USER
```

This would ask the user to set the state of string eight.

Algorithm: Print the string and prompt. Get a true or false from the user. Store it in LT-STATE.

Suggested Extensions: Extend the system to allow a don't know input. More than just this word would have to be modified for this extension.

Definition:

```
: ASK-USER
  CR ." Is the following condition true
  or false ? "
  CR DUP LTEXT $? T/F
  F-S SWAP LT-STATE C! ;
```

IND-RULE (N1 N2- (N3) F)

Find a rule with a specific string as an hypothesis.

Stack on Entry: (N1) – The string to ask about.
(N2) – The start position in the rule list to search.

Stack on Exit: (N3) – The rule number is one is found.
(F) – A Boolean flag, true if a rule is found; false if no rule is found.

Example of Use:

```
8 0 FIND-RULE
```

This would search for a rule that has string eight in its hypothesis.

Algorithm: Search through the array RULES. Start at the position passed on the stack. Check the hypothesis field of each rule, first for a string, and then for the particular string being sought. If it is found, exit the loop.

Suggested Extensions: None.

Definition:

```
: FIND-RULE
0 LROT R#LU @ SWAP DO
  I RULES @ 3 + C@ 1 = IF
    I RULES @ 4 + @ OVER = IF
      2DROP I -1 0 LEAVE
    ENDIF
  ENDIF
LOOP DROP ;
```

[RULE-EVAL] (-)

A forward definition of the word RULE-EVAL.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: The address of the word RULE-EVAL will be stored in /RULE-EVAL/. This word fetches that address and executes it.

Suggested Extensions: None.

Definition:

```
: DUMMY ;

' DUMMY VARIABLE /RULE-EVAL/

: [RULE-EVAL] /RULE-EVAL/ @ EXECUTE ;
```

S-F (N - F)

Convert a literal state to a Boolean flag.

Stack on Entry: (N) - The state, 1 = True, 2 = False.

Stack on Exit: (F) – The Boolean flag.

UNKNOWNS (N –)

Attempt to set the state of an unknown string.

Stack on Entry: (N) – The number of the string to set.

Stack on Exit: Empty.

Example of Use:

```
8 UNKNOWNS
```

This would attempt to ascertain the state of string number eight.

Algorithm: Use FIND–RULE to see if any rules have this string as an hypothesis. If a rule is found, attempt to evaluate it. If the rule sets the state of the string, exit the word. If it does not, continue looking for other rules that may have the string as an hypothesis. If no rules are found, or none set the state of the string, the string will be left in an unknown state.

Suggested Extensions: None.

Definition:

```
: UNKNOWNS 0 BEGIN
    OVER LROT FIND-RULE
  WHILE
    DUP [RULE-EVAL]
    OVER LT-STATE C@ IF
      2DROP EXIT
    ENDIF
    1+ DUP R#LU @ = IF
      2DROP EXIT
    ENDIF
  REPEAT DROP ;
```

Example of Use:

```
8 LT-STATE C@ S-F
```

This would leave a Boolean flag on the stack. The flag will be true if the state of rule eight was true.

Algorithm: If the flag is one, leave a true; otherwise leave a false.

Suggested Extensions: None.

Definition:

```
: S-F 1 = ;
```

$-EVAL (N – F)

Leave the state of a string in the expert system.

Stack on Entry: (N) – The number of the string to evaluate.

Stack on Exit: (F) – A Boolean flag representing the validity of the string.

Example of Use:

```
8 $-EVAL
```

This would leave a flag on the stack, true if string eight is true, false otherwise.

Algorithm: If the state of the string is known, return it and exit. Otherwise, let UNKOWNS try to set the state of the rule. If it could not ask the user as a last resort, convert the state to a flag and leave it on the stack.

Suggested Extensions: None.

Definition:

```
: S-EVAL
  DUP LT-STATE C@ ?DUP IF
    SWAP DROP S-F EXIT
  ENDIF
  DUP UNKNOWNS
  DUP LT-STATE C@ 0= IF
    DUP ASK-USER
  ENDIF
  LT-STATE C@ S-F ;
```

COND-EVAL (A – F)

Determine the truth of a condition.

Stack on Entry: (A) – The address of a condition.

Stack on Exit: (F) – A Boolean flag representing the truth of the condition.

Example of Use: See words defined below.

Algorithm: If the condition is a string, use S–EVAL. If it is a Forth word, execute it. The Forth word should leave a flag on the stack.

Suggested Extensions: Check to make sure that any Forth word that does execute leaves a flag on the stack.

Definition:

```
: COND–EVAL
  DUP C@ CASE
     1 =OF 1+ @ S–EVAL END–OF
     2 =OF 1+ @ EXECUTE END–OF
  ENDCASE ;
```

+WHEN (A –)

Evaluate a WHEN condition in a rule.

Stack on Entry: (A) – The address of a condition.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Since a WHEN must start every rule, store the result of the condition evaluation performed by COND–EVAL in the Boolean array for the level being analyzed.

Suggested Extensions: None.

Definition:

```
0 VARIABLE LEVEL

: +WHEN
   COND–EVAL LEVEL C@ R [] C! ;
```

-WHEN (A -)

Evaluate a WHEN NOT condition in a rule.

Stack on Entry: (A) – The address of a condition.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Since a WHEN must start every rule, store the logical negation of the result of the condition evaluation performed by COND–EVAL in the Boolean array for the level being analyzed.

Suggested Extensions: None.

Definition:

```
: -WHEN
  COND-EVAL NOT LEVEL C@ R [] C! ;
```

+AND (A -)

Evaluate an AND condition in a rule.

Stack on Entry: (A) – The address of a condition.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: An AND requires the current Boolean result for the level it is working on to be true before it attempts to evaluate the condition. If the level flag is false, the truth of this condition cannot change the flag setting. If the level flag is true, the condition will be evaluated using COND–EVAL and the result stored in the current Boolean result for the level.

Suggested Extensions: None.

Definition:

```
: +AND
  LEVEL C@ R [] C@ IF
    COND-EVAL LEVEL C@ R [] C!
```

```
ELSE
   DROP
ENDIF ;
```

2-AND (A -)

Evaluate an AND NOT condition in a rule.

Stack on Entry: (A) – The address of a condition.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: An AND requires the current Boolean result for the level it is working on to be true before it attempts to evaluate the condition. If the level flag is false, the truth of this condition cannot change the flag setting. If the level flag is true, the condition will be evaluated using COND–EVAL and the logical negation of the result stored in the current Boolean result for the level.

Suggested Extensions: None.

Definition:

```
: -AND
  LEVEL C@ R [] C@ IF
    COND-EVAL NOT LEVEL C@ R [] C!
  ELSE
    DROP
  ENDIF ;
```

+OR (A -)

Evaluate an OR condition in a rule.

Stack on Entry: (A) – The address of a condition.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: An OR requires the current Boolean result for the level it is working on to be false before it attempts to evaluate the condition. If the level flag is true, the truth of this condition cannot change the flag setting. If the level flag

is false, the condition will be evaluated using COND–EVAL and the result stored in the current Boolean result for the level.

Suggested Extensions: None.

Definition:

```
: +OR
  LEVEL C@ R [] C@ 0= IF
    COND-EVAL LEVEL C@ R [] C!
  ELSE
    DROP
  ENDIF ;
```

–OR (A –)

Evaluate an OR NOT condition in a rule.

Stack on Entry: (A) – The address of a condition.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: An OR requires the current Boolean result for the level it is working on to be false before it attempts to evaluate the condition. If the level flag is true, the truth of this condition cannot change the flag setting. If the level flag is false, the condition will be evaluated using COND–EVAL and the logical negation of the result stored in the current Boolean result for the level.

Suggested Extensions: None.

Definition:

```
: –OR
  LEVEL C@ R [] C@ 0= IF
    COND–EVAL NOT LEVEL C@ R [] C!
  ELSE
    DROP
  ENDIF ;
```

C/RULE–EVAL (N – F)

Evaluate the conditional statements of a rule.

Stack on Entry: (N) – The rule number to be evaluated.

Stack on Exit: (F) – The Boolean result of evaluating the conditions.

Example of Use: See words defined below.

Algorithm: This word loops through all the conditions for the rule it finds on the stack. First, it gets the address of the rule from the RULES array. Nine is added to this address to point to the start of the conditions. Each is evaluated until the end of the list is reached. The Boolean flag for the level is removed from the R [] array and left on the stack.

Suggested Extensions: None.

Definition:

```
: C/RULE-EVAL
  RULES @ 9 + BEGIN
    DUP C@ OVER 3 + SWAP CASE
      3 =OF +WHEN END-OF
      4 =OF -WHEN END-OF
      5 =OF +AND END-OF
      6 =OF -AND END-OF
      7 =OF +OR END-OF
      8 =OF -OR END-OF
    ENDCASE
    6 + DUP C@ 0=
  UNTIL DROP
  LEVEL C@ R [] C@ ;
```

DO-EXPLAIN (–)

Cause the expert system rule interpreter to explain its conclusions.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
DO-EXPLAIN
```

After this word is executed, whenever the rule interpreter reaches a conclusion, it will use the EXPLAIN clause to explain its actions.

Algorithm: Set the variable EXPLAIN? to true.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE EXPLAIN?

: DO-EXPLAIN EXPLAIN? C1SET ;
```

DON'T-EXPLAIN (-)

Cause the expert system rule interpreter to suppress explanations.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
DON'T-EXPLAIN
```

After this word is executed, no EXPLAIN clauses will be invoked.

Algorithm: Set the variable EXPLAIN? to false.

Suggested Extensions: None.

Definition:

```
: DON'T-EXPLAIN EXPLAIN? C0SET ;
```

.EXPLAIN (A -)

Execute the EXPLAIN clause of a rule definition.

Stack on Entry: (A) - The address of the rule.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: If EXPLAIN? is false, drop the address and exit the word. Otherwise, if the EXPLAIN clause is a string print it out. If the EXPLAIN clause is a word, execute it. If there is no explain clause, drop the address and exit.

Suggested Extensions: None.

Definition:

```
: .EXPLAIN EXPLAIN? C@ IF
    DUP 6 + C@ DUP 1 = IF
      DROP 7 + @ CR ." Because " LTEXT $?
    ELSE
      2 = IF
        7 + @ EXECUTE
      ELSE
        DROP
      ENDIF
    ENDIF
  ELSE
    DROP
  ENDIF ;
```

DO-MON (-)

Cause the expert system rule interpreter to print out all hypotheses reached.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

DO-MON

After this word is executed, all hypotheses set will be printed as they are set.

Algorithm: Set the variable MON? to true.

Suggested Extensions: None.

Definition:

```
0 CVARIABLE MON?

: DO-MON MON? C1SET ;
```

DON'T-MON (-)

Cause the expert system rule interpreter only to print the final conclusion reached.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
DON'T-MON
```

After this word is executed, only the final hypothesis set will be printed as it is set.

Algorithm: Set the variable MON? to false.

Suggested Extensions: None.

Definition:

```
: DON'T-MON MON? C0SET ;
```

.MON (A -)

Print the result of an HYPOTH statement.

Stack on Entry: (A) – The address of the rule.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: If the LEVEL is one, or the variable MON? is true, this word will print the value of the hypothesis being set. Only string hypotheses are printed out.

Suggested Extensions: None.

Definition:

```
: .MON MON? C@ LEVEL C@ 1 = OR IF
    DUP C@ 9 = IF
```

```
      CR ." True => "
    ELSE
      CR ." False => "
    ENDIF
    DUP 3 + C@ 1 = IF
      4 + @ LTEXT $?
    ELSE
      DROP
    ENDIF
  ELSE
    DROP
  ENDIF ;
```

TELL? (– F)

Should a Forth word hypothesis print out what it is doing?

Stack on Entry: Empty.

Stack on Exit: (F) – A Boolean flag, true if the word should print out what it is doing.

Example of Use:

```
: FEVER P–TEMP C@ 100 MIN P–TEMP C! TELL? IF
  ." Patient must have a temperature of at least 100 degrees "
  ENDIF ;
```

FEVER is a conclusion in an expert system. It uses ?TELL to determine whether or not it should report what action it is taking.

Algorithm: If the LEVEL is one or the variable MON? is true, return a true flag.

Suggested Extensions: None.

Definition:

```
: TELL? MON? C@ LEVEL C@ 1 = OR ;
```

$APPLY–RULE (A –)

Set the string hypothesis of a rule.

Stack on Entry: (A) – The address of the rule.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Call .MON to print out the setting of the hypothesis. A nine signifies a set hypothesis true, a ten indicates a set hypothesis false. The proper value is stored in the LT–STATE variable.

Suggested Extensions: None.

Definition:

```
: SAPPLY–RULE
  DUP .MON DUP C@ 9 = F–S
  SWAP 4 + @ LT–STATE C! ;
```

WORD–APPLY–RULE (A –)

Set the word hypothesis of a rule.

Stack on Entry: (A) – The address of the rule.

Stack on Exit: Empty.

Example of Use: See words defined below.

Algorithm: Execute the word stored in the hypothesis field of a rule.

Suggested Extensions: None.

Definition:

```
: WORD–APPLY–RULE
  4 + @ EXECUTE ;
```

RULE–EVAL (N –)

Evaluate a rule.

Stack on Entry: (N) – The number of the rule to evaluate.

Stack on Exit: Empty.

Example of Use:

```
8 RULE-EVAL
```

This would evaluate rule eight and set its hypothesis if the conditions of the rule we're found to be true.

Algorithm: Increment LEVEL for the new rule being evaluated. Use C/RULE-EVAL to determine the truth of the conditions for the rule. If the conditions evaluate true, set the hypothesis of the rule. This is done by executing either $APPLY-RULE or WORD-APPLY-RULE. Execute .EXPLAIN if the rule is applied. Decrement LEVEL when the word is exited. LEVEL is used since RULE-EVAL can be called recursivly through UNKNOWN$.

Suggested Extensions: None.

Definition:

```
: RULE-EVAL
   1 LEVEL C+!
   DUP RULES @ SWAP C/RULE-EVAL IF
      DUP DUP 3 + C@ CASE
         1 =OF $APPLY-RULE END-OF
         2 =OF WORD-APPLY-RULE END-OF
      ENDCASE .EXPLAIN
   ELSE
     DROP
  ENDIF -1 LEVEL C+! ;
```

(Store the address of RULE-EVAL in /RULE-EVAL/ to handle a forward reference.)

```
' RULE-EVAL /RULE-EVAL/ !
```

RESET-SYSTEM (-)

Clear all the variables used by the expert system.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

RESET-SYSTEM

Executing RESET-SYSTEM will allow the rule interpreter to start over fresh.

Algorithm: The number of rules and strings used is set to zero. The RULES array and LT-STATE arrays are cleared.

Suggested Extensions: None.

Definition:

```
: RESET-SYSTEM
   R#LU 0SET LT#LU 0SET
   0 RULES 200 ERASE
   0 LT-STATE 100 ERASE ;
```

RT? (A – F)

Is a rule hypothesis known ?

Stack on Entry: (A) – The address of the rule.

Stack on Exit: (F) – A Boolean flag, true if the state of the hypothesis of a rule is known.

Example of Use:

```
8 RT?
```

This will tell whether or not a rule's hypothesis has a known state.

Algorithm: Find the state of strings from LT-STATE. Execute words and use the flag they return.

Suggested Extensions: None.

Definition:

```
: RT?
3 + DUP C@ CASE
   1 =OF 1+ @ LT-STATE C@ NOT NOT END-OF
   2 =OF 1+ @ EXECUTE END-OF
ENDCASE ;
```

APPLY (–)

Apply the rules of an expert system.

Stack on Entry: None.

Stack on Exit: None.

Example of Use:

APPLY

This will start the execution of the rule interpreter.

Algorithm: Clear the state array. Execute the word found in /INIT/ that can be used by the expert system to initialize any of its variables. Start by trying to find a conclusion for each rule. Stop when an hypothesis is reached on level one. If all the rules are executed and no conclusion is reached, print a message.

Suggested Extensions: None.

Definition:

```
' DUMMY VARIABLE /INIT/

: APPLY
   0 LT-STATE 100 ERASE /INIT/ @ EXECUTE
   0 R [] 32 ERASE
   0 BEGIN
      DUP RULES @ RT? NOT IF
         DUP RULE-EVAL
      ENDIF
      1 R [] C@ IF DROP EXIT ENDIF
      1 + DUP R#LU @ = Y
   UNTIL DROP
   CR ." No conclusions reached " ;
```

A SAMPLE EXPERT SYSTEM

The expert system presented below will advise a baseball manager when to attempt a sacrifice bunt. It is simple and not that complete, but tries to show an example of each feature of the words presented in this chapter.

In our expert system, the order of the rules is crucial. Rules that reach intermediate conclusion should be placed at the end of the list of rules. If they

are not, the rule interpreter will terminate with a true, but not so interesting, conclusion. Of course, this is one area in which the words in this chapter could be expanded.

```
: (Y/N) ." (Y/N) " 1 BEGIN DROP KEY DUP
   95 > IF 32 - ENDIF DUP DUP 89 = SWAP 78
   = OR UNTIL DUP EMIT 89 = ;

0 CVARIABLE OUTS

: GET-OUTS
   CR ." How many outs are there? "
   #IN OUTS C! -1 ;

: NONE-OUT OUTS C@ 0= ;
: 1OUT OUTS C@ 1 = ;
: 2OUT OUTS C@ 2 = ;

0 CVARIABLE 1B
0 CVARIABLE 2B
0 CVARIABLE 3B

: 1B?
   CR ." Is there a runner on first ? "
   (Y/N) 1B C! -1 ;

: 1B?
   CR ." Is there a runner on first ? "
   (Y/N) 1B C! -1 ;

: 2B?
   CR ." Is there a runner on second ? "
   (Y/N) 2B C! -1 ;

: 3B?
   CR ." Is there a runner on third ? "
   (Y/N) 3B C! -1 ;

: 1B-HAS-RUNNER 1B C@ NOT NOT ;
: 2B-HAS-RUNNER 2B C@ NOT NOT ;
: 3B-HAS-RUNNER 3B C@ NOT NOT ;

0 CVARIABLE OUR-SCORE
: OUR-SCORE-GET
   CR ." How many runs do we have ? "
   #IN OUR-SCORE C! -1 ;
```

```
0 CVARIABLE THEIR-SCORE
: THEIR-SCORE-GET
   CR ." How many runs do they have ? "
   #IN THEIR-SCORE C! -1 ;


0 CVARIABLE INNING
: INNING-GET
   CR ." What inning is this ? "
   #IN INNING C! 0 ;
```

(INNING-GET will always be false. This allows this rule to input information. /INPUT/ could also be used.

```
RULE
   WHEN GET-OUTS & 1B? & 2B? & 3B?
   & OUR-SCORE-GET & THEIR-SCORE-GET
   & INNING-GET
   HYPOTH { JUNK }
   END-RULE

: AHEAD-BY-MORE-THAN-1
   OUR-SCORE C@ THEIR-SCORE C@ - 1 > ;

RULE
   WHEN { WE ARE HOME }
      & AHEAD-BY-MORE-THAN-1
   HYPOTH -- { BUNT }
END-RULE

: INNING-LATER-THEN-7TH
   INNING C@ 7 > ;

RULE
   WHEN -- { WE ARE HOME }
      & AHEAD-BY-MORE-THAN-1
      & -- INNING-LATER-THEN-7TH
   HYPOTH -- { BUNT }
   EXPLAIN { TOO EARLY TO BUNT }
END-RULE

RULE
   WHEN { BASES ARE EMPTY }
   HYPOTH -- { BUNT }
   EXPLAIN { NO RUNNERS TO MOVE UP }
```

```
END-RULE

RULE
  WHEN -- { WE ARE HOME }
    & AHEAD-BY-MORE-THAN-1
    & -- INNING-LATER-THEN-7TH
  HYPOTH -- { BUNT }
  EXPLAIN { TOO EARLY TO BUNT }
END-RULE

RULE
  WHEN { BASES ARE EMPTY }
  HYPOTH -- { BUNT }
  EXPLAIN { NO RUNNERS TO MOVE UP }
END-RULE

RULE
  WHEN 2OUT
  HYPOTH -- { BUNT }
  EXPLAIN { NO ONE TO MOVE OVER }
END-RULE

: UP-BY-ONE
  OUR-SCORE C@ THEIR-SCORE C@ - 1 = ;

: DOWN-BY-ONE
  THEIR-SCORE C@ OUR-SCORE C@ - 1 = ;

: TIED
  OUR-SCORE C@ THEIR-SCORE C@ = ;

RULE
  WHEN { FAST RUNNER ON FIRST }
    |  { AVERAGE RUNNER ON FIRST }
    & { ONLY 1B OCCUPIED }
    & { WE ARE HOME }
    & DOWN-BY-ONE
    & INNING-LATER-THEN-7TH
  HYPOTH { BUNT }
END-RULE

RULE
  WHEN { ONLY 1B OCCUPIED }
    & { BATTER IS A GOOD BUNTER }
    & { WE ARE HOME }
    & DOWN-BY-ONE
```

```
    & INNING-LATER-THEN-7TH
  HYPOTH { BUNT }
END-RULE

RULE
  WHEN { ONLY 1B OCCUPIED }
    & { BATTER IS A GOOD BUNTER }
    & -- { WE ARE HOME }
    & NONE-OUT
    & INNING-LATER-THEN-7TH
    & TIED
    | DOWN-BY-ONE
  HYPOTH { BUNT }
  EXPLAIN { CONSERVATIVE ON THE ROAD }
END-RULE

:T -1;
```

(This is used to prevent secondary conclusions as conclusions of the expert system.)

```
RULE WHEN T HYPOTH -- { BUNT } END-RULE

RULE
  WHEN 1B-HAS-RUNNER
    | 2B-HAS-RUNNER
    | 3B-HAS-RUNNER
  HYPOTH -- { BASES ARE EMPTY }
END-RULE

RULE
  WHEN 1B-HAS-RUNNER
    & -- 2B-HAS-RUNNER
    & -- 3B-HAS-RUNNER
  HYPOTH { ONLY 1B OCCUPIED }
END-RULE

RULE
  WHEN -- 1B-HAS-RUNNER
    & -- 2B-HAS-RUNNER
    & -- 3B-HAS-RUNNER
  HYPOTH { BASES ARE EMPTY }
END-RULE
```

13

Debugging Programs

Words Defined in This Chapter:

.S	Nondestructive stack print.
X	Simple debugging word.
ONDEBUG	Turn on complex debugging.
OFFDEBUG	Turn off complex debugging.
?NUMBER	Try to convert a string to a number.
COM-EX	Execute the debug commands.
Y	Debugging word.

The library routines presented in this book will save you a lot of debugging time. After all, the debugging has, hopefully, already been completed for the library words. But as you sit down to use the library routines and write your own programs, you will have to face debugging.

The single most powerful tool you have in debugging your Forth programs is your own programming style. If you keep your words short and self-contained, and you completely test each word as you write it, debugging should be a simple proposition. Forth makes the debugging of individual words simple and straightforward. You can place the arguments your word needs on the stack, store the proper values in the variables it uses, and then just execute your word. It really can be that simple.

The Three Rules:

1. Keep words short. The shorter a word is, the easier it is to debug. There is less that can go wrong. Try not to have words that stretch across more than a

single screen; ideally, they should be even shorter. Try to make each word perform a single logical function.

2. Keep words self-contained. Each word should be as much of a stand alone unit as possible. This means using the stack instead of variables. Many times it is proper to use a variable, but if your words are large and use lots of variables, it is a good indication that they can be broken down into smaller, simpler words.

3. (AND MOST IMPORTANT!!!) Test each word thoroughly as you write it! This is by far the most important principle presented in this chapter. Forth makes it easy for you to test your word. If you spend the time checking your word when you write it, you'll save a tremendous amount of debugging time. There is nothing worse than staring at a nonfunctioning word that looks perfectly correct, only to find out it is and the reason it's not functioning is three levels down. A thoroughly debugged word is a great asset; it can become an important tool for you. A word that has bugs is a time bomb waiting to sabotage your program.

When testing a word, pay particular attention to the upper and lower bounds of its arguments. If you have a string word that can handle strings up to 255 characters in length, check a few 255-character strings. Boundary areas are often a source of bugs.

A Few Helpful Words

When a really obstinate bug is holding you up, these words may be helpful. .S is a nondestructive stack print. It allows you to easily examine the stack without disturbing it. The word X can be placed throughout a section of code you are debugging to get a snapshot of the stack.

The words Y, ONDEBUG, and OFFDEBUG are powerful debugging words. Normally you can only print out the stack, dump memory, and examine the values of variables before and after you execute a word. With this set of words you can do all these things while a word is executing. Placing Y in a word can be thought of as setting a breakpoint. If debugging is trned on when a Y is encountered, the prompt DEBUG> will be printed on the screen. You can execute any word at this prompt. You can use .S to dump the stack, or even change the stack if you desire. Typing the word GO will continue execution. The word SKIP will continue execution and turn off all further breakpoints. To summarize:

Debug Commands:

ONDEBUG – Enable debugging.
OFFDEBUG – Disable debugging.

Y – A breakpoint; place it inside a word definition.

DEBUG> GO – Continue execution.
DEBUG> SKIP – Continue execution, skip all further breakpoints.

Suggested Extensions: The debugging commands are insufficient if you are debugging a word that produces screen output. They could be expanded to save the screen and restore it when a breakpoint is encountered.

.S (–)

Nondestructive stack print.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
1 2 3 .S
```

```
3
2
1
<BOTTOM OF STACK>
```

This is what .S would print if the stack was empty before the above line was typed in.

Algorithm: Determine first if the stack has any entries. If it does not, print the stack empty message and exit. If it is not empty, loop through each value on the stack and print it. The word PICK will not destroy the original stack entry.

Suggested Extensions: None.

Definition:

```
: .S CR DEPTH ?DUP IF
      1+ 1 DO I PICK . CR LOOP
      ." <BOTTOM OF STACK>" CR
  ELSE
      ." <STACK EMPTY>" CR
  ENDIF ;
```

X (–)

Dump the stack and pause.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
: TEST OVER DUP + X + . ;
```

When TEST is executed, the stack will be dumped after the first addition. Execution will be stopped until a key is hit so the programmer can examine the stack.

Algorithm: Use .S to dump the stack, then wait on a key.

Suggested Extensions: None.

Definition:

```
: X CR .S KEY DROP ;
```

ONDEBUG (–)

Turn on debugging.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
ONDEBUG
```

When this word is executed, all breakpoints encountered will allow the user to enter words at the DEBUG> prompt.

Algorithm: Set the variable /SKIP/ to false.

Suggested Extensions: None.

Definition:

```
1 CVARIABLE /SKIP/

: ONDEBUG /SKIP/ C0SET ;
```

OFFDEBUG (–)

Turn off debugging.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
OFFDEBUG
```

When this word is executed, execution will continue normally past all breakpoints.

Algorithm: Set the variable /SKIP/ to true.

Suggested Extensions: None.

Definition:

```
: OFFDEBUG /SKIP/ C1SET ;
```

?NUMBER (A – (N or D) F)

Attempt to convert a string to a number.

Stack on Entry: (A) – The address of the string.

Stack on Exit: (N or D) – The number if one is found. Double length if a comma is in the number.
(F) – A Boolean flag, true if a number is found.

Example of Use: See Y.

Algorithm: First, check for a leading negative sign. If one is found, increment the pointer past it. Then, attempt to convert the string to a number. If the con-

version stops on a comma, assume a double-length number and continue the conversion. In either case, if the conversion stops on a zero or blank, it is considered successful. Leave the number (negative, if appropriate) and a true flag on the stack. If conversion is unsuccessful, drop the number and leave a false flag.

Suggested Extensions: If floating-point numbers are being used, extend this word to allow them as input.

Definition:

```
45 CCONSTANT -KEY
44 CCONSTANT ,KEY

: ?NUMBER
  DUP 1+ C@ -KEY = IF
  1+ -1 >R
  ELSE
  0 >R
  ENDIF
  0. ROT >BINARY DUP C@ ,KEY = IF
     >BINARY C@ DUP BL = SWAP 0= OR IF
       R> IF DNEGATE ENDIF -1
     ELSE
       R> DROP 2DROP 0
     ENDIF
  ELSE
     SWAP DROP C@ DUP BL = SWAP 0= OR IF
       R> IF NEGATE ENDIF -1
     ELSE
       R> 2DROP 0
     ENDIF
  ENDIF ; ->
```

COM-EX (A -)

Check for a debug command. Execute a word if one is not found.

Stack on Entry: (A) – The address of a word.

Stack on Exit: Empty.

Example of Use: See Y.

Algorithm: COM-EX checks for the debug commands GO and SKIP. They

are defined as dummy words. If their address is the one on the stack, process them as commands, not as words to be executed. If the address on the stack is not SKIP or GO, execute the word at that address.

Suggested Extensions: New commands can be added to this debug package in this word. Define a dummy word with the name of your new command and then place an IF statement in COM-EX to handle it.

Definition:

```
: GO ;
: SKIP ;

: COM-EX
    [ ' GO ] LITERAL OVER = IF
      DROP R> DROP EXIT
    ELSE
      [ ' SKIP ] LITERAL OVER = IF
        DROP R> DROP OFFDEBUG EXIT
      ELSE
        EXECUTE
      ENDIF
    ENDIF ;
```

Y (-)

Set a breakpoint in a word.

Stack on Entry: Empty.

Stack on Exit: Empty.

Example of Use:

```
: TEST 10 0 DO
    I EVAL Y
  LOOP ;
```

The word TEST would have a breakpoint after EVAL. Everytime that part of the word is reached and debugging is turned on, Y becomes active.

Algorithm: First, check to see if debugging is on. If it is not, exit the word. Print out the debug prompt and obtain a line of input. Separate the words on the line by using BL WORD. If the first word can be found in the dictionary, pass it to COM-EX to handle. After executing it, determine if there are any words left on the line. If there are not, leave a false flag on the stack so a new

line of input will be obtained. If FIND could not locate the word in the dictionary, try to convert it to a number. If it is converted to a number, leave it on the stack and check for the end of line condition. If the word is not found in the dictionary and cannot be converted to a number, print out an error message, and leave a flag on the stack that will cause a new line of input to be obtained. The only exit from this word once the debug loop has begun is from COM-EX.

Suggested Extensions: None.

Definition:

```
0 VARIABLE H>IN

: Y /SKIP/ C@ IF EXIT ENDIF BEGIN
   >IN 0SET BLK 0SET CR
   ." DEBUG> " QUERY BEGIN
      >IN @ H>IN ! FIND ?DUP IF
         COM-EX
         >IN @ BL WORD C@ SWAP >IN !
      ELSE
         H>IN @ >IN ! BL WORD DUP C@ IF
            DUP >R ?NUMBER IF
               R> DROP
               >IN @ BL WORD C@ SWAP >IN !
            ELSE
               R> TYPE2 ." Eh? " CR 0
            ENDIF
         ELSE
            DROP 0
         ENDIF
      ENDIF
   NOT UNTIL
0 UNTIL ;
```

Appendix A

Stacks for Beginners

The word stack may conjure up visions of a horribly complex entity for the uninitiated. In reality, stacks are among the simplest and most straightforward data structures found in computer science. A stack is a last in, first out (or LIFO) data structure. The most common real-world analogy is a plates dispenser in a cafeteria. Plates are placed in and removed from the top of the container (or stack) that holds them.

The stack is a central feature of Forth, and we can use Forth to learn about stacks. If you type in the word, .S (found in Chapter 13), Forth can help you learn about stacks. .S prints what Forth is holding on its stack. When you type a number in Forth, it places that number on the stack. For example:

```
ATILA OK 23 RETURN
```

If you type a 23 and hit return, Forth will place a 23 on its stack. This is just like writing a 23 on a plate and placing it in the cafeteria dispenser. If a person came and removed the plate from the dispenser, he would find a 23 written on it. In Forth the word . ("dot") does this for us.

```
ATILA OK . [RETURN]

23 ATILA OK
```

Dot removes the top number from the stack and prints it on the display.

Now let's see what happens with more than a single plate, or number. We'll write 67 on a plate and place it in the dispenser. Then we'll deface a plate with a 91 and place it in the dispenser. If someone came along and removed the top plate, what number would he find on it? Let's use Forth to figure it out.

ATILA OK 67 [RETURN]
(Put the plate numbered 67 in the dispenser)

ATILA OK 91 [RETURN]
(Put the plate numbered 91 in the dispenser)

ATILA OK . [RETURN]
(Dot will tell us what the top plate has on it)

91 ATILA OK 91

is on the top plate. What's on the next plate? Let's use Forth again:

ATILA OK . [RETURN]
67 ATILA OK

67 is, of course, the next plate. What if we ask "dot" to look at the next plate?

ATILA OK . [RETURN]
1234 STACK UNDERFLOW

Stack underflow is "dot's" way of telling us there are no more numbers on the stack. .S can be used to look at the stack without changing its contents. Here's how:

ATILA OK 67 91 .S [RETURN]
(Everything on one line!)

91
67
<BOTTOM OF STACK>

ATILA OK . . [RETURN]
(Now print them)

91 67 ATILA OK

This is just how the plates would look also.

Forth includes a large number of words to manipulate words on the stack. DUP (duplicate) is among the most basic. It makes a copy of the top number on the stack, like so:

ATILA OK 4 .S [RETURN]

4

```
<BOTTOM OF STACK>
ATILA OK DUP .S [RETURN]

4
4
<BOTTOM OF STACK>
```

Now we have two fours. Let's print them:

```
ATILA OK . . .S [RETURN]

4  4

<STACK EMPTY>

ATILA OK
```

After we printed what was on the stack, there was nothing left, so .S told us the stack was empty. .S will enable you to play with the stack and learn by getting hands-on experience. Here are some other words you might want to experiment with:

SWAP	Swap the top two numbers on the stack.
DROP	Remove the top number from the stack.
OVER	Move a copy of the second number on the stack to the top.
DEPTH	Leave the number of numbers on the stack on the stack.
ROT	Rotate the top three numbers on the stack.

HAVE FUN !

Appendix B

Extra Atila Words

Any time you enter a word from this book and you find that your version of Forth doesn't know it, this is the place to look. A few of the Atila words can be written in Forth, and these should present you with no problems. Most will also have a corresponding word in your version of Forth. Some, however, really require assembler. For those we'll present the assembler code from MASM, the standard IBM assembler, and how they would be written in the assembler presented in Chapter 5. You will have to refer to the documentation provided with your version of Forth to determine how to implement them.

ENDIF	: ENDIF COMPILE THEN ; IMMEDIATE
<BUILDS	: <BUILDS COMPILE CREATE ;
LROT	: LROT SWAP ROT SWAP ;
HOME	: HOME CLS ;
(Clear the Screen)	
ATILA	: ATILA FORTH ;

VTAB

```
CODE VTAB BH 0 # MOV
   AH 3 # MOV 16 # INT AX POP DH AL MOV
   AH 2 # MOV 16 # INT
NEXT
```

vtab:

```
dw      $+2
mov     bh,0
mov     ah,3
int     010h
pop     ax
mov     dh,al
mov     ah,2
int     010h
jmp     NEXT
```

HTAB

```
CODE HTAB BH 0 # MOV
   AH 3 # MOV 16 # INT AX POP DL AL MOV
   AH 2 # MOV 16 # INR
NEXT
```

htab:

```
dw      $+2
mov     bh,0
mov     ah,3
int     010h
pop     ax
mov     dl,al
mov     ah,2
int     010h
jmp     NEXT
```

Appendix C

Stack Notation

The following stack notation is used in this book:

(Before Execution/After Execution)
(Least Accessible ... Most Accessible/Least Accessible ... Most Accessible)
N/A signed 16–bit number
UN/An unsigned 16–bit number.
D/A signed 32–bit number.
UD/An unsigned 32–bit number.
A/A 16–bit address F/A Boolean Flag.
R/A 32–bit real number.
Multiple instances of the same data type are numbered sequentially.
An example:
–TEXT (A1 N1 A2/N2)
–TEXT expects an address, then an integer, then an address as arguments. It returns an integer result.

INDEX

Boldface indicates a Forth word that is defined in the text.

8087 108-23
8088 41-54, 108-23

Arrays 268
Atila 1, 56, 101, 216
ATN 233-34

CALCULATOR 18
CASE 6
CASE: 6
CODE 54, 101
Color Display 20, 217

Debugging 262, 63

EDIT 39
END-OF 10
END-SUB 54, 99
ENDCASE 54
Expert System 355
Extra Memory 2, 20, 171, 216

GET-INPUT 201

IBM-PC 1, 2, 50, 216
Inference Engine 326

Macintosh 216
MACRO 49, 55, 102
Macros 55
MEND 55, 99
Monochrome Display 20, 217

NEXT 54, 98
NOT-OF 9

=OF 7
>OF 8
<OF 7

Pascal 269

Queues 269, 319
Quicksort 186

Real Numbers 110
Records 269
RNG-OF 8
Robot 233, 266
Rule Compiler 326
Rule Interpreter 326
Rules 325

Screen 23
SORT 185
Stack Notation 375
SUBROUTINE 54, 101

TO Variables 271

Vectored Execution 186

Windows 216

X<CMOVE 2
X! 2
XC! 2
XCE 2
XCMOVE 3
XFILL 3

(y/n) 212

Library of Forth Routines and Utilities

ORDER FORM

TERRY BROTHERS SOFTWARE

CREATOR OF ATILA ... THE FORTH FOR THE IBM PC™

42 Princeton Arms South
Cranbury, New Jersey 08512
(609) 426-0977

THE LIBRARY OF FORTH ROUTINES	UNIT PRICE	X # OF COPIES	= UNIT TOTAL
ATILA IBM SCREEN FORMAT	$19.95		
MS-DOS TEXT FILE FORMAT	$19.95		
ATILA APPLE SCREEN FORMAT	$19.95		
ATILA COMMODORE 64/128 SCREEN FORMAT	$19.95		
ATILA MacINTOSH™ SCREEN FORMAT	$19.95		
ATILA ATARI 520 ST SCREEN FORMAT*	$19.95		
ATILA AMIGA™ SCREEN FORMAT*	$19.95		
ATILA...THE FORTH FOR YOUR COMPUTER!			
IBM PC VERSION (AND 100% COMPATABLES)	$69.95		
APPLE II VERSION	$69.95		
COMMODORE 64/128 VERSION	$69.95		
APPLE MacINTOSH VERSION	$69.95		
ATARI 520 ST VERSION*	$69.95		
AMIGA VERSION*	$69.95		
		SUB-TOTAL	
		NJ RESIDENTS ADD 6% SALES TAX	
		SHIPPING AND HANDLING	$3.00
		TOTAL	

*ATARI AND AMIGA VERSIONS AVALIABLE 7/1/86

CHECK___ MONEY ORDER___ VISA___ MASTERCARD___

CARD #_____ EXP DATE _____

NAME _____

ADDRESS _____

CITY_____ STATE_____ ZIP_____

ON DISK!

The complete source for every word defined in this book is available on diskette: already typed in . . . already debugged!

COMPUTERS • Z5841 • $22.95
CANADA • $31.95

THE READY-TO-USE TIME-SAVING CODE FOR FORTH PROGRAMMERS

The LIBRARY OF FORTH ROUTINES AND UTILITIES is the only comprehensive collection of professional-quality computer code for Forth, the programming language of the future. Full of creative, time-saving routines, this book offers programs that can be put to use in almost any Forth application, including expert systems and natural language interfaces. Among the hundreds of applications included are:

- A unique collection of Forth routines written for the IBM PC or any 8088 computer system
- Hundreds of efficient routines that are easily tailored to applications in expert systems and artificial intelligence
- Completely tested and debugged code ready to splice into any version of Forth
- Assembly language routines for the 8088 and the math co-processor
- A highly flexible mini-database management package
- A sentence parser that can dialogue with the user
- And much more

LIBRARY OF FORTH ROUTINES AND UTILITIES

FULLY ILLUSTRATED

25841
0 51857 02295 0