Chapter 5
Intermediate Forth Tutorial

Note: This chapter is shared between the JForth Manual and the HMSL manual. Information specific to HMSL on the Macintosh or JForth on the Amiga is noted as such.

Editing Programs in Files

As your programs get larger, it becomes impractical to type them in fresh each time. To develop software of any size, you'll need to keep your source code in files. You can then compile directly from these files instead of from the keyboard. Forth programs can be created like programs in other languages by using a text editor. You can use any text editor that you like. Beware of word processing programs because they will put strange information in the file related to margins, fonts, etc. If you use a word processor, save the file in *text only* mode. We recommend that you use the text editor that comes included with your Forth package. The process of editing differs between computers. Please read the appropriate section for your machine

Editing on the Macintosh: To create a new file, select New from the File menu while in Forth. Text can be enterred in the standard Macintosh style. Look in the Edit menu for commands like Cut, Copy and Paste, Upper and Lower case conversion.

Editing on the Amiga: The editor supplied with JForth is called Textra. It was written by Mike Haas using JForth. Full documentation on Textra is supplied in the JTX:DOCS directory. You will probably want to install Textra by either copying it to your C: directory (or any other directory in your current AmigaDOS path) or to a place convienient for double-clicking it's icon. Textra can be found in the JTX: directory on the JTOOLS: disk. Assuming Textra is installed, either double-click on it's icon or enter:

```
RUN TEXTRA
```

If you want to open an existing file, select it using the file selector. To start with a fresh file, hit CANCEL. Look in the Project menu for commands to save files and to create new files.

You can move through the file using cursor keys or the mouse. Drag the mouse (while holding down the left mouse button) to select text. Selected text can be Cut, Copied, or Pasted using commands in the Edit menu.

If you have ARexx, look in Chapter 23, *AREXX Support* for instructions on how to connect JForth and Textra for easy compilation.

Sample Program

Enter into your file, the following code.

```
\ Sample Forth Code
\ Author: your name
: SQUARE  ( n -- n*n , square number )
    DUP *
;
: TEST.SQUARE ( -- )
    CR ." 7 squared = "
    7 SQUARE . CR
;
```

Now save the file as described for your machine.

The text following the \ character is treated as a comment. This would be a REM statement in BASIC or a /*---*/ in 'C'. The text in parentheses is also a comment.

Save this file using the "Save As..." menu option, giving it the name SAMPLE in the process.

INCLUDE the Program

"INCLUDE" in Forth means to compile from a file.

Compiling on the Macintosh: To compile a file directly from the editor, select the "Include From Editor" command from the Include menu. If an error is encountered, it will be highlighted in the editor. Error messages will appear in the Forth window. You can also compile your program from HMSL by selecting "Include from File..." from the File menu. Use the dialog to select the file you just created.

Compiling on the Amiga: If you have ARexx installed, and configured Textra and JForth as described in Chapter 23, you can compile a file from Textra by simply hitting whichever function key is assigned to the jcompile command. If an error is encountered, it will be highlighted in the editor. Error messages will appear in the Forth window.

You can also compile from a file using the INCLUDE command. If you saved your file as WORK:SAMPLE, then compile it by entering:

```
INCLUDE WORK:SAMPLE
```

Forth will compile your file and tell you how many bytes it has added to the dictionary. To test your word, enter:

```
TEST.SQUARE
```

Your two words, SQUARE and TEST.SQUARE are now in the Forth dictionary. We can now do something that is very unusual in a programming language. We can "uncompile" the code by telling Forth to **FORGET** it. Enter:

```
FORGET SQUARE
```

This removes SQUARE and everything that follows it, ie. TEST.SQUARE, from the dictionary. If you now try to execute TEST.SQUARE it won't be found.

Now let's make some changes to our file and reload it. Go back into the editor and make the following changes: (1) Change TEST.SQUARE to use 15 instead of 7 then (2) Add this line right before the definition of SQUARE:

```
ANEW TASK-SAMPLE
```

Now Save your changes and go back to the Forth window.

You're probably wondering what the line starting with **ANEW** was for. ANEW is always used at the beginning of a file. It defines a special marker word in the dictionary before the code. The word typically has "TASK-" as a prefix followed by the name of the file. When you ReInclude a file, ANEW will automatically FORGET the old code starting after the ANEW statement. This allows you to Include a file over and over again without having to manually FORGET the first word. If the code was not forgotten, the dictionary would eventually fill up.

If you have a big project that needs lots of files, you can have a file that will load all the files you need. Sometimes you need some code to be loaded that may already be loaded. The word **INCLUDE?** will only load code if it isn't already in the dictionary. In this next example, I assume the file is on the volume WORK: and called SAMPLE. If not, please substitute the actual name. Enter:

```
FORGET TASK-SAMPLE
INCLUDE? SQUARE WORK:SAMPLE
INCLUDE? SQUARE WORK:SAMPLE
```

Only the first INCLUDE? will result in the file being loaded.

Variables

Forth does not rely as heavily on the use of variables as other compiled languages. This is because values normally reside on the stack. There are situations, of course, where variables are required. To create a variable, use the word **VARIABLE** as follows:

```
VARIABLE MY-VAR
```

This created a variable named MY-VAR . A space in memory is now reserved to hold its 32-bit value. The word VARIABLE is what's known as a "defining word" since it creates new words in the dictionary. Now enter:

```
MY-VAR .
```

The number you see is the address, or location, of the memory that was reserved for MY-VAR. To store data into memory you use the word **!** , pronounced "store". It looks like an exclamation point, but to a Forth programmer it is the way to write 32-bit data to memory. To read the value contained in memory at a given address, use the Forth word **@** , pronounced "fetch". Try entering the following:

```
513 MY-VAR !
MY-VAR @ .
```

This sets the variable MY-VAR to 513 , then reads the value back and prints it. The stack diagrams for these words follows:

```
@  ( address -- value , FETCH value FROM address in memory )

!  ( value address -- , STORE value TO address in memory )

VARIABLE  ( <name> -- , define a 4 byte memory storage location)
```

A handy word for checking the value of a variable is **?** , pronounced "question". Try entering:

```
MY-VAR ?
```

If ? wasn't defined, we could define it as:

```
: ? ( address -- , look at variable )
    @ .
;
```

Imagine you are writing a game and you want to keep track of the highest score. You could keep the highest score in a variable. When you reported a new score, you could check it aginst the highest score. Try entering this code in a file as described in the previous section:

```
VARIABLE HIGH-SCORE
: REPORT.SCORE  ( score -- , print out score )
    DUP CR ." Your Score = "   .   CR
    HIGH-SCORE @ MAX   ( calculate new high )
    DUP ." Highest Score = "   .   CR
    HIGH-SCORE !       ( update variable )
;
```

Save the file to disk, then compile this code using the INCLUDE word. Test your word as follows:

```
123 REPORT.SCORE
9845 REPORT.SCORE
534 REPORT.SCORE
```

The Forth words @ and ! work on 32-bit quantities. Some Forths are "16-bit" Forths. They fetch and store 16-bit quantities. Forth has some words that will work on 8 and 16-bit values. C@ and C! work for 8-bit bytes. The 'C' stands for "Character" since ASCII characters are 8-bit numbers. Use W@ and

W! for 16-bit "Words."

Another useful word is **+!** , pronounced "plus store." It adds a value to a 32-bit value in memory. Try:

```
20 MY-VAR !
5 MY-VAR +!
MY-VAR @ .
```

Forth also provides some other words that are similar to VARIABLE. Look in the glossary for VALUE and ARRAY. Also look at the section on "local variables" which are variables which only exist on the stack while a Forth word is executing.

A word of warning about fetching and storing to memory: You have now learned enough about Forth to be dangerous. The operation of a computer is based on having the right numbers in the right place in memory. You now know how to write new numbers to any place in memory. Since an address is just a number, you could, but shouldn't, enter:

```
73 253000 ! ( Do NOT do this. )
```

The 253000 would be treated as an address and you would set that memory location to 73. I have no idea what will happen after that, maybe nothing. This would be like firing a rifle through the walls of your apartment building. You don't know who or what you are going to hit. Since you share memory with other programs including the operating system, you could easily cause the computer to behave strangely, even crash. Don't let this bother you too much, however. Crashing a computer, unlike crashing a car, does not hurt the computer. You just have to reboot. The worst that could happen is that if you crash while the computer is writing to a disk, you could lose a file. That's why we make backups. This same potential problem exists in any powerful language, not just Forth. This might be less likely in BASIC, however, because BASIC protects you from a lot of things, including the danger of writing powerful programs.

Another way to get into trouble is to do what's called an "odd address memory access." The 68000 processor arranges words and longwords, 16 and 32 bit numbers, on even addresses. If you do a **@** or **!** , or **W@** or **W!** , to an odd address, the 68000 processor will take exception to this and try to abort.

Forth gives you some protection from this by trapping this exception and returning you to the OK prompt. If you really need to access data on an odd address, check out the words **ODD@** and **ODD!** in the glossary. **C@** and **C!** work fine on both odd and even addresses.

("Odd address errors" are not possible on Amiga's that use 68020 CPU's and later...they are made to perform this type of operation. However, if you have one of these newer CPU's, you should always be mindful of all the Amiga's which DO use 68000's and avoid doing such operations.)

The reason I am spending so much time on this is that if your program is crashing and getting those little pictures of bombs, you may likely be storing somewhere you shouldn't be.

Constants

If you have a number that is appearing often in your program, we recommend that you define it as a "constant." Enter:

```
128 CONSTANT MAX_CHARS
MAX_CHARS .
```

We just defined a word called MAX_CHARS that returns the value on the stack when it was defined. It cannot be changed unless you edit the program and recompile. Using **CONSTANT** can improve the readability of your programs and reduce some bugs. Imagine if you refer to the number 128 very often in your program, say 8 times. Then you decide to change this number to 256. If you globally change 128 to 256 you might change something you didn't intend to. If you change it by hand you might miss one, especially if your program occupies more than one file. Using CONSTANT will make it easy to change. The code that results is equally as fast and small as putting the numbers in directly. I recommend defining a constant for almost any number.

Logical Operators

These next two sections are concerned with decision making. This first section deals with answering questions like "Is this value too large?" or "Does the guess match the answer?". The answers to questions like these are either TRUE or FALSE. Forth uses a 0 to represent **FALSE** and a -1 to represent **TRUE**. TRUE and FALSE have been capitalized because they have been defined as Forth constants. Try entering:

```
23 71 = .
18 18 = .
```

You will notice that the first line printed a 0, or FALSE, and the second line a -1, or TRUE. The equal sign in Forth is used as a question, not a statement. It asks whether the top two items on the stack are equal. It does not set them equal. There are other questions that you can ask. Enter:

```
23 198 < .
23 198 > .
254 15 > .
```

In California, the drinking age for alcohol is 21. You could write a simple word now to help bartenders. Enter:

```
: DRINK? ( age -- flag , can this person drink? )
    20 >
;
20 DRINK? .
21 DRINK? .
43 DRINK? .
```

The word FLAG in the stack diagram above refers to a logical value.

Forth provides special words for comparing a number to 0. They are **0= 0>** and **0<** . Using 0> is faster than calling 0 and > separately. Enter:

```
23 0> .  ( print -1 )
-23 0> . ( print 0 )
23 0= .  ( print 0 )
```

For more complex decisions, you can use the *Boolean* operators **OR** , **AND** , and **NOT** . OR returns a TRUE if either one or both of the top two stack items are true.

```
TRUE TRUE OR .
TRUE FALSE OR .
FALSE FALSE OR .
```

AND only returns a TRUE if both of them are true.

```
TRUE TRUE AND .
TRUE FALSE AND .
```

NOT reverses the value of the flag on the stack. Enter:

```
TRUE .
TRUE NOT .
```

Logical operators can be combined.

```
56 3 >    56 123 < AND .
23 45 =   23 23 = OR .
```

Here are stack diagrams for some of these words. See the glossary for a more complete list.

```
< (  a b -- flag , flag is true if A is less than B )
> (  a b -- flag , flag is true if A is greater than B )
= (  a b -- flag , flag is true if A is equal to B )
0=  ( a -- flag , true if a equals zero )
OR  ( a b -- a||b , perform logical OR of bits in A and B )
AND  ( a b -- a&b , perform logical AND of bits in A and B )
NOT  ( flag -- opposite-flag , true if false, false if true )
```

Problem:

1) Write a word called LOWERCASE? that returns TRUE if the number on top of the stack is an ASCII lowercase character. An ASCII 'a' is 97 . An ASCII 'z' is 122 . Test using the characters " A a q z { ".

```
ASCII A LOWERCASE? .    ( should print 0 )
ASCII a LOWERCASE? .    ( should print -1 )
```

Flow of Control

You will now use the TRUE and FALSE flags you learned to generate in the last section. The "flow of control" words accept flags from the stack, and then possibly "branch" depending on the value. Enter the following code.

```
: .L ( flag -- , print logical value )
    IF ." True value on stack!"
    ELSE ." False value on stack!"
    THEN
;
0 .L
FALSE .L
TRUE .L
23 7 < .L
```

You can see that when a TRUE was on the stack, the first part got executed. If a FALSE was on the stack, then the first part was skipped, and the second part was executed. One thing you will find interesting is that if you enter:

```
23 .L
```

the value on the stack will be treated as true. The flow of control words consider any value that does not equal zero to be TRUE.

The **ELSE** word is optional in the **IF...THEN** construct. Try the following:

```
: BIGBUCKS? ( ammount -- )
    1000 >
    IF ." That's TOO expensive!"
    THEN
;
531 BIGBUCKS?
1021 BIGBUCKS?
```

Forth also has a **CASE** statement similar to switch() in 'C'. Enter:

```
: TESTCASE ( N -- , respond appropriately )
    CASE
        0 OF  ." Just a zero!" ENDOF
        1 OF  ." All is ONE!"  ENDOF
        2 OF  WORDS            ENDOF
        DUP . ." Invalid Input!"
    ENDCASE CR
;
0 TESTCASE
1 TESTCASE
5 TESTCASE
```

See CASE in the glossary for more information.

Problems:

1) Write a word called DEDUCT that subtracts a value from a variable containing your checking account balance. Assume the balance is in dollars. Print the balance. Print a warning if the balance is negative.

```
VARIABLE  ACCOUNT
: DEDUCT ( n -- , subtract N from balance )
   ??????????????????????????????????  ( you fill  this in )
;
300 ACCOUNT !  ( initial funds )
40 DEDUCT  ( prints 260 )
200 DEDUCT ( print 60 )
100 DEDUCT  ( print -40 and give warning! )
```

Loops

Another useful pair of words is **BEGIN...UNTIL** . These are used to loop until a given condition is true. Try this:

```
: YAKYAK ( -- )
    BEGIN
        ." I could talk all day!" CR
        ?TERMINAL
    UNTIL
;
YAKYAK
```

This word will keep running until you hit a key on the keyboard. The word **?TERMINAL** will return TRUE if a key has been hit. (You could then immediately call KEY if you wish to know what key was hit.)

If you know how many times you want a loop to execute, you can use the **DO...LOOP** construct. Enter:

```
: SPELL
    ." ba"
    4 0 DO
        ." na"
    LOOP
```

```
;
```

This will print "ba" followed by four occurrences of "na". The ending value is placed on the stack before the beginning value. Be careful that you don't pass the values in reverse. Forth will go "the long way around" which could take awhile. The reason for this order is to make it easier to pass the loop count into a word on the stack. Consider the following word for doing character graphics. Enter:

```
: PLOT# ( n -- )
    0 DO
        ASCII -   EMIT
    LOOP CR
;
CR  9 PLOT#  37 PLOT#
```

If you want to access the loop counter you can use the word I . Here is a simple word that dumps numbers and their associated ASCII characters.

```
: .CHARS ( end start -- , dump characters )
    DO
        CR I . I EMIT
    LOOP CR
;
80 64 .CHARS
```

You could make this word safe by making sure that the parameters are in the right order. The word **-2SORT** will take two numbers and leave the smallest on top. Enter:

```
: SAFE.CHARS ( n1 n2 -- )
    -2SORT  ( put smallest on top )
    .CHARS
;
40 50 SAFE.CHARS
50 40 SAFE.CHARS
```

If you want to leave a DO LOOP before it finishes, you can use the word **LEAVE**. Enter:

```
: ANNOY.ME ( -- , Sing! )
    50000 0
    DO  50000 I - .
        ." bottles of beer on the wall." CR
        ?TERMINAL ( was a key hit )
        IF  ." OK, I'll shut up!" CR
            LEAVE  ( quit looping )
        THEN
    LOOP
;
```

Please consult the manual to learn about the following words **+LOOP** and **RETURN** .

Another useful looping construct is the **BEGIN WHILE REPEAT** loop. This allows you to make a test each time through the loop before you actually do something. The word WHILE will continue looping if the flag on the stack is True. Suppose you want to write a word called SHOWCHARS that prompts the user to hit a key, waits for a key to be hit, then prints the key as a decimal number and as a character. It should keep doing this in a loop until the key is a 'q' or a 'Q'. Here is a word that will do this:

```
: SHOWCHARS  ( -- , loop on chars till 'q' )
   BEGIN
      ." Hit a key or 'q' to quit:" KEY   ( wait for a key )
```

```
      DUP ASCII q =  ( -- char flag )
      OVER ASCII Q = OR NOT
   WHILE  ( -- char , loop if not 'q' or 'Q' )
      CR ." Key = " DUP . EMIT CR
   REPEAT DROP
;
```

Text I/O

You learned earlier how to do single character I/O. This section concentrates on using strings of characters. A text string in Forth consists of a character count in the first byte, followed immediately by the characters themselves. A character string can be created using the Forth word " , pronounced 'quote'. Note that you must follow the " by one space. Any text following that space is copied to a special area in memory called the **PAD**. The text string is terminated by an ending " . Enter:

```
" Fred" .
```

The number that was printed was the address of the start of the string. It should be a byte that contains the number of characters. Now enter:

```
" Fred" C@ .
```

You should see a 4 printed. Remember that C@ fetches one character/byte at the address on the stack. By adding one to this address and doing a C@, you should see the first character. Enter:

```
" Fred" 1+ DUP C@ EMIT
1+ DUP C@ EMIT
1+ C@ EMIT
```

Using this method, you could type out any string. Luckily, there are two words that make this process much easier. Enter:

```
" Hello" COUNT .S
TYPE
```

The word **COUNT** extracts the number of characters and their starting address. **TYPE** accepts the output of count and prints those characters. COUNT will only work with strings of less than 256 characters, since 255 is the largest number that can be stored in the count byte. TYPE will, however, work with longer strings since the length is on the stack. Their stack diagrams follow:

```
COUNT  ( $addr -- addr #bytes , extract string information )

TYPE   ( addr #bytes -- , output characters at addr )
```

The $addr is the address of a count byte. The dollar sign is often used to mark words that relate to strings.

You can easily input a string using the word **EXPECT**. (You may want to put these upcoming examples in a file since they are very handy.) The word EXPECT receives characters from the keyboard and places them at any specified address. EXPECT takes input characters until a maximum is reached or a carriage return is entered. The user variable SPAN contains the number of characters entered. You can write a word for entering text. Enter:

```
: INPUT$ ( -- $addr )
    PAD 1+  ( leave room for byte count )
    128 EXPECT  ( recieve a maximum of 128 chars )
    SPAN @   PAD C!  ( set byte count )
    PAD    ( return address of string )
;
```

You could use this in a program that writes form letters.

```
: FORM.LETTER ( -- )
    ." Enter customer's name." CR
    INPUT$
    CR ." Dear " DUP COUNT TYPE CR
    ." Your cup that says " COUNT TYPE
    ."  is in the mail!" CR
;
```

EXPECT (addr maxbytes -- , input text, save at address)

SPAN (-- addr , contains number of chars EXPECT got)

You can use your word INPUT$ to write a word that will read a number from the keyboard. Enter:

```
: INPUT# ( -- N true | false )
    INPUT$  ( get string )
    NUMBER? ( convert to a string if valid )
    IF DROP TRUE  ( get rid of high cell )
    ELSE FALSE
    THEN
;
```

This word will return a single-precision number and a TRUE, or it will just return FALSE. The word **NUMBER?** returns a double precision number if the input string contains a valid number. Double precision numbers are 64-bit so we DROP the top 32 bits to get a single-precision 32 bit number.

Changing Numeric Base

Our numbering system is decimal, or "base 10." This means that a number like 527 is equal to (5*100 + 2*10 + 7*1). The use of 10 for the numeric base is a completely arbitrary decision. It no doubt has something to do with the fact that most people have 10 fingers (including thumbs). The Babylonians used base 60, which is where we got saddled with the concept of 60 minutes in an hour. Computer hardware uses base 2, or "binary". A computer number like 1101 is equal to (1*8 + 1*4 + 0*2 + 1*1). If you add these up, you get 8+4+1=13 . A 10 in binary is (1*2 + 0*1), or 2. Likewise 10 in any base N is N .

Forth makes it very easy to explore different numeric bases because it can work in any base. Try entering the following:

```
DECIMAL  6   BINARY .
1 1 + .
1101 DECIMAL .
```

Another useful numeric base is *hexadecimal*. which is base 16. One problem with bases over 10 is that our normal numbering system only has digits 0 to 9. For hex numbers we use the letters A to F for the digits 10 to 15. Thus the hex number 3E7 is equal to (3*256 + 14*16 + 7*1). Try entering:

```
DECIMAL
12 .HEX
12 256 *    7 16 * +  10 + .S
DUP BINARY .   .HEX
```

A variable called **BASE** is used to keep track of the current numeric base. The words HEX , **DECIMAL** , and **BINARY** work by changing this variable. You can change the base to anything you want. Try:

```
DECIMAL
7 BASE !
```

```
6 1 + .
BASE @ .
```

You are now in base 7 . When you fetched and printed the value of BASE, it said 10 because 7, in base 7, is 10.

You could define a word like .HEX for any base. What is needed is a way to temporarily set the base while a number is printed, then restore it when we are through. Try the following word:

```
: .BINARY ( N -- , print N in Binary )
    BASE @     ( save current base )
    2 BASE !   ( set to binary )
    SWAP .     ( print number )
    BASE !     ( restore base )
;
DECIMAL
22 .BINARY
22 .
```

Answers to Problems

Logical Operators

```
: LOWERCASE? ( CHAR -- FLAG , true if lowercase )
    DUP 123 <
    SWAP 96 > AND
;
```

Flow of Control

```
: DEDUCT  ( n -- , subtract from account )
   ACCOUNT @   ( -- n acc )
   SWAP - DUP ACCOUNT !  ( -- acc' , update variable )
   ." Balance = $" DUP . CR  ( -- acc' )
   0<  ( are we broke? )
   IF    ." Warning!! Your account is overdrawn!" CR
   THEN
;
```