Chapter 6
Advanced Forth Tutorial

String Handling

Since we will be doing a lot of string output, let's define a very handy word.

```
: $. ( $addr -- , print string )
    COUNT TYPE
;
" Hello" $.
```

As you were experimenting with text in the last lesson, you may have noticed that your string on the PAD got overwritten. This is because the PAD is a popular place to put text in FORTH. If you want to keep your string intact, you will need to copy it to a safe place. To make room for a string you can use two new words called CREATE and ALLOT. Enter:

```
CREATE MY-STRING   130 ALLOT
```

CREATE defines a new word similar to the way VARIABLE does. ALLOT will make room for 130 bytes (characters) after MY-STRING. We will discuss these words in detail later. You can move a string from one location to another using $MOVE. We will use our INPUT$ word from the previous tutorial. Enter:

```
INPUT$     ( hit return then enter a string )
MY-STRING $MOVE
```

The string you entered is now stored at MY-STRING. The PAD can now be used by other words, like " , without destroying your word. Just to prove it, enter:

```
PAD $.
" Smash" $.
PAD $.
MY-STRING $.
```

Comparing two strings is a common task. Enter:

```
" FROG" MY-STRING $MOVE
" TREE" MY-STRING $= .
" FROG" MY-STRING $= .
```

The word $= is useful to check for string matches. This might find application in a security system. Enter:

```
: PASSWORD ( -- flag, check password )
    CR ." Enter password: " INPUT$
    MY-STRING $=  DUP NOT ( test )
    IF ." Invalid password!"
    THEN
;
PASSWORD .
```

If you want to alphabetize strings, you also need to know which string is "higher". Enter:

```
" AARDVARK" MY-STRING $- .
" FROG" MY-STRING     $- .
" ZEBRA" MY-STRING    $- .
" frog" MY-STRING     $- .
```

$- compares two strings. It is sensitive to upper and lower case. Use TEXT=? if you want to ignore case.

If you want to append one string onto another you can do so using $APPEND . Enter:

```
" tree" MY-STRING $MOVE
"  frog" COUNT MY-STRING $APPEND
MY-STRING $.
```

The following example demonstrates Text, DO LOOPs, and LEAVE . It is a handy word that searches for a byte, or character, and tells you its offset in a string.

```
VARIABLE BOFFSET
: SEARCHBYTE ( byte addr count -- offset | -1 )
    -1 BOFFSET !  ( set default answer )
    0
    DO  DUP I + C@  ( get byte )
        ( -- byte addr byte , stack looks like this )
        2 PICK =    ( matches? )
        IF I BOFFSET ! LEAVE  ( save offset )
        THEN
    LOOP    2DROP
    BOFFSET @  ( get result )
;
ASCII t " Look a tree!" COUNT .S
SEARCHBYTE .
```

Saving Forth

You may find that you are always INCLUDEing a certain set of files when you use JForth. Rather than INCLUDE them each time, you can INCLUDE them once and save the compiled result in another file. Insert your JForth disk and enter:

```
DOS EXECUTE JFORTH:ASSIGNS
INCLUDE JU:SQRT
49 SQRT .
```

Now put in your formatted JWORK: disk from the previous tutorial and enter:

```
SAVE-FORTH  JWORK:SQRT4TH   ( save entire Forth in file)
BYE     ( this leaves JForth )
RUN JWORK:SQRT4TH    ( enter in CLI window )
( wait for it to load )
49 SQRT . ( SQRT is now available immediately )
```

When you SAVE-FORTH , you also have the option of expanding the dictionary space available for use. When JForth boots, it looks at a VARIABLE called #K that tells it how much space to allocate. To allocate 20K more dictionary, try.

```
MAP  ( see how much is there now )
20 #K +!
SAVE-FORTH JWORK:BIGGER4TH  ( may need blank disk )
BYE
RUN JWORK:BIGGER4TH  ( from CLI as before )
```

```
MAP  ( there should be more room )
```

You may want to prepare a Forth image with your favorite tools loaded. You can also expand the amount of USER variable space available by using #U in the same way. See #U in the glossary for more information.

Programming Aids

Sometimes a program doesn't behave as expected. This, by definition, is a 'bug'. Luckily, with Forth you have more debugging tools at your disposal than with almost any other language. These tools can be embedded right in the language so they are always available. You can examine any variable or constant. You can also execute most low level words by themselves for incremental testing. We have included, in JForth, some of the tools that we have found handy.

We have already used WORDS. This causes all of the Forth words to be printed out to the screen. You can stop output by hitting a space bar. Once it is stopped, a line of Forth can be entered. The word listing will continue after that line finishes. Another word that is handy is WORDS-LIKE. If you can't remember the exact name of a word but think it had a + in it, you could enter:

```
WORDS-LIKE +
```

also try:

```
WORDS-LIKE EMIT
WORDS-LIKE $
```

Once you have found a word, you might want to see what the code looks like for it. We have a special word called FILE? that tells you what file any word was loaded from. It then asks you if you want to see the source code. As long as the source is not in the assembly language kernel, you will be able to see it. Enter:

```
FILE? <FASTEMIT>
```

Try loading in some of your code and then use FILE? on your words. FILE? works by scanning the file for occurrences of the name. When it finds one, it prints lines until it finds a blank line. This has the result of also showing you examples of how the word is called if it is referenced in that file.

If you are curious about how a word actually works, you can disassemble it using DEF. To fully understand what you are seeing, you should study the chapter on assembly language, and the chapter on the disassembler. Try entering:

```
: ADD3 3 + ;
DEF ADD3
: EMITA ASCII A EMIT ;
DEF EMITA
```

(A quick note to Forth/68000 experts. TOS is D7, which is our cache for the Top Of Stack. DSP is A6, which is our Parameter or Data Stack Pointer.)

The Forth Interpreter and Dictionary

This section will reveal a little of what makes Forth such a unique and powerful language. All Forths keep a list of their defined words in something called a "dictionary". This dictionary is what you see when you call WORDS . When you hit a carriage return, a program called the Forth "interpreter" reads each word on that line and looks it up in the dictionary. If it finds the word, it executes the code that is associated with it. If it doesn't find it, it flashes the screen, then reprints the word followed by a '?'.

You can perform this sequence of operations yourself. Enter:

```
: HI ." HELLO" CR ;
' HI .S
```

```
EXECUTE
```

The apostrophe is a Forth word pronounced "TICK". It takes the word that follows it and looks it up in the dictionary. If it finds the word, it leaves the address of that word's associated code on the stack. That address is referred to as a "CFA", pronounced "c f a". CFA stands for Code Field Address. The word EXECUTE takes that address and executes it.

If you attempt to tick a word that doesn't exist, you will get an error message. This is a handy way of finding out whether a word is defined.

```
' GADCZXW   \ gives error message
```

You can convert the address of the code, CFA, to the address of the name of a word, the NFA, using >NAME. This is pronounced "to name".

Enter:

```
' HI
>NAME ID.
```

ID. takes the address of the word's name and types the name out. You can't use COUNT and TYPE because some extras bits are set in the count byte for internal use.

The behavior of the word ' varies from one dialect of Forth to another. One of the few ways in which JForth deviates from the Forth 83 standard is illustrated as follows:

```
: SAYHI ' HI EXECUTE ;
SAYHI
```

In most Forths written before Forth 83, and JForth, the ' in SAYHI will compile the CFA of HI . In a true Forth 83 system, the ' would not run until you execute SAYHI. SAYHI would then be followed by the word you want to "TICK". See the MULTI-STANDARDS file for information on making JForth compatible with Forth 83.

If you do want to write a word that "TICKS" another word, you must use [COMPILE] . Try entering:

```
: DUMPSWAP  ( -- , just dump SWAP )
    ' SWAP
    >NAME 40 DUMP
;
DUMPSWAP
( This next word is more useful )
: DUMPWORD  ( <name> -- , dump 40 bytes of word )
    [COMPILE] '
    >NAME 40 DUMP
;
DUMPWORD SWAP
DUMPWORD HI
```

Every word in the Forth dictionary is linked to the previous word by its link field. We can find out what is before HI by converting its NAME field to its LINK field and seeing where it points. Enter:

```
' HI   >NAME   .S   ( get NFA )
N>LINK  .S   ( get Link Field Address LFA )
@  ID.   ( get NFA that it points to and print it. )
```

You may want to put these next words in a file. They illustrate techniques for threading through the dictionary. BACKWORDS can be used for listing words defined just prior to another word in the dictionary. Enter:

```
: LOOKBACK ( nfa1 -- nfa2 , show word, link )
     DUP ID. CR
```

```
         N>LINK @  ( LFA points to previous NFA)
    ;
    : BACKWORDS  ( N <word> -- , show N previous words )
        20 MIN   ( do a maximum of 20 )
        [COMPILE] '  ( get CFA )
        >NAME   SWAP 0   ( convert to NFA )
        DO  LOOKBACK
            DUP 0=       ( check for end)
            IF LEAVE THEN
        LOOP  DROP
    ;
```

Include this code and test by entering:

```
    10 BACKWORDS DUP
```

If you are interested in experimenting more, look at the definitions for NAME> , DEFER , INLINE and MAX-INLINE . Also look at the chapter on JForth Internals and Memory Organization.

Return Stack

Consider the following two words:

```
    : BIRD  ." BIRD" ;
    : PROFOUND ." The " BIRD ."  flies!" CR ;
    PROFOUND
```

The word PROFOUND calls BIRD. We hope that when BIRD finishes, the word PROFOUND will continue to completion. Execution must "return" to PROFOUND when BIRD finishes. To do this the processor places the address where execution should resume on the "Return Stack." At the end of BIRD is a 68000 RTS instruction that pops that address off the Return Stack and branches there. The word R@ can be used to examine this stack. Let's use it to look at the return address. Enter:

```
    : RLOOK  R@ . ;
    : TEST RLOOK ." Hello" CR ;
    TEST
    ' TEST 4 + >ABS .
    ' TEST 4 + EXECUTE
    DEF TEST
```

The address you see printed is the address of the code that prints "Hello." This is what was to be executed after RLOOK. The 68000 uses absolute addresses on the return stack thus you had to convert your calculated relative JForth address using >ABS.

The return stack can be also be used for purposes other than storing return addresses. You can temporarily store values there, with care. The only rule is that the return stack must be restored to its original state before the word finishes. Otherwise the 68000 will try to return to the wrong place, with unpredictable results. Enter the following code:

```
    : TESTR  ( N M -- N+1 M )
        >R   ( save M on return stack )
        1+   ( increment N )
        R>   ( get M back from return stack )
    ;
```

The word >R , pronounced "to r", moves the top of the data stack to the return stack. The word R> , pronounced "r from", moves data back from the return stack to the data stack. There must always be an equal number of calls to >R and R> in any given word or that word will return to the wrong place.

Only use R> to get numbers that YOU placed there.

Words for manipulating the return stack are handy for avoiding excessive data stack manipulation and can result in slightly faster code.

The previous example could have been coded using swap, but would have been slower.

```
: TESTS  SWAP 1+ SWAP ;  ( slower than TESTR )
```

Look in the glossary for information on RPICK , R0 , RP@ , RP! , and RDROP .

Extending the Compiler

Words like VARIABLE and CONSTANT can be used to define new words. For this reason they are called "defining words".

Suppose you want to write a new defining word called INTEGER. You can make this data structure behave just like a CONSTANT . Although this will be redundant, it will, hopefully, illustrate the use of two important words, CREATE and DOES> . Enter:

```
: INTEGER    ( value <name> -- )
    CREATE    ,   ( save value in dictionary )
    DOES>     @   ( fetch when executed )
;
173 INTEGER MYINT     ( execute CREATE code )
MYINT .               ( execute DOES> code )
```

A word like INTEGER has two main parts. The first part is the code between CREATE and DOES> . This code executes when the defining word is executed, such as when MYINT is defined in the preceding example. In this case the value on the stack is saved as part of MYINT. The comma takes what is on the stack and saves it in the dictionary.

The second part is the code that between DOES> and ';' . This describes what the newly defined word will do when executed. The address of a data area called the "BODY", or "PFA", is passed to the DOES> code. In the above example, the @ gets the value that was saved in MYINT at definition time.

In this next example, CREATE DOES> is used to describe a new kind of word called a SPEAKER. A SPEAKER is given a phrase to say whenever it is referenced. Enter the following code. (Remember the text in parentheses are comments and need not be entered.)

```
: SPEAKER    ( $string <name> -- )
    CREATE  HERE $MOVE    ( save string in dictionary )
      HERE C@ ALLOT ALIGN  ( adjust DP )
    DOES>   COUNT TYPE CR
;
" Billions and billions!" SPEAKER CARL
" Four score!" SPEAKER ABE  ( execute CREATE code )
CARL      ( execute DOES> code )
ABE
```

The DP that was referred to was the dictionary pointer. It is the address of where new code will be compiled, and should be left at an even location.

For another example of the use of CREATE DOES> , see the file JU:VALUE.

Earlier in the tutorial, we described the Forth interpreter as taking words from the input stream and executing them. This is clearly not the case when a new word is being compiled. When the Forth interpreter finds a colon, it switches to a different state, known as "compile mode". As words are input, instead of being immediately executed, they are compiled into whatever word is being defined. Now you obviously need a way to get out of this state and back to "interpet mode" at the end of a word. To accomplish this, Forth has what are called IMMEDIATE words that are executed immediately, even in compile mode. Semicolon, ';', is one such word. Semicolon finishes the definition of a word and switches the state back to interpret mode.

The state of the interpreter is stored in a VARIABLE called, appropriately enough, STATE. Enter:

```
STATE @ .
```

Since you are in interpret mode, you should see a 0. Let's make our own IMMEDIATE word so that we can spy on STATE in the middle of compiling. Enter:

```
: STATE.PEEK  ( -- , show state )
    ." State = " STATE @ . CR
;
IMMEDIATE ( Makes last word IMMEDIATE. )
STATE.PEEK
: FOO STATE.PEEK  ." Hello." CR ;
```

Forth has two other words which can change STATE. They allow you to switch back temporarily to interpret mode in the middle of a definition. Enter:

```
: BYTE.MASK    ( n -- byte )
    STATE.PEEK
    [ HEX STATE.PEEK ]  FF AND   [ DECIMAL ]
;
```

The left bracket sets STATE to 0 so that the following words are executed immediately. The right bracket sets STATE back to TRUE so that compilation can continue. This can be useful if you want to do a complex calculation at compile time instead of execution time. Compare the following two ways of defining the same word:

```
: DAYS2SECS    ( #days -- #seconds )
    24 * 60 * 60 *
;
DEF DAYS2SECS
( or )
: DAYS2SECS    ( #days -- #seconds )
    [ 24 60 * 60 * ] LITERAL ( calculate factor )
    *   ( only one multiply at compile time )
;
DEF DAYS2SECS
```

In the second case, the calculation of a single multiplication factor is done at compile time. The word LITERAL is used to compile that value into the dictionary as a constant. The use of this technique can result in faster code. The second word is equivalent to:

```
: DAYS2SECS    86400 * ;
```

If IMMEDIATE words always execute immediately whenever referenced, then how can one compile a call to an IMMEDIATE word? You can do this with [COMPILE] which you saw used with ' earlier. [COMPILE] will cause the next word to be compiled instead of executed. Suppose you wanted to write a word that had LITERAL compiled within it. Enter:

```
VARIABLE MCOUNT
10 MCOUNT !
: NEXTNUM  ( -- , Compile number and increment. )
    MCOUNT @ DUP ." MCOUNT = " . CR
    [COMPILE] LITERAL   ( don't execute LITERAL yet)
    1 MCOUNT +!
;
```

```
IMMEDIATE
: FOO  NEXTNUM + ;   ( 10 + ) ( execute LITERAL now )
: GOO  NEXTNUM + ;   ( 11 + )
5 FOO .
5 GOO .
5 FOO .
```

An associated word is COMPILE. Suppose we wanted to make a compiling word similar to the one above. Enter:

```
: +MC ( -- , Compile number and add )
    [COMPILE] NEXTNUM  ( it is immediate )
    COMPILE + ( compile a call to + when +MC is used)
;
IMMEDIATE
: ZOO +MC ;
5 ZOO .
5 FOO .
```

These words are difficult to explain. The best way to get a feel for them is to turn on TRAPS and start experimenting.

These techniques can be used for making new "flow of control" words. The words IF , THEN , BEGIN , UNTIL , etc., are all IMMEDIATE words. They must set up complex branches at compile time. Suppose you want to make a new conditional construct that only stops looping if a key is hit. Enter:

```
: UNTIL-KEY
    COMPILE ?TERMINAL
    [COMPILE] UNTIL
;
IMMEDIATE
: TESTIT
    BEGIN
        ." BLAH BLAH" CR
    UNTIL-KEY
;
TESTIT
```

We are really delving into the guts of Forth with these words. Don't feel bad if this stuff confuses you. It confuses me. Luckily these techniques are only needed for extending the language. Useful and productive lives can be led without them.

Further Exploration

There are a number of useful words that are part of standard Forth that were not covered in this tutorial. Look these up in the glossary and experiment with them on your own. You should now have the necessary skills to do so. Some examples are:

```
DEPTH    CMOVE   FILL    */    U<
U.       FIND    TIB     .R
```

I would also recommend reading the sections on Local Variables, Vocabularies, and Floating Point. When you want to get into the Amiga internals, there is a chapter on Accessing the Amiga. You will probably want to read the section on the demo programs. Print these programs and study them. They are your gateway to the Amiga's special capabilities.