Chapter 4
Forth Tools

Local Variables

In a complicated Forth word it is sometimes hard to keep track of where things are on the stack. If you find you are doing a lot of stack operations like **DUP SWAP ROT PICK** etc. then you may want to use *local variables*. They can greatly simplify your code.

You can declare local variables for a word using a syntax similar to the stack diagram. These variables will only be accessible within that word. Thus they are "local" as opposed to "global" like regular variables. Local variables are *self-fetching*. They automatically put their values on the stack when you give their name. You don't need to @ the contents. Local variables do not take up space in the dictionary. They reside on the return stack where space is made for them as needed. Words written with them can be reentrant and recursive. You can declare how large local variables are to allow double precision, or larger.

Consider a word that calculates the difference of two squares, Here are two ways of writing the same word.

```
: DIFF.SQUARES ( A B -- A*A-B*B )
    DUP *
    SWAP DUP *
    SWAP -
;
  ( or )
: DIFF.SQUARES { A B -- A*A-B*B }
    A A *
    B B * -
;
3 2 DIFF.SQUARES  ( would return 5 )
```

In the second definition of DIFF.SQUARES the curly bracket '{' told the compiler to start declaring local variables. Two locals were defined, A and B. The names could be as long as regular Forth words if desired. The "--" marked the end of the local variable list. When the word is executed, the values will automatically be pulled from the stack and placed in the local variables. When a local variable is executed it places its value on the stack instead of its address. This is called *self-fetching*. Since there is no address, you may wonder how you can store into a local variable. There is a special operator for local variables that does a store. It looks like -> and is pronounced "to".

Local variables need not be passed on the stack. You can declare a local variable that is uninitialized by placing it after a "vertical bar" (|)character. Uninitialized are exactly that. They are not automatically set to zero when created and may have any possible value before being set. Here is a simple example that uses -> and | in a word:

```
: SHOW2*
   { loc1 | unvar --  , 1 regular, 1 uninitialized }
   LOC1  2*  ->  UNVAR
      (set uninitialized local to 2*LOC1 )
   UNVAR   .   ( print UNVAR )
;
3 SHOW2*   ( pass only 1 parameter, prints 6 )
```

Since local variable often used as counters or accumulators, we have a special operator for adding to a local variable It is +-> which is pronounced "plus to". These next two lines are functionally equivalent but the second line is faster and smaller:

```
ACCUM   10 +   -> ACCUM
```

```
10 +-> ACCUM
```

If you don't specify the size of the local variables, they default to 1 cell. You can specify the size in cells by preceding the variable name with a number.

```
: EXAMPLE { 2 a 2 b 4 c -- d1 }
    a b  D+
    c ( note this fetches 2 double numbers )
    D+ D+ ;
5. 4. 6. 9. example  ( will return 24. )
```

Local variables are normally self fetching, but that can be turned on and off using NO@ and YES@. When NO@ is specified, a reference to a local variable will result in its address being placed on the stack.

```
: EXAMPLE { a b c -- sum+1 }
    a b + c +  -> c
    no@ ( turn self fetching off )
    1 c +!
    yes@ ( turn it back on )
    c  ( fetch C ) ;

1 2 3 example ( will return 7 )
```

By combining NO@ and multiple cell variables, you can allocate local arrays

```
: LOCARR { indx | 100 larr -- }
   NO@ LARR YES@ \ get address of base of array
   INDX CELLS \ calculate offset into array
   +  @  \ calculate address in array and fetch value there
;
```

You can also specify that some local variables will automatically be returned:

```
: EXAMPLE { a b c --> c }
    a b + c + -> c
;
1 2 3 example  ( will return 6 )
```

The --> means that what follows up to the ASCII } is a list of variables to return.

If you name a local variable the same as a Forth word in the dictionary, eg. INDEX or COUNT, you will be given a warning message. The local variable will still work but one could easily get confused so we warn you about this. Other errors that can occur include, missing a closing '}', missing '--', or having too many local variables.

Command Line History

You can use the *cursor keys* to edit the command line as you enter them. You can also scroll through and edit previous commands to avoid retyping. Enter:

```
23  45  +  .
```

Now hit the <UP ARROW> then <RETURN> to reexecute that command. Now hit the <UP ARROW> then the <LEFT-ARROW>, then change one of the numbers, then >RETURN>. This can save you a lot of typing.

(This next feature does not work on a Mac Plus. Sorry.) If you want to go back to a certain line, type the first few characters of that line, then hold down the SHIFT key while you hit <UP-ARROW>. This is called *command line completion* . This will search for a matching line. If you match a line you didn't want, keep hitting SHIFT+<UP-ARROW> to continue the search.

Using DEFER for vectored words.

In Forth and other languages you can save the address of a function in a variable. You can later fetch from that variable and execute the function it points to. This is called *vectored execution*. HForth provides a tool that simplifies this process. You can define a word using **DEFER**. This word will contain the address of another Forth function. The executable address of a function is called the "CFA" which stands for "Code Field Address". When you execute the deferred word, it will execute the function it points to. By changing the contents of this deferred word, you can change what it will do. There are several words that support this process.

DEFER (<name> -- , define a deferred word)

IS (CFA <name> -- , set the function for a deferred word)

WHAT'S (<name> -- CFA , return the CFA set by IS)

Simple way to see the name of what's in a deffered word:

```
WHAT'S TYPE  >NAME ID.
```

should print name of current word that's in TYPE.

Here is an example that uses a deferred word.

```
DEFER PRINTIT
'C . IS PRINTIT   ( make PRINTIT use . )
8 3 + PRINTIT

: COUNTUP  ( -- , call deferred word )
   ." Hit RETURN to stop!" CR
   0 ( first value )
   BEGIN 1+ DUP PRINTIT CR
      ?TERMINAL
   UNTIL
;
COUNTUP  ( uses simple . )

: FANCY.PRINT  ( N -- , print in DECIMAL and HEX)
   DUP ." DECIMAL = " .
   ." , HEX = " .HEX
;
'C FANCY.PRINT  IS PRINTIT  ( change printit )
WHAT'S PRINTIT >NAME ID. ( shows use of WHAT'S )
8 3 + PRINTIT
COUNTUP  ( notice that it now uses FANCY.PRINT )
```

Many words in the system have been defined using DEFER which means that we can change how they work without recompiling the entire system. Here is a partial list of those words.

```
KEY  EMIT  ?TERMINAL  TYPE  . NUMBER?  FIND FINDNFA SMUDGE
UNSMUDGE EXPECT :CREATE USERINIT USERTERM $FOPEN_VR ABORT
$INCLUDE_VR
```

Potential Problems with Defer

Deferred words are very handy to use, however, you must be careful with them. One problem that can occur is if you initialize a deferred system more than once. In the below example, suppose we called STUTTER twice. The first time we would save the original EMIT vector in OLD-EMIT and put in a new one. The second time we called it we would take our new function from EMIT and save it in OLD-EMIT overwriting what we had saved previously. Thus we would lose the original vector for

EMIT . You can avoid this if you check to see whether you have already done the defer. Here's an example of this technique.

```
DEFER OLD-EMIT
' QUIT  IS OLD-EMIT  ( set to known value )
: EEMMIITT  ( char --- , our fun EMIT )
    DUP OLD-EMIT OLD-EMIT
;
: STUTTER   ( --- )
    WHAT'S OLD-EMIT  'C QUIT =  ( still the same? )
    IF  ( this must be the first time )
        WHAT'S EMIT  ( get the current value of EMIT )
        IS OLD-EMIT  ( save this value in OLD-EMIT )
        'C EEMMIITT IS EMIT
    ELSE ."  Attempt to STUTTER twice!" CR
    THEN
;
: STOP-IT!  ( --- )
    WHAT'S OLD-EMIT ' QUIT =
    IF  ." STUTTER not installed!" CR
    ELSE  WHAT'S OLD-EMIT IS EMIT
        'C QUIT IS OLD-EMIT
      ( reset to show termination )
    THEN
;
```

In the above example, we could call STUTTER or STOP-IT! as many times as we want and still be safe. Look in the files **HSYS:LOGTO**, **HSYS:DEBUGGER**, and **HSYS:HISTORY** for examples of code that sets deferred system words in a safe manner.

Suppose you forget your word that EMIT now calls. As you compile new code you will overwrite the code that EMIT calls and it will crash miserably. You *must* reset any deferred words that call your code before you FORGET your code. The easiest way to do this is to use the word **IF.FORGOTTEN** to specify a cleanup word to be called if you ever FORGET the code in question. In the above example using EMIT , we could have said:

```
IF.FORGOTTEN  STOP-IT!
```

Tools for FORGET: ANEW, INCLUDE?, TASK-, etc.

When you are testing a file full of code, you will probably recompile many times. You will probably want to **FORGET** the old code before loading the new code. You could put a line at the beginning of your file like this:

```
FORGET XXXX-MINE     : XXXX-MINE ;
```

This would automatically FORGET for you every time you load. Unfortunately, you must define XXXX-MINE before you can ever load this file. We have a word that will automatically define a word for you the first time, then FORGET and redefine it each time after that. It is called **ANEW** and can be found at the beginning of most HMSL files. We use a prefix of **TASK-** followed by the filename just to be consistent. This TASK-name word is handy when working with **INCLUDE?** as well. Here is an example:

```
\ Start of file
INCLUDE? TASK-MYTHING HP:MYTHING
ANEW TASK-THIS-FILE
\ the rest of the file follows...
```

Notice that the INCLUDE? comes before the call to ANEW so that we don't FORGET MYTHING every time we recompile.

FORGET allows you to get rid of code that you have already compiled. This is an unusual feature in a programming language. It is very convenient in Forth but can cause problems. Most problems with

FORGET involve leaving addresses that point to the forgotten code that are not themselves forgotten. This can occur if you set a deferred system word to your word then FORGET your word. The system word which is below your word in the dictionary is pointing up to code that no longer exists. It will probably crash if called. (See discussion of DEFER above.) Another problem is if your code allocates memory, opens files, or opens windows. If your code is forgotten you may have no way to free or close these thing. You could also have a problems if you add addresses from your code to a table that is below your code. This might be a jump table or data table.

Since this is a common problem we have provided a tool for handling it. If you have some code that you know could potentially cause a problem if forgotten, then write a cleanup word that will eliminate the problem. This word could UNdefer words, free memory, etc. Then tell the system to call this word if the code is forgotten. Here is how:

```
: MY.CLEANUP  ( -- , do whatever )
    MY-MEM @ ?DUP
    IF  MM.FREE   0 MY-MEM !
    THEN
;
IF.FORGOTTEN  MY.CLEANUP
```

Notice that the cleanup word checks before doing MM.FREE. This word could be called any number of times and only the first time will have an action. Otherwise you would crash the second time.

IF.FORGOTTEN creates a linked list node containing your CFA that is checked by FORGET. Any nodes that end up above **HERE** (the Forth pointer to the top of the dictionary) after FORGET is done are executed.

Sometimes, you may need to extend the way that FORGET works. FORGET is not deferred, however, because that could cause some real problems. Instead, you can define a new version of [FORGET] which is searched for and executed by FORGET. You MUST call [FORGET] from your program or FORGET will not actually FORGET. Here is an example.

```
: [FORGET]  ( -- , my version )
    ." Change things around!" CR
    [FORGET]  ( must be called )
    ." Now put them back!" CR
;
: FOO ." Hello!" ;
FORGET FOO  ( Will print "Change things around!", etc.)
```

This is recommended over redefining FORGET because words like ANEW that call FORGET will now pick up your changes.

Creating a Turnkeyed Application

If you have a composition that you would like to share with others, you can create a *turnkeyed* version of it. This means that the composition will run when you double click on the icon. The Forth environment will be inaccessible so it cannot be used to program new HMSL compositions.

Important: **To avoid violating copyright laws, you must obtain written permission from Frog Peak Music before distributing a turnkeyed composition.** This is to prevent someone from simply turnkeying the Shape Editor, Splorp, or the Sequencer, and selling it. We reserve the right to do that ourselves. If, however, you write an original composition using HMSL that uses your own interactive screens, then we will probably grant permission. We encourage people to distribute their compositions in source code form so that other HMSL users can benefit from studying the code and modifying it.

A turnkeyed image is identical to a normal HMSL image except that it has no name fields. Names are only required during compilation. Eliminating name fields reduces the size of a turnkeyed composition file.

Before generating a turnkeyed application, you may want to eliminate some of the code you don't use. If you don't use anything but MIDI, you can probably FORGET the code above the HMSL MIDI code to reduce the size of the dictionary. See the file **HH:LOAD_HMSL** for variables you can use to optionally load or not load various modules.

Here are the steps for turnkeying a composition.

1) Make a copy of HMSL4th and give it a new name.

2) Double click on the new copy but do NOT INITIALIZE.

3) If you do not need actions, enter:

FORGET TASK-ACTION_UTILS

4) If you do not need the Shape Editor, enter:

FORGET TASK-SHAPE_EDITOR

5) Compile your composition. You should have a word that initializes HMSL, then initialize any screens, and then your piece. You will also need termination words. When your piece finishes, even if it aborted, the termination word will be called. For example:

```
: MY.INIT  ( --  )
   HMSL.INIT
   USER.INIT  ( initialize screens in a chain )
   PIECE.INIT  ( your initialization word )
;
: MY.TERM  ( -- )
   PIECE.TERM
   USER.TERM
   HMSL.TERM
;
```

6) Compile the file: **HSYS:TURNKEY.F**

7) To prevent confusion if you run use this application along with HMSL, change the MIDI manager Client ID from HMSL to your own 4 character ID. As an example, suppose your program uses chaotic attractors, you might want to do the following:

```
" KAOS"  OSTYPE:  'KAOS'
'KAOS'  ->  MIDIM_CLIENT
```

8) Use the TURNKEY word to save the currently compiled code as a standalone application. The stack diagram for TURNKEY is:

```
TURNKEY  ( cfa-init cfa-piece cfa-term -- )
```

As an example, if your piece is called MYPIECE, enter:

```
'C MY.INIT  'C  MYPIECE  'C MY.TERM  TURNKEY
```

9) Quit from HMSL

10) Test the turnkeyed composition.

11) Use ResEdit to change the icon. Change ICN#. You may have to rebuild the desktop to see your new icon.