Chapter 11
Miscellaneous Forth Tools

Memory Allocation

History

Historically, all memory allocation in traditional Forth systems has been from the available dictionary space, the same area that new definitions are added to. While this is sometimes a preferred technique, it is often lacking as the sole source of pooled memory.

With the advent of sophisticated operating systems such as that available on the Amiga, Forth may inherit the ability to 'request' memory from a free memory pool maintained at no burden to the application programs.

JForth provides a rich set of operators that fully interface with the Amiga memory manager, and additionally provide capabilities specifically suited to the Forth environment.

Allocate and Free Memory.

JForth provides two basic primitives for handling memory requests...

ALLOCBLOCK (type size -- memblk | FALSE)

Allocate from other than the available dictionary, a block of memory of 'size' number of bytes. The type may be appropriate combinations of: MEMF_PUBLIC, MEMF_CHIP, MEMF_FAST, MEMF_CLEAR and MEMF_LARGEST. Returns false if AllocMem() failed. Refer to the AmigaDOS ROM Kernel Manual for details on the Amiga memory map and use of the 'MEMF_' constants. NOTE: a type=0 is default PUBLIC, FAST if available.

FREEBLOCK (memblk --)

Frees the memory, making it available again. If you pass the address of memory that has already been freed then the Amiga will probably crash.

FREEVAR (var-addr --)

If the contents of the variable is non-zero, then the contents is passed to FREEBLOCK then the variable is set to zero. This is a handy word when you store the address of allocated memory in a variable.

```
VARIABLE MY-VAR
MEMF_CLEAR  1000 ALLOCBLOCK   \ allocate memory
DUP .  \ is it zero?
MY-VAR !  \ save address in variable
MY-VAR FREEVAR  \ now free it
```

The address returned from ALLOCBLOCK is a JForth relative address, and as such, it is legal to use the Forth memory access words (@, !, etc) on them.

The programmer may allocate memory with ALLOCBLOCK, and use it as desired. When he no longer needs the area, he should call FREEBLOCK to return it for other other tasks to use.

Note that allocated blocks will not be preserved across the SAVE-FORTH command or when CLONING. Your program should allocate blocks and free them at run-time. You should avoid calling ALLOCBLOCK while compiling. For large areas that are too difficult to build at run-time (such as

SINE or TRIG tables), we recommend storing them in a file, and loading them at run-time. See the chapter on the JForth File Manager.

Use Memory like a Stack.

Certain words provided by JForth are available to perform automatic storage into and out of these areas. These words maintain a 32-bit counter that is allocated along with each memory area; this counter is used to reflect the current amount stored in the memory block (when these words are used).

`FREEBYTE  ( memblk -- next-free-byte-pos )`

Returns the contents of the counter. This counter is set to zero when the memory block is allocated.

`FREEBYTEA ( memblk -- address-of-freebyte-counter )`

This function returns the address of the 'freebyte' counter. The programmer may make this counter available for local storage simply by not using the words that update it. This way, he may write to his area as desired, storing into the FREEBYTE counter to save, for example, how much data is there.

`SIZEMEM   ( memblk -- allocated-size )`

Return the originally allocated size of this block of memory.

The words that automatically update the FREEBYTE counter are concerned with utilizing the memory-block as a STACK. These first two implement it as a simple last-in/first-out buffer:

`PUSH  ( n1 memblk -- )`

Push 'n1' on the stack at 'memblk'.

`POP   ( memblk -- n1 )`

Retrieve the last item 'pushed' on this stack.

Note that, for efficiency reasons, error-checking has not been implemented in PUSH and POP..if there is a chance your stack could become full or empty, you should check the FREEBYTE contents and compare it with SIZEMEM before each PUSH or that it is not zero before a POP.

The next two words also use the area as a stack, but operate somewhat differently. Firstly, they do not expect the address of the area but a location CONTAINING the address.

`+STACK  ( n1 var-addr -- )`

Push 'n1' on the stack whose address is contained in the var-addr. Note that if the variable holds zero, the memory-block will be allocated, and the address placed in the variable.

`-STACK  ( n1 var-addr -- )`

Search for an occurrence of 'n1' on the stack whose address in contained in the var-addr. If found, remove the occurrence and return. If the number of elements on the stack becomes zero during this call, the memory block is de-allocated via FREEBLOCK, and the Var-addr will be cleared.

Note that error-checking has been implemented in +STACK and -STACK. +STACK will automatically expand the stack memory area if necessary (and change the contents of VAR-ADDR in the process). -STACK will deallocate the stack memory and clear VAR-ADDR if it removed the last stack item.

Example.

Allocate 1K of CHIP memory, accessible by anyone...

```
: GET1K  ( -- memblk )
    MEMF_PUBLIC  MEMF_CHIP  OR     1024
    ( -- type size ) ALLOCBLOCK ?DUP 0=
    IF   ." Memory allocation failed!"  QUIT
    THEN
;
VARIABLE  MY-MEM
GET1K MY-MEM !  ( allocate and save pointer )
```

We can now use this memory for any purpose.

```
    2345 MY-MEM @ !  ( set first cell in block )
    77 MY-MEM @ 100 + ! ( set cell at offset 100 )
```

Alternatively, we can this area as a stack. Let's push 11, 22 and 33 to 'MY-MEM'...

```
    11 MY-MEM @ PUSH
    22 MY-MEM @ PUSH
    33 MY-MEM @ PUSH
    765 MY-MEM @ PUSH
    MY-MEM @ POP . ( print 765 )
```

Now selectively remove the '22' ...

```
    22  MY-MEM -STACK
```

The MY-MEM stack now has (-- 11 33) We're all done, so return what we've borrowed...

```
    MY-MEM @  FREEBLOCK
```

Deferred Words

The DEFER facility allows you to call a word before it is actually defined. You can then define and redefine that word without recompiling the code that calls it. This is very useful if you are testing a word that is compiled at the beginning of a large program. It is also useful for redefining system words that you cannot recompile, like KEY , EMIT , and FIND .

Using DEFER to "vector" code.

Let's look at how we can use DEFER to experiment with a word. Enter the following at the keyboard:

```
    DEFER DOIT    ( create a deferred word )
    DOIT   ( executes QUIT )
    : USEIT  ( -- , use DOIT in a loop )
        8 0 DO  DOIT  LOOP
    ;
    USEIT
```

You will notice that nothing appears to happen when you enter DOIT or USEIT. This is because the default action for a deferred word is QUIT. Now let's try changing what DOIT does without recompiling USEIT . Enter:

```
    : HI ." Hello" CR ;
    HI
    ' HI IS DOIT
    DOIT
```

Notice that DOIT is now the same as HI. The Forth word IS takes a CFA from the stack and places it inside the deferred word for use when it is called. You can now enter USEIT and see that its execution has been changed without recompiling. This process of using a word to point to another word is called "vectored execution".

You can use WHAT'S to find out what a deferred word is defined as. Enter:

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

Deferred System Words

Many JForth system words are deferred (vectored) so that you can change how they work. This allows you to do things like redirect I/O and extend the compiler. These words are maintained by the JForth system. (See FREEZE).

Here is a partial list of JForth words which are deferred:

```
EMIT        KEY         ?TERMINAL   CR          QUIT
INTERPRET   FIND        :CREATE     NUMBER      ID.
SOURCE      BLOCK       'WORD       LONGCFA,    CFA,
FLUSHEMIT   COLDEXEC    ABORT       EXPECT      TYPE
```

To create a stuttering EMIT (2 emits/char) , we could try:

```
DEFER OLD-EMIT
```

DEFER creates a new deferred word called OLD-EMIT .

```
: EEMMIITT  ( char --- )
    DUP OLD-EMIT OLD-EMIT
;
: STUTTER   ( --- )
    WHAT'S EMIT  ( get the current value of EMIT )
    IS OLD-EMIT  ( save this value in OLD-EMIT )
    ' EEMMIITT IS EMIT
;
: STOP-IT!  ( --- )  WHAT'S OLD-EMIT IS EMIT  ;
STUTTER  .S
```

Things are going to be hard to read nnooww. After a while you'll get tired of tthhiiss ,, so enter:

```
STOP-IT!   ( this will appear as SSTTOOPP--IITT!! )
```

DEFER and GLOBAL-DEFER are functionally identical in JForth.

We recommend that you use a consistent naming convention for the words that you set the deferred words to. Brackets and parentheses are common, for example:

```
<ROBOT.OUTPUT>   or  (FILE.OUTPUT)
```

If you have a complicated program with a lot of deferred words, you might put in a word to display the state of all your important deferred words:

```
: SHOW-ME
    WHAT'S OUTPUT.FUNCTION >NAME ID.  CR
    WHAT'S INPUT.FUNCTION  >NAME ID.  CR
;
```

Potential Problems with Defer

Deferred words are very handy to use, however, you must be careful with them. Suppose you change EMIT so that it calls your word. Then 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!
```

Another problem that can occur is if you initialize a deferred system more than once. In the above 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 is a rewritten form of the above example that does this check.

```
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  ' 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 )
        ' 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
        ' QUIT IS OLD-EMIT  ( reset to show termination )
    THEN
;
IF.FORGOTTEN  STOP-IT!   ( make sure we clean up )
```

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 JU:LOGTO, JU:DEBUGGER, and JU:HISTORY for examples of code that sets deferred system words in a safe manner.

Finally, it is important to insure that, whenever you install a word into a deferred function, the word being installed does not directly call that deferred function. Doing so will create infinite recursion (actually, it will only recurse until it crashes). For example, since the word . (dot) normally calls EMIT, you cannot:

```
' . IS EMIT   ( never do this )
```

That would cause EMIT to call . which will call EMIT which will call . which will...get the idea?

Tools for FORGET

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 JForth 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-LOGTO JU:LOGTO
```

```
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 LOGTO 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 requires care. 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. (See the section above on DEFERred words)

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 things. You could also have a problem 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  FREEBLOCK   0 MY-MEM !
    THEN
;
IF.FORGOTTEN  MY.CLEANUP
```

Notice that the cleanup word checks before doing FREEBLOCK. 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 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
```

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

Local Variables

The code for Local Variable support is in the file JU:LOCALS . If you want to use locals in your program, place this line at the beginning of your file:

```
INCLUDE?  {  JU:LOCALS
```

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 preceeding 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
        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 the earlier defined word of the same name will not be accessible until the ; at the end of the definition is reached. Other errors that can occur include, missing a closing '}', missing '--', or having too many local variables.

Logging to Files or the Printer

The output of JForth can be made to echo to a file as well as the keyboard by changing the deferred word EMIT. This is useful for generating records of your work, dumping memory to a file for later analysis, etc. If you are trying to generate formatted data files, you can get the code to work properly on the screen first. Then just log your output to a file. By sending the output to the file "PRT:", you can get JForth to send it's output to the printer. The words needed to do this are defined in the utility file called "JU:LOGTO".

$LOGTO ($filename -- , Send copy of output to file.)

LOGTO (<filename> -- , Get filename and pass to $LOGTO)

LOGSTOP (-- , Temporarily stop echoing.)

LOGSTART (-- , Continue echoing, used after LOGSTOP)

LOGEND (-- , Stop echoing, close file opened by LOGTO)

PRINTER.OFF (-- , turn off printer echo)

PRINTER.ON (-- , echo to printer as file PRT:)

As an example, if you want to dump some memory and have it printed, enter the following:

```
: XXXDUMP  ( addr count -- )
   " RAM:XXX" $LOGTO  ( Echo to file RAM:XXX )
   DUMP
   LOGEND
;
' SWAP  100  XXXDUMP
TYPEFILE RAM:XXX
```

Word Usage Analysis

Using this facility it is possible to keep track what of words are referenced, or not referenced, by your code. This is useful when trimming unneeded words from your program. It can also catch words that were defined and should have been used but weren't.

CLEAR.MARKS (-- , Clears flag in all words to UNUSED state)

START.MARKING.WORDS (-- , Tells FIND to mark all words used.)

STOP.MARKING.WORDS (-- , Turns off this facility)

UNUSED.WORDS (-- , Print all UNUSED words in dictionary.)

USED.WORDS (-- , Print all USED words in dictionary.)

To use this facility, include the utility file "JDEV:UNUSED". Then enter START.MARKING.WORDS . From now on, all words referenced from the keyboard or from a file will be marked as used. You can now load the program that you want to analyze. To find out which words have been referenced by that program, enter USED.WORDS . To find out which words were not referenced, enter UNUSED.WORDS . To clear these flags, enter CLEAR.MARKS .

Error Handling

File: JU:ERROR_CODES

What do you do when something goes wrong? If you can't allocate memory or open a file, what do you do next? It is tempting to simply call ABORT when an error is encountered. This may work fine when debugging an application in Forth because you will be put back into the Forth interpreter. There you can investigate the source of the error and do something about it. When you clone an application, however, abort will cause the program to quit. If your user has just spent 2 hours drawing a masterpiece and the program quits when it can't allocate memory for a brush, you will have failed. An application should always do its best to continue executing and to give the user a chance to save her work.

Although it involves extra work, we recommend that routines subject to errors should return an error flag that is FALSE if everything went OK, and non-zero if an error occured. You can return different codes to indicate different types of errors if more than one is possible. A calling program can check the error code and act appropriately.

We have provided a few tools to simplify, and perhaps standardize, error handling in JForth applications. The first of these is a word that defines succesive error codes.

`ERR: ( n <name> -- n+1 , define constant )`

This was used to define codes for the two most common errors in the following way:

```
1
ERR: ERR_FILE_NOT_FOUND        \ equals 1
ERR: ERR_INSUFFICIENT_MEMORY   \ equals 2
DROP  \ don't leave N on stack
```

When you encounter an error, you should inform the user by outputting a text message. You may want to put all of your error message text in a file that is read by your program as needed. This simplifies translation to other languages.

The dreaded GOTO

File: JU:GOTO_ERROR

Many languages have a command called GOTO that lets them jump directly from one place to another. This seems like a handy thing but its use leads to a horrible disease called "spaghetti code". Try to debug some old FORTRAN or BASIC program riddled with GOTOs and you'll know what I mean. It turns out that GOTO is really not needed in a high level language. You can use IF ELSE THEN BEGIN UNTIL etc. to achieve the same effect and produce code that is much more readable and easier to maintain.

You may now all gasp with horror and righteous indignation as I announce the addition of a GOTO to JForth. As I swing by my thumbs over the flames, let me explain why this is really OK.

The one single use of GOTO that has been grudgingly confirmed as a good thing, is when used in error processing. Imagine that you are writing a routine that allocates memory several times. Any one of these allocations could fail. You could use lots of IF statements but soon you will be indenting code right off the side of the screen. An alternative is to use a GOTO.ERROR after each allocation. If the allocation fails, the program will jump to the code following an ERROR: statement. That code can clean up everything and return a proper error code to the caller. Here is a trivial example that demonstrates the use of GOTO.ERROR and ?GOTO.ERROR. It uses local variables because they make it easier to keep the stack clean.

```
include? task-error_codes ju:error_codes
include? task-goto_error ju:goto_error
include? { ju:locals

: TEST.GOTO { aa bb -- error? , simple example }
   aa 0= ?GOTO.ERROR
\
   bb 0<
   IF
      GOTO.ERROR
   THEN
\
   FALSE EXIT  \ exit causes immediate return to caller
\
ERROR:   \ continue from here if error
```

```
      >newline ." Uh Oh!" cr
      TRUE
   ;
```

Try entering:

```
   1  2 TEST.GOTO .
   0  2 TEST.GOTO .
   1 -2 TEST.GOTO .
```

Here is a more complex example that returns different error codes. Instead of calling EXIT we let the code that executes without error continue into the cleanup code.

```
   variable BIG-BUFFER \ hold pointers to allocated memory
   variable MY-FILE

   : DUMP.FILE  ( $filename -- error? )
   \
   \ try to open file
      $fopen ?dup
      IF
         my-file !
      ELSE
         ERR_FILE_NOT_FOUND GOTO.ERROR
      THEN
   \
   \ allocate memory to read data into
      memf_clear 200 allocblock ?dup
      IF
         big-buffer !
      ELSE
         ERR_INSUFFICIENT_MEMORY GOTO.ERROR
      THEN
   \
   \ read file into memory and dump
      MY-FILE @ BIG-BUFFER @ 200 FREAD
      BIG-BUFFER @ SWAP DUMP  \ dump what we got
   \
      FALSE  \ return OK
   \
   ERROR:
   \ free memory pointed to by variable if allocated
      BIG-BUFFER FREEVAR
   \
   \ close file pointed to by variable if open
      MY-FILE FCLOSEVAR
   ;
```

Caution: when using GOTO.ERROR, make sure that you return the proper items on the stack. Using local variables can greatly simplify this task.