Forth!

by Jon Bryan

...there are some fundamental differences between Multi-Forth and JForth.

In my last column, I mentioned that I had received a copy of JForth from Delta Research in Palo Alto. I am pleased to say that it has the makings of a good development system. I have already received an update, including the "real" manual which has been from preliminary version. Delta Research has been very helpful when I have called them with questions; from early indications, the support promises to be excellent.

While Delta Research has been preparing JForth, Creative Solutions has been busy too, working on improvements to Multi-Forth. I received a preliminary version of an on-line "help" disk from them. A Programmer's Toolbox is in the works, and the next revision (1.2) will probably be out by the time you read this article. Creative Solutions continues to improve an already robust product and provide outstanding support.

A CLASSIC COMPROMISE

Perhaps, it's East Coast versus West Coast (I'm being facetious). In any case, there are some fundamental differences between Multi-Forth and JForth. I would like to use this month's column to compare the two approaches, present a few benchmarks, and make some observations about the relative merits of each.

Foremost among the differences is the threading scheme each uses. Multi-Forth is a more traditional implementation of Forth. It is a "threaded interpretive" language, in the sense that it compiles data elements rather than machine instructions. In Multi-Forth's scheme, a word is compiled as a sixteen-bit "token" which represents an offset from a base register. On the other hand, JForth comes closer to the traditional definition of a "compiled" language because it lays down 68000 machine code. It's still "threaded," but by means of subroutine calls.

As a result, what you have is the classic trade-off between size and speed. Because each reference to a word only takes sixteen bits, Multi-Forth is quite small. I was able to squeeze the bouncing ball demo (presented a few issues back) down to about 34k. 25k might be attainable. JForth weighs in at about twice the size of Multi-Forth in its minimum configuration, and grows much faster, but runs certain kinds of tasks two to five times faster than Multi-Forth. Because so much time is absorbed by Amiga ROM, calls the difference in speed should be less in "real" applications. I would still expect JForth to be faster by a significant margin.

RELATIVE MERITS

I have always been impressed with the quality of Multi-Forth. It is the product of Creative Solutions' years of experience working with every conceivable 68000 system. The Amiga provided the testbed for a completely new kernal which was subsequently ported to the Atari and Macintosh. The product does show a few rough edges because of the brand-new implementation. Most of the bugs seem to have been worked out, though,and the next update will incorporate local multi-tasking. CSI has reduced the price from $180 to $90, making an already excellent value even better.

JForth is taking some getting used to. Its designers have taken an approach which differs considerably from Creative Solutions' approach. I've had to make a few mental adjustments in order to make the transition. Forth tends to amplify the differences as well. Everyone has his/her own concept of the right way to do things, and Forth allows each to be pursued with equal ease.

I have to say that I'm tremendously impressed by the amount of effort which has gone into JForth. Much of the kernal is included on disk in source form, allowing users to customize their own systems. Printing out all the source listings included on the two disks resulted in a two-inch-thick stack of paper! For $100, you get a lot for your money!

Both Multi-Forth and JForth are on the cutting edge of Forth technology. Multi-Forth uses what is perhaps the fastest, most compact form of token threading possible on the 68000 and local variables. The next revision should incorporate local multi-tasking . JForth uses subroutine threading for speed (with a heavy penalty in object code size), a different method of implementing local variables. JForth should also have local multi-tasking soon.

A COMPARISON OF FEATURES

JForth is loaded with nice touches. One particularly useful feature is a Motorola syntax disassembler invoked by the word DEF. As an example of its use, consider the word 2*, a fast way of multiplying a number by two. The response to "DEF 2*" is:

```
8E2  ADD.L  TOS,TOS       DE87     ( 6:  6:  0)
8E4  RTS                  4E75     (16: 22:  3)
```

2* is 4 bytes long (1 cells), defined as 'both', indicating that 2* simply adds the top stack value (TOS) to itself. The first column is the relative address of the word, which may be different for you. The second column is the disassembly of

continued...

the instruction. The third column gives the hex values for the instruction and operand(s). The three values in parentheses indicate the number of CPU cycles used by the instruction, the running total of CPU cycles, and that total converted to microseconds. There is some guesswork involved, and occasionally, a question mark will be appended indicating that the disassembler couldn't give the exact number of cycles. This tip should be handy if you're a clock-watcher.

In the comment, a "cell" in JForth is four bytes. "Defined as 'both'" indicates that the word will be compiled in-line or called, depending on the value of a variable, MAX-INLINE. This option allows the user to have some control over the size of the object code. In fact, in a particularly time-critical application, MAX-INLINE could be set temporarily to a large value for a single definition and then reset back to the original value. The entire application doesn't have to suffer the size penalty if the extra time spent in branches and jumps can be tolerated. The word 2* can be used to define 4* like this:

```
: 4*   ( n --- 4n )   2* 2* both ;
```

Which when DEFed would appear as:

```
1AF38  ADD.L  TOS,TOS      DE87     ( 6:  6:  0)
1AF3A  ADD.L  TOS,TOS      DE87     ( 6: 12:  1)
1AF3C  RTS                 4E75     (16: 22:  3)
```

4* is 6 bytes long (1 cells), defined as 'both'. I find the use of "both" curious. It seems that it should be used like "immediate" and follow the semicolon, but it apparently has something to do with a test which checks to see if the word can safely be used in-line. If the compiler decides that the word can't be used, it issues a warning and flags the word as "called."

Another feature of JForth is the mechanism used for calling libraries. Each library has a "formal definition" file which consists of entries for each function in the library. The entry for WritePixel in the graphics_lib.fd file is:

```
WritePixel(rastPort,x,y)(A1,D0/D1)
```

This entry indicates that the function expects a rastPort address and an x and y value and passes them in the A1, D0 and D1 registers. The syntax for calling WritePixel is:

```
call graphics_lib WritePixel
```

This syntax must be used within a definition. Of course, the appropriate values must be on the stack. "Call" would search the "graphics_lib.fd" file for the entry for WritePixel and then compile the instruction sequence to load to the function, the appropriate registers from the stack and jump to the function. JForth library calls always return the D0 register as well, so in the case of WritePixel, the call should properly be followed by "drop." I'll give a real code example shortly.

JForth, like Multi-Forth, includes words for defining the equivalent of 'C' data structures. Considering the Amiga's reliance on C-style data structures, they had very little choice. They also provide a program which converts C include files to JForth syntax, a utility which is sorely lacking in Multi-Forth. An example of a JForth structure definition would be:

```
:STRUCT Node
      APTR ln_Succ
      APTR ln_Pred
      BYTE ln_Type
      BYTE ln_Pri
      APTR ln_Name
   ;STRUCT
```

The fundamental difference between JForth structures and Multi-Forth structures is that members of JForth structures are typed, where those in Multi-Forth are not. Special words named ..@ and ..!, which automatically use the appropriate @ or ! word, are provided for accessing members of a structure. In JForth, LONG or APTR members will use @ and !, SHORT members W@ and W!, and BYTE members C@ and C!. An example of the syntax would be:

```
NT_MSGPORT myNode ..! ln_Type
```

I think this syntax is awkward and confusing. The fact that ..! has to look ahead in the input stream is contrary to standard practice. A better syntax might have had the member leave a flag and name the operators ?@ and ?! (don't you love these cryptic Forth names?). The concept certainly has merit, though.

Local variables are a recent development in Forth. My first exposure to these variables was in Amiga Multi-Forth. I think it's a great concept ! Delta Research must think so too, because they implemented them in JForth. The only problem is that they implemented them differently. In Multi-Forth they are specified using the syntax:

```
LOCALS| c b a |
```

Whereas in JForth the syntax is:

```
{ a b c --- }
```

The JForth syntax looks like a stack comment with braces substituted for parentheses. Multi-Forth locals are taken off the stack in the order in which they appear. JForth locals are called out in stack order, with the top stack item on the right. Multi-Forth allows any legal name to be used for a local, while JForth disallows a leading minus sign. BUT ... JForth gives you the source code for their implementation, so that this quirk can (and probably will) be fixed.

JForth locals include a few more bells and whistles as well. The notation:

```
{ a b | c --> c }
```

This notation allocates an uninitialized entry named "c" on the stack and causes the value of c to be returned. A few other operators are included. An example from the JForth manual reads:

```
: EXAMPLE   { a b | c --> c }
     a b + -> c  a c + -> c ;
```

The "->" is equivalent to "to" in Multi-Forth, and the syntax:

```
a b + -> c
```

The syntax stores the result of the addition of a and b into c.

Locals in both languages are self-fetching, which is not always the way you want things to be. The address of a local is accessed in Multi-Forth with the word "addr.of". JForth requires a different approach. In Multi-Forth, "addr.of" works immediately at the time a word is executed. The decision has to be made at compile time in JForth, using the directives "no@" and "yes@" to control how the local is handled. Because I was expecting behavior similar to Multi-Forth, this one caught me the first time. Again, here is an example from the JForth manual:

```
: EXAMPLE   { a b c --- sum+1 }   \ "sum+1" is a comment
     a b + c + -> c
     no@  ( self-fetching off )  1 c +!
     yes@  c ;
```

The example which will be given shortly in a circle algorithm should help to make this more clear.

The last difference between JForth and Multi-Forth local variables is that JForth locals are left on the data stack, while Multi-Forth moves them to the return stack (the approach used by another company is to move them to a third stack!). Each approach has its own quirks. In Multi-Forth, you must be careful with words like >R and R> when you're using locals. In JForth, you must watch what you're doing with the data stack. Both problems have bitten me. A completely separate stack has the fewest side-effects, but it is also the slowest.

FIERO OR TRANS-AM?

The biggest problem I see with JForth is the size. It starts off big and gets much bigger in a hurry. The system comes up without any graphics support, and adding the support absorbs over 24k. In contrast, Multi-Forth comes up with mostly equivalent graphics support and is half the size of JForth. By the time the demos are compiled on a 512k machine, there is only about 50k of memory left. There were times when I couldn't run Ed without shrinking some windows. If you're going to use JForth as a development system, I would recommend adding at least another 512k of memory. A full megabyte would be better, and you should pray that Delta Research gets their target compiler done soon.

If the target compiler works as promised by Delta Research, then a turnkey of a simple program like "Hello World" should compile to less than 1k. Without that utility, I'm not sure whether you can develop viable applications for 512k machines. Multi-Forth doesn't have a target compiler either, but they have less need for one, since they have a very compact implementation to begin with.

It's debatable how much difference between the two you will see in terms of speed on equivalent applications. JForth is faster, but it achieves that speed by making EVERYTHING fast and big. In Multi-Forth, you would typically optimize only the parts that needed the extra speed. In most applications, that probably wouldn't even be necessary. As both a simple benchmark and an example of JForth code, I have included a version of Bresenham's circle-drawing algorithm. This example illustrates some of the differences between the two implementations. Of particular interest is JForth's handling of local variables and the mechanism for calling library routines. In timing comparisons, JForth is approximately 35% faster than Multi-Forth on this particular algorithm. Each algorithm is presented in the form necessary to run from the default configuration of each language. To try them out bring up the language, compile the code, and then execute "samplewindow" to open a window to draw into. When you finish, "closecurrentwindow" will clean things up. Enjoy.

SOURCE CODE

First, the algorithm in JForth:

```
\ Bresenham/Michener circle algorithm.
\ Written in JForth by Delta Research
\ From "Fundamentals of Interactive Computer Graphics"
\ by Foley and Van Dam.
\ Jon R. Bryan 3/22/87

INCLUDE? GR.INIT JU:AMIGA_GRAPH
GR.INIT

INCLUDE? TASK-LOCALS JU:LOCALS


: 4*   ( n -- n*4 )   2* 2* both ;


newwindow my-window


: samplewindow   ( -- )
   my-window newwindow.setup
   my-window gr.openwindow
   gr.set.curwindow ;


: closecurrentwindow   ( -- )   gr.closecurw  gr.term ;


variable xoffset
variable yoffset


: >xyoffset   ( x\y -- xrelative\yrelative )
   swap xoffset @ +  swap yoffset @ + ;


: dot   ( x\y -- plots a pixel )
   >xyoffset  gr-currport @ rot rot
   call graphics_lib WritePixel drop ;


: circle_points   ( x\y -- )
   over negate over negate
   { x y x- y-  --- }   \ remember, no preceding -
   x y  dot  y  x dot  x- y  dot  y  x- dot
   x y- dot  y- x dot  x- y- dot  y- x- dot ;
decimal
```

continued...

```
: mich_circle    ( xcenter\ycenter\radius -- )
  3 over 2* -  0
  ( xcenter ycenter y d x --- )
  \ xcenter and ycenter must be in the local list
  xcenter xoffset !  ycenter yoffset !
  BEGIN   x y <
  WHILE   x y circle_points
    d 0<
    IF  x 4*  6 + no@ d +!
    ELSE  yes@
      x y - 4*  10 + no@ d +!
      -1 y +!
    THEN
    1 x +!
  REPEAT  yes@
  x y =
  IF  x y circle_points  THEN ;
```

The equivalent in Multi-Forth

```
\ Bresenham/Michener circle algorithm
\ written in Multi-Forth by Creative Solutions
\ Jon R. Bryan 3-22-87


variable xoffset
variable yoffset


: >xyoffset    ( x\y -- xrelative\yrelative )
   swap xoffset @ +  swap yoffset @ + ;

: dot    ( x\y -- )
   >xyoffset Rport !a1 !d1 !d0 graphics 54 ;


: circle_points    ( x\y -- )    \ 8-way symmetry
   over negate over negate
   locals|  -y  -x  y  x  |
   x  y dot    y x dot   -x  y dot    y -x dot
   x -y dot   -y x dot   -x -y dot   -y -x dot ;


: mich_circle    ( xcenter\ycenter\radius -- )
   3 over 2* -  0
   locals|  x  d  y  |
   \ You can still access the data stack here
   xoffset !  yoffset !
   BEGIN   x y <
   WHILE   x y circle_points
     d 0<
     IF  x 4* 6+ addr.of d +!
     ELSE
       x y - 4* 10+ addr.of d +!
       -1 addr.of y +!
     THEN
     1 addr.of x +!
   REPEAT
   x y =
   IF  x y circle_points  THEN ;
```

•AC•

Multi-Forth...*at a new low price... $89*

Simply the best programming environment for the Amiga

Multi-Forth is a new language designed to unleash the full power of the Amiga. Multi-Forth provides complete access to all Amiga libraries including Intuition. It compiles stand-alone applications in seconds (other languages typically take several minutes). There are no royalties, and no "levels." CSI provides the best support of any computer language vendor, including CSI technical hot line, our own CompuServe public forum at GO FORTH with hundreds of public domain/shareware programs, and a comprehensive 350 page manual. Programming the amazing Amiga is interactive and fun with Multi-Forth.

Contact us for a technical data sheet with the complete list of features.

Creative Solutions

Multi-Forth is a registered trademark of Creative Solutions, Inc.
Amiga is a trademark of Commodore Corporation.

4701 Randolph Rd. Suite 12
Rockville, MD 20852
301-984-0262 in MD or
1-800-FORTH-OK (367-8465)