Chapter 17
Miscellaneous Development Tools

Command Line History

Since AmigaDOS 1.3, Amiga users have grown accustomed to the convienience of command line history, the feature in the SHELL that allows you to scroll through previous CLI commands using the arrow cursor keys. Since this feature is only present in the CON: device (in AmigaDOS 1.3, the NEWCON: device), and JForth uses RAW: Amiga windows (to aquire single keystrokes for KEY), this feature was not automatically available. We decided, therefore, to add our own command line history that is similar to the Shell history. We also added the capability of loading the function keys with your own custom commands.

Using the Cursor Keys

[Note: Special keys are marked in this text by angle brackets. For example, <CR> , <HELP> , <UP-ARROW> , and <F7> all refer to single keys and not strings.]

The History feature should be present in the normal JForth image. To test this, hit the <HELP> key. You should see a list of function key assignments. If it is not loaded, enter the following:

```
^X  (to clear characters from <HELP> key.
INCLUDE? HISTORY JU:HISTORY
```

The PANIC.BUTTON key <shift-F1> is for use if the Forth Outer Interpreter gets messed up from a program error. If you cannot enter valid commands at the keyboard without getting an error message, try the PANIC.BUTTON. It tries to "simplify" and reset the environment, hopefully giving you a last chance to enter Forth and see what went wrong. History will be automatically turned on when loaded. Now enter the following lines:

```
23 45 67
SWAP .S
DUP . CR
```

Now use the <UP-ARROW> and <DOWN-ARROW> keys to scroll through these previous commands. At any time you can use the <LEFT-ARROW> and <RIGHT-ARROW> keys to move within a line for editing. If you hold the shift key down while hitting <UP-ARROW> you can search back for lines that start with the same string as the current line. For example, enter the following:

```
<CR>
SW<SHIFT+UP-ARROW>
```

You should now see the line that started with SWAP. Shift left and right arrow will move you to the beginning and end of the line respectively.

If you hit <HELP> you will see a list of commands available on the function keys. Notice that some of the commands, like "INCLUDE" have quotes around them. This means that they will be inserted into the input stream. Try these out.

Wouldn't it be nice if you could easily distinguish between the text you entered and what the computer output. You can make your input text be drawn in color 3 by entering:

```
HIGHLIGHT-INPUT ON
CR ." Hello"  ( in a new color )
```

You can use Preferences to change these colors if need be.

Have you ever entered a great Forth word on the keyboard and wished you had done it in a file. You can use LOGTO to save the past keyboard commands into a file. Enter:

```
INCLUDE? LOGTO JU:LOGTO
LOGTO RAM:SAVEIT
HISTORY
LOGEND
```

Now in the CLI enter:

```
TYPE RAM:SAVEIT
```

Note: History will be disabled in a Cloned Application because it is primarily intended as a development tool. Also many Cloned programs use the CLI which already has its own History. If you want to use it in an application, use a RAW: window for your I/O, and remove the definition of KH.EXPECT from CL:REDEFS.F then recompile Clone. The entry in the REDEFS.f file is what disables it when cloned.

How History Works

HISTORY.ON installs a new function into the deferred word EXPECT. This new EXPECT handler performs it's own backspacing, cursor handling, etc. It uses ANSI commands from JU:ANSI to move the cursor and to parse command keys that are hit. As commands are entered they are added a list of commands stored in a byte array. The buffer contains lines with count bytes at each end of each string. This makes it easier to go back and forth in the list.

History Glossary

These commands are part of the JForth History feature.

$>EXPECT ($string -- , command history insert string)

Place a string in the EXPECT buffer. This is used when building function key words that put a string in the input stream. For example, "INCLUDE" which is on <F1> uses this. To add FORGET to <F8> enter the following:

```
: "FORGET"  ( -- , insert FORGET into input )
    " FORGET " $>EXPECT  ( notice space after last letter )
;
' "FORGET" 8 FKEY-VECTORS !
```

See also FKEY-VECTORS .

HIGHLIGHT-INPUT (-- addr , make user input different color)

Set this variable to TRUE to echo your input in color 3. Output from JForth will be in color 1.

HISTORY (-- , print previous commands)

HISTORY# (-- , print history with numbers for XX)

HISTORY.ON (-- , turn on history buffer and function keys)

HISTORY.OFF (-- , turn off history and function keys)

FKEY-VECTORS (key# -- addr , jump table for function keys)

This is an array that holds CFAs for the 10 function keys. Here is an example of loading a function key with a function.

```
: STATE? ( -- , NO stack activity allowed )
    >NEWLINE ." State = " STATE ? CR
;
\  Assign STATE? to function key 7
```

```
  ' STATE? 7 FKEY-VECTORS !
\ Now test it by hitting <F7> while compiling.
: FOO ( -- , start compiling )
    ." Hello<F7> Fred" CR ;
<HELP>  ( see it in list )
```

If you want to assign a function to a shifted function key, add 10 to the key#. Thus if you had used 17 instead of 7 in the above example, You could hold down <SHIFT> then hit <F7> to see the value of STATE.

KH_HISTORY_SIZE (-- size , size in bytes of buffer)

If you would like a bigger history buffer, edit the file JU:History and change the constant KH_HISTORY_SIZE to a bigger value. Then regenerate COM:JForth as described in Chapter 12, System Internals, in the section labeled *How to Generate a New JForth System*.

TAB-WIDTH (-- addr , variable containing TAB width)

When a TAB character is entered, a number of BLANK characters are inserted in its place so that the cursor will be at a multiple of TAB-WIDTH.

XX (line# -- , execute previous line)

Use HISTORY# (which, by default, is assigned to a function key) to see a list of previous commands with their numbers. To execute a previous command, just pass that number to XX . Once filled, the oldest command lines are lost from the end of the buffer as new command lines are added.

Vocabularies

Vocabularies provide a way to group words that are to be used in a specific context. Examples within JForth are the ASSEMBLER vocabulary and the DISASSEMBLER vocabulary. Within the ASSEMBLER vocabulary are words like NOT , and SWAP , that are 68000 opcodes. These obviously conflict with existing Forth words. By specifying which vocabulary you are using, one can avoid any ambiguity.

Vocabularies can also be used to speed up compilation by temporarily removing a group of words that don't need to be included in the dictionary search, but this will provide benefit only if the HASHed dictionary search is not in effect. JForth, as released enables the HASHed search.

One can specify the order in which to search a set of vocabularies. JForth uses a CONTEXT stack to determine this order. The system is initialized at cold to:

```
FORTH ROOT
```

This means that JForth will search FORTH, and then ROOT, for any word typed in or encountered while compiling or interpreting. FIND is the word that performs this search. You can display the search order at any time using ORDER . At COLD this will display:

```
Searching (CONTEXT) : FORTH       ROOT
Extending (CURRENT) : FORTH
```

The first line shows the entire search path used by FIND on words that are encountered from the keyboard or a file. These are called the CONTEXT vocabularies, because they define the CONTEXT in which the input stream will be INTERPRETed.

The CURRENT vocabulary is that vocabulary to which new words will be added to as they are compiled, a process also known as 'extending' the vocabulary.

Vocabulary Tutorial

Before we begin exploring vocabularies, lets reset the search order so that we start from a known state. Enter:

```
ONLY ROOT FORTH
ORDER
```

Now let's create a vocabulary called MUSIC that we can place MUSIC related words in. Enter:

```
VOCABULARY MUSIC
ORDER
```

Notice that the MUSIC vocabulary is not automatically placed in the search order. We can add this VOCABULARY to the search order using ALSO. Enter:

```
ALSO MUSIC
ORDER
```

We now would like any music related words we define to be added to the MUSIC vocabulary. For this to happen, the CURRENT vocabulary must be set to MUSIC. DEFINITIONS will copy the top of the CONTEXT stack to CURRENT.

```
: FOO ." Regular old FOO!" CR ;
DEFINITIONS ORDER
```

FOO will have been defined in the normal FORTH dictionary because it was before DEFINITIONS. Everything after will go into the MUSIC vocabulary. Now let's define some music stuff.

```
: NOTES ." fa la la!" CR ;
: SYMPHONY ." da da da duuuuh!" CR ;
: FOO NOTES SYMPHONY ;
FOO
```

The musical FOO is executed because MUSIC is first in the search order. We can get the old FOO by dropping MUSIC from the top of the CONTEXT stack.

```
PREVIOUS DEFINITIONS ORDER
FOO
```

Vocabulary Glossary

ALSO (-- , dups the top of the vocabulary stack)

Usually this is followed by a vocabulary call as in:

```
ALSO ASSEMBLER
```

This will cause the ASSEMBLER vocabulary to be added to the search order, first in the list.

DEFINITIONS (--)

Sets the current vocabulary to the top vocabulary on the context stack. This results in new definitions being added to whatever vocabulary is first in the search order.

MAXVOCS (-- n , maximum number of vocabularies allowed)

JForth allows 32 different vocabularies to be defined.

ONLY (-- , reinitialize context stack to Forth and ROOT)

If followed by a vocabulary name, the search order becomes that vocabulary, then ROOT, and no others.

ORDER (-- , print CONTEXT stack and CURRENT)

Displays the current search order, left to right.

PREVIOUS (-- , drops the top item off the vocabulary stack)

Removes the first vocabulary in the search order, the one 'just below' becomes firstmost.

SEARCH-CURRENT (-- addr , variable to control search)

FIND checks SEARCH-CURRENT to decide whether to search the current vocabulay after it searches all the context vocabularies. JForth default is to set SEARCH-CURRENT OFF. (It is particularly useful to not search CURRENT when compiling new Forths or Forth-like packages, as a cross-compiler application might do, where execution of a target word would crash the host system).

VOCABULARY (<name> --)

Define a new vocabulary called NAME. When this vocabulary is executed it will place itself at the top of the CONTEXT search order. Vocabularies are not IMMEDIATE.

Vocabulary Internals

VOC-LINK points to the linked list of vocabularies starting with the most recently defined.

Current JForth vocabulary structure:

```
voc-link@:    ( link to other vocabularies )
voc-latest@:  ( point to last word in vocabulary )
```

The structure of vocabularies may be changed in the future. Therefore, use the words we have supplied for moving around the vocabulary structure:

```
VLINK>
VLINK>VLATEST
VLATEST>VLINK
```

VLINK is the linked list address for a particular vocabulary. VLATEST is the address of the vocabulary pointer to the top of its linked list of words. Each vocabulay list of words is terminated with zero in the LINK field.

Here is an example of using these words.

```
VOC-LINK @ .    ( get pointer to last vocabulary )
VOC-LINK @ VLINK>VLATEST @ ID. ( last word in vocab)
VOC-LINK @ VLINK>' >NAME ID.  ( name of vocab )
VOC-LINK @ @ VLINK>' >NAME ID.  ( print next vocab )
```

Look at the VOCS source file if you want to delve deeper.

SHOWHUNKS - for Analyzing Amiga Binary Files

JU:SHOWHUNKS may prove helpful to anyone that needs to get 'into' object files, libraries, executables, any file that conforms to the Amiga Binary Format.

Using the com:JForth image, type:

```
INCLUDE JU:SHOWHUNKS
```

You may be asked about INCLUDEing JU:DISM while this program compiles. If you want to display CODE hunks as disassembled 68000 code (instead of just HEX numbers), instruct the program to use the disassembler.

Also, if there is not enough dictionary space for JU:DISM, you will be informed as such.

When its done compiling, try:

```
SHOWHUNKS C:DIR
```

What you'll be looking at is the CLI 'dir' command file.

CODE_LIMIT is a variable it uses to tell what the largest HUNK_CODE it should display, in cells. Its default value is 100,000 (displays anything less than 100,000 bytes)...which should show you most, if not all, programs you ask for. If you wish to lower this amount to mask out the larger hunks, just enter:

```
100 CODE_LIMIT !  \ only display CODE_HUNKS less than 100 bytes
in size
```

The final interesting thing about this file is that it uses an execution array to perform specific actions on each type of hunk; you may define your own 'array' of vectors (cfa's)... one for each HUNK_TYPE (see the table in the source for examples)...put the base address of the table in ACTIONSARRAY then type:

```
PROCESSHUNKS <FILENAME>
```

See the definition of SHOWHUNKS, last thing in the file.

Yes, I suppose you COULD write your own Loader/Linker with PROCESSHUNKS...

JForth Optimizing Compiler Extensions

This utility extends the JForth Professional V2.x and V3.x compiler such that it will, under certain circumstances, produce smaller, tighter and faster code.

How to use the Optimiser

1. Load the com:JForth image.

2. Compile the file 'jdev:opt.f'.

3. From the keyboard, type: OPT followed by the RETURN key.

At this point the optimizer is installed, and definitions which are loaded on top of this image will be optimized where possible.

Note that the supplied 'jdev:opttest.f' file illustrates the improvement that can be realized. To run this test...

1. Deactivate the optimizer by entering: OPTOFF

2. Run the test by entering: INCLUDE jdev:opttest.f Note the times that are printed to screen

3. Activate the optimizer by entering: OPT

4. Run the test again by entering: INCLUDE jdev:opttest.f Compare with the previous results.

Optimizer Glossary

OPT (--)

Activates the optimizing extensions by setting INTERPRET vector.

NOOPT (--)

De-activates the opimizer, reverts the compiler behavior to normal by restoring the INTERPRET vector.

OPTON (-- , same as OPT)

OPTOFF (-- , same as NOOPT)

Caveats

1. Dictionary size - this version of the optimizer requires about 7-8K of dictionary. Since it would take a very large application to save this much room due to optimizations, the most benefit is felt in CLONEd programs.

2. The keywords OPT and OPTON (same function, different name) will only operate as expected if typed from the keyboard (vs. INTERPRETed from a file).

3. I believe the JU:DEBUGGER utility and OPT do not co-exist well; they tend to 'un-install' each

other. If you need to use the debugger, just OPTOFF before starting your debug compile.

4. Don't FORGET the optimizer while it's installed!!!! (IF.FORGOTTEN doesn't help here!)

Technicals (for those who care)

OPT checks for sequential compilation of a subset of JForth words, specifically (in this version)...

```
+       -       DROP    SWAP    DUP     ROT     -ROT
1+      1-      2+      2-      I       C@      C!
@       !       OVER    2DROP   >R      R>      AND
OR      NIP     2DUP    R@      CELL+   CELL-   CELLS
2*      2/
```

LITERAL numbers and CONSTANTS are also optimized.

If OPT finds that, while compiling, any sequence of at least 3 of these words is encountered, it creates code that will, at runtime, cache stack items in the CPU data registers. D1, D2, D3, D4 and D7 are utilized for this purpose (permitting up to 5 stack items to be cached). The registers are not blindly loaded; rather, a "load as needed" algorithm is implemented.

Let's consider the sequence: SWAP ROT +

The normal JForth compiler (w/ MAX-INLINE set to 12) will produce...

```
move.l  (dsp),d0        code for SWAP...
move.l  d7,(dsp)              "
move.l  d0,d7                 "
move.l  $4(dsp),d0      code for ROT...
move.l  (dsp)+,(dsp)          "
move.l  d7,-(dsp)             "
move.l  d0,d7                 "
add.l   (dsp)+,d7       code for +
```

The code produced by the new compiler for the same high-level statements will be...

```
move.l  (dsp)+,d1
move.l  (dsp)+,d2
add.l   d1,d2
```

The first two instructions in the optimized example simply load stack items into the CPU, where the '+' opcode (add.l) processes them.

What happened to the SWAP and ROT operations? Well, the optimizer factors them in at COMPILE time. For example, no code needs to be actually created for a SWAP because the new compiler keeps track of which register currently holds the top item, where any second is, etc. Therefore, after a SWAP, the compiler will simply create opcodes that reference whichever register holding the correct data.

It should be pointed out that the new compiler does not specifically look for the sequence 'SWAP ROT +' to do this... it finds ANY SEQUENCE (of 3 or more) of the above words and puts together optimized code. This means that optimized code will be created if 'SWAP ROT <anything>' is compiled, as long as <anything> is in the above list.

Note that the above code leaves register D2 holding the topmost stack-item and register D7 containing the second one. This is radically different from the normal JForth register usage (D7 holds TOP-OF-STACK, other items in memory on data stack), but as long as we continue to compile words in the above list, the new compiler will account for the non-standard parameter locations.

However, if the new compiler finds that no more optimizations can be done, it will put things back to normal by 'flushing the cache'. For the above example, it would do this by appending the following code...

```
move.l  d7,-(dsp)      push out second item...
move.l  d2,d7          put top item in normal place
```

This code is specific to this example; other combinations will probably leave the top item in other registers (or cache a different number of items) and will require different but similar cleanup code.

PROFILE - Performance Analyser

Ask most programmers and they will tell you that "most programs spend 80% of their time in 20% of their code." If you are going to optimize a program it would be nice to know which 20% to optimize. PROFILE is a performance analyser that will interrupt your code while it is running and keep track of where it is spending its time.

To compile PROFILE, enter:

```
INCLUDE? PROFILE JDEV:PROFILE
```

Now lets write a small program and analyse its performance. Enter in a file and INCLUDE the following:

```
ANEW TASK-TESTPRO

: MOOP    12 0 DO LOOP ;
: BOOP   123 0 DO LOOP ;
: GOOP     1 0 DO LOOP ;

: FOOP
   0 DO MOOP BOOP GOOP
   LOOP
;
```

The words MOOP BOOP and GOOP will take different lengths of time to execute because they loop different numbers of times. Lets pretend we don't know which one is the longest. PROFILE works like MEASURE. It will analyse whatever follows on the command line. Enter:

```
PROFILE  5000 FOOP
```

You will see a short report on how PROFILE is setup then a message that PROFILE has begun. PROFILE is now running your program and interrupting it occasionally. During the interrupt it looks on the return stack to see where the program was executing. To keep track of where this was, PROFILE divides your program into slots. The short report tells you the starting and ending address of the slots and the size of each slot. When an interrupt occurs while your program was in a slot, that slot's counter is incremented. When PROFILE is finished, it analyses the slots and reports on which slots took the most time. It will give you the slot number, its address range, how many interrupts fell into that slot, the percentage of total time, and the percentage of the slots analysed. Check the number of interrupts that you collected to make sure you got enough, at least 10 in the least reported slot. Your program should probably execute for at least a few seconds. On a fast Amiga you may have to do more loops like 20,000 instead of 5,000.

Look at the report from PROFILE. Look at the leftmost column to find the slot number with the highest percentage. Lets assume, for example, that it is 106. To find out what words are in that slot, enter:

```
106 PF.WHO  \ or whatever slot you want
```

Notice that probably all the words in TESTPRO are in that slot. To narrow our analysis down, enter:

```
106 PF.ZOOMIN
```

This will zoom in to just the words in slot 106. Now rerun PROFILE.

```
PROFILE  5000 FOOP
```

Look to see, what slot is now the highest and use PF.WHO to see what words are in there. You will probably find that we are spending most of our time in BOOP which is no surprise.

Remember the first time we ran PROFILE. There was another slot that showed significant activity. Let's zoom back out and rerun PROFILE to see which one that was.

```
PF.ZOOMOUT
PROFILE  5000 FOOP
```

Use PF.ZOOMIN like before, rerun PROFILE. then use PF.WHO to find the culprit. It will probably be ((?DO)) which is a primitive part of a DO LOOP.

PROFILE - Glossary

PF.ZOOMIN (slot# -- , adjust slots to detail a slot)

PF.ZOOMOUT (-- , adjust slots to full dictionary)

PF.WHO (slot# -- , print what words are in that slot)

PF-INT-RATE (-- addr , variable holding desired intrs/sec)

PF-%-REPORT (n -- , threshold percentage to report slot)

PROFILE (<executable_forth_statement> --)

PROFILE - Discussion

The technique that PROFILE uses has some good and bad aspects. One problem that can occur is that if your program loops at the same rate as the interrupt occurs, you will always appear to be executing the same small piece of code. To prevent this, try changing the interrupt rate by setting PF-INT-RATE to a different value. Do not go above 1000 or you will bog down the system.

There is another technique for profiling that does not use interrupts. This other technique requires you to compile the program with special profiling code built in. The results can sometimes be more accurate, but program execution can be effected by the extra code. Kernel words that are called in such a system are reflected in the times for the application level words that call them. This may be desirable. A disadvantage of the other system is that execution within a word cannot be analysed. Using PROFILE, a single word can be analysed to see which part is slowest by zooming in.