Chapter 6
Debugging

Introduction

Debugging is the process of finding out why a program doesn't behave as it should. The bad behavior can be anything from printing the wrong answer to crashing the machine. Finding the source of bugs quickly is a skill that one develops through experience. One can easily spend most of one's precious time debugging. It is worth developing these debugging skills so that one has more time for developing and enjoying code.

In this chapter, we will discuss various debugging techniques as well as the various HForth tools specific to debugging.

Tools Overview

These tools can be used together to debug your programs.

DEBUG is HForth's source level debugger. It is described in detail later in this chapter.

DUMP will display the contents of memory at a given address. It is useful for examining data structures like the dictionary, and strings. The stack diagram is:

```
DUMP ( address count -- , dump memory )
```

FILE? is handy when you are examining unfamiliar code and need to know how a word is defined. Enter FILE? followed by the name of the word to study.

MAP can sometimes reveal problems related to dictionary size.

TRAPS.ON sets the 68000 exception vectors so that errors like zero divide and odd addressing are trapped. Activating TRAPS will prevent some of the common crashes. TRAPS can be turned off with NOTRAPS . TRAPS are automatically installed when you start HForth. There's not much reason to turn traps off (one strange example we came across was to simply test another debugger, MacsBug).

UNRAVEL when executed, displays the return stack as a sequence of subroutine calls. I will pause here to mention that RAVEL and UNRAVEL mean the same thing. Look it up in a dictionary if you don't believe me. UNRAVEL is useful in a large program when you want to know who is calling a word. Simply put a call to UNRAVEL inside the word that is being called.

Debugging Hints

Every debugging problem is unique and has a different solution. There is no entirely rational way to find a bug quickly.

Since most debugging is best done intuitively, I have designed this section as a random assortment of hints and questions that may lead you to an answer. At some point in this process you will probably go "Ahah!" and be back in action.

The first step in debugging is to come up with a fairly concise description of the bug. An example might be, "My program is printing one too many numbers." Another might be, "Whenever I click the left mouse button, after drawing a rectangle, I crash." Once you have a concise description, you can proceed.

ISOLATE THE ERROR. Try to figure out exactly which word is not doing the right thing. Execute each word in the offending area individually to make sure it matches its stack diagram.

WALK THROUGH YOUR CODE. Often a step by step analysis of the offending code is the best way to spot where things go wrong.

CHALLENGE YOUR ASSUMPTIONS. If the code is crashing, obviously something that you assumed was OK is not. Reexamine that code that couldn't possibly be the problem. I have known people to systematically reject all of the code as being the possible source of the error, which leads them to the contradictory statement that the bug cannot exist but it does!

DOES YOUR CODE CRASH AT COMPILE TIME? In other words do you get an error while you are INCLUDing a file. If this is true then you might have a syntax error like a missing THEN or a missing ; . You should also check your code for any IMMEDIATE words since they can execute at compile time and cause problems if they are buggy. To find out where the problem is occurring, turn **ECHO ON** , then include the file. The contents of the file will echo as it is being compiled.

IS THE COMPILER NOT FINDING WORDS THAT YOU THINK IT SHOULD? Check for misspelled words, or a missing **(** or **;** . You might also try **REHASH** or **HASH.OFF** in case there is a problem with **HASHING** that is related to something you're doing (this is very rare!). You should also enter **ORDER** to see what vocabularies are being searched.

Remember that if you are using ANEW, you may have to FORGET a file explicitly in order to INCLUDE something underneath it. If you have just added an INCLUDE? this might be the problem. You compile the new code then ANEW comes along and FORGETs it! You may also have overwritten part of the dictionary. **WORDS**, which shows you the directionary, will help you find something like this.

START OVER AGAIN. Sometimes you may be doing things right but are suffering from the lingering effects of a bug that you fixed an hour ago. The old bug may have randomly poked into memory or set some variables wrong. This can be wildly unpredictable. Do a BYE then rerun HForth. You may even want to reboot.

DOES IT WORK THE FIRST TIME THEN NEVER AGAIN? This can be from a number of causes. When you first compile and run, your variables are zero. What are they the second time you run? Did you do an **ALLOCBLOCK** or an **FOPEN** at compile time as you INCLUDE? (Yuck!) If so what happened to that memory or that file?

The best way to handle these problems is to have an "init" word that does all required initialization, opens all files and windows needed, allocates all memory needed, initialize all data structures, etc. Then have a TERM word that cleans up all this stuff. Make sure you call the "term" word before recompiling the code.

DID IT USED TO WORK BEFORE YOU MADE SOME CHANGES? Try to determine exactly what you changed. It is good to test periodically before you go too far along the primrose path. Keep frequent backups.

USE .S LIBERALLY. A few .S calls in your code can be very illuminating. This is not so important now that we have the source level debugger, but good Forth programmers get into the habit of liberally sprinkling their code with stack prints. It can't hurt!

ARE YOU OFF BY ONE? Remember that **DO LOOP**s go *up to* but *don't reach* their limit. Also remember this old problem: You have a row of boxes labelled 15 through 25 consecutively. How many boxes do you have? Ten? Wrong!! Eleven boxes. Count them. This is known as the old "off by one" problem.

Enough random chatter, now let's see how to really debug code.

Source Level Debugger Tutorial

The HForth *Source Level Debugger*, **DEBUG** , allows you to single step through your code. This shows you exactly what each word is doing. You can see the result of every DUP , SWAP , ROT , whatever, as it happens.

Compiling with DEBUG{

Let's try out the debugger and see how it works. First we need to load the debugger. It is a good idea to do this before you load the program you are working on. To load the debugger, enter:

```
INCLUDE HSYS:DEBUGGER
```

To debug a word with DEBUG, we need to compile it with the debugger on. You use **DEBUG{** and **} DEBUG** to turn on and off the debugger. When the debugger is on, any word you compile has special information compiled into it for the debugger to use. This makes the code much larger and slower so we only want to do this when debugging. After you finish debugging, make sure you recompile without the debugger on.

Let's define a simple word from the keyboard and then step through it. Enter:

```
DEBUG{  ( turn on debugger )
: SUMN  ( N -- , print N*[N+1]/2 )
        DUP 1+
        * 2/
;
}DEBUG ( turn off debugger )
3 SUMN .
```

After the last line, you should see the answer 6 printed.

Examining Code

We can now single step through this program to see how it arrived at its answer. Enter:

```
0SP  ( clear stack if not already clear )
3 DEBUG SUMN
```

You should see a display that tells you you are entering SUMN.

```
Entering:  SUMN >>>>>>>>>>>>>>>>>>>>
------------------------------------------------------------
[1] 3
3DD46 : ------(   DUP                                  |?_
```

You will also see a stack diagram with the 3 on the stack. The number in brackets tells you how many things are on the stack. Below that will be a hexadecimal address and the word DUP. This tells you that DUP is about to execute. The cursor will be at the far (|**?**_)right of the screen waiting for a key. Press the space bar.

```
<SPACE>
```

You should now see 3 3 on the stack which is what we would expect after DUP finishes. We are now ready to do the 1+ . Keep hitting the space bar until we finish the word. At the end we saw an RTS instruction. This meant that we were about to do a 68000 RTS, Return From Subroutine, instruction. This corresponds to the semicolon at the end of a word.

Now let's add some complexity by calling this word from another. Enter:

```
DEBUG{
: DOSUMS ( -- )
        100 5 DO
                I DUP . SUMN . CR
        LOOP
;
}DEBUG
DEBUG DOSUMS
```

Keep hitting the space bar until you see SUMN ready to execute. Hit the space bar again. You should now see a 15 on the stack. This is the result of SUMN for N=5. You are now ready to execute the . instruction. Hit <**SPACE>** and notice that the number 15 appeared to the right of the debugger display.

Now let's go through the loop again but this time let's reexamine what SUMN is doing. Keep hitting the space bar until SUMN reappears in the command area. If you miss it just go around again. At this point we have two choices, we can hit space and skim over SUMN like before, or we can dive into SUMN and see it work. With SUMN ready to execute, hit a **D** key or hit the return key. The display should now say that you are in SUMN. The display of commands will indent to show that you are nested inside another word.

To verify that we are indeed in SUMN, hit the W key. This will display "Where" we are. The display will indicate that DOSUMS has called SUMN.

If you now continue to hit spaces, you will move through SUMN and eventually return to DOSUMS. Thus <SPACE> will advance though a word while <RETURN> will dive into a nested word.

Stopping with a Breakpoint

Now hit the space bar until we get back to the SUMN command. Let's suppose we wanted to move more quickly through the loop. It would be nice if we could just stop before each SUMN, see what N was then zoom forward to the next one. To do this:

```
Hit the B key.
```

You should see a message that a "Breakpoint" was set. This is like putting a stop sign by the call to SUMN. You can have up to 16 breakpoints. To advance quickly:

```
Hit the G key.
```

A **G** tells DEBUG to "Go" until it hits the Breakpoint or finishes the word. Notice that the answer was printed to the right of the DEBUG display. Hit G a few more times. You can watch the value of N on the stack increasing. You will also see the answers appearing for each time around the loop.

If we get tired of just doing one at a time, we can skip past the breakpoint several times.

```
Hit the # key.
Enter 7 , hit <RETURN>
```

Notice that we saw seven answers appear before it stopped again. If we want to just finish the word, we can "Clear" the breakpoint then "Go".

```
Hit the C key.
Hit the G key.
```

You should now advance through the word at a rapid pace until you finish.

Note: You can also specify a breakpoint by before you run the code using **BREAKAT** which is described in the Debugger glossary.

Stopping with Control-D

When code is free running and you decide you want to start debugging, you can interrupt the code with a **Control-D**. Enter

```
DOSUMS  ( let it print a few )
Hit ^D  ( that's Control-D)
```

(Control-D can be hit by HOLDING DOWN the <CTRL> key, then hitting a D, then releasing the <CTRL> key.)

You should now be back in the debugger. Hit **W** to find out which word you are in.

You can now continue debugging as before.

Debugging a Large Program

If you want to debug a large program from a file, place the DEBUG{ }DEBUG commands around the include statement. For example:

```
DEBUG{
INCLUDE my-file  ( compile one of your programs )
}DEBUG
DEBUG my-word  ( now debug it )
```

If you don't want to have the entire file in DEBUG mode you can place DEBUG{ }DEBUG around individual words in the file. You can also use them within a word since they are defined as **IMMEDIATE**.

Debugging a Turnkeyed Program

The Debugger will work with Turnkeyed programs. Read the section of this manual on TURNKEY before attempting this test. To debug a Turnkeyed program, you must use **DEBUG.START** and **DEBUG.STOP** instead of the DEBUG command. Here is an example of a debugging a turnkeyed program. First make a copy of HMSL and call it TEST, then run it without initializing.

```
DEBUG{  ( compile with debug )
: TEST  ( -- , turnkey this puppy )
    DEBUG.START  ( start debugging )
    ." Answer = " 2 2 + . CR  ( fancy program eh? )
    DEBUG.STOP   ( close window )
;
}DEBUG

'C NOOP  'C TEST  'N  NOOP  TURNKEY
BYE
```

Now double click on TEST. You should see a window open and a normal debugging session will follow. Note: Since Turnkey removes the Forth headers, you cannot use the **F** command (moves you back into the Forth interpreter temporarily) in the debugger. You may, therefore, want to assign custom functions to keys 7,8,9 using DEBUG.USER.7 , etc. Don't forget to recompile and returnkey without the debugger when you are finished. The debugger will add at least 15K to an application and makes it run *much* slower so don't release a program with the debug stuff in there.

IMMEDIATE Words and Locals

You will notice that IMMEDIATE words will also show the following word. This is so that words like ' , ..@ , ." , IS , NEW: , etc. will print the word they are operating on. For ." you will not see the entire string, just the first word. Other IMMEDIATE words like IF ELSE and THEN will also show the following words even though this is not so important.

Source Level Debugger Glossary

These words are supported by the Source Level Debugger - DEBUG.

BREAKAT (<word> [<command-in-word>] -- , set breakpoint)

This command can be used to set a breakpoint at a given command in a given word. Consider the following example.

```
DEBUG{
: FOO ( N -- 1 )
   DUP 1+ SWAP - ;     }DEBUG
\  Break  ^ <- right there before SWAP in FOO
BREAKAT FOO SWAP
FOO
```

If you call BREAKAT without a command specified then it will set a breakpoint at the **entry point** of a word. Use NOBREAKS to clear all breakpoints or hit 'C' in the debugger.

DEBUG{ (-- , compile with debug on)

}DEBUG (-- , compile normally)

DEBUG (<word> -- , debug the word whose name follows)

DEBUG.BREAK (-- , enter debugger when encountered)

This word can be sprinkled through your program to force breakpoints.

DEBUG.START (-- , open debugger window and start debugging)

DEBUG.STOP (-- , close window and exit debugger)

DEBUG.USER.7 (-- , deferred action for hitting '7')

You can specify your own function to be executed when a '7' digit key is hit. This allows you to use your own dump and diagnostic routines from the debugger, even when cloned.

```
: MYDUMP    ( -- , dump stuff of interest to user )
    ." my-var = " MY-VAR ?  cr
;
' MYDUMP  IS DEBUG.USER.7
```

DEBUG.USER.8 (-- , deferred like DEBUG.USER.7)

DEBUG.USER.9 (-- , deferred like DEBUG.USER.7)

NOBREAKS (-- , Clear all breakpoints except USER.BREAK?)

USER.BREAK? (address -- break? , user defined break)

This is a deferred word that will be passed the address of the next Forth word to be executed. The word can then decide whether to break into the debugger. This is handy for making logical breakpoints. You could, for instance break if a variable was out of range. Here is an example user break.

```
}DEBUG  ( turn off to avoid recursion )
: MY.BREAK ( address -- break? )
        drop
        VAR1 @ 100 >
;
' MY.BREAK IS USER.BREAK?
```

Debugger One Key Commands

When you are in the debugger,you can hit the **?** key for a list of available interactive commands. I recommend playing with these commands to see what they do.

Command Descriptions

```
Information
w - where?, who called who
6 - 680x0 register dump
      Dump all 68000 Data and Address registers.
      See assembler for more information.
m - memory dump from address on stack
      This will treat the top of stack as an address
      and dump the following 32 bytes.
s - regular stack dump
      This just calls .S
```

```
r - return stack hex dump
h - here 256 dump
      Accurately display what's at HERE and on the PAD.
      The PAD is at HERE 128 +.

Action
f - forth, interpret one line
      This will put you in Forth.  You can then
      check variables. Check other words.
      Do whatever.  Enter a blank line to finish.
      The programs stack will be unaffected by your
      actions. Remember, Forth itself is your most
      powerful debugging tool.  (This feature is not
      available in Turnkeyed programs.)
x - drop one number from stack
n - push a number onto stack
+ - add a number to top of stack
      These last three commands can be used to
      alter the stack contents.
Bases
1 - decimal, set BASE to 10
2 - binary, set BASE to 2
3 - hex, set BASE to 16

User Keys
7,8,9 - Execute DEBUG.USER.7,8,9

Control
b - set the breakpoint here
      Set a breakpoint so that you will stop here
      again if you ever come back.
c - clear the breakpoint
# - enter # breaks to skip
u - up, continue until return
      Finish the current word, go back into
      DEBUG when you return to the calling word.
j - jump over next instruction, don't execute it.
z - set user.break? to 0= , disabled
l - look but don't touch
      The program will continue while displaying
      debugger information.  It will not stop
      for Key Commands until you press a key.
g - go, continue execution without debugging
<SPACE> - single step on same level
<CR> or d - dive down into word
q - quit. Aborts. Handy if continuing will cause a crash.
```

I hope that this debugger will give you a window into your code. It can also be a good learning tool for Forth beginners.