Dr. Dobb's Toolbook of 68000 Programming

The Editors of Dr. Dobb's Journal of Software Tools

A Brady Book
Published by Prentice-Hall Press
New York, New York 10023
# Table of Contents

Introduction

I. INTRODUCTION TO THE 68000 FAMILY

1. MC68000 Family History, Design and Philosophy
   *Daniel Appleman*  
   Page 5

2. The 68000 Instruction Set
   *Daniel Appleman*  
   Page 21

3. The 68000 Family
   *Daniel Appleman*  
   Page 37

II. DEVELOPMENT TOOLS FOR THE 68000 FAMILY

4. Bringing Up the 68000: A First Step
   *Alan D. Wilcox*  
   Page 53

5. Motorola's Tutor Firmware
   *Alan D. Wilcox*  
   Page 65

6. Tiny BASIC
   *Gordon Brandly*  
   Page 73

7. Comfort: A Faster Forth
   *Alexander Burger and Ronald Greene*  
   Page 107

8. A Forth Native-Code Cross-Compiler
   *Raymond Buvel*  
   Page 123

9. A 68000 Forth Assembler
   *Michael A. Perry*  
   Page 159

10. A 68000 Cross-Assembler
    *Brian Anderson*  
    Page 175

III. USEFUL 68000 ROUTINES AND TECHNIQUES

11. 68000 Coding Conventions
    *Jan Steinman*  
    Page 285

12. A Simple Multitasking Kernel for Real-Time Applications
    *Nicholas Turner*  
    Page 303
13. A Commercial Multitasking Kernel  
   Steve Passe  
   321
14. A Pseudo Random-Number Generator  
   Michael P. McLaughlin  
   335
15. Generating Nonuniform Distributions of Random Numbers  
   Chris Crawford  
   337
16. The Worm Memory Test  
   Jan Steinman  
   341
17. Improved Integer Square Root Routine  
   Jim Cathey  
   351
18. A Mandelbrot Program for the Macintosh  
   Howard Katz  
   357
19. Improved Binary Search Routine  
   Michael P. McLaughlin  
   389

About the Authors  
   391
The Worm Memory Test

Jan Steinman

The Worm is a memory test that has the unusual characteristic of being able to overlay itself while it's running. It makes special use of the 68000's instruction prefetch register.

No, the Worm Memory Test is not a method for quantifying the mental retentive powers of long, cylindrical invertebrates. It is a test that could help diagnose certain types of computer memory errors. Worm (see listing) uses a dynamically executing program as the actual test data. Unlike previous memory test programs of this type, this one has a special twist: It can overlay itself while it is executing, thanks to the MC68000's prefetch register.

Some Fetching Facts

Never heard of the prefetch register? To understand how the memory test works, it might help to review the way the MC68000 fetches and executes instructions. The MC68000 uses instruction pipelining in order to speed execution. There is, in effect, a 16-bit register between the data bus and the instruction decoding logic. (The MC68010 has 32 bits of prefetch and the 68020 has a 64-entry instruction cache, but the results should be similar.) When an instruction is executed, the opcode for that instruction is first loaded into the prefetch register (often while the previously fetched instruction is being executed), then the instruction is moved into the instruction decoding register, where it is executed. The net effect is that the processor usually has a handle on the next thing it is supposed to do.

Prefetch works fine most of the time, but it does slow things down during certain operations. If the instruction being executed causes a nonsequential instruction to be executed, execution may be either faster or slower. In the case of a conditional branch instruction, a branch taken is quite fast because the prefetch register already holds the displacement that must be added to the program counter in order to fetch the next nonsequential instruction. A branch
not taken, however, will be a little faster if it is a short branch, because the
next instruction is already in the prefetch register and the two clocks needed to
add a displacement to the program counter can be saved. The worst case
happens when a branch is not taken and the branch displacement is 16 bits. In
that case, the processor has useless information in the prefetch register and
must flush that information before it can fetch the next instruction.

Other nonsequential instructions cause an immediate flush of the prefetch
register and use an extra four clocks simply to restart the pipeline. One
exception is the decrement-and-branch instruction, which, like the taken
branches, benefits from having the branch displacement handy. (The MC68010, with its 32-bit prefetch register, actually executes many 16-bit
instructions out of the prefetch register if they precede a decrement-and-branch
instruction.)

How the Worm Crawls

Worm depends on these characteristics of pipelining in order to overlay
itself while it is running, but it needs some management and control in order
to be useful—a Worm on the loose would quickly destroy all memory!
Besides Worm, a complete memory test requires two additional parts: an
initialization sequence and a routine for controlling Worm and reporting its
findings.

The initialization routine, Init, has some special characteristics and
includes most of the system dependencies. It is executed only once—at the
beginning—and is therefore throwaway code. That is why it is placed last;
Worm actually crawls right over its initialization code in this implementa-
tion. The registers are set up to the specifications of Worm and several
important system functions are performed. In particular, it is important that
page faulting does not occur in systems that support virtual memory; if
special hocus-pocus is needed to turn off interrupts, it should be done here.

Manager exercises control over Worm and is responsible for communi-
cating errors it discovers and for displaying progress messages if desired.
When Manager is entered upon completion of a Worm pass, it must decide if
it has been entered because of an error or simply as a point of control. If there
has been an error, Worm is no longer runnable, so Manager will have to
report the error and terminate. If no error is detected, Manager must check the
progress of Worm to keep it from consuming all memory. At this point,
Manager can decide that enough memory has been checked to warrant a
progress report of some kind.

The real heart of the whole thing is, after all, Worm. Worm simply
replicates itself, one longword lower in memory, while comparing the new
copy of itself against the original, which never executes. Worm may be the
heart of the memory test, but the three instructions starting at Crawl are
where the magic happens. This loop starts at the beginning of Worm, and
copies the first longword down to Worm-4. It continues with each additional
longword, until it gets to the longword at Crawl+4, which is a dbne
instruction with its 16-bit displacement. The preceding move.l and cmp.l have already been copied down.

At this point, it becomes a little difficult to keep track of what is data and what is code. When the move.l is in the instruction decode register, ready to be executed, the following cmp.l is in the prefetch register, waiting its turn to be executed. When the move.l at Crawl executes, it moves the dbne instruction into the location it and the following cmp.l are currently occupying. The processor has no way of knowing it has just invalidated its prefetch register, so it continues—moving the cmp.l instruction into the instruction decode register and moving the following dbne into the prefetch register. The cmp.l executes, comparing the dbne just moved with the original while moving the branch displacement for the dbne into the prefetch register.

Assuming the compare was successful, the dbne executes, decrementing d0 and branching backward 4 bytes to where the move.l used to be. The prefetch register is flushed because of the branch, so the value at that location is loaded into the prefetch register and immediately into the instruction decode register. But what is loaded? A copy of the dbne, complete with the same negative displacement value. The condition codes have not changed, and the count register d0 should not be anywhere near 0, so the copy of the dbne gets executed identically to its predecessor, which still resides in the next longword. The dbne copy branches to the move.l copy, and the loop continues moving the code down 4 bytes. (See table.)

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crawl-4</td>
<td>move.l (a0)+,(a1)</td>
</tr>
<tr>
<td>Crawl-2</td>
<td>cmp.l (a1)+,(a2)+</td>
</tr>
<tr>
<td>Crawl</td>
<td>dbne d0,-6</td>
</tr>
<tr>
<td>Crawl+2</td>
<td>dbne d0,-6</td>
</tr>
<tr>
<td>Crawl+4</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 16.1 The test in action.

When the count register d0 underflows, the dbne copy drops through, interrupts are enabled, Worm's dynamic image pointer a5 is adjusted to point to the new Worm copy, and the Worm reports back to Manager. Note that none of the Worm code is ever executed before it has been compared and verified.

It is vitally important to disable interrupts when the move.l overlays itself and the following cmp.l. An interrupt at this point causes the prefetch to be flushed when the interrupt is serviced. Upon return from the interrupt, the displacement part of the dbne (hex FFFA) will be fetched as an instruction. This will cause a "line 1111 emulator exception" message unless your system has a coprocessor with an ID code of 7, but either way Worm will be broken and the memory test will fail. And of course, it is important that the length of Worm remains a multiple of 4 if you decide to modify it!
But What Good Is It?

I originally developed the MC68000 Worm Memory Test for an embedded processor application that was having dynamic-RAM refresh problems. It was discovered that conventional RAM tests, which move smoothly up through consecutive addresses, were masking the problem by unintentionally providing software refresh. The test is not long enough to cause a complete cycle of all a dynamic RAM's row-address-strobe (RAS) lines and was able to help diagnose the problem.

In the form presented, this implementation is useful primarily as an illustrative example of position-independent coding, modular design and, of course, a unique use of the prefetch register. It could be put to practical use in several ways.

The best use of the memory test might be to have it running continuously as a very low priority task. Manager would have to take some of the responsibility of Init by allocating test memory and restarting Worm when it finished testing a buffer. The interrupt disabling code may be simpler on systems without virtual memory (on the Commodore Amiga, for example, it is a simple memory store).

Virtual-memory systems would also need to add code to branch around the interrupt disabling code on the copy of the first longword only, which would allow the memory test to generate page faults whenever it first crosses a page boundary. To make it practical in such systems, Manager would have to access the memory-management hardware in order to map faulty virtual locations to broken chips.

The Worm routine itself can hold much more code if desired. I originally had much of Manager's decision code in Worm, which did speed it up but at the expense of simplicity. In a message-based system, such as the Amiga, Manager could be totally deleted. Worm could contain all the task code, merrily crawling through any available RAM it could find and sending error reports through intertask messages—all with minimal impact on the user.
Listing 16.1

The Worm Memory Test

* Author: Jan W. Steinman, 2002 Parkside Ct., West Linn, OR 97068.

* The Worm memory test has three parts: Init sets up the registers for the
  * Worm. The Display Manager interacts with the Worm each pass and periodically
  * Displays the Worm's progress. The Worm itself Worms itself through memory,
  * from high to low, checking memory against a copy of itself. The Droppings
  * form a pattern through memory when the test is complete.

* This version runs on the Tektronix 4044 under uniflex. System-dependent code
  * is mostly segregated to the Init, Display, Disable and Enable routines. Two
  * instructions in the Worm routine are system-dependent, for enabling and
  * disabling interrupts.

* Register usage:
  * D0: scratch register.
  * D1: scratch register.
  * D2: scratch register.
  * D3: scratch register.
  * D4: address mask for determining if time to show progress.
  * D5: base of memory area under test.
  * D6: length of Worm in long words.
  * A0: scratch register.
  * A1: scratch register.
  * A2: scratch register.
  * A3: pointer to Display manager for position independent access.
  * A4: pointer to permanent Worm image for comparison.
  * A5: pointer to crawling Worm image.
  * A6: scratch pointer.

* These included files contain system definitions and interrupt signal
  * numbers for the Uniflex operating system. Don't bother to list these.

  CPT   .1ix
  DEFINE (This makes all labels global for debug)

* Set D_MASK with the bits that are zero at each progress report.

| D_MASK | EQU | 00000000 | Report each boundary passed.
| REL_SII | EQU | 4 | Relocation is four bytes at a time.
| MEM_SII | EQU | #2000+REL_SII | Test a 32K chunk.
| DISABLE | EQU | 0 | Trap number for Disable routine.
| ENABLE | EQU | 1 | Trap number for Enable routine.
| CR | EQU | 0D | Carriage return.
| LF | EQU | 0A | Line feed.

* Uniflex will not allow intersection math, so put all the code in the DATA
  * section, and don't use TEXT or NEQ at all.

  DATA EQU * | Assemble into writable data section.
**** hexadecimal ******

* hexadecimal converts a long word to eight ASCII hexadecimal characters.
* This routine is machine and OS independent. It uses a simple table look-up
  to generate the hexadecimal string.

  Entry:  d0 -- Long word to be converted to hex.
  a0 -- Pointer to buffer where hex characters will go.

  Exit:  d2 -- -1. (Just in case someone cares)
  d0 -- unchanged.
  +d0 -- points to eight ASCII characters.

  Users:  d3 -- nybble mask; constant 3DF.
  d2 -- nybble counter.
  d1 -- current nybble to convert is LSN.

CharTab: DC.B  '0123456789ABCDEF' Where we keep our hex characters.

hexadecimal
  move.1 #7,d2  Bytes to make - 1.
  move.1 #5F,d2  Nybble mask.
  NextLoop rpl.1 #4,d0  shift the next nybble into the LSD.
  move.1 d0,d1  make a copy for masking.
  and.1 d3,d1  mask out all but least significant nybble.
  move.1 CharTab[d1+(a)],(a)
  dbra d2,NextLoop  Repeat until done, and when done,
  rts  hit the road, Jack.

**** Manager ******

* Manager checks the Worm's progress, and periodically reports to the Display.
* This routine is also entered if an error is encountered.

  Entry:  d0 -- W_LEN or complement of pass count if error, else -1.
  a0 -- text address pass/fail value.

  Exit:  via direct jump to worm at (A5).

  Users:  d1, d2, d3, d0, d7, a0, a0

  Stack: one level, plus needs of Display.

ErrMsg  DC.B  CR,"Worm reports memory error at ".
ErrAddrMsg  DC.B  '00000000 on pass '.
ErrCountMsg  DC.B  '00000000 on pass '.
E_SIE EQU  "-ErrMsg.
DoneMsg  DC.B  CR,"Worm tested memory from ".
DoneMsgAddrMsg  DC.B  '00000000 through ".
DoneEndMsgAddrMsg  DC.B  '00000000 successfully. ".
D_SIE EQU  "-DoneMsg.
PassMsg  DC.B  '00000000 on pass ".
P_Count EQU  "-PassMsg.
EVEN  (Stay on legal instruction boundary.)
Manager int.w  d0  Was loop exited by error, or countdown?
  bpl.s  GetErrMsg Error, go report it.
  cmp.1 a5,d6  Countdown, so are we done yet?
  beq.s  GetDoneMsg Yes, Go finish up.
  move.1 a5,d3  No, put the new source where we can.
look at the button bits; on boundary?
Yes, set up for progress report.
No, keep on cruisin'...

Finish up. Get the pointer to start addr.

GetDoneMsg lea DoneMsgAddrMsg(pcl), a0
move.l a1, d0 and the value to plug in.
bpl Hexadecimalize which gets converted, likewise, get
lea DoneEndMsgAddrMsg(pcl), a0
move.l #MEM_SIZE, d0 the end address and its value.
bpl Hexadecimalize also converted to hexascii.
lea DoneMsg(pcl), a0 Get pointer to complete done message.
move.l #S_SIZE, d3 length of the done message.
pea Exit(pcl) push a return pointer.
bra.s Display and go display the message.

Make an error report. Get message ptr.

GetErrMsg lea ErrCountMsg(pcl), a0
sub.l #M_LONG-1, d0 convert worm count to a pass count,
bpl Hexadecimalize make it hex for Display.
lea ErrAddrMsg(pcl), a0 Get addr of ASCII error addr.
move.l #-4, d0 get bad long addr to display,
add.l a1, d0 less four to account for postincrement,
bpl Hexadecimalize make it hex for Display.
lea ErrMsg(pcl), a0 Get pointer to whole err msg.
move.l #E_SIZE, d3 the size for the write.
pea Exit(pcl) push a return pointer.
bra.s Display and Display the message.

Progress report. Get message ptr.

PrintMsg(pcl), a0
load the checked address.
lea HexaDecimalize make it hex for Display.
sub.l #0, d0 Regain pointer to the message.
move.l #D_SIZE, d3 get the size for the write.
pea (ab) push a return ptr to the new Worm,
and drop through into Display.

Display

Display is an implementation-dependent scheme for reporting the Worm's
progress. Upon entry, A0 contains a pointer to a string to Display, and D3
contains the length of the string to Display.

Entry: d3 -- number of bytes to display.
a0 -- address of a string to display.

Uses: d0 -- file descriptor of stdout.
a1 -- scratch register for pointing to SysCall param block.

Stack: as needed by system call.

BEGIN SYSTEM-DEPENDENT CODE

Display move.l d3, -(a7) load the byte count.
move.l a0, -(a7) the actual string pointer.
move.w #write, -(a7) and the system call index.
move.l a7, a0 point to the syscall parameter block.
move.l #1, a0 load file descriptor for stdout.
STI index and write the message.
add.l #10, a7 Remove the params from the stack, and
rts return somewhere.

For lack of a better place to put it, the system-dependent exit code is here.
Exit  STS term Terminate this program. (System dependent.)

****** END SYSTEM-DEPENDENT CODE **********

**** Disable, Enable ****************************************************
* These routines provide the exclusion mechanism for the non-interruptible code
* in Wrm at Crawl. These routines must execute in supervisor state, therefore
* they are executed via the TRAP exception instruction. Enable requires that
* D1 be preserved from the preceding Disable.
*
* Uses: SR -- interrupt mask is raised and lowered.
* d2 -- scratch register for restoring original interrupt mask.
* d1 -- scratch register storage place for old interrupt mask.

****** BEGIN SYSTEM-DEPENDENT CODE **********

Disable: move sr,10. Grab the status register,
and.w #00300,d1 keep only the interrupt bits,
and #00300,er and disable all interrupts
STS cplnt,STGTRAP2,Disable
itr
before entering critical code region.

Enable: move sr,d2 Regain the status register,
or.w d1,d2 reset the previous interrupt level,
move d2,er and enable the proper interrupts
STS cplnt,STGTRAP3,Enable
itr
before entering critical code region.

****** Wrm ****************************

* Wrm is a self-modifying, self-relocating procedure which starts at some
* location in high memory and works its way down to its end address,
* periodically reporting its progress.
*
* The loop at Crawl depends strongly on the 68000 prefetch mechanism. This
* loop will not work on a 68026 machine which has a 64 entry cache, nor on
* most simulators (which often do not bother to simulate prefetch accurately).
* This loop will also not work with the TRACK bit set, and must be protected
* from all interrupts, including page faults in virtual memory systems.
*
* When this loop moves the DBNE long word at Crawl+4, it overlays the MOVEIL
* and the CMPLT at Crawl. The CMPLT is in the prefetch queue, so it gets
* executed even though its memory image has just been clobbered. The DBNE is
* fetched, and its execution flushes the prefetch queue as is the case with all
* branches. Execution continues with the copy of the DBNE just moved, which
* executes again, branching to Crawl+4, the new loop location. Note that the
* loop count gets decremented twice in this scenario, removing the need for the
* usual predecrement before entering the loop.

Entry: a7 -- length of Wrm in long words.
a6 -- base of memory area to test.
a5 -- address mask for display boundary.
a4 -- first long word address of Wrm at present.
a3 -- first long word address of Wrm's original image.
a2 -- display manager's address.

Exit: d0 = W.LONGS complement of pass count if error.
a5 -- entry value less relocation, i.e.: next pass entry value.
a1 -- address pass/fail report value.
THE WORM MEMORY TEST

* Use:
  d0 -- decrementing Worm length.
  a1 -- incrementing COMPARE address.
  a1 -- incrementing TO address.
  a0 -- incrementing FROM address.

* Unused:
  d4, d5, a7, a6.

Worm
move.w d1, d0          Restore the Worm's length.
move.l a5, a0          its starting point.
move.l a4, a2          and its original address.
lea -4(a5), a1          Get the destination for this pass.

******** BEGIN SYSTEM-DEPENDENT CODE ********
  trap #DISABLE Don't interrupt this critical passage!
******** END SYSTEM-DEPENDENT CODE ********

Crawl
move.l (a0)+, (a1)    Move a long word piece of Worm,
cmpu.l (a1)+, (a2)+    and check it against the original,
djne d0, Crawl one long word at a time.

******** BEGIN SYSTEM-DEPENDENT CODE ********
  trap #ENABLE    Allow interrupts - critical section over.
******** END SYSTEM-DEPENDENT CODE ********
sub.l #REL_SIZ, a3    Update the new Worm address,
sup keep the whole thing on long boundary,
jmp (a3) report to the Manager.

* The following pattern (which is notoriously hard on 16-bit dynamic RAM
* memories) gets left in memory and can be checked later if desired.

Droppings
DC.L $55555AAAA   Pattern to be left in RAM.
W_SIZ EQU *Worm   Length of self-relocating code, in bytes
W_LONGE EQU W_SIZ/4 and longer.

******** Init  ***************

* Init performs system-dependent initialization and sets up registers for use
* of Worm and Manager. Init then copies the Worm into the top of test memory
* and starts the Worm crawling.

* Entry: not applicable.

* Exit:
  a5 -- Worm's test image address at top of memory to be tested.
  a4 -- Worm's permanent image address.
  a3 -- Manager routine pointer.
  d7 -- length of Worm in long words.
  d6 -- base of memory area to test.
  d5 -- address mask for testing display boundary.

Only . EQU * This area will be overlaid with the worm.
LogMsg DC.B 'Worm memory tester,'
DC.B 'Sheed1.worm.a-v 1.2 86/05/24 01:44:36 jans Exp 5'
DC.B '5k, 'Memory checked down to locations', OR
L_SIZ EQU +-LogMsg

EVEN
GLOBAL Init

* First, perform some system-dependent initialization: set up the TRAPs needed
to protect the Worm from interrupts, protect the area to be tested from page
faults, and write a welcome message.

BEGIN SYSTEM-DEPENDENT CODE

STD opint, SIGTRAP, Disable
Set up the exception handlers for the
STD opint, SIGTRAP, Enable
interrupt exclusion routines.
STD MemStart, MemEnd, MemEnd
Protect memory image from page faults.
move.l #1, d0
Prepare and write a startup
STD write, LogMsg, L_Blk
welcome message.

END SYSTEM-DEPENDENT CODE

Next, set up registers that will be used by the Worm and Manager.

move.l #D_Mask, d5
Get the Display address boundary mask.
lea Ordys, a1 Load the lowest address to test
move.l a0, d6
Load a data register for comparison,
lea Manager, a3
get the display Manager's address,
lea Worm, a4
the Worm's non-crawling image address,
move.l #MemEnd-M_Blk, a5
and the high-end Worm start address.
move.w #L_START, b7
Get the Worm's length in longs.

Finally, move the Worm to the top of memory to be tested.

move.l a1, a0
Get a copy of Worm's permanent image pointer,
movw a0, a1
its test image pointer,
move.w d7, d0
and its length in longs.
sub.w #1, d0

MoveWorm move.l (a0), (a1) Move, and compare
cmp.l (a0)+, (a1)+
da0, MoveWorm
a long word of the Worm
at a time.

mov.w e0
Exit loop by error, or countdown?
beq Manager
Error, go Report it.
jeq (a3)
Countdown. Start Crawling!

C_Blk EQU
"MemEnd
8 Blk size of non-relocating code.

DO.B MEM_Blk-C_Blk
MemEnd
EQU
ENCOD
END
Init
Set transfer address to the Init.)
From Dr. Dobb's Journal, the highly respected resource for programmers, comes a comprehensive guide to the powerful 68000 family of microprocessors. The editors of Dr. Dobb's have gathered their best articles, updated them, and added new material to create a virtual encyclopedia for using Motorola's 68000 chip.

Inside, you'll find a whole set of development tools, routines, and techniques for getting the best performance possible out of 68000-based machines. Discover great tools for programming: A BASIC interpreter, a FORTH compiler, an assembler, and a native-code cross compiler are a few included here. A section of routines and techniques includes coding conventions, multitasking kernels, random numbers generation, and memory testing.

This reference will serve you over and over in your development of 68000-based software.