Brady

EDITORS OF DR. DOBB'S JOURNAL

DR. DOBB'S TOOLBOOK OF 68000 PROGRAMMING

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

Dr. Dobb's Toolbook of 68000 Programming

Copyright © 1986 by M & T Publishing, Inc.
All rights reserved
including the right of reproduction in whole or in part in any form.

A Brady Book
Published by Prentice Hall Press
A Division of Simon & Schuster, Inc.
Gulf + Western Plaza
New York, NY 10023

PRENTICE HALL PRESS is a trademark of Simon & Schuster, Inc.

Manufactured in the United States of America

1 2 3 4 5 6 7 8 9 10

Library of Congress Cataloging-in Publication Data

Dr. Dobb's toolbook of 68000 programming.

"A Brady Book."
1. Motorola 68000 (Microprocessor)—Programming.
I. Dr. Dobb's journal of software tools for the professional programmer. II. Title: Doctor Dobb's toolbook of 68000 programming.
QA76.8.M67D72 1986 005.265 86-25308
ISBN 0-13-216649-6 (case)
ISBN 0-13-216557-0 (paperback)

Special volume discounts are available by contacting the Special Sales Department, Englewood Cliffs, NJ 07632.

For information about foreign rights, contact M & T Publishing, 501 Galveston Drive, Redwood City, CA 94063.

Educational Computer Board, Tutor, MACSbug, 68000, 6800, 6500, 68008, 68010, 68020, 68881, 68851 and 68452 are trademarks of Motorola, Inc. 8080 and 8086 are trademarks of Intel Corp. Macintosh, Mac Plus and RMaker are trademarks of Apple Computer, Inc. Unix is a trademark of AT&T Bell Laboratories. CP/M is a trademark of Digital Research, Inc.

Table of Contents

Introduction 1

I. INTRODUCTION TO THE 68000 FAMILY

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

2. The 68000 Instruction Set 21
   Daniel Appleman

3. The 68000 Family 37
   Daniel Appleman

II. DEVELOPMENT TOOLS FOR THE 68000 FAMILY

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

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

6. Tiny BASIC 73
   Gordon Brandly

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

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

9. A 68000 Forth Assembler 159
   Michael A. Perry

10. A 68000 Cross-Assembler 175
    Brian Anderson

III. USEFUL 68000 ROUTINES AND TECHNIQUES

11. 68000 Coding Conventions 285
    Jan Steinman

12. A Simple Multitasking Kernel for Real-Time Applications 303
    Nicholas Turner

13. A Commercial Multitasking Kernel 321
Steve Passe

14. A Pseudo Random-Number Generator 335
Michael P. McLaughlin

15. Generating Nonuniform Distributions of Random Numbers 337
Chris Crawford

16. The Worm Memory Test 341
Jan Steinman

17. Improved Integer Square Root Routine 351
Jim Cathey

18. A Mandelbrot Program for the Macintosh 357
Howard Katz

19. Improved Binary Search Routine 389
Michael P. McLaughlin

About the Authors 391

11

68000 Coding Conventions

Jan Steinman

Assembly language programming requires a special sort of discipline, especially if others will be modifying your code. The practical advice offered below will help make the process easier.

So you have your Motorola reference manual, a shiny new machine, and you're tired of running games written in BASIC. What happens next? If you're reading this, you probably aren't the sort of person who is content with the canned programs on the market, and you probably get a big kick out of writing programs that run as efficiently as possible. The answer is, of course, assembly code. But before jumping into coding tricks, let's think a bit about what makes good assembly code.

Practices Make Perfect

Assembly coding practices are ways of dealing with the problem, and the MC68000 fosters a certain style of coding practice. I will present a number of ideas about good coding style, but it is up to you to come up with a set of standards you can live with. It really helps to be consistent, as you will appreciate when you first look through you own "pre-standards" code after adopting some standards.

Modular design is the best way to manage assembly coding projects. Due to its inherently unstructured nature, assembly code often wanders about through branches and jumps, in what is known as "spaghetti code," which is a maintenance nightmare!

Good modular design requires short code sequences that perform a single function in each sequence. Such "modules" should generally be less than one page in length. Also important is having a single entry point and a single exit point, although this is not a firm rule.

Once you have decided to use modular design, a form of structured commentary called the *module specification* is important. It can vary widely in content, but should include, as a minimum:

- a brief statement of the purpose of the module;
- the conditions that need to be set up before the module is entered;
- the conditions produced as a result of the execution of the module;
- side effects, such as register usage, global variables affected, amount of stack needed and so on.

Also desirable, especially for multiuser projects and for modifying existing code, are:

- a revision log, giving the modifier's name, the date modified and a brief description of the changes made;
- a list of global or external references;
- a list of callers of this routine and routines called;
- exceptional termination conditions and the results they produce;
- box drawings of key data structures;
- pseudo-code description of module's operation.

Figure 11.1 shows a template that might be used for a module specification.

```
****    Dinglewalte    ************************************************
*****
* This module performs certain functions that should be described in this
* paragraph, but since this is only a template, it can't really, now, can
* it)
*
*       Entry:  a0 -- pointer to part of the input data.
*               a1 -- pointer to some more of the input data.
*               d0 -- flag used to signal alternate functions.
*
*       Exit:   a2 -- pointer to the output data generated.
*
*       Uses:   d1 -- intermediate results.
*               d2 -- mask for MSB.
*
*       Stack:  requires 60 bytes.
*
*       Globals: none used.
*
*       Called: Init, system startup code.
*               Exit, system exit code.
*               Furblesnotzer, mutually recursive call.
*
*       Calls:  Furblesnotzer routine to obtain Hrals's constant.
*
*       Log:
* 860703 jens:  Initial entry.
* 860705 jens:  Massive revisions refurbishing the furblesnotzer section.
* 860706 hey:   Undid stupid changes done in haste by jens.
*
```

FIGURE 11.1 *A module specification template.*

Non-specification commentary is extremely important, and the care with which it is written has a direct impact on the understandability of the code. A comment per line, plus one or more comment lines per block of "difficult" code, is recommended. The most easily understood comments read like a story, sort of a running commentary on what the code is doing. "Useless" comments are those that merely put the assembly code into words—INCREMENT D0 and MOVE A4 INTO A6, for example.

I spend nearly as much time on the commentary as I do on the code, but find that it takes nearly twice as long to make use of previously written code that has poor commentary. If you ever plan to reuse your old code, the time spent producing good commentary is free!

Why Do I Need Standards?

Writing assembly code that is both fast and maintainable seems an impossible task. There are historical reasons why assembly code has lacked standards. Many crack assembly coders, often treated with reverence usually reserved for movie stars and sports heroes, consider "fast" and "maintainable" to be contradictory, and cite their expertise as justification for producing "write-only" (unreadable) code. Writing maintainable code has traditionally been seen as an impediment to job security. ("If you can write your way into a job, why write yourself out of one?" as someone once said.) This argument loses, however, when an exciting new project comes along and the maintainer isn't available because no one will take over maintenance of his code.

For hobbyists, the "standards problem" is even more prevalent. Small operating systems found on today's affordable 68000-based systems lack facilities for code revision control, and few hobbyists can afford hard disks. These limitations have led to poor documentation and little commentary, as hobbyists try to squeeze every last byte out of their systems. This practice is often regretted when bugs are discovered or improvements are desired.

The use of coding standards is every bit as important as the actual code written. Such standards foster communication of the intent of the code—not only to other users but to the author—and subsequently speed coding and reduce the occurrence of bugs and design flaws.

Different Styles for Different Applications

After you decide on a project, decompose it down into modules and set the standards to use, the next step is to examine the needs of the application for things that will impact coding style. Is the code going in ROM? Will it be used on a multitasking system? Will it be used recursively? Will it need to work at different locations?

The Commodore Amiga, for example, will run your code at an arbitrary location that is different each time you run the program. A *position-independent* program will load much faster in such an environment. Code that is *reentrant* can be shared among different tasks, or be called recursively. Code in ROM must pay attention to separating initialized data from data in RAM, and usually must be kept as small as possible.

Declaration of Position Independence

Position-independent (PI) code can be loaded and run without modification at any arbitrary location in memory. Such code is desirable in systems that do not have memory management hardware. Such systems often have *relocating loaders*, which take non-PI code and sum an offset into all the addresses found in the code. Although adjusting the addresses is generally not very time-consuming, avoiding the relocation process entirely often speeds program loading.

Of course, if you are at all interested in assembly coding, it is probably because of speed or space limitations. Here's a nice bonus: PI code is often both faster and smaller than non-PI code, especially if the target data has an address that cannot be expressed as a simple displacement—that is, at absolute addresses less than 8000 hex. Figure 11.2 offers a comparison of PI and non-PI data accesses.

It is when data needs to be written that PI code incurs overhead. PC-relative addressing is not allowed as a destination operand on the 68000. One way to overcome this problem is to use an extra LEA instruction to resolve the destination address before it is written. However, writing a PC-relative location in this manner produces non-reentrant code. The optimal solution is closely related to reentrancy issues.

Reentrant Modules Can Be Shared

In multitasking systems, it is often desirable to have a single copy of code that can be used by different tasks. Library routines are often sharable among tasks, and such things as display format and graphics routines need only have a single copy of the code in memory. Multiuser systems have an even greater need for sharable code; if each of 50 users on a Unix minicomputer needed a separate copy of the shell in memory, it would quickly run out of memory!

```
CharTab DC.B    '0123456789ABCDEF'
*
* Position independent indexed data references use PC relative addressing.
*
        move.b   CharTab(pc,d5),(a0)+           4 bytes, 18 clocks
*
* If "CharTab" is in the first 32k, Register relative with displacement
* addressing is slightly faster.
*
        move.b   CharTab(d5),(a0)+        4 bytes, 16 clocks
*
* If "CharTab" has an address greater than 7FFF, register relative cannot be
* used. Two instructions and a scratch register are needed. Addresses over
* 8FFF incur even more overhead.
*
        lea      CharTab,a1       CharTab<=7FFF    4 bytes,  8 clocks
*                                 CharTab>7FFF     8 bytes, 12 clocks
        move.b   (a1,d5),(a0)+                    4 bytes, 18 clocks
*                                 Total)           8-12 bytes, 24-30 clocks
```

FIGURE 11.2 *Comparison of PC-relative indexed and absolute indexed.*

Reentrant-capable code does not change in any way as a result of being executed. This behavior ensures that the code in one task can be interrupted, a second task can completely execute the same code and the first task can then resume execution of the interrupted code without ill effect.

The key to writing reentrant code is to avoid modifying persistent data. That means that global variables are out, that all parameters must be passed on the stack and that local variables must be written in a way that keeps them from getting clobbered if the module is reentered.

The typical way to provide reentrant-capable writable storage is to allocate it on the processor stack, a procedure that has the side benefit of producing position-independent code. This results in smaller load modules than when large, static buffers are used, and the code is inherently ROMable because all writable storage is segregated from constants and program code. An often-overlooked benefit is that writing reentrant-capable code tends to produce structured, modular code because the programmer cannot arbitrarily access data in other modules.

Motorola knew programmers would want to write reentrant code and included the LINK and UNLK instructions for that purpose. To use these instructions, you have to know how much room you'll need for your writable data. It is a good idea to use a EQU directive so that changes will only have to be made in one place. Note that the size used in the LINK instruction must be negative, or previously written stack data will get destroyed.

The LINK instruction is then used with a register and the size needed. The register used will then be the base of the local storage area, or the *frame pointer*. Indices or offsets from the frame pointer now refer to your local data area, which can be written and read in a position-independent manner. The frame pointer will eventually be needed by the UNLK instruction, so don't change its value unless you can get it back when needed. Figure 11.3 shows an example of using the LINK and UNLK instructions to create local storage and subsequent use of the storage.

Reentrant-capable code does have a few disadvantages. When speed is more important than reusability, the overhead of the LINK and UNLK instructions can be saved and an extra register that would normally be used for the frame pointer is available. Local variables that are used only once are expensive in reentrant-capable code, and it is a bit harder to write. But unless speed is the overriding issue, it is good practice to write reentrant-capable code whenever possible.

Recursion in Assembly Code

Writing *recursive* assembly code routines is not nearly as hard as it seems. Recursion is nothing but a special case of reentry. If you don't actually need reentrant-capable code because of reasons cited earlier, the rules are different when writing recursive code. For one thing, you know exactly where the code will be "interrupted"—namely, at the recursive call. If you're taking the trouble to write recursive assembly code, you probably don't want the overhead of saving and restoring all the processor registers at each recursive

```
* For position-independent, writable storage, use the LINK and UNLK
* instructions to allocate space on the stack.  Then use register
* relative addressing to access the storage.  A6 is traditionally used
* for this purpose.
*
CharTab DC.B    '0123456789ABCDEF'
TABSIZE EQU     *-CharTab
STR     EQU     0                   First local variable, string,
INT     EQU     STR+TABSIZE           of proper size for CharTab.Second var
CHAR    EQU     INT+4                 is an integer Third local variable
SHORT   EQU     CHAR+1                is a character Fourth local variable
LOCAL_1 EQU     SHORT+2               is 16 bits.

Module1 link    a6,#-LOCAL_1        Create local storage area for Module1.
        .
        .
        .
        move.w  #TABSIZE-1,d0               Get the size of the source
data,
loop1:  move.b   CharTab(pc,d0),(a6,d0)      move it using PC relative,
        dbra     d0,loop1                    until it's all gone.
        .
        .
        .
        move.b   STR(a6),CHAR(a6)Move local variables:
        move.w  #$1234,SHORT(a6)
        move.l  #$56789ABC,INT(a6)
        .
        .
        .
        unlk     a6                 Release the storage back to the stack,
        rts                          before leaving the module.
```

FIGURE 11.3 *Example of position-independent, writable storage.*

call. Only registers that represent persistent data (data that will be needed after the recursive call) need be saved. Such code is called *serially reusable* code.

Recursive solutions are generally not as efficient as iterative ones. In some cases, though, eliminating the recursion is difficult, particularly if there is more than one recursive call. When working in assembly code, the overhead of recursion can be reduced considerably by identifying and saving only the persistent data. These factors combine to make recursion attractive for some assembly code problems.

Jump Tables

A typical way to change flow of control when there is more than a two-way choice is with a *jump table*. Such tables are quite easy to code using absolute addressing, but cannot easily be loaded in locations other than the original location. Relative jump tables, (branch tables) require that the number stored in the jump table be relative to some point in the code, usually the base of the jump table itself. Note that the branch table can use 16-bit entries if all its modules are within 32K of the table, whereas jump tables can use 16-bit entries only if all the routines are in the first 32K of memory. Figure 11.4 shows how jump tables and branch tables are constructed.

```
AbsTab  DC.L    SlartyBartfarst Addresses of modules.
        DC.L    _Hexadecimalize
        DC.L    _Quicksort

RelTab  DC.W    SlartyBartfarst-RelTab    Offsets of modules.
        DC.W    _Hexadecimalize-RelTab
        DC.W    _Quicksort-RelTab
```

FIGURE 11.4 *Absolute jump and relative branch tables.*

```
****    DispatcherAbs    ************************************************
*******
* Jump to the routine selected by an index.  The jump table must be in the
* first 32k of memory space.
*
*       Entry:  a0 -- index of routine to jump to.
*
DispatcherAbs
        add.l   a0,a0                   Make simple index into a word
offset,
        add.l   a0,a0                     then into a longword offset.
        move.l  AbsTab(a0),a0           Fetch the address from the jump
table,
        jump    (a0)                      and jump to it.
* 10 bytes, 40 clocks.

****    DispatcherRel    ************************************************
*******
* Branch to the routine selected by an index.  The Branch table can be
anywhere
* in memory, but must be within 32k of the Dispatcher and all dispatched
* routines.
*
*       Entry:  a0 -- index of routine to branch to.
*
DispatcherRel
        add.l   a0,a0                   Make simple index into a word
offset,
        move.w  RelTab(pc,a0.w),a0        fetch the branch offset from
table,
        jmp     RelTab(pc,a0.w)   and jump, summing in the table base.
* 10 bytes, 36 clocks.
```

FIGURE 11.5 *Comparison of absolute and relative jump table dispatchers.*

The table dispatcher gets a bit more complicated when coded in a position-independent manner. There are two memory references that must be resolved in a position-independent manner: the address of the jump table entry and the address of the module referenced in that entry. *PC-relative indexed* addressing is used in each reference, and the ability to use just the lower half of a register as an index saves four clock cycles. Figure 11.5 shows the code for each type of dispatch routine. If you can assume that all of your routines will be in the first 32K of memory space, a 16-bit absolute dispatch routine will be faster although much less flexible.

In Figures 11.4 and 11.5, calling either Dispatcher with 1 in a0 will execute the routine _Hexadecimalize; using 0 will cause the routine SlartyBartfarst to be executed.

Jump tables that are built dynamically would, of course, have to be constructed in a stack frame in order to be reentrant capable.

Mixing Assembly Code With Other Languages

The lack of structure inherent in assembly code tends to keep us reinventing wheels. After spending hours getting a tight, efficient assembly code module to work properly, it seems a waste to use it in only one program or application. If you have used the methods outlined for writing reentrant-capable code, you have licked most of the hard part of *high-level language* (HLL) interface. Although most HLLs do not demand reentrant-capable code, it tends to be easier to interface. I'll use C for illustration; common procedural languages should be similar.

C uses the stack for parameter passing and uses stack frames for local variables, exactly as shown earlier. The order in which parameters are passed on the stack is not specified by the C language in order to allow compiler writers to choose the most efficient way. Needless to say, your 68000 assembly modules will not be portable to other processors, and you should be aware that they may not even be portable to other 68000-based C compilers.

Determining the parameter-passing protocol for your C compiler will require some experimentation on your part. Figure 11.6 is a simple test program that can be used to discover the way your compiler passes parameters. Many compilers have an option that causes the compiler to emit assembly code for inspection. If yours does not, you will have to actually execute the test program using a disassembling debugger. If you don't have a disassembling debugger, you may just have to dump the object code, sit down with the Motorola book and disassemble it by hand.

Figure 11.7 shows the assembly code produced by the Uniflex C compiler. It is obvious that this compiler pushes the parameters on the stack in reverse order. This method turns out to be the most common; it is probably safe to assume that your 68000 C compiler behaves the same if you can't actually check it. If you plan to do this often, you will probably want to build an "include" file that will have symbolic labels for the proper offsets into the stack for the various parameters.

There are other "gotchas" associated with mixing assembler with HLL. Many compilers reserve certain registers for special purposes, and most that allow register variables expect the called module to save and restore any registers it uses. Your favorite assembly routine might well break some programs and not others by not following the compiler's conventions.

Let's Get Practical

By now you have enough knowledge to get dangerous, and the examples presented have probably stirred some thought. Let's put together some of the things we've covered to come up with a practical, working program.

```
/*
 * Parameter Test Program.  This program is compiled to assembly code
 * for examination, NOT executed.
 */

main()
 {
        dummy(1, 2, 3);          /*Pass some things to dummy routine.*/
 }
```

FIGURE 11.6 *Test to determine parameter-passing protocol.*

```
        global  _main
_main   link    a6,#-(4)
        move.l  #3,d2           Moves the last parameter first,
        move.l  d2,(sp)
        move.l  #2,d2             followed by the middle,
        move.l  d2,-(sp)
        move.l  #1,d2             and the first.
        move.l  d2,-(sp)
        jsr     _dummy
        add.l   #6,sp
        unlk    a6
        rts
```

FIGURE 11.7 *Assembler output of parameter test program.*

Listing 11.1 shows a module that converts a 32-bit quantity to eight hexadecimal ASCII characters. This is a "naive" implementation, using straightforward computation to get the job done. First, the arguments are copied from the stack into two registers. Constants for masking the least significant nybble and for the number of nybbles to convert are then loaded into registers. Now everything is ready for the main loop of the module.

A copy of the integer to convert is shifted by a different amount each time through the loop, in order to process the most significant nybble first. Next, the shifted integer is masked for the least significant nybble. On line 12, the decision of whether the nybble is one of 0 through 9 or A through F is made. If the nybble is not expressible as a decimal digit, it has a special constant added to it—namely, the difference between the end of the digit sequence and the beginning of the alpha sequence in the standard ASCII set.

In either case, the magic constant needed to turn a nybble into an ASCII character is summed in at line 1A. The resultant ASCII character is then put into the output string, the number of bits to shift is decremented for the next time through and the loop is repeated—unless the shift count was already 0, in which case the loop is exited and the module returns to whatever module called it.

This version of Hexadecimalize requires 38 bytes and 1552 clock cycles, on the average, to execute. The number of clock cycles will vary depending on how many non-decimal characters are produced. There are 10 instructions consuming an average of 94.25 clock cycles each time through the loop.

An Improved Hexadecimalizer

A problem with the computation based Hexadecimalize is that the output data is not a simple function of the input data. A test and a decision must be made on the input data in order to chose one of two methods of generating the output. This test is in the crucial inner loop. If we can eliminate the test, we should be able to speed things up considerably.

A simple way to make the output data a simple function of the input data is to use a constant lookup table. Listing 11.2 shows such an implementation. Lines 10 through 1A are much the same as in the previous version; arguments are copied into registers, and shift and mask constants are loaded.

Now, instead of shifting the integer to be converted by huge amounts, we simply rotate the most significant nybble into the least significant nybble. Each time through the loop the next less significant nybble is shifted in. Next a copy is made, which is masked for the nybble of interest.

Now our constant table comes into play. Using PC-relative addressing for position independence, we simply index into the table and pick out the proper character. This character is placed in the output buffer and a special looping instruction is used to repeat until our count of nybbles to process is exhausted.

This version of Hexadecimalize is superior for several reasons. Because of the simple algorithm, the number of bytes produced is easily changed. (It is left as an exercise to the reader to make the number of bytes produced a parameter that is passed into the module!) Although slightly larger at 44 bytes, it is nearly twice as fast, executing in 880 clock cycles. The loop contains only six instructions, consuming 52 clocks each time through. This can be stated with certainty, since the number of clock cycles is deterministic, not dependent on the input data.

Recursive Quicksort

Now that we've demonstrated position independence, we might as well do something useful to demonstrate some features of reentrant-capable code. One of the more feared assembly code tasks is recursion, which requires careful thought about reentrancy issues. Recursive code is guaranteed to be reentered, albeit in a more controlled fashion than via rude interrupts.

When was the last time you needed to sort something? When was the last time you sorted in some HLL and found it too slow? One of the most efficient general-purpose sorting algorithms, called Quicksort, was invented in 1960 by C. A. R. Hoare. Quicksort is of special interest as an example of the appropriate use of recursion in assembly code; the recursion is not easily removed because it calls itself in two separate places. Because we are using assembly code, we can dispense with some of the overhead associated with recursion in HLLs and come up with an efficient version in spite of the recursion.

Briefly, Quicksort operates under the "divide and conquer" idea. Small sets of data can be sorted much quicker than large data sets, and Quicksort

alternately sorts and divides through the data until it has used each item in the data set as a comparison key, at which point it returns up through the recursive calls.

The algorithm uses an arbitrarily chosen partition value for dividing the array to sort. The array is then scanned from both ends, looking for value pairs that are "backwards" with respect to the partition value. The out-of-sequence items are then swapped. The scanning and swapping continues until the scanning pointers meet. At that point, a recursive call is made to sort the lower part of the array, followed by a recursive call to sort the upper part.

In Listing 11.3, lines 0 and 4 copy the arguments off the stack into registers. The first thing that must be done is to see if it is time to quit recursing! If not for the compare and branch on lines 8 and A, this module would never return. The partition value is arbitrarily taken as the last item in the area to be sorted, and pointers to the first and last items to be sorted are copied into the registers that will be scanning from either end.

The action begins at line 12, which is the top of both the outer loop and the first of the inner loops. Lines 12 and 14 start from the bottom and compare each value with the partition value, looking for a value that is greater than or equal to the partition value. When it is found, line 16 backs the pointer up so that it will be pointing *to* the item of interest, not *past* it.

Similar but opposite action happens at lines 18 and 1A. Starting at the top, values are compared to the partition value, stopping when one is found that is less than or equal to the partition value.

Now we have two pointers pointing at values that lie on either side of a partition value. Before swapping them, we need to check (in lines 1C and 1E) that our pointers have not already crossed. Assuming that the pointers have not crossed, we then swap the values pointed to by the scanning pointers in lines 20 through 24 before repeating the outer loop.

If the pointers had been found to have crossed in the test on line 1C, it would be time to exit the loop and recurse on the two partitions. Lines 28 through 2C swap the final value in order to provide the recursive call with a new partition value.

The best part is coming. Assembly code gives the programmer the freedom to violate the rules imposed by HLLs when efficiency is of paramount importance. In a HLL implementation of Quicksort (such as the C implementation in the right margin of Listing 11.3), the recursive call would save on the stack all the registers that are used in the module. This time-consuming action is the biggest cause of recursion's poor reputation for speed. However, all our local variables are in registers, and we need to save only those that will be needed after the recursive call. It turns out that only two pieces of information, ls+1 and r, need to be saved. Lines 2E and 30 push these values on the stack in preparation for the second recursive call. Line 34 calculates the value ls-1 while moving it into the proper register for the first recursive call.

Since there is no more state to save here, we can pass the parameters for the first recursive call in registers—something you simply cannot do in most

FIGURE 11.8 *Output of test driver program.*

HLLs. In order to do this, we use a second entry point at line 8, which slightly violates the rules of modularity but saves four stack accesses (two pushes before the call and two reads after the call). Note that PEA and LEA can be used to sum constants with registers, storing the results in different registers. Upon return from the second recursive call, the arguments are removed from the stack and control returns to the caller in lines 3C and 3E.

One of the problems with assembly code is the lack of tools for testing and debugging. It is no fun single-stepping through a huge assembly code project, peering at hexadecimal values that should be decimal or ASCII. For this reason alone, calling assembly code from C makes debugging much easier. Listing 11.4 is a short test driver that gives a quick check of both the Hexadecimal and the Quicksort modules.

The routine main() sets up an array of random integers to sort and calls dump() to display the results. Getting this much functionality out of assembly code would have taken many times longer to write and debug, and would have detracted from the real problem at hand: getting Hexadecimalize and Quicksort working. Figure 11.8 shows the output produced by the test driver and shows that some hexadecimal values were indeed sorted. Although this test is not exhaustive, it would have taken much longer to get just this far if assembly code were the only tool available.

Summary

Assembly code is not a general-purpose tool. It requires self-discipline in areas that are normally mechanically enforced in HLLs. Local storage management for re-entrance capability, careful thought to addressing modes and table structures for position independence, and careful use of commentary for documenting the types of things that are documented free in HLLs are all things that must be done if assembly code projects are to be successful. But the reward is great—that wonderful feeling you get from knowing that

something you wrote is running just as fast as is possible without changing processors.

References

Motorola, Inc. *MC68000 16/32 Microprocessor: Programmer's Reference Manual*. 4th ed. Englewood Cliffs, NJ: Prentice-Hall. 1984.

Sedgewick. *Algorithms*. Reading, MA: Addison-Wesley. 1983.

Listing 11.1

```
00000004                arg1    EQU     4
00000008                arg2    EQU     8
                                global  _Hexadecimallize
00000000                _Hexadecimallize
00000000 202F 0004              move.l  arg1(a7),d0     Get the integer to convert to hex.
00000004 206F 0008              move.l  arg2(a7),a0       and the pointer to the storage buffer.
00000008                Hexadecimallize
00000008 7E1C                   move.l  #7*4,d7         Bits to shift the first time through.
0000000A 7C0F                   move.l  #$0F,d6         Nybble mask.
0000000C 2A00           hexLoop move.l  d0,d5           Get the long to convert, <----------------+
0000000E E2AD                   lsr.l   d7,d5             shift it by the proper amount,          |
00000010 CA06                   and.b   d6,d5             and mask out the upper portion.         |
00000012 0C05 000A              cmp.b   #10,d5          Is it greater than 10?                    |
00000016 6502                   bcs.s   under             No, no need to adjust it. -----+        |
00000018 5E05                   add.b   #('A'-'9')-1,d5   Yes, sum in the magic offset. |        |
0000001A 0605 0030      under   add.b   #'0',d5         Sum in the ascii offset, <-------+        |
0000001E 10C5                   move.b  d5,(a0)+          Add stuff it in the output string.      |
00000020 5907                   sub.b   #4,d7           Reduce the shift count.                   |
00000022 64E8                   bcc     hexLoop         Are we all done now? ---------------------+
00000024 4E75                   rts                       Yes. Go home. -->
00000026
```

Listing 11.2

```
                        ****    Hexadecimalize ********************************************************
                        * Hexadecimalize 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:  4(a7) or d0 -- Long word to be converted to hex.
                        *               8(a7) or a0 -- Pointer to buffer where hex characters will go.
                        *
                        *       Exit:   d0 -- unchanged.
                        *               -8(a0) -- points to eight ASCII characters.
                        *
                        *       Uses:   d5 -- current nybble to convert is LSN.
                        *               d6 -- nybble mask: constant $0F.
                        *               d7 -- nybble counter.
                        *
        00000004        arg1    EQU     4
        00000008        arg2    EQU     8
                                global  _Hexadecimalize
00000000 30313233343536 CharTab DC.B    '0123456789ABCDEF'      Where we keep our hex characters.
00000010                _Hexadecimalize
00000010 2E2F 0004              move.l  arg1(a7),d0     Get the integer to convert to hex,
00000014 206F 0008              move.l  arg2(a7),a0       and the pointer to the storage buffer.
00000018                Hexadecimalize
00000018 7E07                   move.l  #7,d7           Bytes to make, less 1.
0000001A 7C0F                   move.l  #$0F,d6         Nybble mask.
0000001C E998           HexLoop rol.l   #4,d0           Shift the next nybble into the LSN, <---------,
0000001E 3A00                   move.w  d0,d5             make a copy for masking,                    |
00000020 CA46                   and.w   d6,d5             mask out all but least significant nybble,  |
                                                            index into char table and store result.   |
00000022 10FB 5000              move.b  CharTab(pc,d5),(a0)+                                          |
00000026 51CF FFF4              dbra    d7,HexLoop      Repeat until done, and when done, ------------'
0000002A 4E75                   rts                       hit the road, Jack. -->
```

Listing 11.3

```
*****   Quicksort   ***************************************************
* Quicksort sorts a random array of integers in N*log(N) time.
*
*       void Quicksort(left, right)
*        int *left;             /*pointer to first element of array to sort.*/
*        int *right;            /*pointer to last element of array to sort.*/
*
*       Entry:  4(a7) -- pointer to first element in array.
*               8(a7) -- pointer to last element in array.
*
*       Exit:   none.
*
*       Uses:   a0 -- int *l;   /*pointer to first element in the array.*/
*               a1 -- int *r;   /*pointer to last element in the array.*/
*               d0 -- int  v;   /*the "partition" element.*/
*               d1 -- int  t;   /*temporary used to exchange elements.*/
*               a2 -- int *ls;  /*Scanning pointer, scans up from left.*/
*               a3 -- int *rs;  /*Scanning pointer, scans down from right.*/
*
*
arg1    EQU     4
arg2    EQU     8
        global  _Quicksort

_Quicksort
        move.l  arg1(a7),a0     Retrieve the low end pointer off the stack,
        move.l  arg2(a7),a1       likewise the high end pointer.

Quicksort
        cmp.l   a1,a0           Are we finished yet?
        bgt.s   done              Yes, begin the recursive return.
        move.l  (a1),d0           No.  Load the partition variable,
        move.l  a0,a2             the left side incrementing pointer,
        move.l  a1,a3             and right side decrementing pointer.
*
loop1   cmp.l   (a2)+,d0 <----- Scan left pointer up for an element swap.
        bgt     loop1 >---------                                          |
        sub.l   #4,a2             |   (but get rid of the last post-increment)
loop2   cmp.l   -(a3),d0 <--+     |   scan right pointer down for swap.
        blt     loop2 >-----+-----+
```

```
00000004
00000008

00000000
00000000 224F 0004
00000004 226F 0008
00000008
00000008 B1C9
0000000A 6E3J
0000000C 2011
0000000E 2448
00000010 2689

00000012 B09A
00000014 6E0C
00000016 598A
00000018 B0A3
0000001A 6DFC
```

```
                                                        Quicksort(l, r)
                                                        int *l;
                                                        int *r;
                                                        {
                                                          if(r > l) {
                                                            v  = *r;
                                                            ls =  l;
                                                            rs =  r;
                                                            while(1) {
                                                              while(v > *ls++);

                                                              ls -- 2;
                                                              while(v < *--rs);
```

```
0000001C B9CB                   cmp.l   a3,a2           ; Did the pointers cross?
0000001E 6E28                   bgt.s   break >-------+ ;   Yes, break out of the loop.
00000020 2211                   move.l  (a2),d1       ; ; Through a temporary,
00000022 2493                   move.l  (a3),(a2)     ; ;   swap the two
00000024 2681                   move.l  d1,(a3)       ; ;   out-of-place values,
00000026 60EA                   bra     loop1 >-------|--   and go look for more to swap.
                        *                             |
00000028 2212           break   move.l  (a2),d1 <---+   Through a temporary,
0000002A 2491                   move.l  (a1),(a2)       swap is and r in preparation for
0000002C 2281                   move.l  d1,(a1)         new partition in recursive calls.
0000002E 2F09                   move.l  a1,-(a7)      Push 2nd call's 2nd argument.
00000030 486A 0004              pea     4(a2)           followed by 2nd call's 1st argument.
00000034 43EA FFFC              lea     -4(a2),a1     Set up 1st call's 2nd argument.
00000038 61CE                   bsr     Quicksort     Make the 1st recursive call.
0000003A 61C4                   bsr     _Quicksort    Make the 2nd recursive call,
0000003C 508F                   addl.l  #8,a7           remove its arguments,
0000003E 4E75           done    rts                     and return to caller.
```

```
if(r > l)
  break;
t  = *l;
*l = *r;
*r = t;
}

t  = *l;
*l = *r
*r  =  t

Quicksort(l, l-1);
Quicksort(l+1, r);
```

Listing 11.4

```
/*
 * Test driver for Quicksort and Hexadecimalize assembly code routines.
 */
#include <stdio.h>

#define PER_LINE  8
#define LINES             6
#define  ASIZ             (LINES*PER_LINE)

main()
 {
        int  array[ASIZ], /*An array to sort,*/
            *end,                 /*  which will have an end,*/
            *p;                   /*  and a scanning pointer.*/

        end = &array[ASIZ-1];     /*Get a pointer to the last element.*/

        for(p = array; p <= end; p++) *p = rand();   /*Initialize randomly.*/
```

```
        dump(array, end, "Random Array");           /*  look it over.*/

        Quicksort(array, &array[ASIZ-1]);           /*Sort the array.*/
        dump(array, end, "Sorted Array");           /*  and inspect it.*/
}

/****    dump    ************************************************************
* Dump the contents of an array of integers in hexadecimal, given the beginning
* and end.  Print an identifying message at the top.
*/
dump(first, last, msg)
 int  *first, *last;
 char *msg;
 {
        int  i;                     /*General purpose index variable.*/
        char buf[9];                /*A place to store the hex string.*/

        buf[8] = 0;                 /*Don't to forget to terminate the string!*/

        printf("\n%s\n", msg);                      /*Print the message.*/
        while(first < last) {                       /*While there is data,*/
            for(i = 8; i; i--) {            /*  print, eight to a line,*/
                hexadecimalize(*first++, buf);      /*  the hex representation.*/
                printf("%s ", buf);
            }
            putchar('\n');
        }
        putchar('\n');
}
```