15 YEARS OF POWER PROGRAMMING!

#162 MARCH 1990 $2.95 ($3.95 CANADA)

Dr. Dobb's JOURNAL

SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

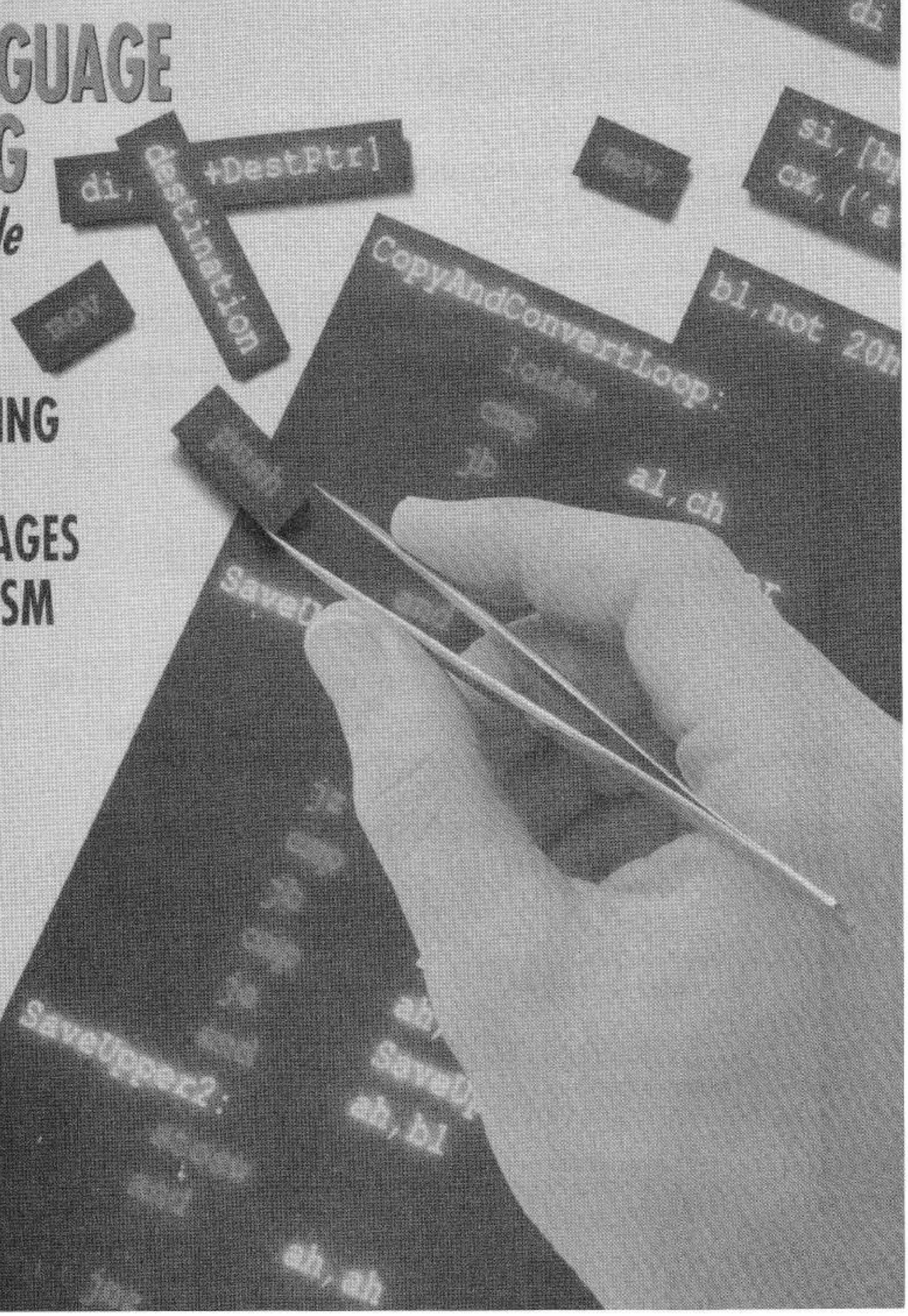
ASSEMBLY LANGUAGE PROGRAMMING

Hand-Picked Hand Code

- ASM LIVES!
- 68040 PROGRAMMING
- MIXING ASM WITH HIGH-LEVEL LANGUAGES
- OBJECT-ORIENTED ASM

80386 DEBUGGING

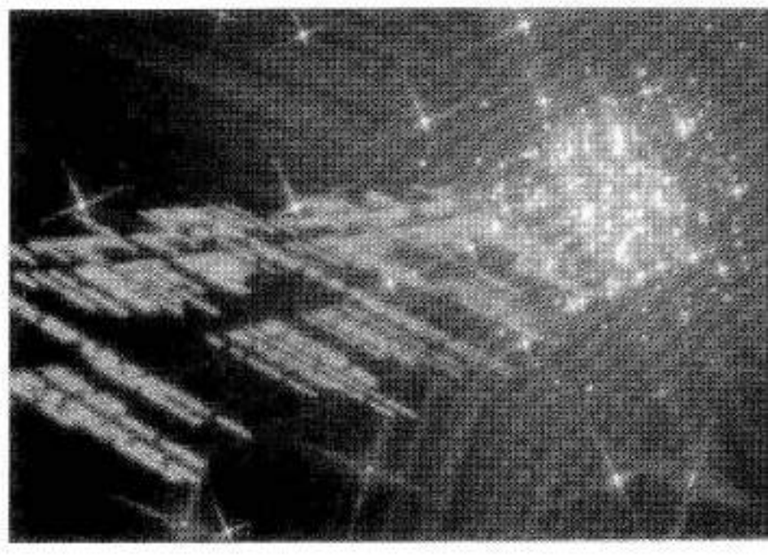
MS WINDOWS MEMORY MANAGEMENT



0 38351 16562 8 03

CONTENTS

MARCH 1990
VOLUME 15, ISSUE 3



FEATURES

EXAMINING ROOM

PROGRAMMER'S WORKBENCH

COLUMNS

DEPARTMENTS

PROGRAMMER'S SERVICES

NEXT ISSUE

If you've been scratching your head over neural networks, you'll want to pay special attention to our April issue. We'll also provide the long-awaited code implementation for our Rhealstone real-time benchmark, begin an in-depth examination of VGA, and include *DDJ's* 1989 index.

Patent Letter Suits

Mark Nelson's article on the LZW data compression algorithm (*DDJ* October, 1989) sparked a forest of fires, at least in respect to patenting algorithms. The first spark, if you recall, was a letter from Ray Gardner, pointing out that the LZW algorithm was patented by Unisys back in 1985 (see "Letters," December 1989). Mark's response answered a few questions but raised several more.

About the time we published Ray's letter and Mark's reply, the U.S. Court of Appeals settled a dispute between the U.S. Patent Office and Sharp Corporation in a case that revolved around Sharp's patent application for a voice-recognition circuit. The Patent Office had rejected Sharp's original application in part because they felt the circuit's only purpose was to execute an algorithm. And, the Patent Office insisted, algorithms can't be patented because they are nothing more than mathematical abstractions. Furthermore, the Patent Office felt that Sharp was trying to patent every possible means of implementing the algorithm, not just the way it was used in this particular voice-recognition circuit.

As it turned out, the Court of Appeals didn't agree with the Patent Office. The court said that an algorithm can be safeguarded, at least as how it is used to describe a physical device (like a circuit) or in terms of other functional equivalents of that algorithm.

To better come to grips with this issue, I called Charles Gorenstein, the Falls Church, Virginia attorney who represented Sharp. Early in our conversation, Mr. Gorenstein stated that "a purely mathematical algorithm is probably not patentable" but, he added, the specific methods of implementing an algorithm are patentable. In other words, what is patentable is the method, not the math. If someone developed a different circuit to execute Sharp's voice-recognition algorithm, that's fine and dandy. And apparently that's part of the basis of the Court of Appeal's decision.

Key to any patent grant is the concept of "new and unobvious," an area that Mr. Gorenstein feels the Patent Office has overlooked. Using a 1979 patent for spreadsheets as an example, he explained that just about anyone with a ledger, a pencil, and some data would fill out the rows and columns in much the same way as they would with an electronic spreadsheet. A ledger — and a spreadsheet — is obvious. He therefore questions whether the spreadsheet patent should ever have been granted. This question of "obvious" raises another important issue. What may be *unobvious* to those in the Patent Office may very well be obvious to technically sophisticated programmers like *DDJ* readers.

What all this leads up to is a letter I received from Bob Bramson, the Unisys patent attorney Mark mentioned in his response. I won't give a blow-by-blow account of the letter, you can read it for yourself on page 8, the first entry in this month's "Letters" section.

I will say that the letter is a politely worded clarification of Unisys's patent on the LZW algorithm, with only a slight sense of the steel behind it, at least in reference to Unisys's intention of going after infringers.

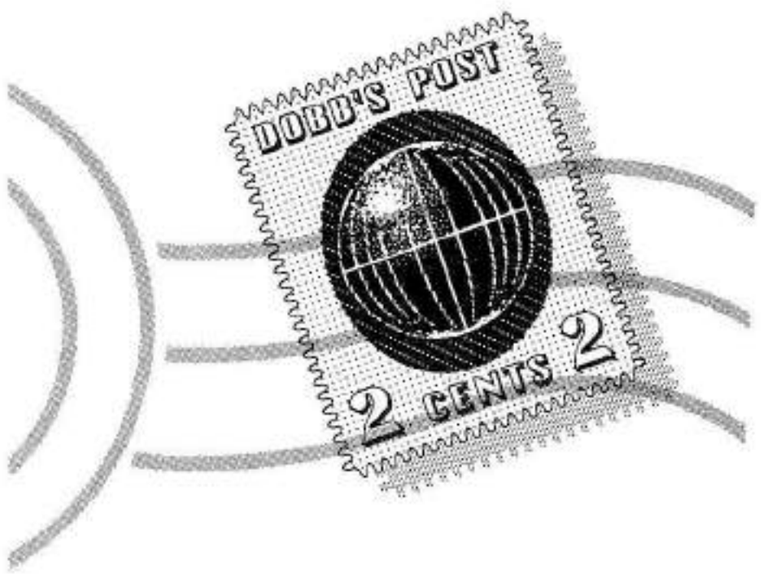
I'm sorry, but I still don't understand. It seems that if, as I think the court ruled, you can use Sharp's algorithm to design a different voice-recognition circuit, you should be able to use Sharp's (or Unisys's or anyone else's) algorithm for an entirely different purpose than it is used in the original patent. That is, you should be able to use the LZW algorithm in a program that has nothing to do with telecommunications or modems. This assumes, of course, that Unisys's patent is for the modem and the algorithm as it helps define the modem. I agree with Mark. Unisys will indeed be very busy tracking down programmers who have implemented some form of the LZW algorithm.

I'm all for any company, large or small, taking steps to protect R&D investments that give it a competitive edge. But it's distasteful for large companies to threaten smaller outfits with litigation that can't be won in the courts, but can be outlasted by a large company with the resources to do so. Now I'm not in any way suggesting this is Unisys's ploy, nor does Mr. Bramson even hint at this, it's far too often the way the world works.

In his response to the letter that started all of this, Mark suggested that software developers who intend on using patented algorithms (like LZW) in commercial products get some legal advice before proceeding. Mr. Gorenstein seconded this, even to the point of suggesting that programmers do a patent search prior to implementation. While this advice is sound and safe, it is also lengthy and expensive, luxuries that software developers usually can't enjoy.

Today's mail didn't bring a letter from a lawyer, but it did include a letter from Dan Abelow, a Newton, Massachusetts reader who specializes in analyzing emerging technologies, and who, coincidentally, proposes to write an article on "Enabling Patents." He calls the topic a "blossoming controversy [that] has failed to germinate positive suggestions" and, from what I can tell, he's making a case that software patents may actually encourage innovation and invention. I don't know that I agree with him, but I'm curious enough to give him a call and find out what he has in mind.

Jonathan Erickson
editor-in-chief



Patented Algorithms

Dear *DDJ*,

In the "Letters" column of your December 1989 issue, Mark Nelson discusses U.S. Patent 4,558,302 entitled "High Speed Data Compression and Decompression Apparatus and Method." This patent was developed by Terry Welch, a former Unisys employee, and is owned by Unisys. According to Mr. Nelson, I have been quoted as saying that Unisys will "license the algorithm for a one time fee of $20,000." As a concession to the modem industry, Unisys has agreed to license the patent to modem manufacturers for use in modems conforming to the V42.bis data compression standard promulgated by CCITT, for a one-time fee of $20,000. This $20,000 license, however, is not a general license under all applications of our patent but is limited to the specific application discussed above.

Responding to the second paragraph of Nelson's remarks, Unisys is actively looking into the possibility that a large number of software developers may be infringing one or more of our data compression patents. We have only recently become aware of these potential infringers and the process of taking action will take some time.

Unisys is happy to accept inquiries from persons interested in acquiring a license to U.S. Patent 4,558,302. If your readers have any further questions, they should contact Mr. Edmund Chung of our licensing office, at 313-972-7114.

Robert S. Bramson
Unisys
Blue Bell, Penn.

Say It Ain't So

Dear *DDJ*,

Dan W. Crockett's assertion in the January 1990 *DDJ* "Letters" section that structured programming requires that each functional node (or implementation unit) have only a single parent is alarming, and damned difficult to program in the real world. I think that he interprets the abstract requirements of structured programming a little too literally when it comes to coding.

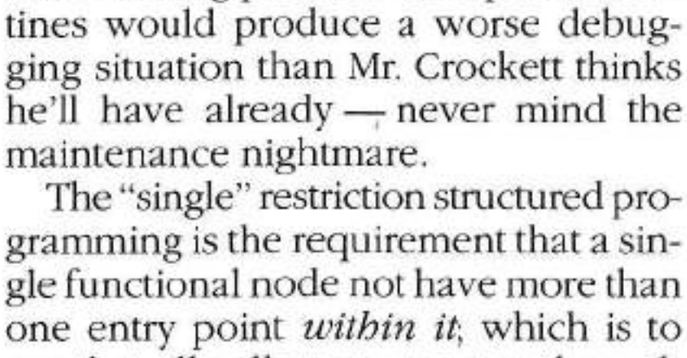
As an example, consider a Pascal function, which formats dollar amounts for output. The function might take a real dollar argument and translate it to a "*$nnn,nnn, . . . ,nnn.nn*" format, and be declared as function *DollarFmt(v : real):string;* The whole point of having the function is that it *can* be called from any procedure or function in a program; if the dollar amounts are formatted incorrectly we can first check to see if the error lies in *DollarFmt*, because it is solely responsible for performing the task.

This is structured programming: Breaking down a task into smaller and smaller (and finally, logically indivisible) subtasks. Subtasks which perform similar or identical tasks can then be coded as a single (probably parameterized) routine.

Mr. Crockett wants program structure to be a B, Quad, or whatever tree, which is fine, but reality demands that the implementation be a threaded tree. Under the Crockett scheme we would be forced to write a separate *DollarFmt* for each caller (*AmountDue_DollarFmt, AmountPaid_DollarFmt, ad nauseam*)! The resulting plethora of duplicate routines would produce a worse debugging situation than Mr. Crockett thinks he'll have already — never mind the maintenance nightmare.

The "single" restriction structured programming is the requirement that a single functional node not have more than one entry point *within it*; which is to say that all callers must enter through the same door. It is perfectly reasonable for a routine to have more than one caller — without multiple callers there would be little reason for building a distinct procedure or function for performing the task.

Going back to the *DollarFmt* example: The structural decomposition of a hypothetical bill printing task might be



It is (hopefully) obvious that "Format Amt," "Format Interest,"and "Format Sum" should be programmed as calls to a single formatting routine, even though they are different tasks in the abstract.

There are dangers in interpreting any abstraction too literally. And there is that other thing, in the word of Will Rogers: "It's not what we don't know that hurts, it's what we know that ain't so."

Brook Monroe
Durham, North Carolina

Locator Fix

Dear *DDJ*,

The listing of Mark Nelson's "Locate tool" in the January 1990 issue has a bug in the *read_header_data* procedure: It occurs in his calculation of *image_size*. The line:

```
image_size = (header.file_
             size_in_pages - 1) * 512;
```

should be replaced with:

```
if (header.image_length_mod_512
                              = = 0)
   image_size = header.file_
                size_in_ pages * 512;
else
   image_size = (header.file_
             size_in_pages - 1) * 512;
```

The bug occurs when the actual image size is an even multiple of 512 bytes. As an example, consider an image size of 1526 (512 * 3). In this case, *header_ file_size_in_ pages* would be three and *header.image_length_mod_ 512* would be zero. Mark's code would produce an incorrect size of 1024 due to the decrement of *header.file_size_ in_ pages*.

I had the opportunity of stumbling into this when writing a combination .EXE loader/relocator/unrelocator for a non-DOS-based embedded control system.

Thank you for your time and keep the interesting articles like Mark's coming.

Bill Trutor
Holden, Mass.

Mark responds: Bill has correctly identified a mistake in my program. I think I might have avoided this mental error with better naming of structure members. In any case, this is one of those program errors that occurs so infrequently (1 out of 512 links) that it can be extremely elusive, so thanks to Bill for pointing it out.

Data Structure Dream Machine

Dear *DDJ*,

In Jeff Duntemann's column in the December 1989 issue of *DDJ*, he mentioned his dream system under Windows 386. I have a question about this. I understand the languages and the PageMaker part, but could he expand

(continued on page 12)

(continued from page 8)
on using Paradox? Do I understand him to mean that you use it to keep track of details about your data structures? Sounds interesting; could he elaborate?

Guy Townsend
CIS 73040,1671

Jeff's response: Hate to be a spoilsport, but mostly what I use Paradox for is to keep my various contact files a keystroke away. The notion of using a real-relational database to manage the gritty details of major development projects is a good one, but the language vendors are going to have to do the integration between the tools and the database. Some major vendors are indeed working on this, (still secretly) and you'll be seeing the results in DDJ *when they surface.*

Intek Heard From

Dear *DDJ*,
From time to time you must hear from disgruntled companies who feel that they have suffered at the hands of one of your writers performing a post-mortem with an axe.

Knowing, however, that Al Stevens is a venerable pilgrim to the hallowed halls of Bell Labs and a proponent of the object-oriented paradigm, we suppose that his summarial execution of our product was caused by a bit of underdone potato.

In his November "C Programming" column, just after explaining to his readership that he was neither rigorous nor controlled, he set about to describe the available C++ compilers and translators available for DOS. Without rigor or control he dismissed our Intek C++ product without evaluating the product at all! He chose instead to fault an example program that we provide with our distribution of the AT&T translator. This example program (which we supply the source to in the product distribution) invokes the C preprocessor, the C++ translator, and the target C compiler in succession. We supplied it to provide our customers with a convenient method of progressing from source to executable if they were invoking from the command line. The mentioned bug occurred only with DOS 3.3, and as with all software companies that stand behind their product with integrity, we supplied the fix to all of our customers long before Al's column went to print. Of course, Al didn't mention that other translator products don't offer anything like this and certainly don't supply the source to such a program.

Did Al mention that the near, far, huge, pascal, fortran, and cdecl keywords don't work with the AT&T distribution or with some other translator products but that they do with Intek C++? No. Ever tried to use a third party header file with some of these keywords in it or try to link to a library, expecting the results of these modifiers, Al?

Did Al mention that if one tried to compile any production size C++ source modules with other products that they would run out of memory? No. Maybe he'd rather make sure that all of his source files were less than 4K in size and that he could only include three or less header files.

Did Al benchmark the fact that the only product from among the group that he mentioned that will compile the AT&T C++ source distribution is Intek C++? No. (That's AT&T's definition of the robustness of a C++ translator implementation by the way.)

Please assure Al that Intek C++ will continue to have a future in the PC world. Our client list has many of the Fortune 500 firms among it. We also use our own product in providing factory automation applications to the major workstation and computer manufacturers in the country.

We feel like you would feel if in a review of magazines *Dr. Dobb's* was dismissed as not being a quality software tools magazine because it sounds like it should be a medical journal.

Mac Cutchins
Intek
Bellevue, Wash.

Al responds: My evaluation of Intek C++ consisted at first of the seemingly simple task of getting it to compile the hello.cpp program that comes with the translator. "Hello, world," nothing more, right out of the box. That simple task involved two days of frustration and several phone calls to Intek.

The Intek technical support person at first insisted that there was something wrong with my setup. The nature of the bug — the translator worked every other time — encouraged both of us to believe that. The compiler failed, I called, he made a suggestion, the compiler worked. I hung up, the compiler failed, I called, and so on. One of those times we changed operating systems, and the technical support person concluded that my copy of DOS 3.3 was the culprit. He must have remembered that episode and subsequently convinced you, Mr. Cutchins, but not me. The bug was identical for DOS Versions 3.0, 3.2, 3.3, and 4.0. Under 3.1, the bug is different, and hello.cpp just never compiles. When I reported these findings, your technical support person, by now tired of hearing from me, curtly announced that there must be some problem, that it would get fixed, goodbye, and thank you very much. If you fixed that bug, I never heard about it, before or after my column went to print. Until now, that is. I guess as a pesky magazine columnist with a free review copy of your pricey product, I don't rate an upgrade. Never once during all those calls did Intek suggest that I abandon the "example" CPLUS program and use the lower-level programs, which I now see is the obvious solution.

In my opinion, Intek C++ is underpackaged and over-priced. The skimpy 40-page spiral-bound manual devotes only 10 pages to installing and using the translator, has exactly two sentences about using it with Turbo C, has some critical typos (the C_COMPILER environment variable is misspelled, for example), and never lets on that the CPLUS program is a mere example to be used at one's own risk. Intek C++ fails to measure up to the standards of quality that PC programmers have come to expect in their language products. My assessment of your future in the PC market was based on my view of the cost and quality of the Intek C++ software, documentation, and support and of the expensive hardware/software foundation necessary to use it. I stand by that assessment. If you believe that Intek C++ has improved substantially since my evaluation of it, I'll be pleased to give it another look.

Round and Round We Go

Dear *DDJ*,
Recently I completed a graphics course, so I read with interest the January issue article by Robert Zigon dealing with generation of circles. I found the article to be a clear and well written exposition of the problem. However, any algorithm based upon the parametric representation of a circle must involve significant overhead in the form of floating-point calculations. A superior algorithm developed by J.E. Bresenham some years ago avoids such overhead.

The Bresenham algorithm makes use of the fact that screen coordinates are integer valued, so it should be possible to select the circle's coordinates using only integer arithmetic as well. Use of only integer arithmetic is the key to the efficiency of the algorithm. The algorithm is used to advance along the perimeter of the circle, selecting the adjacent pixel which is nearest to the circle at each step. Because of circular symmetry, it suffices to determine only one-eighth of the circle using this technique.

An excellent derivation of the algo-
(continued on page 14)

Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

PUBLISHER *Peter Hutchinson*

EDITORIAL
EDITOR-IN-CHIEF *Jonathan Erickson*
MANAGING EDITOR *Monica E. Berg*
TECHNICAL EDITORS *Michael Floyd, Ray Valdes*
EDITORIAL ASSISTANT *Janna Custer*
CONTRIBUTING EDITORS *Al Stevens, Jeff Duntemann, Martin Tracy, David Betz, Tom Genereaux, Andrew Schulman*
COPY EDITORS *Rhoda Simmons, Pamela Dillehay, Nan Fornal*
EDITOR-AT-LARGE *Michael Swaine*

ART/PRODUCTION
ART/PRODUCTION DIRECTOR *Larry L. Clay*
ART DIRECTOR *Michael Hollister*
PRODUCTION SUPERVISOR *Amy Shulman Lesovoy*
TECHNICAL ILLUSTRATOR *Linda Ann Clark*
TYPOGRAPHERS *Teresa Raines, Margaret Anderson, Charlene Carpentier*
COVER PHOTOGRAPHER *Michael Carr*

CIRCULATION
DIRECTOR OF CIRCULATION *Maureen Kaminski*
CIRCULATION MANAGER *Randy Robertson*
CIRCULATION PLANNING MANAGER *Manny Sawit*
DIRECT MARKETING MANAGER *Andrea Weingart*
NEWSSTAND MANAGER *Sarah Forsman*
DIRECT MARKETING COORDINATOR *Francesca Davies*
PROMOTION COORDINATOR *Pam Moore*
FULFILLMENT COORDINATOR *Anne Jean*

ADMINISTRATION
VICE PRESIDENT OF FINANCE *Kate Deschamps*
CONTROLLER *Mary Collopy*
CREDIT MANAGER *Betty Arsene*
ACCOUNTING SUPERVISOR *Renate Kernke*
ACCOUNTS RECEIVABLE *Wendy Ho*
ACCOUNTS PAYABLE *LuAnn Rocklewitz*

MARKETING/ADVERTISING
DIRECTOR OF SALES AND MARKETING *Karla Spormann*
ADVERTISING COORDINATOR *Laura Stack Pullen*
MARKETING ASSISTANT *Sara Noah Ruddy*
ACCOUNT MANAGERS *see page 152*

M&T PUBLISHING INC.
CHAIRMAN OF THE BOARD *Otmar Weber*
DIRECTOR *C. F. von Quadt*
PRESIDENT *Laird Foshay*
VICE PRESIDENT OF PUBLISHING *William P. Howard*
VICE PRESIDENT/GROUP PUBLISHER *Randall L. Stickrod*

DR. DOBB'S JOURNAL (USPS 307690) is published monthly, except semimonthly in December, by M&T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063; 415-366-3600. Second-class postage paid at Redwood City and at additional entry points.

ARTICLE SUBMISSIONS: Send manuscripts and disk (with article and listings) to the editorial assistant 415-366-3600.

***DDJ* ON COMPUSERVE:** Type GO DDJ.

***DDJ* LISTING SERVICE:** 603-882-1599. Supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Type *listings* (use lowercase) at the login prompt.

SUBSCRIPTION: $29.97 for 1 year; $56.97 for 2 years. Foreign orders must be prepaid, including the additional postage (air or surface) in U.S. funds drawn on a U.S. bank. Add $13 for surface mail to all addresses out of the U.S.; add $33 for airmail to Canada and Mexico; or $32 for airlift to all other countries.

POSTMASTER: Send address changes to *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. **ISSN 1044-789X**

CUSTOMER SERVICE: For subscription orders, questions, and changes of address call toll-free 800-456-1215 (U.S. and Canada) or write *Dr. Dobb's Journal*, P.O. Box 56188, Boulder, CO 80322-6188. For book/software orders call 800-533-4372 (in California 800-356-2002).

FOREIGN NEWSSTAND DISTRIBUTOR: Worldwide Media Service Inc., 115 E. 23rd St., New York, New York 10010; 212-420-0588 FAX 212-420-1265.

(continued from page 12)

rithm is given in the text *Computer Graphics* by Donald Hearn and M. Pauline Baker (Prentice Hall, 1986). The derivation depends only upon elementary algebra, but may require somewhat greater mathematical maturity due to the notation used. The text also presents Pascal code for the algorithm. Another reference, which gives a limited explanation and a C code implementation of the algorithm, but which does not attempt to derive the algorithm, is *Graphics Programming in C* by Roger T. Stevens (M&T Books, 1988).

Joseph M. Hovanes Jr.
Pittsburgh, Pennsylvania

Forth Fan

Dear *DDJ*,

Here's 20 cents to fan the Flames of T.S. Kuhn's book, *The Structure of Scientific Revolutions*. It caused me to go cold turkey re. the tube for three days.

Martin Tracy reaffirmed my belief that Forth in its dialects offers the best presently available forum for discussion of "discrete mathematics" and the foundations of computing science. But I would like to see his work in the form of a bootable operating system and not a guest under another commercial product.

I confess that my own present work, "simpli-Forth," which is strongly tied to the 6502, still requires a fig-Forth boot to get off the ground. Perhaps if I work, I can learn enough about target compilers to create my own boot codes.

It seems to me that small operating systems with too-early emphasis on hiding or transportability may not be in the best interest of learners who seek to know in detail how their computing systems work. I would like to see small Forth systems place the user in a programming environment which makes plain the processes of his machine. That is why calls to DOS seem misplaced to me; I'd prefer that all of a small Forth system be available to the decompiler and user.

Would not a system for the programming of "smart" peripherals be more useful and general by omission of read-only memory? One could imagine modified error-handling, perhaps by redirecting the error-message stream to the calling device and transmitting a raise-error-request to it. But I remain convinced that the "smart" external should be executing a standard and expandable Forth kernel, albeit a minimal one, and that communication with it should be in the form of a standard, interpreted input stream.

The user of such a device could then load codes indicating how the forthcoming data is to be handled, followed by the commands to be executed and the data (e.g., 80 PRINTLINE THE QUICK BROWN FOX JUMPS OVER . . .). At the end of such a session, some command such as DONE would then forget the loaded object-behavior back to that formerly executing. Instead of relying on ROM to make our machines robust we would enter a new arena of opportunity for flexibility. It is time for a generation of peripherals which can follow the lead and dance.

On another subject, Brodie encourages us, "Use dumb words." One of the major differences between fig-Forth and Forth-83 is in the use of the STATE variable and its effect on words such as ' and LITERAL. In the process of learning to use STATE-sensitive words correctly, I, too, have been hopelessly confused from time to time. But the fully interactive capabilities possible in a modern Forth machine may require STATE-sensitive behavior.

For this reason I chose to write SIF (STATE @ IF) which may be used:

```
: TEST SIF COMPILE THEN DO-IT ;
                          IMMEDIATE
```

which will cause TEST to compile DO-IT if compiling else execute it (COMPILE is not IMMEDIATE). Although this example makes TEST equal to DO-IT, more-useful examples can be drawn. Another word might be ?COMPILE that would combine the effects of SIF, THEN, and IMMEDIATE and be used: : TEST ?COMPILE DO-IT ; so that all words using ?COMPILE would automatically be made IMMEDIATE.

"Use dumb words" is sound advice. But some quite interesting capabilities arise only when one uses correctly written words with STATE-sensitive behavior.

Jon W. Osterlund
Greeley, Colo.

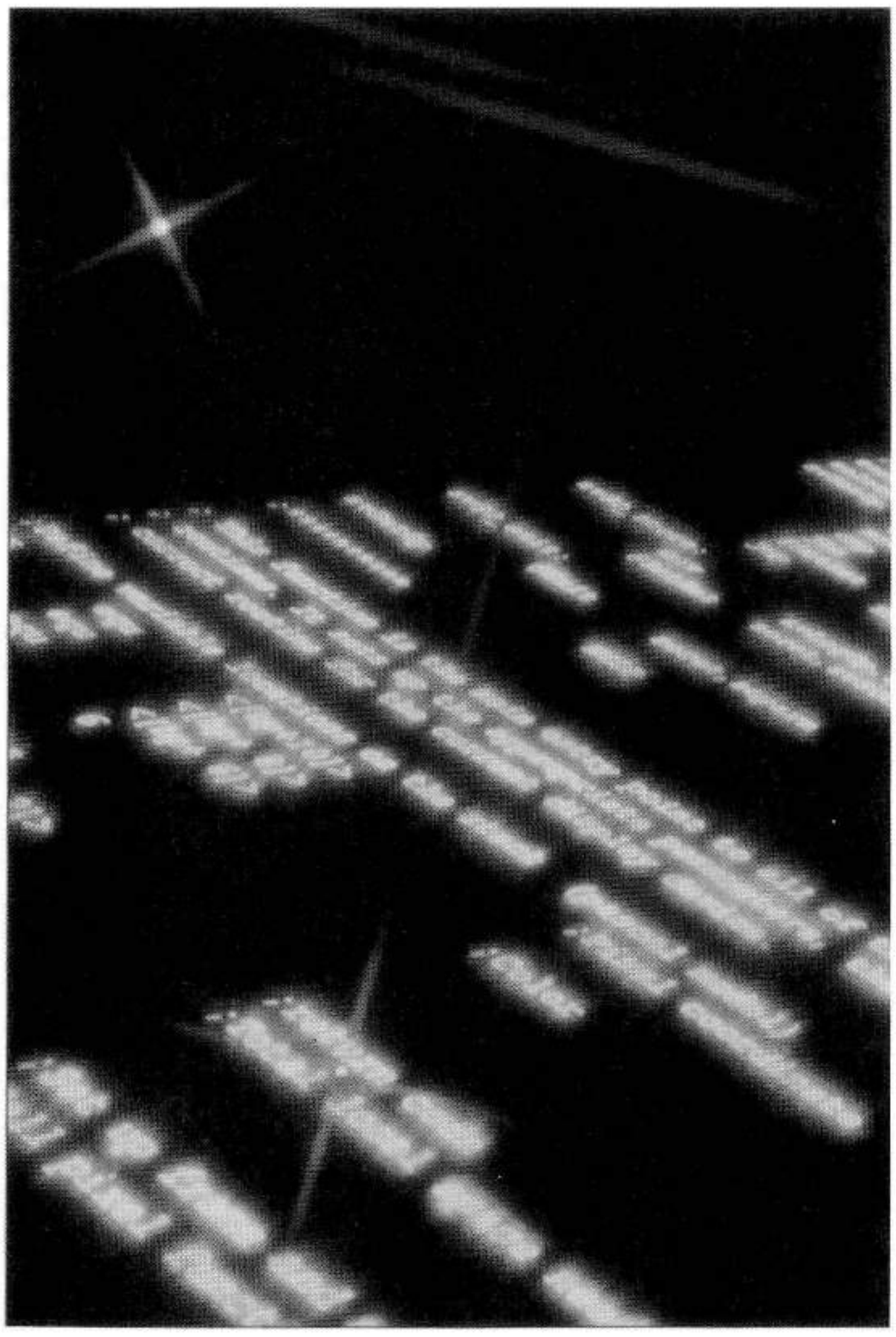
DDJ

We welcome your comments (and suggestions). Mail your letters (include disk if your letter is lengthy or contains code) to DDJ, 501 Galveston Dr., Redwood City, CA 94063, or send them electronically to CompuServe 76704,50 or via MCI Mail, c/o DDJ. *Please include your name, city, and state. We reserve the right to edit letters.*

Assembly Language Lives!

More Speed, Less Filling

Michael Abrash



There's an old joke that goes something like this:

Person #1: Help! My brother thinks he's a chicken, and I don't know what I should do.
Person #2: Have you told him the truth?
Person #1: I would, but I need the eggs.

Updated for the modern age of structured languages and object-oriented programming, that joke would read:

Manager #1: Help! My programmers think assembly language is a viable programming language, and I don't know what I should do.
Manager #2: Have you told them the truth?
Manager #1: I would, but I need the speed.

Assembly language beats everything else hands down when it comes to performance — especially when programming for the 80x86, where assembly language is wild, woolly, and wondrous — yet it gets no respect. When you flat-out need performance, there simply are no substitutes for assembly language — so why doesn't anyone seem to love it?

Assembly Language Isn't Cheap

Experts, pundits, and management types have been beating the drums for the demise of assembly language for years. There are many good reasons for wishing it dead. Compared to compiled code, good assembly-language code is harder to write, is more bug prone, takes more time to create, is harder to maintain, is harder to port to other platforms, and is more difficult to use for complex, multi-programmer projects. That makes assembly language an expensive, demanding, and time-consuming development language. Given the realities of time to market, the relative costs of good assembly language and high-level language programmers, programmer turnover, and ever-increasing

Michael works on high-performance graphics software at Metagraphics in Scotts Valley, Calif. He is also the author of Zen of Assembly Language *published by Scott, Foresman & Co., and* Power Graphics Programming, *from Que.*

software complexity, it's neither surprising nor unreasonable that most of the industry wishes assembly language would go away.

Assembly language lives, though, for one simple reason: Properly applied, it produces the best code of any language. By far.

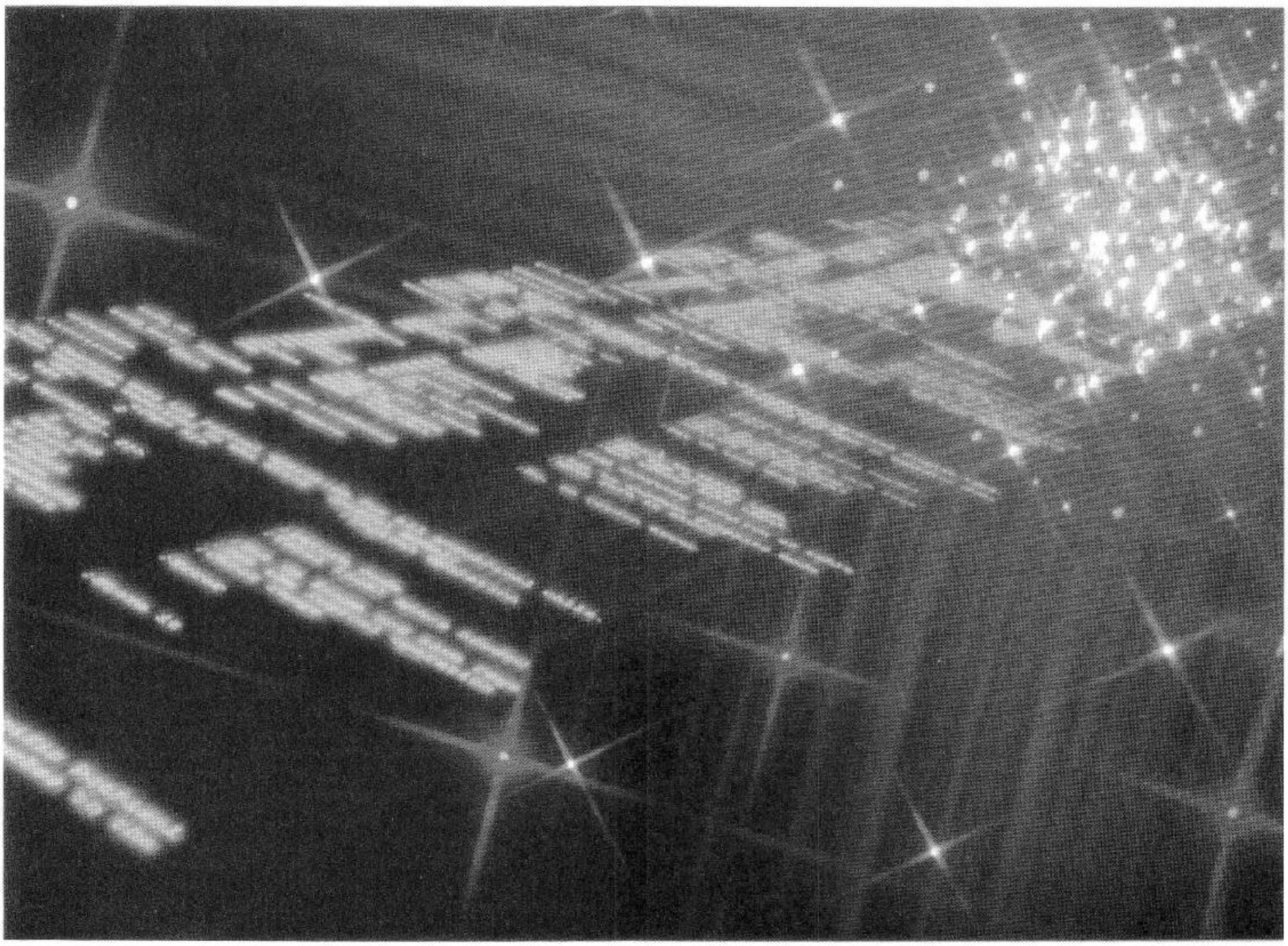
Assembly Language Lives

Don't believe me? Consider this. If the carbon-based computer between your ears were programmed with as good a compiler as Microsoft's, then you'd generate much better code in assembly language than does Microsoft C, because you know vastly more about what you want your program to do and are marvelously effective at integrating that knowledge into a working whole. High-level languages are artificially constrained programming environments, able to pass relatively little of what you know along to the ultimate machine code. There are good reasons for that: High-level languages have to be compilable and comprehensible by humans. Nonetheless, there's no way for a high-level language to know where to focus its efforts, or which way to bias code.

For example, how can a Pascal compiler know that one loop repeats twice, on average, while another repeats 32,767 times? How can a C compiler know that one subroutine is time critical, deserving of all possible optimization, while another subroutine executes in the background while waiting for the next key to be pressed, so speed matters not at all? The answer is: No way. (Actually, *#pragma* can do a little of that, but it's no more than a tiny step in the right direction.)

Just as significantly, no compiler can globally organize your data structures and the code that manipulates those structures to maximum advantage, nor take advantage of the vast number of potential optimizations as flexibly as you can. (Space forbids even a partial listing of optimization techniques for the 80x86 family: The list is astonishingly long and varied. See Tim Paterson's article in this issue for a small but potent sample.) When it comes to integrating all the information about a particular aspect of a program and implementing the code as efficiently as possible given the capabilities of a particular processor, it's not even close: Humans are much better optimizers than compilers are.

Almost any processor can benefit from hand-tuned assembly language, but assembly language lives most vibrantly in the 80x86 family. The 80x86 instruction set is irregular; the register set is small, with most registers dedicated to specific purposes; segments complicate everything; and the prefetching nature of the 80x86 renders actual execution time non-quantifiable — and optimization at best an art and at worst black magic — making the 80x86 family a nightmare for optimizing-compiler writers. The quirky



(and highly assembly language amenable) instructions of the 8086 live on in the latest 80x86-family processors, the 80386 and 80486, and will undoubtedly do the same for many generations to come. Other processors may lend themselves better to compilers, but the 80x86 family is and always will be a wonderland for assembly language.

Consider this: Well-written assembly language provides a 50 to 300 percent boost in performance over compiled code (more sometimes, less others, but that's a conservative range). An 8-MHz AT is about three times faster than a PC, a 16-MHz 80386 machine is about twice as fast as an AT, and a 25-MHz 80386 is about three times as fast as an AT. There are a lot of PCs and ATs out there — 20 to 30 million, I'd guess — and there is a horde of users contemplating the expenditure of thousands of dollars to upgrade.

When you absolutely, positively need to keep program size to a minimum, assembly language is the way to go

Now consider this. Those users don't have to upgrade — they just need to buy better-written software. The performance boost good assembly language provides is about the same as stepping up to the next hardware platform, but the assembly language route is one heck of a lot cheaper.

In other words, better software can eliminate the need for expensive hardware, giving the developer the opportunity to realize a healthy profit for his extra development efforts. Just as important is the fact that good assembly language runs perfectly well on slower computers, making the market for such software considerably larger than the market for average software. If you make your software snappy on an 8088, your potential market doubles instantly and the competition thins.

Finally, it's on the slower computers — the PC and AT — that assembly language optimization has the most effect (see the example later in this article), and that's precisely where improved performance is most needed.

Enter the User

So assembly language produces the best code. What of it? If high-level languages make it easier and faster to create programs, who cares if those programs are slower?

The user, that's who. Users care about perceived performance — how well a program seems to run. Perceived performance includes lack of bugs, ease of use, and, right at the top of the list, responsiveness. Hand users a whiz-bang program that makes them wait at frequent intervals, and they'll leave it on the shelf after trying it once. Give users a program that never gets in their way, and they may love it without ever knowing quite why. In these days of all-too-sluggish graphical interfaces, the performance issue is central to the usability of almost every program.

What users don't care about is how a program was made. Do you care how your car was designed? You care that it's safe, that it's reliable, and that it performs adequately, but you certainly don't care whether the manufacturer used just-in-time manufacturing, or whether mainframe or microcomputer CAD was used in the design process. Likewise, users don't care whether a programmer used OOP or C or Pascal, or COBOL, for that matter; they care that a program does what they need and performs responsively. That's not purely a matter of speed, but without speed the user will never be fully satisfied. And when it comes to speed, assembly language is king.

Use Only as Directed

When you need it, there's no substitute for assembly language, but it can be a drag when you don't need it — so know when to use it. Humans are better large-scale designers and small-scale optimizers than compilers, but they're not very good at the grunt work of compiling, such as setting up stack frames, handling 32-bit values, allocating and accessing automatic variables, and the like. Moreover, humans are much slower at generating code, so it's a good idea to avoid being a "human compiler." Some people create complex macros and assembly language programming conventions and do all their programming in assembly language. That works — but what those macros and conventions do is make assembly language function much like a high-level language, so there's no great benefit, especially given that you can drop into assembly language from a high-level language at any time just by calling an assembly language subroutine (or, better yet, by using in-line assembly language in a compiler that offers that feature, such as Turbo C). Unless you're a masochist, let your favorite compiler do what it's best at — compiling — and save assembly language for those small, well-defined portions of your software where your efforts and unique skills pay off handsomely.

A relevant point is that assembly language alone is not the path to performance. If you have a program that takes as long as a second to update the screen, you have problems that assembly language alone won't solve: Proper overall design and algorithm selection are also essential. However, most software designers consider the job done when the design and algorithm phases are complete, leaving the low-level optimization to the compiler. I repeat: No compiler can match a good assembly language programmer at low-level optimization. Given the irregular nature of the 80x86 family and the huge PC software market, it's well worth the time required to hand-optimize the few critical portions that control perceived performance. Only in assembly language can you take full responsibility for the performance of your code.

Don't Spit into the Wind

While I can't offer a cut-and-dried dictum on when to use assembly language, the practice of using it when the user would notice if you didn't is a good rule of thumb. While some programmers would take this rule too far and use assembly language too often, the vast majority of programmers will lean over backwards the other way, in the face of all evidence to the contrary. Hal Hardenberg's late, lamented DTACK Grounded reveled in the folly of the AT&T programmers who implemented the floating-point routines for a super-micro in C rather than assembly language — with the result that the computer performed floating-point arithmetic not quite so fast as a Commodore VIC-20!

Likewise, I once wrote an article in which I measured the performance of an assembly-language line-drawing implementation at four to five times that of an equivalent C implementation. One reader rewrote the C code for greater efficiency, ran it through Microsoft C rather than Turbo C, and wrote to inform me that I had shortchanged C; assembly language was actually "only" 70 percent faster than C. As it happens, the assembly-language code wasn't fully optimized, but that's not the important point: What really matters is that when programmers go out of their way to produce code that's nearly twice as slow (and in an important user-interface component, no less) in order to use a

(continued from page 18)
high-level language rather than assembly language, it's the user who's getting shortchanged. Commercial developers in particular can't afford to ignore this, and I suspect that most such developers are *DDJ* readers. If you're aiming to sell hundreds of thousands of copies of a program, you're guaranteed to have stiff competition. If you don't go the extra mile to provide snappy response, someone else will — and you'll be left out in the cold.

Assembly language lives, though, for one simple reason: Properly applied, it produces the best code of any language. By far

On the other hand, assembly language code is harder and slower to write, and pays off only in the few most critical portions of any program. There are limits to the levels of complexity humans can handle in assembly language, and limits to the development time that can be taken before a product must come to market. Identify the parts of your programs that significantly affect the performance perceived by the user (a code profiler can help greatly here), and focus your efforts on that code, with especially close attention to oft-repeated loops.

80x86 Assembly Language in Action

Enough talk. Let's look at an example of assembly language in action. Listing One, page 94, shows a C subroutine, *CopyUppercase*, that copies the contents of one *far* zero-terminated string to another *far* zero-terminated string, converting all lowercase characters to uppercase in the process. The subroutine consists of a single, extremely compact loop that should be ideal for compiler optimization. In fact, I organized the loop for the best results with Microsoft C 5.0, the test compiler, and used the intermediate variable *UpperSourceTemp* in order to allow for more efficient compiled code. There may be a more efficient way to code this subroutine, but if you're going to go to the trouble of being compiler-specific and knowing compiler code generation that intimately, why not use assembly language, which provides direct control and gives you the freedom to create the best possible code? Microsoft C 5.0 generates the code shown in Figure 1 from the version of *CopyUppercase* in Listing One when maximum optimization is selected with the /Ox switch. It's not bad code, but neither is it great. The *far* pointers are stored in memory and must be loaded each time through the loop, and a considerable amount of work is expended on determining whether each character is uppercase, although the case check is done with a table look-up, which is generally one of the most desirable 80x86 programming techniques. A serious failing is that none of the 80x86 family's best instructions — the string instructions — are used. The upshot is that Listing One runs in the times listed in Figure 2 on various PC-compatible computers. (All times discussed in this article were measured with the Zen timer described in my book *Zen of Assembly Language*, from Scott, Foresman & Company, modified slightly to work with Microsoft C.)

Can we do better in assembly language? Indeed we can, as Listing Two (page 94), which replaces the C version of

(continued from page 20)

CopyUppercase in Listing One with an assembly language version, illustrates. Listing Two simply keeps both *far* pointers in registers and uses string instructions to access both strings; the return for the 21 assembly-language instructions that do that is a performance improvement ranging from two to three-plus times, as shown in Figure 2. If this code happens to be in a performance-sensitive portion of a program, that's quite a return for a little assembly language.

Now, you may well think that the above example is biased in favor of assembly language, what with the *far* pointers, which assembly language tends to handle much better than do compilers. I would disagree: Almost every PC program now takes advantage of the full 640K of memory, and most of that memory must be accessed via *far* pointers, so access to *far* data is a most important issue to PC developers, and the ability of assembly language to handle *far* data just about as fast as near data is a substantial point in favor of assembly language. In fact, this example is representative of a large class of problems developers face, involving data copying, data transformation, data checking, pointers, and segments. Nonetheless, let's see what happens if we alter *CopyUppercase* to use *near* pointers.

Listing Three (page 94) shows Listing One changed to use *near* pointers. Listing Three, which generates the code shown in Figure 3, is indeed much faster than Listing One; it still takes at least half again as long as Listing Two, but it's closing the gap. By contrast, Listing Two wouldn't much benefit from *near* pointers, because it already keeps the pointers in the registers. Does that mean that for *near* data C almost matches assembly language?

Not a chance. We haven't optimized the assembly language implementation yet; Listing Two is just a straight port of Listing One from C to assembly language. Listing Four (page 94) shows Listing Two converted to use *near* point-

String type/ Language (Listing)	Execution time in microseconds on 8088	80286	80386
Far strings/C (Listing One)	2258 (1.0)	466 (1.0)	140 (1.0)
Far strings/ASM (Listing Two)	662 (3.4)	150 (3.1)	62 (2.3)
Near strings/C (Listing Three)	1183 (1.9)	282 (1.7)	95 (1.5)
Near strings/ ASM (Listing Four)	574(3.9)	115 (4.1)	50 (2.8)
Near strings/ optimized ASM (Listing Five)	410 (5.5)	85 (5.5)	46 (3.0)

Figure 2: *The execution times of the various C and assembly language implementations of* CopyUppercase *shown in Listings One through Five. For a given listing running on a given processor, the number in parentheses represents the performance of that listing relative to the performance of Listing One on that processor; the higher the value, the better the performance. 8088 timings were performed on an IBM XT; 80286 timings were performed on a 10-MHz one-wait-state AT clone; and 80386 timings were performed on a 20-MHz zero-wait-state 32K-cache Toshiba T5200*

```
_CopyUppercase      proc        near
        push        bp
        mov         bp,sp
        sub         sp,0002
Label1:
        les         bx,[bp+08]
        mov         cl,es:[bx]
        inc         word ptr [bp+08]
        mov         ax,cx
        cbw
        mov         bx,ax
        test        byte ptr [bx+0115],02
        je          Label2
        mov         ax,cx
        sub         al,20
        jmp         Label3
Label2:
        mov         ax,cx
Label3:
        les         bx,[bp+04]
        mov         es:[bx],al
        inc         word ptr [bp+04]
        or          cl,cl
        jne         Label1
        mov         [bp-02],cl
        mov         sp,bp
        pop         bp
        ret

_CopyUppercase      proc        near
```

Figure 1: *The code generated for* CopyUppercase *by Microsoft C 5.0 when Listing One is compiled with the /Ox switch (maximum optimization)*

```
_CopyUppercase      proc        near
        push        bp
        mov         bp,sp
        sub         sp,0002
        push        di
        push        si
        mov         di,[bp+04]
        mov         si,[bp+06]
Label1:
        mov         cl,[si]
        inc         si
        mov         ax,cx
        cbw
        mov         bx,ax
        test        byte ptr [bx+0115],02
        je          Label2
        mov         ax,cx
        sub         a1,20
        jmp         Label3
        nop
Label2:
        mov         ax,cx
Label3:
        mov         [di],al
        inc         di
        or          cl,cl
        jne         Label4
        mov         [bp+04],di
        mov         [bp+06],si
        mov         [bp-02],cl
        pop         si
        pop         di
        mov         sp,bp
        pop         bp
        ret
_CopyUppercase      proc        near
```

Figure 3: *The code generated for* CopyUppercase *by Microsoft C 5.0 when Listing Three is compiled with the /Ox switch (maximum optimization)*

(continued from page 22)
ers, plus a couple of twists. First, two bytes are loaded, converted to uppercase, and stored at once, cutting the number of memory-accessing instructions in half. Second, the value used to convert characters to uppercase and the upper- and lowercase bounds are stored in registers outside the loop, so that they can be used more efficiently inside the loop. These are simple optimizations, but ones that I doubt you'll find a compiler using — and they're highly effective. As Figure 2 indicates, Listing Four is approximately 20 percent faster than Listing Two and about two times faster than the *near* C implementation of Listing Three.

If you have a program that takes as long as a second to update the screen, you have problems that assembly language alone won't solve

We're not done optimizing yet, though. We've focused so far on relatively simple, linear optimization. Let's pull out all the stops, throw some unorthodox techniques at the problem, and see what comes of it.

On most PC compatibles, the key is this: The processor is slow at fetching instruction bytes and branching (in fact, all 80x86 processors are relatively slow at branching). If we can keep one or the other of those aspects from dragging the processor down, we can often improve performance considerably. As it happens, we can attack both bottlenecks. Look-up tables shrink code size, thereby easing the instruction fetching problem, and avoid branches as well. Well then, why not simply look up the uppercase version of each character? While we're at it, why not look it up with the remarkably compact and efficient *xlat* instruction? In this way we can convert the five instructions used to convert to uppercase in Listing Four to a single *xlat*. We can also improve performance by repeating multiple instances of the contents of the loop in-line, one after the other; doing this allows virtually all of the conditional jumps to fall through, eliminating branching almost entirely. Both changes appear in Listing Five, page 94. As Figure 2 indicates, those two changes improve performance by 8 to 40 percent — and the improvement is greatest on the slower 8088 and 80286 machines, which is surely where speed matters most. (Nor is this code maxed out even yet; I simply had to draw the line somewhere in the interests of keeping the code readily comprehensible and this article to a reasonable length. For example, we could use *lodsw* to speed up Listing Five much as we did in Listing Four. Never assume that your code is fully optimized!)

Bear in mind, too, that the code in Listing Five can handle *far* pointers as easily as *near* if the look-up table is moved into the code or stack segment and accessed with a segment override, a change that would scarcely affect performance at all. When it comes to handling *far* strings, then, we've improved performance by three to five and one-half times. To put that in perspective, the performance improvement gained by running the original C code on a 20-MHz zero-wait-state 32K-cache 80386 computer rather than a run-of-the-mill 10-MHz one-wait-state 80286 computer was only a little over three times. I think it's obvious which is the cheaper solution to improving performance.

(It's worth noting that carefully crafted assembly language

(continued from page 24)
was required to produce the massive performance improvement measured earlier. Assembly language by itself guarantees nothing, and bad assembly language, which is easy to write, brings new meaning to the word bad.)

Don't think I've picked an example that stacks the deck in favor of assembly language. In fact, assembly language would do considerably better if we worked with arrays or fixed-length Pascal-style strings, and would do better than compiled code in cases where there were more variables to keep in the registers. We also weren't able to use repeated string instructions in the earlier example; when such instructions can be used, as is often the case when an entire program's data structures are organized with efficient assembly language code in mind, the performance advantage of assembly language can approach an order of magnitude. In short, we looked at a simple, limited example (and actually one that lends itself relatively well to compiler optimization), and in optimizing it we've scarcely begun to tap the treasure trove of assembly-language tools and techniques.

Yes, compiler library functions can use string instructions and other assembly-language tricks as readily as your own assembly language code can, but there's a great deal that library functions can't do. Don't assume that library functions are well written, either — some are, but many aren't. And remember that the author of the library knows no more than the author of the compiler about when you most need performance, and so must design code for adequate performance under all circumstances. You, on the other hand, can precision-craft your code for best performance exactly when and where you need it. Also, keep in mind that library functions can work only within the current model. When you're working with data on the *far* heap in a program compiled with the small model (an efficient arrangement for programs that must handle a great deal of data), library functions can't help you.

Finally, Microsoft C is a very good optimizing compiler, considerably better than most of the compilers out there. There are a few compilers that generate somewhat better code than Microsoft C, but I'm willing to bet that most of the C programmers reading this use either Microsoft or Turbo C. (Turbo C did not match Microsoft C on this particular example, so I used Microsoft C in order to give C every advantage.) The C code was written to allow for maximum optimization (the loop is only four lines long, for goodness' sake) and uses a macro — not a function call — that expands to a table look up. In other words, the cream of the C crop, given readily optimized code and using a look-up table, went head-to-head with a few dozen hand-optimized assembly-language lines — and proved to be about two to five times slower.

Size Matters Too

I've focused on performance so far because the primary use of assembly language lies in making software faster. Assembly language can make for far more compact programs as well, although that's less often important because the PC has a large amount of memory available relative to processing power and because saving space is a diffuse effort, requiring attention throughout the program, while enhancing performance is a localized phenomenon, and so offers a better return on programming time.

There are cases where program size is crucial — memory-resident programs, device drivers, utilities, for example — and assembly language can work wonders. Of course, good assembly language code is very tight, and hence very small, but there's more to it than that. It's easy to drive programs with compact data strings in assembly language (see "Roll your Own Minilanguages with Mini-Interpreters" which I co-authored with Dan Illowsky, *DDJ*, September 1989). It's also easy to map in code sections from disk as needed; assembly language can be far more flexible than any overlay manager. Finally, assembly language eliminates the need for non-essential start-up and library code. Co-workers tell me of the time they needed to distribute a program to accept a keypress from the user and return a corresponding error level to a batch file. Written in C, the program was 8K in size; unfortunately, the distribution disk didn't have that much free space. Rewritten in assembly language, the same program was a mere 50 bytes long.

When you absolutely, positively need to keep program size to a minimum, assembly language is the way to go.

Can Live with It, Can't Live without It

Assembly language isn't the be-all and end-all of PC programming, but it is the only game in town when either performance or program size is paramount. Assembly language should be used only when needed and, used wisely, offers unparalleled code quality and an excellent return for programming time invested.

For all the drawbacks of assembly language, eight-plus years of PC software development have proven that developers can live with it; programs containing assembly language have been written in an expeditious manner and work very well, indeed. Those same years have shown that developers can't afford to live without assembly language. I suspect you'd be hard pressed to find any important PC software that contains no assembly language at all, and I can assure you that any application with a graphical user interface either contains assembly language or is a dog. (Sure, Windows applications and applications that link in third-party libraries may not contain assembly language, but that's because they've passed that responsibility off to other developers. And just who are those developers? *DDJ* readers, that's who. Somebody has to create the good code that top-notch software requires.)

For all the wishing, 80x86 assembly language isn't going away soon; in fact, it's not going to go away at all. The 80x86 architecture lends itself beautifully to assembly language, and performance will always be at a premium, no matter how fast processors get. Back, when I used a PC, I thought if I had a computer that was ten times faster, all my software would run so fast that I'd never have to wait. Well, now I use just such a computer, and much of the software I use is faster as well (MASM, for example, is about ten times faster than it used to be, and TASM is even faster) — and still I spend a lot of time waiting. Software is never fast enough, and better software is one heck of a lot cheaper than better hardware.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 94.)

Assembly Language
Tricks of the Trade

Hand-picked code for smaller, faster programs

Tim Paterson

It is the nature of assembly language programmers to always look for ways to make their programs faster and smaller. Over the years, the individual programmer develops a personal catalog of tricks and techniques that squeeze out a few bytes here or a few clocks there. My own catalog of 8086 tricks has been 13 years in the making, including a few from the 8080 that survived the translation.

One of the original motivations for finding some of these alternatives to the obvious approach is the severe "branch penalty" of the 8086 and 8088. When a conditional jump is taken on the 8086/8088, four times as many clock cycles are required (16) as when the jump is not taken. However, this penalty has been reduced on the 286 and 386. When taking a conditional jump, the newer processors require only seven clocks, plus one clock for each byte in the instruction at the target of the jump. That is, if you're jumping to an instruction that is 2 bytes long, the conditional jump takes nine clocks. This improvement means that several of the nine tricks presented here are of little or no value on the 286 and 386. However, I have presented them anyway so you'll know what they do if you see them. They are also still useful for code targeted to the 8086/8088.

For each of these tricks, I have compared its size and speed to the "direct" approach. Because the 286 is now the largest selling processor in PCs, I have used 286 clock counts to compare timing. When conditional jumps branch out of the presented code sequence, I assume the target instruction is 2-bytes long so that the branch would take nine clocks.

Tim is the original author of MS-DOS, Versions 1.x, which he wrote in 1980–82 while employed by Seattle Computer Products and Microsoft. He was also the founder of Falcon Technology, which was eventually sold, to Phoenix Technologies, the ROM BIOS maker. He can be reached through the DDJ *office.*

#1 Binary-to-ASCII Conversion

Converts a binary number in *AL*, range *0* to *0FH*, to the appropriate ASCII character.

```
  add   al,"0"                  ;Handle 0 - 9
  cmp   al,"9"                  ;Did it work?
  jbe   HaveAscii
  add   al,"A"-("9"+1)          ;Apply correction for 0AH-0FH
HaveAscii:
```

Direct approach: 8 bytes, 12 clocks for *0AH-0FH*, 15 clocks for *0–9*.

```
  add   al,90H    ;90H-9FH
  daa             ;90H-99H, 00H-05H+CY
  adc   al,40H    ;0D0H-0D9H+CY, 41H-46H
  daa             ;30H-39H, 41H-46H = "0"-"9", "A"-"F"
```

Trick: 6 bytes, 12 clocks.

#2 Absolute Value

Find absolute value of signed integer in AX.

```
  or    ax,ax           ;Set flags
  jns   AxPositive      ;Already the right answer if positive
  neg   ax              ;It was negative, so flip sign
AxPositive:
```

Direct approach: 6 bytes, 7 clocks if negative, 11 clocks if positive.

```
  cwd                   ;Extend sign through dx
  xor   ax,dx           ;Complement ax if negative
  sub   ax,dx           ;Increment ax if it was negative
```

Trick: 5 bytes, 6 clocks.

(continued on page 32)

(continued from page 38)

#3 Smaller of Two Values ("MIN")

Given signed integers in AX and BX, return smaller in AX.

```
 cmp    ax,bx
 jl     AxSmaller
 xchg   ax,bx          ;Swap smaller into ax
AxSmaller:
```

Direct approach: 5 bytes, 8 clocks if *ax* >= *bx*, 11 clocks otherwise.

```
 sub    ax,bx      ;Could overflow if signs are different!!
 cwd               ;dx = 0 if ax >= bx, dx = 0FFFFH if ax < bx
 and    ax,dx      ;ax = 0 if ax >- bx, ax = ax - bx if ax < bx
 add    ax,bx      ;ax = bx if ax >=bx, ax = ax if ax < bx
```

Trick: 7 bytes, 8 clocks. Doesn't work if |*ax* - *bx*| > 32K. Not recommended.

#4 Convert to Uppercase

Convert ASCII character in AL to uppercase if it's lowercase, otherwise leave unchanged.

```
    cmp    al,"a"
    jb     CaseOk
    cmp    al,"z"
    ja     CaseOk
    sub    al,"a" - "A" ;In range "a" - "z", apply correction
CaseOk:
```

Direct approach: 10 bytes, 12 clocks if less than "a" (number, capital letter, control character, most symbols), 15 clocks if lowercase, 18 clocks if greater than "z" (a few symbols and graphics characters).

```
sub    al,"a"           ;Lowercase now 0 - 25
cmp    al,"z" - "a" +1  ;Set CY flag if lowercase
sbb    ah,ah            ;ah = 0FFH if lowercase, else 0
and    ah,"a" - "A"     ;ah = correction or zero
sub    al,ah            ;Apply correction, lower to upper
add    al,"a"           ;Restore base
```

Trick: 13 bytes, 16 clocks. Although occasionally faster, it is bigger and slower on the average. Not recommended. Used by Microsoft C 5.1 *stricmp( )* routine.

#5 Fast String Move

Assume setup for a standard string move, with *DS:SI* pointing to source, *ES:DI* pointing to destination, and byte count in CX. Double the speed by moving words, accounting for a possible odd byte.

```
        shr     cx,1       ;Convert to word count
rep     movsw              ;Move words
        jnc     AllMoved   ;CY clear if no odd byte
        movsb              ;Copy that last odd byte
AllMoved:
```

Direct: 7 bytes, 10 clocks if odd, 11 clocks if even (plus time for repeated move).

```
        shr     cx,1       ;Convert to word count
rep     movsw              ;Move words
        adc     cx,cx      ;Move carry back into cx
rep     movsb              ;Move one more if odd count
```

Trick: 8 bytes, 9 clocks if even, 13 clocks if odd (plus time for repeated move). Not recommended.

#6 Binary/Decimal Conversion

The 8086 instruction *AAM* (ASCII adjust for multiplication) is actually a binary-to-decimal conversion instruction. Given a binary number in *AL* less than 100, *AAM* will convert it directly to unpacked BCD digits in *AL* and *AH* (ones in *AL*, tens in *AH*). If the value in *AL* isn't necessarily less than 100, then *AAM* can be applied twice to return three *BCD* digits. For example:

```
aam                ;al = ones, ah = tens & hundreds
mov    cl,al       ;Save ones in cl
mov    al,ah       ;Set up to do it again
aam                ;ah = hundreds, al = tens, cl = ones
```

AAM is really a divide-by-ten instruction, returning the quotient in *AH* and the remainder in *AL*. It takes 16 clocks, which are actually two clocks more than a byte *DIV*. However, you easily save those two clocks and more with reduced setup. There's no need to extend the dividend to 16 bits, nor to move the value 10 into a register.

The inverse of the *AAM* instruction is *AAD* (ASCII adjust for division). It multiplies *AH* by 10 and adds it to *AL*, then zeros *AH*. Given two unpacked *BCD* digits (tens in *AH* and ones in *AL*), *AAD* will convert them directly into a binary number. Of course, given only two digits, the resulting binary number will be less than 100. But *AAD* can be used twice to convert three unpacked *BCD* digits, provided the result is less than 256. For example:

```
;ah = hundreds, al = tens, cl = ones
    aad             ;Combine hundreds and tens
    mov   ah,al
    mov   al,cl     ;Move ones to al
    aad             ;Binary result in ax, mod 256
```

AAD takes 14 clocks, which is one clock more than a byte MUL. Again, that time can be saved because of reduced setup.

#7 Multiple Bit Testing

Test for all four combinations of 2 bits of a flag byte in memory.

```
    mov       al,[Flag]
    test      al,Bit1
    jnz       Bit1Set
    test      al,Bit2
    jz        BothZero
Bit2Only:
    . . .

Bit1Set:
    test      al,Bit2
    jnz       BothOne
Bit1Only:
```

Direct approach: 15 bytes, up to 29 clocks (to *BothOne*).

The parity flag is often thought of as a holdover from earlier days, useful only for error detection in communications. However, it does have a useful application to cases such as this bit testing. Recall that the parity flag is EVEN if there are an even number of "one" bits in the byte being tested, and ODD otherwise. When testing only 2 bits, the parity flag will tell you if they are equal — it is EVEN for no "one" bits or for 2 "one" bits, ODD for 1 "one" bit.

The sign flag is also handy for bit testing, because it directly gives you the value of bit 7 in the byte. The obvious drawback is you only get to use it on 1 bit.

(continued on page 34)

(continued from page 32)

```
        test        [Flag],Bit1 + Bit2
        jz          BothZero
        jpe         BothOne ;Bits are equal, but not both zero
;One (and only one) bit is set
.erre  Bit1 EQ 80H              ;Verify Bit1 is the sign bit
        js          Bit1Only
Bit2Only:
```

Trick: 11 bytes, up to 21 clocks (to *Bit1Only*).

Note that the parity flag is only set on the low 8 bits of a 16-bit (or 32-bit 386) operation. Suppose you test 2 bits in a 16-bit word, where 1 bit is in the low byte while the other is in the high byte. The parity flag will be set on the value of the 1 bit in the low byte — EVEN if zero, ODD if one. This is potentially useful in certain cases of bit testing, as long as you are aware of it!

Another example of using dedicated bit positions is to assign flags to bits 6 and 7 of a byte. Then test it by shifting it left 1 bit. The carry and sign flags will directly hold the values in those 2 bits. In addition, the overflow flag will be set if the bits are different (because the sign has changed).

Finally, there is a way to test up to 4 bits at once. Loading the flag byte into *AH* and executing the *SAHF* instruction will copy bits 0, 2, 6, and 7 directly into the carry, parity, zero, and sign flags, respectively.

#8 Function Dispatcher

Given a function number in a register with value *0* to *n – 1*, dispatch to the respective one of *n* functions.

```
;Function number in cx
    jcxz        Function0
    dec         cx
    jz          Function1
    dec         cx
    jz          Function2
    . . .
```

Direct approach 1: *3*n* - 4 bytes, *5*n* clocks maximum. Not bad for small *n (n < 10)*.

```
;Function number in bx
    shl         bx,1
    jmp         tDispatch[bx]
```

Direct approach 2: *2*n* + 6 bytes, 15 clocks. The best approach for large *n* when speed is a consideration.

```
;Function number in cx
    jcxz      Function0
    loop      NotFunc1
Function1:
    . . .
NotFunc1:
    loop      NotFunc2
Function2:
    . . .
NotFunc2:
    loop      NotFunc3
Function3:
    . . .
```

Trick: *2*n* - 2 bytes, *10*n* - 16 clocks maximum. Slow, but compact.

#9 Skipping Instructions

Sometimes a routine will have two or more entry points, but the only difference between the entry points is the first instruction. For example, the instruction that differs from one entry point to the next could be initializing a register to different values to be used as a flag later on in the routine.

(continued on page 36)

(continued from page 34)

```
Entry1:
        mov     al,0
        jmp     Body
Entry2:
        mov     al,1
        jmp     Body
Entry3:
        mov     al,-1
Body:
```

Direct approach: 10 bytes, 11 clocks (from *Entry1*).

Instead of using jump instructions to skip over the alternative entry points, a somewhat sleazy trick allows you to simply skip over those instructions. The technique goes back at least to 1975 with the first Microsoft Basic for the 8080. It became known as a "LXI trick" (pronounced "liksee"), after the 8080 mnemonic for a 16-bit *move-immediate* into register. Essentially, it allows you to skip a 2-byte instruction by hiding it as immediate data. A variation, the "MVI trick" (pronounced "movie"), uses an 8-bit immediate instruction to hide a 1-byte instruction.

Applied to the 8086, there is another variation. The skip can use a *move-immediate* instruction and destroy the contents of one register, or it can use a *compare-immediate* instruction and destroy the flags. Using the latter case the example above could be code such as this:

```
SKIP2F  MACRO
        db      3DH     ;Opcode byte for CMP AX,<immed>
        ENDM
Entry1:
        mov     al,0
        SKIP2F          ;Next 2 bytes are immediate data
Entry2:
        mov     al,1
        SKIP2F          ;Next 2 bytes are immediate data
Entry3:
        mov     al,-1
Body:
```

The effect of this when entered at *Entry1* is:

```
Entry1:
      mov           al,0
      cmp           ax,01B0H    ;Data is MOV AL,1
      cmp           ax,0FFB0H   ;Data is MOV AL,-1
Body:
```

Trick: 8 bytes, 8 clocks (from *Entry1*).

This trick should always be hidden in a macro. Here is a more complete macro that requires an argument specifying what register or flags to destroy. The argument is any 16-bit general register or "F" for flags.

```
SKIP2   MACRO   ModReg
IFIDNI  <ModReg>,<f>        ;Modify flags?
        db      3DH         ;Opcode byte for CMP AX,<immed>
ELSE
?_i     =       0
        IRP     Reg,<ax,cx,dx,bx,sp,bp,si,di>
IFIDN   <ModReg>,<Reg>      ;Find the register in list yet?
        db      0B8H + ?_i
        EXITM
ELSE
?_i     =       ?_i + 1
ENDIF   ;IF ModReg = Reg
        ENDM    ;IRP
.errnz  ?_i EQ 8            ;Flag an error if no match
ENDIF   ;IF ModReg = F
        ENDM    ;SKIP2

;Examples
        SKIP2   f           ;Modify flags only
        SKIP2   ax          ;Destroy ax, flags preserved
```

DDJ

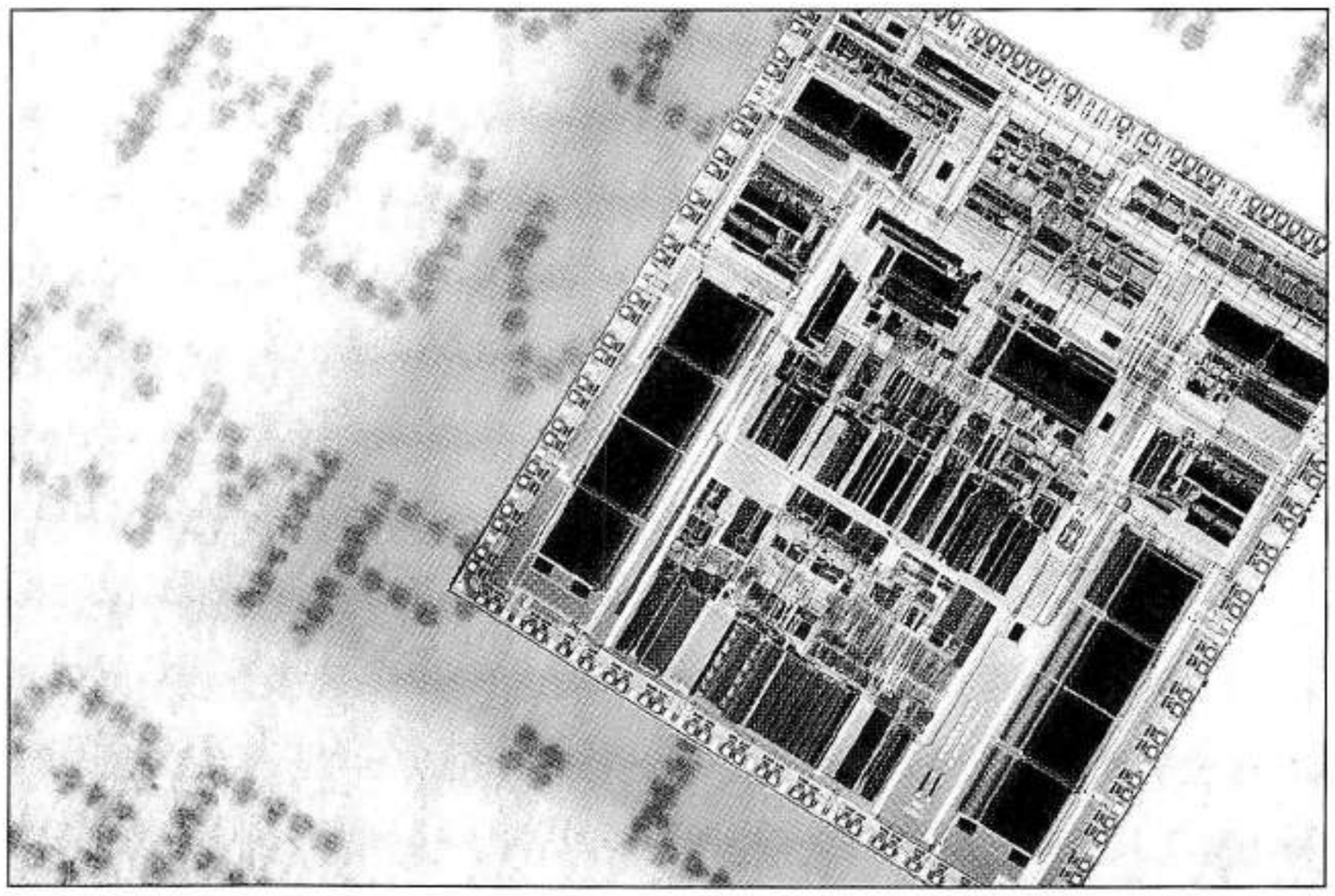
Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

68040 Programming

More than just an 030 with floating point

Stephen Satchell

The newest entry in the CPU chip wars is now ready for the system builders: The Motorola 68040. The first available chips will work at 25 MHz, with 33 MHz and faster parts becoming available later this year. Don't think, though, this is just a faster 68030: Motorola built in some nifty features to make multiprocessing hardware much easier to design and build.



68000 Family Overview

Motorola has gone to great pains to make a line of compatible 32-bit microcomputer chips. Like IBM did with the System/360 mainframe computers of the mid-1960s, Motorola made sure that applications code written for the earlier members of the 68000 family would run without modification on later chips. This scheme makes the assumption that programmers segregate I/O and chip control code from the rest of the system.

The general programming model for the 68000 family is the same: Eight 32-bit data registers, seven 32-bit address registers, one 32-bit user stack pointer, one 32-bit supervisor stack pointer, and chip-specific registers. The 68000 family supports operations on individual bits, 8-bit bytes, 16-bit words, 32-bit longwords, and packed binary coded decimal (BCD) data. Address calculations are all 32 bits, although some CPUs have limited addressing capability.

The 68008 (1980) is much the same as the Intel 8088 in that it talks to the outside world over a 20-bit address bus and an 8-bit data bus.

The 68000 (1979), the first CPU in the family, and the low-power CMOS 68HC000 use a 24-bit address bus and 16-bit data bus.

The 68010 (1982) takes the 68000 and adds virtual memory support, using an external memory management unit (MMU) and a special three-instruction "loop mode" that lets the 68010 execute a tight three-instruction loop repeatedly without fetching the instructions from memory more than once.

The 68020 (1984) is the first true 32-bit member of the 68000 family. The address and data busses are both a full 32-bits wide, allowing the chip to directly access four gigabytes (4096 Mbytes) of memory, up to 32 bits at a time. Memory management is provided by an external MMU. Instead of the 68010's "loop mode," the 68020 implements a 256-byte (64 × 4 direct mapped) instruction cache so that most loops run out of on-chip cache memory — improving execution time 33 percent and reducing the load on the system bus. Bit-field instructions let you deal with data of varying bit lengths. Instructions for multiprocessing were added into the 68020 as well.

The 68030 (1987) moves demand-page memory management on-chip, and adds a 256-byte (64 × 4 direct

Steve is free-lance writer and co-founder of Project Notify, a non-profit, emergency communications network. He can be reached at P.O. Box 8656, Incline Village, NV 89450 or on CompuServe at 70007,3351.

(continued from page 38)
mapped) data cache on-chip to complement the 68020's 256-byte instruction cache. The data cache uses a write-through philosophy. The bus system implements a burst transfer mode, that lets the chip effectively use page-mode, nibble-mode, and static-column DRAM to load data and instructions into cache memory quickly.

Enter the 68040

The newest member of the 68000 family, the 68040, essentially combines a beefed-up 68030 and the low-level functions of the 68881 floating-point coprocessor onto the same chip. The improvements, however, go much beyond that. Motorola's goal appears to be to make the 68040 as suitable as possible for large-scale multiprocessing systems.

Instead of one MMU trying to serve the entire chip, the 68040 gives you two: One for instructions, one for data. This keeps data and instruction accesses from causing page table entry faults (not to be confused with page faults) so as to minimize the amount of time the 68040 has to go to RAM to fetch address translation information.

The two on-chip memory caches are completely changed. Not only do you have a 4-Kbyte data cache and a 4-Kbyte instruction cache, but the cache system — particularly the data cache — is designed to minimize the number of times you have to go to the system bus. The two caches are organized as 64 four-way associative maps (256 locations), with 16 bytes of data in each cache location. The data cache can be write through, as it is in the 68030, or the 68040 can use a copyback philosophy that delays the *write* to memory until the chip needs the cache location for something else or the CPU's supervisor empties the cache.

When using cache in a multiprocessing system, you can have data that is one value in cache and another value in main memory. This problem is called "cache coherency." The 68040 takes care of this problem with "bus snooping" — the chip looks at the system bus, and when a *write* memory cycle is detected, any on-chip cache location containing data for the changed location is marked invalid.

What happens, though, when one 68040 has changed data, but hasn't written it back to DRAM yet? The bus snoop hardware has another trick up its sleeve. When a *read* memory cycle is detected, the 68040 checks its data cache to see if it changed the requested location; if so, it inhibits the RAM memory cycle and sends the correct data to the other CPU. This reduces the amount of work programmers have to do to keep data up-to-date.

If you do a lot of scientific work, watch out for the floating-point unit. On the 68040, the only floating-point operations supported are absolute value, add, branch on condition, compare, decrement and branch conditionally, divide, move, move multiple, multiply, negate, nop, restore internal state, save internal state, set on condition, square root, subtract, trap on condition, and test. Other operations supported by the 68881, such as the trig and logarithmic functions, have to be handled by software emulation.

Assembler Programming Considerations

Portability When writing code that needs to run on different systems, you need to limit yourself to those instructions common to all the 68000 family. (See Table 1 for those instructions to avoid.) In particular, pay attention to addressing modes. The 68020, '30, and '40 support some additional modes not found on the '00, '08, and '10. Also try to segregate chip-dependent functions from the rest of your program. This limits how much code has to be replaced as you shift from CPU to CPU. The majority of your code should be

(continued from page 40)

running in user mode anyway.

Loops The loop mode of the '10 is of limited use, being composed of a loop-able instruction and a DBcc instruction. Use this construct when you can on the off chance you end up running on a '10, such as one of the older Sun workstations. Where possible, try to keep loops under 256 bytes, the size

68010 from 68000 and 68008	
Move from CCR	Move from Condition Code register
Move from SR	Move from Status register
MOVEC	Move Control register
MOVES	Move Status register
RTD	Return and Deallocate
68020 from 68010 **Data alignment restriction dropped**	
Bcc	Branch conditionally (allow 32-bit displacements)
BFCHG	Test Bit Field and Change
BFCLR	Test Bit Field and Clear
BFEXTS	Bit Field Extract Signed
BFEXTU	Bit Field Extract Unsigned
BFFFO	Bit Field Find First One-bit
BFINS	Bit Field Insert
BFSET	Test Bit Field and Set
BFTST	Test Bit Field
BKPT	Breakpoint
CALLM	Call Module
CAS	Compare and Swap Operands
CAS2	Compare and Swap Dual Operands
CHK2	Check register against upper and lower bound
CMP2	Compare register against upper and lower bound (between)
cpBcc	Branch on CoProcessor condition
cpDBcc	Test CoProcessor condition Decrement and Branch
cpGEN	CoProcessor General function
cpRESTORE	CoProcessor Restore functon
cpSAVE	CoProcessor Save function
cpScc	Set on CoProcessor condition
cpTRAPcc	Trap on CoProcessor condition
DIVSL	Long signed divide
DIVUL	Long unsigned divide
EXTB	Extend byte to long
PACK	Pack binary coded decimal (BCD)
RTM	Return from Module (*not* "Read the manual")
TRAPcc	Trap conditionally
UNPK	Unpack binary coded decimal (BCD)
68030 from 68020	
(CALLM)	
PFLUSH	Invalidates specific entry in the address translation cache (ATC)
PFLUSHA	Invalidates all entries in the address translation cache (ATC)
PLOAD	Load an entry into the address translation cache
PMOVE	Load an entry into the address translation cache
PTEST	Get information about a logical address
(RTM)	
68040 from 68030	
CINV	Invalidate cache entries
(cpBcc)	
(cpDBcc)	
(cpGEN)	
(cpRESTORE)	
(cpSAVE)	
(cpScc)	
(cpTRAPcc)	
CPUSH	Push, then invalidate, cache entries
Floating-point instructions	
MOVE16	Move 16-byte block; block must be aligned
(PFLUSHA)	
(PLOAD)	
(PMOVE)	

Table 1: *680x0 family instruction set differences. An instruction or capability added or changed is in the open. An instruction or capability removed is in parens. For example, the CALLM instruction was removed in the 68030, so in the table it shows as (CALLM).*

(continued from page 42)
of the instruction cache on the '20. If a much-repeating loop can't be squeezed down that far, move seldom-executed code such as exception code outside of the loop. The longer you can stay in the cache, the faster that loop executes.

Loop Data In assembler, it is usually easier to whip through an array word by adjacent word, so most assembler language programmers won't have to concentrate on what order data gets accessed. If you are writing a table-driven package, though, pay attention to how table information makes you access data. Where possible, the table should be optimized so your program sweeps through any array. This is somewhat important on the '30, and much more important on the '40 — particularly in multiprocessing systems.

Tests Many times, you have to load one of two values into a register or location based on some test condition. The "IF . . . THEN . . . ELSE . . . " construction is easy to understand, but the multiple branches can play hob with instruction fetching. Instead, try " . . . IF . . . THEN . . . " where you set the less common value, perform the test, and conditionally branch around the more common value. The penalty on '00, '08, and '10 CPUs is almost zero, but the savings on the '20, '30, and '40 can be significant. In fact, the first way requires at least five instructions (test, branch-false, set-1, branch, set-2) while the other way saves one instruction (set-2, test, branch-false, set-1).

High-Level Language Considerations

Portability Chip-dependent functions usually have to be written in assembler, so make sure the design of the system routines are as generic as possible so you don't have to change applications code when the next gee-whiz feature is introduced in the 68050. You'll need to package separate interface modules for each chip. High-level code should always be run in user mode.

Loops If your compiler can optimize for the loop mode on the '10 or if the library includes routines to perform functions using loop mode, use them. When structuring loops that are executed often consider dropping structured programming practices to pack the loop as tight as possible. The goal is to get the loop within the 256-byte window of the instruction cache of the '20. Branches are much cheaper than function calls to get the seldom-used code out of the loop. You have more latitude with the larger cache on the '30 and '40.

Loop Data Be very careful when transversing arrays that you know *exactly* how your compiler is working. Fortran programmers need to remember that they have to vary the first subscript first in order to walk through data sequentially. For PL/I and Pascal programmers, most compilers require you to vary the last subscript first to sweep an array. C programmers need to remember that when accessing a multidimensional array using the array operators that are in the construct "a[i][j]", the fragment "a[i]" loads a pointer, then "<e>[j]" loads the desired word; use an intermediate pointer where possible to limit the amount of pointer loading when the first subscript is held locally constant.

Tests You are at the mercy of the compiler when it comes to ordering tests to save time. Because compilers vary so much in what they do, it probably isn't worth it to change the way you select values.

Conclusion

The 68040 is more than "just a 68030 with floating point" and more than Motorola's weapon to fight the Intel 80486. It is a well-designed product in its own right. Graphics programmers like the support for manipulating bits, particularly the bit-field instructions introduced by the '20 and continued in the '40.

Homegrown Debugging—*386 Style!*

Use hardware breakpoints to sniff through your C and assembly code

Al Williams

Although the installed base of 80386-based machines is ever increasing, most use this souped-up machine as a faster 8086. One of the problems in running the 80386 under DOS is that you lose many of the advantages of the 386. In addition, many of the 80386's powerful features are only usable in protected mode. Of course, developers loath to use special 80386 features because this can shut them out of the large 8086/80286 market.

Still, some features are usable while the 80386 is operating as an 8086 (the so-called "real mode"). For instance, the 80386 has powerful on-board hardware that allows sophisticated debugging techniques that require hardware debugging boards on other processors. This on-board hardware is available in real mode (as well as the other modes). With a little ingenuity, you can put this hardware to work while debugging programs.

This article puts a little of that kind of ingenuity in your hands by showing how you can use the 80386 hardware to debug your programs. I'll provide a program that can be included in your assembly code to establish breakpoints for the purpose of debugging either C or assembly language programs. In addition, I'll provide an example program and a quick utility that I'll explain shortly.

Al Williams is a staff systems engineer for Quad-S Consultants Inc. His current work includes a hypertext system, several expert systems, and a 386 DOS extender package. He can be reached at 2525 South Shore Boulevard, Suite 309, League City, TX 77573.

All examples presented in this article compile under either MASM 5.0 or Microsoft C 5.1.

BREAK386

BREAK386 (Listing One, BREAK386.ASM, page 96) is not a traditional debugger in the sense of, say, DEBUG or CodeView. By adding BREAK386 to your assembly language code, you can study it with code, data, and single-step breakpoints. You can also examine DOS or BIOS interrupts that your program calls. In addition, BREAK386 can add the same 386 hardware debugging to your Microsoft C programs.

BREAK386 provides functions to set up 386 debugging (*setup386( )*), set breakpoints (*break386( )*), and reset 80386 debugging (*clear386( )*). In addition, BREAK386 provides an optional interrupt handler (*int1_386( )*) that supports register, stack, and code dumps along with single stepping. You can use any of these functions from either C or assembly language.

There are cases where you may wish to modify *int1_386( )* or write your own interrupt handler. For example, you may want to send the register dumps to a printer and automatically restart your program. With C, you will often want the interrupt handler to print out variables instead of registers. I'll provide some example interrupt handlers in C in a later section.

Using BREAK386

You must assemble BREAK386 before you can use it. Be sure to change the *.MODEL* statement to reflect the model you are using. If you are using explicit segment definitions in assembly, you must decide how to integrate BREAK386's code and data segments with your own. Assemble BREAK386 with the */Ml* option to prevent MASM from converting all labels to uppercase. The resulting .OBJ file can be linked with your programs just as with any other object module.

If you are using programs (such as memory managers or multitaskers) that also use 386-specific functions, you may have to remove these programs before BREAK386 will function. The other program will usually report a "privilege exception" or something similar. Simply remove the other 386 programs and try again.

Adding 386 breakpoints to your program requires three steps:

- Call *setup386( )* to set the debug interrupt handler address
- Set up breakpoints with the *break386( )* call
- Call *clear386( )* before your program returns to DOS

Note that when calling these routines from assembly, the routine names contain leading underscores. For convenience, Listing Two (BREAK386.INC, page 102) contains the assembly language definitions to use BREAK386. Listing Three (BREAK386.H, page 102) contains the same definitions for C. BREAK386.INC also includes two macros, *traceon* and *traceoff*, which are used to turn single stepping on and off from within the program.

Figure 1 shows the output from a breakpoint dump when using *int1_*

(continued from page 46)

386( ). The hexadecimal number on the first line is the contents of the low half of the DR6 register at the time of the breakpoint. The display shows all 16-bit and segment registers (except FS and GS). Following that is a dump of 32 words of memory starting at the bottom of the stack (1CB1:09FA in the example). The first three words of the stack are from the debug interrupt. The first word is the IP register, followed by the CS register and the flags. A simple change in the interrupt handler can remove this extra data from the display (see "Detailed Program Operation" in the next section).

Below the stack dump is a dump of program code. This dump usually consists of 16 bytes; 8 bytes before the current instruction and 8 bytes at the instruction pointer. This is convenient for data breakpoints because they occur after the offending instruction. The dump shows the starting memory address (1B66:0049) followed by the bytes at that address. An asterisk marks the current CS:IP location, followed by the remaining 8 bytes. If IP is less than 8, the code dump will start at CS:0 resulting in fewer than 8 bytes before the asterisk.

The last line of the dump prompts you for further action. You can:

1. View your program's output screen. When you select this option, BREAK386 replaces the current screen with your program's original output. To restore the debugging screen, press any key.
2. Toggle the trace flag. This will switch the state of the trace or single-step flag, and continue the program in the same manner as the "C" command (see number 3). To determine whether or not tracing is on, examine the value of DR6. If bit 14 is set (4000 hex), tracing is on.
3. Continue execution of the program. Selecting this option will resume the program where it left off. The program will execute until the next breakpoint (if the trace flag is clear) or to the next instruction (if the trace flag is set).
4. Abort the program. This will cause the program to exit. Be careful, however, when using this selection. If you have interrupt vectors intercepted, expanded memory allocated, or anything else that needs fixing before you quit, the "A" command will not take care of these things unless you rewrite the interrupt handler or *clear386( )*. (Also, if your program spawns child processes, and the breakpoint occurred in the child, the abort command will terminate the child and the parent program will continue without breakpoints.)

```
Program breakpoint:OFF1
AX=0000 FL=7216 BX=0080 CX=0007 DX=06AA
SI=0000 DI=0A00 SP=09FA BP=0882
CS=1B66 IP=0051 DS=1BAD ES=1B56 SS=1CB1
Stack dump:( 1CB1 : 09FA )
0051 1B66 7216 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

CODE=1B66 : 0049 =6A 04 E8 3F 00 83 C4 08 * B9 14 00 8A D1 80 C2 41

<V>iew output, <T>race toggle, <C>ontinue or <A>bort? _
```

Figure 1: *Sample output from a breakpoint dump*

Listings Four and Five, page 102, show examples of using BREAK386 in assembly and C. The identifiers beginning with *BP_* are defined in BREAK386.H and

(continued from page 48)
BREAK 386.INC.

A few notes on these functions are in order. Your program must call *setup386( )* before any other BREAK386 calls. You should pass it a segment and an offset pointing to the interrupt handler. After calling *setup386( )*, you may use *break386( )* to set and clear breakpoints. Figure 2 shows the parameters *break386( )* requires.

You must keep in mind a few facts about the 80386 when setting breakpoints or tracing. First, 2- and 4-byte data breakpoints must be aligned according to their size. For example, it is incorrect to set a 2-byte breakpoint at location 1000:0015 because that location is on an odd byte. Similarly, a 4-byte breakpoint can monitor address 1000:0010 or 1000:0014 but not address 1000:0013. If you must watch an unaligned data item, you will have to set multiple breakpoints. For example, to monitor 2 bytes at 1000:0015, set a 1-byte breakpoint at 1000:0015 and another at 1000:0016.

Also, keep in mind that a data breakpoint will occur even if you only access a portion of its range. For instance, if

80386 Debugging Features

Most PC developers are familiar with some aspect of chip debug assistance. Even the 8088 has a breakpoint interrupt and a "single-step flag," which allows debuggers to trace code one instruction at a time. The 386 shares these same features with the earlier processors, but adds eight debug registers (two of which Intel reserves). These debug registers control the hardware breakpoint features.

Hardware breakpoints are much more powerful than ordinary breakpoints (such as those in DEBUG) for two reasons. First, hardware breakpoints don't actually modify your program. This means that you can set breakpoints anywhere, even in ROM. Also, a program can't overwrite a breakpoint when it modifies itself or loads an overlay. Second, it is possible to set breakpoints on data. A data breakpoint triggers when your program accesses a certain memory location.

Microsoft's CodeView implements a similar data breakpoint capability, called "tracepoints." To maintain compatibility with non-386 PCs, however, CodeView doesn't use 386 features. As a result, CodeView checks tracepoints after the execution of each instruction. This, of course, is terribly slow. By moving the tracepoints to 386 hardware, execution isn't slowed down at all. Actually, you will usually want to slow down execution just a bit (see the discussion of the exact bit). Even then, the slowdown in execution is imperceptible.

Because there are four debug address registers in the 80386, it is possible to have four active breakpoints at once. Each address register (DR0-DR3) represents a linear address at which a different breakpoint will occur. In protected mode, the concept of a linear address is not straightforward. In real mode, however, a linear address can easily be calculated from a segment/offset pair. Simply multiply the segment value by 10 hex (shift left 4 bits) and add the offset. For example, to set a data breakpoint at B800:0020 (somewhere in the CGA video buffer), you would need a linear address of:

B800 × 10 + 20 = B8020

Once you have loaded the address registers, you must enable the breakpoints you wish to use and tell the processor what type of breakpoints they are. This is done via the debug control register (DR7). DR7 contains bits to enable each breakpoint and to set their type individually (see Figure 4). You will notice that DR7 has global and local enable bits as well as global and local exact bits (explained shortly). The difference between the various global bits and local bits is only important when the 80386 is multitasking in protected mode. For the purpose of this article, they are the same.

The Exact Bits

The exact bits are flags to tell the 80386 to slow down. At first glance, this doesn't seem to be helpful, but a detailed look at the 80386 architecture reveals the purpose of this bit.

The 80386 gains some of its speed by overlapping instruction fetches and data fetches. This is an excellent idea when executing code, but causes problems in debugging data. Without the exact bit set, a data breakpoint will not occur at the instruction that caused the data access! Being somewhat of an inconvenience, Intel included the GE/LE bits. With either (or both) of them set, data breakpoints will occur immediately after the instruction that caused them, although the processor

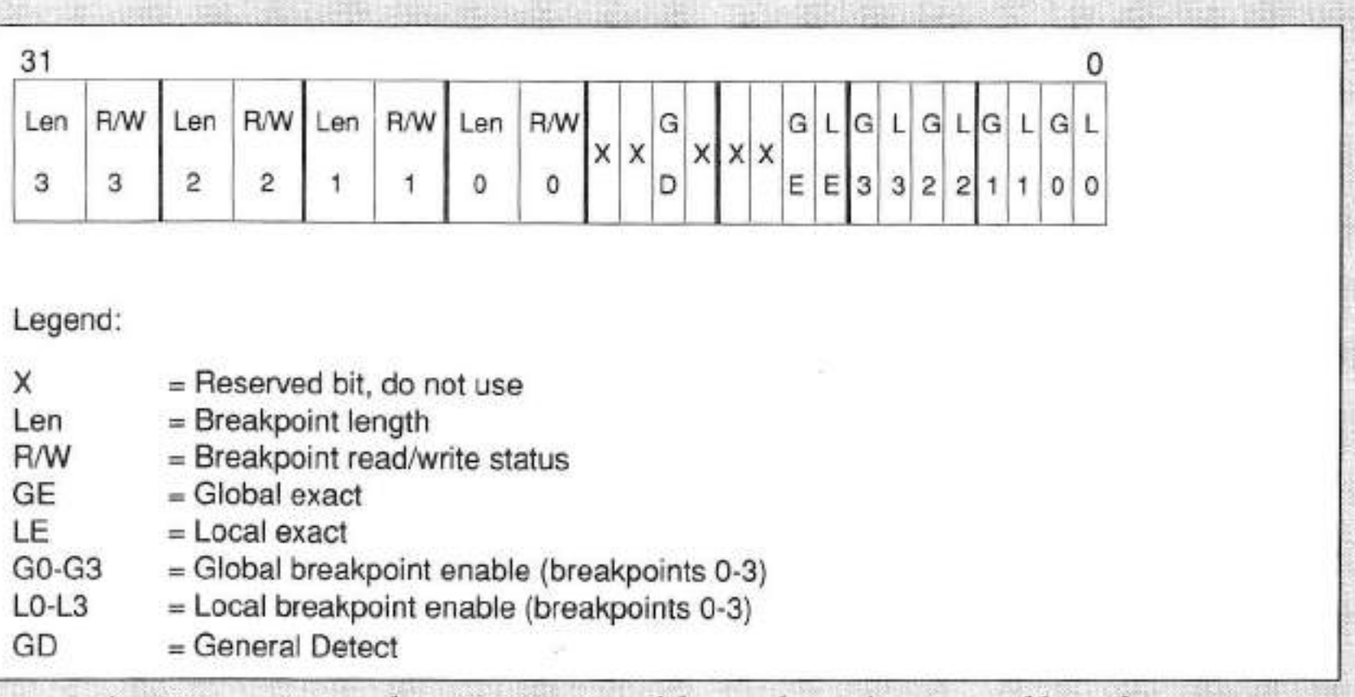


Figure 4: *Bits contained in DR7 to enable and set the type of breakpoints*

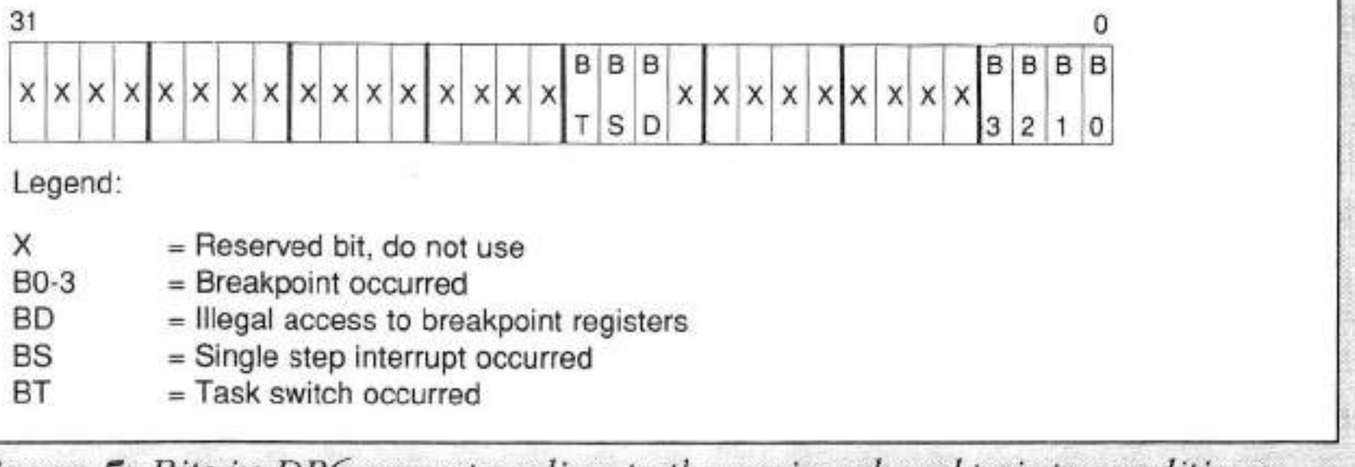


Figure 5: *Bits in DR6 corresponding to the various breakpoints conditions*

The original Volume 15 book had a printing error:
pages 223-254 were missing and
pages 255-286 were repeated twice.

The missing pages were recreated in this PDF by taking the equivalent pages from the March 1990 magazine issue.

you are monitoring a word at 2200:00F0 and a program writes a byte to 2200:00F1, a breakpoint will occur.

Setting a data breakpoint with *break386( )* will also set the global exact bit. When all the data breakpoints are either reassigned or deactivated, *break386( )* will clear the exact bit.

Because *int1_386( )* always sets the resume flag, you will find that a code

will lose a slight amount of speed.

Other Bits

All debug breakpoints generate an interrupt 1. To distinguish the various breakpoints, you must read the debug status register (DR6). DR6 has bits corresponding to the various breakpoint conditions (see Figure 5). Note the BT flag at bit 15. As with the local bits in DR7, only multitasking systems use the BT flag. Therefore, the flag is not considered in this article. The 386 never clears the bits in DR6, so after you determine what caused the interrupt, you should clear DR6.

The Only Other Bit We Haven't Discussed is . . .

With the general detect (GD) bit set in DR7, the 80386 prohibits access to the debug registers. Any attempt to access the debug registers will cause an interrupt 1 with the BD flag set in DR6. Intel's in-circuit emulator uses this feature, although you can use it if you have any reason to disable or control access to the debug registers. When a GD interrupt occurs, the interrupt handler is invoked and the GD bit is cleared. Otherwise, the routine would fault (with an endless loop) when the interrupt routine attempted to read DR6.

You can decide from the interrupt routine whether to terminate the user program, or to allow access to the registers. BREAK386 does not use the GD bit.

The Resume Flag

The last consideration with breakpoint interrupts is how to resume the interrupted program. If we simply return (as in a normal interrupt), there is nothing to stop a code breakpoint from occurring again immediately. The resume flag (found in the flag's register) prevents this from occurring. This flag inhibits further debug exceptions while set, and resets automatically as soon as one instruction successfully executes. Control of the resume flag is automatic in protected mode. Handling it from real mode, however, is somewhat of a trick, as seen in BREAK386. — A.W.

breakpoint that immediately follows a data breakpoint won't work. I'll show how this can be rectified shortly.

Because *INT* and *INTO* instructions temporarily clear the trace flag, BREAK-386 will not single step through interrupt handlers. If you wish to single step through an interrupt routine, you will have to set a breakpoint on its first instruction. A replacement for *int1_386( )* might emulate *INT* and *INTO* instructions to solve this problem.

Because BREAK386 uses BIOS keyboard and video routines, take care when placing breakpoints in these routines. In addition, single-stepping BIOS keyboard and video routines should be avoided. If you must debug in these areas, reassemble BREAK386 so that it doesn't use BIOS (see the DIRECT equate in BREAK386.ASM). Note, however, that many of its features will no longer function. Finally, you should avoid setting breakpoints in BREAK386's code or data.

BREAK386.INC contains two macros, *traceon* and *traceoff*, that can be used to control tracing. You may insert them anywhere in your code to enable or disable tracing. Remember, however, that you will see the *traceoff* macro as well as your own code when single stepping.

The function *clear386( )* must be called prior to exiting the program. This turns off the breakpoint handlers. If you fail to call *clear386( )* for any reason (a control-break, or a critical error), the next program that uses a location you have breakpointed will cause the break to occur. This can have unfortunate consequences because your interrupt 1 handler is probably no longer in memory. If you find that you have exited a program without turning off debugging and you have not encountered a breakpoint, run DBGOFF (Listing Six, page 104) to turn off hardware debugging.

With some care, BREAK386 can be used with other debuggers. In CodeView, for example, BREAK386 seems to work fine, as long as you are not single stepping (via CodeView). When you single step data breakpoints will be ignored and BREAK386 code breakpoints will "freeze" CodeView at that step. If you are using BREAK386 with CodeView, it is probably a good idea to leave the code breakpoints and single stepping to CodeView.

Detailed Program Operation

BREAK386 (Listing One) begins with the .386P directive, which ensures that MASM 5.0 will generate references to the debug registers. Be careful to place the .MODEL directive before the .386P,

(continued from page 52)
otherwise 32-bit segments will be generated (which doesn't work well with unmodified DOS!).

The parameters you may want to change are near the top of the source file. The equate to DIRECT controls the video mode. If DIRECT is 0, BREAK386 uses BIOS for input and output. If, however, you want to poke around in the keyboard or video routines, you must set DIRECT to 1. This causes BREAK386 to use direct video output for the debug dump. It will share the screen with your program (no video swapping) and breakpoints will simply terminate the program in a similar manner to the "A" command mentioned earlier.

You can change the STKWRD equate to control how many words are dumped from the stack when using *int1_386( )*. Setting STKWRD to zero will completely disable stack dumping. Similarly, if you set INTSTACK to zero, the display will not show the IP/CS/FLAGS at the top of the stack. If you are writing your own interrupt handler and don't need *int1_386( )*, you can assemble with *ENABLE_INT1* set to zero to reduce BREAK386's size.

The operations of *start386( )*, *clear386( )*, and *break386( )* are fairly straightforward. The implementation of *int1_386( )* deserves some comment. It is important to realize that *int1_386( )* only debugs non-386-specific programs because it only saves the 16-bit registers and the 8086 segment registers (*int1_386( )* does not destroy FS and GS). Because *int1_386( )* only runs on a 386, it does use the 32-bit registers. You can easily modify *int1_386( )* to save all the 386 registers, but it requires more space on the interrupted program's stack.

The most difficult aspect of the interrupt handler is managing the resume flag. The code below label c1 converts the three words at the top of the stack into six words so that setting the resume flag is possible. There are three things to remember about the way the resume flag is managed:

2- and 4-byte data breakpoints must be aligned according to their size. For example, it is incorrect to set a 2-byte breakpoint at location 1000:0015 because that location is on an odd byte

1. As mentioned earlier, *int1_386( )* always sets the resume flag. As a consequence, a code breakpoint that occurs immediately after a data breakpoint will not cause an interrupt. This is due to the resume flag being set even though the instruction that generated the data breakpoint has already executed. When the program restarts, the next instruction will execute with the resume flag set. This could be rectified by not setting the resume flag in the interrupt handler when processing data breakpoints.
2. An interrupt handler written entirely in C has no way to manipulate the resume flag properly. Listing Seven, page 104, however, shows two assembly language functions that allow you to write your handler in C. (See the next section for more details on writing C interrupt handlers.)
3. In real mode, hardware interrupt handlers (for example, those in the BIOS) will probably not preserve the resume flag. This means that if your code runs with interrupts enabled, there is some chance that one breakpoint will cause two interrupts. This chance increases greatly if interrupts remain disabled during the interrupt 1 processing. Why is this true? If the 80386 receives a hardware interrupt just before executing an instruction with the resume flag set, it will process that interrupt. When the interrupt returns, the resume flag is clear and the breakpoint occurs again. When interrupts are disabled during breakpoint processing, it is far more likely that an interrupt is pending when the program restarts. If interrupts were enabled while processing the debug interrupt, however, there is little chance of this happening. If it does, simply press "C" (when using *int1_386( )*).

```
retcode=break386(n,type,address);
where:
  n is the breakpoint number (from 1 to 4).
  type is the type of breakpoint. This should be one of the manifest constants defined in
  BREAK386.H (or BREAK386.INC). If you are clearing the breakpoint, the type is not
  meaningful.
  address is the address to set the breakpoint. This must be a far address (that is, one
  with both segment and offset). If you are using small model C, you should cast the
  pointer to be a far type (see the example). To clear a breakpoint, set address to
  0000:0000 (or a far NULL in C).
retcode is returned by the function. A zero indicates success. A non-zero value means
that you tried to set a breakpoint less than 1 or greater than 4. Note that the type
parameter is not checked for validity.

The types available are:
        BP_CODE      - Code breakpoint
        BP_DATAW1    - One byte data write breakpoint
        BP_DATARW1   - One byte data read/write breakpoint
        BP_DATAW2    - Two byte data write breakpoint
        BP_DATARW2   - Two byte data read/write breakpoint
        BP_DATAW4    - Four byte data write breakpoint
        BP_DATARW4   - Four byte data read/write breakpoint
```

Figure 2: *The parameters required by* break386( )

Advanced Interrupt Handlers in C

It is possible to write an interrupt handler completely in C to monitor data breakpoints. The handler must be declared as a *far* interrupt function. For example, the following function could be linked with the example in Listing Five:

```
  void interrupt far
new1(Res,Rds,Rdi,Rsi,Rbp,Rsp,Rbx,Rdx,
                                Rcx,Rax)
  {
  printf("\nBreakpoint reached.\n");
  }
```

By calling *setup386new1( )* instead of *setup386(int1_386)*, *new1( )* will be invoked for every breakpoint. Your function can read and write the interrupted program's registers using the supplied parameters (*Rax*, *Rbx*, and so on). Keep in mind that you cannot use this technique for code breakpoints. C's inability to manipulate the resume flag will cause an endless loop on a code breakpoint.

Listing Seven, provides the functions to write interrupt handlers in C. The procedure is much the same as described earlier, except that you must

(continued on page 57)

(continued from page 54)
call *csetup386( )* instead of *setup386( )*. The argument to *csetup386( )* is always a pointer to an ordinary *far* function (even in small model).

The actual interrupt handler is *_cint1_386( )*. This function will call your C code when an interrupt occurs. *_cint1_386( )* passes your routine two arguments. The first argument, a *far* void pointer, is set to the beginning of the interrupted stack frame (see Figure 3 for the format of the stack frame). The second argument is an unsigned long *int* that contains the contents of DR6.

All registers, and local variables on the stack can be read using the pointer to the stack frame (if you know where to look). In addition, all values (except SS) can be modified. It is usually wise not to modify SP, CS, or IP.

_cint1_386( ) switches to a local stack. The size of the stack can be controlled using STACKSIZE (near the top of Listing Seven). Be sure to adjust the stack if you need more space.

Listing Eight (page 105) shows an example of an interrupt handler in C. The example interrupt handler displays a breakpoint message and allows you to continue with or without breakpoints, abort the program, or change the value of a local variable in the *loop( )* function.

Future Directions

Many enhancements and modifications are possible with BREAK386. By altering the words on *int1_386( )*'s stack, for example, you can modify registers. You can redirect output to the printer (although you can screen print the display now) by replacing the OUCH routine. Perhaps the most ambitious enhancement would be to use BREAK386 as the core of your own debugger. You could write a stand-alone debugger or a TSR debugger that would pop up over another debugger (DEBUG or CodeView).

Keep in mind that 386 hardware breakpoints aren't just for debugging. The data breakpoint capability has many uses. For example, you might want to monitor the BIOS keyboard typeahead buffer's head and tail pointers to see when a keystroke is entered or removed. In this manner you could capture the keyboard interrupt in such a way that other programs couldn't reprogram your interrupt vector.

You can also use data breakpoints to detect interrupt vector changes or interrupt processing. Some assembly language programs could use data breakpoints for automatic stack overflow detection. Programs that decrement the stack pointer without using a push instruction (Microsoft C programs, for example) are not candidates for this type of stack protection.

Debugging with 386 assistance is quite practical and useful. The programs presented here should get you started and help you develop your own programs with this powerful hardware feature.

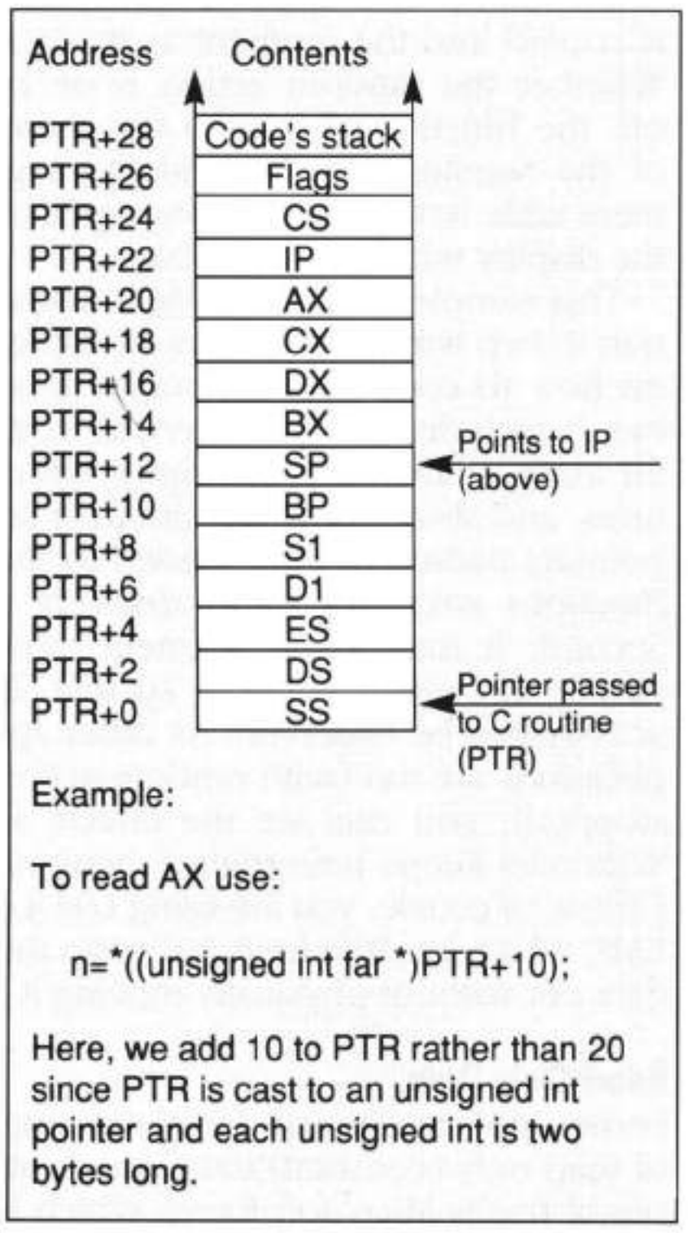


Figure 3: *Stack frame passed to the C interrupt handler*

Bibliography

Turley, James L., *Advanced 80386 Programming Techniques*, Osborne McGraw-Hill, Berkeley, Calif., 1988.

Intel Corporation, *80386 Programmer's Reference Manual*, Intel Corp., Santa Clara, Calif., 1986.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 96.)

Part II

Managing Multiple Data Segments Under *Microsoft Windows*

The segment table provides a little-known way of managing multiple data segments

Tim Paterson and Steve Flenniken

In last month's installment, we presented a method for managing multiple data segments under MS Windows using a little-known Windows feature, the segment table, along with a library of macros and functions to assist in applying the technique. For this month's installment, we've prepared a sample Windows program called "segments" that demonstrates the *segtable* library. In its "random action" phase, it randomly allocates, reallocates, and frees global memory. A window displays statistics about each memory block, including its *pSeg* (the address of its *SegmentTable* entry), the current segment number, the previous segment number, and the number of times it has moved since it was allocated. A timer function is used to keep the window continuously updated, even when another application has the input focus.

The sample application in Listing One (page 106) uses one segment as the place to keep track of all the other segments that it fiddles with and displays in the window. That segment contains an array of structures, one for each additional segment. Because it is referenced so often, the macro *FARDATAP* is defined to return the far pointer to the first structure in this segment. Listings Two through Five (beginning on page 108) provide the rest of the files required by the application.

The menu bar is used to start and stop the random action mode. When on, the timer function picks one of the structures. If the structure does not yet have a *pSeg*, it allocates one with a random amount of memory. If it already has a *pSeg*, it will do one of three things: Reallocate the *pSeg* with a different memory size; free the data, but keep the *pSeg*; or free the *pSeg* altogether. Whenever a segment is allocated or reallocated, a text string containing the last action ("A" for allocate or "R" for reallocate) and the size of the segment (for example, "1484 bytes") is copied into the segment as its data. Whether the random action is on or off, the function checks to see if any of the segment numbers in the segment table have changed, and updates the display window if they have.

This sample is a useful demonstration in two ways. First, it has examples on how to code with the segment table. It includes many references to its far array of memory descriptor structures, and shows how IFP (indirect far pointer) parameters are passed to the functions *strcpyifp( )* and *strlenifp( )*. Second, it makes the segment table visible through a window so that its activity can be observed. As other applications are run (with random action stopped), you can see the effects as Windows keeps rearranging memory. Unless, of course, you are using LIM 4.0 EMS, which lets Windows just swap the data out without physically moving it.

Read-Only Data

Some applications use large amounts of read-only (constant) data. An example of this is Microsoft Excel, which is written in C and compiled into pcode, not native 8086 code. The pcode is

Tim is the original author of MS-DOS, Versions 1.x, which he wrote in 1980-82 while employed by Seattle Computer Products and Microsoft. He was also the founder of Falcon Technology, which was eventually sold to Phoenix Technologies, the ROM BIOS maker. Steve formerly worked at Seattle Computer Products, Rosesoft (makers of ProKey), and is now with Microrim, working with OS/2 and Presentation Manager. Both can be reached c/o DDJ.

(continued from page 58)
data, not code, because it is never actually executed. Other applications could simply have large amounts of data in the form of tables or other structures.

Like code, read-only data should be marked as discardable in the linker definition file. This allows Windows to throw it away to make room, but reload it from disk later when needed. Another good practice is to keep segment size to less than 16K, the size of the LIM 3.2 expanded memory page frame. Windows can then choose to use space in EMS for those segments that fit, entirely transparent to the application.

Code and read-only data don't sound any different so far, but there is an important distinction. Windows keeps track of how often each code segment is used, in order to help it make a good decision on discarding one when it needs to free some memory. It does this with the reload thunk. Every far call to a discardable segment actually calls a thunk specific to that entry point. If the segment being called is present in memory, the thunk will contain a jump to the entry point. If the segment is not loaded, the thunk will cause Windows to load it. Either way, the thunk also notes the fact that a call to that segment was made. Windows uses a least-recently used (LRU) algorithm for determining the best segment to discard when memory is needed. The thunks are the source of its information.

The easiest way to deal with discardable read-only data segments is to put a little code in them. These lines of assembly language belong in each segment (but with a unique entry point name for each):

```
Load_This_Segment:
    mov     ax,cs
    retf
```

To ensure that a segment is loaded, and to find out where, call this entry point. The return value in *ax* is the segment of the data. The call to this entry point is, of course, actually a call to a thunk that ensures the segment is loaded.

The segment number returned by this call can be stuffed into an empty entry in *SegmentTable* so that it will stay updated in case of movement. But recall that this segment can also be discarded. In that case, Windows will update the segment table with the (even-numbered) handle for that segment. This complicates things a bit. Now we could make a reference to the segment table and find an even number, indicating that the segment we want has been discarded. Calling the entry point (the reload thunk) is the easiest way to bring the segment back.

Once the segment has been loaded, we can use it as much as we want, as long as Windows doesn't discard it. But if we never call the segment's entry point again, Windows will think we've stopped using it — after all, it's the calls through the thunk that keep track of usage. Without periodic calls to the entry point, this segment will be one of the first to be discarded, no matter how much we've actually been using it.

Fortunately, Windows provides a mechanism to remind us to call the entry point periodically. On a regular basis (typically every fourth timer tick, or 4.5 times per second), Windows performs an "LRU sweep." One of the things Windows will do during the LRU sweep is to fill part of our segment table with zeros. The number of words set to zero is specified in *SegmentTable[1]*; the zero fill starts at *SegmentTable[2]*. In addition, *SegmentTable[1]* itself is also set to zero, which means nothing will be zero-filled again until it is reset to some value. This use of *SegmentTable[1]* suggests using a macro to give it the name *cwClear*.

Some applications use large amounts of read-only (constant) data. An example of this is Microsoft Excel

The idea is to set aside the first portion of the segment table for read-only data. At every LRU sweep, Windows will zero fill the segment numbers that were stored in there. When we try to access a segment number that has been zeroed, we will see an even number and conclude it was discarded. Then we call the segment's entry point to reload it, and the thunk will record the activity. Hopefully, this will prevent the segment from being discarded while it is still needed. The overhead of zero filling the table and calling the entry point is quite small compared with the time to reload a segment from disk.

Note that the *segtable* library, as written, is not set up for this type of use. The non discardable data segments, such as *segDgroup*, must be moved above the zero-fill area in *SegmentTable*. Because there are a fixed number of read-only data segments, they would probably each have their own fixed

(continued from page 60)
segment table entry. New access macros would be required that could deal with a segment that was not present.

Debugging Considerations

Microsoft considers the ideal environment for running Windows to be a 80386 computer with extended memory running 386MAX by Qualitas of Bethesda, Maryland. 386MAX puts the computer into Virtual 8086 Mode and manages memory by using the 386's paging mechanism. It provides three important benefits for Windows. First, it fully emulates LIM 4.0 expanded memory (EMS). Second, it performs the same function as the Windows program HIMEM.SYS, making available the first 64K of extended memory for use by Windows. Third, it allows TSR programs such as mouse and network drivers to be loaded out of the way of conventional memory — the base 640K memory space.

When Windows finds itself loaded into a computer with LIM 4.0 EMS, and there's a fair amount (like 256K) of conventional memory left, it will use "large frame" EMS. This means that the base 640K memory space becomes part of the EMS page frame. Windows can then swap different logical memory pages into the base 640K.

While this is a great way to run Windows, especially when running several large applications, it's not so good for debugging with Symdeb. Symdeb seems to get confused by the EMS swapping, and we've gotten some very strange results. Now we always disable 386MAX whenever we will be debugging a Windows program with Symdeb. On the other hand, CodeView for Windows is apparently so large that Windows doesn't use large frame EMS. CodeView is so big that it requires EMS to run, and it works fine with 386MAX.

While my comments about EMS apply generally to Windows debugging, there is a booby trap specific to working with a segment table. (Naturally we're telling you this because it happened to us.) Recall that, during Windows' LRU sweep, *cwClear* (*SegmentTable[1]*) is used as a count of words in the segment table to zero fill. Should this word get accidently set through a programming error, unbelievably strange results can occur. A random value stored in *cwClear* will zero out a random amount of *DGROUP*, possibly including your stack. What makes this bug so nasty is that the LRU sweep is driven by the timer tick interrupt, so the data gets wiped out without you ever seeing how. Even a 386 hardware breakpoint will not necessarily catch it. (In our experience, the hardware breakpoint caught this bug when debugging with a serial terminal, but not when using a monochrome monitor.)

Extensions

As written, the *segtable* library and associated macros assume that the segments in the table are always present in memory. This is guaranteed by the fact that none of the segments in the table are marked as discardable. Except for *DGROUP*, they are all allocated by *SegmentAlloc( )*, which does not set the *GMEM_DISCARDABLE* flag.

If the use of the segment table was expanded to include read-only segments as discussed above, then there would be discardable segments in the table. An even value in a table entry would signify that that segment had been discarded. More complicated access macros would be needed to account for this possibility and to provide the mechanism to reload the segment. The macros could take one of two approaches. The first method would be to always call a near function for each segment reference, and that function would test for an even entry and perform the reload if needed. The alternative is to make the test for an even entry in line, and call a function only when reloading is necessary. In fact, having both of these forms available might be handy so that the speed/size tradeoff can be made on a case-by-case basis. It is likely that read-only segments would be used only in special ways, so that many segment table references could still assume the segment was always present and use the original, more efficient macros.

We have been describing the whole idea of the segment table as being suitable for large applications with multiple segments of data. There is, however, a limit on how much data a Windows program can have. Being nondiscardable, the data must be present in memory at all times. This usually limits an application to not more than 300K under the best conditions. Large frame EMS does not increase this limit, but it does allow each of several applications running simultaneously to have about as much data space as if they were running alone.

The problem is the 640K limit on conventional memory, and one possible answer is EMS. Windows will allow individual applications to control the small (LIM 3.2-style) EMS frame, which provides four 16K portholes into the EMS space. It is completely up to the application to manage its expanded memory, using interrupt *67H* to access EMS functions.

One way to go about this is to integrate EMS management with the mem-

ory management functions of the *segtable* library. Any data segment of less than 16K is a candidate for allocation in EMS instead of using *GlobalAlloc( ). SegmentAlloc( )* could be modified to do this, putting the EMS segment into the segment table and returning a *pSeg*. In this way, the use of EMS becomes completely transparent to the rest of the application.

There is, however, a serious drawback. Because there is space for only four EMS pages in the page frame, we can't allocate more than four pages before we run out of places to put them. Of course, the whole point of EMS is that we can have many megabytes of data, but we only need to use a few pages at any one time. Some of the EMS pages we allocate for data will have to be mapped out of the page frame — becoming momentarily inaccessible — so that others can be mapped in when we need them.

Fortunately, the segment table mechanism provides a handy way to do this. *pSeg*s are the handle by which the application can refer to any chunk of memory, whether conventional, accessible EMS, or inaccessible EMS. If the *pSeg* points to an odd-numbered value in the segment table, then that segment is present; if it points to an even-numbered value, then it is not present. This is exactly the same rule that is used for read-only data segments.

To take this approach, the application's EMS manager must ensure that EMS segments are odd. Whenever it must change the EMS map, it will have to update the segment table. When a page is mapped out, its segment number in the table must be found and replaced with an even-numbered marker. This marker must represent sufficient information to make the page accessible again. For example, 1 byte of the marker could represent an index into a table that includes the EMS handle, while the other byte is the logical page number. Remember that only 15 bits are available, because the least significant bit must be zero.

The access macros must understand how to deal with segments that aren't present, using the same general techniques as they would for read-only segments. However, the segment is "reloaded" by calling the EMS manager, instead of by calling a Windows reload thunk. The application's memory manager will need to have some reasonable way to decide which logical page to map out when a different one must be mapped in. One approach would be to approximate the LRU algorithm by discarding the least-recently mapped-in page. Then when two different segments, say A and B, are needed at the same time, this can be ensured by the sequence access-A, access-B, access-A. The second access-A is required because the access-B might have caused A to get mapped out. This could happen only if A was already present at the start,

Microsoft's own Windows applications use all of the techniques discussed here

so that the first access-A did nothing.

To support cases when more than two segments were needed at once, a locking mechanism could be used. This would be similar to Windows' *GlobalLock( )* and *GlobalUnlock( )*, except that it would be handled by the application's memory manager. A streamlined alternative to making function calls for locking would be to set aside one or more special locations in the segment table. The presence of the segment in a special location would tell the memory manager not to map it out.

If the computer has no (or not enough) EMS, we can still do something to handle large amounts of data. By using the segment table and some additional help from Windows, we can set up a virtual memory system — that is, disk swapping. The key is to allocate memory with the Windows function *GlobalAlloc( )* by using the flags *GMEM_DISCARDABLE* and *GMEM_NOTIFY*. This tells Windows that it can discard the memory if it needs to, but to ask permission first. When Windows notifies the application that it would like to discard a segment, we can write that segment to disk first, then stick a marker for that segment in the segment table. As with EMS, the marker will represent the information needed to reload the segment the next time it is accessed.

The function that Windows will call to ask permission to discard a segment is set by using *GlobalNotify( )*. This function is documented in the Windows 2.0 SDK update booklet, with additional information in the Windows 2.1 SDK update. The function we register with Windows in this manner could be declared as:

```
BOOL FAR PASCAL
        NotifyProc(HANDLE hmem);
```

(continued from page 63)
The argument is supposed to be the handle of the segment being discarded. However, the Windows 2.1 SDK update says that in Version 2.03, it was actually the segment number, not the handle. This can be straightened out for both versions by calling *GlobalHandle( )*, which can take either the handle or segment number as its argument, and will return them both, as mentioned earlier.

NotifyProc( ) is a function in the application, but it must be in a fixed code segment. It will be called for each segment Windows would like to discard. If the application wants the segment locked, the function can return a false (zero) value and Windows will not discard it. The locking protocols could be the same as we suggested for EMS: Adding lock and unlock functions, and/or reserving special locations in the segment table. If the segment isn't locked, *NotifyProc( )* can write it to a disk file that has already been created for that purpose. Then it returns true and Windows will reclaim the space.

Any of these extensions — read-only data, EMS, disk swapping — may be combined. Using any one of them requires handling the case of segments that are not currently accessible. Once this jump has been made, the others can be added with little or no additional change to the main body of the application. Microsoft's own Windows applications use all of the techniques discussed here (a great deal of time was spent using Symdeb on Excel in preparing this article). While we haven't covered all of the procedures in detail, these ideas can be used to build Windows applications with virtually unlimited data capacity.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 106.)

Object-Oriented Programming in Assembly Language

OOP applies equally well to assembly language and high-level language programs

Randall L. Hyde

One of the promises of the object-oriented paradigm is that it will reduce program complexity and implementation effort for many different types of programs. Object-oriented programming, however, is no panacea. It is a technique, like recursion, that you can apply in certain cases to reduce programming effort. While there are certain types of programs whose object-oriented implementation is better, examples abound where object-oriented programming systems (OOPS) buy you nothing. Nonetheless, object-oriented programming techniques are a valuable tool to have in one's war chest.

OOPS are nothing new. They have been around since the late 1960s. Yet the object-oriented paradigm was languishing until Object Pascal and C++ began generating mainstream interest. The success of these languages demonstrates that OOP is not the domain of a few esoteric programming languages. Rather, object-oriented programming is applicable to almost any programming language.

Still, assembly language may not seem like the place to apply the object-oriented programming paradigm. But keep in mind that people were saying the same thing about Pascal and C five years ago.

What does an object-oriented assembly language program look like? A better question to ask is, "What is the essence of an object-oriented program, and how does one capture it within an assembly language program?" Once you strip away the gloss and notation convenience provided by languages such as C++, you'll find that the two main features of an object-oriented program are polymorphism and inheritance.

Polymorphism takes two basic forms in most programming languages: static and dynamic. The general idea, however, is the same. You call different subroutines by the same name. Static polymorphism provides notational convenience in the form of operator/function overloading in languages such as C++. Static polymorphism uses the parameter list, along with the routine's name (together they form the routine's signature), to determine which routine to call. For example, consider the C routines:

```
CmplxAddCC(C1, C2, C3);
                        /*C1=C2+C3;*/
CmplxAddCR(C1, C2, R1);
              /*C1=C2+ToCmplx(R1);*/
CmplxAddRC(C1, R1, C2);
              /*C1=ToCmplx(R1)+C2;*/
```

In C++ you could write:

```
CmplxAdd(C1, C2, C3);
CmplxAdd(C1, C2, R1);
CmplxAdd(C1, R1, C2);
```

and the C++ compiler would figure out whether to call *CmplxAddCC*, *CmplxAddCR*, or *CmplxAddRC*. (Actually, you could overload C++'s "+" operator and use the three forms *C1=C2+C3;*, *C1=C2+R1;*, or *C1=R1+C2;*, but the example above would be still valid.)

Static overloading, while convenient, does not add any power to the language. The calls to *CmplxAdd* call three different routines. *CmplxAdd-(C1,C2,C3)* calls *CmplxAddCC*, *CmplxAdd (C1,C2,R)* calls *CmplxAddCR*, and *CmplxAdd-(C1,R,C2)* calls *CmplxAddRC*. The C++ compiler determines which routine this code will call at compile time. Static polymorphism is a mechanism that lets the compiler choose one of several different routines to call depending upon the calling signature.

Sometimes you may want to use the same signature to call different routines. For example, suppose you have a class *shape* in which there are three graphical objects: circles, rectangles, and triangles. If you have an arbitrary object of type *shape*, the compiler cannot determine which *DRAW* routine to call. The program determines this at run time. This allows a single call to draw circles, rectangles, triangles at run time with the same machine instructions. This is dynamic polymorphism — determining at run time which routine to call. C++ uses virtual functions and Ob-

Randy is the designer of numerous hardware and software projects, including assemblers for a variety of systems. In addition to consulting, he is currently a part-time instructor in computer science at California Polytechnic University in Pomona and at UC Riverside. He can be contacted at 9570 Calle La Cuesta, Riverside, CA 92503.

(continued from page 66)
ject Pascal uses override procedures and functions to implement dynamic polymorphism.

Inheritance lets you build up data structures as supersets of existing data structures. This provides a mechanism whereby you can generalize data types, allowing you to handle various objects regardless of their actual type. This lets you define such diverse shapes as circles, rectangles, and triangles and treat them as compatible structures.

Implementing Classes and Inheritance

Because structures and classes are closely related, it may be instructive to look at the implementation of structures before looking at classes. Consider *S*, a variable of the type in Example 1. Somewhere in memory the compiler needs to generate storage for the fields of *S*. Traditionally, compilers allocate these fields contiguously (see Figure 1). Indeed, Microsoft's assembler (MASM) allows you to declare structures in a similar fashion, as shown in Example 2. If *S* provides the base address of this structure, *S+0* is the address of *S.i*, *S+2* is the address of *S.j*, and *S+4* is the address of *S.c*.

Now consider the case of a pair of C++ classes (*Sc* and *Tc*) in Example 3. Pointers to objects (*pS* and *pT*) may point at an object of the prescribed type or to an object that is a descendant of the pointer's base class. For example, *pS* can point at an object of type *Sc* or at an object of type *Tc*. Remember, accessing **pS.j* is equivalent to *(int) *(pS+2)*, so if *pS* points at an object of type *Tc*, the *j* field must also appear at offset two within the structure. For inheritance to work properly, the common fields must appear at the same offset within the structure (see Figure 2).

Additional fields in the subclass often appear after the fields in the parent class, so most compilers implement class *Tc* as in Example 4. Note that the offsets to *i*, *j*, and *c* are the same for both *Sc* and *Tc*.

When I first began exploring how to implement inheritance in assembly, I got the bright idea of using macros inside structure definitions to handle the problem of inheritance. Briefly, I wanted to implement *Sc* and *Tc* as in Example 5. Unfortunately, MASM doesn't allow you to expand macros or strucs inside a structure. Disappointed, I tried the brute force way to implement *Sc* and *Tc*, as illustrated in Example 6.

Unfortunately, I'd forgotten that MASM doesn't treat these symbols as part of the structure. Names such as *i*, *j*, *c*, and so on must be unique in the program. As you can plainly see in Example 6, I declared *i* twice, and the assembler gave me a "redefinition of symbol" error. Almost ready to give up, I tried the method in Example 7.

MASM simply equates the field names to the offsets within the structure. So it equates *i* to zero, *j* to two, and so on. MASM does not associate *i* with labels of type *Sc*. You can use the symbols *T.j* and *S.j* in your program. Because the "." operator behaves like the "+" operator, *T.j* is just like *T+2*.

For *Tc* to inherit the fields of *Sc*, all we have to do is reserve enough space at the beginning of the *Tc* structure for

Example 1: *The variable* S

```
struct
{
        int i;
        int j;
        char *c;
} S;
```

Example 2: *Declaring structures in MASM*

```
SType   struc
i       dw      ?
j       dw      ?
c       dd      ?
SType   ends
```

Example 3: *C++ classes*

```
class Sc
{
          int i;
          int j;
          char *c;
};

Sc *pS;
Tc *pT;

-------------------------------

Tclass Tc:Sc
{
          int k;
          char *d;
};
```

Example 4: *The way most compilers implement a class like* Tc

```
struct Tc
{
      int i;
      int j;
      char *c;
      int k;
      char *d
};
```

Example 5: *One approach to implementing* Sc *and* Tc

```
ScItems macro
i       dw      ?
j       dw      ?
c       dd      ?
endm

;
TcItems   macro
          ScItems
k         dw      ?
d         dd      ?
          endm

;
Sc        struc
          ScItems
Sc        ends
;
Tc        struc
          TcItems
Tc        ends
```

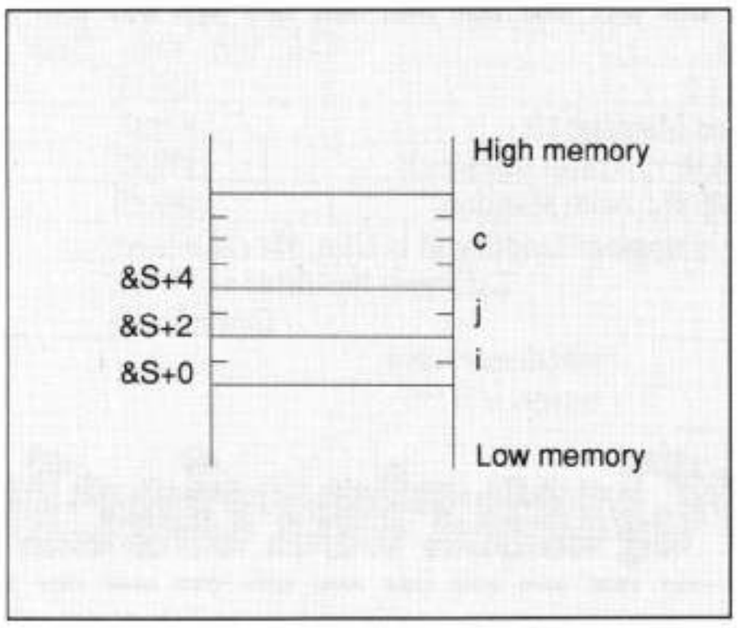


Figure 1: *Storage allocation for* S

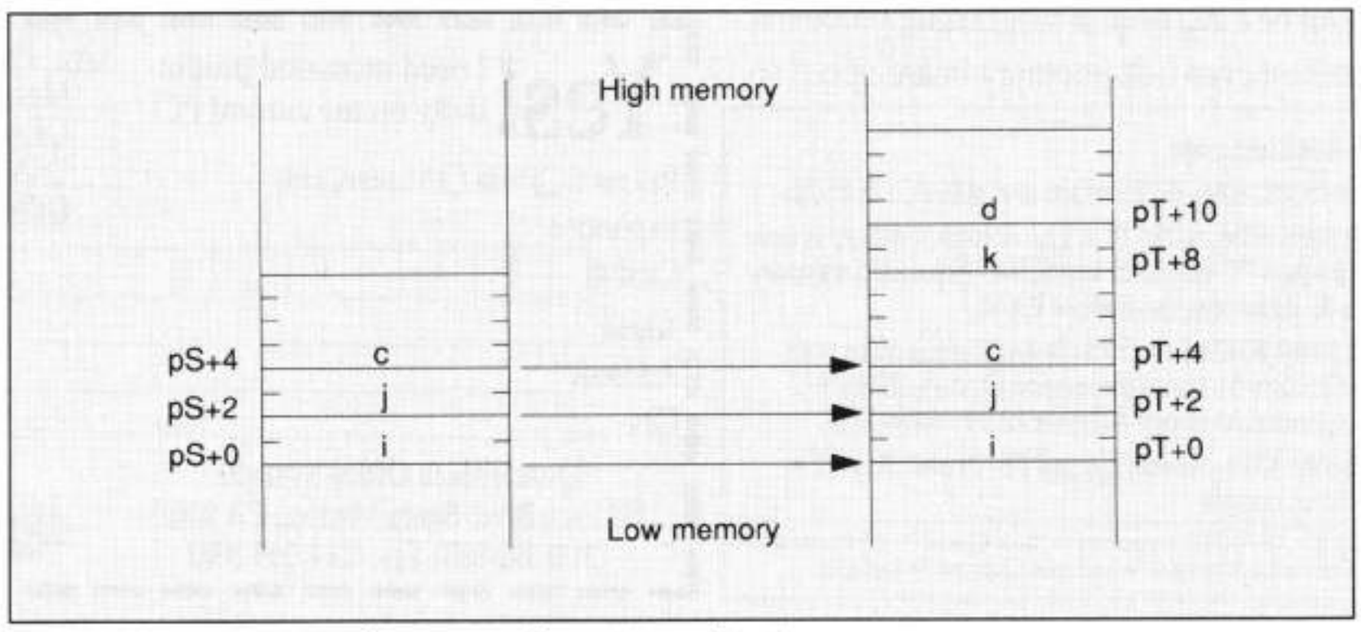


Figure 2: *Storage allocation for* *pT *and* *pS

(continued from page 68)
each of the fields of *Sc*. Above, I stuck in the two *DW* and the *DD* pseudo-opcodes to reserve space for the *i*, *j*, and *c* fields. This technique might get inconvenient if the number of inherited fields is large. The code in Example 8 solves this problem.

The first *DB* pseudoopcode in *Tc* reserves the necessary space for the fields *Tc* inherits from *Sc*. Likewise, *Uc* (which is a subclass of *Tc*) reserves space at the beginning of the structure for the fields inherited from *Tc* and *Sc*. The code in Example 8 works great if you don't need to initialize any of the fields inherited from *Sc*; if you need to initialize some fields, you'll have to use the brute force method and redeclare space for each field.

Methods

The earlier paragraphs discuss how to implement objects whose fields are all variables. What happens when you introduce methods? If you're not overloading a method, you can treat it in the same manner as any other assembly language procedure and call it directly. If you are overloading a method, you must call it indirectly via a pointer within the object.

Consider the C++ class declaration in Example 9. The assembly code implementing this class is shown in Example 10. To call *S.geti*, you would use the 8086 instruction: *CALL S.geti*.

Because *S.geti* is a double word memory variable, the *CALL* instruction will call the procedure *S.geti*, which points at *Sc_geti*. The fact that we're calling the methods indirectly will be useful when we look at overloading a little later.

THIS

Suppose we have three instances of class *Sc*, say *S1*, *S2*, and *S3* declared in assembly language as follows:

```
S1  Sc
S2  Sc
S3  Sc
```

S1.geti, *S2.geti*, and *S3.geti* all call the same procedure (call it *Sc_geti*). How does *Sc_geti* differentiate between *S1.i*, *S2.i*, and *S3.i*? In object-oriented languages such as Object Pascal and C++, the compiler automatically passes a special parameter named *this* to the method. *this* always points at the object through which you've invoked the method. When you execute *S1.geti*, the compiler passes the address of *S1* in *this* to *geti*. Likewise, the compiler passes the address of *S2* in *this* when you call *S2.geti*.

You can pass *this* to a method just as any other parameter. Because the most efficient way of passing parameters is in the 8086's registers, I've adopted the convention of passing *this* in the ES:BX registers. The *Sc_geti* method would look something like Example 11 (assuming we're returning *i* in the AX register). This example demonstrates a major problem with object-oriented programming — it is very inefficient. To load *S1.i* into AX, see Example 12. This requires six instructions where, logically, you should only need one (*mov ax, S1.i*). Welcome to the wonderful world of object-oriented programming! Yet circumventing all this overhead by loading *S1.i* directly into AX will eliminate the benefits of object-oriented programming.

Actually, this isn't as bad as it looks. A good part of the time ES:BX will already be pointing at the object you want to access. Nevertheless, the call and return are considerable overhead just to load the AX register with a word value. Stroustrup anticipated this problem when designing C++ and he solved it by providing inline functions (a.k.a.

Example 6: *The brute force method of implementing* Sc *and* Tc

```
Sc      struc
i       dw      ?
j       dw      ?
c       dd      ?
Sc      ends

Tc      struc
i       dw      ?
j       dw      ?
c       dd      ?
k       dw      ?
d       dd      ?
Tc      ends
```

Example 7: *Yet another attempt at implementing* Sc *and* Tc

```
Sc      struc
i       dw      ?
j       dw      ?
c       dd      ?
Sc      ends

Tc      struc
        dw      ?
        dw      ?
        dd      ?
k       dw      ?
d       dd      ?
Tc      ends

S       Sc
T       Tc
```

Example 8: *The solution to implementing* Sc *and* Tc

```
Sc      struc
i       dw      ?
j       dw      ?
c       dd      ?
Sc      ends

Tc      struc
        db      (size Sc) dup (?)
k       dw      ?
d       dd      ?
Tc      ends

Uc      struc
        db      (size Tc) dup (?)
e       dw      ?
Uc      ends

S       Sc
T       Tc
U       Uc
```

Example 9: *A C++ class declaration*

```
class Sc
{
      int i,j;
      char *c;

      public:

      int geti() {return i};    /* Ignore the fact that C++ */
      int getj() {return j};    /* would implement these    */
      void seti(x) int x; {i = x;};  /* methods in-line.    */
      void setj(x) int x; {j = x;};
};
```

Example 10: *Assembly code for implementing the code in Example 9*

```
Sc      struc
i       dw      ?
j       dw      ?
c       dd      ?
geti    dd      Sc_geti
getj    dd      Sc_getj
seti    dd      Sc_seti
setj    dd      Sc_setj
Sc      ends

S       Sc
```

(continued from page 70)
macros). We can use this same technique in assembly language to improve efficiency as Example 13 illustrates. This code snippet demonstrates another convention I adhere to: I make macros for all method calls, even those that are actual calls. This lets me use a consistent calling format for all methods, whether they are actual subroutines or are expanded in-line.

There is one major drawback to expanding a procedure inline; you cannot overload procedures (C++'s inline functions suffer from this as well. You cannot have an inline virtual function). Therefore, you should only use this technique for those particular methods that you will never need to overload. Fortunately, the macro implementation makes it easy to switch to a call later if you need to overload the procedure. Just substitute a call for the inline code inside the macro.

Polymorphism and Overloading

Overloaded procedures allow the "same" method to perform different operations, depending upon the object passed to the method. Consider the class definitions in Example 14. *Rect* and *Circle* are types derived from *Shape*. If ES:BX points at a generic shape (that is, ES:BX points at an object of type *Shape*, *Rect*, or *Circle*) then *CALL_THIS.Draw* will call *Shape_Draw*, *Rect_Draw*, or *Circle_Draw*, depending upon where ES:BX points. This lets you write generic code that needn't know the particular details of the shape it's drawing. The object itself knows how to draw itself via the pointer to the specific draw routine.

Allocation of Objects

High-level object-oriented languages such as Object Pascal and C++ tend to hide many of the allocation details from you. In assembly language, naturally, the programmer has to handle all of the allocation details. Although a complete discussion of dynamic allocation of objects is beyond the scope of this article, the subject is so pervasive that it warrants a brief mention.

Static allocation of an object in assembly language is quite simple. If you have the *shape* class definitions (*shape*, *rect*, and *circle*) mentioned earlier, you can easily declare variables of these types using declarations of the form:

```
MyRect      rect
MyCircle    circle
MyShape     shape
```

This automatically fills in the *DRAW* field for these variables (the linker/loader fills in such addresses when it loads the program into memory). What happens if you are dynamically allocating storage for an object? Assume we have a routine, *alloc*, to which we pass a byte count in CX, and it returns a pointer to a block of memory that size in ES:BX. Now suppose we allocate a rectangle with the code in Example 15. *Alloc* will not be smart enough to fill in the pointer to the *rect.DRAW* routine. This is something we'll have to do ourselves. This requires the four instructions in Example 16.

Eight instructions may not seem like a lot to create a simple object. Keep in mind, however, that our simple *shape* object only has one overridden method. If there were a dozen methods, you would need 52 instructions. Clearly, a *CREATE* procedure begins to make a lot of sense. Each subclass (*shape*, *rect*, and *circle*) will need its own *CREATE* method. *CREATE* is not a method you normally overload, because during the creation process you know exactly the type of object you're creating. By convention, the *CREATE* methods I write always allocate the appropriate amount of storage, initialize any important fields, and then return a pointer to the new object in ES:BX. The code in Example 17 provides an example, using the *rect*

Example 11: *The* Sc_geti *method*

```
_THIS       equ     es:[bx]
Sc_geti     proc    far
            mov     ax, _THIS.i
            ret
Sc_geti     endp
```

Example 12: *Loading* S1.i *into AX*

```
mov     bx, seg S1
mov     es, bx
mov     bx, offset S1
call    S1.geti         ;Assuming S1 is in the data seg
```

Example 13: *Improving efficiency*

```
;
; Inline expansion of geti to improve efficiency:
;
_Geti               macro
                    mov             ax, _THIS.i
                    endm
;
; Perform actual call to routines which are too big to
; expand in-line in our code:
;
_Printi             macro
                    call            _THIS.Printi
                    endm
                      .
                      .
                      .
                    _Geti           ;Get i into AX.
                      .
                      .
                      .
                    _Printi         ;Call Printi routine.
```

Example 14: *Typical class definitions*

```
Shape       struc
ulx         dw      ?       ;Upper left X coordinate
uly         dw      ?       ;Upper left Y coordinate
lrx         dw      ?       ;Lower right X coordinate
lry         dw      ?       ;Lower right Y coordinate
Draw        dd      Shape_Draw  ;Default (overridden) DRAW routine
Shape       ends
;
Rect        struc
            dw      4 dup (?)   ;Reserve space for coordinates
            dd      Rect_Draw   ;Draw a rectangle
Rect        ends
;
Circle      struc
            dw      4 dup (?)   ;Reserve space for coordinates
            dd      Circle_Draw ;Draw a circle
Circle      ends
```

and *circle* types. To manipulate these objects, we need only load the appropriate pointer into ES:BX and access the appropriate fields or call the appropriate methods via *this*.

Other Conventions

While writing object-oriented programs in assembly language, I've found certain guidelines helpful in the initial design phases (that is, before having to take efficiency into consideration). Most of these guidelines are widely accepted object-oriented practices; others pertain mainly to assembly language. Here are the major ones I'm using:

- Try to use dynamic allocation for objects wherever possible. In the best object-oriented programs, instances of an object appear and disappear throughout the program. Rarely will a single instance exist throughout the execution of a program. Because an object's methods always reference fields of an object indirectly, there is little benefit to statically allocated objects. Converting a statically allocated object to a dynamically allocated one later on is messy. Get it right the first time!
- Avoid accessing the individual variables (fields) within an object. Write methods that store values into these fields and retrieve values from them. This information-hiding technique is well proven in OOP and isn't particularly worthy of further discussion.
- Overload as many methods as possible. *CREATE* is probably the only method you shouldn't overload. Access methods, which provide access to the fields of the outermost class, might be another candidate for direct access. But the loss of generality for a small increase in efficiency is rarely worth it.
- Always use macros to call methods, especially those you're not calling indirectly. This provides a consistent calling mechanism for methods and lets you easily overload methods you choose to implement inline or without overloading. This applies equally well to accessing fields in an object.
- As a bare minimum, each class should have the following methods: *CREATE*, *DISPOSE*, *COPY*, and a set of access methods for each of the fields. *COPY* should copy the contents of one instance variable's fields to another variable.

Naturally, these are just guidelines, not rules etched in stone. But a certain amount of discipline early in a project helps prevent considerable kludging later on.

An Example

The example in Listing One (page 110) is a program that adds, subtracts, and compares signed binary integers, unsigned binary integers, and BCD values. While not a complete example (it's missing several important methods such as *CREATE*, *PRINT*, *DISPOSE*, and so on) it demonstrates the flavor of object-oriented programming in assembly language.

What About Your Programs?

Object-oriented programming is a concept that can reduce the time you spend developing certain classes of programs. The OOP concept applies equally well to assembly language and high-level language programs. The only drawback is that you don't have a large library of classes to build upon. Of course, these same problems exist for Object Pascal and C++ users. Time will solve this problem for those languages as users begin developing reusable modules for both, which is all that is preventing object-oriented assembly language from taking off. Perhaps someday you will be able to buy off-the-shelf object-oriented assembly language libraries; until then, you'll have to write your own. Even so, the tricks and techniques of object-oriented programming are well worth considering for your next assembly language project.

DDJ

(Listing begins on page 110.)

Example 15: *Code to allocate a rectangle*

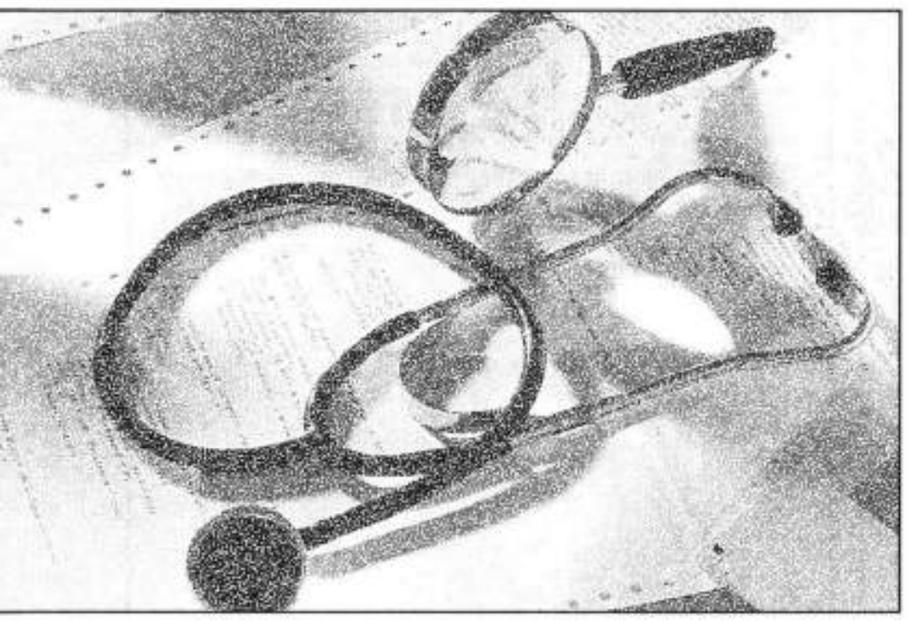
```
mov     cx, size rect
call    alloc
mov     word ptr MyRectPtr, bx
mov     word ptr MyRectPtr+2, es
```

Example 16: *Filling in the pointer to the* rect.DRAW *routine*

```
mov     ax, offset rectDRAW
mov     _this.DRAW, ax
mov     ax, seg rectDRAW
mov     _this.DRAW+2, ax
```

Example 17: *Code for example using the* rect *and* circle *types*

```
    mov     cx, size circle
    call    CreateCircle
    mov     word ptr CircVarPtr, bx
    mov     word ptr CircVarPtr+2, es
;
    mov     cx, size rect
    call    CreateRect
    mov     word ptr RectVarPtr, bx
    mov     word ptr RectVarPtr+2, es
```



Inside Watcom C 7.0/386

32-bit code can speed up your programs on an already quick machine

Andrew Schulman

Over two years ago, the cover of the July 1987 issue of *Dr. Dobb's* carried the title "386 Development Tools Within Your Lifetime" a photograph of a skeleton that rotted away in front of its computer while waiting for decent 386 tools, which summed up everyone's feelings about programming for the Intel 80386 microprocessor.

Things have improved a great deal since that issue. Watcom C7.0/386, for instance, produces 32-bit code (such as *MOV EAX, 12345678h*, and *MOV FS:[EAX], ESI*) while staying keyword and library compatible with the de facto 16-bit industry standard, Microsoft C 5.1 (MSC51). Even weird low-level routines such as *intdosx( ), _dos_setvect( ), _dos_keep( )*, and *_chain_intr( )* do the right thing in 32-bit protected mode.

Of course, Watcom C7.0/386 (WAT386) has many of the same features as Watcom's 16-bit C compiler (see "Examining Room," *DDJ* September 1989). This includes Watcom's famous register-based parameter passing. Many of Watcom's innovations involve the reduction, and sometimes elimination, of function call overhead. Any block of code that takes input from registers and puts output into registers is effectively a functional object, and WAT386 takes advantage of this fact in several places, including the nifty *#pragma aux* feature.

Andrew is a software engineer in Cambridge, Mass., working on CD-ROM network applications, and is also a contributing editor for DDJ. *He can be reached at 32 Andrew St., Cambridge, MA 02139.*

Buying In

WAT386 produces very different code from either Microsoft C or Turbo C (neither of which has an option to generate 386 instructions, much less 32-bit code). Yet, this compiler will fit seamlessly into your current work habits. Unlike MetaWare's High C 386 compiler, using WAT386 does not produce "culture shock."

Still, all is not rosy. It will cost you over $1000 in software to get into 386 development. WAT386, like High C, costs $895, and you will also need a 32-bit DOS extender, like the industry-standard Phar Lap 386 toolkit, which costs $495.

Further, the new Watcom C7.0/386 compiler is just that — new. While writing this review, I found a number of bugs in the compiler and its standard library. Watcom was undoubtedly under pressure from its major client, Novell, to get the 386 compiler out the door. By the time you read this review, though, a second, more stable, release of WAT386 should be available.

Primarily because of its newness, WAT386 in some ways is not as good a product as MetaWare's High C 386, which has been around for two and a half years. Still, there is value in WAT386. For many PC programmers, this will be a much easier product to use than MetaWare's High C. WAT386's Microsoft compatibility is very important. On the other hand, the next release (1.6) of High C 386, in addition to many other changes, is scheduled to have what a MetaWare press release calls "86% compatibility with Microsoft's C libraries."

32 Bits!

WAT386 generates code for 32-bit protected mode. Thus, *sizeof(int)* and *sizeof(unsigned)* are each 4 bytes, not 2 bytes. Likewise, *sizeof(void *)* is 4 bytes. Note that *sizeof(void near *)* is also 4 bytes.

The all-important ANSI C identifier *size_t*, which is the unsigned type of the result of the *sizeof( )* operator and the type used by function parameters that accept the size of an object, is also 4 bytes (*typedef unsigned size_t*).

C standard library functions such as *malloc( ), fwrite( )*, and *strncpy( )* all take *size_t* parameters, and *strlen( )* returns a *size_t*. These standard library functions deal in quantities between *0* and *UINT_MAX*. In the 16-bit code generated by PC compilers like MSC51, *UINT_MAX* is *0xFFFF* (65,535), yielding the familiar 64K limit on PC array lengths, string lengths, malloc blocks, and so on.

But in 32-bit code, *UINT_MAX* is *0xFFFFFFFF*, or 4,294,967,295 — the magical upper "limit" of 4 gigabytes! In the native mode of the 386, this is the upper bound set on array lengths, string lengths, and malloc blocks. Effectively, no limit at all.

The Environment

If *fwrite( )* can write 4 gigabytes at a time (which might be handy if you're working with CD-ROM or some other form of mass optical storage), how can it possibly work with MS-DOS? DOS is a 16-bit operating system. (So is OS/2.) The DOS *Write* function (*INT 21, function 40H*), which *fwrite( )* must eventually call, expects the number of bytes

(continued from page 74)

to write in the 16-bit CX register. The maximum is 64K. How can WAT386, or any 32-bit C compiler for DOS, produce code that's compatible with 16-bit DOS?

The answer is that 386 C compilers (for DOS) produce code to be run under a 32-bit DOS extender. Programs such as Phar Lap's 386|DOS-Extender and Eclipse Computer Solutions' OS/386 do not replace DOS. Instead, they (almost invisibly) manage the interface between 16-bit real-mode DOS and your 32-bit protected-mode program.

In the example of *furite( )*, the 32-bit code produced by WAT386 or High C (which MetaWare actually calls "High C for MS-DOS/386") continues to call *INT 21, function 40H*. But now, the number of bytes to write goes into the full 32-bit ECX register rather than the 16-bit CX register.

A DOS extender takes over *INT 21* (as well as other software interrupts like *INT 10, INT 16*, and so on), handles some functions itself, and passes others on to DOS. A program running under a 32-bit DOS extender is effectively running under "MS-DOS/386," because, for example, a call to write 640K is really going to write 640K. The DOS extender will invisibly break this up into multiple calls to the "real" *INT 21, function 40H*.

Another interesting example is *malloc( )*. If your 386 computer came with 4 gigabytes of memory, you could grab it all with a single call to *malloc( )*. As in 16-bit real-mode C compilers, the C memory manager eventually calls *INT 21, function 48* (allocate memory). Here, however, the DOS extender provides a complete replacement, not a front end, for the DOS routine. There is one difference between Phar Lap and Eclipse: 386|DOS-Extender expects in EBX the number of 4K pages to allocate, where OS/386 more closely mimics DOS, expecting the number of 16-byte paragraphs. The WAT386 standard library detects which DOS extender it is running under and allocates memory appropriately.

By default, WAT386 produces code to be run under Phar Lap Software's 386|DOS-Extender. The Phar Lap toolkit (DOS extender, linker, assembler, and debugger) must be purchased separately, however.

Oddly, you don't need a 386 machine or a DOS extender to run the WAT386 compiler. By the time you read this review, Watcom should be shipping a 32-bit protected-mode version of the compiler. In the version I reviewed, however, all compiler components were 16-bit real-mode programs. To avoid "Not enough memory to fully optimize procedure" warnings, I had to specify that the compiler use a large-model version of the code generator. Pretty crazy for a 386 development system!

Presumably, if your customers had 386s but you didn't (which is probably the exact opposite of the real situation), you could use these 16-bit tools to generate 386 code on your AT.

Programs compiled with WAT386 and linked with Phar Lap's 386|LINK will only run on 386-based machines. To sell such programs, and to acquire a program that will "bind" the DOS extender into the executable so that your customers don't need to know anything about the DOS extender, you must acquire a redistribution package from Phar Lap. This costs an extra $1000 for unlimited distribution.

So the entrance fee for 386 development is still pretty steep. What do you get in return? A lot: Code that runs several times faster than 16-bit code; the elimination of 64K limits on array sizes or function parameters; and the elimination of the 640K boundary, allowing you to use all physical memory in the machine.

Note that this "MS-DOS/386" gives you big memory, but not virtual memory (VM). This is an important difference from OS/2. However, a VM manager (386|VMM) is available for $295 from Phar Lap, and WAT386 code, like High C code, runs without change under 386|VMM.

WAT386 code runs under one other environment: Novell's new 32-bit network operating system, NetWare 386 (see the accompanying box).

The Code

How can a 32-bit C compiler such as WAT386 produce code that runs several times faster than 16-bit code run on the same machine? Consider the following two lines of code:

```
extern char*Env;
char *p=Env
```

Compiling under the "large model" (which is what most commercial PC software uses), any 16-bit C compiler, including Watcom's non-386 compiler, produce code something like that shown in the first portion of Example 1, in which the 4-byte *far* pointers are transferred piecemeal from one location to another.

Because *mov mem, reg* takes 2 clock cycles on a 386 and *mov reg, mem* takes 4 cycles regardless of whether the compiler uses the 8-bit (AL), 16-bit (AX), or 32-bit (EAX) form of the register, this takes (2*2) + (3*4) = 16 cycles. In con-

(continued on page 79)

(continued from page 76)
trast, the 32-bit equivalent takes 2 + 4 = 6 cycles (shown in the second portion of Example 1).

The 32-bit code is similar to the code that would be generated by a 16-bit compiler working with 2-byte *near* pointers:

```
mov ax, _Env
mov word ptr _p, ax
```

In fact, "flat model" 32-bit code and "tiny model" 16-bit code are very similar. The only difference is that the 16-bit code can handle quantities up to

Watcom and Novell

In addition to producing code for Phar Lap's 386|DOS-Extender and, with some difficulty, for Eclipse's OS/386, the 32-bit Watcom C compiler also works with Novell's new network operating system, NetWare 386. In fact, Watcom C7.0/386 is being repackaged by Novell as its C Network Compiler/386. (This is the subject of Novell's strange "See Dick and Jane" ads.)

NetWare 386 is a 32-bit operating system, and this allows for several performance leaps over the existing 286-based NetWare. Instead of the current limit of 100 users per file server, which is dictated by the single 64K data segment available in "medium model" (used in 286-based NetWare), the new NetWare 386 allows 250 simultaneous users per file server. Likewise, Novell claims that network throughput is two to three times greater than its already zippy throughput figures.

In NetWare 386, the lack of segmentation in "flat model" is taken to its logical (but scary) extreme — no memory protection. Novell baldly states that, "There is no memory or other application-level protection: All applications and device drivers run in kernel mode" (*NetWare Technical Journal*, July 1989).

When used with NetWare 386, the Watcom C compiler produces server applications — programs that run in file-server memory (the so-called "file server" thus becomes a generic server). These server applications are called "NetWare Loadable Modules," or NLMs, and are somewhat like value-added processes (VAPs) in pre-386 NetWare; except unlike VAPs, NLMs can be loaded or unloaded at any time, without taking down the file server. NLMs, in fact, are dynamic-link libraries and, in addition to providing services to clients on the network, can provide functions to be called by other NLMs.

For instance, when calling a C standard library such as *open( )* from an NLM, you are actually calling a routine in CLIB.NLM, which is the C standard library provided as a dynamic-link library. The code for *open( )* is not linked into your executable.

To produce such an NLM, use the NLMLINK provided by Novell rather than the Phar Lap linker. Similar to the OS/2 linker, NLMLINK requires a .DEF file with import statements. The module produced by the Novell linker essentially contains unresolved externals that are resolved when the NLM is loaded into file server memory (either by invoking the LOAD command at the file server console, or by spawning one NLM from within another).

The library included with C Network Compiler/386 includes many functions not available in the standard Watcom library. Naturally, functions are provided to support network communications with Novell's IPX and SPX. The Btrieve data management library is provided as BTRIEVE.NLM. The Novell library includes functions (for example, *TestAndSetBit( )* and *BitScan( )*) to interface to the 386-bit test instructions.

Network servers are inherently multitasking (multiple operations must be in progress simultaneously on behalf of multiple clients), so the library contains functions for "execution threads," such as *BeginThread( ), EnterCritSec( ), ExitCritSec( ), SuspendThread( )*, and so on. There are also functions to manage semaphores and queues.

While this part of the Novell API seems modeled on OS/2, it is important to note that NetWare 386 uses non-preemptive multitasking. Inside a "big job," it is therefore necessary to call a routine such as *delay( )* or *ThreadSwitch( )* so that other threads are not starved.

The library that Watcom provided for Novell contains a few modifications to support multiple threads. Global variables such as *errno* are in fact allocated on a prethread basis. Static data such as used by the notorious *strtok( )* function is also handled differently than in a single-threaded library. No new keywords (such as *private*, used in Lattice C 6.0 for OS/2) have been added, however. — A.S.

64K, whereas the 32-bit code can handle quantities up to 4 gigabytes.

Right now, WAT386 supports flat model and small model. In the flat memory model, the application's code and data must total less than 4 gigabytes in size. In the small memory model, your code and data are each "limited" to 4 gigabytes. By default, WAT386 uses the flat model. When linking with the Lahey linker (LINK-EM/32) provided with OS/386, you must compile with the small model.

Because an offset into a segment is 4 bytes while the segment registers are still 2 bytes, *sizeof(void far *)* is 6 bytes (an FWORD, not a DWORD). But because a *near* pointer is a 4-byte quantity, you almost never have to deal with *far* pointers. When a segment takes a 4-byte offset, even the most sloppily written, bloated program in the world should do fine with the flat model. Once loaded, DS and CS stay constant. Effectively, this is a linear address space.

Real-World Benchmarks

Interpreters are better for benchmarking compilers than the tiny programs that are usually used. Such benchmarks usually involve a fair amount of source code. The C source code for several interpreters is readily available, and to execute one line in the interpreted language, the interpreter needs to crunch through a lot of C code.

In the remainder of this review, I'll describe using WAT386 (and MetaWare High C) to port a larger program to the 386: ISETL (Interactive Set Language), written in C by Gary Levin (Dept. of Mathematics and Computer Science, Clarkson University, Potsdam, N.Y.). ISETL is an interpreter for working with sets, tuples, propositions, several different types of functional objects, matrices, and other constructs useful for studying the mathematical foundations of computer science. It is described in the book *Learning Discrete Mathematics with ISETL* by Nancy Baxter, Ed Dubinsky, and Gary Levin (New York: Springer-Verlag, 1989). ISETL deserves a full discussion, but for now I'll just describe the process of producing ISETL/386.

Due to space considerations, the ISETL/386 listings are not included in this issue. They are available through *DDJ* (see the end of this article for information). The ISETL implementation consists of 29 .C files and 14 .H files, and totals about 13,000 lines of code. Some of the code is YACC output.

To sell such programs, and to acquire a program that will "bind" the DOS extender into the executable so that your customers don't need to know anything about the DOS extender, you must acquire a redistribution package from Phar Lap

When I tried to produce a 386 version of this real program, my opinion about WAT386 vs. High C nearly reversed. As long as I was working on small one- or two-module programs, WAT386's similarity to Microsoft C and Turbo C made it preferable to MetaWare High C. But once I started working on ISETL/386, with more source code, written by someone else, my allegiance shifted to High C.

High C provides better warning messages than WAT386; the High C compiler is faster than WAT386 (remember, the WAT386 compiler I used was a 16-bit real-mode program); surprisingly, High C seems to produce better overall code than WAT386; and, most important, High C and its standard library isn't buggy like WAT386.

I should mention that Watcom has terrific technical support. If you call up with a problem, you get to talk to the person responsible for the library or the compiler. Watcom is quick to find and fix bugs and, with the WPATCH utility that comes with WAT386, they have made the patch a fine art. Watcom runs a well-organized BBS. On the other hand, I don't even know how good MetaWare's technical support is, because I never needed to use it.

At one time or another, we've all thought we've found a compiler bug only to discover that in fact we have a bug in our own code. But after working with WAT386 for about a month, I found that nearly every time it was a compiler or library bug.

First of all, one of the key switch statements in ISETL was behaving bizarrely. The value of the variable being switched on was correct, we would jump to the correct case label, but a function call to *Emit(42)* wasn't working. The problem is that any constant (for example, *42*), used (anywhere) inside a switch statement is scrambled if that constant happens to match the number of case labels in the switch statement! This bug should be fixed by the time you read this. If you have this same release of the compiler, you can download a patch from the Watcom BBS.

Another problem occurs because the ISETL initialization file opens the DOS device CON (to implement a *pause( )* routine for use in ISETL programs) and tries to read from this device. The problem is, when reading from any of the DOS device files (CON, AUX, and so on), the WAT386 library gets confused between binary and text mode; a call to wait for one character actually waits for 512 characters, that makes it seem like the machine is hung.

In another project, I found that *intdosx(. . .)* was not working, even though *int386x(0x21, . . .)* worked fine. If this has not been corrected by the time you read this, a patch is available from the Watcom BBS.

In that same project, I found an obscure bug in Watcom's use of the "interrupt" keyword that had to do with calling an interrupt function rather than generating an interrupt. Basically, functions defined with *void interrupt (far *f)( )* work. But functions defined with *void (interrupt far *f)( )* (note the placement of parentheses) don't do a *PUSHFD* when you call them.

There is one problem that's not Watcom's fault: Debugging with the 386 flat memory model is hardly better than debugging in real mode. With one single segment working as a linear address space, it is nowhere as easy to catch bugs as when you have lots of little segments (for example, a 286-based protected-mode DOS extender such as DOS/16M). In fact, to debug ISETL/386, I found it necessary to cre-

```
16-bit code:
     mov es, seg _Env
     mov ax, word ptr es:_Env
     mov dx, word ptr es:_Env+2
     mov word ptr _p, ax
     mov word ptr _p+2, dx

-------------------------------

32-bit code:

     mov eax, _Env
     mov _p, eax
```

Example 1: *32- and 16-bit code generated under the large memory model*

(continued from page 80)
ate a DOS/16M version (ISETL/286). This shows that segmentation is not such a bad idea, after all, it's crucial for genuine memory protection. The ideal situation is to use lots of segments for development, and then switch over to the flat model for production.

The only assistance you get in catching memory protection violations from the WAT386 flat memory is the Phar Lap linker's OFFSET switch, which allows you to load code or data starting at some offset other than zero. This way, you get page faults when dereferencing bad pointers, though you often won't know where they come from.

Benchmarking with ISETL/386

Once ISETL/386 was up and running with WAT386, I was able to write some ISETL programs and use them for benchmarking the compilers. In addition to contrasting WAT386 and High C, I was able once again to compare 32-bit code with 16-bit code, using the Turbo C-produced executable from the ISETL distribution.

Figure 1 shows the results for two different ISETL programs to generate prime numbers, for an ISETL program to generate the first 1000 Fibonacci numbers, and for an overall test of ISETL operations.

	WAT386	HIGH C 386	TURBO C
PRIME.SET 2000	18.0	16.3	24.8
PRIME.SET 4000	42.6	40.0	N/A
PRIME.TUP 2000	1:03.7	52.7	1:11.3
PRIME.TUP 4000	4:14.9	3:27.6	N/A
FIB.SET 1000	15.0	14.1	20.4
FIB.SET 1200	18.0	17.0	N/A
overall test	1:05.3	59.4	1:30
total	477.5	407.1	N/A
ISETL filesize	133K	148K	209K
ISETL full compile	12:52 min.	11:45 min.	3:30 min.

Figure 1: *ISETL test execution times in seconds (Watcom and High C run times using Phar Lap 386\|DOS-Extender)*

Rather than use explicit loops, the ISETL prime number program in Listing One (page 115) uses set notation. This program creates the set of all odd numbers less than *n*, takes the union of this set with the singleton set {2}, then takes the difference between the resulting set and the set of all odd composite numbers less than *n*. The resulting set is the set of all primes <= *n*. This can be expressed in a few lines of ISETL code.

Listing Two (page 115) performs the same operation, but uses ordered tuples (sets are, of course, unordered). I had to choose a small number *n* because, even with garbage collection, ISETL gobbles up a lot of memory.

Listing Three (page 115) is a program to generate the first 1000 Fibonacci numbers. This relies on ISETL's support for assignment to the return value of a function (which allows one to write functions that "remember" past values-dynamic programming) and ISETL's arbitrary-precision arithmetic. Fibonacci(1000) is a 209-digit number. ISETL/386 takes 15 seconds to compute the first 1000 Fibonacci numbers

in the WAT386 version and 14 seconds in the High C version. The 16-bit Turbo C ISETL takes 20.4 seconds.

The High C 386 version of ISETL was faster than the WAT386 version in every case tested. Overall, the High C version was about 15 percent faster than the WAT386 version. This is somewhat surprising since, as is well known, MetaWare produces High C by using an automatic compiler-compiler (which MetaWare markets separately as the Translator Writing System).

Programs such as Phar Lap's 386|DOS-Extender and Eclipse Computer Solutions's OS/386 do not replace DOS. They manage the interface between 16-bit real-mode DOS and your 32-bit protected-mode program

Profiling with the DOS/16M protected-mode debugger from Rational Systems (DOS/16M currently has the only decent protected mode C source-level debugging tools available), I found that ISETL generally spends 50 percent of its time in only four routines. Perhaps this test is somewhat lopsided. Any real program, on the other hand, will have similar "hot spots."

The Future

Over the next few months, both Watcom and MetaWare are planning major upgrades that may be out by the time you read this. One obvious change in High C is that while the 1.5 libraries are missing functions such as *open( )*, *fdopen( )*, *dup( )*, *fileno( )*, and *signal( )*, High C 1.6 is scheduled to include both a Microsoft-compatible standard library (including *_dos_keep( )*, *int86x( )*), a 32-bit version of the GFX graphics library, and a 32-bit version of the Sterling Castle C library.

WAT386's new release should include a 32-bit protected mode source-level debugger, a 32-bit version of Watcom's graphics library (which is identical to the MSC51 graphics library), a 32-bit version of the WAT386 compiler, and a 32-bit version of Watcom's Express in-memory quick compiler. The source-level debugger is urgently needed, and should put Watcom ahead in the 386 development tool race.

A 386 compiler war may indeed be starting. While WAT386 itself is not fully mature, its arrival is a sign of the growing strength of the market for 386 development tools. And about time too, now that the first 486s are rolling off the assembly line. But remember, even an 80586 will not save you from bad code.

Product Information

Watcom C7.0/386
Watcom
415 Phillip Street
Waterloo, Ontario, Canada N2L 3X2
800-265-4555
Price: $895
Requirements: 386-based PC- or PS/2 compatible, MS-DOS 3.1 or higher, 386 DOS extender toolkit: 386|DOS-Extender (Phar Lap) or OS/386 (Eclipse Computer Solutions)

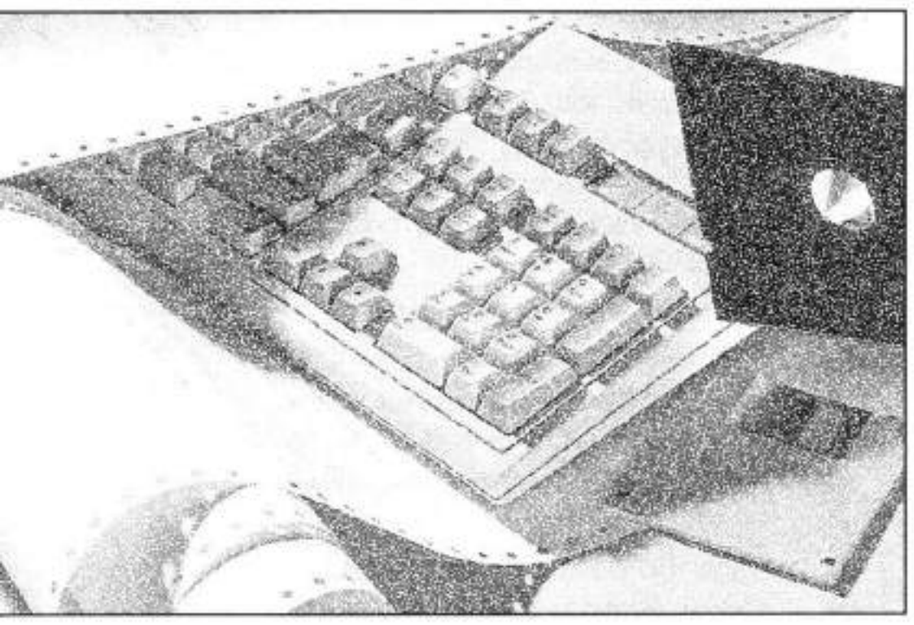
C Network Compiler/386
Novell Development Products
P.O. Box 9802
Austin, Texas 78766
512-346-8380
Price: $995

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 115.)



Mixed-Language Programming with ASM

Getting the job done often requires blending models and languages

Karl Wright and Rick Schell

As applications get larger, fewer and fewer are written in a single language. Large software projects tend to come together in a piecemeal fashion — some parts are borrowed from previous projects, other parts may be purchased from various vendor sources, and, let's face it, every programmer has a favorite language. Assembly languages have made great strides recently in the area of mixed language programming. Now more than ever before, it makes sense to write applications with more than one language and to include assembly language in the mix.

Furthermore, every programming language ever created has inherent strengths and weaknesses. One area in which different languages have distinct strengths is in how procedures are called. This is an extremely important issue, because in many applications more time and effort is spent getting in and out of procedures than doing anything else! Conversely, a good choice of procedure calling conventions can actually make the difference between an application that can be written quickly and one which cannot be written at all.

Usually, higher-level languages such as C and Pascal use an argument passing technique known as the "stack frame method," where arguments are pushed onto a stack and addressed as an offset from some "frame" pointer. It is a good general technique in that it allows for an unlimited number of arguments with built-in recursion.

C and Pascal each make use of a slightly different flavor of the stack frame method. The C-style stack frame permits a variable number of arguments to be passed to a procedure. This requires that the caller remove the arguments from the stack after the procedure call, because it is the caller who knows best how many arguments were passed. In Pascal, on the other hand, the number of arguments is fixed, so the procedure itself is responsible for removing its arguments from the stack. Typically, this is done efficiently with the single machine instruction *RET xx*.

Until recently, assembly language was generally limited to what is known as the "register passing method" of passing arguments. With register passing, arguments are passed to procedures in machine registers or at fixed memory locations. (Stack frames could be constructed in assembly language, but with considerable effort on the part of the programmer.) Register passing is not a general argument passing method. There are a limited number of registers in any machine, and explicit PUSH and POP instructions must be used to retain the availability of arguments during recursion. Nevertheless, register passing is a much more efficient method of passing arguments than the stack frame when the number of arguments to a procedure is small and the particular argument registers are chosen carefully in light of the instructions, which are to be done inside the procedure.

A Text "Spectrum Analyzer" Example

The example used to illustrate this point is a program that reads one or more text files, breaks them into words, and counts the individual words. It then sorts the resulting array by word count, and displays the word and the associated count together in a neat, tabular form.

This example emphasizes speed of execution, with the additional criteria that modularity is preserved and nasty tricks like self-modifying code are not used. This will permit the program to be relatively easy to change or to upgrade, and still be considerably faster than anything written wholly in a single language.

The major points that need to be covered are the interfaces between modules and what each module is responsible for, as well as the overall organization of the application.

The command line that this program will accept has the following format: *SPECTRUM <file_spec> <file_spec> . . .* where each *<file_spec>* can include wild cards. If a file name is given more than once, its spectrum will be taken more than once. The output of the application will be a table that is written to Standard Out and is sorted in order of reference count, the most referenced words being listed first.

Karl is the principal developer of Turbo Assembler and he can be reached at P.O. Box 39, Bedford, MA 01730. Rick is director of language development for Borland International and can be reached at 1800 Green Hills Road, Scotts Valley, CA 95065.

(continued from page 84)

The basic steps are: 1. Initialize all data structures. 2. Parse the command line. For each file spec, read the file(s) and break it (them) into words. Keep a reference count for each unique word. 3. Build a list of unique words and sort it by reference count. 4. Scan the sorted list and print out the reference count and associated word for each list element.

For the sake of performance, the work of reading a file, breaking it into words, and hashing them into a symbol table is best handled in assembly language, as is the other time bottleneck that occurs when the sort is done. Less time-critical areas, such as command-line parsing and table formatting, are written in C to provide greater flexibility in the user interface. Finally, the generality of assembly language, another inherent strength, makes it best for dealing with the heap and error handling modules.

The major modules we need and their respective languages are: ERROR.ASM, the assembly language error handler (see Listing One, page 116); HEAP.ASM, the assembly language memory allocator (Listing Two, page 116); WORD.ASM, the assembly language lexer/word, table/file input (Listing Three, page 116); SORT.ASM, the assembly language general-sort procedure (Listing Four, page 119); and SPECTRUM.C, the command-line parsing, text formatting, and output written in C (Listing Five, page 120). The *make* file is shown in Listing Six, page 121.

Throughout the program, we've made every effort to use an appropriate calling convention for the situation. On procedures with stack frames, Pascal-style calling conventions are most frequently used because of their inherently faster execution and smaller code requirements. Only on procedures that require a variable number of arguments do we use a C-style stack frame.

The extensive modularity we use in this application is not absolutely necessary given its small size. We have tried, however, to put forth as general a treatment as possible, demonstrating techniques that are appropriate even for very large applications. The use of strong data abstraction is one of these techniques. In strong data abstraction, the details of an actual data structure are known only to a small set of procedures that manage that data structure. The data structure and the procedures that manage it are taken together to form a module. Any other code in the program that deals with the data structure must do so through the appropriate procedures — any other access is considered to be a breach of modularity. In this application, the HEAP and WORD modules are good examples of strong data abstractions.

The program uses SMALL model with a NEAR stack. All of the code is in segment _TEXT (except for any code in the C libraries), so CS is always set to _TEXT. Data, uninitialized data, and stacks are all in DGROUP, so SS must always be set to DGROUP. DS is also set to DGROUP in the C sections of the program, but is used as a general segment register in the assembly language code.

The interfaces to the procedures in the various modules pretty well spell out the function of each module:

Error Handling Module Because errors need only to be caught and dis-

Assembler Specific Features

The assembly language section of the application was written in Borland's Turbo Assembler 2.0 and uses several features unique to that assembler. If you are using another assembler, you may need to modify portions of the example so that your assembler will accept it. The following are the features I used and how you can work around them in your assembler.

Extended CALL automatically builds a calling stack frame by generating a series of PUSHes in the order appropriate to the specified language. For example, *CALL foo pascal,ax,bx,wordptr* would PUSH the three arguments AX, BX, and WORDPTR onto the stack in the order appropriate for Pascal stack frames, and is equivalent to

```
PUSH ax
PUSH bx
PUSH wordptr
CALL foo
```

Multiple PUSHes/POPs permit more than one item at a time to be PUSHed or POPed with a single instruction. For example,

```
PUSH AX BX
POP BX AX
```

is equivalent to

```
PUSH AX
PUSH BX
POP BX
POP AX
```

Local Symbols are enabled with the LOCALS directive. All local symbols begin with the two characters @@. They are scoped to be local to the enclosing procedure. For example

```
foo1 proc
   jmp @@exit
@@exit:  ret
endp

foo2 proc
   jmp @@exit
@@exit:  ret  ;This @@EXIT can co-exist amicably with the former one.
endp
```

If you are using an assembler that does not support this feature, one way to work around it is to change the .MODEL statement at the start of each module to .MODEL SMALL, PASCAL. This will cause all symbols within a procedure to become local.

ARG and USES Statements the assembler used for the example has a way of setting up procedure stack frames that is somewhat easier to read than the standard method. For example:

```
foo proc pascal
arg a1,a2
uses ds,si
   . . .
```

is equivalent to the statement:

```
foo proc pascal uses ds si,a1,a2
   . . .
```

Some assemblers require a language to be specified in the .MODEL statement before the language keyword PASCAL is recognized. If this is true for your assembler, you will need to change the .MODEL statement at the start of each module to .MODEL SMALL,PASCAL.

The CODEPTR type is used occasionally in the example. It means either WORD or DWORD depending on whether the selected model has NEAR or FAR code, respectively. Because the example is SMALL model, you may replace CODEPTR with WORD wherever it is found.

— R.S.

```
void pascal ERROR_INIT (void)
        Initializes error module.

unsigned pascal ERROR_TRAP (void pascal (*execution_procedure)() )
        Returns 0 if no error occurred in the execution of
        EXECUTION_PROCEDURE or any procedures it calls. (Otherwise,
        an error code is returned.) EXECUTION_PROCEDURE is a
        generic procedure which can generate errors in its execution
        (via ERROR_LOG) and might be declared in C as follows:
        void pascal execution_procedure(void)

void pascal ERROR_LOG (unsigned error_code)
        Causes control to pass to the nearest enclosing ERROR_TRAP.
        Execution resumes with that instance of function ERROR_TRAP
        returning error_code.
```

Table 1: *Required procedures for error handling*

```
void pascal HEAP_INIT (unsigned starting_segment, unsigned segment_count)
        Initializes the heap to start at a certain segment and be
        a certain size.

void far * pascal HEAP_ALLOC (unsigned paragraph_count)
        Allocates the requested number of paragraphs from the
        heap and returns the far address of the memory in DX:AX.
        NOTE: The offset part of the address is always 0.
```

Table 2: *Required procedures for stack heap*

```
void pascal WORD_INIT (unsigned maximum_word_count)
        Initializes symbol table. The maximum number of
        different words allowed is passed so that a hash table
        can be initialized.

void pascal WORD_READ (unsigned file_handle)
        Reads all the text there is from the specified file
        handle and analyzes it.

void pascal WORD_SCAN (void pascal (*word_procedure)() )
        Calls the specified procedure once for each individual
        symbol. The word descriptor for the symbol is passed to
        WORD_PROCEDURE as an argument. WORD_PROCEDURE might
        be declared in C as follows:
        void pascal word_procedure(unsigned word_descriptor).

char far * pascal WORD_NAME (unsigned word_descriptor)
        Returns the FAR address of the name of the described symbol.

unsigned pascal WORD_REFCOUNT (unsigned word_descriptor)
        Returns the total reference count of the described symbol.

unsigned pascal WORD_COUNT (void)
        Returns the total number of distinct words processed so far.

int pascal WORD_COMPREF (unsigned word_descriptor1, unsigned
word_descriptor2)
        Compares the reference counts of two word descriptors.
        Returns flags for refcount(word_descriptor2) -
        refcount(word_descriptor1). NOTE: This procedure, while
        it obeys Pascal calling conventions, is not callable
        directly from C because it returns its result in the flag
        register. It also has the requirement that the registers
        CX and DX are preserved.

        This procedure might be described as using a sort of
        "hybrid" calling convention, where a stack frame is
        used but high-level language register conventions are not
        obeyed.
```

Table 3: *Procedures for symbol table*

(continued from page 86)

played without the ability to resume execution of the application, the error handling scheme this program uses is a mechanism whereby the stack pointer is saved at some point in the execution of the program, and if an error is encountered, the program is resumed at that point. The required procedures are listed in Table 1.

Heap Module Because data structures are allocated but never freed, a simple stack heap is the best choice for both performance and simplicity. The application uses a paragraph-based heap where memory is allocated with 16-byte granularity. This turns out to be useful because it permits any data item allocated from the heap to be described with a single 16-bit segment address. See Table 2.

Symbol Table Module The symbol table module is responsible for much of the actual work of reading in a file, converting it to words, and recording the word usage information. After it is read in, each symbol is represented by an area of memory allocated from the heap containing the reference count for the symbol and the actual text of the symbol. Because it is allocated from the heap, each symbol can be addressed by using a 16-bit word descriptor. Refer to Table 3.

Sorting Module The *sort* routine is written in assembly language because a recursive algorithm was chosen and recursion tends to be faster if register passing can be used appropriately. In this case, there are a small number of registers that are used directly; more importantly, during the innermost step of the recursion (which is done most often) no registers whatsoever need to be saved on the stack. Recursion with a stack frame can't make a decision this intelligent, because access to the arguments is needed first.

The *sort* procedure operates on an array of words, calling a generic comparison routine whose address is passed as an argument. This comparison routine uses a hybrid calling convention, where a stack frame is present but registers are not necessarily consistent with C. The level of generality this arrangement achieves is high, but it does require that the comparison routine be written in assembly language. See Table 4.

If raw speed were the only concern, the SORT_DO procedure might best be integrated entirely into the symbol table module, which would permit the comparison to be performed directly and would remove the need to call the comparison routine. But we felt that a more general treatment was superior in terms of modifiability — it is rela-

tively straightforward to add a switch to control the particular sorting method, for example.

The Command-line Parsing and Text Formatting Module We are now ready to lay out the full-scale sequencing of the program. Given the assembly language interface listed earlier, the following steps should be taken by the C portion of the program:

Assembly language's flexibility can assist in everything from optimization to the creation of programs using more than one interfacing convention

1. Allocate memory from DOS, call ERROR_INIT, and set up an error trap using ERROR_TRAP.
2. Call HEAP_INIT and WORD_INIT appropriately.
3. Parse the command line. For each file spec, call WORD_READ for all files matching the file spec (the C code is responsible for resolving all wild cards and for opening and closing each file).
4. Request the total number of unique words using WORD_COUNT, and allocate an array of 16-bit word descriptors using HEAP_ALLOC that is large enough to hold them. Call WORD_SCAN appropriately to fill up the array with word descriptors.
5. Sort the array using SORT_DO with the comparison routine WORD_COMP REF, which compares the count of references for two word descriptors.
6. Write the table title.
7. Scan the array to write out the table entries. Use WORD_REFCOUNT to get the reference count for each word descriptor, and WORD_NAME to get the name string for each word descriptor.

```
void pascal SORT_DO (unsigned far *sort_array, unsigned sort_count,
        int pascal (*compare_procedure)())
        Uses the specified compare procedure to order the array.
        COMPARE_PROCEDURE is called with two array values, and
        returns flags appropriate to a comparison of those
        values. Note that compare_procedure cannot be written in
        C because the value is returned in the machine flags. In
        addition, the segment registers are not guaranteed to be
        set up in a manner consistent with C when
        compare_procedure is called. Compare_procedure itself is
        expected to preserve CX and DX. The definition for
        compare_procedure might be stated:
        int pascal compare_procedure(unsigned value1, unsigned value2)
```

Table 4: *Procedures for sorting*

Theory of Operation

The SPECTRUM program uses a hash function and hash table to achieve its level of performance. Inside the WORD module, the procedure WORD_READ reads text into a buffer. This text is copied to a storage area one word at a time. During the copy operation, which uses the LODSB and STOSB instruc-

tions, the text is converted to uppercase and the hash value for the word is calculated, all on-the-fly.

The hash table is an array of word descriptors. An element in the hash table is 0 if there is not yet an associated symbol. The hash function is calculated by looking at each character in the word, rotating the previous hash value circularly left by five, and XORing in the character value. The final hash value is masked off to become an index into the hash table.

Now more than ever before, it makes sense to write applications with more than one language and to include assembly language in the mix

After the hash index is calculated, the corresponding hash table entry is checked. If it is 0, a new symbol is created, and its reference count is initialized to 1. Otherwise, the text of the word is compared against the text stored in the symbol whose word descriptor is found in the hash table. If it agrees, the correct symbol has been located, and its reference count is incremented. If not, a collision has occurred, and the next hash value is calculated by adding *11*2* to the current hash index (this number must be relatively prime to the size of the hash table). The process then repeats until the correct hash table entry or a 0 is found.

An unusual technique is used to speed the recognition of the various different character types during the lexing process. BX is initialized to point to a translation table, which contains a bit for each pertinent character type. An XLAT instruction followed by a *TEST AL,xxx* is then all that is needed to identify a character as a numeral, delimiter, lowercase alphabetic, and so on.

Another unusual technique is used to describe objects in the assembly language section of the program. Rather than use a full 32 bits to describe the address of a data object, which is somewhat cumbersome, a paragraph address is used instead. This paragraph address becomes the "descriptor" for the object. Data within the object is addressed by loading an appropriate segment register with the object descriptor and accessing the data with a constant offset using that segment register.

After all files have been read in and parsed, an array of word descriptors is built using the routine WORD_SCAN. This array is then sorted using SORT_DO with the comparison routine WORD_COMPREF. SORT_DO is a recursive sort that requires N*LOG(N) comparisons. It operates by dividing the array into two roughly equal parts, recursively sorting each part, and then merging the two parts in place.

Finally, to output the table, the array is scanned sequentially. For each word descriptor in the array, WORD_NAME is used to obtain the actual text of the word, and WORD_REFCOUNT is used to obtain the reference count. These values are displayed using PRINTF.

Conclusion

It is not only practical but advisable to mix languages and models in order to achieve the best results. Modern assembly language is a vital part of this mix, and will continue to be important in the future, because space and performance are always important for competitive software, no matter how powerful the hardware becomes. Assembly language's flexibility can assist in everything from optimization to the creation of programs using more than one interfacing convention.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 116.)

Listing One *(Text begins on page 16.)*

```
/* Sample program to copy one far string to another far string,
 * converting lowercase letters to uppercase letters in the process. */

#include <ctype.h>
char Source[] = "AbCdEfGhIjKlMnOpQrStUvWxYz0123456789!";
char Dest[100];
/* Copies one far string to another far string, converting all lower
 * case letters to upper case before storing them.  */
void CopyUppercase(char far *DestPtr, char far *SourcePtr) {
   char UpperSourceTemp;
   do {
      /* Using UpperSourceTemp avoids a second load of the far pointer
         SourcePtr as the toupper macro is expanded */
      UpperSourceTemp = *SourcePtr++;
      *DestPtr++ = toupper(UpperSourceTemp);
   } while (UpperSourceTemp);
}
main() {
   CopyUppercase((char far *)Dest,(char far *)Source);
}
```

End Listing One

Listing Two

```
; C near-callable subroutine, callable as:
;       void CopyUppercase(char far *DestPtr, char far *SourcePtr);
; Copies one far string to another, converting all lowercase letters
; to upper case before storing them. Strings must be zero-terminated.
;
parms   struc
        dw      ?       ;pushed BP
        dw      ?       ;return address
DestPtr dd      ?       ;destination string
SourcePtr dd    ?       ;source string
parms   ends
;
        .model small
        .code
        public _CopyUppercase
_CopyUppercase  proc    near
        push    bp
        mov     bp,sp                   ;set up stack frame
        push    si                      ;preserve C's register vars
        push    di
        push    ds                      ;we'll point DS to source
                                        ;segment for the duration of the loop.
        les     di,[bp+DestPtr]         ;point ES:DI to destination
        lds     si,[bp+SourcePtr]       ;point DS:SI to source
CopyAndConvertLoop:
        lodsb                           ;get next source byte
        cmp     al,'a'                  ;is it lowercase?
        jb      SaveUpper               ;no
        cmp     al,'z'                  ;is it lowercase?
        ja      SaveUpper               ;no
        and     al,not 20h              ;convert to uppercase
SaveUpper:
        stosb                           ;store the byte to the dest
        and     al,al                   ;is this the terminating 0?
        jnz     CopyAndConvertLoop      ;if not, repeat loop
        pop     ds                      ;restore caller's DS
        pop     di                      ;restore C's register vars
        pop     si
        pop     bp                      ;restore caller's stack frame
        ret
_CopyUppercase  endp
        end
```

End Listing Two

Listing Three

```
/* Sample program to copy one near string to another near string,
 * converting lower case letters to upper case letters in the process.  */

#include <ctype.h>
char Source[] = "AbCdEfGhIjKlMnOpQrStUvWxYz0123456789!";
char Dest[100];
/* Copies one near string to another near string, converting all lower
 * case letters to upper case before storing them.  */
void CopyUppercase(char *DestPtr, char *SourcePtr) {
   char UpperSourceTemp;
   do {
      /* Using UpperSourceTemp allows slightly better optimization
         than using *SourcePtr directly */
      UpperSourceTemp = *SourcePtr++;
      *DestPtr++ = toupper(UpperSourceTemp);
   } while (UpperSourceTemp);
}
main() {
   CopyUppercase(Dest,Source);
}
```

End Listing Three

Listing Four

```
; C near-callable subroutine, callable as:
;       void CopyUppercase(char *DestPtr, char *SourcePtr);
; Copies one near string to another, converting all lowercase letters to
; uppercase before storing them. Strings must be zero-terminated.
;
parms   struc
        dw      ?       ;pushed BP
        dw      ?       ;return address
DestPtr dw      ?       ;destination string
SourcePtr dw    ?       ;source string
parms   ends
;
        .code
        public _CopyUppercase
_CopyUppercase  proc    near
        push    bp
        mov     bp,sp                   ;set up stack frame
        push    si                      ;preserve C's register vars
        push    di
        mov     di,[bp+DestPtr]         ;point DI to destination
        mov     si,[bp+SourcePtr]       ;point SI to source
        mov     cx,('a' shl 8) + 'z'    ;preload CH with lower end of
                                        ; lowercase range and CL with
                                        ; upperend of that range
        mov     bl,not 20h              ;preload BL with value used to
                                        ; convert to uppercase
CopyAndConvertLoop:
        lodsw                           ;get next two source bytes
        cmp     al,ch                   ;is the 1st byte lowercase?
        jb      SaveUpper               ;no
        cmp     al,cl                   ;is the 1st byte lowercase?
        ja      SaveUpper               ;no
        and     al,bl                   ;convert 1st byte to uppercase
SaveUpper:
        and     al,al                   ;is the 1st byte the terminating 0?
        jz      SaveLastAndDone         ;yes, save it & done
        cmp     ah,ch                   ;is the 2nd byte lowercase?
        jb      SaveUpper2              ;no
        cmp     ah,cl                   ;is the 2nd byte lowercase?
        ja      SaveUpper2              ;no
        and     ah,bl                   ;convert 2nd byte to uppercase
SaveUpper2:
        stosw                           ;store both bytes to the dest
        and     ah,ah                   ;is the 2nd byte the terminating 0?
        jnz     CopyAndConvertLoop      ;if not, repeat loop
        jmp     short Done              ;if so, we're done
SaveLastAndDone:
        stosb                           ;store the final 0 to the dest
Done:
        pop     di                      ;restore C's register vars
        pop     si
        pop     bp                      ;restore caller's stack frame
        ret
_CopyUppercase  endp
        end
```

End Listing Four

Listing Five

```
; C near-callable subroutine, callable as:
;       void CopyUppercase(char *DestPtr, char *SourcePtr);
; Copies one near string to another, converting all lowercase letters to
; upper case before storing them. Strings must be zero-terminated. Uses
; extensive optimization for enhanced performance.
;
parms   struc
        dw      ?       ;pushed BP
        dw      ?       ;return address
DestPtr dw      ?       ;destination string
SourcePtr dw    ?       ;source string
parms   ends
;
        .model small
        .data
; Table of mappings to uppercase for all 256 ASCII characters.
UppercaseConversionTable        label   byte
ASCII_VALUE=0
        rept    256
if (ASCII_VALUE lt 'a') or (ASCII_VALUE gt 'z')
        db  ASCII_VALUE             ;non-lowercase characters map to themselves

else
        db  ASCII_VALUE and not 20h  ;lowercase chars map to upper equivalents
endif
ASCII_VALUE=ASCII_VALUE+1
        endm
        .code
        public _CopyUppercase
_CopyUppercase  proc    near
        push    bp
        mov     bp,sp                   ;set up stack frame
        push    si                      ;preserve C's register vars
        push    di
        mov     di,[bp+DestPtr]         ;point DI to destination
        mov     si,[bp+SourcePtr]       ;point SI to source
        mov     bx,offset UppercaseConversionTable
                                        ;point BX to lowercase to
                                        ; uppercase mapping table
; This loop processes up to 16 bytes from the source string at a time,
; branching only every 16 bytes or after the terminating 0 is copied.
CopyAndConvertLoop:
        rept    15                      ;for up to 15 bytes in a row...
        lodsb                           ;get the next source byte
        xlat                            ;make sure it's upper case
        stosb                           ;save it to the destination
        and     al,al                   ;is this the terminating 0?
        jz      Done                    ;if so, then we're done
        endm
        lodsb                           ;get the next source byte
        xlat                            ;make sure it's upper case
        stosb                           ;save it to the destination
        and     al,al                   ;is this the terminating 0?
        jnz     CopyAndConvertLoop      ;if not, repeat loop
Done:
        pop     di                      ;restore C's register vars
        pop     si
        pop     bp                      ;restore caller's stack frame
        ret
_CopyUppercase  endp
        end
```

End Listings

Listing One *(Text begins on page 46.)*

```
;*****************************************************************************
;* File: BREAK386.ASM                                                        *
;* BREAK386 "main programs". Contains setup386, clear386, break386 and       *
;* int1_386.                                                                 *
;* Williams - June, 1989                                                     *
;* Compile with: MASM /Ml BREAK386;                                          *
;*****************************************************************************
.MODEL small
.386P

      public _break386,_clear386,_setup386,_int1_386

; Set up stack offsets for word size arguments based on the code size
; Be careful, regardless of what Microsoft's documentation says,
; you must use @CodeSize (not @codesize, etc.) when compiling with /Ml

IF @CodeSize                ; True for models with far code
arg1       EQU  <[BP+6]>
arg2       EQU  <[BP+8]>
arg3       EQU  <[BP+10]>
arg4       EQU  <[BP+12]>
ELSE
arg1       EQU  <[BP+4]>
arg2       EQU  <[BP+6]>
arg3       EQU  <[BP+8]>
arg4       EQU  <[BP+10]>
ENDIF



.DATA
; Things you may want to change:
DIRECT      EQU 0          ; IF 0 use BIOS; IF 1 use direct video access
STKWRD      EQU 32         ; # of words to dump off the stack
INTSTACK    EQU 1          ; When 0 don't display interrupt stack words
USE_INT1    EQU 1          ; Set to 0 to disable int1_386()

oldoffset   dw 0           ; old interrupt 1 vector offset
oldsegment  dw 0           ; old interrupt 1 vector segment

IF USE_INT1
video       dw 0b000H      ; segment of video adapter (changed by vinit)
csip        db 'CODE=',0
done        db 'Program terminated normally.',0
notdone     db 'Program breakpoint:',0
stkmess     db 'Stack dump:',0

vpage       db 0
vcols       db 80

IFE DIRECT
prompt      db '<V>iew output, <T>race toggle, <C>ontinue or <A>bort? ',0
savcursor   dw 0           ; inactive video cursor
ALIGN 4
vbuff       dd 1000 dup (07200720H)
ELSE
cursor      dw 0
color       db 7
ENDIF
ENDIF

.CODE

; This is the start up code. The old interrupt one vector is saved in
; oldsegment, oldoffset. int1_386 does not chain to the old vector, it
; simply replaces it.

_setup386 proc
        push bp
        mov bp,sp
        push es
        mov ax,3501H               ; get old int1 vector
        int 21h
        mov ax,es
        mov oldsegment,ax
        mov oldoffset,bx
        pop es
        mov ax,arg2                ; get new interrupt handler address
        push ds
        mov dx,arg1
; If int1_386 is being assembled, setup386 will check to see if you are
; installing int1386. If so, it will call vinit to set up video parameters
; that int1_386 requires.
IF USE_INT1
        cmp ax,seg _int1_386
        jnz notus
        cmp dx,offset _int1_386
        jnz notus
        push dx
        push ax
        call vinit                 ; Int'l video if it is our handler
        pop ds
        pop dx
ENDIF
notus:  mov ax,2501H               ; Store interrupt address in vector table
        int 21H
        pop ds
        xor eax,eax                ; Clear DR7/DR6 (just in case)
        mov dr7,eax
        mov dr6,eax
        pop bp
        ret
_setup386 endp


; This routine sets/clears breakpoints
; Inputs:
;    breakpoint # (1-4)
;    breakpoint type (see BREAK386.INC)
;    segment/offset of break address (or null to clear breakpoint)
; Outputs:
;    AX=0 If successful
;    AX=-1 If not successful

_break386 proc
        push bp
        mov bp,sp
        mov bx,arg1                ; breakpoint # (1-4)
        cmp bx,1
        jb outrange
        cmp bx,4
        jna nothigh
outrange:
        mov ax,0ffffH              ; error: breakpoint # out of range
        pop bp
        ret
nothigh:
        movzx eax,word ptr arg4    ; get breakpoint address
        shl eax,4
        movzx edx,word ptr arg3    ; calculate linear address
        add eax,edx                ; if address = 0 then
        jz resetbp                 ; turn breakpoint off!
        dec bx                     ; set correct address register
        jz bp0
        dec bx
        jz bp1
        dec bx
        jz bp2
        mov dr3,eax
        jmp short brcont
bp0:    mov dr0,eax
        jmp short brcont
bp1:    mov dr1,eax
        jmp short brcont
bp2:    mov dr2,eax
brcont:
        movzx eax,word ptr arg2    ; get type
        mov cx,arg1                ; calculate proper position
        push cx
        dec cx
        shl cx,2
        add cx,16
        shl eax,cl                 ; rotate type
        mov edx,0fh
        shl edx,cl                 ; calculate type mask
        not edx
        pop cx
        shl cx,1                   ; calculate position of enable bit
        dec cx
        mov ebx,1
        shl ebx,cl
        or eax,ebx                 ; enable bp
        mov ebx,dr7                ; get old DR7
        and ebx,edx                ; mask out old type
        or ebx,eax                 ; set new type/enable bits
; Adjust enable bit (set on for data bp's, off if no data bp's)
adjge:
        mov eax,200H
        and ebx,0fffffdffH         ; reset GE bit
        test ebx,033330000H        ; test for data bp's
        jz nodatabp
        or ebx,512
nodatabp:
        mov dr7,ebx
        pop bp
        xor ax,ax
        ret
; Here we reset a breakpoint by turning off its enable bit & setting type to 0
; Clearing the type is required so that disabling all data breakpoints will
; clear the GE bit also.
resetbp:
        mov cx,bx                  ; calculate type/len bit positions
        mov edx,0fh
        dec cx
        shl cx,2
        add cx,16
        shl edx,cl
        not edx
        mov cx,bx                  ; calculate enable bit position
        shl cx,1
        dec cx
        mov eax,1
        shl eax,cl
        not ax                     ; flip bits
        mov ebx,dr7
        and ebx,eax                ; clear enable
        and ebx,edx                ; clear type
        jmp adjge
_break386 endp
; Reset the debug register, disabling all breakpoint. Also restore the old
; interrupt 1 vector
_clear386 proc
        pushf
        pop ax
        and ax,0FEFFH              ; turn off trace flag
        push ax
        popf
        xor eax,eax                ; turn off all other breakpoints
        mov dr7,eax
        mov dr0,eax
        mov dr1,eax
        mov dr2,eax
        mov dr3,eax
        mov dr6,eax
        mov ax,2501H               ; restore old int 1 vector
        push ds
        mov bx,oldsegment
        mov dx,oldoffset
        mov ds,bx
        int 21H
        pop ds
```

(continued on page 98)

Listing One *(Listing continued, text begins on page 46.)*

```
        ret
_clear386 endp

IF USE_INT1
; This is all code relating to the optional INT 1 handler

; This macro is used to get a register value off the stack and display it
; R is the register name and n is the position of the register on the stack
; i.e.: outreg 'AX',10

outreg   macro r,n
         mov ax,&r
         mov dx,[ebp+&n SHL 1]
         call regout
         endm


; This is the interrupt 1 handler
_int1_386  proc far
         sti                            ; Enable interrupts (see text)
         pusha                          ; Save all Registers
         push ds
         push es
         push ss
         push @data
         pop ds                         ; Reload DS
         mov bp,sp                      ; point ebp to top of stack
IFE DIRECT
        call savevideo
ENDIF
         mov ax,video                   ; get video addressabilty
         mov es,ax
         assume cs:@code,ds:@data
         mov bx,offset notdone          ; Display breakpoint message
         call outstr
         mov edx,dr6
         call  hexout
         xor edx,edx
         mov dr6,edx
         call  crlf
;do register dump
         outreg 'AX',10
         outreg 'FL',13
         outreg 'BX',7
         outreg 'CX',9
         outreg 'DX',8
         call  crlf
         outreg 'SI',4
         outreg 'DI',3
         outreg 'SP',6
         outreg 'BP',5
         call  crlf
         outreg 'CS',12
         outreg 'IP',11
         outreg 'DS',2
         outreg 'ES',1
         outreg 'SS',0
         call  crlf
 ; do stack dump
IF STKWRD
         mov bx,offset stkmess
         call outstr                    ; Print stack dump title
         push fs
         mov dx,[ebp]                   ; get program's ss
         mov fs,dx
         mov al,'('
         call  ouch
         mov al,' '
         call  ouch
         call  hexout
         mov al,':'
         call  ouch
         mov al,' '
         call  ouch
         mov bx,[ebp+12]                ; get stack pointer (before pusha)
IFE INTSTACK
         add bx,6                       ; skip interrupt info if desired
ENDIF
         mov dx,bx
         push bx
         call  hexout
         mov al,')'
         call  ouch
         call  crlf
         pop bx
         mov cx,STKWRD
 sloop:

         mov dx,fs:[bx]                 ; get word at stack
         push bx
         push cx
         call  hexout                    ; display it
         pop cx
         pop bx
         inc bx
         inc bx
         loop sloop
         pop fs
ENDIF
nostack:
; Here we will dump 16 bytes starting 8 bytes prior to the instruction
; that caused the break
         push fs
         call  crlf
         mov bx, offset csip
         call outstr
         mov cx,8
         mov ax,[ebp+24]                ; get cs
         mov fs,ax
         mov bx,[ebp+22]                ; get ip
         cmp bx,8                       ; make sure we have 8 bytes before
         jnb ipbegin                    ; the begining of the segment
         mov cx,bx                      ; If not, only dump from the start
ipbegin: sub bx,cx                      ; of the segment
         push bx
         push cx
         mov dx,ax                      ; display address
         call  hexout
         mov al,':'
         call  ouch
         mov al,' '
         call  ouch
         mov dx,bx
         call  hexout
         mov al,'='
         call  ouch
         pop cx
         pop bx
         or bx,bx                       ; if starting at 0, don't display any
         jz ipskip                      ; before IP
iploop:
         mov dl,fs:[bx]                 ; get byte
         push bx
         push cx
         call  hexlout                   ; output it
         pop cx
         pop bx
         inc bx
         loop iploop
ipskip:
         push bx
         mov al,'*'                     ; put '*' before IP location
         call  ouch
         mov al,' '
         call  ouch
         pop bx
; This is basically a repeat of the above loop except it dumps the 8 bytes
; starting at IP
         mov cx,8
xiploop:
         mov dl,fs:[bx]
         push bx
         push cx
         call  hexlout
         pop cx
         pop bx
         inc bx
         loop xiploop
         call  crlf
         call  crlf
         pop fs
IFE DIRECT
; Here we will ask if we should continue or abort
         mov bx,offset prompt
         call outstr
keyloop:
         xor ah,ah                      ; Get keyboard input
         int 16H
         and al,0dfh                    ; make upper case
         cmp al,'T'
         jz ttoggle
         cmp al,'A'
         jz ql
         cmp al,'C'
         jz cl
         cmp al,'V'
         jnz keyloop
; Display program's screen until any key is pressed
         call savevideo
         xor ah,ah
         int 16H
         call savevideo
         jmp keyloop

; Execution comes here to toggle trace flag and continue
ttoggle:
         xor word ptr [bp+26],256       ; toggle trace flag on stack

; Execution comes here to continue running the target program
cl:
         call  crlf
IFE DIRECT
         call savevideo
ELSE
         xor ax,ax
         mov cursor,ax
ENDIF
         pop ss
         pop es
         pop ds
         popa
; This seems complicated at first.
; You MUST insure that RF is set before continuing. If RF is not set
; you will just cause a breakpoint immediately!
; In protected mode, this is handled automatically. In real mode it
; isn't since RF is in the high 16 bits of the flags register.
; Essentially we have to convert the stack from:
;
;       16 bit Flags                    32 bit flags (top word = 1 to set RF)
;       16 bit CS       to ----->       32 bit CS    (garbage in top 16 bits)
;       16 bit IP                       32 bit IP    (top word = 0)
;
; All this so we can execute an IRETD which will change RF.

         sub esp,6              ; make a double stack frame
         xchg ax,[esp+6]        ; get ip in ax
         mov [esp],ax           ; store it
         xor ax,ax
         mov [esp+2],ax         ; eip = 0000:ip
         mov ax,[esp+6]
         xchg ax,[esp+8]        ; get cs
```

(continued on page 100)

Listing One *Listing continued, text begins on page 46.)*

```
        mov [esp+4],ax
        xor ax,ax
        mov [esp+6],ax
        mov ax,[esp+8]          ; zero that stack word & restore ax
        xchg ax,[esp+10]        ; get flags
        mov [esp+8],ax
        mov ax,1                ; set RF
        xchg ax,[esp+10]
        iretd                   ; DOUBLE IRET (32 bits!)

ENDIF

; Execution resumes here to abort the target program
ql:
IFE DIRECT
        call savevideo
ENDIF
        call quit
_int1_386  endp

IFE DIRECT
; save video screen & restore ours (only with BIOS please!)
; (assumes 25 lines/page)
savevideo proc near
        pusha
        push es
        mov ah,0fh
        int 10h                         ; reread video page/size in case
        mov vpage,bh                    ; program changed it
        mov vcols,ah

        push savcursor
        mov ah,3                        ; get old cursor
        mov bh,vpage
        int 10H
        mov savcursor,dx
        pop dx
        mov ah,2                        ; set new cursor
        int 10H
        movzx ax,vpage
        mov cl,vcols                    ; compute # bytes/page
        xor ch,ch
        mov dx,cx                       ; vcols * 25 * 2
        shl cx,3
        shl dx,1
        add cx,dx
        mov dx,cx
        shl cx,2
        add cx,dx
        push cx
        mul cx
        mov di,ax                       ; start at beginning of page
        pop cx
        shr cx,2                        ; # of double words to transfer
        mov ax,video
        mov es,ax
        mov si,offset vbuff             ; store inactive screen in vbuff
xloop:  mov eax,es:[di]                 ; swap screens
        xchg eax,[si]
        mov es:[di],eax
        add si,4
        add di,4
        loop xloop
        pop es
        popa
        ret
savevideo endp
ENDIF


; This routine prints a register value complete with label
; The register name is in AX and the value is in dx (see the outreg macro)
regout  proc near
        push dx
        push ax
        mov al,ah
        call  ouch
        pop ax
        call  ouch
        mov al,'='
        call  ouch
        pop dx
        call  hexout
        ret
regout  endp

; Plain vanilla hexadecimal digit output routine
hexdout proc near
        and dl,0fh
        add dl,'0'
        cmp dl,3ah
        jb ddigit
        add dl,'A'-3ah
ddigit:
        mov al,dl
        call ouch
        ret
hexdout endp

; Plain vanilla hexadecimal word output routine
hexout  proc near
        push dx
        shr dx,12
        call hexdout
        pop dx
        push dx
        shr dx,8
        call hexdout
        pop dx

; Call with this entry point to output just a byte
hexlout:
        push dx
        shr dx,4
        call hexdout
        pop dx
        call hexdout
        mov al,' '
        call ouch
        ret
hexout  endp


; These routines are for direct video output. Using them allows you to
; debug video bios calls, but prevents you from single stepping IF DIRECT
;output a character in al assumes ds=dat es=video destroys bx,ah
ouch    proc near
        mov bx,cursor
        mov ah,color
        mov es:[bx],ax
        inc bx
        inc bx
        mov cursor,bx
        ret
ouch    endp

; <CR> <LF> output.  assumes ds=dat es=video  destroys ax,cx,dx,di  clears
df
crlf    proc near
        mov ax,cursor
        mov cx,160
        xor dx,dx
        div cx
        inc ax
        mul cx
        mov cursor,ax
        mov cx,80
        mov ah,color
        mov al,' '
        mov di,cursor
        cld
        rep stosw
        ret
crlf    endp

ELSE
; These are the BIOS output routines
; Output a character
ouch    proc near
        mov ah,0eh
        mov bh,vpage
        int 10h
        ret
ouch    endp
; <CR> <LF> output.
crlf    proc near
        mov al,0dh
        call ouch
        mov al,0ah
        call ouch
        ret
crlf    endp


ENDIF

; Intialize the video routines
vinit   proc near
        mov ah,0fh
        int 10h
        mov vcols,ah
        mov vpage,bh
        cmp al,7                        ; monochrome
        mov ax,0b000H
        jz vexit
        mov ax,0b800H
vexit:  mov video,ax
        ret
vinit   endp


; outputs string pointed to by ds:bx (ds must be dat) es= video when DIRECT=1
outstr  proc near
outagn:
        mov al,[bx]
        or al,al
        jz outout
        push bx
        call ouch
        pop bx
        inc bx
        jmp outagn
outout: ret
outstr  endp


; This routine is called to return to DOS
quit    proc near
        call _clear386
        mov ax,4c00h                    ; Return to DOS
        int 21h
quit    endp


ENDIF

        end
```

End Listing One

(continued on page 100)

Listing Two *(Text begins on page 46.)*

```
;*****************************************************************************
;* File: BREAK386.INC                                                        *
;* Header file to include with assembly language programs using BREAK386    *
;* Williams - June, 1989                                                     *
;*                                                                           *
;*****************************************************************************
IF @CodeSize       ; If large style models
        extrn _break386:far,_clear386:far,_setup386:far,_int1_386:far
ELSE
        extrn _break386:near,_clear386:near,_setup386:near,_int1_386:far
ENDIF

; Breakpoint equates
BP_CODE     EQU  0              ; CODE BREAKPOINT
BP_DATAW1   EQU  1              ; ONE BYTE DATA WRITE BREAKPOINT
BP_DATARW1  EQU  3              ; ONE BYTE DATA R/W BREAKPOINT
BP_DATAW2   EQU  5              ; TWO BYTE DATA WRITE BREAKPOINT
BP_DATARW2  EQU  7              ; TWO BYTE DATA R/W BREAKPOINT
BP_DATAW4   EQU 13              ; FOUR BYTE DATA WRITE BREAKPOINT
BP_DATARW4  EQU 15              ; FOUR BYTE DATA R/W BREAKPOINT

; Macros to turn tracing on and off
; Note: When tracing, you will actually "see" traceoff before it turns
;       tracing off

traceon  macro
         push bp
         pushf
         mov bp,sp
         xchg ax,[bp]
         or ax,100H
         xchg ax,[bp]
         popf
         pop bp
         endm

traceoff macro
         push bp
         pushf
         mov bp,sp
         xchg ax,[bp]
         and ax,0FEFFH
         xchg ax,[bp]
         popf
         pop bp
         endm
```

End Listing Two

Listing Three

```
/*****************************************************************************
* File: BREAK386.H                                                           *
*                                                                            *
* Header for C programs using BREAK386 or CBRK386                            *
* Williams - June, 1989                                                      *
*                                                                            *
*****************************************************************************/

#ifndef NO_EXT_KEYS
  #define _CDECL cdecl
#else
  #define _CDECL
#endif

#ifndef BR386_HEADER
  #define BR386_HEADER

  /* declare functions */
  void  _CDECL setup386(void (_CDECL interrupt far *)());
  void  _CDECL csetup386(void (_CDECL far *)());
  void  _CDECL clear386(void);
  int   _CDECL break386(int,int, void far *);
  void  _CDECL far interrupt int1_386();

  /* breakpoint types */
  #define BP_CODE        0          /* CODE BREAKPOINT*/
  #define BP_DATAW1      1          /* ONE BYTE DATA WRITE BREAKPOINT*/
  #define BP_DATARW1     3          /* ONE BYTE DATA R/W BREAKPOINT*/
  #define BP_DATAW2      5          /* TWO BYTE DATA WRITE BREAKPOINT
  #define BP_DATARW2     7          /* TWO BYTE DATA R/W BREAKPOINT*/
  #define BP_DATAW4     13          /* FOUR BYTE DATA WRITE BREAKPOINT*/
  #define BP_DATARW4    15          /* FOUR BYTE DATA R/W BREAKPOINT*/

#endif
```

End Listing Three

Listing Four

```
;*****************************************************************************
;* File: DEBUG386.ASM                                                        *
;* Example assembly language program for use with BREAK386                   *
;* Williams - June, 1989                                                     *
;* Compile with: MASM /Ml DEBUG386.ASM;                                      *
;*****************************************************************************
.model large
.386
INCLUDE break386.inc
.stack 0a00H

.data
align 2                                 ; make sure this is word aligned
memcell dw 0                            ; cell to write to

.code
main    proc
;setup data segment
        mov ax,@data
        mov ds,ax
        assume cs:@code,ds:@data

; start debugging
        push seg _int1_386              ; segment of interrupt handler
        push offset _int1_386           ; offset of interrupt handler
        call _setup386
        add sp,4                        ; balance stack (like a call to C)
; set up a starting breakpoint
        push seg bp1                    ; segment of breakpoint
        push offset bp1                 ; offset of breakpoint
        push BP_CODE                    ; breakpoint type
        push 1                          ; breakpoint # (1-4)
        call _break386
        add sp,8                        ; balance the stack

        push seg bp2                    ; set up breakpoint #2
        push offset bp2
        push BP_CODE
        push 2
        call _break386
        add sp,8

        push seg bp3                    ; set up breakpoint #3
        push offset bp3
        push BP_CODE
        pus' -
        call _break386
        add sp,8

        push @data                      ; set up breakpoint #4 (data)
        push offset memcell
        push BP_DATAW2
        push 4
        call _break386
        add sp,8


bp1:
        mov cx,20                       ; loop 20 times
loop1:
        mov dl,cl                       ; print some letters
        add dl,'@'
        mov ah,2
bp2:
        int 21h
bp3:
        loop loop1                      ; repeat

        mov bx,offset memcell           ; point bx at memory cell
        mov ax,[bx]                     ; read cell (no breakpoint)
        mov [bx],ah                     ; this should cause breakpoint 4
        call _clear386                  ; shut off debugging
        mov ah,4ch
        int 21h                         ; back to DOS
main    endp
        end main
```

End Listing Four

Listing Five

```
/*****************************************************************************
*                                                                            *
* File: DBG386.C                                                             *
* Example C program using BREAK386 with the built in interrupt handler      *
*                                                                            *
* Al Williams  -- 15 July 1989                                               *
* Compile with: CL DBG386.C BREAK386                                         *
*****************************************************************************/
#include <stdio.h>
#include <dos.h>
#include "break386.h"

int here[10];
void far *bp;
int i;

main()
{
  int j;
  setup386(int1_386);           /* set up debugging */
  bp=(void far *)&here[2];      /* make long pointer to data word */
  break386(1,BP_DATAW2,bp);     /* set breakpoint */
  for (j=0;j<2;j++) {           /* loop twice */
    for (i=0;i<10;i++)          /* for each element in here[] */
     {
     char x;
     putchar(i+'0');            /* print index digit */
     here[i]=i;                 /* assign # to array element */
     }
  break386(1,0,NULL);           /* turn off breakpoint on 2nd pass */
  }
  clear386();                   /* turn off debugging */
}
```

End Listing Five

(continued on page 104)

Listing Six *(Text begins on page 46.)*

```
;*****************************************************************************
;* File: DBGOFF.ASM                                                          *
;*                                                                           *
;* Try this program if you leave a program abnormally (say, with a stack     *
;* overflow). It will reset the debug register.                              *
;* Williams - June, 1989                                                     *
;* Compile with: MASM DBGOFF;                                                *
;*****************************************************************************

.model small
.386P
.stack 32
.code

main proc
     xor eax,eax                        ; clear dr7
     mov dr7,eax
     mov ah,4ch                         ; exit to DOS
     int 21H
main endp
     end main
```

End Listing Six

Listing Seven

```
;*****************************************************************************
;* File: CBRK386.ASM                                                         *
;* Functions to allow breakpoint handlers to be written in C.                *
;* Williams - June, 1989                                                     *
;* Compile with: MASM /Ml CBRK386.ASM;                                       *
;*****************************************************************************
.MODEL small
.386P

      public _csetup386

; Set up stack offsets for word size arguments based on the code size
; Be careful, regardless of what Microsoft's documentation says,
; you must use @CodeSize (not @codesize, etc.)

IF @CodeSize                ; True for models with far code
arg1        EQU  <[BP+6]>
arg2        EQU  <[BP+8]>
arg3        EQU  <[BP+10]>
arg4        EQU  <[BP+12]>
ELSE
arg1        EQU  <[BP+4]>
arg2        EQU  <[BP+6]>
arg3        EQU  <[BP+8]>
arg4        EQU  <[BP+10]>
ENDIF


.DATA
; You may need to change the next line to expand the stack your breakpoint
; handler runs with
STACKSIZE   EQU 2048

oldoffset   dw 0                        ; old interrupt 1 vector offset
oldsegment  dw 0                        ; old interrupt 1 vector segment
oldstack    equ this dword
sp_save     dw 0
ss_save     dw 0
ds_save     dw 0
es_save     dw 0
ccall       equ this dword              ; C routine's adress is saved here
c_off       dw 0
c_seg       dw 0
oldstkhqq   dw 0                        ; Old start of stack


newsp       equ this dword              ; New stack address for C routine
            dw offset stacktop
            dw seg newstack

; Here is the new stack. DO NOT MOVE IT OUT OF DGROUP
; That is, leave it in the DATA or DATA? segment.
newstack    db STACKSIZE DUP (0)
stacktop    EQU $
            extrn STKHQQ:word           ; Microsoft heap/stack bound

.CODE
; This routine is called in place of setup386(). You pass it the address of
; a void far function that you want invoked on a breakpoint.
; It's operation is identical to setup386() except for:
;
;       1) The interrupt 1 vector is set to cintl_386() (see below)
;       2) The address passed is stored in location CCALL
;       3) DS and ES are stored in ds_save and es_save

_csetup386 proc
        push bp
        mov bp,sp
        push es
        mov ax,es
        mov es_save,ax
        mov ax,ds
        mov ds_save,ax
        mov ax,3501H
        int 21h
        mov ax,es
        mov oldsegment,ax
        mov oldoffset,bx
        pop es
        mov ax,arg2
        push ds
        mov dx,arg1
        mov c_seg,ax
        mov c_off,dx
        mov ax,seg _cintl_386
        mov ds,ax
        mov dx,offset _cintl_386
        mov ax,2501H
        int 21H
        pop ds
        xor eax,eax
        mov dr6,eax
        pop bp
        ret
_csetup386 endp

;*****************************************************************************
;*                                                                           *
;* Here is the interrupt handler!!!                                          *
;* Two arguments are passed to C, a far pointer to the base of the stack     *
;* frame and the complete contents of dr6 as a long unsigned int.            *
;*                                                                           *
;* The stack frame is as follows:                                            *
;*                                                                           *
;*    .                                                                      *
;*    .                                                                      *
;*   (Interrupted code's stack)                                              *
;*   FLAGS                                                                   *
;*   CS                                                                      *
;*   IP <-----+                                                              *
;*   AX       |                                                              *
;*   CX       |                                                              *
;*   DX       |                                                              *
;*   BX       |                                                              *
;*   SP ------+     (Stack pointer points to IP above)                       *
;*   BP                                                                      *
;*   SI                                                                      *
;*   DI                                                                      *
;*   ES                                                                      *
;*   DS                                                                      *
;*   SS <------ pointer passed to your routine points here                   *
;*                                                                           *
;* The pointer is two way. That is, you can read the values or set any of    *
;* them except SS. You should, however, refrain from changing CS,IP,or SP.   *
;*                                                                           *
;*****************************************************************************

_cintl_386 proc
        pusha                           ; save registers
        push es
        push ds
        push ss
        mov ax,@data                    ; point at our data segment
        mov ds,ax
        mov ax,ss
        mov ss_save,ax                  ; remember old stack location
        mov sp_save,sp
        cld
        lss sp,newsp                    ; switch stacks
        mov ax,STKHQQ                   ; save old end of stack
        mov oldstkhqq,ax
        mov ax,offset newstack          ; load new end of stack
        mov STKHQQ,ax
        sti
        mov eax,dr6                     ; put DR6 on stack for C
        push eax
        push ss_save                    ; put far pointer to stack frame
        push sp_save                    ; on new stack for C
        mov ax,es_save                  ; restore es/ds from csetup386()
        mov es,ax
        mov ax,ds_save
        mov ds,ax
        call ccall                      ; call the C program
        xor eax,eax                     ; clear DR6
        mov dr6,eax
        mov ax,@data
        mov ds,ax                       ; regain access to data
        lss sp,oldstack                 ; restore old stack
        add sp,2                        ; don't pop off SS
                                        ; (in case user changed it)
        mov ax,oldstkhqq                ; restore end of stack
        mov STKHQQ,ax
        pop ds
        pop es
        popa

; This seems complicated at first.
; You MUST insure that RF is set before continuing. If RF is not set
; you will just cause a breakpoint immediately!
; In protected mode, this is handled automatically. In real mode it
; isn't since RF is in the high 16 bits of the flags register.
; Essentially we have to convert the stack from:
;
;       16 bit Flags                32 bit flags (top word = 1 to set RF)
;       16 bit CS       to ----->   32 bit CS    (garbage in top 16 bits)
;       16 bit IP                   32 bit IP    (top word = 0)
;
; All this so we can execute an IRETD which will change RF.

        sub esp,6                       ; make a double stack frame
        xchg ax,[esp+6]                 ; get ip in ax
        mov [esp],ax                    ; store it
        xor ax,ax
        mov [esp+2],ax                  ; eip = 0000:ip
        mov ax,[esp+6]
        xchg ax,[esp+8]                 ; get cs
        mov [esp+4],ax
        xor ax,ax
```

```
        mov [esp+6],ax
        mov ax,[esp+8]                  ; zero that stack word & restore ax
        xchg ax,[esp+10]                ; get flags
        mov [esp+8],ax
        mov ax,1                        ; set RF
        xchg ax,[esp+10]
        iretd                           ; DOUBLE IRET (32 bits!)
_cint1_386 endp
        end
```

End Listing Seven

Listing Eight

```
/*****************************************************************************
**
* File: CBRKDEMO.C
* Example C interrupt handler for use with CBRK386
* Williams - June, 1989
* Compile with: CL CBRKDEMO.C BREAK386 CBRK386
*****************************************************************************/


#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <dos.h>
#include "break386.h"

/* functions we will reference */
int loop();
void far broke();


main()
  {
  int i;
/* declare function broke as our interrupt handler */
  csetup386(broke);
  break386(1,BP_CODE,(void far *)loop); /* set break at function loop */

  for (i=0;i<10;i++) loop(i);
  printf("Returned to main.\n");

  clear386();                           /* turn off debugging */
  }


/* This function has a breakpoint on its entry */
loop(int j)
  {
  printf("Now in loop (%d)\n",j);
  }

/*****************************************************************************
 *
 * Here is the interrupt handler!!!
 * Note it must be a far function (normal int the LARGE, HUGE & MEDIUM
 * models). Two arguments are passed: a far pointer to the base of the stack
 * frame and the complete contents of dr6 as a long unsigned int.
 *
 * The stack frame is as follows:
 *
 *     .
 *     .
 *    (Interrupted code's stack)
 *    FLAGS
 *    CS
 *    IP <---+
 *    AX     |
 *    CX     |
 *    DX     |
 *    BX     |
 *    SP ----+      (Stack pointer points to IP above)
 *    BP
 *    SI
 *    DI
 *    ES
 *    DS
 *    SS <------ pointer passed to your routine points here
 *
 * The pointer is two way. That is, you can read the values or set any of
 * them except SS. You should, however, refrain from changing CS,IP,or SP.
 *
 *****************************************************************************/

void far broke(void far *p,long dr6)
  {
  static int breaking=1;  /* don't do anything if breaking=0 */
  int c;
  if (breaking)
    {
    int n;
    int far *ip;
/*****************************************************************************
 * Here we will read the local variable off the interrupted program's stack!
 * Assuming small model, the stack above our stack frame looks like this:
 *       i   -  variable sent to loop
 *       add -  address to return to main with
 *     <our stack frame starts here>
 *
 * This makes i the 15th word on the stack (16th on models with far code)
 *
*****************************************************************************/
#define IOFFSET 15    /* use 16 for large, medium or huge models */
    n=*((unsigned int far *)p+IOFFSET);
    printf("\nBreakpoint reached! (DR6=%lX i=%d)\n",dr6,n);
/* Ask user what to do. */
    do {
       printf("<C>ontinue, <M>odify i, <A>bort, or <N>o breakpoint? ");
       c=getche();
       putch('\r');
       putch('\n');                     /* start a new line */
       if (!c)                          /* function key pressed */
          {
          getch();
          continue;
          }
       c=toupper(c);
/* Modify loop's copy of i (doesn't change main's i) */
       if (c=='M')
          {
          int newi;
          printf("Enter new value for i: ");
          scanf("%d",&newi);
          *((unsigned int far *)p+IOFFSET)=newi;
          continue;
          }
       if (c=='A')                      /* Exiting */
         {
         clear386();                    /* ALWAYS turn off debugging!!! */
         exit(0);
         }
       if (c=='N')
         breaking=0;  /* We could have turned off breakpoints instead */
       } while (c!='A'&&c!='N'&&c!='C');
    }
  }
```

End Listings

Listing One *(Text begins on page 58.)*

```
/* segments.c */

#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include "segments.h"
#include "segtable.h"

int szAppNameLength = 8;
char *szAppName = "Segments";
char *szClocks = "Too many clocks or timers!";
char *szOutOfMemory = "Not enough memory.";

#define MAX_VARIABLE_PSEGS (MAXPSEGS - MINPSEGS - 1)

typedef struct data {
        PSEG    pseg;
        SEG             lastseg;
        SEG             oldseg;
        short   changed;
} DATA, FAR * DATAP;

PSEG    psegdata;
#define FARDATAP ( (DATAP)FARPTR(0, *psegdata) )

short   xchar;
short   ychar;
BOOL    random_action = TRUE;
int             action_count = 0;
HWND    hWindow;

PSEG allocate(LONG size, char *string);
BOOL reallocate(PSEG pseg, LONG size, char *string);
LONG FAR PASCAL SegmentsWndProc(HWND, unsigned, WORD, LONG);
int FAR PASCAL timer_routine(HWND hwnd, unsigned message, short id, LONG time);
IFP strcpyifp(IFP string1, IFP string2);
int strlenifp(IFP string);

int FAR PASCAL timer_routine(HWND hwnd, unsigned message, short id, LONG time)
{
/* Randomly allocate/free a segment in the Segment Table or
monitor the Segment Table for movement.  Update the line in the window
that changes.
*/
        int             i;
        LONG    size;
        char    buffer[40];
        RECT    rect;
        int             random_switch;
message;
id;
time;

        if (random_action)
        {
                if (++action_count < 10)
                        return(0);

                action_count = 0;

                i = rand() % MAX_VARIABLE_PSEGS;

                size = (LONG)rand();    /* 0 <= size <= 32767 */
                sprintf(buffer, "  %d bytes", (short)size);
                random_switch = rand();

                if (FARDATAP[i].pseg)
                {
                        if (random_switch > 2*32767/4)
                        {
                          if (FARDATAP[i].lastseg == 0) /* if data is free */
                              FARDATAP[i].changed = -1; /* reset the count */
                          buffer[0] = 'R';
                          reallocate(FARDATAP[i].pseg, size, buffer);
                        }
                        else if (random_switch > 1*32767/4)
                        {
                                SegmentFree(FARDATAP[i].pseg);
                                FARDATAP[i].pseg = 0;
                        }
                        else if (*FARDATAP[i].pseg)
                                DataFree(FARDATAP[i].pseg);
                }
                else
                {
                        buffer[0] = 'A';
                        FARDATAP[i].pseg = allocate(size, buffer);
                        FARDATAP[i].changed = -1;
                }

                SetRect(&rect, 9*xchar, (i+2)*ychar, 46*xchar, (i+3)*ychar);
                InvalidateRect(hwnd, &rect, TRUE);
        }

        for (i = 0; i < MAX_VARIABLE_PSEGS; i++)
        {
                if (FARDATAP[i].lastseg != *FARDATAP[i].pseg)
                {
                 FARDATAP[i].oldseg = FARDATAP[i].lastseg;
                 FARDATAP[i].lastseg = *FARDATAP[i].pseg;
                 FARDATAP[i].changed++;
                 SetRect(&rect, 9*xchar, (i+2)*ychar, 46*xchar, (i+3)*ychar);
                 InvalidateRect(hwnd, &rect, TRUE);
                }
        }
        return(0);
}

void SegmentsPaint(HDC hDC)
{
        char    buffer[100];
        short   len;
        int             i;

        TextOut(hDC, 9*xchar, ychar, "pseg  seg  oldseg moved", 23);
        for (i = 0; i < MAX_VARIABLE_PSEGS; i++)
        {
           len = sprintf(buffer, "data[%d] %.4X  %.4X", i, FARDATAP[i].pseg,

*FARDATAP[i].pseg);
                TextOut(hDC, xchar, (i+2)*ychar, buffer, len);
                    if (FARDATAP[i].pseg)
                    {
                       if (*FARDATAP[i].pseg == 0)
                       TextOut(hDC, 31*xchar, (i+2)*ychar, "Data Free", 9);
                       else
                       {
                       len = sprintf(buffer, "%.4X  %.2X", FARDATAP[i].oldseg,
                                                FARDATAP[i].changed);
                          TextOut(hDC, 21*xchar, (i+2)*ychar, buffer, len);
                                  strcpyifp(MAKEIFP(buffer, &segDgroup),
                                                  MAKEIFP(0, FARDATAP[i].pseg));
                                  len = strlenifp(MAKEIFP(buffer, &segDgroup));
                          TextOut(hDC, 31*xchar, (i+2)*ychar, buffer, len);
                       }
                    }
                    else
                    TextOut(hDC, 31*xchar, (i+2)*ychar, "Free", 4);
        }

 }

IFP strcpyifp(IFP string1, IFP string2)
{
        char FAR *str1;
        char FAR *str2;

        str1 = IFP2PTR(string1);
        str2 = IFP2PTR(string2);

        while (1)
        {
                *str1++ = *str2;
                if (*str2 == 0)
                        break;
                str2++;
        }
        return(string1);
}

int strlenifp(IFP string)
{
        char FAR *str;
        int     len;

        str = IFP2PTR(string);

        for (len = 0; str[len] != 0; len++)
                ;
        return(len);
}

BOOL SegmentsInit(HANDLE hInstance)
{
    WNDCLASS    SegmentsClass;

    SegmentsClass.hCursor        = LoadCursor(NULL, IDC_ARROW);
    SegmentsClass.hIcon          = LoadIcon(hInstance,

 MAKEINTRESOURCE(SEGTABLEICON));
    SegmentsClass.lpszMenuName   = "segmentsmenu";
    SegmentsClass.lpszClassName  = szAppName;
    SegmentsClass.hbrBackground  = (HBRUSH)GetStockObject(WHITE_BRUSH);
    SegmentsClass.hInstance      = hInstance;
    SegmentsClass.style          = CS_HREDRAW | CS_VREDRAW;
    SegmentsClass.lpfnWndProc    = SegmentsWndProc;

    if (!RegisterClass((LPWNDCLASS)&SegmentsClass))
        return FALSE;

    return TRUE;
 }

 PSEG allocate(LONG size, char *string)
 {
 /*
 Allocate 'size' bytes from the global heap.  Copy a null terminated
 'string' into the allocated memory.
*/
        PSEG    pseg;
        char FAR *farptr;
        int             i;

        if (!(pseg = SegmentAlloc(size)))
        return NULL;
        farptr = FARPTR(0, *pseg);
        for (i = 0; string[i] && i < (int)size-1; i++)
                farptr[i] = string[i];
        farptr[i] = 0;
        return pseg;
}

BOOL reallocate(PSEG pseg, LONG size, char *string)
{
/*
Allocate 'size' bytes from the global heap.  Copy a null terminated string
'string' into the allocated memory.
*/
        char FAR *farptr;
        int             i;

        if (!(SegmentRealloc(pseg, size)))
        return FALSE;
        farptr = FARPTR(0, *pseg);
        for (i = 0; string[i] && i < (int)size-1; i++)
                farptr[i] = string[i];
        farptr[i] = 0;
        return TRUE;
}
```

(continued on page 108)

Listing One *(Listing continued, text begins on page 58.)*

```
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR lpszCmdLine,
                int cmdShow)
{
    MSG   msg;
    HWND  hWnd;
        int   i;
        TEXTMETRIC tm;
        HDC     hdc;
        FARPROC lpprocTimer;
        DATAP   datap;
lpszCmdLine;

    if (!hPrevInstance)
        if (!SegmentsInit(hInstance))
            return FALSE;

        SegmentInit();
        if (!(psegdata =
SegmentAlloc((DWORD)sizeof(DATA)*MAX_VARIABLE_PSEGS)))
        {
                MessageBox(hWnd, szOutOfMemory, szAppName, MB_OK|;
                return FALSE;
        }

        datap = FARPTR(0, *psegdata);
        for (i = 0; i < MAX_VARIABLE_PSEGS; i++)
        {
                datap[i].lastseg = 0;
                datap[i].pseg = 0;
        }

        hdc = CreateIC("DISPLAY", NULL, NULL, NULL);
        GetTextMetrics(hdc, &tm);
        xchar = tm.tmAveCharWidth;
        ychar = tm.tmHeight;
        DeleteDC(hdc);

      hWindow = hWnd = CreateWindow(szAppName, szAppName, WS_TILEDWINDOW, 0,0,
                46*xchar, 14*ychar, NULL, NULL, hInstance, NULL);

    lpprocTimer = MakeProcInstance(timer_routine, hInstance);
        while (!SetTimer(hWnd, 1, 100, lpprocTimer))
        {
                if (IDCANCEL == MessageBox(hWnd, szClocks, szAppName,
                        MB_ICONEXCLAMATION : MB_RETRYCANCEL))
        return FALSE;
        }
    ShowWindow(hWnd, cmdShow);
    UpdateWindow(hWnd);

    while (GetMessage(&msg, NULL, 0, 0))
        {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
        return (int)msg.wParam;
}

LONG FAR PASCAL SegmentsWndProc(HWND hWnd, unsigned message, WORD wParam,
                LONG lParam)
{
    PAINTSTRUCT ps;

    switch (message)
    {
                case WM_COMMAND:
                        switch (wParam)
                        {
                                case MENU_START:
                                        random_action = TRUE;
                                        break;
                                case MENU_STOP:
                                        random_action = FALSE;
                                        break;
                                default:
                                        break;
                        }
                    break;

            case WM_DESTROY:
                        KillTimer(hWnd, 1);
                    PostQuitMessage(0);
                    break;

            case WM_PAINT:
                    BeginPaint(hWnd, &ps);
                    SegmentsPaint(ps.hdc);
                    EndPaint(hWnd, &ps);
                    break;

            default:
                    return DefWindowProc(hWnd, message, wParam, lParam);
                    break;
    }
    return(0L);
}
```

End Listing One

Listing Two

```
/* segments.h */

#define SEGTABLEICON 1
#define MENU_START   50
#define MENU_STOP    51
```

End Listing Two

Listing Three

```
# segments.mak

cp=cl -d -DDEBUG -c -W2 -DLINT_ARGS -AM -Gswc -Os -Zdpi

.c.obj:
    $(cp) $*.c >$*.err
     type $*.err

segtable.obj: segtable.c segtable.h

segments.obj: segments.c segments.h segtable.h

segments.res: segments.rc segments.ico segments.h
    rc -r segments.rc

segments.exe: segments.obj segments.res segments.def segtable.obj
  link4 /linenumbers/co segments segtable,/align:16,/map,mlibw/noe,segments.def
  mapsym segments
  rc segments.res
```

End Listing Three

Listing Four

```
/* segments.rc */

#include "segments.h"

SEGTABLEICON  ICON    segments.ico

segmentsmenu MENU
BEGIN
    MENUITEM "Start!", MENU_START
    MENUITEM "Stop!", MENU_STOP
END
```

End Listing Four

Listing Five

```
; segments.def

NAME    Segments

DESCRIPTION 'Segments'

STUB    'WINSTUB.EXE'

CODE    MOVEABLE
DATA    MOVEABLE MULTIPLE

HEAPSIZE  10000
STACKSIZE 4096

EXPORTS
    SegmentsWndProc @1
    timer_routine @2
```

End Listings

Listing One *(Text begins on page 66.)*

```
                page    62, 132
;
;*****************************************************************************
; OBJECTS.ASM  -- This program demonstrates object-oriented programming
; techniques in 8086 assembly language.
;
dseg            segment byte public 'data'
;
; Unsigned Data Type:
Unsigned        struc
Value           dw      0
_Get_           dd      ?       ;AX = This
_Put_           dd      ?       ;This = AX
_Add_           dd      ?       ;AX = AX + This
_Sub_           dd      ?       ;AX = AX - This
_Eq_            dd      ?       ;Zero flag = AX == This
_Lt_            dd      ?       ;Zero flag = AX < This
Unsigned        ends
; UVar lets you (easily) declare an unsigned variable.
UVar            macro   var
var             Unsigned <,uGet,uPut,uAdd,uSub,uEq,uLt>
                endm
; Signed Data Type:
Signed          struc
                dw      0
                dd      ?       ;Get method
                dd      ?       ;Put method
                dd      ?       ;Add method
                dd      ?       ;Sub method
                dd      ?       ;Eq method
                dd      ?       ;Lt method
Signed          ends
; SVar lets you easily declare a signed variable.
SVar            macro   var
var             Signed  <,sGet, sPut, sAdd, sSub, sEq, sLt>
                endm
; BCD Data Type:
BCD             struc
                dw      0       ;Value
                dd      ?       ;Get method
                dd      ?       ;Put method
                dd      ?       ;Add method
                dd      ?       ;Subtract method
                dd      ?       ;Eq method
                dd      ?       ;Lt method
BCD             ends
; BCDVar lets you (easily) declare a BCD variable.
BCDVar          macro   var
var             BCD     <,bGet, bPut, bAdd, bSub, bEq, bLt>
                endm
; Declare variables of the appropriate types (For the sample pgm below):
; Also declare a set of DWORD values which point at each of the variables.
; This provides a simple mechanism for obtaining the address of an object.
                UVar    u1
U1Adr           dd      U1              ;Provide convenient address for U1.
                UVar    u2
U2Adr           dd      U2              ;Ditto for other variables.
                SVar    s1
S1Adr           dd      s1
                SVar    s2
S2Adr           dd      s2
                BCDVar  b1
B1Adr           dd      b1
                BCDVar  b2
B2Adr           dd      b2
;
; Generic Pointer Variables:
Generic1        dd      ?
Generic2        dd      ?
;
dseg            ends
;
cseg            segment byte public 'CODE'
                assume  cs:cseg, ds:dseg, es:dseg, ss:sseg
;
_This           equ     es:[bx]         ;Provide a mnemonic name for THIS.
;
; Macros to simplify calling the various methods
_Get            macro
                call    _This._Get_
                endm
_Put            macro
                call    _This._Put_
                endm
_Add            macro
                call    _This._Add_
                endm
_Sub            macro
                call    _This._Sub_
                endm
_Eq             macro
                call    _This._Eq_
                endm
_Lt             macro
                call    _This._Lt_
                endm
;*****************************************************************************
; Methods for the unsigned data type:
uGet            proc    far
                mov     ax, _This
                ret
uGet            endp
uPut            proc    far
                mov     _This,ax
                ret
uPut            endp
uAdd            proc    far
                add     ax, _This
                ret
uAdd            endp
uSub            proc    far
                sub     ax, _This
                ret
uSub            endp
uEq             proc    far
                cmp     ax, _This
                ret
uEq             endp
uLt             proc    far
                cmp     ax, _This
                jb      uIsLt
                cmp     ax, 0           ;Force Z flag to zero.
                jne     uLtRtn
                cmp     ax, 1
uLtRtn:         ret
uIsLt:          cmp     ax, ax          ;Force Z flag to one.
                ret
uLt             endp
;*****************************************************************************
; Methods for the unsigned data type.
sPut            equ     uPut            ;Same code, why duplicate it?
sGet            equ     uGet
sAdd            equ     uAdd
sSub            equ     uSub
sEq             equ     uEq
;
sLt             proc    far
                cmp     ax, _This
                jl      sIsLt
                cmp     ax, 0           ;Force Z flag to zero.
                jne     sLtRtn
                cmp     ax, 1
sLtRtn:         ret
sIsLt:          cmp     ax, ax          ;Force Z flag to one.
                ret
sLt             endp
;*****************************************************************************
; Methods for the BCD data type
bGet            equ     uGet            ;Same code, don't duplicate it.
bPut            equ     uPut
bEq             equ     uEq
bLt             equ     uLt
;
bAdd            proc    far
                add     ax, _This
                daa
                ret
bAdd            endp
bSub            proc    far
                sub     ax, _This
                das
                ret
bSub            endp
;*****************************************************************************
```

```
; Test code for this program:
TestSample      proc    near
                push    ax
                push    bx
                push    es
  Compute "Generic1 = Generic1 + Generic2;"
                les     bx, Generic1
                _Get
                les     bx, Generic2
                _Add
                les     bx, Generic1
                _Put

                pop     es
                pop     bx
                pop     ax
                ret
TestSample      endp
; Main driver program
MainPgm         proc    far
                mov     ax, dseg
                mov     ds, ax
; Initialize the objects:
; u1 = 39876.  Also initialize Generic1 to point at u1 for later use.
                les     bx, U1Adr
                mov     ax, 39876
                _Put
                mov     word ptr Generic1, bx
                mov     word ptr Generic1+2, es
; u2 = 45677.  Also point Generic2 at u2 for later use.
                les     bx, U2Adr
                mov     ax, 45677
                _Put
                mov     word ptr Generic2, bx
                mov     word ptr Generic2+2, es

; s1 = -5.
                les     bx, S1Adr
                mov     ax, -5
                _Put
; s2 = 12345.
                les     bx, S2Adr
                mov     ax, 12345
                _Put
; b1 = 2899.
                les     bx, B1Adr
                mov     ax, 2899h
                _Put
; b2 = 195.
                les     bx, B2Adr
                mov     ax, 195h
                _Put
; Call TestSample to add u1 & u2.
                call    TestSample
; Call TestSample to add s1 & s2.
                les     bx, S1Adr
                mov     word ptr Generic1, bx
                mov     word ptr Generic1+2, es
                les     bx, S2Adr
                mov     word ptr Generic2, bx
                mov     word ptr Generic2+2, es
                call    TestSample
; Call TestSample to add b1 & b2.
                les     bx, B1Adr
                mov     word ptr Generic1, bx
                mov     word ptr Generic1+2, es
                les     bx, B2Adr
                mov     word ptr Generic2, bx
                mov     word ptr Generic2+2, es
                call    TestSample

                mov     ah, 4ch                 ;Terminate process DOS cmd.
                int     21h
;
MainPgm         endp
cseg            ends
;
sseg            segment byte stack 'stack'
stk             dw      0f0h dup (?)
endstk          dw      ?
sseg            ends
;
                end     MainPgm
```

End Listing

Listing One *(Text begins on page 74.)*

```
$ PRIMES.SET
$ ISETL program to find number of primes <= n, using set notation

size := 1000 ;
sqrt_size := fix(sqrt(size)) ;
composites := {i*j : i in {3,5..sqrt_size}, j in {i..size div i}} ;
primes := {2} + {3,5..size} - composites ;
print size ;
print #primes ;
```

End Listing One

Listing Two

```
$ PRIMES.TUP
$ ISETL program to find number of primes <= n, using ordered tuples

$ tuple difference operator
diff := func(t1, t2);
    return [i : i in t1 | i notin t2 ] ;
end;

size := 1000 ;
sqrt_size := fix(sqrt(size)) ;
composites := [i*j : i in [3,5..sqrt_size], j in [i..size div i]] ;
primes := [2] + [3,5..size] .diff composites ;
print size ;
print #primes ;
```

End Listing Two

Listing Three

```
$ FIB.TUP
$ ISETL program to find Fibonacci numbers, using dynamic programming

$ uses log(): only accurate up to 308 digits
digits := func(x);
    if (x = 0) then return 1 ;
    else return 1 + floor(log(abs(x))) ;
    end;
end;

$ use "dynamic programming" to assign to fib()
fib := func(x);
    fib(x) := fib(x-1) + fib(x-2) ;
    return fib(x) ;
end;

fib(0) := 1 ;
fib(1) := 1 ;

fibonacci := [fib(x) : x in [1 .. 1000 ] ] ;
print fibonacci(1000) ;
print digits(fibonacci(1000)) ;
```

End Listings

Listing One *(Text begins on page 84.)*

```
;* Module description * This module takes care of error trapping. The scheme
;used records the trapping routine stack pointer so that an error can cause
;the stack to return to a consistent state. This module was written using
;Borland's Turbo Assembler 2.0.

;** Environment **
.model small    ;Set up for SMALL model.
locals          ;Enable local symbols.

;** Macros **
;<<Generate correct return based on model>>
procret macro
if @codesize
        retf
else
        retn
endif
endm

;** Public operations **
public pascal ERROR_INIT        ;Initialize error handler.
public pascal ERROR_TRAP        ;Set up error trap.
public pascal ERROR_LOG         ;Log error.

;** Uninitialized data **
.data?
errstk  dw ?    ;SP at last error log (-1 if none).

;** Code **
.code
;Set up DS to nothing since that is the typical arrangement.
assume ds:nothing

;[Initialize error manager]
error_init proc pascal          ;Declare proc with PASCAL calling conventions.
        mov errstk,-1
        ret
endp

;[Set up error trap]
;This procedure preserves the previous ERRSTK, sets up a new ERRSTK, and
;calls the passed procedure.  On exit, the previous ERRSTK is restored.
error_trap proc pascal          ;Pascal calling conventions.
arg @@proc:codeptr              ;Only argument is procedure to call.
uses ds,si,es,di                ;Force a save of all registers C cares for.
        push errstk
        ;Call internal routine to record return address on stack.
        call @@rtn
        pop errstk
        ret
@@rtn   label proc
        mov errstk,sp           ;Save SP so we can restore it later.
        call @@proc pascal      ;Call procedure.
        xor ax,ax               ;Return code = 0 for normal return.
        procret
endp

;[Log error]
;Control is passed to the last ERROR_TRAP, if any.
;Error code is passed and returned in AX.
error_log proc pascal
arg @@error_code:word
        cmp errstk,-1           ;Lock up if no error address.
@@1:    jz @@1
        mov ax,@@error_code
        mov sp,errstk
        procret
endp
end
```

End Listing One

Listing Two

```
;* Module description * This module manages a simple stack-based heap.
;Deallocation is not supported. NOTE: This module must be assembled with /MX
;to publish symbols in the correct case. This module is written using
;Borland's Turbo Assembler 2.0.

;** Environment **
.model small    ;Set up for SMALL model.
locals          ;Enable local symbols.

;** Equates **
err_memory = 1  ;Out of memory error number.

;** Public operations **
public pascal HEAP_INIT         ;Initialize heap.
public pascal HEAP_ALLOC        ;Allocate memory from heap.

;** External operations **

;<<Error handler>>
extrn pascal ERROR_LOG:proc     ;Long jump library procedure for errors.

;** Uninitialized data **
.data?
memptr  dw ?    ;Pointer to first free segment.
memsiz  dw ?    ;Remaining paragraphs in heap.

;** Code **
.code
;Set up DS to nothing since that is the typical arrangement.
assume ds:nothing

;[Initialize the heap]
heap_init proc pascal   ;Declare proc with PASCAL calling conventions.
arg @@start_seg:word,@@para_size:word
                        ;Arguments are starting segment and para count.
        mov ax,@@start_seg
        mov memptr,ax
        mov ax,@@para_size
        mov memsiz,ax
        ret
heap_init endp

;[Allocate memory from the heap]
heap_alloc proc pascal          ;Declare proc with PASCAL calling conventions.
arg @@para_count:word           ;Only argument is count of paragraphs.
        ;See if there is enough remaining.
        mov ax,@@para_count
        cmp memsiz,ax
        jc @@err
        sub memsiz,ax
        add ax,memptr
        xchg ax,memptr
        mov dx,ax
        xor ax,ax
        ret
@@err:  ;Out-of-memory error.
        mov ax,err_memory
        call error_log pascal,ax
        ;Never returns.
heap_alloc endp

end
```

End Listing Two

Listing Three

```
;* Module description * This module reads source files and converts them into
;words, then files the words away in a symbol table with the help of a hash
;function. This module was written using Borland's Turbo Assembler 2.0.

;** Environment **
.model small    ;Set up for SMALL model.
locals          ;Enable local symbols.

;** Equates **
;<<Error numbers>>
err_hash = 2    ;Out of hash space error number.
err_read = 3    ;Read error.

;<<Hash function>>
hash_rotate = 5 ;Amount to rotate for hash function.
hash_skip   = 11;Number of entries to skip on hash collision.

;<<Read buffer>>
rbf_size = 800h ;Size of read buffer in paragraphs.

;** Public operations **
public pascal WORD_INIT         ;Initialize hash table.
public pascal WORD_READ         ;Read file, convert to words, and hash them.
public pascal WORD_COUNT        ;Get total word count.
public pascal WORD_NAME         ;Get name of word.
public pascal WORD_REFCOUNT     ;Get reference count of word.
public pascal WORD_SCAN         ;Scan all words.
public pascal WORD_COMPREF      ;Compare word reference counts.

;** External operations **
;<<Heap>>
extrn pascal HEAP_ALLOC:proc    ;Heap allocation.

;<<Error handling>>
extrn pascal ERROR_LOG:proc     ;Trap an error.

;** Data structure **
;<<Symbol table entry>>
symtbl struc
symref  dw ?    ;Reference count.
symsiz  dw ?    ;Length of word.
ends
symnam  = size symtbl   ;Offset of start of name text.

;** Initialized data **
.data
;<<Translation character type table>>
typdlm  = 1     ;Delimiter bit.
typnum  = 2     ;Numerical digit.
typcas  = 20h   ;Lower case bit: Set if lower case letter.
xlttbl  label byte
        db '0' dup (typdlm)
        db 10 dup (typnum)
        db ('A'-1)-'9' dup (typdlm)
        db 'Z'-('A'-1) dup (0)
        db ('a'-1)-'Z' dup (typdlm)
        db 'z'-('a'-1) dup (typcas)
        db 255-'z' dup (typdlm)
;** Uninitialized data **
.data?

;<<Hash table values>>
hshptr  dw ?    ;Segment address of hash table.
hshsiz  dw ?    ;Total number of hash entries.  Must be a power of 2!
hshcnt  dw ?    ;Total free entries remaining in hash table.
hshmsk  dw ?    ;Mask for converting hash value to address.

;<<Read buffer values>>
rbfptr  dw ?    ;Segment address of read buffer.

;<<Word buffer>>
wrdbuf  db 256 dup (?)

;** Code **
.code
;Set up DS to nothing since that is the typical arrangement.
assume ds:nothing

;[Initialize hash table]
```

(Listing continued on page 118)

Listing Three *(Listing continued, text begins on page 84.)*

```
word_init proc pascal
arg @@max_word_count:word       ;Argument: Maximum number of words.
uses es,di
        ;First, allocate read buffer.
        mov ax,rbf_size
        call heap_alloc pascal,ax
        mov rbfptr,dx
        ;Now convert maximum word count to power of 2.
        mov ax,@@max_word_count
        mov cl,16+1
@@l1:   dec cl
        shl ax,1
        jnc @@l1
        mov ax,1
        shl ax,cl
        ;Initialize some hash parameters.
        mov hshsiz,ax
        mov hshcnt,ax
        dec ax
        shl ax,1
        mov hshmsk,ax
        ;Now, allocate hash table from heap.
        mov ax,hshsiz           ;Size of hash table in words.
        add ax,7
        mov cl,3
        shr ax,cl               ;Convert to paragraphs.
        call heap_alloc pascal,ax
        mov hshptr,dx
        ;Clear out hash table: 0 means 'no value'.
        mov es,dx
        xor di,di
        cld
        mov cx,hshsiz
        xor ax,ax
        rep stosw
        ret
word_init endp

;[Read file and assimilate all words]
word_read proc pascal
arg @@handle:word               ;Argument is file handle.
uses ds,si,es,di
        ;Load XLAT buffer address.  The XLAT table is used for case conversion
        ;and for character type identification.
        mov bx,offset xlttbl
@@read: ;Read next buffer while delimiter processing.
        call @@brd
        jcxz @@done
@@skip: ;Skip all delimeters, etc.
        lodsb
        xlat xlttbl
        test al,typdlm
        loopnz @@skip
        jnz @@read
        ;Adjust pointer & count.
        dec si
        inc cx
        ;If it is a number, skip to end.
        test al,typnum
        jnz @@num
        ;It is a word. We'll transfer a word at a time to the word buffer,
        ;hashing it as we go. DX will be the current hash value. CX is the
        ;amount remaining in the buffer.
        xor dx,dx
        ;Initialize output address.
        push ss
        pop es
        mov di,offset wrdbuf
@@clp:  ;Transfer.  This is THE most time-critical loop in the program.
        lodsb                   ;Read character.
        mov ah,al
        xlat xlttbl             ;Get its type.
        test al,typdlm          ;Abort if delimiter.
        jnz @@wend
        and al,typcas           ;Use case bit to convert to upper case.
        neg al
        add al,ah
        stosb                   ;Save it in word buffer.
        ;Calculate hash value.
        mov ah,cl
        mov cl,hash_rotate
        rol dx,cl
        mov cl,ah
        xor dl,al
        loop @@clp              ;Keep going until end of buffer.
        ;End of buffer while word processing.  Read more.
        call @@brd
        jcxz @@wnd2
        jmp @@clp
@@nrd:  ;Read next buffer while number processing.
        call @@brd
        jcxz @@done
@@num:  ;Numbers are not considered 'words' and should be skipped.
        ;Skip up to first delimiter.
        lodsb
        xlat xlttbl
        test al,typdlm
        loopz @@num
        jz @@nrd
        ;Adjust pointer and count.
        dec si
        inc cx
        jmp @@skip
@@done: ret
@@wend: ;End of word.  Adjust buffer pointer.
        dec si
@@wnd2: ;End of word.  Hash value is in DX, upper-case word is in WRDBUF,
        ;DI points to end of word + 1.
        push ds si cx bx        ;Save the registers we will use for this step.
        xor al,al               ;Null-terminate the word.
        stosb
        mov cx,di               ;Calculate the word's length.
        sub cx,offset wrdbuf
        mov bx,dx               ;Put the hash value in a useable register.
        shl bx,1                ;Lower bit will be discarded, so shift.
        push ss                 ;Initialize DS.
        pop ds
        assume ds:dgroup
        ;Now it is time to locate the word in the hash table if it is there,
        ;or create an entry if it is not.
@@hlp:  mov es,hshptr
        and bx,hshmsk
        mov ax,es:[bx]
        and ax,ax
        jz @@make
        ;Verify that the hash entry is the correct one.
        mov es,ax
        mov ax,cx
        cmp es:[symsiz],ax      ;Compare length of word.
        jnz @@coll
        mov si,offset wrdbuf    ;Compare actual text if that agrees.
        mov di,symnam
        repz cmpsb
        mov cx,ax
        jz @@fd
@@coll: ;Collision!  Advance to the next candidate hash entry.
        add bx,hash_skip*2
        jmp @@hlp
@@dne2: ret
@@make: ;We have encountered this word for the first time.
        ;We must create a new symbol entry of the appropriate size.
        ;First decrement remaining free hash count.
        dec hshcnt
        jz @@herr
        push cx
        push bx
        mov ax,cx               ;Calculate length of symbol descriptor.
        add ax,symnam+15
        mov cl,4
        shr ax,cl
        call heap_alloc pascal,ax
        pop bx                  ;Record symbol descriptor in hash table.
        mov es:[bx],dx
        pop cx                  ;Record length.
        mov es,dx
        mov es:[symsiz],cx
        mov di,symnam           ;Move text of word into symbol table.
        mov si,offset wrdbuf
        shr cx,1
        rep movsw
        rcl cx,1
        rep movsb
        mov es:[symref],0       ;Clear reference count.
@@fd:   ;Matching entry found!  Increment reference count.
        inc es:[symref]
@@nwd:  ;Go on to the next word in the buffer, if any.
        pop bx cx si ds
        assume ds:nothing
        jcxz @@dne2
        jmp @@skip
@@herr: ;Out of hash space error.
        mov ax,err_hash
        call error_log pascal,ax
        ;No return from ERROR_LOG.

;(Read buffer)
;Reads the next hunk of buffer.  Returns actual amount read in CX,
;DS:SI as start of data to read.
@@brd:  push dx bx
        mov cx,rbf_size*16
        mov bx,@@handle
        mov ah,3fh
        mov ds,rbfptr
        xor dx,dx
        int 21h
        jc @@err
        mov cx,ax
        xor si,si
        pop bx dx
        cld
        retn            ;Use RETN so stack frame return won't be generated.
@@err:  ;Read error.
        mov ax,err_read
        call error_log pascal,ax
        ;No return is needed because ERROR_LOG never returns.
word_read endp

;[Get total word count]
word_count proc pascal
        mov ax,hshsiz           ;Load total word capacity.
        sub ax,hshcnt           ;Subtract actual remaining free words.
        ret
word_count endp

;[Get address of name of word]
word_name proc pascal
arg @@word_desc:word            ;Argument is word descriptor.
        mov dx,@@word_desc
        mov ax,symnam
        ret
word_name endp

;[Get refcount for word]
word_refcount proc pascal
arg @@word_desc:word            ;Argument is word descriptor.
uses ds
        mov ds,@@word_desc
        mov ax,ds:[symref]
        ret
word_refcount endp
```

```
;[Scan all words]
word_scan proc pascal
arg @@scan_proc:codeptr         ;Argument is procedure to call for each word.
uses ds,si
        mov ds,hshptr
        xor si,si
        mov cx,hshsiz
        cld
@@l1:   lodsw
        and ax,ax
        jnz @@take
@@next: loop @@l1
        ret
@@take: push cx ds
        push ss
        pop ds
        call @@scan_proc pascal,ax
        pop ds cx
        cld
        jmp @@next
word_scan endp

;[Compare reference counts for two word descriptors]
word_compref proc pascal
arg @@word_desc1:word,@@word_desc2:word
uses ds
        mov ds,@@word_desc2
        mov ax,ds:[symref]
        mov ds,@@word_desc1
        sub ax,ds:[symref]
       ret
word_compref endp
end
```

End Listing Three

Listing Four

```
;* Module description * This module contains the sort routine for SPECTRUM.
;This module was written using Borland's Turbo Assembler 2.0.

;** Environment **
.model small    ;Set up for SMALL model.
locals          ;Enable local symbols.

;** Public operations **
public pascal SORT_DO           ;Perform sort.

;** Code **
.code
;Set up DS to nothing since that is the typical arrangement.
assume ds:nothing

;[Sort procedure]
sort_do proc pascal
arg @@array:dword,@@count:word,@@compare_proc:codeptr
uses ds,si,di

        ;First load up registers for internal recursion.  DS:SI will be
        ;the current sort array address, CX the count of elements to sort.
        lds si,@@array
        mov cx,@@count
        call @@sort
        ret

;Internally recursive sort routine. This routine accepts DS:SI as the sort
;array address, and CX as the count of elements to sort.
@@sort: cmp cx,2
        jnc @@go
        retn
@@go:   ;Save all registers we will change.
        ;Internally, DI and DX will be start and count of second merge area.
        push si cx di dx
        ;Divide into two parts and sort each one.
        mov dx,cx
        shr cx,1
        sub dx,cx
        call @@sort
        mov di,si
        add di,cx
        add di,cx
        xchg si,di
        xchg cx,dx
        call @@sort
        xchg cx,dx
        xchg si,di
        ;Now, merge the two areas in place.
        ;Each area must be at least size 1.
@@mrgl: ;Compare - DS:DI - DS:SI.
        call @@compare_proc pascal,ds:[di],ds:[si]
;;The following commented-out sequence is the code that would be required
;;if strict Pascal calling conventions were adhered to for calling
;;COMPARE_PROC.  You can see how much extra work this is!!
;;      push cx dx
;;      push ds
;;      mov ax,ds:[di]
;;      mov bx,ds:[si]
;;      push ss
;;      pop ds
;;      call @@compare_proc pascal,ax,bx
;;      pop ds
;;      pop dx cx
;;      and ax,ax
        jns @@ok
        ;Slide up first merge area using starting value from DI.
        mov ax,ds:[di]
```

(Listing continued on page 120)

Listing Four *(Listing continued, text begins on page 84.)*

```
        push si cx
@@sllp: xchg ax,ds:[si]
        add si,2
        loop @@sllp
        xchg ax,ds:[si]
        pop cx si
        add si,2
        add di,2
        dec dx
        jnz @@mrgl
        jmp short @@exi
@@ok:   ;Correct so far.  Advance SI.
        add si,2
        loop @@mrgl
@@exi:  ;Restore registers.
        pop dx di cx si
        retn
sort_do endp

end
```

End Listing Four

Listing Five

```
/***** File: SPECTRUM.C *****/
/* This C module is written using Borland's Turbo C 2.0 and can be
   compiled using the default switches. It should be linked with the file
   WILDARGS.OBJ from the Turbo C examples directory to enable the wild card
   file name expansion facility. Without WILDARGS, SPECTRUM will still work
   but will not be capable of expanding file names with wild cards.

   The following is an example make file, where TA is the assembler name, TCC
   is the C compiler name, TLINK is the linker name, \TC\LIB contains the C
   libraries, and \TC\EXA contains the Turbo C examples:

spectrum.exe: spectrum.obj heap.obj word.obj error.obj sort.obj
   tlink \tc\lib\c0s+\tc\exa\wildargs+spectrum+heap+word+error+sort,spectrum,,

\tc\lib\cs.lib;
heap.obj: heap.asm
        ta heap /mx;
word.obj: word.asm
        ta word /mx;
error.obj: error.asm
        ta error /mx;
sort.obj: sort.asm
        ta sort /mx;
spectrum.obj: spectrum.c
        tcc -c spectrum
*/

/*** Header Files ***/
#include <dos.h>
#include <stdio.h>
#include <fcntl.h>

/*** Function Protypes ***/
/* Used Locally */
int allocmem( unsigned, unsigned * );
int freemem ( unsigned );
int _open( const char *, int oflags );
int _close( int );
/* Error trapper */
extern void pascal error_init (void);
extern unsigned pascal error_trap (void pascal (*execution_procedure)() );
extern void pascal error_log (unsigned error_code);
/* Heap */
extern void pascal heap_init (unsigned starting_segment,
                                        unsigned segment_count);
extern void far * pascal heap_alloc (unsigned paragraph_count);
/* Symbol table */
extern void pascal word_init (unsigned maximum_word_count);
extern void pascal word_read (unsigned file_handle);
extern void pascal word_scan (void pascal (*word_procedure)() );
extern char far * pascal word_name (unsigned word_descriptor);
extern unsigned pascal word_refcount (unsigned word_descriptor);
extern unsigned pascal word_count (void);
extern int pascal word_compref (unsigned word_desc1, unsigned word_desc2);
/* Sorting procedure */
extern void pascal sort_do (unsigned far *sort_array, unsigned sort_count,
        int pascal (*compare_procedure)() );

/*** Global Variables ***/
/* Error table */
char * error_table [] = {
"Insufficient Memory\n",
"Out of Hash Space\n",
"File Read Error\n",
"Usage: SPECTRUM filespec [filespec] ... [filespec]\n(filespec may have ?,*)\n"
};

/* Arguments */
int global_argc;
char **global_argv;

/* Memory */
unsigned segment_count;
unsigned starting_segment;

/* Sort array */
unsigned sort_index;
unsigned far *sort_array;

/**** Procedures ****/
/* Fill sort array with descriptors */
```

```
void pascal array_fill(unsigned word_desc)
{
    sort_array[sort_index++] = word_desc;
}

/* Main execution procedure */
void pascal main2 (void)
{
    int i;
    unsigned j;
    int words = 0;
    int file_handle;
    if( global_argc < 2 ) {
        error_log(4);
        }
    heap_init (starting_segment, segment_count);
    word_init (32767);
    for( i=1 ; i<global_argc ; i++ ) {
        file_handle = _open (global_argv[i], O_RDONLY);
        if (file_handle != -1 ) {
            word_read( file_handle);
            _close( file_handle );
            } else {
            error_log(3);
            }
        }

    /* Obtain array address */
    sort_array = (unsigned far *)heap_alloc((word_count()+7)/8);
    /* Fill array */
    sort_index = 0;
    word_scan(array_fill);
    /* Sort array */
    printf ("Sorting...\n");
    sort_do (sort_array, sort_index, word_compref);

    /* Display output */
    printf ("\nCount\tWord\n");
    printf ("-----\t----\n");
    for (i=0 ; i<sort_index-1 ; i++) {
        j = word_refcount(sort_array[i]);
        words = words + j;
        printf ("%d",j);
        printf ("\t");
        printf ("%Fs",word_name(sort_array[i]));
        printf ("\n");
        }
    printf ("\nTotal unique words:\t%d\n",sort_index);
    printf ("Total words:\t\t%d\n",words);
}

/* Main procedure */
int main( int argc, char *argv[] )
{
    int i;
    /* Copy arguments */
    global_argc = argc;
    global_argv = argv;
    error_init();
    segment_count = allocmem(65535,&starting_segment);
    allocmem( segment_count, &starting_segment );
    i = error_trap ( main2 );
    if (i != 0) {
        /* Print error message */
        printf (error_table[i-1]);
        }
    freemem (starting_segment);
    return (i);
}
```

End Listing Five

Listing Six

```
spectrum.exe: spectrum.obj heap.obj word.obj error.obj sort.obj
  tlink /v \tc\lib\c0s+\tc\exa\wildargs+spectrum+heap+word+error+sort,

spectrum,,\tc\lib\cs.lib;
heap.obj: heap.asm
        ta heap /mx /zi
word.obj: word.asm
        ta word /mx /zi
error.obj: error.asm
        ta error /mx /zi
sort.obj: sort.asm
        ta sort /mx /zi
spectrum.obj: spectrum.c
        tcc -c -v spectrum
```

End Listings

Getting CLOS

Michael Swaine

This column's beat is paradigms, and it seemed worthwhile to take a look at how one paradigm (functional programming) is extended to another (object-oriented programming). In January we looked at "pure" Lisp; in February we saw how this pure functional paradigm has evolved with the widespread acceptance of Common Lisp, and this month we'll take a look at the objectification of Lisp in the form of the Common Lisp Object System. We'll examine two themes: How the Common Lisp data-type system underlies the CLOS class system, and how the basic concept of a function, a key aspect of Common Lisp as well as of "pure" Lisp, has been extended to the object world.

What makes Lisp relevant today is that it is converging, in terms of features and performance, with other development environments for large software projects. When Guy Steele published *Common Lisp: The Language* (Digital Press, 1984), he codified what quickly became the de facto standard for Lisp; now the ANSI subcommittee X3J13 has nearly completed a draft standard for Common Lisp that includes the Common Lisp Object System (CLOS), an object-oriented extension to the language. I had this column half written when the second edition of Steele's book arrived, containing much new material, including an entirely new chapter on CLOS. It forced me to go back and rewrite several things; this column also corrects some things I said last month that are now out of date. Steele's treatment of CLOS is essentially the ANSI committee's treatment, and should be very close to the final draft standard, due out this year.

This convergence, though, is turning Lisp into something new. At last year's OOPSLA meeting, Bjarne Stroustrup summed up CLOS by calling it a multi-paradigm language. The circumstances (the developer of C++ being asked to deliver a lecture on the virtues of CLOS) left it unclear whether he meant it as a term of opprobrium or as a compliment.

Typing Tutor

Some of the things I said last month have been superseded by the new edition of Steele's book, and this edition makes some things more official than they were previously. Because of these things and also because CLOS classes map into the Common Lisp hierarchy, I'll spell out the Common Lisp data type relationships in some detail.

To begin with, it's not really a hierarchy, but an overlapping structure that Rosemary Simpson, in her *Common Lisp: The Index* (Coral Software and Franz, Inc., 1987) calls a "heterarchy." Two types stand at the very top and bottom of the Common Lisp data type heterarchy. *t* is a supertype of every other type, and *nil* is a subtype of every other type. No object is of type *nil*. Every object is of type *t*.

The following subtypes of type *t* are of interest because X3J13 has defined them to be pairwise disjoint: *character*, *number*, *symbol*, *cons*, *array*, *random-state*, *hash-table*, *read-table*, *package*, *pathname*, and *stream*. A Common Lisp object cannot belong to more than one of these types, although it need not belong to any of them.

In addition to these types, any data type created by the *defstruct* or *defclass* macros (a user-defined structure or a CLOS class, respectively) is also disjoint from any of the above types. Any two user-defined structures are disjoint from one another unless defined otherwise, and the same goes for classes. Classes, though, are always defined in terms of other classes. I won't say much about structures here, and I'll discuss classes later.

Functions are data objects, too, and the data type *function* is disjoint from some of the above types, specifically from *character*, *number*, *symbol*, *cons*, and *array*. The types *character*, *number*, *symbol*, *cons*, *array*, and *function* are worthy of some elaboration.

Lisp Has Character

First, I'll discuss characters and numbers, correcting some outdated info from last month.

X3J13 redefined the character subtypes that were given in the first edition of Steele's book. Now the *base-character* and *extended-character* subtypes form an exhaustive partition of the type *character*. All characters are one or the other of these types. *Base-character* is implementation-defined, but must be a supertype of standard char, which is a set of 96 characters that any Lisp implementation must support; the *extended-character* type seems to be X3J13's way of dodging the confusion of bit and font attributes prevalent in Lisp.

Formerly, the data type *number* contained three disjoint subtypes, *rational*, *float*, and *complex*. Now a new type, *real*, has been introduced. The hierarchy runs like this: Types *real* and *complex* are disjoint subtypes of type *number*; other subtypes of type *number* can be defined. Each of these two subtypes also has two disjoint subtypes. Type *real* has the disjoint subtypes *rational* and *float*; it's possible to define other real subtypes. Type *rational* has the disjoint subtypes *integer* and *ratio*; other rational types can be defined.

However, type *integer* has exactly two subtypes, and Common Lisp does not allow other subtypes of *integer* to be defined. The two integer subtypes are *fixnum* and *bignum*. The *fixnum* data type is a conventional fixed-word-length integer, the word length being

implementation-dependent. *bignums* are "true" integers, their size dependent only on storage limits, not on word length. *fixnums* are more efficient than *bignums*, and are used where efficiency is more important than being able to represent precisely the number of grains of sand required to fill the universe. For example, *fixnum* is the required data type for array indices.

An object of type *ratio* represents the ratio of two integers. The Lisp system is required to reduce all ratios to the lowest terms, representing a ratio as an integer if that is possible.

Common Lisp defines four subtypes of type *float*, but an implementation need not have all four as distinct types. Types *short-float*, *single-float*, *double-float*, and *long-float*, in nondecreasing order of word length, all must be supplied, but any adjacent pair or triplet of these may be identical. Any *float* subtypes that are not identical must be disjoint.

An object of type *complex* represents a complex number in Cartesian form, as a pair of numbers. The two numbers must be of type *real*, and both must be rational or both must be of the same floating-point type.

Everything in Lisp is a List

Characters and numbers are straightforward data types, but symbols and lists are trickier. Symbols are named data objects. Type *symbol* includes among its subtypes one peculiar subtype: type *null*. *null* is the type of exactly one Lisp data object: the object *nil*. The status of type *null* is one reason that the type relationships of Common Lisp form a heterarchy rather than a hierarchy. *null* is a subtype of two types, neither of which is a subtype of the other: *symbol* and *list*. *nil* is the only object that is both a *list* and a *symbol*.

Actually, at another level, all symbols have a list-like structure. Each symbol has an associated data structure called a "property list," a list of pairs, the first elements being (typically) symbols, and the second elements being any Lisp data objects. The purpose of the property list of a symbol has evolved over time; in Common Lisp it is less important than in earlier Lisps, being used now for data not needed frequently, such as debugging, documentation, or compiler information. Neither a property list nor a symbol is of type *list*, but somehow everything in Lisp is a list of some sort. (Viewed another way, almost everything in Lisp is a function, as we'll see shortly.)

The data type *list*, though, is not regarded as being as basic as type *cons*.

These are alternate ways of viewing the same thing. A *list* is recursively defined to be either the object *nil* or a *cons* whose second component is a *list*. A *cons* is a data structure with two components, which can be pretty much anything; usually, though, the second component of a *cons* is a *list* (or *nil*, the empty list). The first components of the *cons*es making up a *list* are the elements of the list.

The data type *cons*, then, is the type of the basic data structure used to build lists. Any object that is a *cons* is also a *list*, so *list* is a supertype of *cons*. The data type *list* has exactly two subtypes, and they are disjoint: *cons* and *null*. In this sense, *null* is the (type of the) empty list. *list* itself is a subtype of the data type *sequence*, which has one other subtype: *vector*. *vector* and *list* are disjoint.

Vectors, and arrays generally, can be rather complex. Arrays can be complex, with the ability to share data with other arrays, be dynamically sized, and have fill pointers. An array that has none of these features is called a "simple array." Vectors are one-dimensional arrays; they differ from lists in performance characteristics. Accessing an element of a list is, on average, a linear function of list length, while the time to access an element of a vector is constant. When it comes to adding an element to the beginning of a list or vector, though, the relationship is reversed: constant for the list, and a linear function of vector length for the vector.

One of vector's more interesting subtypes is type *string*. Type *string* is the union of one or more vector types with the characteristic that the types of the vector's elements are subtypes of type character.

According to X3J13, the data type *function* is strictly disjoint from data types *cons* and *symbol*. But lists and symbols are the only tools available for referring to functions, or for invoking them. This is probably a use-mention distinction, but in any case, when a *list* or *symbol* is used in this way it is automatically coerced to type *function*. As we'll see shortly, there's some truth to the exaggeration that everything in Lisp is a function.

Lisp Has Class

CLOS is an object-oriented extension to CL, adding four kinds of objects to CL: classes, instances, generic functions, and methods. The key aspects are generic functions, multiple inheritance, declarative method combination, and a metaobject protocol. Classes and instances are tied to data types, generic functions to functions. I'll say only a little bit here about the metaobject protocol, which is not yet officially a part of CLOS.

The Common Lisp Object System maps classes into the data types just described. Many Common Lisp types have corresponding classes with the same names, but not all. Normally, a class has a corresponding type with the same name.

Because the types do not form a simple tree, and a type can be a subtype of two types neither of which is a subtype of the other, you might expect CLOS to support multiple inheritance, in which a class can inherit from more than one superclass. In fact, this is the case. The heterarchical structure of types is mirrored in the inheritance structure of classes, but CLOS requires that more structure be added to establish a clear precedence order for inheritance. For example, the class *vector* has superclasses *sequence* and *array*, just as the type *vector* has supertypes *sequence* and *array*, but from which superclass does *vector* inherit what?

CLOS resolves questions such as this by requiring that you specify an ordering of direct superclasses when you define a class (and by supplying this ordering for predefined classes). The business of deriving a full precedence order is fairly complex, but the CLOS class precedence order for predefined classes resolves such issues. In particular, the precedence order for the class *null* is *null*, *symbol*, *list*, *sequence*, *t*; and the precedence order for the class *string* is *string*, *vector*, *array*, *sequence*, *t*. By implication, the precedence order for the class vector is *vector*, *array*, *sequence*, *t*; so array methods have precedence over sequence methods when class vector is inheriting methods.

Everything in Lisp is a Function

The simplifying generalization is that everything in Lisp is a function. It's nearly true; any data object can be treated as a function, or rather, as a form. A form is simply a data object treated as a function. You treat a data object as a function when you hand it to the evaluator, which is the mechanism that executes Lisp programs. The evaluator accepts a form and does whatever computation the form specifies.

The evaluator can be implemented in various ways, such as by an interpreter that traverses the form recursively, performing the required calculations along the way; or as a pure compiler; or by some mixed form. Common Lisp requires that correct programs produce the same results, regardless of the method of implementation. The evaluator is available to the user via the function *eval*, and also the special form *eval-when*, which allows specifying that a form should be evaluated, say, only at compile time.

Not every data object specifies a meaningful function, but most do. To the evaluator, there are three kinds of forms, corresponding to three nearly disjoint data types. There are symbols, lists, and self-evaluating forms (per X3J13, all standard Common Lisp objects, except symbols and lists, are self-evaluating forms).

Self-evaluating forms are taken literally by the evaluator; they return themselves on evaluation.

Symbols name variables, constants, keywords and functions. They evaluate to whatever they name; for example, what they are bound to or what they are set to.

Lists, from the viewpoint of the evaluator, come in three varieties: special forms, macro calls, and function calls. Note that while a function is not a list, a function call is.

Special forms are structural elements of the language that don't fit the functional paradigm well, such as the if-then-else structure. These deviations from the purity of the paradigm have been a part of Lisp since the beginning, and new special forms have been added over the years, but in Common Lisp the set of special forms is fixed and cannot be extended by the programmer. A macro is a function from forms to forms, much as in other languages. A macro call, when evaluated, is said to be expanded. Programmers can extend the set of macros. Despite the fact that they are not true functions, special forms look like functions syntactically, as do

macros. The consequence of this is that when you are sitting at the keyboard typing in Lisp code, it feels like you are dealing with one kind of construct: A parenthesized list that represents a function and its arguments.

A form that is a function call consists of a list whose first element is a function name. The other elements of the list, if any, are treated by the evaluator as forms to be evaluated to provide the function with arguments. There are two levels of evaluation that take place whenever the evaluator deals with a function call: The arguments get evaluated, then the function is evaluated with these arguments. Typically, the evaluation of the function produces a value, which becomes the value of the original form.

There are two ways in which the first element of a form can name a function, one involving a symbol and the other involving a list. Because symbols are used to name functions, this is the most direct and obvious way. The other way involves the use of a lambda expression. A lambda expression is technically not a form, and cannot be evaluated. It is a list, the first element being the word lambda. The second element is a list of parameters, and this is followed by some number of forms to be evaluated, which can use the parameters. When the function that the lambda expression names is applied to arguments, the parameters are bound to the arguments and the forms are executed with these bindings.

Using a lambda expression as a function name is like slipping physical actions into your speech, as you would be doing if you referred to what comes at the end of a joke by making a punching motion, then saying the word "line." Lambda expressions see their main use in defining functions, roughly like this:

```
defun <fn-name> <lambda-list>
                        <forms>
```

CLOS adds generic functions to Lisp. Because the evaluation of functions is central to Lisp, the extension of functions to generic functions has a lot to say about how it feels to program in CLOS.

A generic function is a true Lisp function, is called with the same syntax, and can be used in the same contexts in which a Lisp function can be used.

Defining a generic function object is similar to defining a function. You use the *defgeneric* macro, basically like this:

```
defgeneric <fn-name> <lambda-list>
                        <methods>
```

The difference is that, rather than a fixed set of forms to be evaluated, the generic function has a collection of method descriptions, each of which may consist of a number of forms. The method descriptions have their own lambda lists that must be congruent with the main lambda list. Texas Instruments has implemented generic functions in its TICLOS as normal compiled functions with pointers to data structures containing their slots. When the function is called, it is up to the object system to select the appropriate method from its methods. Actually, not select; the technique is more general than this, and is called "method combination." The code eventually executed is called the "effective method."

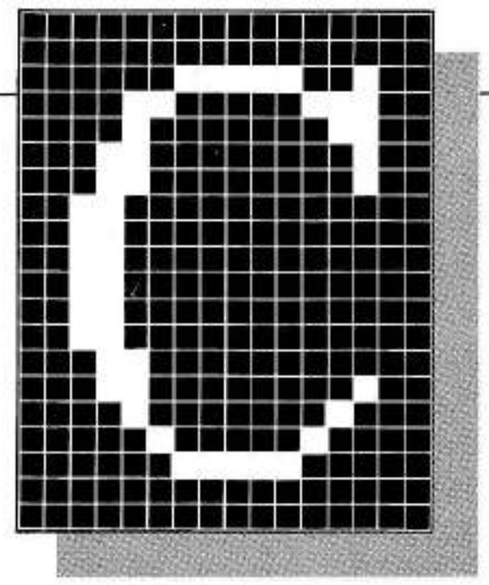
The selection/combination has three stages: select applicable methods, order them by precedence, and apply method combination. The method combination, defined in the definition of the generic function, can be as simple as using the most specific method, or it can be some function of some of the applicable methods. Some built-in method combination types are +, *and*, *or*, *append*, *max*, and *min*, which perform the corresponding functions on the applicable methods to produce the effective method.

Some of the most interesting CLOS functions are those that allow customization of the object system itself, by manipulating metaobjects and metaclasses. Unfortunately, these have not yet been approved by X3J13 for inclusion in the standard. They do, however, support the original spirit of Lisp as an introspective language, with all the strangeness that Douglas Hofstadter suggested when I quoted him last month, a quote that I here double-quote:

"*A . . . double-entendre can happen with LISP programs that are designed to reach in and change their own structure. If you look at them on the LISP level, you will say that they change themselves; but if you shift levels, and think of LISP programs as data to the LISP interpreter . . . then in fact the sole program that is running is the interpreter, and the changes being made are merely changes in pieces of data.*"

Editor's Note: *For a general discussion of functional programming, see "Functional Programming and FPCA '89" by Ronald Fischer,* DDJ, *December 1989. Also, see "A Neural Networks Instantiation Environment" by Andrew J. Czuchry, Jr. in next month's* DDJ *for more information on programming in Lisp.*

DDJ



A Thousand CURSES on TEXTSRCH

Al Stevens

Last month we completed the retrieval processes of the TEXTSRCH project, a "C Programming" column project that we started in December of last year. It builds and maintains a text indexing and retrieval data-base system that allows a user to find text files by composing key word query expressions. The program has two passes: an index builder and a query retrieval program. The query retrieval program searches the text file indexes for files that match the criteria of a Boolean key word search. It delivers a list of the file names that match the search. With the software, developed through last month's installment, a user can determine which files in the text data base match the criteria of the query, and from there he or she can move the files into another application, for example, a word processor.

This month we will add a new feature to TEXTSRCH to allow the user to select and view one of the files from within the TEXTSRCH retrieval program itself. Instead of merely displaying a list of file names that match the query, TEXTSRCH will display them in a menu window from which the user can select. Then it will display the contents of the selected file with the query expression's key words highlighted.

We use this new feature to explore the screen driver software called "CURSES." CURSES is a library of functions that were originally implemented in Unix V. Its purpose is to allow you to write portable, terminal device-independent C programs. The Unix system and the C language are still inexorably oriented to the simple teletype-like console device. The standard input and output devices are such that they can be anything from a clunky old ASR-33 teletype to a high-resolution, many MIPS, full-color, belch-fire, neck-snapper graphics workstation. To support them all, *stdin* and *stdout* must speak to the lowest-common denominator.

There are still many installations that use simple terminal devices, and these devices are grist for the *stdin*, *stdout* mill. Terminals are the same yet they are different. A system's local devices may be many and varied, and the remote dial-up users are likely to be calling in from any one of a number of different terminal types. These different video display terminal devices can work as one because they share the common ability to send and receive ASCII text with carriage returns and line feeds. If that is the only way a program needs to communicate with a user, then these devices share all the commonality they will ever need.

There are, however, features in the typical video display terminal that a program can use to enhance its user interface. Most such terminals have command sequences to clear the screen, position the cursor, and so forth. As you might expect, there is no one way to do all this. ANSI published a standard, and some terminal devices comply. The ANSI.SYS device driver that comes with MS-DOS allows a PC to use the ANSI protocols.

Many terminals have their own, non-ANSI ways to clear the screen, position the cursor, scroll, and achieve other video effects. A program written specifically to use the features of one of these terminals must be modified if an incompatible terminal is connected to the program. As a programmer in such an environment you have three choices: You can write to the common base, which means simple, unadorned, glass-teletype ASCII text; you can use the unique features of the terminal du jour and modify your program every time a new terminal comes into the picture; or you can write to a higher-level video protocol and have a system-level interpreter library translate your video commands into the commands of whatever terminal a user signs on with. The first choice is the appropriate one for text filter programs and console command programs. The second one is appropriate when the operating environment is well-defined and contained, and perhaps when user language performance is an issue. The third choice is the best one to make when you are striving for portability and device independence.

CURSES

To provide for an environment where users with different terminals can use the same software, and where the software can use the video terminal features that go beyond simple text display, the Unix system contains the "CURSES" library and the "termcaps" data base. The data base describes the video protocols of each of the terminals, and the library provides functions that translate a higher-level common protocol into that of the user's terminal device.

CURSES functions facilitate a primitive window-oriented display architecture. You can define windows and use them as virtual terminals. There are character and string display operations, cursor positioning operations, video attributes (such as highlighting and normal displays), keyboard character and string input, scrolling, and simple text editing operations such as inserting and deleting characters and lines.

CURSES works in memory buffers. You address your operations to a defined window, and CURSES makes the changes in memory. These changes do not appear on the screen until you tell CURSES to refresh the window. This method might seem peculiar to a PC

programmer who is accustomed to instantaneous video memory updates. But it reflects its roots in the RS-232 ASCII terminal. It takes more time to update a terminal's screen than it does to write characters into a PC's video memory. For example, a 24 × 80 terminal operating at 19,200 baud will use about a second to refresh its screen. A well-behaved video library can keep a copy of the current screen image and be building another copy to contain whatever changes you are making. When you tell it to refresh, the library can, if the terminals features allow, refresh only that part of the screen that changes.

Lattice C 6.0

Lattice C is an old PC workhorse that has been around since Gates was in short pants and Kahn was a dynasty. It was one of the original full K&R C compilers for the PC. The first Microsoft C was in fact Lattice in a Microsoft binder giving Microsoft an entrance into the C compiler marketplace while they took their time building one of their own. Because Microsoft's own C compiler targeted upward compatibility for programs written with their earlier Lattice version and because the rest of the C compiler business strives for compatibility with Microsoft C, it can be said that Lattice had a strong influence on what C compilers for the PC would become.

There are Lattice versions now for other platforms, including the amazing and wonderful Commodore Amiga. The most recent version for the PC, Version 6.0, supports DOS and OS/2, conforms with the ANSI proposed draft, and comes with a source-level debugger, an editor, an assembler, a librarian, a linker, lots of utility programs, a communications function library, a database library that supports dBase III formats, a graphics library, a library of DOS-OS/2 Family Mode functions, and a CURSES library. If you do not require an Integrated Development Environment after the fashion of Turbo C, QuickC, and others (and many of us do not), this is as complete a C language development environment as you'd want.

The Lattice CURSES Library

The Lattice CURSES library is available in source code for $125 so you can port it to the compiler of your choice. This CURSES library provides a means for developing screen programs that can be ported between DOS, OS/2, and Unix with minimum changes. I used the Lattice compiler and this library to build the document viewing feature that we are adding to TEXTSRCH this month.

Porting Crotchety TEXTSRCH to Lattice C

I wrote the first three installments of TEXTSRCH in Turbo C 2.0. My intention was to make the code as close to ANSI C and as far from the PC architecture as possible to avoid restricting the program to a particular platform. To use the Lattice CURSES library, I decided to port the code to Lattice C rather than to port the CURSES code to Turbo C. Somehow I figured I'd have an easier time of it by porting my own stuff. Maybe, maybe not.

CURSES is a library of functions that were originally implemented in Unix V to allow you to write portable, terminal device-independent C programs

The port was reasonably easy with just a few hitches. Here is what I ran up against, and what follows is a new crotchet that I hereby induct into the "C Programming" column Crotchet Hall of Fame.

It is said that a compiler that compiles programs that comply with the ANSI standard is considered to be an ANSI-conforming compiler. But what about those compilers that extend the standard? For example, Watcom C supports the C++ convention for double-slash comments. The Turbo C *fopen* function allows the use of a non-standard mode parameter. To be sure, both compilers will compile programs that do not use these extensions. But, because you can write programs that use them, you can unintentionally write code that is not ANSI-conforming. Turbo C has, of course, many other extensions, such as pseudo-register variables and interrupt functions. Many compilers now include the interrupt function type, which I first saw in Wizard C, the ancestor of Turbo C. Usually you can tell the compilers to disallow such extensions, that you are interested in writing portable code, and the compilers will comply. But when an extension takes the form of the values accepted as a function's parameters, the compiler does not preempt the extension. So, in all my innocence and with good intentions aforethought, I used the Turbo C "rt" and "wt" formats for the *fopen* mode parameter. The Lattice *fopen* function, in true ANSI compliance, simply refused to open those files because it did not recognize the modes. Turbo C also supports mode formats such as "r+b" where ANSI and the Lattice documentation specify "rb+." Naturally, I used the non-standard formats in my *fopen* calls. You should go through all the code in <index.c> from last month and change every "r+b" to "rb+" and "w+b" to "wb+." Change all *fopen* modes that include the "t" to remove the "t." I believe that the definition of compliance should exclude such extensions.

The next portability issue came with header files. Turbo C puts some function prototypes into more than one header file. In this case, I included the non-standard <process.h> to get the prototype for the *exit* function. According to ANSI, this prototype is in <stdlib.h>, and that is where Lattice keeps it.

The moral of the story has to be: Get a good ANSI function library reference book and ignore the library documentation that comes with your compiler.

Other storms in my port were the result of issues unrelated to ANSI C. The TEXTSRCH <cmdline.c> source file uses the Turbo C *findfirst* and *findnext* functions to search a file directory. ANSI C has no equivalent functions because, I suppose, there are some C platforms that have no analogue to the DOS directory search. When I wrote about those functions last month, I said you would need to make substitutions if you are using a different compiler. Now I find myself in that same boat. Because Lattice has equivalent functions in its *dfind* and *dnext* functions and because it does not have the <dir.h> file that cmdline.c includes, I coded a <dir.h> that substitutes with macros the Lattice functions for the Turbo C functions. You will find <dir.h> as Listing One on page 144.

I had the global variable OK defined as 0, and the Lattice <curses.h> defines it as 1. If you use the Lattice definition, all the TEXTSRCH code works fine.

The next set of problems occurs because of errors in the Lattice header files. It's difficult to imagine how these errors have gone undetected until now. The <curses.h> file includes definitions of keystroke values for the keypad keys. One of these is *KEY_PGDN*, which defines the value returned when you press the PgDn key. The definition, 0x0181, is wrong. It should be 0x0151. The

(continued from page 128)
macros for the CURSES *wstandout* and *wstandend* functions are incorrect. They do not include the *win* parameter in the macro expansion. Not only do you get compiler warnings, but the functions do not work. Finally, the Lattice <stdlib.h> header file specifies in the *free* function prototype that *free* returns *void*, which is wrong. It returns *int*. I had to repair the Lattice header files to proceed.

My final problem was with the CURSES screen driver software. For some reason it reprograms the video mode of my Vega Video 7 in a way that makes the display go off into the weeds at unexpected times, usually after I exit my program. To solve this problem I would need to look at the source code for CURSES, and time and deadlines do not permit. A workaround solution is to run the TEXTSRCH program from a batch file that executes the DOS command MODE CO80 after the TEXTSRCH program exits to DOS.

TEXTSRCH

To install the new file-viewing functions of TEXTSRCH, you must replace the source file named <search.c> from last month with the one in Listing Two, page 144. You must also compile and link <display.c>, Listing Three, page 144, and <error.c>, Listing Four, page 149, into the <textsrch.exe> program.

The BLDINDEX program works the same way that it did before. The new feature is in the TEXTSRCH program. When you enter a query expression the results are now displayed in a screen window with an ASCII -> cursor to the left of each file name. With the up and down arrow keys, you move that cursor and scroll the display. When the cursor points to a file you might want to view, press the Enter key. The first page of the selected document text displays in a new full-screen window. The up and down arrow keys will scroll the display. The up and down page keys will page the display. The Home key goes to the first page and the End key to the last. During the display all occurrences of the key words from the query expression display in a highlighted mode. You can move to the next page where a key word appears by pressing the right arrow key. The left arrow key moves you to the previous page where a key word appears.

Here is how to use CURSES to achieve these results. The *process_result* function in <search.c> is changed. Instead of displaying the matching file names it builds an array of those names. Then it calls the CURSES *initscr* function to initialize the screen manager, calls *select_text* so the user can select a file to look at, and calls the CURSES *endwin* function to shut down the screen manager.

The *select_text* function is where the user picks a file to view. We use the CURSES *newwin* function to build a menu window. The *keypad* function allows the CURSES keyboard routines to recognize the keypad characters, and the *wsetscrreg* function defines the scrolling boundaries of the window. Use of this function prevents the window borders from scrolling along with the rest of it.

The *display_page* function displays a specified page of the file menu in the window. Initially we call it to display the first page. Then we draw a box around the window, write the ASCII -> selector cursor, and read the keyboard. The various cases under the keystroke switch, take care of moving the selector cursor up and down, and paging and scrolling the file selector menu. When the user presses the Enter key, that case calls the *display_text* function, passing the name of the selected file as shown in the menu window.

At this point we must consider the values assigned to the different keys we are interpreting. They are taken

(continued from page 130)
from the Lattice <curses.h> header file and they correspond to what the Lattice version of the CURSES *wgetch* function returns for the cursor keys when the CURSES *keypad* mode is on. These values might not apply to different environments. Also see the use of the *VERT_DOUBLE* and *HORIZ_DOUBLE* global variables in the call to the CURSES *box* function. These too appear in <curses.h> and they correspond to the PC's graphics characters for border characters. You might need to change these values to something that matches your system. CURSES does not provide for border corner characters, but the Lattice implementation recognizes the IBM set and uses the matching corner graphics characters.

Look now at Listing Three to see the code that displays a text file. The function named *display_text* opens the file and calls its *do_display* function if the file opens OK. If not, it calls the *error_handler* function that you will find in Listing Four. This general-purpose function displays an error message in a window, waits for a key press, and clears the message.

The *do_display* function reads all the lines of text from the chosen file and stores them in a linked list in the heap. The list connects each line to its following line and records the positions of any key words in each line.

The *findkeys* function takes care of finding and storing key word occurrences. It scans the line of text comparing each word to the ones in the query expression. If a word matches one of the keys, its character offset relative to the start of the line goes into the header block of the line's linked list entry. The header block can contain up to five key words for each line, which should be enough to call your attention to the line.

After all the lines of text are tucked away in the linked list, the program builds a full-screen window to display the text. The *display_textpage* function displays a page of text beginning with a specified line. It displays the lines a character at a time. If the current character position is marked in the line's header block as the position of a key word, the program calls the CURSES *wpstandout* function to cause the word to be highlighted. When the program finds the next white space character, it calls the CURSES *wpstandend* function to return the display to the normal, non-highlighted mode.

Once a page is displayed, the program reads the keyboard. As with the file selector menu, the keystroke values control the screen display. You can

page and scroll up and down, and you can move the next or previous page where a marked key word appears. The *pagemarked* function makes this test, finding the first line of the specified page and looking at each entry in the list to see if any line has a marked key word.

When you press the ESC key, the function calls *wclear* to clear the text display window and *wrefresh* to refresh that clearing to the screen. Then it deletes the window and frees the heap of the linked list entries.

Back in the *select_text* function the file selector window gets redisplayed and the user can pick out another file to look at.

TEXTSRCH Performance

How effective is the CURSES approach to the development of portable code? The proof would be in the successful porting of a program such as this one to another platform. I am sure that this program would port to a Unix system with no more fuss than I had moving it to Lattice C. There is, however, one big area of concern in such a move. We do not know how efficiently the program would operate. CURSES is a technique for the portability of screen driver code to a multitude of display devices. Its implementation in the Lattice library makes for an effective and efficient program because they used all the PC tricks for fast screen updates. What's more, I developed this program on and for a 20-MHz 386 computer. The only way to know how well or poorly this particular use of CURSES would work on a slower machine or with a different terminal is to move the program. So, with that in mind, I moved TEXTSRCH to the slowest compatible computer at my house, an 8-MHz COMPAQ II. I am happy to report that it works fine. This does not, however, qualify it for an environment where the terminal device is a serial VDT. I would suspect that some of the ways I used CURSES are not the best choices for such a setup. A seasoned CURSES programmer probably knows intuitively what to do and what to avoid to support the most effective user interface.

The collective abilities and shortcomings of CURSES across a wide selection of terminals would, no doubt, influence the way you would design a user interface. Given that one could learn these boundaries and with all this in mind, I can conclude that CURSES is an effective technique for wide platform independence of text-based screen management. That, of course, is no news to Unix programmers, who have had CURSES for several years. It is news to those others of us who might be looking for tidy ways to develop programs on the PC that can be moved to other operating environments.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

(Listings begin on page 144.)

Sifting for Sharks' Teeth

Prowling the 23 miles of aisles at Comdex Fall, looking for programmer tools, is like sifting the sand hills over in Lockhart Gulch west of Scotts Valley, looking for sharks' teeth. You know that they're down there, and if you dig long enough you'll find a few. However, the smart guys run down to New Age Annie's Kosmic Krystal Koop in Santa Cruz and buy one of the nice clean sharks' teeth Annie keeps in a "Save the Whales" bowl next to the two-for-a-dollar tiger eyes. Saves a heap o' diggin' — which is what you're doing by buying this magazine.

Into the Outback

What wild and wonderful programmer stuff there is is not on the main floor, by and large. (Exceptions might include the Microsoft booth, which was the size of a small county in Arkansas.) Finding the good stuff means traipsing around the outlying hotels such as the Tropicana and Bally's.

The #1 Neat Comdex Idea for programmers comes from two different vendors, who solved the same knotty problem using two different technologies. The problem is a common one: Running out of DOS memory while doing a build on a large application using command-line compilers and linkers. QuickPascal has this problem in spades; for all its many virtues, QP uses memory like cheap cologne and always runs out before Turbo Pascal. Even a memory miser such as Turbo will run out eventually if you hand it a big enough application.

Jeff Duntemann, KI6RA

Qualitas' superb 386-to-the-MAX nibbles on the problem by using the 386's hardware memory manager to remap some of 386 extended memory down beneath the video refresh buffer. You can get a contiguous DOS memory area as large as 704K if you're using a monochrome display adapter. A small San Jose, California company named V Communications takes the idea much further, by moving the video refresh buffer entirely to some other location in 386 memory and making BIOS aware of the move. Their Memory Commander product can give you as much as 924K of contiguous DOS memory, depending on what TSRs, device drivers, and BIOS software needs space in the first megabyte.

924K is an extreme case. The company says a typical system should be able to have about 860K available for compiles, if no attempt is made to address screen memory directly. Because command-line compilers and linkers typically write to standard output rather than the refresh buffer, this is not a problem. And 860K could allow you to build a much larger app. Think of all that symbol table space . . .

Invisible Software of Foster City, Calif. has a product that does much the same thing, only they use a little-known and less-understood feature called "shadow RAM," supported by several of the Chips and Technologies VLSI chip sets for 286 and 386 motherboards. Shadow RAM is present only in those machines using those chip sets. If the motherboard is equipped with a minimum 1 Mbyte of RAM, (rather than the canonical 640K) the chip set can map portions of that RAM where it needs to. The feature was developed to allow the copying of code from slower BIOS ROMs into faster RAM to improve performance, but it can also map RAM into the segment space between $A000 and $B800 (assuming you don't have a monochrome display board) giving you a contiguous DOS space of as much as 736K. So while the Invisible RAM product does not give you quite as much potential space as Memory Commander, it has the advantage of working in the great many inexpensive Asian 286 motherboards that use the Chips chip sets. (Memory Commander, remember, is a 386-only product.) You can download a test program from Invisible Software's BBS to detect and report on whether you have the necessary chip set in your system. Call them for details if you're interested; it's a very slick product.

Documentation on Demand

The #2 Neat Comdex Idea for programmers solves an ugly logistical problem facing shareware authors: How to provide attractive printed documentation without going broke. As one of the inducements to registering a shareware package, many authors offer typeset printed documentation. The catch is that manuals cannot be printed economically in batches of fewer than 500 or so, and costs don't really go down until the numbers head up into the tens of thousands.

However, when you punt your shareware creation out into the brave, cold world, you have no idea how many registrations you're likely to get. Worse, products generally evolve far more quickly than 500 manuals are likely to be needed, leaving authors stuck with piles of obsolete manuals that are fully paid for — and worthless.

Workhorse laser printers (especially HP's that prints on both sides of a sheet at once) and desktop publishing packages such as Ventura Publisher allow high-quality, short-run printed output. What's needed is a mechanism to bind loose sheets together in a professional-looking way, and at Comdex I found one: The Unibind binding system.

In a nutshell, Unibind works like this: The sheets to be bound are placed inside a plastic or card-stock folder with a thermoplastic adhesive bar running down the middle. This assemblage is then placed in a toaster-gadget that positions the sheets and cover accurately, and heats them until the adhesive melts and glues the sheets together at the spine and the spine to the cover. The system can bind stacks from 2 sheets to 650 sheets in size, and each volume takes about 45 seconds to bind.

Systems similar to this have been available for some time, but the ones I've seen and used (typically from Cheshire) are extremely messy and mechanically fragile. Unibind is neither; the bound volumes are tidy and show no loose traces of adhesive, and the binder device has far fewer moving parts than Cheshire and similar systems.

Once bound, the sheets are in there for the long haul; I was unable to pull any of the sheets from the bound volume without tearing them. On the downside, the system has significant upfront costs, and the per-piece cost of the bound volumes is higher than volumes printed and bound at a printing plant. However, there is no waste and no obsolescence, because the system truly allows "documentation on demand." You print what you need as you need it, folding in updates as they happen, no sooner, no later. You can support several low-volume shareware products without going broke printing 500 manuals for each while expecting to sell maybe 20 or 30 manuals per year.

It's getting tougher and tougher all the time to put low-cost specialty software products on the market and make them pay. Shareware is our last best hope in this regard, and Unibind can help solve that ugly documentation issue. If you're a shareware author you ought to look into it.

Stereo-On-A-Card

The #3 Neat Comdex Idea for programmers may seem a little loopy, but it solved an infuriating problem for me and may solve that same problem for you if you're one of the many programmers who listens to music while programming. The Desktop Stereo product from Optronics of Ashland, Ore. is a half-sized board for the PC bus containing a world-class FM stereo receiver and 4 watts per channel amplifier. There are no tuning knobs on the board bracket; all controls are done electronically, through pop-up dialog boxes containing, among other things (dare I say it?) radio buttons. You can view the FM band as a graph of vertical bars displaying signal intensity at various frequencies (neat touch!) and preset up to ten frequencies with mnemonic names such as "KRAP" or "Hillbilly Rock" and punch them up like buttons on your car radio.

The problem that this board solves is that the expensive Japanese CD-equipped boom boxes that many of us place beside our RAM charged 386 boxes leak like sieves. Unless your favorite FM station's towers are on the next block, what you'll hear on your FM receiver is likely to be your machine's switching transients playing solo, and that is dull (if powerful) music. I'd long since abandoned FM and simply play my CDs. The FM module on the Desktop Stereo card is extremely well shielded (it had better be!) and absolutely quiet in the absence of signal modulation.

Now I can listen to PBS again. 20 plus stations accessible from fringey Scotts Valley. No racket. Jeff-Bob says check it out.

All Set with Modula

Let's continue our discussion of the vice president of Structured Languages, Modula-2. Will Modula ever overtake Pascal for small machines? Probably not. Unless . . . the president decides not to run in OS/2 land, in which case the race gets interesting. Modula-2 is already very big over on the OS/2 side of things, second (so far as I can tell) only to You Know What. If this continues for a few more years, the OS/2 products could achieve a formidable critical mass, especially since Modula contains standard syntactic support for multitasking. (More on that very thorny issue when I get OS/2 running reliably on this sorry excuse for a 386 machine.) If you're contemplating a project for OS/2, ignore those C-sirens claiming that C is the only way to go. You can do very well with Modula-2, according to sources that I trust. Someday I'll know from firsthand experience, sigh.

No, in this issue we're going to talk about sets. Sets are what drove me out of Modula-2 several years ago. When the language spec was first released I jumped on it, with full intent to port over my disks full of code, written in

the faltering corpse of Pascal/MT+ for CP/M-80. I dug in and discovered several days into the project that I couldn't do it. My code was absolutely peppered with the killer type definition:

```
TYPE
    CharSet = SET OF Char;
```

Uh-uh, said the compiler. Sets in Modula-2 may have no more than 16 elements.

This is a serious semantic bite in the buns. Sets work well for me and I use them a lot, especially for building systems to handle characters moving from one place to another, as from the keyboard to the screen or from a serial port to the screen or to a disk file. Like Maxwell's Demon, a set is a filter that can pass odd characters among the ASCII throng while denying passage to others in a group just as odd. Consider the elegance of this classic construct:

```
IF AnswerChar IN ['Y','y']
    THEN DoIt ELSE DontDoIt;
```

The alternative is this:

```
IF (AnswerChar = 'Y') OR
    (Answer-Char = 'y')
    THEN . . .
```

You might argue that the second form resolves to fewer machine instructions, and I'd argue back that you're rarely going to have to execute 17,000 such tests in a tight loop. Furthermore, what about this:

```
IF IncomingChar IN WhiteSpaceSet
                        THEN . . .
```

There's simply nothing like sets for character filters such as this. It was just possibly possible in some cases to pull tricks with subranges of fewer than 16 characters, but the whole notion offended me: Niklaus Wirth threw character sets out the window to make it easier to implement Modula-2. There are maybe two or three hundred potential Modula-2 compiler implementors in this world. There are hundreds of thousands of potential Modula-2 programmers. One suspects he skipped Marketing 101 as an undergrad.

About then Turbo Pascal happened, and Modula-2 slipped into eclipse for some years. Logitech held the torch alight all that time, but their product, while solid, was complex and slow and admittedly intended for internal use. It wasn't until JPI introduced TopSpeed Modula-2 that the language showed any serious life. Soon afterward, the Stony Brook compiler made its debut, and I've begun to do some serious work in Modula again.

The reason is pretty simple: TopSpeed and Stony Brook have done the Awful Thing: Extended Modula-2 by allowing sets to have as many as 65,536 elements. Horrors. You might not be able to port your dog kennel management package to the Lilith operating system. It is to cry real tears.

Niklaus Wirth threw character sets out the window

Duntemann's One Law of Portability

Remember this, chilluns: For any platform with I/O more complex than a batch system, semantic differences between platforms makes portability impossible. In other words, even if you wrote your character-based PC kennel manager in absolutely standard Modula-2, could you port it to the Macintosh? If you had written it for multiple terminals under Unix, could you port it to DOS? Get real — the effort spent resolving semantic conflicts would far outweigh trifles like the shape of an IF statement.

So let's quit arguing about something that's never been worth a plugged nickel outside of academe anyway.

Watch the Corral, Not the Cows!

A set is an abstraction of a group of values, indicating whether one or more of those values are present or not present. It's like a corral on a farm with seven cows; at any given time a cow is either in the corral or not. The cows are in no particular order within the corral. They're either there or else out making things for the unwary to step in.

It's important to remember that the set is not the cows; the set is the corral. It's still a set even when it is empty.

In Modula-2, a set is defined in terms of some ordinal type or subrange of an ordinal type, including enumerations such as the insufferable list of colors that every writer on the subject (myself included) has used in books explaining the concept:

```
TYPE
    Colors = (Red, Orange,
    Yellow, Green, Blue, Indigo,
                                Violet);
    WarmColors = [Red . . Yellow];
    ColorSet = SET OF Colors;
    WarmSet = SET OF WarmColors;
    CardSet = {0..65535}
    CharSet = SET OF CHAR; (* Yay! *)
```

Beneath it all, in physical memory, a set is a bitmap. There is one bit in the set for each value that may legally be present in the set. Each bit carries one Boolean fact: Whether the value that the bit stands for is present or not present in the set. Adding a value to the set is done by raising that value's bit to binary 1. Removing a value from the set is done by changing that value's bit back to a binary 0.

A "full" set (that is, one having all values present) is not one bit larger than an empty set. Again, the set is the corral, not the cows!

Set Operators

There are a number of operators and standard procedures that work on sets in Modula-2. The two most obvious are *INCL*, which places a value in a set, and *EXCL*, which removes a value from a set. These are not present in Pascal. *IN* is still there, doing exactly what it does in Pascal: Return a Boolean value indicating whether the value on the left is present in the set on the right. Ditto >= (set inclusion, right in left), and <= (set exclusion, left in right) which do much the same but for whole sets: >= returns TRUE if all values of the set on its right are present in the set on its left; and <= returns TRUE if all values in the set on its left are present in the set on its right.

There are actually only four operators that are true set operators in that they act on sets and return sets: + (set union) – (set difference) * (set intersection) and / (set symmetric difference). Of these, only the first three are present in Pascal.

Set union of two sets returns the set that contains all the elements present in both of the sets taken as one. Set intersection of two sets returns the set of values that are present in both sets, but none of those values that may be present in one or the other but not both.

Set difference is a little trickier; my Pascal prof explained it badly (getting it mixed up with symmetric difference, see below) and I misunderstood it through ten years and two editions of my book. Set difference of two sets returns the set that consists of the elements in the set on the left once those in the set on the right have been removed from it.

Basically, set difference is a way of pulling several elements out of a set without using *EXCL* to do it one element at a time:

(continued from page 136)

{'A'..'Z'} – {'M'..'Z'}

This set expression returns the set *{'A'..'L'}*. (Keep in mind that Modula-2 uses curly brackets for set constructors rather than straight brackets.)

Finally, set symmetric difference (which is not in any Pascal implementation I'm aware of) is rather like set union turned inside out. The symmetric difference of two sets is the set of all elements that are present in one or the other set, but not in both sets. In a sense, the symmetric difference of two sets is what the two sets don't have in common; for example, what remains once their intersection (overlap) has been removed.

Remember that the set is not the cows; the set is the corral

Among them, these operators allow you to do just about anything with a set that you'd ever want to do. And now that sets can have up to 65,535 elements in Modula-2, that's a lot.

The Naked Set

Wirth's original language definition did not hard-code 16 as the number of elements in a set. The number of elements in a Modula-2 set was originally defined as the number of elements in the machine word used by the system for which the compiler was implemented. In other words, in a system with a 32-bit word there would be 32 possible elements in a Modula-2 set.

This makes those limited set operations very easy to implement, and very fast, because they can be done using the native bit-manipulation instructions present in all modern-day CPUs. Remember that sets are bitmaps. Furthermore, the four true set operators bear a certain uncanny functional resemblance to certain logical operators such as *AND*, *OR*, and *XOR*.

OR the bits of two sets together and whammo, suddenly you have the union of the two sets. *AND* the bits of two sets together, and what remains is the intersection of the two sets. *AND* the bits of one set with the complement (reversed) bits of another set, and you remove the bits of the complemented set from the other set, that is, set difference. Finally, *XOR* the bits in two sets together and what's left are the bits that are present in one set or the other but not in both sets, since *XOR* drives identical bit pairs to 0. Voilà: Symmetric set difference.

This is, of course, exactly what Wirth intended, and he intended for it all to happen within the accumulator of the host CPU, ensuring speed and minimal fussing. Happily, in this brave new world of fast global optimizing compilers (Stony Brook's is fabulous) we can have it both ways: When we're fiddling small sets we can do it fast at one shot inside the accumulator; when we're fiddling big sets we can do it a word at a time and take the performance hit.

Now, Wirth defined a specific kind of set that has no true analog in Pascal: *BITSET*, a standard type supported in all Modula-2 compilers. A *BITSET* is a machine word used as a bitmap. All of the set operators operate on *BITSET* values. A *BITSET*'s nominal values are 0 . . 15, but these are bit numbers more than values. A *BITSET* is thus a sort of naked set, in which the bitmap nature of the set is laid bare and can be manipulated directly. A bit in a *BITSET* does not abstract a color, or a character, or a cardinal number, or a cow; a bit in a *BITSET* represents a bit, period.

Twiddling Bits in Other Types

With very little futzing, this fills an apparent gap in Modula-2: The lack of explicit bit-manipulation facilities. Turbo Pascal has explicit bitwise *AND*, *OR*, *NOT*, and *XOR* operators for numeric ordinal types, and it can also shift bits in numeric ordinal values with its *SHR* and *SHL* operators. Modula-2 has none of these . . . or does it?

It does . . . but they only operate on values of type *BITSET*.

No problem — just ask Pizza Terra. (For those unfamiliar with the reference, see my May 1989 column.) Modula-2 has explicit type casting (which Wirth calls type coercion), so if you want to fiddle bits in type *CHAR*, cast type *CHAR* onto type *BITSET*, and fiddle away! Any type can be cast onto any other type of identical size, and there are transfer functions such as *Ord* to cast 8-bit types like *CHAR* and *BOOLEAN* onto 16-bit types like *CARDINAL*.

For example, to *AND* a *CARDINAL* variable *MyCard* with the value 128, you could do this:

```
NewCard :=
   CARDINAL(BITSET
         (MyCard) * BITSET(128));
```

Here, *MyCard* and the value 128 are

(continued on page 141)

(continued from page 138)
both cast onto *BITSET*s, which are then *AND*ed together by using the set intersection operator, which is equivalent (on a bit level) to *AND*. Finally, the result of the set intersection operation is cast back onto a *CARDINAL* for assignment to the *CARDINAL* variable *NewCard*.

This works . . . but it sure as hell isn't obvious. Unfortunately, in Modula this is how the game is played. Better to disguise all this arm-twisting of types (coercion is such a lovely word!) behind some procedures with more mnemonic names. This is what I've done in the listings for this column, which present a Modula-2 module called *Bitwise*. Listing One, page 150, is the definition module for *Bitwise*, and Listing Two, page 150, is the implementation module.

Bitwise provides function procedures to perform bitwise *AND*, *OR*, *XOR*, and *NOT* operations. (See Table 1.) Note that the capitalization is different from that used here in the descriptive text, in order to differentiate my procedure *And* from the existing (and incompatible) Boolean logical operator *AND*. (Case is significant in Modula-2, and this is the first time in my career I've caught myself being glad. Crazy world, ain't it?) Additionally, *Bitwise* contains procedures to set, clear, and test individual bits, and also to shift values right or left by up to 16 bits. This suite of routines provides roughly the same bit-banging power you get stock in Turbo Pascal. This seems to be the lot of Modula-2 programmers: To perpetually build what those Turbo guys have come to take for granted!

The formal parameters for all of the routines in *Bitwise* are type *CARDINAL*, because *CARDINAL* is the unsigned 16-bit numeric type in Modula-2, equivalent to *Word* in Turbo Pascal. It's a good basic foundation upon which to cast all other ordinal types in Modula-2. (And it's used quite a bit by itself.) If you want to set bit number 3 in a character, for example, you could do this:

```
NewChar :=
        CHAR(SetBit(ORD('A'),3));
```

The *ORD* transfer function casts the character value onto a *CARDINAL* value for passing to the *SetBit* function procedure, and finally the *CARDINAL* value returned by *SetBit* is cast back onto a character for assignment to *NewChar*.

Read over the code implementing *Bitwise* and it all makes sense to you. Again, understand type casting/coercion and you've got it in your hip pocket.

When Words Runneth Over

There is something a little bit hazardous about *Bitwise*. The *SHR* and *SHL* routines can cause overflow errors if you shift bits to the extent that 1-bits roll out of either side of the 16-bit word in which they exist. Stony Brook Modula-2 code checks for overflow errors and will crash your program when you shift bits out of the word they live in.

Now, shifting bits off the edge of their word is not necessarily a bad thing. Sometimes you do it deliberately to get rid of the bits in question. There's nothing inherently damaging about it, because on a machine level the bits get shunted first into the carry flag and then off into nothingness. (What we affectionately call "the bit bucket.") Adjacent data is never overwritten, no matter if we try to shift a bit by (a meaningless) 245 positions.

The way out is to turn off overflow error checking. Enter here one of my major arguments with Modula-2: For portability's sake (gakkh!) there are no compiler toggles. Turbo Pascal has a whole raft of them, things like *{$R-}* and so on. The situation would seem to call for bracketing the *SHR* and *SHL* routines between compiler toggles that switch overflow checking off only for the duration of the routine, then on again once the routine terminates.

Sorry, Charlie. As every good tuna fish knows, compiler toggles are implementation dependent and destroy the prospects for portability. Lord knows, we can't have that, now, can we? The best that can be done with the Stony Brook compiler is to turn off overflow checking entirely within the *Bitwise* module by changing the compile options on a by-module basis. Be sure to do this when you compile and use *Bitwise*! If you're using a Modula compiler in which overflow checking cannot be turned off, you'd better add

Bitwise operators	Set operation	
AND	*	Intersection
OR	+	Union
XOR	/	Symmetric difference
NOT	{0..15} - BITSET	"Full" set - target set

Table 1: *Relating bitwise operators to set operations*

safety belts to any code that uses *SHL* and *SHR*.

The Boss DOS Book

There is a certain type of book I call a "category killer;" it's *the* book on a certain subject and tends to keep other books of its type from being published. One of these is Ray Duncan's *Advanced MS-DOS* (Microsoft Press), a book that has never been very far from my left hand while sitting in this particular chair. I'm pleased to report that Ray has company, in the form of Que Corporation's *DOS Programmer's Reference*, by Terry Dettmann. On 892 pages Terry has managed to summarize every BIOS function through PS/2, every DOS call through V4.0, all mouse function calls, all EMS function calls, and a blizzard of other information including low-level disk structure, device driver and interrupt programming, serial port programming, and lots more.

The very best part about this book, however, may well be its index. Having 892 pages of information is small comfort if you can't find anything when you need it in a hurry. The index occupies 33 pages, with about 100 citations per page, set small in two columns. Everything I tried to look up was either indexed or not covered in the book. (And things that weren't covered really shouldn't have been anyway, like VGA hardware architecture details.)

Altogether, the best hacker's book to cross my desk in a good long while. Get it.

Dredging the Channel

There are millions — nay, tens of millions — of DOS machines out there, and various research reports I've seen indicate that the greatest growth potential lies in machines of modest cost and capabilities: The "bare bone" 88 and 286 clones that fill *Computer Shopper* to a depth of 800+ pages every month. There are already 30 million of them (conservative estimate) and in another few years there could be as many as 100 million of them out there, plugging away. This is an utterly unbelievable market for software products, and yet the distribution channel has closed up to the point that a small-time operator (like most of us) has no chance to make those millions of people even aware of the existence of their products.

There has got to be a way. Any ideas? Pass them by me. I'll be talking about this subject in future months, and I'll share some guerrilla marketing concepts I've devised, and will discuss how the little guys can shove some very big rear ends out of their monopoly position in the retail channel.

Write to Jeff Duntemann on MCI Mail as JDuntemann, or on CompuServe to ID 76117,1426.

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

Products Mentioned

Memory Commander
V Communications
3031 Tisch Way, Ste. 802
San Jose, CA 95128
408-296-4224
$129.95

Invisible RAM
Invisible Software
1165 Chess Drive, Ste. D
Foster City, CA 94404
415-570-5967
$39.95

Unibind
Unibind Systems
7900 Capwell Drive
Oakland, CA 94621
415-638-1060
Various configurations and prices
Contact the vendor for specifics

Desktop Stereo
Optronics Technology
P.O. Box 3239
Ashland, OR 97520
503-488-5040
$199

DOS Programmer's Reference, 2nd edition
Terry Dettmann, revised by Jim Kyle
Que Corporation, 1989
ISBN 0-88022-458-4
Softcover, 892 pages, $27.95

(Listings begin on page 150.)

Listing One *(Text begins on page 127.)*

```
/* ------------ dir.h ----------- */

/* Substitute Lattice directory functions for
 * Turbo C directory functions
 */

#include <dos.h>

#define ffblk FILEINFO
#define ff_name name

#define findfirst(path,ff,attr) dfind(ff,path,attr)
#define findnext(ff) dnext(ff)
```

End Listing One

Listing Two

```
/* ---------- search.c ----------- */

/*
 * the TEXTSRCH retrieval process
 */

#include <stdio.h>
#include <string.h>
#include <curses.h>
#include "textsrch.h"

static char fnames[MAXFILES] [65];
static int fctr;

static void select_text(void);
static void display_page(WINDOW *file_selector, int pg);
void display_text(char *fname);

/* ---- process the result of a query expression search ---- */
void process_result(struct bitmap mapl)
{
    int i;
    extern int file_count;
    for (i = 0; i < file_count; i++)
        if (getbit(&mapl, i))
            strncpy(fnames[fctr++], text_filename(i), 64);
    initscr();                  /* initialize curses */
    select_text();              /* select a file to view */
    endwin();                   /* turn off curses */
    fctr = 0;
}

/* ------- search the data base for a word match -------- */
struct bitmap search(char *word)
{
    struct bitmap mapl;

    memset(&mapl, 0xff, sizeof (struct bitmap));
    if (srchtree(word) != 0)
        mapl = search_index(word);
    return mapl;
}

#define HEIGHT 8
#define WIDTH 70
#define HOMEY 3
#define HOMEX 3

#define ESC 27

/* --- select text file from those satisfying the query ---- */
static void select_text(void)
{
    WINDOW *file_selector;
    int selector = 0; /*selector cursor relative to the table */
    int cursor = 0;   /*selector cursor relative to the screen*/
    int keystroke = 0;

    /* --- use a window with a border to display the files -- */
    file_selector = newwin(HEIGHT+2, WIDTH+2, HOMEY, HOMEX);

    keypad(file_selector, 1);       /* turn on keypad mode    */
    noecho();                       /* turn off echo mode     */
    wsetscrreg(file_selector, 1, HEIGHT);/* set scroll limits */

    /* -------- display the first page of the table --------- */
    display_page(file_selector, 0);

    while (keystroke != ESC)    {
        /* ----- draw the window frame ------ */
        box(file_selector, VERT_DOUBLE, HORIZ_DOUBLE);

        /* ------------ fill the selector window ------------ */
        mvwaddstr(file_selector, cursor+1, 1, "->");
        wrefresh(file_selector);

        /* -------------- make a selection ------------------ */
        keystroke = wgetch(file_selector);/* read a keystroke */
        mvwaddstr(file_selector, cursor+1, 1, "  ");
        switch (keystroke)  {
            case KEY_HOME:
                /* -------- Home key (to top of list) ------- */
                selector = cursor = 0;
                display_page(file_selector, 0);
                break;

            case KEY_END:
                /* ------- End key (to bottom of list) ------ */
                selector = fctr - HEIGHT;
                if (selector < 0)   {
                    selector = 0;
                    cursor = fctr-1;
                }
                else
                    cursor = HEIGHT-1;
                display_page(file_selector, selector);
                break;

            case KEY_DOWN:
                /* - down arrow (move the selector cursor) -- */
                /* --------- test at bottom of list --------- */
                if (selector < fctr-1)  {
                    selector++;
                    /* ------ test at bottom of window ------ */
                    if (cursor < HEIGHT-1)
                        cursor++;
                    else    {
                        /* ---- scroll the window up one ---- */
                        scroll(file_selector);
                        /* --- paint the new bottom line ---- */
                        mvwprintw(file_selector, cursor+1, 3,
                            fnames[selector]);
                    }
                }
                break;

            case KEY_UP:
                /* --- up arrow (move the selector cursor) -- */
                /* ---------- test at top of list ---------- */
                if (selector)   {
                    --selector;
                    /* -------- test at top of window ------- */
                    if (cursor)
                        --cursor;
                    else    {
                        /* --- scroll the window down one --- */
                        winsertln(file_selector);
                        /* ----- paint the new top line ----- */
                        mvwprintw(file_selector, 1, 3,
                            fnames[selector]);
                    }
                }
                break;

            case '\n':
                /* -- user selected a file, go display it --- */
                display_text(fnames[selector]);
                break;

            case ESC:
                /* --------- exit from the display ---------- */
                break;

            default:
                /* ---------- invalid keystroke ------------ */
                beep();
                break;
        }
    }
    delwin(file_selector);  /* delete the selector window */
    clear();                /* clear the standard window  */
    refresh();
}

/* ------ display a page of the file selector window ------- */
static void display_page(WINDOW *file_selector, int line)
{
    int y = 0;
    werase(file_selector);
    while (line < fctr && y < HEIGHT)
        mvwprintw(file_selector, ++y, 3, fnames[line++]);
}
```

End Listing Two

Listing Three

```
/* ---------------- display.c ----------------- */

/* Display a text file on the screen.
 * User may scroll and page the file.
 * Highlight key words from the search.
 * User may jump to the next and previous key word.
 */

#include <stdio.h>
#include <stdlib.h>
#include <curses.h>
#include <ctype.h>
#include <string.h>
#include "textsrch.h"
#define ESC 27

/* ---------- header block for a line of text ---------- */
struct textline {
```

(continued on page 146)

Listing Three *(Listing continued, text begins on page 127.)*

```
    char keys[5];               /* offsets to key words     */
    struct textline *nextline;  /* pointer to next line     */
    char text;                  /* first character of text  */
};

/* --------- listhead for text line linked list -------- */
struct textline *firstline;
struct textline *lastline;

int pagemarked(int topline);
static void do_display(FILE *fp);
static void findkeys(struct textline *thisline);
static void display_textpage(WINDOW *text_window, int line);

/* ---------- display the text in a selected file --------- */
void display_text(char *filepath)
{
    FILE *fp;

    fp = fopen(filepath, "r");
    if (fp != NULL) {
        do_display(fp);
        fclose(fp);
    }
    else    {
        /* ----- the selected file does not exist ----- */
        char ermsg[80];
        sprintf(ermsg, "%s: No such file", filepath);
        error_handler(ermsg);
    }
}

static void do_display(FILE *fp)
{
    char line[120];
    WINDOW *text_window;
    int keystroke = 0;
    int topline = 0;
    int linect = 0;
    struct textline *thisline;

    firstline = lastline = NULL;

    /* --------- read the text file into the heap ------- */
    while (fgets(line, sizeof line, fp) != NULL)    {
        line[78] = '\0';
        thisline =
            malloc(sizeof(struct textline)+strlen(line)+1);
        if (thisline == NULL)
            break;      /* no more room */

        /* ----- clear the text line record space -------- */
        memset(thisline, '\0', sizeof(struct textline) +
                        strlen(line)+1);

        /* ---- build the text line linked list entry ---- */
        if (lastline != NULL)
            lastline->nextline = thisline;
        lastline = thisline;
        if (firstline == NULL)
            firstline = thisline;
        thisline->nextline = NULL;
        strcpy(&thisline->text, line);

        /* ------------ mark the key words ------------ */
        findkeys(thisline);
        linect++;
    }

    /* ------- build a window to display the text ------- */
    text_window = newwin(LINES, COLS, 0, 0);
    keypad(text_window, 1);     /* turn on keypad mode    */

    while (keystroke != ESC)    {
        /* --- display the text and draw the window frame --- */
        display_textpage(text_window, topline);
        box(text_window, VERT_SINGLE, HORIZ_SINGLE);
        wrefresh(text_window);

        /* ------------ read a keystroke ------------- */
        keystroke = wgetch(text_window);
        switch (keystroke)  {
            case KEY_HOME:
                /* ------- Home key (to top of file) ------ */
                topline = 0;
                break;
            case KEY_DOWN:
                /* --- down arrow (scroll up) ---- */
                if (topline < linect-(LINES-2))
                    topline++;
                break;
            case KEY_UP:
                /* ----- up arrow (scroll down) ---- */
                if (topline)
                    --topline;
                break;
            case KEY_PGUP:
                /* -------- PgUp key (previous page) -------- */
                topline -= LINES-2;
                if (topline < 0)
                    topline = 0;
                break;
            case KEY_PGDN:
                /* -------- PgDn key (next page) ------------ */
                topline += LINES-2;
                if (topline <= linect-(LINES-2))
                    break;
            case KEY_END:
```

(continued on page 148)

Listing Three *(Listing continued, text begins on page 127.)*

```
            /* ------- End key (to bottom of file) ------ */
            topline = linect-(LINES-2);
            if (topline < 0)
                topline = 0;
            break;
        case KEY_RIGHT:
            /* - Right arrow. Go to next marked key word  */
            do  {
                /* -- repeat PGDN until we find a mark -- */
                topline += LINES-2;
                if (topline > linect-(LINES-2)) {
                    topline = linect-(LINES-2);
                    if (topline < 0)
                        topline = 0;
                }
                if (pagemarked(topline))
                    break;
            }   while (topline &&
                    topline < linect-(LINES-2));
            break;
        case KEY_LEFT:
            /* Left arrow. Go to previous marked key word */
            do  {
                /* -- repeat PGUP until we find a mark -- */
                topline -= LINES-2;
                if (topline < 0)
                    topline = 0;
                if (pagemarked(topline))
                    break;
            }   while (topline > 0);
            break;
        case ESC:
            break;
        default:
            beep();
            break;
        }
    }
    /* -------- clean up and exit --------- */
    wclear(text_window);
    wrefresh(text_window);
    delwin(text_window);
    thisline = firstline;
    while (thisline != NULL)    {
        free(thisline);
        thisline = thisline-> nextline;
    }
}

/* ---- test a page to see if a marked keyword is on it ---- */
int pagemarked(int topline)
{
    struct textline *tl = firstline;
    int line;
    while (topline-- && tl != NULL)
        tl = tl->nextline;
    for (line = 0; tl != NULL && line < LINES-2; line++)    {
        if (*tl->keys)
            break;
        tl = tl->nextline;
    }
    return *tl->keys;
}

#define iswhite(c) ((c)==' '||(c)=='\t'||(c)=='\n')

/* ---- Find the key words in a line of text. Mark their
   character positions in the text structure ------- */
static void findkeys(struct textline *thisline)
{
    char *cp = &thisline->text;
    int ofptr = 0;

    while (*cp && ofptr < 5)    {
        struct postfix *pf = pftokens;/* the query expression */
        while (iswhite(*cp))  /* skip the white space */
            cp++;
        if (*cp)    {
            /* ---- test this word against each argument in the
                query expression ------- */
            while (pf->pfix != TERM)    {
                if (pf->pfix == OPERAND &&
                        strnicmp(cp, pf->pfixop,
                            strlen(pf->pfixop)) == 0)
                    break;
                pf++;
            }
            if (pf->pfix != TERM)
                /* ----- the word matches a query argument.
                   Put its offset into the line's header --- */
                thisline->keys[ofptr++] =
                    (cp - &thisline->text) & 255;
            /* --- skip to the next word in the line --- */
            while (*cp && !iswhite(*cp))
                cp++;
        }
    }
}

/* --- display page of text starting with specified line --- */
static void display_textpage(WINDOW *text_window, int line)
{
    struct textline *thisline = firstline;
    int y = 1;

    wclear(text_window);
    wmove(text_window, 0, 0);
```

```
/* ---- point to the first line of the page ----- */
while (line-- && thisline != NULL)
    thisline = thisline->nextline;

/* ------- display all the lines on the page ------ */
while (thisline != NULL && y < LINES-1) {
    char *cp = &thisline->text;
    char *kp = thisline->keys;
    char off = 0;
    wmove(text_window, y++, 1);

    /* ------ a character at a time -------- */
    while (*cp) {
        /* --- is this character position a key word? --- */
        if (*kp && off == *kp)  {
            wstandout(text_window); /* highlight key words*/
            kp++;
        }

        /* ---- is this character white space? ---- */
        if (iswhite(*cp))
            wstandend(text_window); /* turn off hightlight*/

        /* ---- write the character to the window ------ */
        waddch(text_window, *cp);
        off++;
        cp++;
    }
    /* -------- a line at a time ---------- */
    thisline = thisline->nextline;
  }
}
```

End Listing Three

Listing Four

```
/* ------------ error.c ------------ */

/* General-purpose error handler */

#include <curses.h>
#include <string.h>

void error_handler(char *ermsg)
{
    int x, y;
    WINDOW *error_window;

    x = (COLS - (strlen(ermsg)+2)) / 2;
    y = LINES/2-1;
    error_window = newwin(3, 2+strlen(ermsg), y, x);
    box(error_window, VERT_SINGLE, HORIZ_SINGLE);
    mvwprintw(error_window, 1, 1, ermsg);
    wrefresh(error_window);
    beep();
    getch();
    wclear(error_window);
    wrefresh(error_window);
    delwin(error_window);
}
```

End Listings

Listing One *(Text begins on page 134.)*

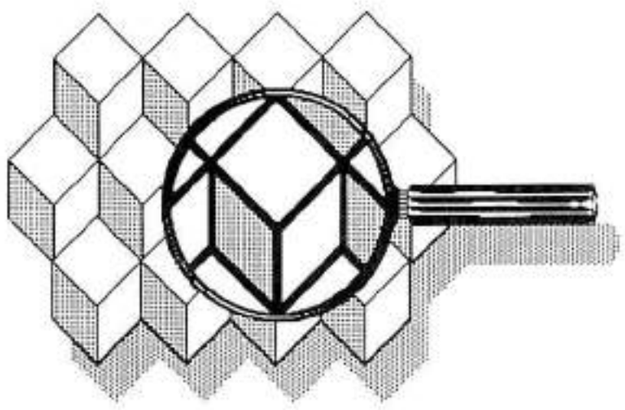
```
(*-------------------------------------------------*)
(*                  BITWISE.MOD                    *)
(*              Definition Module                  *)
(*                                                 *)
(*     Bit-manipulation routines for Modula-2      *)
(*                                                 *)
(*                            by Jeff Duntemann    *)
(*                            For DDJ : March 1990 *)
(*                            Last modified 11/25/89 *)
(*-------------------------------------------------*)

DEFINITION MODULE Bitwise;

PROCEDURE And(A,B : CARDINAL) : CARDINAL;

PROCEDURE Or(A,B : CARDINAL) : CARDINAL;

PROCEDURE Xor(A,B : CARDINAL) : CARDINAL;

PROCEDURE Not(Target : CARDINAL) : CARDINAL;

PROCEDURE SetBit(Target : CARDINAL; BitNum : CARDINAL) : CARDINAL;

PROCEDURE ClearBit(Target : CARDINAL; BitNum : CARDINAL) : CARDINAL;

PROCEDURE TestBit(Target : CARDINAL; BitNum : CARDINAL) : BOOLEAN;

PROCEDURE SHR(Target : CARDINAL; By : CARDINAL) : CARDINAL;

PROCEDURE SHL(Target : CARDINAL; By : CARDINAL) : CARDINAL;

END Bitwise.
```

End Listing One

Listing Two

```
(*-------------------------------------------------*)
(*                  BITWISE.MOD                    *)
(*            Implementation Module                *)
(*                                                 *)
(*     Bit-manipulation routines for Modula-2      *)
(*                                                 *)
(*                            by Jeff Duntemann    *)
(*                            For DDJ : March 1990 *)
(*                            Last modified 11/25/89 *)
(*                                                 *)
(*  NOTES ON THE CODE:                             *)
(*                                                 *)
(*  In all cases below, BitNum MOD 16 is used as a *)
(* means of ensuring that BitNum will be in the    *)
(* range of 0..15.  MOD 16 divides by 16 but returns *)
(* the remainder, which cannot be over 15 when you *)
(* divide by 16.                                   *)
(*-------------------------------------------------*)

IMPLEMENTATION MODULE Bitwise;

VAR
  I       : CARDINAL;
  TempSet : BITSET;


PROCEDURE And(A,B : CARDINAL) : CARDINAL;

BEGIN
  RETURN CARDINAL(BITSET(A) * BITSET(B));
END And;


PROCEDURE Or(A,B : CARDINAL) : CARDINAL;

BEGIN
  RETURN CARDINAL(BITSET(A) + BITSET(B));
END Or;


PROCEDURE Xor(A,B : CARDINAL) : CARDINAL;

BEGIN
  RETURN CARDINAL(BITSET(A) / BITSET(B));
END Xor;


PROCEDURE Not(Target : CARDINAL) : CARDINAL;

BEGIN
  RETURN CARDINAL({0..15} - BITSET(Target));
END Not;


PROCEDURE SetBit(Target : CARDINAL; BitNum : CARDINAL) : CARDINAL;

BEGIN
  TempSet := BITSET(Target);  (* INCL does not operate on expressions! *)
  INCL(TempSet,BitNum MOD 16);
  RETURN CARDINAL(TempSet);   (* Cast the target back to type CARDINAL *)
END SetBit;


PROCEDURE ClearBit(Target : CARDINAL; BitNum : CARDINAL) : CARDINAL;

BEGIN
  TempSet := BITSET(Target);  (* EXCL does not operate on expressions! *)
  EXCL(TempSet,BitNum MOD 16);
  RETURN CARDINAL(TempSet);   (* Cast the target back to type CARDINAL *)
END ClearBit;


PROCEDURE TestBit(Target : CARDINAL; BitNum : CARDINAL) : BOOLEAN;

BEGIN
  IF (BitNum MOD 16) IN BITSET(Target) THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END;
END TestBit;


PROCEDURE SHR(Target : CARDINAL; By : CARDINAL) : CARDINAL;

BEGIN
  FOR I := 1 TO By DO
    Target := Target DIV 2;
  END;
  RETURN Target;
END SHR;


PROCEDURE SHL(Target : CARDINAL; By : CARDINAL) : CARDINAL;

BEGIN
  FOR I := 1 TO By DO
    Target := Target * 2;
  END;
  RETURN Target;
END SHL;


END Bitwise.
```

End Listings



Codecheck, a rule-based expert system that checks C and C++ source code for maintainability, portability, and compliance with in-house style, has been announced by **Conley Computing**. Codecheck has the ability to identify the number of operators per expression and lines per statement, and it provides a statistical analysis of code complexity and style, allowing programmers to check for both industry standards and those established by their company.

Codecheck also reviews code for its portability to ANSI C and K&R C, among others. Company president Patrick Conley told *DDJ* that Codecheck can be beneficial to both corporations and individuals, but especially to corporations that use many programmers for single projects. "The problem is getting programmers to adhere to standards; since everyone has their own Tower of Babel concerning standards, Codecheck can be programmed to check in-house style."

Codecheck supports all C compilers from major vendors, and is available for PC-DOS and Macintosh at $495, for OS/2 at $695, and for AIX, PC/IX, and QNX at $995. Multiple copy and educational discounts are also available. Reader service no 21.
Conley Computing
7033 SW Macadam Ave.
Portland, OR 97219
503-244-5253

The Paradox Engine, a C library for the relational database Paradox, has been announced by **Borland International**. The company claims that this product will enable C programmers to build applications that create or access Paradox data because programs that use the Paradox Engine are standard .EXE files. The benefit is interoperability among Borland's major business applications and languages, which theoretically allows the building of customized computing environments.

A program written with the Paradox Engine is compiled in C and linked with the Paradox Engine library to build an executable application that can dynamically access Paradox data. The PAL language can also access Paradox tables.

The engine provides an API of more than 70 funcitons, which allows the manipulation of Paradox tables in single and multiuser environments. The C version should be shipping this quarter, and will cost $495. A Pascal version is scheduled for release sometime in the middle of the year, and OS/2 and Windows versions are also under development. During the first 90 days of availability, registered Borland users can purchase the product for $195. Reader service no. 22.
Borland International
P.O. Box 660001
Scotts Valley, CA 95066-0001
408-439-1622

VRTX-PC, a real-time environment for the PC/XT/AT compatibles that allows these machines to be used as both development platforms and embedded computers, has been introduced by **Ready Systems**. Time-critical applications in which deterministic operating system performance is necessary can now be controlled by PCs. The company is excited that the VRTX-PC allows simultaneous development and execution of real-time multitasking applications, eliminating the need for low-level hardware control on the PC. They believe that this technology will reduce development costs and get products on the shelf faster.

The VRTX-PC real-time operating system supports MS-DOS functions, including all MS-DOS file and device I/O, and can be executed as a DOS resident program.

VRTX-PC includes a real-time kernel, a real-time debugger, an input/output file executive, a run-time library, a PC support executive, and a window manager that provides a user interface. For application development, VRTX-PC supports Microsoft C and Borland Turbo C. The price for a single user is $7600. Reader service no. 23.
Ready Systems
P.O. Box 60217
Sunnyvale, CA 94086
408-736-2600

The Sierra C toolset for the M68000 is available from **Sierra Systems**. The toolset includes an optimizing C compiler and complete C run-time library, two assemblers, linker, librarian, code management and debugging utilities, a serial downloader, a high-speed parallel downloader, and a source-level debugger. The company claims that the code produced is position independent, ROMable, and re-entrant.

The Sierra C compiler that is included in the toolset is ANSI compatible and supports the keywords and functionality required for embedded systems programming.

Compiler flags control individual suppression of optimization techniques, generation of floating point code (inline or for the 68881), formatting and contents of the listing and assembler output files, generation of source level debugger information, IEEE floating point operation modes, and register usage, among others. Reader service no. 24.
Sierra Systems
6728 Evergreen Ave.
Oakland, CA 94611
415-339-8200

PC Techniques, a new magazine for programmers, has been announced by **The Coriolis Group**. The first bimonthly issue will be published with a March/April 1990 cover date. The magazine will become a monthly publication in January of 1991.

PC Techniques will cover the DOS, Windows, OS/2, and Presentation Manager platforms. C, Pascal, Basic, and assembly language will be covered in every issue. Specialty languages like C++, Object Pascal, Smalltalk, and Actor will also find coverage.

The Coriolis Group was founded by *DDJ* columnist Jeff Duntemann and by Keith Weiskamp, occcasional *DDJ* author. *PC Techniques* is available for $21.95 for one year and $37.95 for two. Reader service no. 25.
Coriolis Group
3202 E. Greenway, Ste. 1307-302
Phoenix, AZ 85032
602-493-3070

Two new journals, *Inside Turbo C* and *Inside Turbo Pascal*, which offer programmers ongoing support of these two Borland languages, have been announced by **The Cobb Group**. The purpose of the two journals is to explore new algorithms, system tricks, and product updates, including complete source code. They will also contain tips, programming techniques, product news and reviews, as well as advice. And *Inside Turbo Pascal* covers OOP with Turbo Pascal.

Each journal costs $59 for 12 issues; sample issues are available. Source code in both issues can be downloaded from Cobb's BBS, for a yearly fee of $30. Reader service no. 26.
The Cobb Group
P.O. Box 24480
Louisville, KY 40224
800-223-8720

The original developer of Turbo Prolog, the **Prolog Development Center**

(continued on page 157)

(continued from page 152)
(PDC), has been granted the rights to the product by Borland International. The PDC will publish and market new versions under the name PDC Prolog. According to Michael Alexander at PDC, "The new version is a superset of the current Turbo Prolog. With the exception of the turtle graphics predicates, it is source-compatible with Turbo Prolog, so existing Turbo Prolog programs can be compiled 'as is' with PDC Prolog." And PDC Prolog supports the Borland BGI graphics interface.

A new DOS version should be available by now, and registered users of the DOS version of Turbo Prolog will be able to upgrade for $79. The OS/2 version should also be available, and will cost $599. Network support and a SCO 386 Unix version is scheduled for release in the second quarter of this year. Reader service no. 27.
Prolog Development Center
568 14th Street N.W.
Atlanta, GA 30318
404-873-1366

Intek C++ 2.0 is now available from **Intek Integration Technologies**. The company claims the product has as much power as AT&T's C++ 2.0 in an 80386 MS-DOS or Unix environment. Intek C++ 2.0 translates C++ code into C code. It supports most DOS C compilers, including Microsoft C, Turbo C, MetaWare High C and High C 386, Watcom C and Watcom C 386, and Novell Network C and Network C 386.

This support also includes the C extended keywords *near*, *far*, *huge*, *cdel*, *pascal*, and *fortran*, which makes it useful with Microsoft Windows and OS/2.

The Intek C++ translator uses 386 protected memory mode, and can compile large programs — up to 4 gigabytes. It supports multiple inheritance, type-safe linkage, new and delete operators as class members, overloading of the ->, ->*, and, operators, const and static member functions, and static initialization. It requires 1 Mbyte of memory, MS-DOS 3.1 or later or Unix System V/386, and costs $495. Reader service no. 28.
Intek
1400 112th Ave. SE, Ste. 202
Bellevue, WA 98004
206-455-9935

A C++ compiler for 80386/486 Unix-based systems has been released by **Peritus International**. In addition to AT&T C++ 2.0, the highly-optimized C++ compiler also provides support for K&R C and ANSI C; programmers select the appropriate C dialect by setting a compiler switch.

The compiler supports an extensive set of data types, including 8-, 16-, 32-, and 64-bit integers, IEEE-compatible 32-, 64-, and 80-bit floating point, user-defined aggregate types, and C++ class data types. The optimizations include global register allocation, constant propagation and folding, backward code motion with loop invariant removal, induction variable elimination, redundant store and dead code removal, and constant elevation.

Company president Ron Price told *DDJ* that Peritus intends on providing class libraries and development tools within the near future, including a package to provide a graphical interface to the X Windows system. He also said that the C++ is compliant to the AT&T 2.0 spec, except for multiple inheritance, which will also be supported in the near future.

The Peritus C++ compiler, which runs on 386/486 systems under SVR3 Unix and SunOS 4.0 Unix, sells for $1000. Reader service no. 29.
Peritus International
10201 Torre Ave., Ste. 295
Cupertino, CA 95014
408-725-0882

A few new assembly tools are now available. An assembly language library written entirely in assembly language has been released by **Quantasm Corporation**. Quantasm Power Lib (QPL) contains over 256 routines, provides high-level functionality, and has the ability to be customized.

QPL can be used by both novice and expert programmers. The documentation is coordinated with example programs on disk. The company claims that the compactness of QPL makes it convenient for programming memory resident programs or TSRs.

The product includes a menu and windowing system, over 75 string handling functions, extended precision math functions, a set of date/time functions, encryption/decryption algorithms, file name parsing, and sound control. The company intends to have high-level language interface routines available in the first quarter of this year. QPL requires MS- or PC-DOS 2.1 or above; 256K RAM; IBM PC/XT/AT, PS/2 or compatible; Microsoft MASM, Borland TASM, or SLR OPTASM. This product is not copy protected, nor has run-time royalties. The price is $99.95 without source code, $299.95 with. Reader service no. 30.
Quantasm Corporation
19855 Stevens Creek Blvd.
Cupertino, CA 95014
408-244-6826

From **Base Two Development** comes Spontaneous Assembly, an assembly-language library that contains over 600 functions and macros, including string and memory manipulation, near/far/relative heap management, doubleword/quadword integer math, date and time manipulation, and more. The company claims that every routine is hand-coded and optimized, and are easy to use because of the register-oriented parameter-passing convention. Company spokesman Alan Collins told *DDJ* that "this product does for 8088-family assembly language programming what Borland did for high-level language programming."

Spontaneous Assembly supports all Microsoft/Borland standard memory models, as well as custom models and mixed-model programming. The tool sports a full-overlapping windowing system with custom shadowing that allows direct memory via screen access or BIOS. DOS 2.0 or higher is required, and MASM 5.1 or TASM 1.0 are recommended. It costs $199, includes all source code, and comes with a money back, 60-day guarantee. Reader service no. 3.

Base Two Development
11 East 200 North
Orem, UT 84057
800-277-3625

Another is DASM, a disassembler for the 8086, 8088, and 80286, available from **JBSoftware**. DASM is able to disassemble and modify programs for which the source code is unavailable. It takes binary run files for DOS and compatible operating systems as input, and creates an assembly language file suitable for modification and reassembly as output. It acts as a virtual machine and maps the program being disassembled. It tracks register usage and determines the code, data, and labels, allowing the user to then edit the output and change the program.

DASM works by viewing commands and procedures in their real-time processing order, rather than in the sequence they appear in the program, which JBSoftware claims makes the programs easier to interpret and edit. Some of DASM's other features include the ability to generate appropriate ASSUMEs and segment maps, to handle multiple entry points, transfer vectors, and .EXE, .COM, and .BIN files up to 200K. It costs $250. Reader service no. 2.

JBSoftware
701 Cathedral St., Ste. 81
Baltimore, MD 21201
301-752-1348

Two new software products for Motorola's 88000 RISC microprocessor are available from **Diab Data**. The D-CC/88K, an optimizing C compiler, complies with the 88000 object code compatibility standard (OCS) and the Binary compatibility standard (BCS), and conforms to the proposed ANSI C standard. Optimizations include global common subexpression elimination, lifetime analysis (color), reaching analysis, automatic register allocation, loop invariant code motion, constant propagation and folding, dead code elimination, switch optimizations, and the ability to pass parameters into registers.

Diab's MC88000 toolkit is made up of the D-AS/88K Assembler, the D-LD/88K Linker, and the D-AR/88K Archiver. This package includes the D-CC/88K optimizing C compiler. The assembler is also OCS and BCS compliant, produces COFF object modules, supports standard MC88000 mnemonics, produces standard Unix directives for organizing code, among other things. The linker performs literal synthesis, generates warnings for unidentified external references, and is able to perform incremental links. The archiver maintains multiple files in a single archive file, and supports Unix System V command-line options. The compiler and toolkit are available for the Sun3/SunOS, Mac II/MPW, DECstation/Ultrix, and DEC VAX/VMS, among others. Reader service no. 33.

Diab Data Inc.
323 Vintage Park Dr.
Foster City, CA 94404
415-573-7562

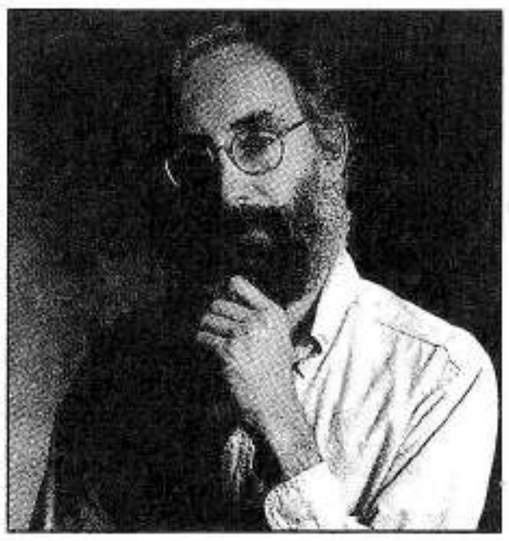
Books of Interest

A comprehensive treatment of concurrent programming techniques in the Strand programming language has been published by **Prentice Hall**. *Strand: New Concepts in Parallel Programming*, by Stephen Taylor and Ian Foster, covers an introduction to Strand, basic and advanced programming techniques, and how to apply Strand, with examples from both the academic and real worlds. The price is $30. ISBN 013-850587-X. Reader service no. 38.

Prentice Hall
Englewood Cliffs, NJ 07632
201-767-5937

DDJ

Pub Crawler



I read a lot of magazines. I read during meals, while talking to Jon on the phone, and while visiting the little programmer's room. I also follow magazine's fortunes, and I thought I'd pass along the latest rumors regarding some in which you may be interested.

CD-ROM End User. If you are interested in CD-ROM and haven't seen this, give it a look. Once you get past the uninspired name, the amateur editing, and the boring design, you'll find a bimonthly packed with information of solid value for both CD-ROM users and developers, written and compiled by knowledgeable people.

Embedded Systems Programming. Those whose realm is the other kind of ROM should know that *ESP* has gone to controlled circulation. What this means if you're a subscriber or potential ditto is that you may get it for free. What it means if you're an advertiser or potential ditto is that you can look for increased rates. It's a zero-sum game.

Micro Cornucopia. Dave Thompson is considering taking his 50-issue-old hacker's magazine monthly. He's looking for a "partner" — one with money to invest, I gather.

Microsoft Systems Journal. MSJ has been redesigned, and it's an improvement, though the publication still works too hard at being taken seriously. It's probably too much to expect that *MSJ*'s editors could learn from someone such as Dave Thompson how wit and playfulness can coexist with solid technical content.

Other captive magazines. Sun's user magazine is about to be sold — "given" is a better word, from what I hear of the deal — to IDG, publisher of *Computer World, InfoWorld, PC World, Macworld*, etc.; while Aldus has launched a magazine with a surprisingly drab look. The content is too self-serving, but the first issue contains a few good things, including what may be the most quick-and-dirty DTP how-to ever written, and an interview with Steve Ballmer on OS/2.

Ziff-Davis. The company that publishes *PC Magazine, MacUser, PC Computing, Digital Review*, and others (and that killed off *Creative Computing, Popular Electronics, PC Tech Journal*, and others) has been rumored for the past six months to be on the block. The rumors, which are making ulcers for Z-D employees, have been vehemently denied by Bill Ziff. The rumors are remarkably detailed: Pat McGovern, chairman of IDG, has perused the perspectus; Cahners, publisher of *Mini-Micro Systems*, has tendered an offer; the asking price is in the $800 million range; Goldman Sachs & Co. is handling the deal. If you believe Ziff's denials, you are led to believe that the rumors were started by one of Z-D's competitors. Whatever the truth, somebody is an awfully big liar.

Buzzwords

"Done deal" is one of those buzzwords that should buzz off, and I apologize for using it. Another buzzword that I hope won't catch on in the 90s is "experience," as in "user experience." Apparently the multimedia types within Apple are pushing to use it in the place of "user interface." I get the point, but I hope they keep this one in house.

My pick for the buzzword of the 90s is "facilitate." At least it has the right polysyllabic, academic aura. But I actually think it could be a GOOD buzzword. No, really. Here's why.

I believe fervently in the value of education, but I don't buy into the myth of teaching. The existence of this verb "teach" conveys the erroneous impression that it is possible to force-feed knowledge. The best teachers seem to understand that there is no such thing: Richard Feynman, on being given a teaching excellence award by the American Association of Physics Teachers, said, "I don't know how to teach. I have nothing to say about teaching," then went on to deliver a brilliant and entertaining lecture.

If you can't teach anyone anything, then all you can do is get out of the way, move any obvious obstacles aside, and let them learn. Facilitating learning, you might call it. The problem, I guess, is that it's hard to do. Clearing the student's path is one of those subtle acts that succeeds only by making itself invisible.

Like good writing, and like good user interface design. Good writer Esther Dyson discussed the desktop metaphor in the January issue of *PC Computing*, saying that it "is not meant to suggest that the computer is a desktop, but to provide a sense of recognition and reasonable expectations. This metaphor, so popular now, suggests tasks the computer can reasonably be expected to do." Suggest things. Create an environment the user can explore, letting the user discover things by recognizing the familiar and following reasonable expectations into the unfamiliar. Get out of the user's way. Facilitate. Yeah. I like the word. The trouble is that if it catches on, people will start ringing the changes on it: facilitator, facilitation, facile. And sooner or later some user is going to walk into a computer store and ask to be shown the facilities. And be taken to the little programmer's room. Might be all right if there are some good magazines in there.

Michael Swaine

Michael Swaine
editor-at-large

Dr. Dobb's Journal Bound Volume 15 - 1990

The new decade begins and *DDJ* enters its 15th year of power programming. Every 1990 issue of *Dr. Dobb's Journal* has been collected and bound into this single volume. Included are the popular Annual C and Annual Operating Systems issues. Featured are well-known contributors such as Michael Swaine, Al Stevens, and Jeff Duntemann. There's also the *Dr. Dobb's Journal* 1989 index, and all the source code for 1990 (supplied on disk, PC/MS-DOS format). Some of the 1990 topics include:

- **Real-Time Data Acquisition.** Valuable information on all the tools you need (both hardware and software), for your own data acquisition system.
- **Three-dimensional Graphics Using The X Window System.** 3-D graphics are possible with X Window systems! Here's what can be expected from porting 3-D graphics to X, plus solutions to some thorny problems.
- **68040 Programming.** This member of the 680x0 family provides challenges for programmers at all levels.
- **Neural Nets.** DDJ presents an environment that dynamically creates neural networks. Also included are discussions of the similarities and differences of various neural net models.
- **Memory Management.** Everything from how to take advantage of "handle pointers" to object swapping.
- **Hypertext.** A behind-the-scenes look at the *DDJ* hypertext project.
- **Graphics.** From Super VGA programming to drawing character shapes with Bezier curves.
- **C Programming.** Porting C programs to 80386 protected mode, encapsulating C memory allocation, parallel extensions to C, and much more!
- **Unraveling Optimization.** Examined are the practical and theoretical aspects of code optimization using Microsoft C 6.0.
- **Communications & Connectivity.** Controlling Unix processes, designing for OSI, and programming with Mac Comm toolbox.

53995
9 781558 511835
ISBN 1-55851-183-0
>$39.95

M&T BOOKS

M&T Publishing, Inc.
501 Galveston Drive
Redwood City, CA 94063