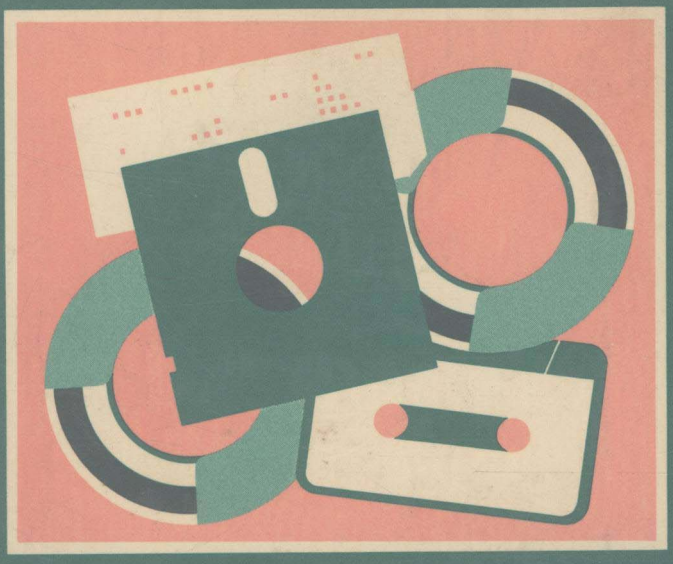
microprocessor software design



Edited by Max J. Schindler

Selected from ELECTRONIC DESIGN

HAYDEN

MICROPROCESSOR SOFTWARE DESIGN

Selected from
ELECTRONIC DESIGN

Edited by

MAX J. SCHINDLER

Software/Systems Editor, *Electronic Design*

HAYDEN BOOK COMPANY, INC.
Rochelle Park, New Jersey

Library of Congress Cataloging in Publication Data
Main entry under title:

Microprocessor software design.

1. Microprocessors—Programming. I. Schindler, Max J. II. Electronic design.
QA76.6.M488 001.64'25 79-27834
ISBN 0-8104-5190-5

Printed in the United States of America

1 2 3 4 5 6 7 8 9 PRINTING

80 81 82 83 84 85 86 87 88 YEAR

MICROPROCESSOR SOFTWARE DESIGN

Preface

To future generations of engineers, the 1970s will be known as the decade of the microcomputer revolution. Already, the slide rule is only a memory, and labs are filled with instruments spitting out complex test results that took days to calculate from raw data less than 10 years ago. Without test automation, the shortage of skilled technicians would by now have slowed R & D to a crawl in practically every engineering discipline. And microcomputer "brains" are transplanting robots from sci-fi movies into factories.

As applications mushroom, IC prices drop. And so, new more complex applications can be implemented, boosting IC production and lowering microcomputer costs even more. Chip and board computers seem set to take over the world. But while microprocessors, ROMs, RAMs, PIAs, and math chips may be the equivalent to the electronic brain's gray matter, one thing is missing: life. Without a program, even the most sophisticated computer is as dead as the sand from which its ICs are made.

But programs, unlike chips, can't be mass produced on an assembly line. They must be custom designed, to suit each application, by software engineers. The number of these specialists, their productivity, and their ingenuity are the factors that will determine computer use in the 1980s.

As more and more circuits, from op amps to custom logic arrays, become available off the shelf, yesterday's systems designer turns more and more into a software designer. Whether he works on a smart multimeter or a sophisticated video processor, he needs an operating system to make the instrument tick. So tomorrow's engineers will not only have to design applications software, but systems software—a specialty that used to be reserved for the elite of the computer fraternity.

But to write systems software, the designer must also know the hardware. In fact, that's true even for applications software written in assembly language, and several hardware-oriented articles are therefore included in this book. Several earlier Hayden books cover microcomputer hardware and architecture in more detail, including *Microprocessor Basics* (1977) and *Microprocessors: New Directions for Designers* (1976). For an overview of available microprocessors, their features, and characteristics, the *Microprocessor Data Manual* (1978) is an excellent source. All of these books were compiled from articles published in *ELECTRONIC DESIGN*.

While assembly languages still account for more microcomputer software than higher-level languages, the trend is definitely away from assembly. Why? Assembly-language programs are not only hard to write and debug, but are firmly tied to one processor. Since new and more powerful processors appear on the market at a breathtaking clip, it's getting ever more important that software be "portable." After all, hundreds of thousands of dollars may be invested in a software package, and no company can afford to throw away such an investment every time a $10 processor is replaced.

Higher-level languages offer such portability—no wonder the section covering them is the biggest in this book by far. The two languages covered in depth are as different as two high-level languages can be. BASIC, the Be-

ginner's All-purpose Symbolic Implementation Code, was designed for easy learning and has been adopted by the hobby community. So today perhaps as many as 100 BASIC dialects are in use—a characteristic that somewhat defeats the purpose of a high-level language. Nevertheless, BASIC is firmly entrenched in the microcomputer field, and nearly monopolizes the small-business market.

PASCAL was also developed as a teaching tool, but certainly not for beginners. It incorporates all the features today's computer scientists hold essential for a high-level language: strong type checking of data, modularity, and constructs that encourage structured programming. In fact, some engineers find PASCAL awkward to use, because it imposes so much more "overhead" than BASIC. But for large programs, it's important to start out with clearly defined variables. And if for no other reason, PASCAL is bound to gain popularity because it's the basis for the Defense Department's "universal" language, ADA.

Clearly, there is room in the software spectrum for efficient assembly languages as well as for powerful PASCAL. In fact, where speed is most important, microcode—essentially machine language—is also gaining wider acceptance. The tradeoffs depend on each application, and Section V provides some examples.

So where is software headed? In all directions, because that's where computers are headed. The spectrum of applications for electronic "brains" continues to widen and will soon encompass all levels of engineering, from toys and toasters to autos, from communications and energy management to chip design and advanced research. There's room for 4-bit micros with a dozen assembly instructions and lightning-fast mainframes that run programs many megabytes long.

But the mainstream of digital development will be dominated by 8- and 16-bit microprocessors for many years to come, and that's where most of the software effort will go—providing the "smarts" for consumer products and automation from the sales office to the production line. Not everybody will be happy with the impact those tiny pieces of silicon will have on our lives, but nobody can stop their great march forward.

MAX J. SCHINDLER

Contents

Section VI. Hardware–Software Tradeoffs Depend on the Application

Section VII. Program Development Aids Help with System Integration, Too

Section VIII. High-Level Languages Boost Microcomputer Power

Section IX. Software Will Determine Computer Systems' Future

SECTION I

Software for Any Occasion

This section gives a broad overview of available software and skims the wide horizon of engineering applications. As the cost of human brainpower rises while silicon "brains" keep getting cheaper, the design engineer must learn to shift more and more of his job to these untiring servants.

The first article examines the different kinds of languages for microcomputers, from assembly over intermediate ("systems implementation") to higher-level languages. Hardware–software tradeoffs are discussed, and sources for ready- or tailor-made software are listed.

The second article examines the realm of computer-aided design (CAD) and its pitfalls. CAD ranges from board layout over circuit analysis to self-optimization—and a listing of program sources by CAD application makes it easy to get started.

Fit Your Microcomputer with the Right Software Package

MAX J. SCHINDLER
Software Editor,
Electronic Design

Software is threatening to smother the microcomputer explosion. Words like software crisis and software gap aren't empty slogans.

But why not simply adapt the mountains of software developed for minicomputers? After all, micros are looking more and more like minis, anyway. Still, very few μCs are fully software-compatible with their mini ancestors, and mini software will rarely run on micros without major revisions.

The problem with μC software is that it will consume a much larger part of your development budget than you are accustomed to with minis and mainframes. When you can buy or assemble computers for under $100, a software package that costs a hundred or even a thousand times as much, obviously can't be considered expendable. With mainframes or minis the computer is the capital investment, but with μCs it's the software that must keep going, while the computer becomes expendable.

What makes it even worse, with μC software you're pretty much on your own; you can't expect much software support from vendors who sell you $5 chips or $99 boards. It's ironic that you lose that help just when you start playing with higher stakes: a software error in a μC system can wipe you out.

Say your company makes 100,000 units of a "smart" product. A software development bill of $100,000 would be no big deal, but spending just $10 apiece to fix a bug in the field—not to mention a recall—is a nightmare you want to avoid.

Microcomputer software should be machine-independent (or "portable") so you don't have to throw it out every time a better μP comes up, and it should be reliable so you don't embed dangerous bugs in your programs. Yet, μP software should be written quickly to keep its cost in check. How do you reconcile these goals?

Not with machine language, which is, by definition chip-dependent. The next higher level, mnemonic assembly language, would be better. But chip makers have been much too busy carving out their share of the market to worry about standardizing their instruction sets. Today, disgruntled users are taking vendors —and the IEEE—to task for the mnemonics jungle, but you can't wait for it to be cleared.

That leaves only two solutions: Either you put your own system together (e.g., from bit slices), or you switch to higher-level languages.

There's another, and even better, reason for going to a higher-level language: It's easier to use.

Look at a simple problem: Selecting the larger of two numbers in memory, and storing the "winner" in a third location. In 8080 hexadecimal machine code, the program looks like this:

3A FF 01 21 D3 00 BE DA 43 01 7E 32 88 01.

These 14 instructions can be compressed into just six assembly language statements, namely

```
      LDA   Y
      LX1   H, X
      CMP   M
      JMC   GO
      MOV   A, M
GO:   STA   Z
```

In a higher-level language, the whole problem could well be expressed in a single statement:

IF(X>Y)Z=X;ELSE Z=Y

A programming language that's easy to work with can't help but be welcomed. One reason is that more and more software is being written not by full-time programmers, but by design engineers with system responsibilities. Engineers need higher-level languages more than ever.

Babel revisited

While use of higher-level languages is attractive, choosing one can be a nightmare. The crowd of competing higher-level languages has become the system designer's Tower of Babel. There are now hundreds of languages if you include major dialects.

So which language should you pick? That depends on how you answer questions like these:

Are you designing an operating system? A small-business system? A lathe controller? Will your production run be 10, or 100, or 1000? Is your background in hardware or in software? Or do you belong to the new "mixed-ware" generation?

If your system is intended for a small production run, you need a quickly learned language that's also

easy to debug (Fig. 1). On the other hand, if you are developing operating software, the code must be efficient, or it will slow down your system.

While you're usually looking for a language that speeds up programming, in the case of an operating system (OS), you'll be more concerned about reliability. Assembly language, in the hands of a competent programmer, generates very efficient code, but it's also very prone to hard-to-find mistakes. Here, too higher level languages provide a solution, which will be discussed later.

Fortunately, a big obstacle to higher-level language implementation on μCs is crumbling: the cost of memory. If car prices had followed the same decline over the last 10 years, a 1978 Cadillac would cost about $100.

The wordy higher-level languages require not only more space for program storage, they also need resident translators (compilers and interpreters) to convert English-like inputs into machine-readable object code.

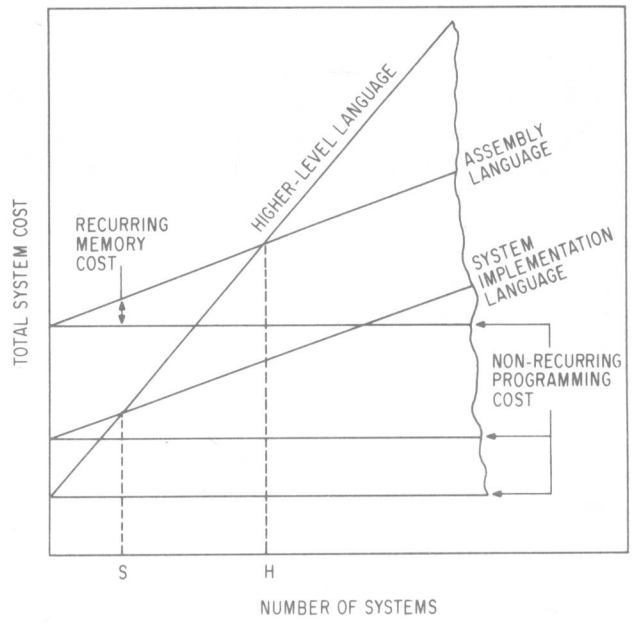
Not all higher-level languages use compilers. Basic, for instance, mostly employs interpreters. So which is better? Only you can decide. An interpreter converts your statements (source code) into machine language one by one, and is therefore best suited for interactive use. But in such an application you expect error messages as you go along, and that can slow you down badly. Especially if the error messages are their usual cryptic selves.

Syntax error 37, line 20

Say your input reads PRINTFORB. A simple interpreter might accept that as PRINT FORB, while a more sophisticated one would ask you whether you perhaps mean PRINT F OR B. And a *really* smart one would realize that it's impossible to print F *or* B—without telling when you want which. But on a small μC, the smart interpreter might well be too slow for you, even if you can fit it into your memory. Another drawback is that an interpreter doesn't create object code (i.e., save the translated machine language). And for highly repetitive tasks, the need to translate again and again makes execution agonizingly slow.

A compiler, on the other hand, typically doesn't complain much until the program has started to execute. So you may get a cryptic error message for line 2.1, but no chance to fix the mistake until the object code for several hundred (or thousand) statements has been completed. But if the compiler is any good it will generate (and save) efficient object code, and execute your program very quickly. Logically, then, Fortran—a number-crunching language as the name (formula translator) implies—should be run with a compiler, and so it nearly always is.

Compare a typical computer run using an interpreted language like Basic (Fig. 2a) with one using a compiler (Fig. 2b). Compiled programs (e.g., Fortran) are typically more complex than interpreted ones. But compiled runs usually have two moments of truth: Does it run? Does it work? Getting the program to run may be the tougher part because you have no numerical output to guide you. On the other hand, if a complex program doesn't work, you can go right back to square one: your algorithm may not hack it.



1. **Total μC system cost** includes nonrecurring software and recurring memory costs. The ratio depends on the language used. Assembly beats higher-level languages below H, but SILs are more economical above point S.

Compilers and interpreters can be single or multiple-pass. But even a highly optimizing seven-pass compiler will do you little good if the language you choose doesn't suit your needs.

Take microForth, a very flexible language that permits you to define your own commands in terms of basic operations. That's great when you want to replace a complicated relay chain by a μC. But you don't want to define all mathematical functions from scratch, if you're trying to solve differential equations.

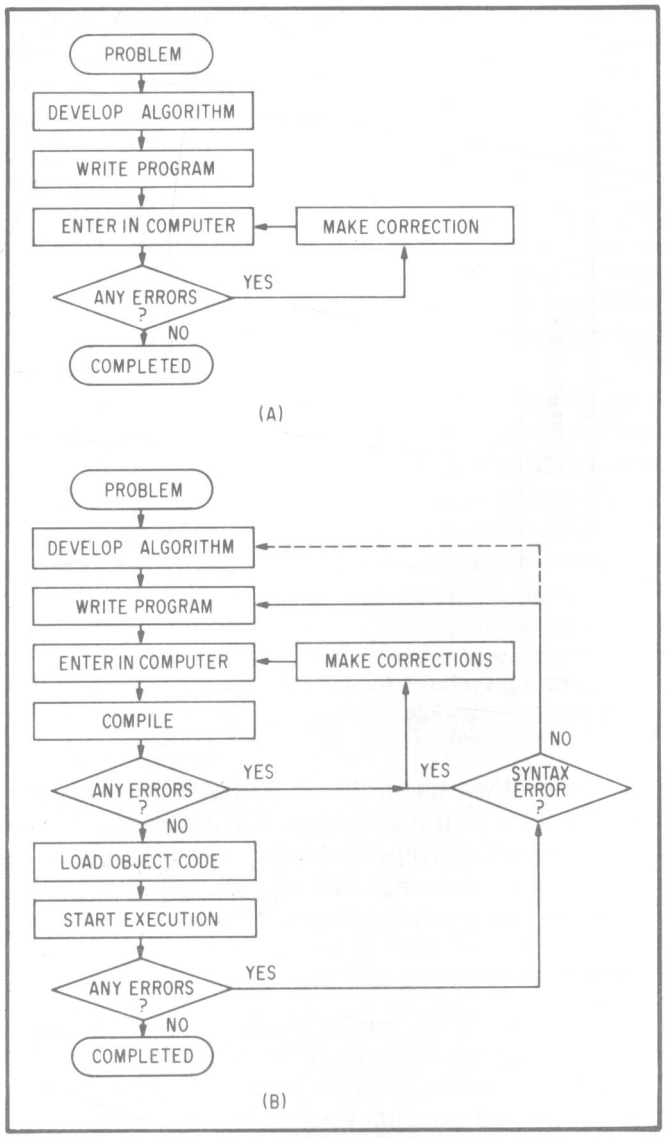
Before μCs, few engineers got involved with business programs—the bookkeeping kind. But now that small-business systems are the rage, you may well have to buckle down and learn Cobol. It's great for setting up a ledger, but it may take you a day to describe Ohm's law in Cobol. Higher-level languages are often quite specialized, and likely to remain so.

Halfbreed or new species?

Assembly language is efficient, and high-level languages are easy to use. So why not mate them, and see what you get?

It's been done, and the results are usually called systems-implementation languages, or SILs. For example, MOV D, S turns into D=S, JMP AB becomes GOTO AB, and RXY may turn into IF (XY) THEN RETURN.

A SIL adds another software interface, and you'll



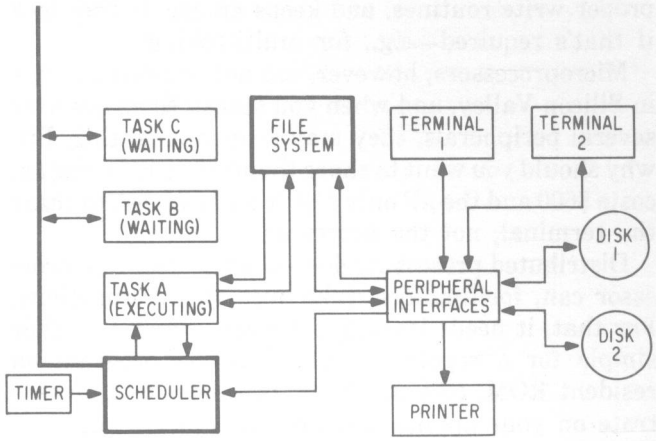
2. **Program development proceeds differently** with an interpreter (a), and a compiler (b). In the second case, error correction requires several stages.

write lengthier source code, but look at what you have accomplished: You'll isolate the μP from the user, and you pay practically no speed penalty.

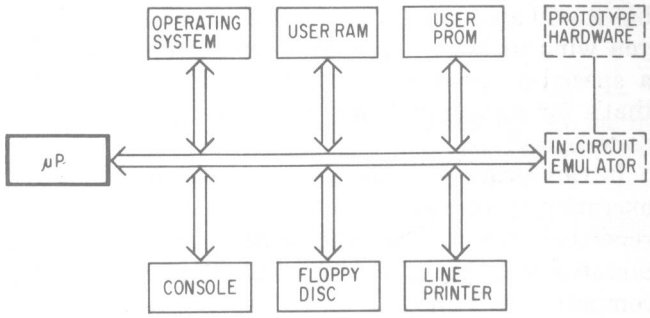
SILs can also make a big difference when you consider the over-all cost of your system. In Fig. 1, higher-level languages are shown to be more cost-effective than assembly language below the crossover point H. But in comparison with a SIL, assembly language loses a lot of territory—the crossover moves down to point S.

Such SILs as MUPRO's BSAL/80 (see ED No. 13, June 21, 1978: "Operating Systems Enhance μCs") are often tailored to a specific μP, with each SIL statement corresponding to a mnemonic. Yet programs get easier to write and debug. And if you code your application program in the same SIL as your operating system's, you don't have to jump back and forth when you check out your programs at the terminal.

Of course, if you buy a complete μC board, your choice of operating systems may be limited. Or, you



3. **A simple operating system** is just a collection of utility routines, controlled by a scheduler.



4. **To develop μC software** you need at least a terminal and a μP with the right operating system. In-circuit emulators then help you debug the hardware.

may not even need one. You can punch in a small program on a hex keyboard, or load it from a cassette, or put it in ROM.

So what does an operating system do? It's essentially a collection of utility routines (Fig. 3). In other words, to program a "naked" μP for keyboard input, you have to tell it in detail, character for character, to watch for a flag, look at a byte, put it in a register, wait for a moment, compare the new data with the byte in the register and—if they are the same—file that byte away, and set another flag. Rather tedious, you'll find.

I/O routines are, however, the lowliest parts of an OS. If you use a disc, one of the routines provides disc I/O, and the OS itself is stored on the disc. The combination is called a disc operating system, or DOS. Of course that part of the system that permits loading from the disc must be in ROM, and it's usually known as a bootstrap loader. As ROM prices keep dropping, larger, more sophisticated operating systems, including compilers, editors—even math libraries—become practical in ROM form.

The wizardry of OS

An OS is nothing without its scheduler. It keeps track of the executing program, watches for interrupts from peripherals, channels program outputs to the

proper write routines, and keeps an eye on the clock if that's required—e.g., for multitasking.

Microprocessors, however, are not the fastest guns in Silicon Valley, and when you spread them out over several peripherals, they may keep you waiting. But why should you want to share them? If a CRT terminal costs $600 and the μP only $10, it makes sense to share the terminal, not the processor.

Distributed processing does exactly that. One processor can, for instance, take over all I/O functions. For that, it needs its own software, which is rather simple for a repetitive task. Put the program on resident ROM, and the "real" processor can concentrate on your application, and at a faster clip.

If an operating system consists of utility routines, can't you put the whole OS on hardware? Yes, it's happening. Call it firmware, call it silicon software, but it still amounts to "hardwired" random logic, and you wind up with a general-purpose computer with a special-purpose computer built-in to run it. But that's for tomorrow. How do you get your software done today?

It can prove tedious to develop a sophisticated operating system for a μC on another μC. Compilation reportedly can take as long as 24 hours, but even 24 minutes may be intolerable. The solution: a larger computer. Anything a micro can do, a mini or a mainframe can do better.

Mainframes mimic micros

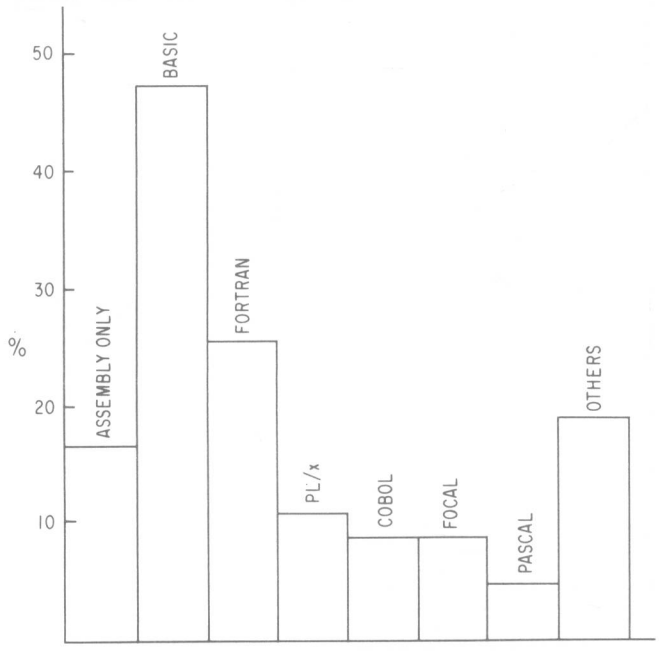
Properly programmed, a larger computer can easily mimic a micro—either at the μP's actual speed, or at a much faster clip. You can then use such a simulator to try out your programs—quickly.

If you're developing a μC operating system, all you have to simulate is the μP. But for applications programs, you also have to simulate the OS. If a mini —or time-shared mainframe—simulates an assembler, the setup is called a cross-assembler. For a compiled program you need a cross-compiler.

Cross software can save you not only precious time (because it runs faster), but also a great deal of money. Working on a large machine, a cross-compiler can, for instance, provide much more detailed diagnostics than a μP-based development system. And cross software is usually "universal" in that it simulates all the popular μPs, under software control.

Nevertheless, many designers prefer to develop software on a μP-based development system. One reason may be that such systems can usually double as general-purpose computers (Fig. 4).

So, microcomputer software means many different things to different people. The engineer who works with mainframes and mature minis rarely has to worry about operating systems. But with μCs he often does. Nor does he write much of his software in assembly (or machine) language. But with μCs he often must.



5. **Operating systems for today's μCs** are seldom limited to assembly language. Higher-level languages are acceptable to most, with Basic still strongly in the lead.

On the other hand, fewer and fewer OEMs are putting their own boards together. So the hardware part of your job keeps shrinking while the software section keeps growing. The signals are clear: Pure hardware designers are headed for extinction.

Place your bets

At this point, say you're ready to select the most suitable language for your μC project—you've studied several candidates at great depth. That's fine. But unless you find out what the rest of the world is doing, you may pick a language that's marked for obsolescence.

A recent ELECTRONIC DESIGN survey of microcomputer companies may help you with the choice. The companies identified themselves as marketing the following: μCs (65%), related hardware (63%), system software (56%), applications software (53%), development systems (49%), microprocessors (35%) and firmware (35%), while 33% offer some software consultation.

The companies that offer operating systems supply them on floppy discs (49%), ROM (46%), paper tape (17%) and cassettes (14%). Other media add up to another 10%. The sum is well over 100% because many vendors offer a choice of media.

And what about languages? Only 15% of the offered operating systems are limited to assembly language. Of the rest, 45% accept Basic, 25% Fortran, 10% PL/I-based languages, 7% each Cobol and Focal, 3% Pascal, and 17% other diverse languages (Fig. 5). But many respondents pointed out that Fortran compilers are in their pipelines, and several respondents are working on Pascal.

The vendors supplying applications software or offering a program library revealed that 59% of their programs are written in assembly language, 17% in Basic, 13% in Fortran, about 3% each in Cobol and PL/x, and 5% in other languages (Fig. 6). But don't let those numbers fool you.

Asked about new languages to watch for, most respondents mentioned Pascal, and a few pointed to PL/M-based languages. Portal, Casual, and C "also ran." But enthusiasm for the newcomers was tempered by complaints about poor documentation and support.

Other gripes about the current software crop may give you additional clues as to where—or where not—to place your own bets: "Not compatible—even within its family" was a farily common refrain. "Tied to short-lived hardware," "limited by 8-bit chips," or "geared for expensive development system" were also echoed. Others turned thumbs down on "limited I/O capability," "poor reliability," and "not enough variety," but one respondent complained about "too much variety."

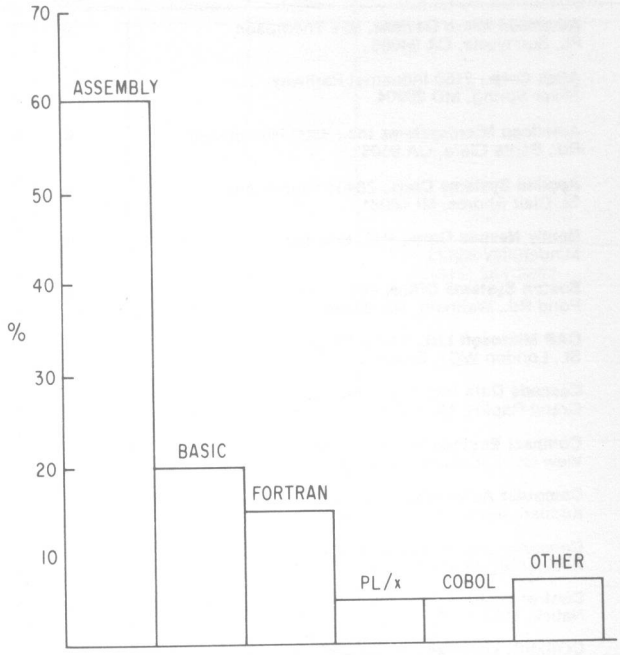
Many of the complaints stem directly from the youth of current μP programs. Software, like Beaujalais, improves with age—up to a point, that is. You'll find few programs written in Fortran I still around.

Today, software designers pretty much consider assembly language obsolete, although it will probably never disappear. Just try to rotate an array of words by one bit in Fortran. So it may be wise to pick a language for your project that permits machine code (or assembly) inserts. Many programs have critical passages that are executed over and over. A few mnemonics in the right place can cut execution time tremendously.

Another pet peeve that crops up wherever a few μP engineers gather is standardization. At least 30 dialects of Basic are used on μCs today, and hardly a week goes by without the announcement of another SIL to end all SILs. There are 20 languages for numerical lathe control alone, with two ANSI committies trying to standardize them. But don't despair—the country's richest customer is also clamoring for standardization. The U.S. Department of Defense has already settled on a test language, Atlas, and two final-round contractors (Honeywell and Stanford Research Institute) will soon be slugging it out for a general-purpose μC language.

Whoever the "winner" may be, it's generally assumed it will be based on Pascal—a fairly recent import from Switzerland—that is about to stamp out Fortran at American universities as well. Pascal (named after the French mathematician) is a descendant of Algol and was developed under the leadership of Niklaus Wirth (ETH, Zurich, see bibliography).

Pascal's main innovation is a variety of structuring methods for data, especially programmer-definable types. Structured programming is definitely in. Both compilers and interpreters for Pascal have been developed for many mainframes and minis, and are now in the works for the major μCs.



6. **Libraries of μC application programs** reflect the development of the hardware: assembly language programs dominate by far. But that's bound to change.

Software is mostly paper

At this point, you may still think that "software" is synonomous with programs. If that were so, the end of a software project would be marked by a single printout, or cassette, or ROM package. It just ain't so.

Say you're told, "Here is a 4-k ROM with Charlie's OTB program. Before he retired he picked the Daily Double without fail. See if we can place a bet by tomorrow night." What would you do?

Chances are only the CIA can afford to crack a machine-language program. Even if you get a printout in Basic (without comments), you're probably better off starting from scratch than trying to unravel somebody else's program.

Documentation (at the very least, abundant comments) is essential, but not enough. Even more important is an application example—a TTY output from an actual working session. Say you've written a circuit-analysis program that accommodates five complex circuit elements. Your program might *run* on another computer, but come up with erroneous results because that machine looks only at your real inputs and disregards the imaginary ones. Only if you get the same results can you be sure a program works on a different machine.

But the paperwork shouldn't start with your final report. Only a genius can sit down at the keyboard and compose good software. The toughest part in software design is algorithm development, leading to a detailed flow chart. Coding then comes almost automatically.

Company	Micro comp.	Systems software	Applic. software	Custom software	Firm-ware	User library contact and telephone number
Advanced Micro Devices, 901 Thompson Pl., Sunnyvale, CA 94086.	●	●			●	
Altek Corp., 2150 Industrial Parkway, Silver Spring, MD 20904.					●	
American Microsystems Inc., 3800 Homestead Rd., Santa Clara, CA 95051.	●	●		●		
Applied Systems Corp., 26401 Harper Ave., St. Clair Shores, MI 48081.	●	●	●	●	●	M. W. Wyrod (313) 779-8700
Bently Nevada Corp., P.O. Box 157, Minden, NV 89423.	●	●	●		●	
Boston Systems Office, 400-1 Totten Pond Rd., Waltham, MA 02154.	●	●	●	●	●	
CAP Microsoft Ltd., 14-15 GT James St., London WC1, England.		●	●	●	●	
Cascade Data Inc., 6300 28th St., S.E. Grand Rapids, MI 49506.	●	●	●	●		Deane Ledsworth (616) 942-1420
Compact Engineering Inc., 1088 Valley View Ct., Los Altos, CA 94022.			●	●		
Computer Automation Inc., 18651 Von Karman, Irvine, CA 92713.	●	●	●		●	
Computer Inquiry Systems, 516 Sylvan Ave., Englewood Cliffs, NJ 07632.	●	●	●	●		
Control Logic, 9 Tech Circle, Natick, MA 01760.	●	●				D. Tingley (617) 665-1170
COSMIC, University of Georgia, Barrow Hall, Suite 112, Athens, GA 30602.			●			Mostly large programs (404) 542-3265
Data General Corp., 15 Turnpike Rd., Westboro, MA 01581.	●	●	●	●	●	Dale Silva (617 366-8911
Digital Equipment Corp., 146 Main St., Maynard, MA 01754.	●	●	●			DECUS (617) 493-5086
Dynage Inc., 1331 Blue Hills Ave., Bloomfield, CT 06002.	●	●	●	●	●	
Dynatech R/D Co., 99 Erie St., Cambridge, MA 02139.		●	●		●	
Electronic Memories and Magnetics, 12621 Chadron Ave., Hawthorne, CA 90250.	●	●				
Evans & Sutherland Computer, 580 Arapeen Dr., Salt Lake City, UT 84108.	●	●				
FORTH Inc., 815 Manhattan Ave., Manhattan Beach, CA 90266.		●		●		
Futuredata Computer Corp., 11205 S. La Cienega Blvd., Los Angeles, CA 90045.	●	●	●		●	
General Automation, 1055 S. East St., Anaheim, CA 92805.	●	●	●	●	●	(305) 485-8270
GNAT Computers Inc., 7895 Convoy Ct., Unit 6, San Diego, CA 92111.	●	●				
Harris Semiconductors, P.O. Box 883, Melbourne, FL 32901.	●	●				
Heurikon Corp., 700 W. Badger Rd., Madison, WI 53713.	●	●	●	●		
Hewlett-Packard, 1501 Page Mill Rd., Palo Alto, CA 94304.	●	●	●			Several libraries-contact field engineer
Hodge, Taylor & Associates Inc., 1161 Tustin Ave., Orange, CA 92667.	●	●	●	●	●	
Hughes Solid State Div., 500 Superior Ave., Newport Beach, CA 92663.	●	●			●	
IMSAI Mfg. Corp., 14860 Wicks Blvd., San Leandro, CA 94577.	●	●			●	
Infinite Inc., 1942 Wavely Pl., Melbourne, FL 32901.	●	●	●	●	●	
Information Control Corp., 9610 Bellanca Ave., Los Angeles, CA 90045.	●	●				
Intel Corp., 3065 Bowers Ave., Santa Clara, CA 95051.	●	●	●		●	INSITE (408) 987-8080
Interactive Science Corp., 60 Brooks Dr., Braintree, MA 02184.		●				
Interdata, Div. of Perkin Elmer, 2 Crescent Pl., Oceanport, NJ 07757.		●	●		●	Alma Efthyvoulou (201) 229-4040
International Computer Products, 2925 Merrell Rd., Dallas, TX 75229.		●		●		
International Management Services, Inc. 215 Oak St., Natick, MA 01760.			●	●		

Company	Micro comp.	Systems software	Applic. software	Custom software	Firm-ware	User library contact and telephone number
Intersil, Inc., 10710 N. Tantau Ave., Cupertino, CA 95014.		●				
Kone Oy, Instrum. Div., PL2, 02327, Espoo 32 Finland.	●	●	●			
Landis & Gyr Co., Zug Central Lab., Zug, Switzerland.		●				
Microcomputer Associates, P.O. Box 304, Cupertino, CA 95014.	●	●			●	
Microgamma Systems, Inc. 413 Pattie, Wichita, KS 57206.	●	●	●	●	●	Dave Gilbert (316) 264-2864
Midland Standard, Inc., P.O. Box 38 (603 E. Chicago) Elgin, IL 50120.	●	●	●	●	●	
Milertronics, 303 Airport Rd., Greenville, SC 29607.	●	●		●	●	
Millennium Information Systems, 19020 Pruneridge, Santa Clara, CA 95050.	●	●		●	●	
Monolithic Systems Corp., 14 Inverness Dr. E., Englewood, CO 80110.	●	●			●	
Mostek Corp., 1215 W. Crosby Rd., Carrollton, TX 75006.	●	●			●	
Motorola Integrated Circuits, 3102 N. 56th St., Phoenix, AZ 85018.	●	●	●		●	Jeannine Middleton (602) 244-6454
Multisonics Inc., 6444 Sierra Ct., Dublin, CA 94566.	●	●			●	
MUPRO Inc., 424 Oakmead Parkway, Sunnyvale, CA 94086.	●	●			●	
National Semiconductor, 2900 Semiconductor Dr., Santa Clara, CA 95051.	●	●	●		●	Georgia Marszalek (408) 737-6181
NEC Microcomputers Inc., 5 Militia Dr., Lexington, MA 02173.	●	●		●	●	
Pieper Electric, Inc., Automation Controls Div., 5070 North 35th St., Milwaukee, WI 53209	●	●		●	●	
Process Computers Systems Inc., 750 N. Maple Rd., Saline, MI 48176.	●	●		●		
Processor Technology Corp., 7100 Johnson Industrial Dr., Pleasanton, CA 94566.	●	●	●			
PRO-LOG Corp., 2411 Garden Rd., Monterey, CA 93940.	●				●	
RCA Solid State Div., Route 202, Somerville, NJ 08876.	●	●			●	
Realistic Controls Corp., 404 W. 35th St., Davenport, IA 52806.	●	●	●	●		
Research Technology Inc., 4700 Chase, Lincolnwood, IL 60646.				●		
Rockwell Electronic Devices, 3310 Miraloma Ave., Anaheim, CA 92803.	●	●		●	●	
R 2 E, S. A. De Courtaboeuf 91403 Orsay, France.	●	●				
Siemens S. A., Chaussee de Charleroi 116, 1060 Bruxelles, Belguim	●	●	●			
Signetics Corp., 811 E. Argues Ave., Sunnyvale, CA 94086.	●	●	●			
Spantronics Engineering, 702 Bowling Green, Moorestown, NJ 08057.	●			●		
Teledyne Geotech, P.O. Box 28277 Dallas, TX 75228.	●	●	●	●	●	
Texas Instruments Inc., 8500 Commerce Park Dr., MS 653, Houston, TX 77036	●	●		●	●	
Vertrol Data Systems Inc., 2500 13th Ave., Vero Beach, FL 32960	●		●	●		
Warner & Swasey Computer, 7413 Wash. Ave. South, Minneapolis, MN 55435.	●	●	●	●		
Wintek Corp., 902 N. 9th St., Lafayette, IN 47904.	●	●		●	●	
Worthington Instruments, 1897 Red Fern Dr., Columbus, OH 43229.	●	●		●	●	
Wyle Laboratories/Computer Products, 3200 Magruder Blvd., Hampton, VA 28666	●	●	●	●	●	
Zilog Inc., 10460 Bubb Rd., Cupertino, CA 95014.	●	●			●	Bruce Weiner (408) 446-4666
Zonic Technical Laboratories, 8927 Rossash Rd., Cincinnati, OH 45236.	●	●	●	●	●	