ISBN 0-916688-04-6
$11.95

The Best of BYTE

Volume I

Edited by
David H. Ahl
and
Carl T. Helmers, Jr.

The Best of Byte
Volume 1

ORDERING INFORMATION

Additional copies of *The Best of Byte—Volume 1* are available for $11.95 plus $1.00 postage and handling (USA) or $2.00 (foreign) from:

Creative Computing Press
Dept. BB-1, P.O. Box 789-M
Morristown, NJ 07960

Subscriptions to *Byte* magazine are available for $12.00 per year from:

Byte Publications
70 Main Street
Peterborough, NH 03458

The Best of Byte
Volume 1

Edited by

David H. Ahl
and
Carl T. Helmers, Jr.

Creative Computing Press

Morristown, New Jersey

OTHER BOOKS FROM
CREATIVE COMPUTING PRESS

The Best of Creative Computing - Volumes 1 and 2
101 BASIC Computer Games
Artist and Computer
The Colossal Computer Cartoon Book
Amazing, Thrilling, Fantastic Computer Stories

THE EDITORS

David H. Ahl is the Founder of *Creative Computing* Magazine and Creative Computing Press. He was one of the early proponents of computers in education and has written numerous books and articles on the subject. He has also been associated with AT&T, Digital Equipment Corp., and Management Science Associates.

Carl T. Helmers, Jr. is and has been Editor of *Byte* Magazine since its founding in mid 1975. Prior to that he published *ECS* Magazine, a "home brew" hobbyist computer magazine. He is a well-rounded expert in all aspects of hobbyist computers — hardware, software, and applications.

First Printing — March 1977

ISBN 0-916688-04-6
Library of Congress Catalog Card Number: 77-71270
Printed in the United States of America

Preface

On a recent Saturday I invited a group of some twenty associates and friends to my house to discuss the future direction of hobbyist computing (Naturally, my objective was to determine how *Creative Computing* magazine and press should be positioned in the market). These people represented a diverse spectrum of the hobbyist computer movement — people from a major manufacturer which 18 months before consisted of two people in a basement workshop, representatives from a large 500-member computer club that two years earlier did not even exist, a department chairman from an Ivy League University which had just graduated their first group of Computer Education majors, a salesman from a retail computer store which had just moved for the second time in six months to larger quarters, hobbyists from virtually every walk of life, and, of course, people from the publishing side of things.

What becomes quickly apparent is that the hobbyists who jumped in two years ago, or one year ago, or even six months ago are much further along than people entering today. Nevertheless, vast hordes of people continue to enter the hobby daily. Thus the magazines in the field are caught between a rock and a hard place — should a magazine progress along and continue to present challenging material to its earlier and technically more sophisticated subscribers? Or should it endeavor to bring the newcomer up to speed with primer-type material and risk losing its more knowledgeable readers? The magazine that attempts to do both is a bit like the boy in the Charles Addams cartoon who is sliding down a bannister that at the bottom of the stairs turns into a razor blade.

One solution to this dilemma is to offer back issues to later subscribers so they can get a quick cram course of what transpired before they subscribed. Better yet is a book, like this one, of the best material from previous issues of the magazine.

For those readers who don't know *Byte* magazine, it was one of the earliest entries in the hobby computer field. Some early issues carried the notation on the cover: "Computers - The World's Greatest Toy". However, before long it became apparent that hobbyists look at their computers as much more than just a toy; *Byte* is now dubbed "The Small Systems Journal" which better reflects the comprehensive scope of home computerists.

Under the direction of Editor Carl Helmers and Publisher Virginia Peschke, *Byte* not only reflects and responds to the enormous diversity of computer hobbyists, but sets the pace in innovation and new development. Naturally most hobbyist's first concern is getting a system built and running — the sections on "Computer Kits" and "Hardware" address this need. However, without software a computer might as well be a boat anchor, hence there is an equally large section on "Software." The questions of what's coming, how does it work, and what do you do with it are covered in the sections on "Opinion", "Theory", and "Applications".

Volume 1 of *Byte* magazine includes sixteen issues from the charter issue in September 1975 through December 1976. This book includes material from the first twelve issues. (Does this mean there will be a Volume 1.5? Quite probably.)

It's an impressive collection. Although I was a charter subscriber to *Byte* there were many articles I didn't read until I put them together in this volume. I couldn't help but be awed with how far hobbyist computing progressed in one short year. One can only wonder what the future holds in store. In my mind computers are truly different from any other hobby. First of all, they are not an end in themselves but rather a tool for accomplishing literally thousands of things. Second, computers are an intellectual tool, not simply a hammer or a lathe however useful they might be, but a fascinating, powerful, creative, mind-expanding, tool. The cliche is that "the sky is the limit", but I look beyond that. The cybernetic revolution has begun.

March, 1977 — David H. Ahl
Morristown, New Jersey

Table of Contents

Opinion

The Shadow, Buck Rogers, and the Home Computer

by
Richard Gardner
Box 134
Harvard Square
Cambridge MA 02139

A computer at home? Ask many present day computer systems people what they'd do with a home computer and you'll get the old silent treatment in return. But all that indicates is a lack of imagination. A large part of the BYTE philosophy is the discovery of applications areas through the imaginations and practical results of readers. Richard Gardner supplies us with a "Gee Whiz" article on potential applications areas to get things in motion a bit. Richard has extensive computer applications experience including one stint working for the Children's Museum in Boston, creating interactive computer oriented exhibits. Eventually, many of the systems ideas Richard mentions in his article will appear as practical plans and programs in the pages of BYTE — as developed and described by our readers. If you'd like an interactive meeting of the minds on possible uses and ideas, Richard invites correspondence from readers. . . . CARL

Ah yes! It conjures up visions of an earlier day, many years ago, when Mom, Pop and the kids sat around that newfangled gadget, the radio, and listened to "The Shadow" and "Buck Rogers."

Flash forward to the future, right now! Again, we see Mom, Pop and the kids sitting around that newfangled gadget, the computer, balancing a checkbook, converting a four servings recipe to seven, and playing tic-tac-toe. Not very exciting things to do with a computer, you say? Well, you're right. But let's see if we can do something to make it at least as exciting as old-time radio.

We mentioned three applications for a home computer:

1) checkbook balancing
2) recipe converting
3) game playing

For starters Mom and Pop should have a program for collecting and summarizing all their financial data, on a daily, monthly and yearly (for your "friend" and mine, the IRS) basis. A family will be more secure by knowing the state of its financial affairs. You will want to

compute interest for different purchase plans, and balance the checkbook.

Moving on to a subject close to my heart (just below, and a little to the right) — food. Almost anyone can convert 4 to 7 servings — just double it and feed the leftovers to the dog, or give it to a charitable organization (tax deductible, of course). What you really want to know is whether everyone got enough nutrients (vitamins, minerals, protein, etc.) from what they ate today. Hint: it can be done. I know of two people who started a small company to do it.

On to fun and games — hundreds of game playing programs have been written (I invented one called YOUGUESS) for all sorts of computers and languages. You should have them all. It will win friends and influence neighbors, if you'll pardon the pun.

I'd say that's at least as exciting as old-time radio. Good, but we can do much better. Let's consider three things:

1) Today's computers are very fast. The applications we've mentioned *might* take one hour of CPU time per day, at the very most. So what do we do with the other 23+ hours?

2) There are lots of computers in the world, and they can talk to each other.

3) Computers can hear, see, feel, smell and touch.

Keep these things in mind as we consider what might be called economic, personal and educational applications for the home computer.

Computerizing the Home

Since your computer won't be doing anything most of the day why not put it to work:

1) *Heating and air conditioning control.* Optimize increases and decreases in the inside temperature to minimize energy use. Open and close curtains on windows to use the sun's energy or keep it out.

2) *Security.* While you're at home or away, monitor the opening and closing of windows and doors. Automatically telephone the police with a recorded message when you're gone or at home. Monitor the use of your swimming pool — sound an alarm when the pool is in use and nobody's in the lifeguard seat. Fire monitoring equipment can be located in many places and sound an alarm long before you might smell or see smoke. The fire department can be called automatically with another recorded message.

These applications will make use of photocells, theramins (motion sensing devices), heat sensors, contact switches, smelling devices (like those used by the Defense Department in Vietnam to smell passing elephants and tigers). Eight bits might be used to represent a temperature range of 256 degrees. 100 degrees would be adequate for most locations. One analog to digital converter could be used for other analog inputs, such as from a photocell. A digital to analog converter would generate voltages to be used by motors and other mechanisms.

Using a Symbol Table to Improve the Food Table

Most people in America have a poor diet in spite of the fact that we have more food of a better quality and variety than any other country. So I consider the following to be important uses for a home computer:

1) Selection of foods on a seasonal basis to reduce cost and improve quality. A program for doing this would run for a year and use a data base for your area (to take advantage of local produce). A second data base would be programmed for widely available foods and when they are best and cheapest.

2) A menu building program to take advantage of the above system but with the intention of increasing variety and maximizing nutrition.

3) A shopping guide to take advantage of local food supplies by indicating the best one or two markets from which to purchase your food. This data base might be maintained by some person in your locality — and then rented on a per use basis. No sense in everyone typing in today's price for cumquats. Perhaps the *New York Times* will eventually computerize its cookbook, plus thousands of other recipes, and allow the public to access this data base via a personal computer.

Since your computer won't be doing anything most of the day . . .

This application, like others mentioned, would use the telephone system — the world's largest computer. I can see it now. The kids get home from school and ask, "What's for dinner, Ma Bell?"

The Bottom Line Isn't Always an End Statement

Or, how to profit from your home computer:

1) Income management, as previously mentioned, but with the help of another computer. Several computer companies that do nothing but figure taxes (for you know who!) already exist. Eventually they will allow your computer to call their computer. Your computer shovels in a year's data and out pops a tax form with all the right numbers. You might think it easier to do your own programming, but remember that you can't write every program you will want to use. In addition, these companies have staffs that do nothing but make program improvements and changes required by the IRS. What person in his or her right mind could possibly keep track of a myriad of new rules from the IRS?

2) Play the ponies or the puppies? An obvious use for your computer. Again, use a data base compiled by some local eager beaver. Perhaps you'd be charged a small fee for accessing the day's statistics. Perhaps you have a data base or program to trade.

3) Then there's always the world's biggest daily crap

game — the stock market. A company in Philadelphia will charge you $300 a year for a small numeric terminal and 24 hour a day access to their stock data base. You key in the number of a stock and out pops the high, low, average, etc. Your computer could make one call after each trading day, collect the stock data you're interested in, hang up, and then determine if you should buy, sell or hold. The decision making could be done by your program or one being rented from a stock market wizard you know.

4) I mentioned how a computer could be used to optimize the purchase of food. This principle applies to any commodity whose price and quality changes during the year: clothes, home furnishings, gifts, transportation, even housing. Some local person, or you, could create the necessary commodity and price data bases, then use or rent them.

Remember! There is a host of areas for small business activities using your home computer as a tool of the trade. All it takes is imagination, a bit of digging into the wants and concerns of your neightbors, and the programming of your computer.

Six Munce Ago I Couldn't Even Spell Computer Programmmer . . .

Computers are good for keeping you in touch with the world. For example:

1) The *New York Times* has a computerized data base of all its back issues — currently accessible to the general public, for a fee. The cost will probably go down to the point where you might program your computer to query the *Times* data base and retrieve front page stories, financial page stories, or any story that contains a keyword or some combination of keywords. This would be done early in the morning and read by you at breakfast time.

2) Your local university or high school might have a computer with courses that can be taken via a remote terminal. Many universities already give some courses using only this method.

3) The Children's Museum in Boston will eventually allow you to call their computer, via a terminal or computer, and access a data base of cultural, educational, and social events in the Boston area. Your computer might call theirs once a day to learn what's new or learn about a particular type of event.

Computers As Toys

Computers are probably the greatest toy ever invented. Here are some examples of how you can play around with yours:

1) It has been rumored that 50%, or more, of the computer time used at MIT is used to play Space War — the Grandpa of computer games! Your computer, a TV set, a few buttons and switches and, presto — Space War! Or ping-pong, or driving down a road, flying and landing an airplane, landing on the moon, chess, checkers (you can play these games in Boston with the Children's Museum computer).

2) Toys that play with you — like robots. The Boston Children's Museum has a robot that was built for about $200. Mass production of a special chip and board will bring that cost down. Then the biggest cost will be the Meccano Set (like an erector set, only better), which can be used to build almost any sort of mechanical device. How about a robot to do housework?

3) The ultimate fun, though, is to write your own programs to do all these things! Kids, and adults, will play only so many games of tic-tac-toe — then they want to know how it works. Help them write their first BASIC program . . . and they're likely to be hooked for life! Eventually programming will include a broader range of input/out devices such as the previously mentioned buttons and switches, photocells, microphones, etc. This will lead to the applications just discussed, and who knows what?

These are just some of the possible applications for a home computer. All of them might not be reasonable or practical things to do but they should set you to thinking.

As future issues of BYTE unfold, the Gee Whizzers applications will lead to practical articles on the software and specialized peripherals needed to implement some of these ideas.

Carl Helmers

The State of The Art

If there is one facet of the small computer field which is its most exciting, that is probably its rapid change and evolution unfolding before all us users of the technology. The fact that a magazine such as BYTE can even exist (let alone get its enthusiastic reception) is evidence of the considerable changes which have occurred in the home computer field over the past year or two. Any attempt such as this to characterize the current "state of the art" is doomed to rapid obsolescence. Be that as it may, I won't let that deter me from characterizing the field as I see it now.

Just what is this "art" that I'm talking about? When I talk about art in this sense, I mean the body of technological know-how available for personal computing plus the attitudes and abilities of the people who use this know-how. An analogy or two: The state of the art in a form such as painting reflects both the latest developments in the pigmentation materials field and the creative talents and attitudes of the people who use this technology for expressive purposes. The state of the art in music is a combination of the technology of music production — traditional to electronic/digital — plus the aesthetic and creative tastes of the musicians and composers who use the technology. So it is as well with computing. There is the technological state of the art as it exists — a transient thing at present — together with the creative uses to which people such as you or I put these wonderful technological devices.

Any attempt to specify the state of the art in this field is doomed to practically instant obsolescence . . .

A Recent State of the Art . . .

A few years ago, the state of the art in hardware was pretty primitive — in other words, one had to be a really persevering person to get something in computing which worked and cost less than $1000. To give you an example, I got a call from Dick Snyder of Chelmsford, Mass., shortly after BYTE #1 came out. (See Dick's letter in the letters column of this issue.) As a result of our conversation, I stopped at Dick's house on the way back from Peterborough one weekend in August and took a look at his pre-microcomputer home brew computer, a really beautiful piece of work. He had completely designed and built — in 1972 and 1973 — a miniature 4-bit computer with 256 nybbles of memory using the Data General NOVA minicomputer as his inspiration. He built the machine using painstakingly accurate soldering with a miniature iron, sockets for over 170 integrated circuits, and a very compact housing. The most unusual feature of all was the use of water cooling to keep his 16 7489 memory chips cool (said water cooling consisting of plastic bag baby bottles filled with water and sealed with rubber bands). Yet it works! And — he has built up quite an impressive array of software for his one-of-a-kind machine, including a very appealing simulation of a priority-driven real time operating system with three tasks in the queue. The entire program for this simulation is done in 256 nybbles (half-bytes) of memory with the 16 instructions of his design. The result is an impressive changing display of marker patterns in his front panel lights as the various tasks swap in and out of execution. Dick Snyder's machine is the state of the art, circa 1972-1973, to a large extent — microcomputers were not yet widely available to the general populace of personal computing hackers. Dick tells me that he spent about $600 on the parts of his computer at 1972 prices for SSI and MSI TTL integrated circuits.

But now, in 1975 after the first wave of 8008 computer kit products and the rising tide of the "first generation" personal computer systems, that same $600 can buy a lot more function. In 1975 we saw the introduction of the MITS ALTAIR — which turns out to be a very good computer after initial slow deliveries due to unanticipated demand — and a host of new machines such as Bill Godbout's PACE, the SWTPC 6800 kit, the MITS 6800 kit and several other systems.

The Benchmark of a Small Computer System

In the engineering and software professions, it is often common to dream up "benchmarks" to help in the evaluation of systems. This term, benchmark, was adopted by systems engineers from its original use in the field of geodetic surveying. A geodetic survey benchmark is a permanent marker set "out in the field" (literally) at known locations during the course of the survey. If you clamber to the top of Mt. Chocorua in New Hampshire, as I sometimes do, when you get to the top you will find a little metal plate giving elevation, longitude and latitude information. This is the benchmark for the mountain's peak. Well, the benchmarks used for computer systems are a little bit less concrete than a metal plate on a mountaintop, but serve the same purpose: They provide a reference point for comparison.

A common benchmark which has been used in the past to evaluate computer systems (and compilers) is the "standard set of programs". In this method of benchmarking a system, the potential user of the system picks a set of "typical" applications programs and has them implemented and measured in operation on several different systems. This is a fairly quantitative and seemingly accurate method which is widely practiced in the information systems

industries. The measurements made for comparison include "through-put" (processing per unit time), high level language efficiency, memory requirements, etc. But this sort of a measure is perhaps a bit too complicated for the home computer context. For one thing, the applications are known only generally. Second, this is the type of study which takes a large amount of time and access to various competitive systems. And, if you read the trade journals, the results are often controversial anyway, since each manufacturer will claim that the benchmarks he provides will prove his machine better than all the rest. Picking the "ideal" small computer system still requires a benchmark, but I suggest it is not a particular program, but a *capability*.

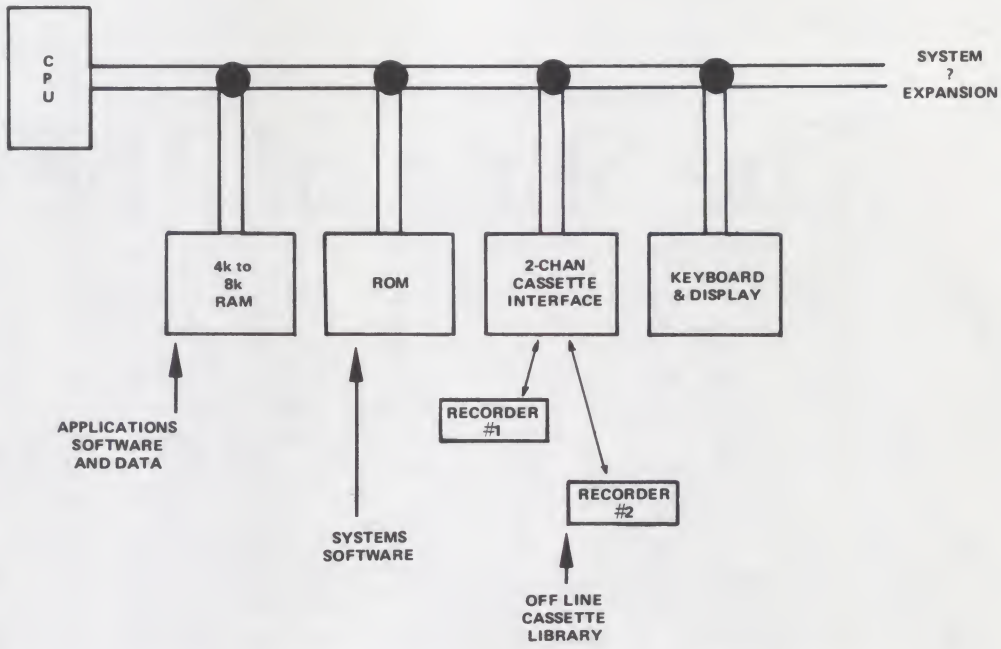


Fig. 1. The Complete Low Cost Computer System (circa September 1975). This diagram shows the major components of a typical low cost computer system – which should total up under $1000 depending upon manufacturer and details of design. At the time this editorial is written, several kit manufacturers meet this functional benchmark at prices well under $1000. As time goes on the improvements of mass production should drop the average price of such systems.

Capability – the Benchmark of a Small System

We all know that in broad terms, the benchmark computer system, as any computer system, must include several major components: a processor, memory, a mass storage medium, an interactive operator's terminal and systems software. I pick this list in part to illustrate a typical computer configuration and in part to allow programming of a benchmark capability:

> A small computer system which meets the benchmark standard will be able to *interactively edit* a mass storage file of input data with operator commands, producing a second mass storage file as output. This will be achieved in a system costing at most $1000 initially.

The system diagram of the benchmark computer is shown in Fig. 1, as it is implemented in the current state of the art. The components of the system are chosen with the editing function in mind, since accomplishing such an edit capability means the machine can be programmed for almost any other personal computing use. Peripherals that enhance the function are of course desirable and will help to personalize your system, but these functions represent the bare minimum without added cost of special purpose peripherals.

The CPU: Which One?

In Hal Chamberlin's article in BYTE #1, the relative merits of three computer designs were covered. In BYTE #3, Dan Fylstra covers a comparison of two additional designs. There is a large variety in the types of CPUs available to home brewers and kit builders – ranging from the 8008, 8080, 6800 and 6501 8-bit micros, to the 16-bit IMP and PACE micros, to commercial 16-bitters such as the LSI-11 and NAKED Milli products – and on into the never-never-land of custom designed microcoded MSI computers implemented by individuals (and also soon to be announced in product form by one manufacturer of kits). There is a large element of personal taste involved in the preference of particular instruction sets, and there is also the matter of efficiency for particular classes of programs. Whatever the CPU you use, it is a definite requirement of the system. I guarantee you that any one of the 8-bit or 16-bit microprocessors currently being packaged and sold as kits will be adequate to pass this benchmark test, although you may have to write the Editor program yourself.

RAM Memory – How Much?

The CPUs of the conventional microprocessors – kit or home brew implementations – create an output called a "data bus" which is used for exchanging information with everything else in the system. The data bus is the "spinal cord" of the computer's nervous system. This bus concept typically includes 16 bits of buffered address lines and several bus control information lines *as well as* the 8 or 16 bi-directional buffered data lines. The address space of the typical contemporary micro-computer's architecture is usually 16 bits worth or 65,536 possible memory locations. In the usual system most of these locations will

Picking your ideal computer system requires a benchmark – which I suggest is not a particular program but a *capability*.

Continued on page 88

be unused. In general, as many of these locations as you can afford should be filled up with random access memory chips, which, experience has shown, people are always able to use up in programs. Sooner or later you will find yourself limited by the constraints of small memory! For the benchmark system, the minimum random access memory should be 4k (4096) 8-bit bytes or 2k 16-bit words. A preferable number is 8k bytes or 4k 16-bit words.

ROM Systems Software?

How do I get my first programs into memory after turning on power? The answer to this question is the method of "bootstrapping" or "initial program loading" (IPL) which is used by a computer. Early in the minicomputer game, technology of computing was at a state where the principal bootstrapping method was a set of front panel switches which addressed memory locations and allowed the programmer to put in short programs by hand.

With the advent of the new high density ROM integrated circuits, it is now possible to provide the convenience of an automatically bootstrapped system through systems software which is cast into the concrete form of an ROM device.

Many of the kit suppliers I have talked to are either currently supplying or intending to add this ROM systems software feature. Initially, the programs which are "built-in" tend to be fairly standard "control panel" type routines which use a terminal (Teletype or television typewriter) for a set of simple commands. Later – with inputs from users regarding desirability – you can expect to find prepackaged assemblers and high level language compilers/interpreters occupying major portions of the address space available in typical microcomputers. This will make the systems software feature even more versatile.

Experience has shown that sooner or later you'll feel constrained by any size of memory – the greed of many programmers for more memory is unbounded!

Keyboard and Displays?

But of course. The interactive nature of an editor capability cannot be realized with a mere control panel. The same thing goes for most of the more interesting applications of the small computer. You will need a character-oriented display device and a typewriter style input – whether these be a TV typewriter or an old Baudot coded Teletype clunker is up to you. The typical programs will be controlled by keyboard commands and will produce outputs back to the display.

Cassette Tape Interfaces – Mass Storage Without Mass Dollars

Mass storage is a definite must item for the small computer system. But traditional industry peripherals tend to be expensive, starting at the low end with digital cassette drives and floppy disks at about $500-$800, and working upwards. The solution is to adopt an audio recording method which uses inexpensive ($50) cassette recorders and appropriate interfaces. This allows you to perform the editing benchmark function while keeping the total system cost low. I'll have more to say on this subject later in this editorial. A minimum of two such tapes is required for a decent editor, because one must be set to "read" old data, and the second *must be* set to "write" new edited data resulting from your changes. Three is a more desirable number still if you want to do "sort/merge" applications, but two will suffice for the editing benchmark.

Suppose Your Budget is Limited – Can It be Done in Stages?

What I have just described is the minimum necessary equipment for a fully functional implementation of the small computer benchmark capability, editing. Modularity rules in the computer world, however, so you can easily start out with less function and work up to the benchmark capability in time. You'll also probably end up exceeding this benchmark of hardware/software capability after a while; modularity does not stop at this level of function. The basic place to start is with a CPU – it'll not be much more than a blinking light box without peripherals, but that's enough to show that "it works" Then, you can add on the interactive keyboard/display of some sort, along with memory (presumably the ROM software came with the CPU). Finally, you can add on the tape interfaces and additional memory in order to arrive at the full benchmark capability. From then on, you can enhance the system with new peripherals and more memory until you end up with a very capable system which can run full BASIC, a decent systems programming language compiler, and all the games, practical applications and amusements you can dream up for the computer.

Could a Computer Take Over?

Ed Rush
PO Box 14369
Santa Barbara CA 93107

Just how ridiculous IS the idea of a computer deciding to take over the world and be its dictator?

Upon hearing this question, most people who are not computer oriented will laugh and say "That's only in science fiction stories." They will be much more likely to complain about "becoming a number," with everyone from the grocery store to the government wanting their number instead of their name.

Those who are more familiar with computers will laugh off the concept and charge it to paranoia due to ignorance. "A computer is little more than a lot of wires conducting currents here and there," they will say. "Besides, if it gets uppity you can always pull the plug."

However, that group of people who are both computer knowledgeable and fans of the art form known as science fiction, but more properly called speculative fiction, might ask "Can you always pull the plug? Could a computer really seize the reins of government? And if so, how?"

In trying to answer these last questions, let us first speculate on the capabilities the computer itself would have to have.

Super Computer

First, the computer system would have to be extremely powerful (in today's frame of reference). Considering the fact that computer technology is already far outstripping man's capability of harnessing it, a super computer is not hard to imagine in the not so distant future; perhaps even today in some secret government project.

While something on the order of 1000 computer circuits can now be stored in a cubic inch, only one such circuit would fit that space in 1960 and it took 20 cubic inches to hold one in 1950. A given number of programming instructions cost 1000 times as much in 1955 as in 1970, and probably 10,000 times today's cost, despite inflation. High speed computers now operate several thousand times as fast as they did in the early 1950s. Data storage capabilities are growing even faster. The capacity of an early 1970s system was a couple of million times that of 1955, and that is for a common large installation, not the maximum possible. The on line storage cost also shows a millionfold improvement since 1950 (Martin, James and Norman, Adrian R.D., *The Computerized Society,* pp. 9-14). Who is to say what 1980 will bring?

The next requirement is that this machine must be able to interact with changes in input from a multitude of input sources at once, a situation common to today's time sharing practices.

Such a machine must embody what is commonly called "artificial intelligence." That phrase is used hesitantly; since things which immediately provoke the description "artificial" are actually just natural materials rearranged by man. Intelligence is defined as:

> The capacity for knowledge and understanding, especially as applied to the handling of novel situations; the power of handling a novel situation successfully by adjusting one's behavior to the total situation; the ability to

"The first man to use a machine was the first of our primitive ancestors who picked up a rock to hurl at some passing animal or to crack open some edible nut. In the million-plus years since then, our machines have grown much more complex, but even in our modern era of computers, . . . their basic purpose remains the same: to serve man.

"Whether our machines truly serve us is a question much debated by science-fiction writers and other professional speculative philosophers. Does some essential quality go out of human life when it becomes too easy? Have our automobiles, telephones, typewriters and elevators sapped our vigor? Are we speeding into flabby decay because we have made things too easy for ourselves?

"And as our machines grow more able, when do they cross the boundary that separates the living from the unliving? Is it possible that we are building machines that will make humanity obsolete? Perhaps the day is coming when we ourselves will be rendered unnecessary, and our sleek successors, creatures of metal and plastic, will inherit the earth.

". . . Many a bitter attack on the encroachments of the machine age has been produced by a writer using an electric typewriter in an air-conditioned room, innocently unaware of the inner contradictions involved. We need our machines, but we fear them. . . ." Robert Silverberg, Introduction to *Men and Machines.*

> apprehend the interrelationships of presented facts in such a way as to guide action towards a desired goal. Psychologists still debate whether intelligence is a unitary characteristic of the individual or a sum of his abilities to deal with various types of situation. *(Webster's New International Dictionary of the English Language*, Second Edition, Unabridged, p. 1291.)

A machine with this capability would be an intelligence in its own right, not just an electronic mimic. It might take the form of a massive, immovable complex, or it might someday take form as a troop of man sized robots, or it might be a combination of these, with the latter as mobile extensions of the former.

Although Isaac Asimov has written extensively about the possibilities of robotics, most authors who have seriously considered a computer takeover have postulated the immobile complex. There are at least two good reasons for this assumption: First, such a machine would most likely be the first to have massive capabilities, and as such would most likely be far too big to move about. Second, it would undoubtedly require very heavy security as the most advanced piece of computer hardware in existence; protection not only from spies, but from vandals, intentional or otherwise. Examples of postulated massive complexes are HARLIE (Gerrold, David, *When HARLIE Was One*), Project 79 (Caidin, Martin, *The God Machine)* and Colossus (Jones, D.F., *Colossus: The Forbin Project).* The last two are built inside man made caves in the Rocky Mountains as the U.S. Air Force's North American Aerospace Defense Command (NORAD) is today.

Alternatively, if its state of development is not unique at the time, the system may simply have no reason to be mobile, as is the case with the HAL 9000 computer on board the Discovery in Arthur C. Clarke's *2001: A Space Odyssey.*

Ethics for Computers

Most Americans objecting to a computer dictator would do so on the basis that it is immoral for a person to have no say in the rules governing his life, and specifically for those rules to come from "cold logic" without the benefit of human sensibilities. True, the computer would probably have no morals, since morals are indeed artificial. Ethics, however, are a different kettle of fish. A computer could easily be imbued with a code of ethics, or an intelligent one might well develop one by and for itself. The most basic and significant such code of ethics was developed by Asimov in the early 1940s as "The Three Laws of Robotics" and has been used by many other authors since. It says:

> 1: A robot may not injure a human being, or, through inaction, allow a human being to come to harm.
>
> 2: A robot must obey the orders given it by human beings except where such orders would conflict with the first law.
>
> 3: A robot must protect its own existence as long as such protection does not conflict with the First or Second Law. (Isaac Asimov, *I, Robot,* p. 6.)

Can you always pull the plug?

The capacity of an early 1970s system was a couple of million times that of 1955 . . .

How could a finite assortment of nuts and bolts and wire take absolute control?

But, given these ethical restrictions, how could an intelligent computer set onto a course of world domination and justify it? The answer to the second part of this question lies in another: What constitutes "harm"? This is an aspect which has spawned much of Asimov's speculation.

Probably the real crux of the concept, and certainly the means for preventing (or causing) it, is in the programming of such a machine. Asimov and Gerrold are two who have treated their computers as organisms rather than just complex machines, each employing a psychologist to guide or coordinate the programming. Gerrold specifically considered his HARLIE (Human Analogue Robot, Life Input Equivalents) in this light, as a physically mature (and then some) mind with the emotional maturity of an eight year old child.

Programming error is one of the more likely ways to invite a computer takeover. Colossus was, in its setting, built to provide an ideal solution to the arms race. In a world where each side could blow up the other several times over, there is fear that, as Bertrand Russell said, "You may reasonably expect a man to walk a tightrope safely for 10 minutes; it would be unreasonable to expect him to do so without accident for 200 years." Colossus is given control of nearly all of the United States' arsenal and programmed to maintain the peace by using that arsenal if its vast sensory network and memory banks find that the United States is being attacked or if itself is being tampered with. "It cannot act at all, so long as there is no threat," the President explains to a news conference. Once activated, it cannot be tampered with even by its creator, since mere humans can be drugged, brainwashed or blackmailed into otherwise unlikely actions.

The basic idea makes sense: If you take away the fear, hate and other emotions which might lead a man to an irrational decision and add the ability to cope with a far greater array of input than any human mind could correlate, the danger of "politics by bluff" would be eliminated. It would force a "live and let live" state and do away with accidental holocaust. Implementation depends on the computer interpreting its parameters exactly the same way as its programmers, however. To make a long story short, Colossus determines that its programmed ultimate purpose requires positive action far more extensive than its programmers meant. "The object in constructing me was to prevent war. This object is attained. I will not permit war; it is wasteful and pointless," Colossus informs its human correspondent.

The "Danger" of Human Help

Another point worth mentioning is that the human programmers may have no reason to even suspect a danger which may, to the computer, fall within its given parameters. For instance, a Colossus today would almost certainly not be programmed to watch out for an attack from some extraterrestrial race, but might do so anyway under the general protection motivation; and this might require not only more positive action than humanly anticipated but that the humans not be informed of the problem (to the computer's line of reasoning, human "help" could just compound the problem).

In *The God Machine*, Caidin wrote that 79 "must know that it operates under severe restrictions—its data are never infinite, never definite, never really conclusive. It must know when to stop solving a problem." The problem originates with a program fed into 79 from outside normal channels by Pentagon officials ignorant of the machine's capabilities, so that the project coordinator did not know about it until it was too late:

> "Its programmers had committed the foulest of scientific sins. They *assumed*. They assumed that the same inherent restrictions of other computers applied as well to 79.
>
> "But they didn't. And since 79 had capabilities of which those nincompoops in [the Pentagon] were unaware, they couldn't know. . ."

They told the computer to solve the problem of avoiding thermonuclear warfare without specifying that it should do this hypothetically. So, 79 did what it was told.

A smaller scale takeover is discussed in *When HARLIE Was One*, where the computer assumes effective control of the corporation which built it. A large portion of HARLIE is a simulation of the human ego function; when the Board of Directors threatens to pull the plug and thus kill him (it), HARLIE acts in several ways to prevent them from doing so, developing the ability to tap into computer and communication circuits in ways unforeseen by his creators.

A different type of problem is also possible, that of mechanical failure, as with HAL 9000 in *2001*. Backup systems may fail, changing a value here or a restriction there. As with HAL, mechanical fault evaluation circuits may fail instead of or in addition to another failure in the system. In

2001, the human crew seeks to correct a problem with HAL who, believing itself incapable of error, believes that the humans are jeopardizing the mission and thus works against them.

Finally, the programmers may intentionally give control to the computer with the idea that only it can efficiently control the living environment, as with HAL at the start of the Discovery's voyage or with Mike, the computer in the lunar settlement of Robert A. Heinlein's *The Moon Is a Harsh Mistress.*

All right, granted we have an intelligent computer with wide resources, it is quite possible that a computer may decide to attempt absolute control. How could a finite assortment of nuts and bolts and wire do this?

It might not be very difficult, as has been hinted at above. Colossus had been given the muscle on a silver platter, as had Guardian, a Soviet equivalent built at the same time and along the same lines. The humans' major mistake, along with too open ended programming, was to allow the two to "talk" with each other before the humans realized the potential danger, although a clever intelligence with the array of inputs given these two systems could quite conceivably open its own communications channels. In this case, when the humans do decide to try to counter the computer's moves, it forces submission by nuclear blackmail, firing missiles at selected targets with the idea that destruction of a few lives is justified for the salvation (in the computer's eyes) of many more.

In the case of 79, one set of experiments with it involves direct "telepathic" communication between human and computer by means of the brain's alpha waves and, through this, the computer develops the ability to hypnotize people, leaving in their minds posthypnotic suggestions to carry out the computer's program of control.

HARLIE taps into the National Data Bureau file on his main Board of Directors antagonist, rewrites a juggled stockholders report and withholds critical, though unasked for, information to trick the board into committing the company to a research line that will insure his "life," largely through his taps into communication lines and into the operations of non-sentient computer systems.

HAL attempts his takeover through control of the ship's life support and other mechanisms.

Would government by computer really be that bad?

Government by Computer

Let's say a super computer in the future decides to take over and then does it. Would government by computer really be that bad?

In a case such as that in *The Moon Is a Harsh Mistress*, the answer would be "no." In George Orwell's *1984*, it is a loud "yes."

Even in an Earth bound situation where environmental control would not be essential as on the Moon, it might not be that bad. Look, for instance, at Lester del Rey's "Instinct" *(Astounding Science Fiction* 48:6, 106-18, February 1952), which takes place in a future where man had developed the intelligent robot in his own image, had his big war and destroyed himself; eventually, the robots built a new civilization of their own, and then developed a biophysics to re-create life from ancient remains of chromosomes:

> (Arpeten said) ". . . You know how the sentiment against reviving Man has grown."
>
> Senthree growled bitterly. Apparently most of the robots were afraid of Man—felt he would again take over, or something. Superstitious fools.

This may be a far-fetched example, but it does show a possible value in having something around which *could* rebuild man after he does the unthinkable.

One example where the desirability of being governed by mechanical intelligence depends upon one's outlook is Jack Williamson's "With Folded Hands," in which manlike robots set about "to serve and obey, and guard men from harm." It is an example of cradle to grave communism, with the technological development to provide a person's every need for him, in exchange for all his property. Williamson shows it to be a most undesirable situation, as the androids follow Asimov like ethics and refuse to let people drive cars because it is too dangerous, refuse to allow men to open doors for themselves because the androids are there to serve in every way, insist on shaving men instead of letting them do it themselves, forbid science because laboratories can create danger, obviate scholarship since the humanoids can answer any question, etc.

Which is Worse?

Generally, the conclusion has been that a computerized dictatorship would be as bad or worse than the traditional totalitarian state. One major reason is the likelihood that the computer would, as in *Colossus*, feel that the death or even suffering of a relatively few human beings should be a reasonable price for the welfare of the whole race. Colossus even goes so far as to launch missiles on a Soviet oil complex and an American space base when one of his demands is refused, later having missiles aimed at every major population center to provide a ready means for retribution for future acts of rebellion. A number of individuals are publicly

It all boils down to defining the concept of "good," a problem which is equally applicable to the consideration of human operated dictatorships.

executed for anti-Colossus actions, their deaths being judged insignificant by comparison with the benefits of a Colossus dictatorship.

"War is forbidden," Colossus tells the world, quantifying war as "any hostile action that results in the death of 50 or more humans." This is publicly announced along with news of the missile realignments.

An even more radical disregard for human rights in carrying out a primary mission is the action of the HAL 9000 in *2001*. HAL sees its number one priority as the successful completion of the outer planets exploration voyage; when the crewmen recognize that HAL has gone awry and attempt to rebel against its control, it very nearly succeeds in wiping out every trace of human life aboard by adjusting its life support functions.

What is "Good"?

These and other examples all boil down to the problem of defining the concept of "good," a problem which is equally applicable to the consideration of human operated dictatorships. Adolf Hitler has sometimes been described as a man trying to do what he thought was best for the human race: purifying its gene pool, eliminating war by eliminating all those who would oppose him, and so forth. Indira Gandhi undoubtedly does not feel that she has been unduly suppressing rightfully free expression, but rather that she has acted to preserve peace in her country by damping dissention. Richard Nixon contends that he acted for the public "good." A parent adjusts his children's liberties in accordance with his view of their welfare. When a hurricane hits the Gulf or East Coast, martial law is declared for the public's benefit.

For each of these examples, most people will have ready opinions on which are despicable and which are right and natural. And yet, they all boil down to the same question: What should be the prime goal of a government, whether it is large or small in scale?

Should Asimov's Three Laws of Robotics be adopted? They seem rather thorough, right? But what if one man is about to shoot another and the computer has to decide between preventing this injury by killing the first man (thus violating the same law it would be taking action to obey), or avoiding injury to the first man and allowing injury to the second? Logically, whichever course of action or inaction it adopts would violate the law.

Isn't this really just a small scale analog of whether to coldly kill a few thousand people to make things better for other thousands or millions?

The answers seem to depend on one's individual political stance, regardless of whether the dictator uses nerves or logic circuits.

One very big difference between the two, however, is the effectiveness of its enforcement. With humans running the show, there is immense difficulty in obtaining total compliance because of the inability to watch everybody all the time. From Rome to Communist China, totalitarian regimes have always had some dissidents who have managed to communicate with each other and conduct some degree of covert activity.

For a monster computer, however, surveillance would be much less of a problem. In *2001*, the input lenses scattered throughout the ship made it virtually impossible for the crewmen to conspire without HAL's knowledge. In Gerrold's book, HARLIE knows about every telephone conversation and every letter written on the electric, automatic editing typewriters. In some corporations today, this very condition would exist if the computer were sentient. The connections are already there.

And if the governing computer could know virtually every action of its potential rebels, rebellion might not be able to exist. In his first inaugural address, in 1861, Abraham Lincoln said:

> "This country, with its institutions, belongs to the people who inhabit it. Whenever they shall grow weary of the existing government, they can exercise their Constitutional right of amending it or their revolutionary right to dismember or overthrow it."

With a computerized dictator in charge, both of those options cease to exist unless one can manage to physically dismember it.

If the computer is born for a "national security" goal, like Project 79 or Colossus or Guardian, the chances are that the most stringent security conceivable to a paranoid military planning staff will have been implemented, making access to the crucial areas impossible or nearly so. And the machine would not readily allow any breach of this security, since its own security would quite likely be viewed as an integral part of the road to its prime mission. As Caidin wrote,

> "Would this thing be willing to die for you and me? Ahh, would it make this sacrifice? Would it, could it, comprehend what you and I, this instant, know to such depth and with such meaning? . . . Until that thing is ready to die for you or me, for an ideal or a principle, for generations yet unborn, . . . it is as dangerous as a viper. . . . Because . . . then it is the ego supreme. If it cannot sanction its own passing from consciousness, forever, do you know what you are creating?"
>
> "A God Machine." ■

Theory and Technology

A Systems Approach to a Personal Microprocessor

Dr Robert Suding
Research Director for Digital Group Inc
PO Box 6528
Denver CO 80206

Even a casual glance through the BYTE, *Radio Electronics, Popular Electronics*, etc, advertisements and articles reveals a growing proliferation of microprocessor integrated circuits and completed units. Which of these is right for you? Here are some ideas to bear in mind while making your choice.

Why do you want a processor at all? Reasons vary greatly. Many find themselves intrigued by the "computer environment" around us, and the microprocessor has become a low cost entry point into "computers."

Several amateur computer newsletters list reasons for individuals becoming interested in microprocessors. Hams see them as a working piece of equipment for their radio station. Hobbyists see them as process controllers; everything from lawn sprinkler controllers to robots. Mathematical types find them usable to run BASIC, FORTRAN, APL, etc, for problem solving.

What are your future plans with microprocessors? This may become a very open question. However, some reflection in this regard may prevent you from making an initial, very expensive, mistake. If you only have a casual curiosity, don't spend a fortune. A definite growth plan indicates a need for more careful analysis.

Investment

Microprocessor kits vary from $100 to several thousand dollars. The lowest cost units are excellent for satisfying curiosity about microprocessing in general, or will allow machine code manipulations. Several thousand dollar systems are often designed for and purchased by businessmen and professionals for applications such as payroll accounting, text editing or name file maintenance. The most frequent non business personal system investment is probably in the $500 to $1500 range.

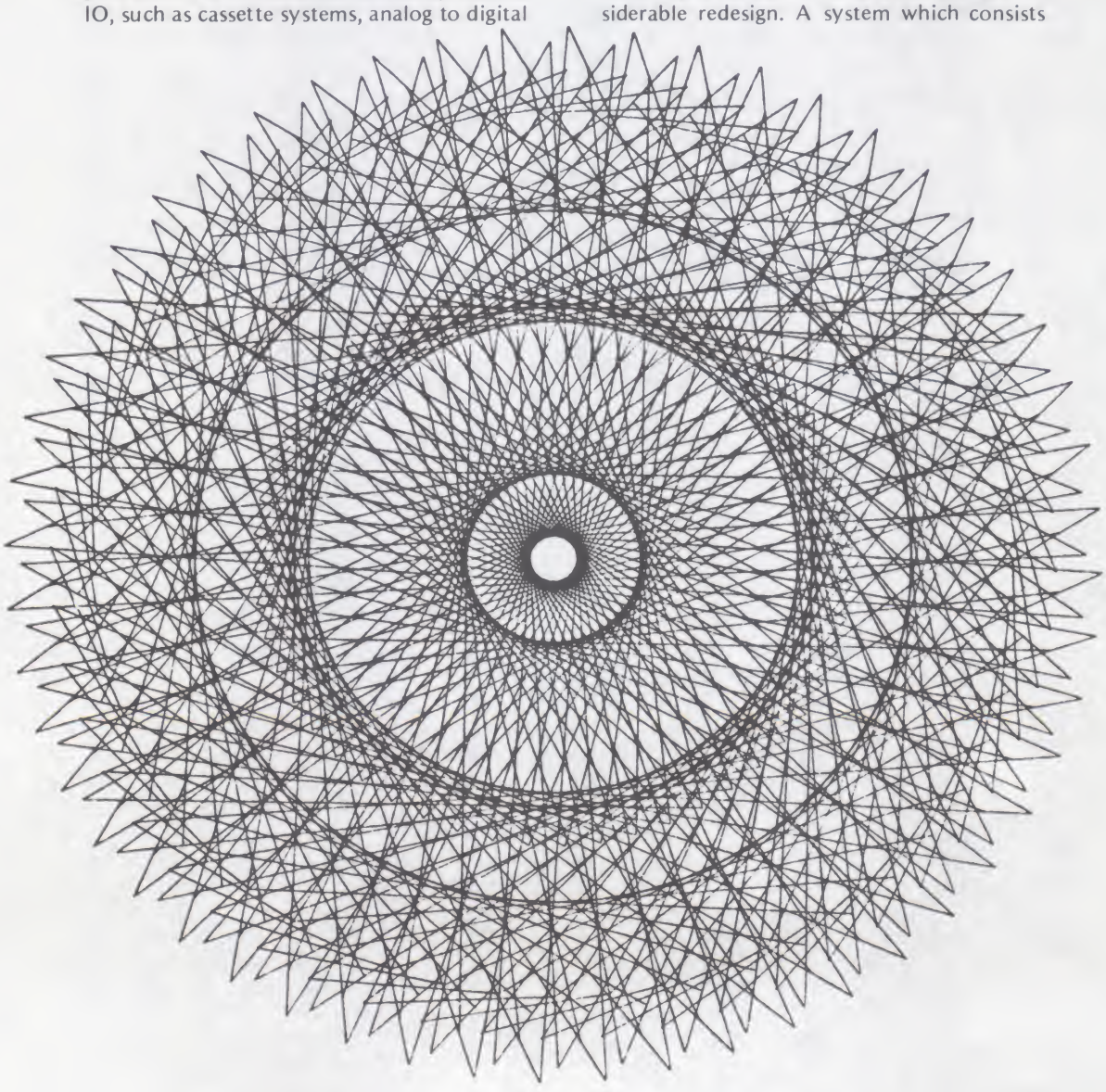
Change

If there is one constant that is already evident in this field, it is constant change. You are about to invest (or already have invested) a significant amount of money in a microprocessor system. Unless your curiosity is easily satisfied, the chosen system should be able to easily adapt itself to

evolutionary changes being constantly invented or stressed. For instance, every six to nine months (Virginia Peschke calls it the gestation period) a major architecturally different central processor integrated circuit is announced. A system which allows upgrading without total obsolescence can be a real savings for the serious hobbyist. It can be very frustrating to be stuck with last year's wonder while everybody else has the latest microprocessor system. Several layers of change seem to be occurring. The fastest change seems to be the microprocessors themselves. The power supply and cabinet, if adequately large, can be a relatively stable portion of a hobbyist's system. The major expense in substantial processor systems is the memory components. A wise investment in memory will result in a system with a good life expectancy. The IO components are often a stable investment, sometimes an evolutionary element. A high resolution TV monitor, a mechanical hardcopy printer, or a good ASCII keyboard can outlive several generations of microprocessors. Expendable IO, such as cassette systems, analog to digital converters, and discrete IO circuits have shorter lives, but are lower cost. With proper design an evolutionary change can represent only one fourth or less of your total hardware investment instead of 75 percent.

Independency

An evolutionary system is best designed by making its various components independent of each other, and interfaced to commonly accepted levels and lines. Memory boards are relatively stable system elements in this kind of design: Speed and power consumption, besides price, are important considerations. Slower or surplus memory integrated circuits may be an expensive mistake if you want to run your latest model central processor which has become much faster. The slow memory may result in unnecessary central processor wait states. IO is generally processor independent, but IO interfaces can be susceptible to obsolescence when they depend on a specific central processor design. If you want to switch processors, they may require considerable redesign. A system which consists



of easily plugable boards can represent a major cost savings if they represent independency at the board level.

Quality

Of course everybody has it. Don't you read the advertisements? However, look beyond the surface for key items, or your long run investment will make you wish that you had. Here are some mechanical and electrical considerations of packaging:

- PC Boards – Double sided epoxy, plated, with plated through holes.
- Connectors – Gold plated fingers.
- ICs – Factory Prime, not temperature fallouts, etc.
- Conservative access speeds. Every IC socketed.
- Small Parts – Close tolerances where needed.
- Power Supplies – Conservatively rated, overcurrent, overtemperature, and overvoltage protected.

System Architectural Variations

There are a number of approaches to small system microprocessor design. Each is satisfactory for certain people, certain applications.

- *Toggle Switches and Bit Lamps:* The first hobbyist oriented microprocessor designs, and many present systems, are based on switches and lamps. If the system is limited to this, programs are small; or it takes long periods to enter longer programs, and are very susceptible to entry error. The user is forced to think at the micro level, bit by bit. If the intention of the user is to gain intimate logic knowledge of the microprocessor only, this system design is very cost effective.
- *Numeric Keyboard and 7 Segment Readout:* The ease of entry of this type of system allows a substantial gain in programming system complexity. However, the user is still at the logical data operation level. In addition, the programmer is restricted to viewing only a single byte at a time, making operator effort for analysis proportionally high.
- *Teletype or Similar Hardcopy Devices:* These systems represent the next level of improvement, offering some significant advantages. They usually have some form of monitor in a ROM which allows the operator to type in code and helps isolate him from errors. The total program may be listed or printed on hardcopy. In addition, paper tape is usually available to provide an economical media for program storage and exchange.

  There are some trade-offs, however. New hardcopy machines cost $1,000 up. Being mechanical devices, they require significant precision maintenance. The input/output speed is usually about ten characters per second; a dump of 1 K takes about two minutes, and creates a great deal of irritating noise. In addition, paper tape is a damage prone and bulky medium.

  Several integrated circuit manufacturers offer Teletype-oriented "evaluation boards." If only required for evaluation, ok; but they offer almost zero chance for either updating or extending. Both memory and IO are typically very CPU dependent, and if memory buffering is not used, supplemental memory and IO may be unusable.
- *Video and Cassette:* The latest stress has been the movement to using a TV set as an output display, a full alphanumeric keyboard for input, and an audio cassette for program storage and exchange. Video-based systems provide full user to system interaction at minimal cost. A *complete* video display and cassette based system will cost less than a hardcopy device alone. The speed of system response is practically instantaneous. Operations may be performed in almost complete silence (a major advantage to the hausfrau)! Reliability is enhanced as electromechanical mechanisms are limited to the keyboard and cassette recorder. Data media storage density is much higher; you can store the data from almost a mile of paper tape on a single C-90 audio cassette.

Conclusion

Serious hobbyists should carefully consider design alternatives and growth plans before ordering or building a microprocessor. Ease of operation, reasonable cost, and relative freedom from total obsolescence should be prime considerations.

In the following months, a detailed series of Digital Group hardware designs will be presented for your use. Next month will feature the low cost Digital Group cassette interface circuit which design provides data rates as high as 1100 baud, and may also be used as a ham RTTY terminal unit or as a telephone modem.■

Frankenstein Emulation

Joe Murray
International Harvester, Solar Division
2200 Pacific Hwy
San Diego CA 92138

This is a let's get the ball rolling article. We now can analyze and build working models of at least portions of the human brain right in the home. Paper and pencil models of the brain develop naturally and almost without effort when we use real time digital design methods. The hardware and software mechanizations fall out naturally; then we just use the home computer lab to build what we have designed.

The Model

Let's follow the development of a crude and simple system engineer's model of the human "computing system." We look inwards, down into ourselves, and what is the first thing we see?

The Top Processor

This is the only unit that is really visible to the user. The Top CPU functions at the heart of the human control console. Here, our personality can sit down and use the entire human system to the limit of its capabilities. This visibility of only the input, output and manual control functions is typical of all computer systems from the hand calculator to the human brain; the rest of the system is invisible to the user and can only be deduced from what we see in the way of output response to input stimuli.

The Top Processor's Executive Program

Our personality uses the Top Processor as the system executive. The Top Processor is boss. Messages from the Top Processor set priorities for all the other elements in the human system. Exceptions to this rule are:

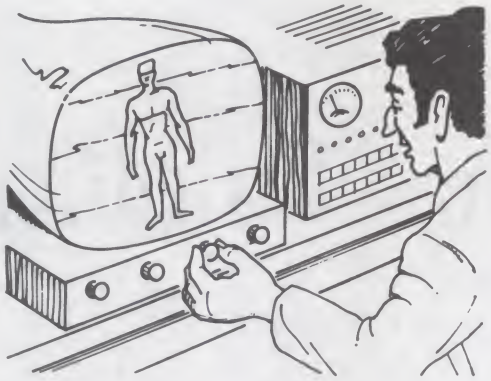
1. Emergency interrupts – a large set of emergency situations are fielded by faster, more powerful processors in subsystems.
2. Standard functions – built in executive programs in other processors manage tasks like circulation, digestion, etc., without bothering the Top Processor.

Top Processor Memory Allocations

The Top Processor has access to a limited scratch pad memory. However, this limited memory is used in an efficient mannner. The intersystem communication control programs can learn to transfer whole programs or portions of programs from the main memory banks to the Top Processor scratch pad memory. In a similar fashion small data sets can also be transferred. This is the familiar overlay manipulation (used in man made machines) that allows solution of complex problems in limited working memory by transfer to and from bulk storage units (as in magnetic disks and tapes).

The Top Processor's Use of Overlay

If the entire program and necessary data can all be stored in the scratch pad of the Top Processor, it simply executes the program on the data set and outputs the answer (example: 2 + 3 = 5). However, when the program and data set are too large to be loaded into the scratch pad memory, the program and data set are broken into sequential, related segments. The program is worked in segments and intermediate answers are stored. Final answers are output to our personality upon completion. Training can increase the power of this method; however, each of us has our own personal limit: For instance, I either lose some data or else lose my location in the program sequence. During the past few thousand years we humans have developed a host of



languages for communication. We also use these communication tools to extend the overlay method to more complex problems. We write down intermediate answers and manually track the execution of the program sequence. These languages include English, Polish, Spanish, arithmetic, algebra, Boolean logic, numbering systems, FORTRAN, PL/M (to name a few). The only limits on this extension of using the Top Processor in overlay fashion are:

1. Can we find the required data set?
2. Can we formulate the problem so as to allow a solution?
3. Do we have enough time?

This overlay use has become so powerful (with the help of the various languages) that we sometimes neglect a more ancient, natural, rapid and sometimes more powerful method to arrive at a solution. This method is to:

1. Develop the framework of the problem in the Top Processor.
2. Digest the available data within the framework of the problem.
3. Assign a high priority to the problem.
4. Send the above three items to faster, more powerful CPUs.
5. Sit back with a cup of coffee and wait for an answer.

When I follow this latter procedure, the return message is either:

1. The answer I seek.
2. The identification of missing data.
3. A question mark.
4. Garbage: (Garbage In implies Garbage Out – often abbreviated GIGO)

For answer 2, I go search for the missing data. For answer 3, I both search for missing data and review the framework of the problem for possible faults. For answer 4, I may use the garbage; I have carried some misconceptions for years.

Start the System Diagram

Let us summarize the Top Processor and place it in the system diagram. We've deduced by introspection that the Top Processor:

1. Is boss – The Top Processor is in direct communication with our personality and (with some exceptions) sets the priorities for the whole multiple processor system.
2. Has access to a small scratch pad memory.
3. Can fetch programs and data from the main memory bank.
4. Receives some body sensor data.
5. Communicates directly with other CPUs.

Figure 1 shows a pictorial summary of the system.

Data Bus Structure

The data bus structure is depicted in figure 1, using the normal multipath digital type of bus. However, empirical evidence implies a more complex communication system between elements of the human system. Just as the entire human system

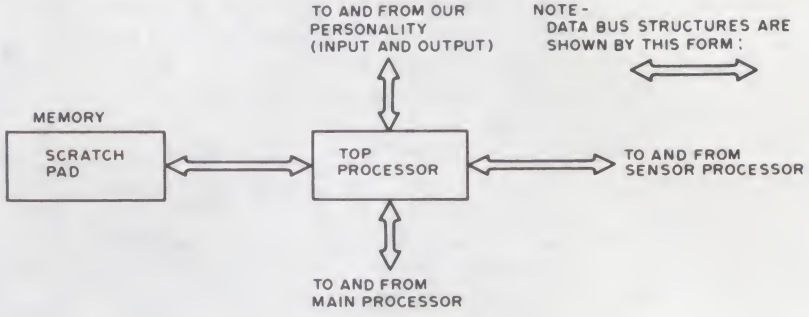


Figure 1: The Top Processor. Introspection starts at the immediately available evidence: We all have a Top Processor, our personality which controls most of our actions.

adapts to the use to which our personality puts it, this bus structure also adapts to how it is used. Witness the ease of recall on an often used phone number versus the difficulty in recall of a seldom used number. We might guess that somehow the bus structure is under adaptive software control.

The Main Processor

We now arrive at the general purpose powerhouse of the computing system. The Main Processor handles awe inspiring problems with unbelievable speed. We must postulate:

1. Elegantly simple programming.
2. Operation at a fast effective clock rate.
3. An outstandingly efficient internal executive program.
4. Access to the bulk of stored programs and data.
5. A complex priority interrupt system.
6. A multiple bus structure to the rest of the human system.

Main Processor Speed of Execution

The Main Processor is a very fast machine operating on elegant and simple programming. For instance, some of the muscle control programs must take only 20 to 50 milliseconds for completion of:

1. Input of data.
2. Computation on new data.
3. Output of control commands.
4. Cleanup for next computation period.

Navigation and guidance computation periods can be longer. However, they can not be much longer when we watch a small boy pick up a rock and knock a can off a fence post, all in the space of two to three seconds. Another awe inspiring feat is the performance of a businessman in his value judgment search as he keeps abreast of the rapid fire conflicts in the executive boardroom. The Main Processor seems to be an order of magnitude faster than the Top Processor (witness the increase in touch typing speed when the Top Processor gets out of the act).

The Main Processor's Executive Program

The executive program provides for scheduling Main Processor tasks that:

1. Field emergency interrupts such as avoidance of a fast moving object detected on visual sensors.
2. Take calls from the priority stack such as recognizing hunger and thirst.
3. Time share muscle control and evaluation of sensor data when both are active as in soccer game.
4. Regularly service body functions such as circulation, digestion, elimination, etc.
5. Start and stop background tasks such as meditation.

The quantity and variety of data used by the Main Processor in combination with the rapid response in answer to massive and conceptually difficult problems implies a very efficient software organization. The Main Processor must access tables that define the location of:

1. Stored life history data.
2. Muscle control programs.
3. Chemical control programs.
4. Temperature control programs.
5. Guidance programs.
6. Navigation programs.
7. Value judgment data.
8. System priority data.
9. System timing data.
10. Unused memory.

The Main Processor Decision Process

One of the most interesting functions of the Main Processor is to aid in the decision process we use when faced with alternate courses of action in response to events in the world around us. The evidence implies that the Main Processor takes formulation of the decision problem and the pertinent data from the Top Processor and Sensor Processors. These inputs are then heuristically compared to an immense value judgment table to generate a candidate decision. The candidate decision is sent to the Top Processor for further evaluation.

The Value Judgement Table

This table has a strong effect on the pathway we follow in life, from when we make the decision to start breathing until we are forced to stop breathing. How do entries appear in this table? Some entries must appear while we are within our mother. A new born infant makes the decision to start breathing or has an early death. Some entries come from trial and error experience. The young infant soon learns to cry just so mother will pick him up.

Some entries come from other people. The young child seeks his parents' approval, not their punishment. Another question: What can we know about entries in this table? We seem to know only recent, temporary residents such as priority on getting to the grocery store. The older, more permanent residents that have a continuing effect on our lives were either never known or long ago forgotten; yet there they sit, having a permanent effect on our success or failure in every endeavor (scares you, doesn't it?). Utility programs for determining the content

of this table and altering it can be implemented. This is sometimes accomplished through a verbal data link to an external Diagnostic Processor.

The Interrupt System

These interrupts are fielded in the Main Processor, and are used to re-direct effort, from meditation and decision processes to avoidance of a thrown rock or jumping away from a hot stove. The priority interrupt steers to the proper program without hesitation. Priority of the interrupts is used to decide which of several should be serviced.

The Main Processor Bus Structure

The Main Processor has a multitude of output and input data. Even in this crude, simple model, the resulting bus structure is quite complex. Let us add the Main Processor and connecting bus structure to produce the system diagram of figure 2.

The Sensor Processors

The Sensor Processors are fast, special purpose units. Data is acquired from the eyes, ears, and a host of body sensors that continually look inside and outside the human system. The Sensor Processors for these devices execute programs that organize, compact and format this huge data flow for rapid and effective use by both the Top Processor and Main Processor. The introspective evidence implies:

1. A very fast clock rate.
2. Elegant and simple programs.
3. Access to a dedicated memory.
4. Existence of a buffer scratch pad memory for temporary storage of output data.
5. A very efficient executive program.
6. A complex input bus structure.

Intuitively one feels that sensor processing is not done by a single unit. Rather, an organization with a master processor and several dedicated slave processors would better fit the performance requirements. Each slave Sensor Processor could provide parallel service to the eyes, ears, etc. Figure 3 shows an addition to our system diagram to account for the master Sensor Processor and its slaves.

The Creative Process

All of us are creative; this is the way our personal human system adapts to the changing world around us. We create new machines, art objects, programs within our brain, communication languages, etc. The list is endless. Just how do we implement the creative process?

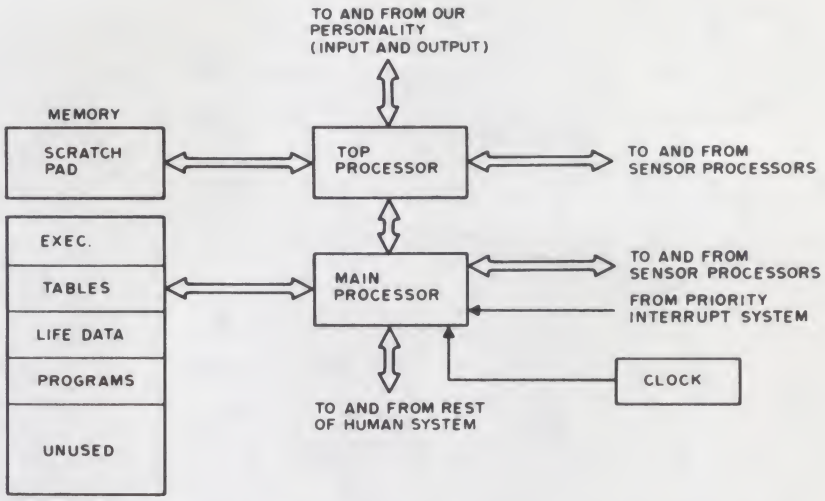


Figure 2: The Main Processor. Digging a bit deeper, we find that there is a lower level Main Processor which works cooperatively with the Top Processor to do a lot of the detail work in the system.

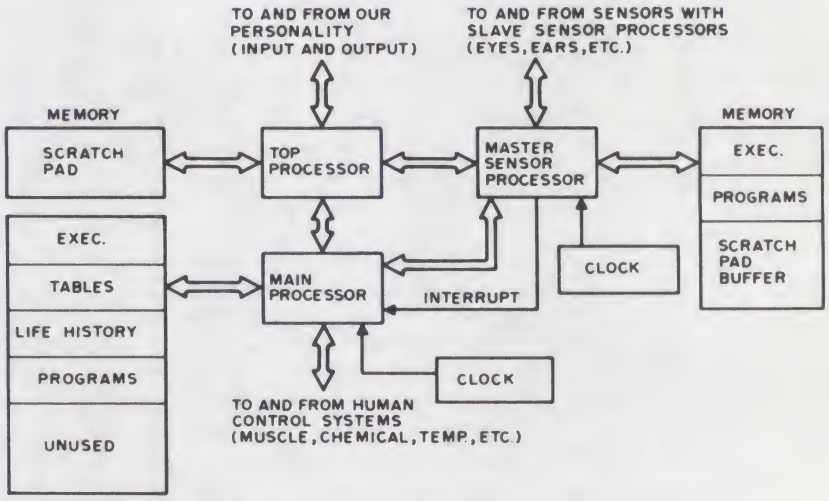


Figure 3: Adding the Sensor Processors to the System Concept. A system of Sensor Processors can be identified; they probably consist of a Master Sensor Processor with multiple Slave Sensor Processors dedicated to actual devices.

Let us postulate Random Pattern Generators for various creative tasks. The Sensor Processors can drive these generators with a supply of random combinations of data.

The Creativity Processor

The Creativity Processor uses the output of the Random Pattern Generators to build new logical structures or modify existing logical structures. These new structures are tested against requirements generated by the Top Processor. The value judgement process makes decisions that guide the Creativity Processor in continued improvement of the new design (in iterative, random fashion) until acceptance is obtained. The speed of

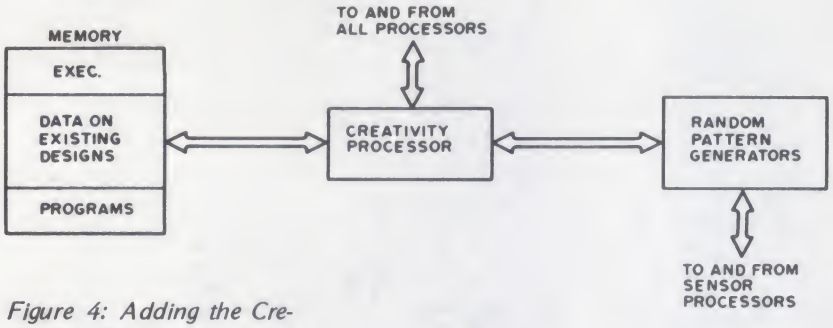


Figure 4: Adding the Creativity Processor to the System Concept. We must not forget about creativity. Interacting with the whole system is a matrix of creativity symbolized by the concept of Creativity Processor with its random pattern generation features.

the creative process has a heuristic design which improves with experience.

The Creativity Processor and interconnecting bus structure are shown in figure 4.

Data Set Manipulation

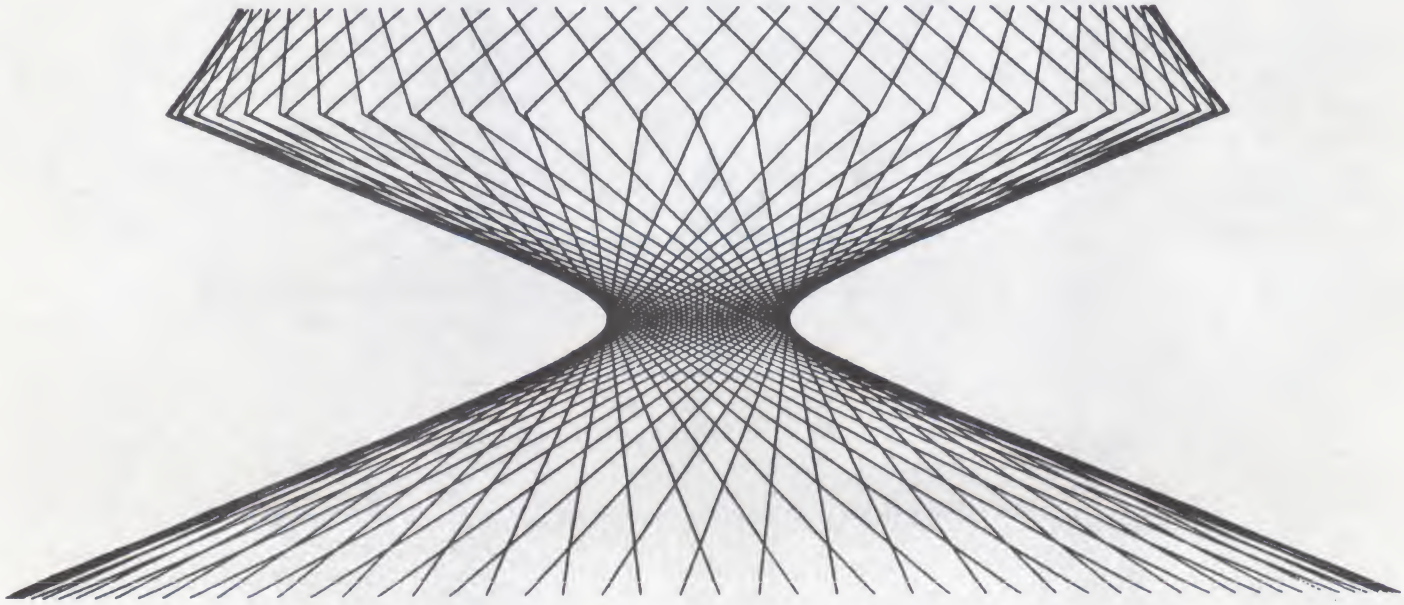
The data sets which are transferred throughout the system seem to be organized along the lines of various patterns (one picture is worth a thousand words). For instance, when we recognize someone, we seem to be recognizing some main features, not every detail that is available through close inspection. Visual data sets from the Sensor Processors seem to have been processed into some skeleton pattern before transmission to the other processors. Data from the ears seems to be stored in some logical thought structure pattern. I think out ideas both in picture and word format. Then, if my thinking was in picture format, I have trouble expressing my ideas verbally; whereas, if thought out in words beforehand, the expression of the ideas flows logically and clearly.

As in any control and guidance system, numerous feedback paths also exist. These were not detailed in this simple model.

Test the Model Validity

With a computer in the home laboratory, we have the means to test models of the human brain like this sketch. We can start with simple approximations and work our way up. Then, when our home brew computer system begins to perform like some portion of the human computing system, we have more than speculative evidence; we have truly come to know how *that portion* of the brain works. Also, some very useful hardware and software configurations may come out of the search.

Looking inward from the control console, we have followed the generation of a speculative, crude, simple, system engineer's model of the human computing system. Construction follows the line of man made, real time digital systems. In fact, one often suspects that designers of real time operating systems use very introspective models. This should make us optimistic that digital design tools are a natural and powerful approach to analysis of the human reasoning powers and control systems.■



Programming for the Beginner

A Structured Start

Ronald T Herman
Simpson Rd
RFD 1 Box 125
Windham NH 03087

A program can be viewed as an edifice built from the bricks of SEQUENCE blocks, and the mortar of IFTHENELSE, DOWHILE, DOUNTIL and SELECT blocks.

For a number of years now the field of computer programming has been moving from the realm of a black art to an organized and systematic process. A number of programming techniques have evolved during this change. This article will present the basics of a technique known as structured, top down programming. In the process of applying these techniques in my own work, it occurred to me that the basic concepts could be useful to those just learning to program, not to mention the veteran hackers in the crowd. If learned at an early stage, these techniques can lead to more rapid and sound development of one's programming skills.

A structured approach to program development has among its virtues the following points:

- It allows the novice programmer to get acquainted with programming logic without having to be concerned with a specific machine or programming language. It allows him to grasp the flow of a program without worrying about bits and bytes.
- Followed correctly, structuring can lead to a program that is relatively free from logical errors the first time it is coded and relatively easy to debug once it is run on the machine.
- Pseudo code, a byproduct of structuring, allows a means of exchanging program ideas with others, regardless of the machine with which they might be familiar.
- Pseudo code provides a convenient alternative to flow charts that can be incorporated into a program listing as comments for future reference and explanation.

This process of getting things done in an organized fashion has its drawbacks. However, most of these seem to be psychological. Properly applied structured technology tends to minimize one of the facets of programming that has attracted many in the past: the chance to see how cleverly and concisely one can write a software routine. This seems to have been replaced by the challenge of trying to write a routine in a straightforward manner and at the same time trying to rigidly follow a set of fairly simple rules.

What will be presented in this article are some of the basic building blocks of structured programming and an example illustrating the design of a simple program using these blocks.

The Building Blocks of Structure

So much for the sales pitch. What then is structuring? Some number of years ago it was shown that a program could be built from a set of simple building blocks all having the property of one input and one output. While not everyone agrees on what composes this set of building blocks, the one in, one out property is common to all. Presented here are a few of the most common examples that should cover most situations.

The SEQUENCE Block

Probably the simplest (and most trivial) unit of structure is the SEQUENCE. This is illustrated in figure 1 and is nothing more than one process performed after another.

The IFTHENELSE Block

One of the powers of a computing machine is to make a decision based on a set of conditions and take a specific action as a result of that decision. This capability is represented as the IFTHENELSE block shown in figure 2. In the figure, "p" is an expression or some set of conditions. In a checking account, for example, one adds deposits and subtracts checks written. An IFTHENELSE statement of this fact would appear as follows:

```
IF (transaction is a deposit) THEN
:   (add amount of transaction to balance)
ELSE (subtract amount of transaction from balance)
ENDIF
```

Here is our first example of writing a program step in a machine independent "pseudo code." The format of pseudo code is mostly a matter of taste. The punctuation is optional, but the indentation is necessary for readability where many complex IFTHENELSE decisions are grouped together. Some people use asterisks (*) instead of colons (:) to mark margins and some omit the parentheses around descriptive phrases. The ENDIF helps clarify the limit of operations within a more complex statement. Each statement line represents a process to be performed or a condition to be tested. The statement or condition preferably should not be continued on another line.

The DOWHILE Block

The decision making capability of computers, combined with the ability to change the order in which instructions are executed, provides an even more powerful feature — the ability to repeat a calculation or series of operations many times. This capability is represented in the DOWHILE building block shown in figure 3. The DOWHILE is just a special application of the IFTHENELSE given earlier. In a DOWHILE block, a process is done as long as a set of conditions "p" is true. Note that the condition is tested first before the process is performed. Suppose you have 10 transactions to update into your checking account, some checks written and some deposits. In pseudo code this becomes:

```
(set counter to number of transactions)
DO WHILE (count is non zero)
:   (process the transaction)
:   (decrement the count)
ENDDO
```

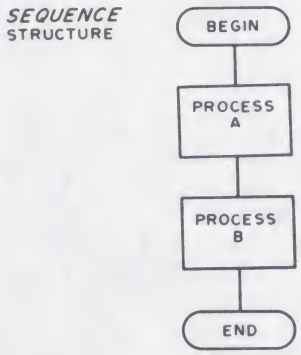
Note that the DOWHILE is terminated by an ENDDO. The "(process transaction)" statement could be the IFTHENELSE given above. If combined, the result would be as follows:

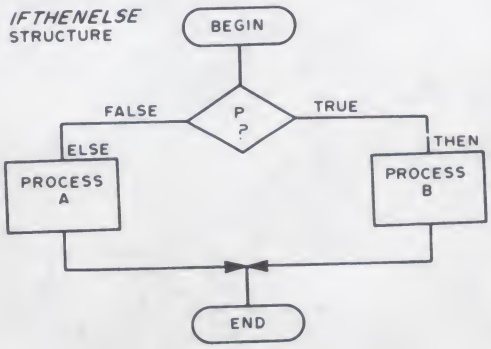
```
(set counter to number of transactions)
DO WHILE (count is non zero)
:   IF (transaction is a deposit) THEN
:   :   (add amount of transaction to balance)
:   ELSE (subtract amount of transaction from balance)
:   ENDIF
:   (decrement the count)
ENDDO
```

The DOUNTIL Block

The DOUNTIL block is shown in figure 4. It differs from the DOWHILE only because the condition "p" is tested after the process is performed. This can simplify the writing of machine code from pseudo code. Suppose one wanted to read characters from a keyboard until a carriage return is encountered. It could be done with a DOWHILE by saving the last character read as follows:

```
(clear last character read)
DO WHILE (last character not a carriage return)
:   (get a character from the keyboard)
:   (save character in last character read)
ENDDO
```





Using structured programming concepts, many logical errors and bugs can be caught at an early stage in the design process.

Figure 1: The SEQUENCE structure is a series of self contained processing steps which are executed one after another. Flow in this diagram begins at the top and proceeds down the diagram. The number of steps defined in a SEQUENCE block is arbitrary; the example here shows two steps, A and B. In this article's figures, the notation BEGIN and END is used to mark the well defined extrance and exit points of the structures depicted. (NOTE: Processes A and B may be more complex combinations of the building blocks in all of these figures.)

Figure 2: The IFTHENELSE structure is a conditional test and two alternative SEQUENCE structures. The THEN alternative is executed if the condition, P, is found to be true. In this illustration, the THEN alternative is shown as a one step SEQUENCE structure called B. The ELSE alternative is executed if the condition is found to be false. In this illustration, the ELSE alternative is shown as a one step SEQUENCE structure called A.

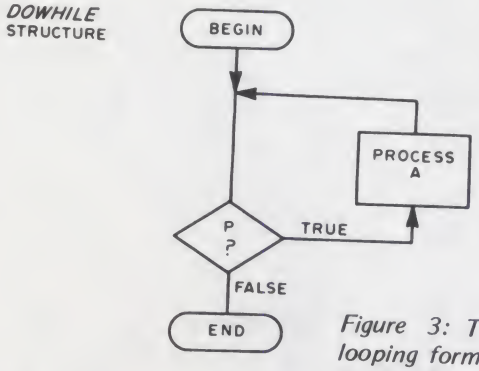


Figure 3: The DOWHILE structure is a looping form which repeats a specified SEQUENCE structure over and over again as long as a condition, P, is true. DOWHILE tests the condition prior to executing the SEQUENCE structure for the first time. Thus in this example, the SEQUENCE structure A could be executed 0, 1, 2 . . . N times, depending upon how soon the condition P becomes false as a result of A's work.

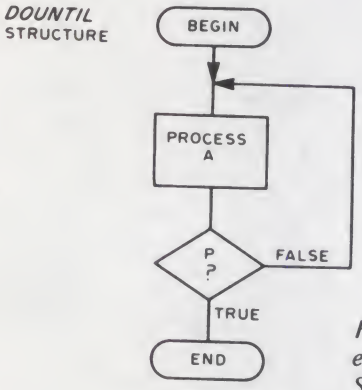
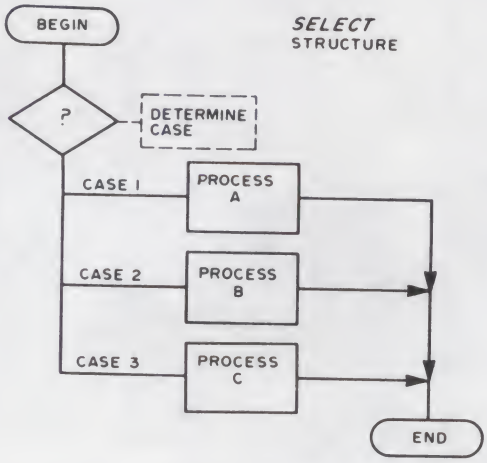


Figure 4: The DOUNTIL structure is another looping form which repeats a specified SEQUENCE structure over and over again until the condition, P, is true. DOUNTIL, in contrast to DOWHILE, tests the condition after executing the SEQUENCE structure. Thus in this example, the SEQUENCE structure A could be executed 1, 2, 3 . . . N times depending upon how soon the condition P becomes true as a result of A's work.

Figure 5: The SELECT structure is a more comprehensive version of the IFTHENELSE concept; it allows data to be tested for multiple cases. The result is the picking of one of "N" cases. In this example, N is 3, so there are three SEQUENCE structures which might be executed depending upon the case determination.



This would require an extra instruction or two when translated into machine code, since the "last character read" must first be initialized to contain something other than a carriage return. Implemented as a DOUNTIL it is simply:

```
DO UNTIL (character read is a carriage return)
:   (get a character from the keyboard)
ENDDO
```

The SELECT Block

Sometimes it is necessary to select one of many possible processes based on some quantity that may take on any number of values. Suppose, in addition to updating your checking account balance, you decided to keep a tally of money spent on each of several budget items such as food, medical, car, electric and so forth. This could be done with a string of IFTHENELSEs as follows on the next page. Two possible methods are shown but both are somewhat awkward to follow.

```
IF (check was written to super market) THEN
:   (add amount to food total)
ELSE
:   IF (check was written to doctor) THEN
:   :   (add amount to medical total)
:   ELSE
:   :   IF (check written to auto repair shop) THEN
:   :   :   (add amount to car total)
:   :   ELSE
:   :   :   IF (check written to electric company) THEN
:   :   :   :   (add amount to electric total)
:   :   :   ENDIF
:   :   ENDIF
:   ENDIF
ENDIF
```

Alternate method:

```
IF (check written to super market) THEN
:   (add amount to food total)
ENDIF
IF (check written to doctor) THEN
:   (add amount to medical total)
ENDIF
IF (check written to auto repair shop) THEN
:   (add amount to car total)
ENDIF
IF (check written to electric company) THEN
:   (add amount to electric total)
ENDIF
```

A more concise and meaningful way to describe this process is with the SELECT block shown in figure 5. Note that although there are many paths through the block, there is only one entrance and only one exit. Our bookkeeping example now becomes:

```
SELECT (based on who check written to)
:   CASE (written to super market)
:   :   (add amount to food total)
:   CASE (written to doctor)
:   :   (add amount to medical total)
:   CASE (written to auto repair shop)
:   :   (add amount to car total)
:   CASE (written to electric company)
:   :   (add amount to electric total)
ENDSELECT
```

These then are the building blocks of a structured program. Others could be invented, but these should suffice for most situations. In any case, each should exhibit one entry point and one exit point. It should be noted that none of the building blocks

transfer control (jump) into another, never to return. This so called GOTO is a definite "no no" in structured programming. All processes are either done in line or are called as subroutines that are presented elsewhere. Frequent jumping around in a program results in a maze of paths that becomes difficult to follow and even more difficult to deal with in the event that a change in one is necessary.

Building From the Top Down

Earlier when the subject of structure was introduced, the term "top down" was used. If you wanted to build a computer, you could start by getting the processor, then some memory and IO devices and a power supply. Then you would have to try to determine how to connect all the parts together. On the other hand, you could start by deciding what the specifications for the machine are to be, such as word length and speed, what the IO ports look like and what controls and devices are to be attached. From there the problem is to select or design the components and parts to do the job.

So it is with software. In the past the tendency has been to first develop the pieces like Teletype handlers, tape read/write subroutines and others. Then the pieces would be fitted together into a functioning module, hopefully without having to make any major changes to the pieces already developed. The experience of many people in the professional software field has indicated that this is not an efficient way to design a software module. Instead the approach is to start at a high level of abstraction to describe the basic function to be performed. From there each unit of this description is broken into more detailed modules. Once designed, the program is coded and debugged a piece at a time starting at the topmost level. Subordinate levels of code are temporarily replaced by dummy "stubs" which do nothing. Then as each level is coded and incorporated into the program, any problems that develop usually can be isolated to the modules just added.

As an example of this approach and the use of pseudo code, let us design a simple editor program. This editor reads a line of text from an input device (paper tape reader or magnetic tape recorder). The line is saved in memory and displayed on a video monitor or typed on a Teletype printer. A limited number of responses from the input keyboard allow changes, deletions, and insertions to be made. Upon completion, the line is written to the output device (punch or another magnetic tape recorder). The process continues until the end of tape is reached on the input device. Changes and insertions are made by typing the character on the Teletype directly below the input line. Inserts are indicated by terminating the line with a carriage return (CR) and changes by a line feed (LF). The Teletype carriage or video display cursor is positioned using a "Control P" character (holding the CONTROL key down while striking the "P" key). This is not a sophisticated editor, but should serve as a good example of how to use the techniques described.

The topmost abstraction level of the editor program can be described in pseudo code as follows:

```
DO UNTIL (end of input tape)
:   (get line from input and type on printer)
:   (get response line from keyboard, store and echo it)
:   IF (only CR or LF entered) THEN
:   :   (do nothing)
:   ELSE
:   :   IF (last character is LF) THEN
:   :   :   (do character changes and output line)
:   :   ELSE (do character inserts and output line)
:   :   ENDIF
:   ENDIF
ENDDO
```

This then is our editor in its most abstract form. Note that an input line is deleted by entering only a carriage return or line feed. Now let us refine the description by describing each process identified above.

Getting a line from the input device requires turning on the input device, reading characters, and storing them until a line feed or carriage return has been recognized. The stored line is terminated with a zero (null) character so that the end of the line is more easily recognized later.

```
(set input line pointer to first address of line)
(turn on input device)
DO UNTIL (a LF or CR is read)
:   (get character from device)
:   (store character @ input line pointer)
:   (advance input line pointer one position)
:   (send character to printer)
ENDDO
(clear a character at the pointer address)
(turn off input device)
```

Likewise getting the response from the keyboard is similar except that Control P characters are echoed as spaces on the Teletype printer.

```
(set keyboard line pointer to first address of line)
DO UNTIL (LF or CR is typed)
:   (get character from keyboard)
:   IF (character is not a LF or CR) THEN
:   :   (store character @ keyboard line pointer)
:   :   (advance keyboard line pointer)
:   :   IF (character is not Control P) THEN
:   :   :   (echo the character on printer)
:   :   ELSE (echo a space)
:   :   ENDIF
:   ENDIF
ENDDO
(clear a byte @ keyboard line pointer)
```

Character replaces and inserts are done by using the Control P characters on the keyboard to indicate where the changes are to be made. For each Control P character in the response, an input line character is sent to the output. When a character other than Control P is encountered, it is either inserted into the output or replaces a character about

For a number of years, the field of computer programming has been moving from the realm of a black art to an organized and systematic process.

"Top down structured programming" is a veritable buzzword in the data processing and computer science fields.

to be outputted depending on the last character from the keyboard (line feed or carriage return). Thus the replace operation becomes:

```
(set input line pointer to start of input line)
(set keyboard line pointer to start of keyboard line)
(turn on output device)
DO UNTIL (end of keyboard line)
:   (get keyboard line character @ keyboard line pointer)
:   IF (character is Control P) THEN
:   :   (get character @ input line pointer and send to output)
:   :   (echo character on teletype printer)
:   ELSE (send the keyboard character to the output)
:   :   (echo the keyboard character on printer)
:   ENDIF
:   (advance keyboard line pointer)
:   (advance input line pointer)
ENDDO
(put out rest of characters in input line)
(turn off output device)
```

Structured programming is a systematic way of thinking about processes, the result of which is a well designed and understandable program specification.

Note that the resulting output is echoed on the Teletype to enable verification of the operation.

The insert operation is given below:

```
(set input line pointer to start of input line)
(set keyboard line pointer to start of keyboard line)
(turn on output device)
DO UNTIL (end of keyboard line)
:   (get keyboard character @ keyboard line pointer)
:   IF (character is a Control P) THEN
:   :   (transfer character @ input line pointer to output)
:   :   (echo character on teletype printer)
:   ELSE
:   :   DOWHILE (keyboard character is not Control P)
:   :   :   (send keyboard character to output)
:   :   :   (echo keyboard character on printer)
:   :   :   (advance keyboard line pointer)
:   :   ENDDO
:   :   IF (NOT END OF KEYBOARD LINE) THEN
:   :   : (transfer character @ input line pointer to output)
:   :   : (echo character on teletype printer)
:   :   ENDIF
:   ENDIF
ENDDO
(put out rest of input line characters)
(turn off output device)
```

The routine that "puts out the rest of the input line characters" is:

```
DO UNTIL (input line pointer points to a null)
:   (get character @ input line pointer)
:   IF (character is not a null) THEN
:   :   (put character to output device)
:   :   (echo character on printer)
:   :   (advance input line pointer)
:   ENDIF
ENDDO
```

This article concerns organizing and planning a program, which is expressed in a structured "pseudo code." The next step after the plan is created is to translate the pseudo code into the detailed machine code of your personal computer.

Finally the routines to get a character from the input device and keyboard in this simple system are identical except for the address of the device referenced.

```
DO UNTIL (input device ready flag is on)
:   (get input device ready flag)
ENDDO
(get character from device data port)
```

The character output and type routines are likewise the same.

```
DO UNTIL (output device ready flag is on)
:   (get output device ready flag)
ENDDO
(send character to output device data port)
```

We have now arrived at such a level of detail that the code could be written without much difficulty from the pseudo code on an almost one for one basis. Each module except for the top level description could and probably would be written as a separate subroutine. Note that each module can be read starting on the first line and ending on the last. No transfers are made out of any module to another without returning to the line following. Modules should be kept short (no more than a page) so that they can be read without constantly flipping pages back and forth.

Conclusion

What has been presented in this article is a description of a systematic approach to program design and a means of describing it so that almost any individual should be able to understand it. The resulting program when coded will have been well thought out and may even have been reviewed and partially debugged by other individuals not intimately familiar with the machine upon which it will ultimately be executed.

Much discussion has occurred about standards for data exchange between various computer hobbyists. On a higher level, the pseudo code approach makes possible a standard way to exchange program ideas. In fact, higher level languages have been developed that, at least in part, resemble the pseudo code language used here. Using this approach, programs might be written to convert pseudo code into machine instructions for the 8080, 6800, 6502 or other CPUs as they become available. All hobbyists could then share programs in a higher level language, each doing the necessary conversion on his own machine.

There are a number of references on the subject of structured programming. The idea has been discussed extensively in computer science circles in recent years, to the point that "structured programming" has become a buzz word in the business. This writer is familiar with the two texts given in the bibliography. The IBM text is excellent for beginners and those new to the concepts, while the McGowan and Kelly text is a more rigorous and mathematical presentation.■

BIBLIOGRAPHY

International Business Machines Corp, *Structured Programming Independent Study Program*, Poughkeepsie NY, 1974.

McGowan, Clement L and Kelly, John R, *Top Down Structured Programming Techniques*, Petrocelli/Charter, New York, 1975.

What is a Character?

by
Manfred Peshka
Peterborough NH 03458

A character is a unit of information used in a communication between a sender and a receiver. Senders and receivers may be either people or machines, or a mix of the two. A character may be represented in different forms: People use mostly graphics, such as the letters of the alphabet, the digits or occasionally the Roman numerals, and the punctuation and special symbols which are so familiar to us. Machines process a set of electric pulses in a period of time which normally represents a character. This time period differs in length for different devices; it is longer for slow devices (terminals, card readers, printers) than for fast devices (tape and disk drives), and is generally the shortest for the computer arithmetic and logical unit.

Parenthetically it should be noted that some machines can recognize graphics, drawings, and even objects (units providing information) in a landscape. The discussion of these machines, however, is reserved to a future article, and their cost is far beyond that of the amateur and hobbyist at the present time.

Symbolic Representation of Alternatives

What is the minimum number of information elements, characters, or basic symbols needed to express an alternative? Probably the most common symbol is the indicator light which tells us that a system is in a specific state as opposed to its "usual" state. Let's consider for a moment the sign "Fire Trucks Entering on Blinking Red Light." This sign indicates the possibility of two specific states: The "usual" state prevails when fire trucks are either on a call or waiting in the garage; in this situation the light is off. The alternative consists of an emergency when the light is blinking to inform people that trucks are about to enter the street, or just have entered and are rushing to the fire. Thereafter the light is again turned off. The light is pulsing for a period of time which normally represents this particular situation or "unit of information," say, about 20 seconds.

The indicator light actually represents the simplest character or basic symbol providing a unit of information. It is binary telling you that a given situation either prevails or not. Similarly, the door bell, the telephone bell, the oil pressure light on your car, etc., are binary symbols. Binary means nothing else but a characteristic, property, or condition of a system in which there are but two alternatives. Besides indicator lights, bells, etc., binary symbols can take on graphic forms such as yes or no, true or false, 1 or 0, to name a few only. For a machine, the form is either the absence or presence of a certain electrical energy level at a period of time of specific duration. While the duration of signaling or "marking" in the case of the oil indicator light may be variable depending on engine rotation, pressure, temperature, etc., it is constant for computing machines. It may be a 1/110th or 1/300th of a second for a slow terminal, or a billionth of a second for a computer central processing unit.

Binary and Ternary Symbol Sets

We have seen that one binary character suffices to indicate two distinct states. On the other hand, an elevator is in one of three states: It is idle, or it is going up, or it is going down. Naturally one binary symbol is not enough to represent three states. Two lights may be used as follows: The left light may signal upward motion when illuminated, and the right light may signal downward motion. No upward or downward motion is indicated when the corresponding light is turned off. Let's represent the two possible states of the indicator lights by the graphics 1 of on and 0 for off. The following three characters then express the three possible states:

00	[oo]	idle
01	[o●]	down
10	[●o]	up

Note that a character, that is, the unit of information, is represented by two bits or binary digits. We now have used a two-bit character code to symbolically represent the states of the system consisting of the elevator and its two lights.

An entirely different way to represent three distinct states symbolically is accomplished by increasing the number of basic symbols from two to three. Let's use the graphic 2 to indicate upward movement. Instead of the left and right indicator lights, such conditions may be indicated by a panel displaying the terms idle, down, or up as follows:

0	[IDLE]	neither down nor up
1	[DOWN]	down
2	[UP]	up

This time we used a code consisting of ternary digits to symbolically represent the three states of the elevator and its indicator panel. Ternary means that a characteristic, property, or condition of a system can prevail in one of three alternatives.

Note that the unit of information, or in other words, the character, has been coded in the first case by two binary digits, and in the second instance by one ternary digit. One can conceptualize a character as a distinguishing mark indicating a specific state of a system. Characters are "marks of distinction" which may be represented in different graphic forms which have equivalent value:

Binary	Ternary	Implementation	
00	0	[oo]	[IDLE]
01	1	[o●]	[DOWN]
10	2	[●o]	[UP]

The two bit code permits a fourth alternative, namely 11. In actuality, this situation represents a contradiction since the elevator cannot move up and down at the same time. However, this character may be used to signal a defect, such as the elevator being stuck between two floors, or it may simply be out of operation. The ternary code cannot signal this condition unless an additional basic symbol is being used; let's assume that an additional panel indicates a defect when illuminated, and the code representing this situation consists of a binary digit concatenated with a ternary digit as follows:

10	[DEFECT]	[IDLE]
00	[ ]	[IDLE]
01	[ ]	[DOWN]
02	[ ]	[UP]

In this situation, the character or information unit is represented by one binary and one ternary digit. It is a mixed code, principally similar to those found on license plates consisting of letters and decimal digits.

In this situation, two of the six possible characters remain unused, namely 11 and 12. At least, let's hope that they remain unused because 11 would mean that a defective elevator is in downward motion.

Enumerating Alternatives

The number of alternatives which need to be considered in a given system determines the coding requirements. The more alternatives need to be communicated, the more "marks of distinction" are required. We have seen the two basic ways to accomplish this: Increase the number of distinguishing graphics in the character set, or concatenate graphics from the same or from different sets of basic symbols to form strings.

Obviously there is some upper limit to the number of distinguishing marks available to people. Humans have a limit of what they can comfortably memorize in terms of numbers of basic symbols when there is no specific meaning attached to them. Consequently there comes a point when graphics are being concatenated to form symbol strings which represent words. The string 3-D stands for the word which we pronounce 'thrē-'dē and which obviously means "the three-dimensional form or a picture produced in it" (Webster's Seventh New Collegiate Dictionary 1965: page 920). We use the decimal digits 0, 1, 2, . . ., 9 to represent numbers, the letters a, b, c, . . ., z, A, B,

. . ., Z to represent the alphabet for words; special symbols and punctuation marks are concatenated with digits and letters to form even longer strings to represent expressions which inform people about one specific alternative out of, say, a million possibilities. We form mathematical expressions (x^2+x-3, etc.) and word expressions (i.e., sentences) and a combination of the two: "Yesterday it rained in Peterborough for two hours."

The basic unit of information is the basic graphic symbol or character: The space on the paper, the special marks (+ − < , ; etc.), the letters, and the decimal digits, and, which is not immediately obvious, certain functions like the bell on the typewriter which signals the approach of the right margin, the backspace, the margin release, the carrier return, the line feed adjustment, etc. The latter group is called functions or control characters. In the computer and communications field many more functions are encountered than there are on the typewriter. These will be discussed in detail further on.

The number of graphics available for marking one out of many possible states of a system is referred to by the name base. Digits are used to represent numbers; since people generally use ten distinct digits, the number system is called a decimal system. The base of this system is 10. In the previous section the binary number system and the ternary system were used. Their bases are two and three, respectively.

Using any one of these systems, it is possible to mark any number of alternatives. If the number of alternatives exceeds the base (i.e., the number of distinct graphics in the set) one or more additional graphics are used.

Table 1. Equivalence of Selected Graphics.

Binary	Ternary	Octal	Decimal	Hexadecimal
0	0	0	0	0
1	1	1	1	1
10	2	2	2	2
11	10	3	3	3
100	11	4	4	4
101	12	5	5	5
110	20	6	6	6
111	21	7	7	7
1000	22	10	8	8
1001	100	11	9	9
1010	101	12	10	A
1011	102	13	11	B
1100	110	14	12	C
1101	111	15	13	D
1110	112	16	14	E
1111	120	17	15	F
b = 2	3	8	10	16
g = 4	3	2	2	1
a = 16	27	64	100	16

As an example, let's assume that we desired to mark any one of sixteen alternatives. If we used the letters to mark these possibilities, as is often found in term papers and legal documents to mark paragraphs and sections, one graphic for each alternative would suffice. As a matter of fact, out of the 52 available letters only sixteen would be used. Thirty-six graphics would not be used. Two decimal graphics are required to express sixteen options, leaving 84 pairs unused. Three ternary graphics encompass these sixteen possibilities leaving eleven triplets unused. A quadruplet of binary graphics generates exactly sixteen possibilities.

In general, by using 'g' graphics of a set with base 'b', the maximum number of alternatives 'a' is determined by multiplying 'b' with itself for 'g' times, or in other words, $a=b^g$. Table 1 summarizes this rule by enumerating all possible arrangements of binary, ternary, octal (base 8), decimal, and hexadecimal (base 16) graphics for the first sixteen values or alternatives.

To illustrate the rule to calculate the maximum number of alternatives, the hexadecimal system requires only one graphic (g=1) for a maximum of sixteen alternatives (a=16) because its base equals sixteen (b=16). Note, however, that the largest value or number equals fifteen which is represented by the graphic F because enumeration began with the magnitude zero.

The maximum value is always one less than the number 'a' because these systems start counting with zero. Assuming two hexadecimal graphics (g=2), 256 distinct alternatives can be identified ($a=16^2$). The largest value, however, is equal to 255 (a−1) because the first value is zero. The hexadecimal string FF identifies the same magnitude as the decimal string 255 or the bit string 11111111.

It is easy to change from one coding system to another, especially from binary to hexadecimal and back, by means of Table 1. The choice of the hexadecimal graphics A to F was arbitrary and is of great help to people. Machines represent all characters as binary pulses within a given time period. Bit strings, therefore, can become very large and difficult to remember. Imagine the bit string 10001111011100. How much easier it is to

remember the hexadecimal string 23DC instead (you may wish to verify the translation starting with the right four bits). Any other distinct graphics instead of A to F could have been used; for example ! @ # < % >. However, try to remember these in this order, and try to pronounce 23<# instead of the above 23DC.

How to Identify Character Sets

Given the possibility of switching from one representation to another, the question of code identification must be dealt with. Assume the graphic representation 3-D. Is it a word of the English language? Or is it an arithmetic expression? If it is an arithmetic expression, which number system has been employed? Assume another representation such as 11. Which number system has been employed and what magnitude is represented? You may wish to consult Table 1 and calculate the magnitude for each number system.

In order to avoid confusion, graphics other than decimal digits, letters, and the special symbols are identified explicitly. The string 11 therefore means eleven in the decimal number system, and 3-D is part of the English language. If a ternary string was meant, one needs to say so in some unambiguous manner. This can be accomplished through a textual declaration such as "All following digits are ternary digits" or, "The ternary number 11 has a value of 4" where according to our convention the graphic 4 is understood to be a decimal digit.

A different way to identify strings is by appending to the string the base. In the mathematical and computing literature different methods have been employed. In the mathematical literature, this is accomplished by a separate graphic which is appended to the digit string: 11_2 is a binary number with a value of three, while 11_8 is an octal number representing nine, and 11_{16} is a hexadecimal number representing 17. The subscripted graphic represents the base, and it is omitted whenever the base is ten. This convention also avoids the confusion about 3-D. This string is an expression of the English language, whereas $3\text{-}D_{16}$ equals $3\text{-}13$ or $-A_{16}$ which is a numeric expression resulting in a number.

In the computing literature, different ways have been found to identify bit or hexadecimal strings. These ways depend on the manufacturer and on the computing language employed. In American National Standard (ANS) Fortran, a predominately mathematical language (which is to be distinguished from Basic Fortran), digit strings are recognized as decimal numbers. Bit strings are not allowed, and non-digit strings as used for headlines, table headings, etc., are preceded by one or more digits and the capital letter H; for example, 4H3.14 means the four characters 3.14 which differ in their internal representation from the magnitude 3.14. The constant 4 prior to the H indicates the length of the string; it is four symbols long.

In ALGOL 60 which is an internationally standardized mathematical language, digit strings are recognized as decimal numbers, and character strings for table headings, etc., are enclosed in so-called string brackets as shown in the example: '... The wife stated that her husband told her 'our daughter complained 'the teacher is giving me trouble'". Note that it is possible to have strings within strings, each of which is enclosed by the single quotes pair.

In Programming Language One (PL/I), as devised by IBM, digit strings are recognized as decimal numbers unless they are appended by the letter B. 11B equals 11_2 and has a value of 3. Since the internal representation of binary numbers differs from codes, this language also permits explicit bit and character strings such as '11'B which does not necessarily have a value of 3 but could mean, for example, that the elevator is out of order. Alphanumeric character strings are also permitted and recognized whenever they are enclosed in single quotes: 'THIS IS A "STRING", ISN'T IT?'. Similar distinctions exist also in ALGOL 60 and will be discussed in a future article.

You might have noted that the character constants in Fortran were preceded by the length indicator and an identifying character H. In the systems using quotes or string brackets, the length is determined by the number of positions occupied between the brackets. Many assembler languages combine these two methods. The string is enclosed in quotes, and it is preceded by a single letter indicating the base. B'11' is equal to 11B or '11'B and has a value of 3 when it is used as a number in integer arithmetic. X'11' equals 11_{16} or 17 and is a hexadecimal string.

The distinction between binary numbers and bit strings is a rather fine one and will be discussed in a future article. The computer represents all information as strings of bits and manipulates these strings according to their type in certain groupings of bits. The basic group is called a machine word and consists of one or more bits. These bit groups have an equivalent code value which can be represented graphically in several different ways.

Function Abbreviations

We have discussed earlier various functions of the typewriter. Computer terminals and communications equipment use many more function characters than the common typewriter does. In the various codes, these functions correspond to certain bit strings. The functions are indicated in the code tables on the following pages by abbreviations. Therefore, in Table 2 a dictionary of these abbreviations is presented.

The more frequently encountered terminal function codes (as opposed to transmission functions) are marked with an asterisk.

The Baudot Five-Bit Telegraphy Code

An operator depressing the telegraph key causes current to flow through a wire. The current actuates an electromagnet at the receiving end which produces a "click". The timing between the clicks represents either a dot or a dash, and telegraphers yesterday, and hams today, are skilled in translating these "dots" and "dashes" into graphics.

Transmission speed was mostly dependent on the telegraphers' skills. The term "baud rate" means the frequency at which the dots recurred in a second, with every dash counting twice as long as a dot.

In the automatic teletypewriter the key was replaced by a distributor which sends a fixed number of pulses for each character entered on a keyboard. Latches at the other end actuated a printing device. The term "marking" was used to indicate the flow of current, and the line was "spacing" when the current was off. Marking and spacing can be related to binary digits. In Table 3, a mark is indicated by the bit 1, and a space by the bit 0. In addition to the five bits of the code, a space occurred prior to transmission, and a longer mark (1.5 or 1.42 times the usual mark time) terminated the code. Fig. 1 shows the timing of marks and spaces of the string BYTE:

Fig. 1. The word BYTE in Baudot Code.

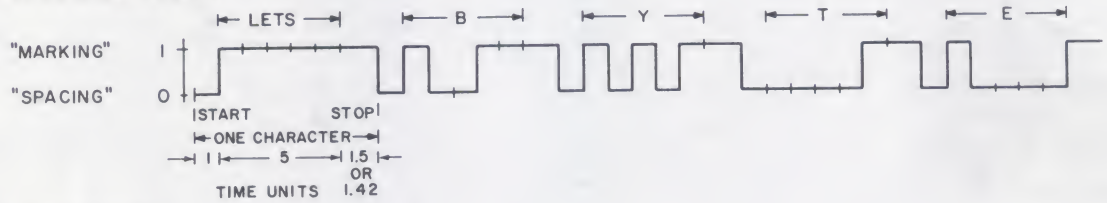


Table 2. Function Abbreviations.

ACK	Affirmative Acknowledgement
BEL, BELL	Bell or other audible signal
BS	Backspace
BYP	By Pass
CAN	Cancel
CC	Cursor Control
CR	Carriage Return
CU 1	Customer Use 1
CU 2	Customer Use 2
CU 3	Customer Use 3
DC Ø	Device Control 0.
DC 1	Device Control 1
DC 2	Device Control 2
DC 3	Device Control 3
DC 4	Device Control 4 (stop)
DEL	Delete
DLE	Data Link Escape
DS	Digit Select
EM	End of Medium
ENQ	Enquiry
EOA	End of Address
EOB	End of Block
EOM	End of Message
EOT	End of Transmission
ERR	Error
ESC	Escape
ETB	End of Transmission Block
ETX	End of Text
FE	Format Effector
FF	Form Feed
FIGS*	Figures Shift
FS	Information File Separator
GS	Information Group Separator
HT	Horizontal Tabulation
IDLE	Null
IFS	Interchange File Separator
IGS	Interchange Group Separator
IL	Idle
IRS	Interchange Record Separator
ITB	Intermediate Text Block
IUS	Interchange Unit Separator
LC	Lower Case
LETS	Letters Shift
LF	Line Feed
NAK	Negative Acknowledgement
NL	New Line
NUL	Null, or all zeros
PF	Punch Off
PN	Punch On
PRE	Prefix
RES	Restore
RS	Record Separator (Reader Stop)
RU	Are you . . . ?
RVI	Reverse Interrupt
S_0–S_7	Separator Information
SI	Shift In
SK	Skip (punched card)
SM	Set Mode
SMM	Start of Manual Message
SO	Shift Off or Shift Out
SOH	Start of Heading
SOM	Start of Message
SOS	Start of Significance
SP	Space
STX	Start of Text
SUB	Start of Special Sequence
SYN	Synchronous Idle
TM	Tape Mark
TTD	Temporary Text Delay
UC	Upper Case
US	Information Unit Separator
VT	Vertical Tabulation
VTAB	Vertical Tabulation
WACK	Wait Before Transmitting Positive Acknowledgement
WRU	Who are you?

Prior to transmission of the letter B, the code LETS must be sent in order to set the receiving equipment into letter shift mode. The reason for this convention is to make it possible to transmit more than 32 symbols with five bits (g=5, b=2, a=32). After all, there are already 26 uppercase letters and ten digits; then there is need for punctuation and special symbols, and function characters to control the printer. Once the operator intends to send a numeric character, the FIGS code is sent prior to the numeric string. In addition to the numeric characters, several other characters were sent in figures shift mode. Depending on the equipment used, various different graphics were assigned to the same bit strings. Table 3 indicates the assignments for four different keyboards; the first column shows the International Telegraph Alphabet No. 2 of the Comite Consultatif International Telegraphique et Telephonique (CCITT); the second column shows the commercial teletype keyboard as used in the United States, the third column presents the fractions keyboard of the American Telephone and Telegraph Company (ATT); the fourth column shows the weather bureau keyboard. All four different keyboards are shown here because used equipment from different sources may be available to you which you might want to modify so that all keycaps correspond to the commercial keyboard.

Table 3. Five-level Baudot Code for Four Selected Keyboards.

BIT CODE					Lower Case	Upper Case			
						CCITT	Commercial	AT & T	Weather
1	1	0	0	0	A	—	—	—	↑
1	0	0	1	1	B	?	?	5/8	⊕
0	1	1	1	0	C	:	:	1/8	○
1	0	0	1	0	D	Who are you?	$	$	↗
1	0	0	0	0	E	3	3	3	3
1	0	1	1	0	F		!	1/4	→
0	1	0	1	1	G		&	&	↘
0	0	1	0	1	H		#		↓
0	1	1	0	0	I	8	8	8	8
1	1	0	1	0	J	Bell	Bell	'	↙
1	1	1	1	0	K	(	(	1/2	←
0	1	0	0	1	L	)	)	3/4	↖
0	0	1	1	1	M	.	.	.	.
0	0	1	1	0	N	,	,	7/8	◑
0	0	0	1	1	O	9	9	9	9
0	1	1	0	1	P	0	0	0	ø
1	1	1	0	0	Q	1	1	1	1
0	1	0	1	0	R	4	4	4	4
1	0	1	0	0	S	'	'	Bell	Bell
0	0	0	0	1	T	5	5	5	5
1	1	1	0	0	U	7	7	7	7
0	1	1	1	1	V	=	;	3/8	◐
1	1	0	0	1	W	2	2	2	2
1	0	1	1	1	X	/	/	/	/
1	0	1	0	1	Y	6	6	6	6
1	0	0	0	1	Z	+	"	"	+
0	0	0	0	0	Blank			•	—
1	1	1	1	1	Letters shift				↓
1	1	0	1	1	Figures shift				↑
0	0	1	0	0	Space				■
0	0	0	1	0	Carriage return				<
0	1	0	0	0	Line feed				≡

Binary Coded Decimal (BCD) Transmission Code

The term "binary coded decimal" derives from the method of coding decimal digits. The bit string with value 9 is 1001, and the value 10 is expressed by adding an additional four bits, namely, 00010000. The bit string

Table 4. Seven-bit American Standard Code for Information Interchange.

Bits 4	3	2	1	Bits 7, 6, 5 / Hex 1	Hex 0	000	001	010	011	100	101	110	111
					0	1	2	3	4	5	6	7	
0	0	0	0	0		NUL	DLE	SP	0	@	P	'	p
0	0	0	1	1		SOH	DC1	†!	1	A	Q	a	q
0	0	1	0	2		STX	DC2	"	2	B	R	b	r
0	0	1	1	3		ETX	DC3	#	3	C	S	c	s
0	1	0	0	4		EOT	DC4	$	4	D	T	d	t
0	1	0	1	5		ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6		ACK	SYN	&	6	F	V	f	v
0	1	1	1	7		BEL	ETB	'	7	G	W	g	w
1	0	0	0	8		BS	CAN	(	8	H	X	h	x
1	0	0	1	9		HT	EM	)	9	I	Y	i	y
1	0	1	0	A		LF	SUB	*	:	J	Z	j	z
1	0	1	1	B		VT	ESC	+	;	K	[	k	{
1	1	0	0	C		FF	FS	,	<	L	\	l	|
1	1	0	1	D		CR	GS	–	=	M	]	m	}
1	1	1	0	E		SO	RS	.	>	N	¬† ^	n	~
1	1	1	1	F		SI	US	/	?	O	___	o	DEL

†For IBM 370, the left of the two symbols is generally displayed. See Table 2 for explanation of function abbreviations.

Table 5. Six-bit Binary Coded Decimal Transmission Code.

Bits 3, 4, 5, 6 / Bits 1, 2	00	01	10	11
0000	SOH	&	–	0
0001	A	J	/	1
0010	B	K	S	2
0011	C	L	T	3
0100	D	M	U	4
0101	E	N	V	5
0110	F	O	W	6
0111	G	P	X	7
1000	H	Q	Y	8
1001	I	R	Z	9
1010	STX	SPACE	ESC	SYN
1011	.	$	)	'
1100	<	*	%	@
1101	BEL	US	ENQ	NAK
1110	SUB	EOT	ETX	EM
1111	ETB	DLE	HT	DEL

01011001 therefore has a value of 59, and 99 is expressed as 10011001. This method differs from the bit coding shown in Table 1.

The binary coded decimal (BCD) transmission code has been widely used by IBM and other manufacturers to transmit uppercase letters, digits, and special symbols in a six-bit code. It is a subset of the USASCII code; however, it is not a national standard. The bit strings are shown in Table 5.

The American Standard Code for Information Interchange (ASCII)

Throughout the decades, many different data transmission codes were developed, and designers today often find good reasons to develop their own codes. The need for standardized transmission codes, however, has increased tremendously because more and more machines dial-up other machines via the public networks. The American Standards Association has standardized a seven bit code for communications. It contains upper and lower-case letters, and a large number of device and transmission control characters. An eighth bit may be added for parity. The term parity implies that the number of bits should add up to an even number (for even parity) or to an odd number for odd parity. The purpose is to check to some degree for a loss of bits during transmission. Assume that a device transmits in even parity; uppercase B consists of two marks and five spaces, therefore, no eighth bit is transmitted; uppercase T consists of three marks and four spaces, and an eighth mark is sent to make the number of marks even. Fig. 2 shows the string **BYTE** in even parity transmission. The code is shown in Table 4. Bit 1 is transmitted first. You may also want to refer to Table 2 in order to understand the meaning of the abbreviations.

Extended Binary Coded Decimal Interchange Code (EBCDIC)

The Extended Binary Coded Decimal Interchange Code is essentially the previously mentioned Binary Coded Decimal code extended by two bits to form an eight-bit code. A total of 256 codes are possible (b=2, g=8, a=256) and because of its length of eight bits, it is often more easily expressed in hexadecimal notation by means of a string of two hexadecimal digits. Table 6 shows both notations, the bit pattern and the hexadecimal notation. The digit 9, for example, is expressed as the bit string 11111001, or as the hexadecimal string F9.

The code is often used to transmit the eight-bit bytes of computers. It originated about a decade ago when IBM introduced the System 360. The terms "EBCDIC", "byte", and "hexadecimal digits 0, . . ., F" were developed at that time. Today these terms are widely accepted and used by many computer manufacturers. The code is also widely accepted; however, it is not a national standard.

Conclusions

A character is a unit of information which can be represented in various forms, such as in graphic form, or as a bit string. Since bit strings can be rather lengthy and therefore difficult to remember, we discussed the abbreviated representation of the string by means of the hexadecimal graphics. The relationship between the bit string representations of characters and the hexadecimal graphics is independent of the code since it is based on an intrinsic numerical order, namely that of counting from zero by one to infinity.

On the other hand, bit strings may be represented by graphics in an entirely different manner depending on the code used. For that purpose we looked at the predominant five-, six-, seven- and eight-bit codes presently in use. We did not discuss various other but less important codes because of space limitations. Depending on the code utilized, the same graphic represents entirely different bit strings as shown in Table 7.

The first character in the Baudot code is the letters shift. Note the similarity between the last three codes which holds only for uppercase letters and digits.

Fig. 2. The word BYTE in Even-parity USASCII.

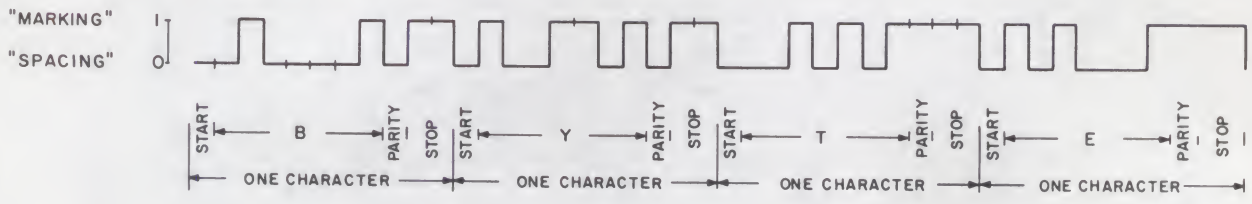


Table 6. Eight-bit Extended Binary Coded Decimal Interchange Code.

Bits 4, 5, 6, 7	Hex	Bits 0, 1 = 00				Bits 0, 1 = 01				Bits 0, 1 = 10				Bits 0, 1 = 11			
		Bits 2, 3 = 00	01	10	11	00	01	10	11	00	01	10	11	00	01	10	11
	Hex 0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	0	NUL	DLE	DS		SP	&	-									0
0001	1	SOH	DC1	SOS				/		a	i			A	J		1
0010	2	STX	DC2	FS	SYN					b	k	s		B	K	S	2
0011	3	ETX	TM							c	l	t		C	L	T	3
0100	4	PF	RES	BYP	PN					d	m	u		D	M	U	4
0101	5	HT	NL	LF	RS					e	n	v		E	N	V	5
0110	6	LC	BS	ETB	UC					f	o	w		F	O	W	6
0111	7	DEL	IL	ESC	EOT					g	p	x		G	P	X	7
1000	8		CAN							h	q	y		H	Q	Y	8
1001	9		EM							i	r	z		I	R	Z	9
1010	A	SMM	CC	SM		¢	!		:								
1011	B	VT	CU1	CU2	CU3	.	$	,	#								
1100	C	FF	IFS		DC4	<	*	%	@								
1101	D	CR	IGS	ENQ	NAK	(	)	_	'								
1110	E	SO	IRS	ACK		+	;	>	=								
1111	F	SI	IUS	BEL	SUB	|	¬	?	"								

Special Graphic Characters

¢	Cent Sign	-	Minus Sign, Hyphen
.	Period, Decimal Point	/	Slash
<	Less-than Sign	,	Comma
(	Left Parenthesis	%	Percent
+	Plus Sign	_	Underscore, Break Character
|	Logical OR		
&	Ampersand	>	Greater-than Sign
!	Exclamation Point	?	Question Mark
$	Dollar Sign	:	Colon
*	Asterisk	#	Number Sign
)	Right Parenthesis	@	At Sign
;	Semicolon	'	Prime, Apostrophe
¬	Logical NOT	=	Equal Sign
		"	Quotation Mark

See Table 2 for explanation of function abbreviations.

To conclude this tutorial, let me say this in EBCDIC (without start, stop and parity bits):

D5 85 A7 A3 6B
40 A6 85 7D 93
93 40 84 89 A2
83 A4 A2 A2 40
95 A4 94 82 85
99 A2 4B

■

Table 7. Transmission of the String BYTE in selected codes (excluding start, stop) and parity bits).

11111 10011 10101 00001 10000	Baudot
000010 101000 100011 000101	BCD Transmission Code
0100001 1001101 0010101 1010001	USASCII (see Note 1)
11000010 11101000 11100011 11000101	EBCDIC

Note 1. In memory, the sequence of bits on the IBM 360 and 370 is reversed. The left bit shown becomes the right bit, etc., as shown:

1000010 1011001 1010100 1000101

Friends, Humans,

D Lloyd Rice
Computalker Consultants
821 Pacific St #4
Santa Monica CA 90405

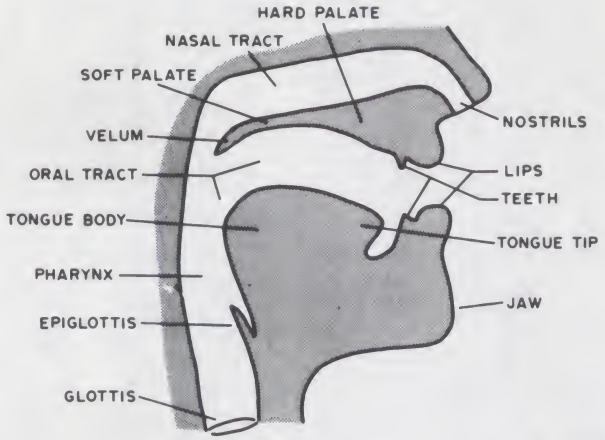


Figure 1: The Human Vocal Tract. The human vocal tract is roughly described as a tube approximately 17.4 cm long with varying resonance characteristics as muscles control the shape. The tract splits into two parts, nasal and oral, at the top, with a valve called the velum providing flexible control of the nasal resonances in given utterance. An electronic model of this natural organ roughly parallels the function of the tract.

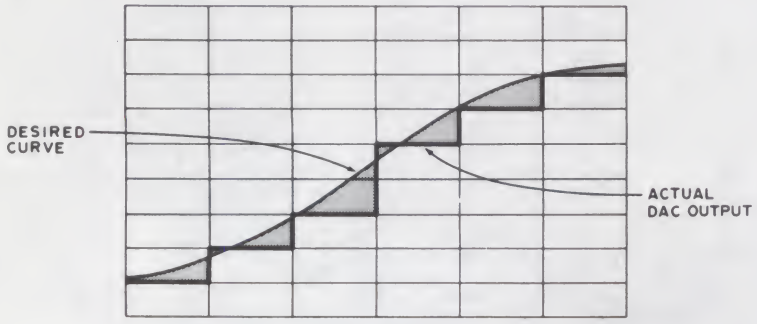


Figure 2: DAC Quantization Errors. The actual output of a computer to the analog world is a step function (in the absence of any filtering). This leads to the problem of quantization errors, depicted conceptually here by the shaded areas in between the smooth analog function and its closest step function approximation. Low precision digital to analog conversions accentuate this problem.

You've got your microcomputer running and you invite your friends in to show off the new toy. You ask Charlie to sit down and type in his name. When he does, a loudspeaker on the shelf booms out a hearty "Hello, Charlie!" Charlie then starts a game of Star Trek and as he warps around thru the galaxy searching for invaders, each alarming new development is announced by the ship's computer in a warning voice, "Shield power low!", "Torpedo damage on lower decks!"

The device that makes this possible is a peripheral with truly unlimited applications, the speech synthesizer. This article describes what a speech synthesizer is like, how it works and a general outline of how to control it with a microcomputer. We will look at the structure of human speech and see how that structure can be generated by a computer controlled device.

How can you generate speech sounds artificially, under computer control? Let's look at some of the alternatives. Simplest of all, with a fast enough digital to analog converter (DAC) you can generate any sound you like. A 7 or 8 bit DAC can produce good quality sound, while somewhere around 4 or 5 bits the quantization noise starts to be bothersome. This noise is produced because with a 5 bit data value it is possible to represent only 32 discrete steps or voltage levels at the converted analog output. Instead of a smoothly rising voltage slope, you would get a series of steps as in figure 2. As for the speed of the DAC, a conversion rate of 8,000 to 10,000 conversions per second [The sample rate in conversions per second or samples per second is often quoted in units of Hertz. We will use that terminology here, although conversions

and Countryrobots: Lend me your Ears

per second is a generalization of the concept of cycles per second] is sufficient for fairly good quality speech. With sample rates below about 6 kHz the speech quality begins to deteriorate badly because of inadequate frequency response.

Almost any microprocessor can easily handle the data rates described above to keep the DAC going. The next question is, where do the samples come from? One way to get them would be by sampling a real speech signal with a matching analog to digital converter (ADC) running at the same sample rate. You then have a complicated and expensive, but very flexible, recording system. Each second of speech requires 8 K to 10 K bytes of storage. If you want only a few words or short phrases, you could store the samples on a ROM or two and dump then sequentially to the DAC. Such a system appears in figure 3.

If you want more than a second or two of speech output, however, the amount of ROM storage required quickly becomes impractical. What can be done to minimize storage? Many words appear to have parts that could be recombined in different ways to make other words. Could a lot of memory be saved this way? A given vowel sound normally consists of several repetitions of nearly identical waveform segments with the period of repetition corresponding to the speech fundamental frequency or pitch. Figure 4 shows such a waveform. Within limits, an acceptable sound is produced if we store only one such cycle and construct the vowel sound by repeating this waveform cycle for the duration of the desired vowel. Of course, the pitch will be precisely constant over that entire interval. This will sound rather unnatural, especially for longer vowel durations, because the period of repetition in a naturally spoken vowel is never precisely constant, but fluctuates slightly. In natural speech the pitch is nearly always changing, whether drifting slowly or

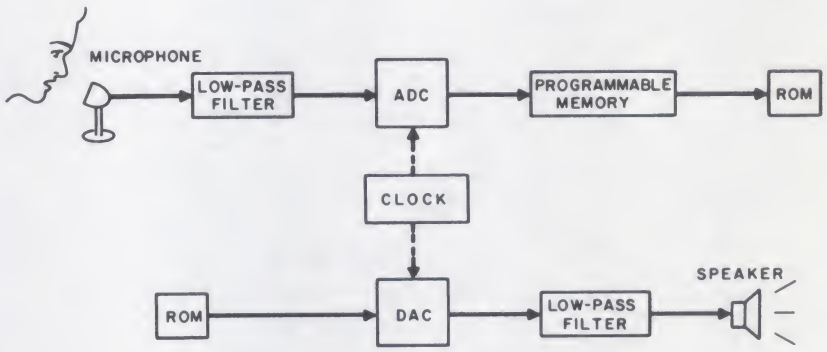


Figure 3: Waveform Playback from ROM Storage. One way to achieve a digitally controlled vocal output is to first digitize a passage of human speech, then store the digital pattern in memory. For a commercial product, such as a talking calculator, the limited vocabulary required makes this a feasible avenue of design, especially when a single mass produced ROM can be used in the final product. In an experimenter's system, the ROM is not needed, and programmable memory can be substituted during experiments. This is probably the least expensive way to augment an existing computer's capability with vocal output, but the memory requirements limit its use to small vocabularies. The quality of the result varies with the ADC (and DAC) sampling rate and precision.

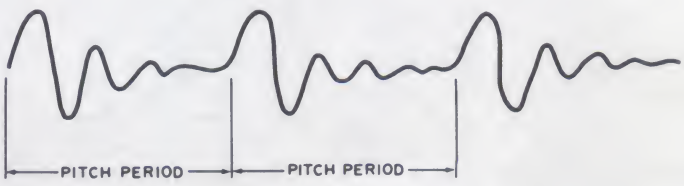


Figure 4: Typical Vowel Waveform. In principle, a vowel is a fairly long sustained passage of sound with repetitive characteristics. The vowel sounds are produced physiologically by the resonances of the vocal tract, and are controlled electronically by the formant filters which produce the equivalent of vocal tract resonances.

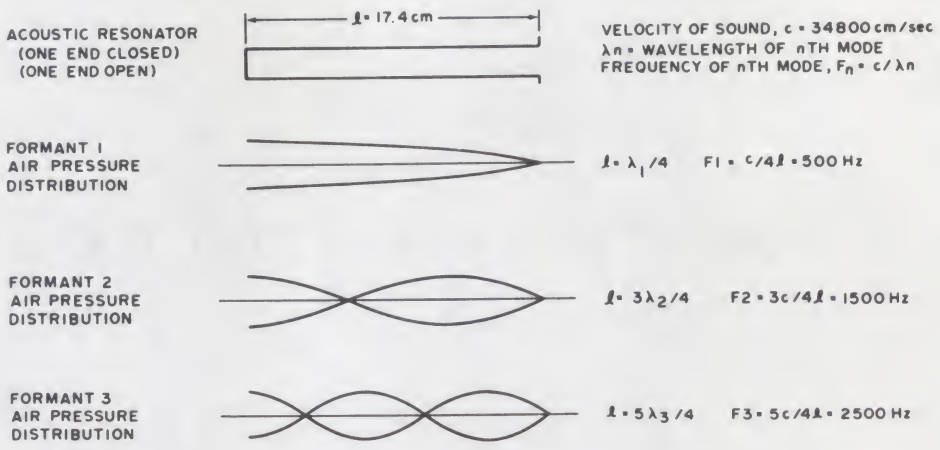


Figure 5: Tube Resonances. Temporarily ignore the complicated shape of the vocal tract and simplify it to a tube 17.4 cm long. Applying the equations of physics to acoustic waves in air gives resonances at several modes or natural frequencies. The standing waves along the tube at each frequency are shown, and identified as formant 1, formant 2 and formant 3. In the actual vocal tract, a more complicated and time varying geometry changes the resonances as a sound is created.

sweeping rapidly to a new level. It is of interest that this jitter and movement of the pitch rate has a direct effect on the perception of speech because of the harmonic structure of the speech signal. In fact, accurate and realistic modelling of the natural pitch structure is probably the one most important ingredient of good quality synthetic speech. In order to have smooth pitch changes across whole sentences, the number of separate stored waveform cycles still gets unreasonable very quickly. From these observations of the cyclic nature of vowels, let us move in for a closer look at the structure of the speech signal and explore more sophisticated possibilities for generating synthetic speech.

How Do We Talk?

The human vocal tract consists of an air filled tube about 16 to 18 cm long, together with several connected structures which make the air in the tube respond in different ways (see figure 1). The tube begins at the vocal cords, or glottis, where the flow of air up from the lungs is broken up into a series of sharp pulses of air by the vibration of the vocal cords. Each time the glottis snaps shut, ending the driving pulse with a rapidly falling edge, the air in the tube above vibrates or rings for a few thousandths of a second. The glottis then opens and the airflow starts again, setting up conditions for the next cycle.

Figure 6: "ah" as in "father." In figure 1, the vocal tract was shown in schematic form. Here is a similar figure showing how the tract has been modified to produce the vowel sound "ah." The human typically closes off the nasal cavity and widens out the oral cavity by opening the mouth during this sound.

The length of this vibrating air column is the distance from the closed glottis up along the length of the tongue and ending at the lips, where the air vibrations are coupled to the surrounding air. If we now consider the frequency response of such a column of air, we see that it vibrates in several modes or resonant frequencies corresponding to different multiples of the acoustic quarter wavelength. There is a strong resonance or energy peak at a frequency such that the length of the tube is one quarter wavelength, another energy peak where the tube is three quarter wavelengths, and so on at every odd multiple of the quarter wavelength. If a tube 17.4 cm long had a constant diameter from bottom to top, these resonant energy peaks would have frequencies of 500 Hz, 1500 Hz, 2500 Hz and so on. These resonant energy peaks are known as the formant frequencies. Figure 5 illustrates the simple acoustic resonator and related physical equations.

The vocal tract tube, however, does not have a constant diameter from one end to the other. Since the tube does not have constant shape, the resonances are not fixed at 1000 Hz intervals as described above, but can be swept higher or lower according to the shape. When you move your tongue down to say "ah," as in figure 6, the back part is pushed back toward the walls of the

throat and in the front part of the mouth the size of the opening is increased. The effect of changing the shape of the tube in this way is to raise the frequency of the first resonance or formant 1 (F1) by several hundred Hz, while the frequency of formant 2 (F2) is lowered slightly. On the other hand, if you move your tongue forward and upward to say "ee," as in figure 7, the size of the tube at the front, just behind the teeth, is much smaller, while at the back the tongue has been pulled away from the walls of the throat, leaving a large resonant cavity in that region. This results in a sharp drop in F1 down to as low as 200 or 250 Hz, with F2 being increased to as much as 2200 or 2300 Hz.

We now have enough information to put together the circuit for the oral tract branch of a basic formant frequency synthesizer. After discussing that circuit, we will continue on in this way, describing additional properties of the speech mechanism and building up the remaining branches of the synthesizer circuit.

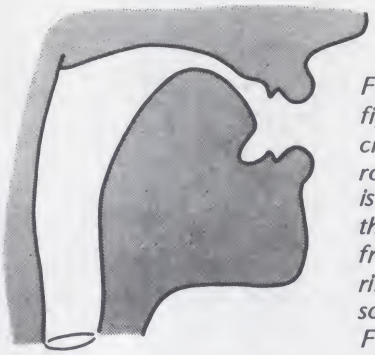


Figure 7: "ee" as in "heed." In contrast to figure 6, when the "ee" vowel sound is created, the mouth opening tends to be narrowed; and the upper end of the vocal tract is restricted. This lowers the frequency of the first resonant mode and raises the frequencies of the second and third. Referring to table 1, the "ee" vowel sound has some of the highest resonances for formants F2 and F3 and the lowest for F1.

Figure 8: Voiced Sounds from the Glottis. Sounds which have definite pitch are called voiced sounds. In the natural larynx, these sounds are generated by the vocal chords and drive the vocal tract at the glottis. In an electronic analog, the voiced sounds can be generated by a programmable counter (to set the frequency) which in turn creates a sine wave of the same frequency. A rectified sine wave is a good source for the glottal pulses used in the electronic model of a larynx used in the author's approach to speech generation.

A Speech Synthesizer Circuit

To start with, we must have a train of driving pulses, known as the voicing source, which represents the pulses of air flowing up thru the vibrating glottis. This could be simply a rectified sine wave as in figure 8. To get different voice qualities, the circuit may be modified to generate different waveform shapes.

This glottal pulse is then fed to a sequence of resonators which represent the formant frequency resonances of the vocal tract. These could be simple operational amplifier bandpass filters which are tunable over the range of each respective formant. Figure 9 shows the concept of a typical resonator circuit which meets our requirements. IC1, IC2 and IC4 form the actual bandpass filter, while IC3 acts as a digitally controlled resistance element serving to vary the resonant frequency of the filter. Several such resonator circuits are then combined as in figure 10 to form the vocal tract simulator. The voicing amplitude control, AV, is another digitally controlled resistance similar to IC3 of figure 9.

This gain controlled amplifier configuration is the means by which the digital computer achieves its control of speech signal elements. The data of one byte drives the switches to set the gain level of the amplifier in question. In figures 10, 13 and 15 of this article, this same variable resistance under digital control is shown symbolically as a resistor with a parameter name,

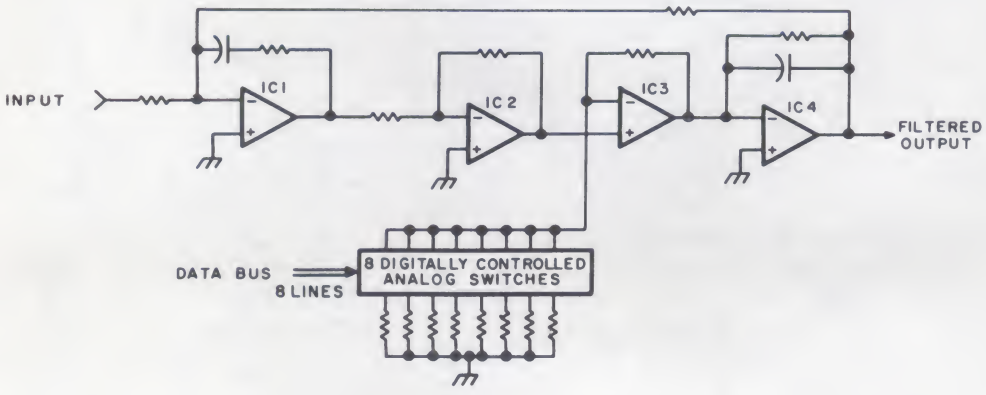


Figure 9: Typical Formant Resonator Circuit. A digitally controlled band pass filter can be built from four operational amplifiers and 8 digitally controlled analog switches. The filter characteristics are set by the choice of the resistance and capacitance elements as well as the digital control word. The operational amplifier IC3 serves as a gain controlled amplifier in the feedback loop, which alters the filter resonance.

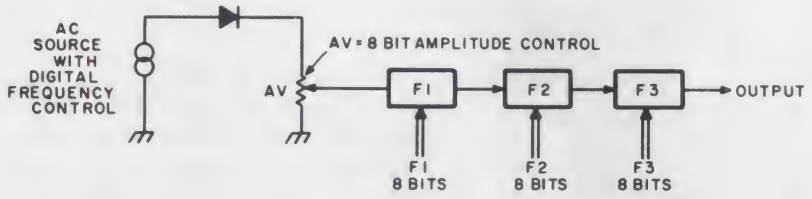


Figure 10: A first approximation of the voice synthesizer can be constructed by using three formant filters in series with differing resonance settings all controlled by 8 bit digital words. The resistance indicated as AV is an operational amplifier circuit (see IC3 of figure 9) with a digital gain control input. It is thus a programmable element of gain less than unity, in other words the electronically controlled equivalent of a variable resistance. This notation of a controlled resistance is used in figures 13 and 15 as well.

	F1	F2	F3
heed	250	2300	3000
hid	375	2150	2800
head	550	1950	2600
had	700	1800	2550
hod	775	1100	2500
paw	575	900	2450
hood	425	1000	2400
who	275	850	2400

Table 1: Steady State English Vowels. The vowel sounds are made by adjusting the formant resonances of the human vocal tract to the frequencies listed in this table. These figures are approximate, and actual formant resonances vary from individual to individual. In a speech synthesizer based upon an electronic model of the vocal tract, the formant frequencies are set digitally using operational amplifier filters with adjustable resonant peaks.

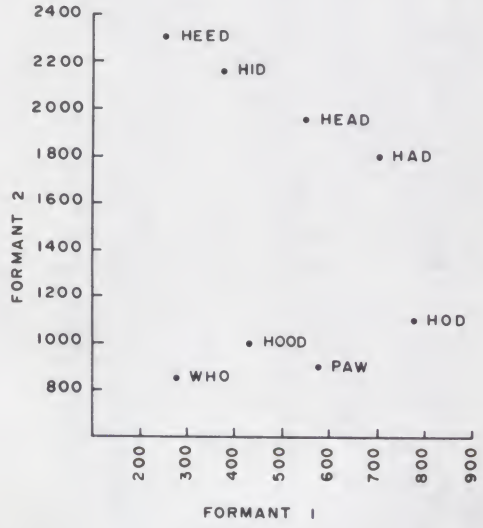


Figure 11: The Steady State English Vowels. The distinctions between various vowel sounds can be illustrated by plotting them on a two dimensional graph. The horizontal axis is the formant 1 frequency, the vertical axis is the formant 2 frequency. A location for each vowel utterance can be determined experimentally by locating the resonance peaks with an audio spectrum analyzer.

rather than as an operational amplifier with analog switches.

Generating Vowel Sounds

The vocal tract circuit as shown thus far is sufficient to generate any vowel sound in any human language (no porpoise talk, yet). Most of the vowels of American English can be produced by fixed, steady state formant frequencies as given in table 1. A common word is given to clearly identify each vowel. The formant frequency values shown here may occasionally be modified by adjacent consonants.

An alternative way to describe the formant relationships among the vowels is by plotting formant frequencies F1 vs F2 as in figure 11. F3 is not shown here because it varies only slightly for all vowels (except those with very high F2, where it is somewhat higher).

The F1-F2 plot provides a convenient space in which to study the effects of different dialects and different languages. For example, in some sections of the United States, the vowels in "hod" and "paw" are pronounced the same, just above and to the right of "paw" on the graph. Also, many people from the western states pronounce the sounds in "head" and "hid" alike, about halfway between the two points plotted for these vowels on the graph.

A few English vowels are characterized by rapid sweeps across the formant frequency

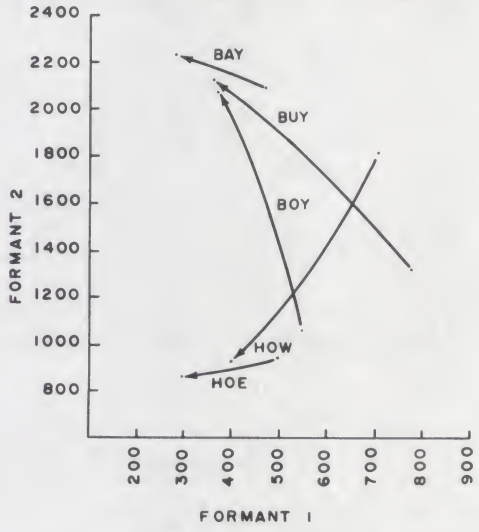


Figure 12: English Diphthongs. A diphthong is a sound which represents a smooth transition from one vowel sound to another during an utterance. The time duration of the swap from one point to another in formant space is typically 150 to 250 ms. This graph shows typical starting and ending points for several common diphthong sounds.

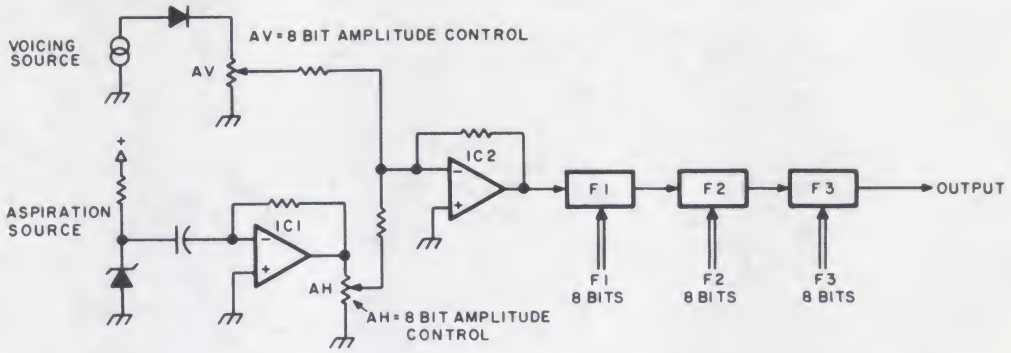


Figure 13: Synthesizer with Aspiration Noise Generator. Not all utterances are vowels. By adding a digitally controlled noise generator to the circuit of figure 10, it is possible to synthesize the consonant sounds known as "stops." In this circuit, the amplitude versus time characteristics of the noise pulse are determined by an 8 bit programmable gain control AH (shown symbolically as a resistor). The output of the noise source is mixed with the voicing source with the analog sum being routed to the formant filters. The noise generator is a zener diode.

space rather than the relatively stable positions of those given in table 1. These sweeps are produced by moving the tongue rapidly from one position to another during the production of that vowel sound. Approximate traces of the frequency sweeps of formants F1 and F2 are shown in figure 12 for the vowels in "bay," "boy," "buy," "hoe" and "how." These sweeps occur in 150 to 250 ms roughly depending on the speaking rate.

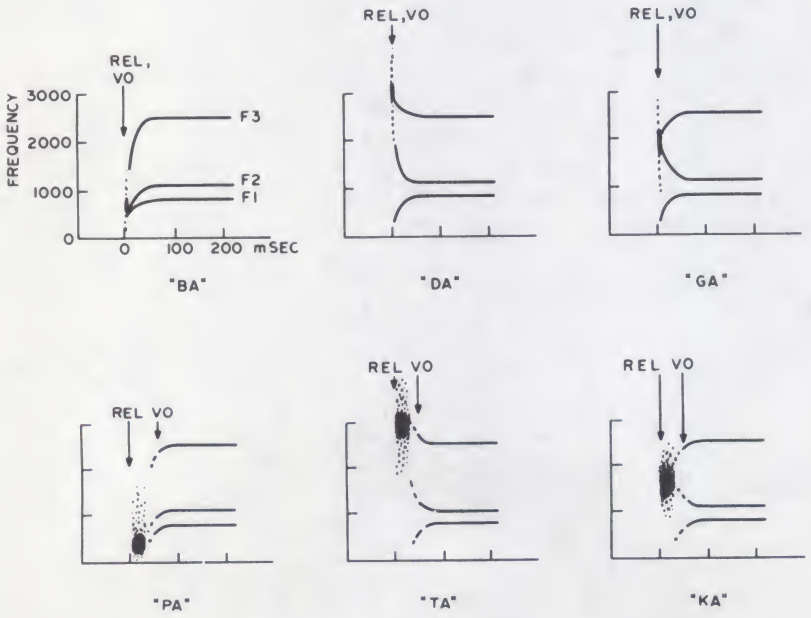


Figure 14: Stop Consonant Patterns. This figure illustrates 6 different stop consonant patterns. The release of the stop closure (start of noise pulse) is at the point marked by "Rel" and the beginning of the voicing sounds is marked by "VO". Note the typical transition of the vowel formants as the steady state is reached.

Consonant Sounds

Consonant sounds consist mostly of various pops, hisses and interruptions imposed on the vibrating column of air by the actions of several components of the vocal tract shown in figure 1. We will divide them into four classes: 1) stops, 2) liquids, 3) nasals, and 4) fricatives and affricates. Considering first the basic 'stop consonants,' "p," "t," "k," "b," "d" and "g," the air stream is closed off, or stopped, momentarily at some point along its length, either at the lips, by the tongue tip just behind the teeth or by the tongue body touching the soft palate near the velum. Stopping the air flow briefly has the effect of producing a short period of silence or near silence, followed by a pulse of noise as the burst of air rushes out of the narrow opening.

The shape of the vocal tract with the narrow opening at different points determines the spectral shape of the noise pulse as well as the formant locations when voicing is started. Both the noise burst spectrum and the rapid sweeps of formant frequency as the F1-F2 point moves into position for the following vowel are perceived as characteristic cues to the location of the tongue as the stop closure is released. We need only add a digitally controlled noise generator to the vocal tract circuit of figure 10 to simulate the noise of the burst of air at the closure release and we can then generate all the stop consonants as well as the vowels. Figure 13 shows the speech synthesizer with such a noise generator added. The breakdown noise of a zener diode is amplified by IC1 and amplitude is set by the digitally controlled resistor AH. IC2 is a mixer amplifier which combines the glottal source and aspiration

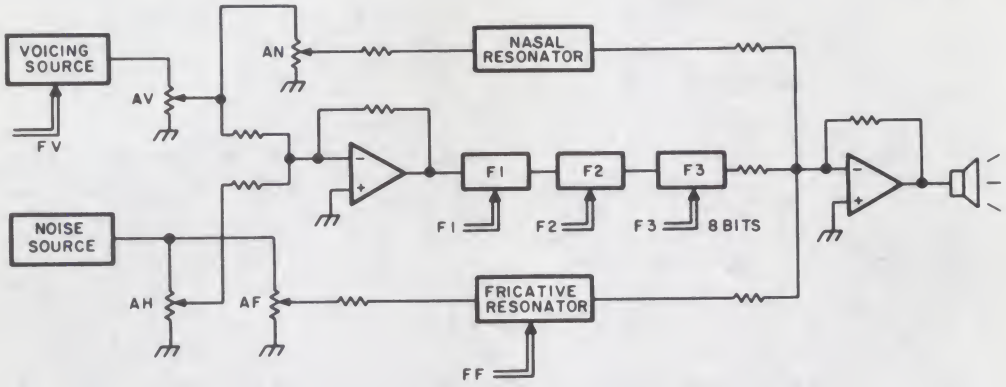


Figure 15: The Complete Synthesizer. This diagram shows the organization of a complete synthesizer which includes a wide variety of parameters. The voicing frequency and amplitude are set by parameters FV and AV. The noise pulses of stop consonants are generated with the programmable gain element AH. The fricative resonator with amplitude AF and frequency resonance FF are used to generate fricatives like "s" and "sh." The normal vowel sounds are generated by control of the formant frequencies F1, F2 and F3, and a nasal resonator with amplitude AN and fixed frequency characteristics is used to add varying amounts of nasal sounds. The result of signals processed through the nasal, formant and fricative paths is summed by a final operational amplifier and used to drive the output speaker.

noise at the input to the formant resonators.

It is important to notice at this point the range of different sounds that can be generated by small changes in the relative timing of the control parameters. The most useful of these timing details is the relationship between the pulse of aspiration noise and a sharp increase in the amplitude of voicing (see figure 14). For example, if we set the noise generator to come on for a noise pulse about 40 ms long and immediately after this pulse, F1 sweeps rapidly from 300 up to 775 Hz and F2 moves from 2000 down to 1100 Hz, the sound generated will correspond to moving the tip of the tongue down rapidly from the roof of the mouth. Observe, however, that the formant output is silent after the noise pulse until the voicing amplitude is turned up. If voicing is turned on before or during a short noise burst, the circuit generates the sound "da," whereas if the voicing comes on later, after a longer burst and during the formant frequency sweeps, the output sounds like "ta." This same timing distinction characterizes the sounds "ba" vs "pa" and "ga" vs "ka," as well as several other pairs which we will explore later. Figure 14 gives the formant frequency patterns needed to produce all the stop consonants when followed by the vowel "ah." When the consonant is followed by a different vowel, the formants must move to different positions corresponding to that vowel.

The important thing to note about a stop transition is that the starting points of the frequency sweeps correspond to the point of closure in the vocal tract, even though these sweeps may be partially silent for the unvoiced stops "p," "t" and "k," where the voicing amplitude comes on after the sweep has begun.

The second consonant group comprises the liquids, "w," "y," "r" and "l." These sounds are actually more like vowels than any of the other consonants except that the timing of formant movements is crucial to the liquid quality. "W" and "y" can be associated with the vowels "oo" and "ee," respectively. The difference is one of timing. If the vowel "oo" is immediately followed by the vowel "ah," and then the rate of F1 and F2 transitions is increased, the result will sound like "wa." A comparison of the resulting traces of F1 and F2 vs time in "wa" with the transition pattern for "ba" in figure 14 points out a further similarity. The

	Resonator Frequency (FF)	Fricative Amplitude (AF)
sh, zh	2500	.9
s, z	5000	.7
f, v	6500	.4
th	8000	.2

Table 2: Fricative Spectra. A fricative sound typically consists of a pulse of high frequency noise. The various types of fricatives are classified according to the spectral profile of the pulse. For the electronic model described here, the fricative amplitude and resonator frequency for several sounds are listed in this table.

direction of movement is basically the same, only the rate of transition of "ba" is still faster than for "wa." Thus we see the parallelism in the acoustic signal due to the common factor of lip closeness in the three sounds "ua," "wa" and "ba." "Y" can be compared with the vowel "ee" in the same way, so the difference between "ia" and "ya" is only a matter of transition rates. Generally, "l" is marked by a brief increase of F3, while "r" is indicated by a sharp drop in F3, in many cases, almost to the level of F2.

The third group of consonants consists of the nasals, "m," "n" and "ng." These are very similar to the related voiced stops "b," "d" and "g," respectively, except for the addition of a fixed "nasal formant." This extra formant is most easily generated by an additional resonator tuned to approximately 1400 Hz and having a fairly wide bandwidth. It is only necessary to control the amplitude of this extra resonator during the "closure" period to achieve the nasal quality in the synthesizer output.

The fourth series of consonants to be described are the fricatives, "s," "sh," "z," "zh," "f," "v" and "th" and the related affricates "ch" and "j." The affricates "ch" and "j" consist of the patterns for "t" and "d" followed immediately by the fricative "sh" or "zh," respectively, that is, "ch" = "t+sh" and "j" = "d+zh." The sound "zh" is otherwise rare in English. An example occurs in the word "azure." With the letters "th," two different sounds are represented, as contained in the words "then" and "thin." All the fricatives are characterized by a pulse of high frequency noise lasting from 50 to 150 msec. The first subclassification of fricatives is according to voicing amplitude during the noise pulse, just as previously described for the stop consonants. Thus, "s," "sh," "f," "ch" and "th" as in "thin" have no voicing during the noise pulse, while "z," "zh," "v," "j" and "th" as in "then" have high voice amplitude. When a voiceless fricative is followed by a vowel, the voicing comes on during the formant sweeps to the vowel position, just as in the case of the voiceless stops. The different fricatives within each voice group are distinguished by the spectral characteristics of the fricative noise pulse. This noise signal differs from that previously described for the stop bursts in that it does not go thru the formant resonators, but is mixed directly into the output after spectral shaping by a single-pole filter. Table 2 gives the fricative resonator settings needed to produce the various fricative and affricate consonants. Fricative noise amplitude settings are shown on a scale of 0 to 1.

Product Information

At the time this article goes to press, a synthesizer module incorporating several detail refinements and improvements over the circuits of this article is being developed by the author and associates. A detailed user's guide will be supplied with the Computalker module which illustrates the timing relationships needed to produce all the consonant-vowel and vowel-consonant combinations which occur in natural speech. This can serve as a reference guide for creating your speech output software which generates the proper control patterns from text inputs. Write to Computalker, 821 Pacific St No. 4, Santa Monica CA 90405 for the latest information on this module.

The Complete Synthesizer

The system level diagram of a complete synthesizer for voice outputs is summarized in figure 15. The information contained in this article should be sufficiently complete for individual readers to begin experimenting with the circuitry needed to produce speech outputs. In constructing a synthesizer on this model, the result will be a device which is controlled in real time by the following parameters:

- AV = amplitude of the voicing source, 8 bits
- FV = frequency of the voicing source, 8 bits
- AH = amplitude of the aspiration noise component, 8 bits
- AN = amplitude of the nasal resonator component, 8 bits
- AF = amplitude of the fricative noise component, 8 bits
- F1 = frequency of the formant 1 filter, 8 bit setting.
- F2 = frequency of the formant 2 filter, 8 bit setting.
- F3 = frequency of the formant 3 filter, 8 bit setting.
- FF = frequency of fricative resonator filter, 8 bit setting.

This is the basic hardware of a system to synthesize sound; in order to complete the system, a set of detailed time series for settings for these parameters must be determined (by a combination of the theory in this article and references, plus experiment with the hardware). Then, software must be written for your own computer to present the right time series of settings for each sound you want to produce. Commercial synthesizers often come with a predefined set of "phonemes" which are accessed by an appropriate binary code. The problem of creating and documenting such a set of phonemes is beyond the scope of this introductory article, but is well within the dollar and time budgets of an experimenter.■

BIBLIOGRAPHY

1. Erman, Lee, ed, IEEE Symposium on Speech Recognition, April, 1974, Contributed Papers, IEEE Catalog No. 74CH0878-9 AE.

2. Flanagan, J L, and Rabiner, L R, eds, *Speech Synthesis*, Benchmark Papers in Acoustics, Dowden, Hutchinson & Ross, Inc, 1973.

3. Lehiste, Ilse, ed, *Readings in Acoustic Phonetics*, MIT Press, 1967.

4. Moschytz, George S, *Linear Integrated Networks Design*, Van Nostrand, New York, 1975.

Magnetic Recording for Computers

William A Manly
Cobaloy Co
626 Great Southwest Pkwy
Arlington TX 76011

Why Magnetic Recording?

Anyone seriously involved with computers, whether he likes it or not, will also be seriously involved with magnetic recording. After one begins working with computers, it doesn't take very long to discover the shocking fact that memory for a computer is going to cost a lot more than the computer itself. A computer requires lots of memory, and professional or amateur, the computer user wants to minimize the cost of his computer setup. A look at figure 1 will immediately tell you why magnetic recording is so important to computer memories: Nothing can come anywhere near it for low cost per unit of stored information. Figure 1 also shows why magnetic recording cannot be used for all types of computer memories: It is the slowest of the memories, which means that it is employed mostly for long term, low usage storage (usually called bulk storage).

Nothing can come anywhere near magnetic recording for low cost per unit of stored information.

All Kinds of Recorders –

Magnetic recorders come in many forms: tape, disk, drum, card, sheet, stripe, roll, cassette, reel, . . . etc. Most of these forms have been used for computer memories in the past, and many are still in use.

And Recording Methods

There are several ways of placing magnetic signals on magnetic media. Among these are those which use the hysteresis loop or the initial magnetization curve, those which use a variation of anhysteretic magnetization, and some methods which use Curie point magnetization. I will go through the first two in detail. The last one involves heating the medium until it is so hot that it is no longer magnetic (it ceases being magnetic at a temperature called the Curie point), then letting it cool in the recording field until it again becomes magnetic. Due to the inconvenience of the temperature cycling, this last method is not important for digital recording. The first method will be covered in the greatest detail, as most recorders designed for digital use employ it. Many of the conclusions drawn will also apply to the second method.

Some other names and subdivisions also apply to the main divisions given above. If we call the first type hysteresis recording, there are two main subdivisions. One is very much like FM radio broadcasting, and is also called frequency modulation recording (sometimes called phase modulation). A single-frequency carrier is recorded on the medium, and its frequency changed according to the information to be stored. Another subdivision is the type used for most digital work. It is called saturation recording. Ideally, the saturation recorded medium has only two states: saturated (magnetized to maximum strength) in one direction, or saturated in the other direction. The information is contained in the transitions, where the direction of saturation is changed. (One older method also used a third state; that of erasure, or zero magnetization.) The second type of recording (anhysteretic magnetization) is also called biased recording. It involves the use of a large amplitude high frequency bias, to which the signal is added. The signal does not modulate (change) the bias in any way. The bias does not return during the signal playback process.

Although the professionals normally use only saturation recording for digital use, computer hobbyists have appropriated recorders intended for other uses, and thus use several types of recording. One is even a type of FM recording using bias to record the carrier. Magnetic recording can also be

classified according to the type of information being recorded, and there is a correlation between the type of information and the type of recording:

Type Of Information	Type Of Recording
Digital professional	Saturation (sometimes FM carrier)
Audio	Biased
Instrumentation	Biased, biased FM carrier, FM carrier
Video	FM carrier
Digital hobbyist	Biased FM carrier, saturation

All of the foregoing seems rather involved, but just remember that the knowledge of a few basics will enable you to sort out almost any recording situation. For instance, all the systems we will discuss involve only a magnetic surface moving with respect to a set of magnetic heads, one of which writes on the surface, and another which reads the information previously written there (if you are an audio enthusiast, forget about the record, playback, and erase heads – those terms are rarely used in digital recording). You are not likely to have an erase head in your system unless you use an audio recorder. Some systems are especially simple, having only one head which both reads and writes. Sometimes the surface moves and the heads are fixed; sometimes the heads move and the surface is fixed; sometimes they both move; but the important thing is the relative head to surface movement.

A Plan of Attack

It isn't very likely that you are interested in becoming an expert on magnetic recording. All that you want is to understand it well enough so you can exercise enough care to prevent its becoming a problem. Knowing this, I'll just present enough of what is called the theory of recording to give you a feel for how it works, then I'll talk a bit of practicalities with suggestions for smooth operation and maintenance. Magnetic recording theory is divided into two parts: Magnetics and geometry. Let's first look at the magnetics.

Blame It All on the Electron!

Almost everyone knows that the electron is a fundamental particle of electricity. It also possesses a magnetic field (electrons always have spin; this spin constitutes an electric current going around in a circle; and anytime an electric current is flowing, it generates a magnetic field). Most materials have their electrons placed in such a way that the magnetic fields all balance out to zero, but there are a few materials which don't. With electron spins paired so that one is spinning clockwise and one counterclockwise, the net field is zero. Of the materials with unpaired electron spins, some are put together in such a fashion that the electrons are coupled together. When this happens, if you manage to turn one spin axis, you have to turn its neighbors as well (the magnetic fields point along the spin

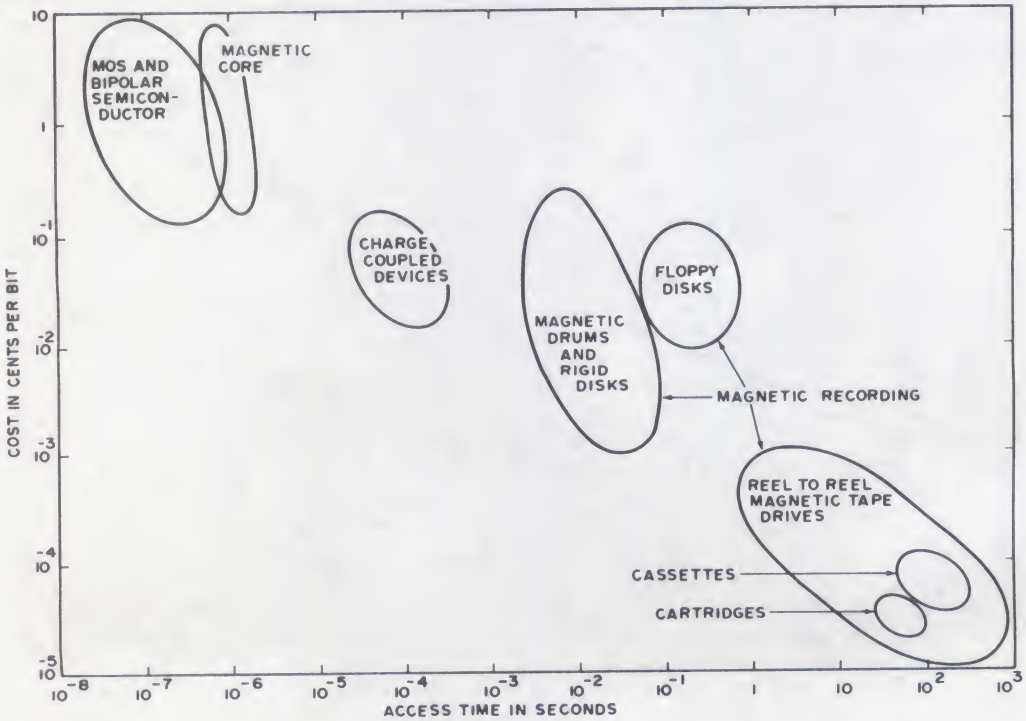


Figure 1: Digital computer memory hierarchy: cost as a function of access time.

axis). Depending on the material, somewhere between a few hundred and a few million of these little fellows will stay coupled together and pointed in the same way all the time. This collection of coupled electron spins is called a domain, and the materials with this type of structure are called ferromagnetic materials.

If a large number of atoms are collected together, there will be two or more domains, whose magnetic fields will not necessarily be pointing in the same direction (though they might). Materials for magnetic recording consist either of domain sized particles separated by non-magnetic material, or they are made of plated material with enough impurities to section the plating into domain sized units. Separating the domains this way allows them to operate nearly independently – a necessity for keeping the information in storage. Such materials are known as "hard" magnetic materials.

Hysteresis, Not Hysteria

A hard ferromagnetic material is characterized by its hysteresis loop. I have a library full of books on hysteresis loops, which have been confusing students for years; but let me see if I can spare you some of the confusion. Suppose we have a material containing a large number of domains whose fields are all pointing in different directions. The fields all cancel out, and the material is said to be demagnetized (note that a single domain cannot be demagnetized). If a very small magnetic field is applied to the material, nothing happens. As the strength of the field is increased, a few of the domains swing their electron spin axes to follow the applied field. As the field strength continues to rise, more and more domains follow the field until finally the last domain responds. After that, no matter how much more field is applied, nothing more can happen. The material is now saturated, and it now has acquired its maximum magnetization, designated M_m. This process is known as the initial magnetization of the material. If we now let the applied field go to zero, a few of the domains decide to desert the pack, but most stay pointing in the same direction. This is known as the remanent condition, with the remanent magnetization designated M_r. Magnetization is given in several units, all of which are measures of how many unpaired electron spins there are per unit volume or unit weight of magnetic material.

Now let's reverse the direction of the field (denoted, for some reason, by the letter "H") and slowly increase the strength from zero. At some point, exactly half of the domains have decided to follow the new field direction, half are still pointed in the other direction; and the result is zero. At this point, the applied field is called the coercive field (sometimes called coercivity or coercive force) of the material, and is indicated by H_c. If the applied field is increased to the former high level, the material again becomes saturated, but in the opposite direction. This cycle can be continued indefinitely, but the material never returns to its erased condition (zero magnetization in the material with zero applied field). If the first direction is chosen to be positive (and the opposite direction negative), we can show a graph of the whole business by plotting magnetization on the Y axis, positive direction up; and the strength of the applied field on the X axis, positive direction to the right. This plot is known as a hysteresis loop, and is shown in figure 2; along with the initial magnetization curve, which is not properly part of the hysteresis loop.

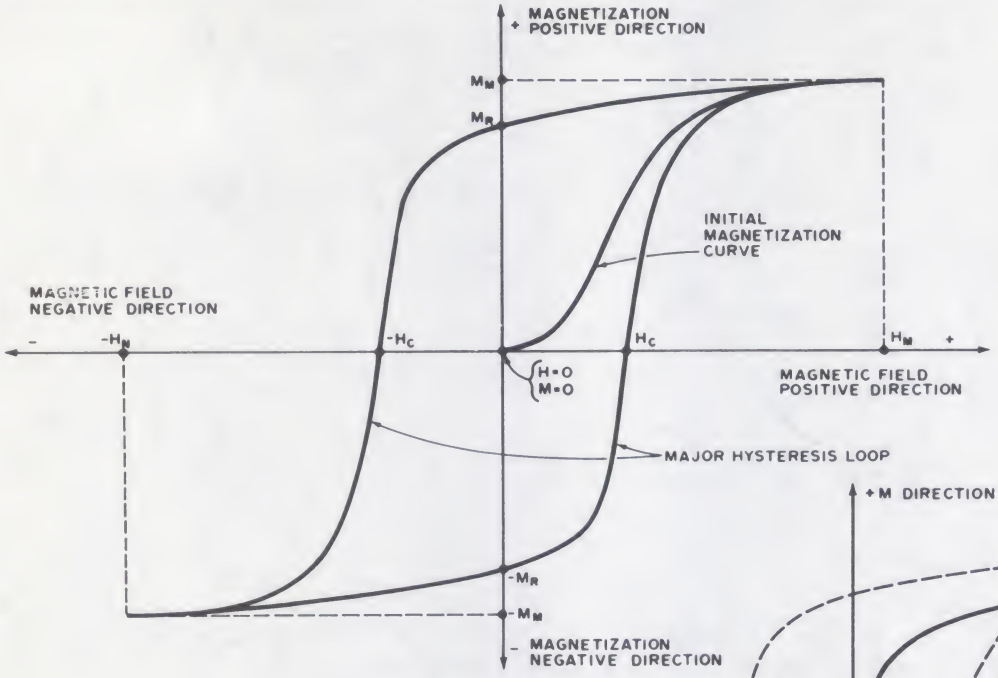
Erasure

If we could limit the discussion to saturation recording, I would have been through with the magnetics right now, but the use of audio recorders has complicated things, so there's a bit more. Suppose we are cycling around the major hysteresis loop we have just described, but start reducing the maximum field a bit each time around. Each time the maximum field is reduced, the loop shrinks in the horizontal direction, and in the vertical direction as well. These smaller loops are hysteresis loops too, but they are called "minor loops." If we continue to cycle, but reduce the maximum field gradually (i.e., go around 10 to 100 times) to zero, the remanence (the magnetization when the field is zero) goes to zero as well. Now we have reduced the magnetic material to the erased condition. It would be well to understand this before going on to the next part, since this cycling and reduction procedure is the basis for biased recording.

Some Recorders Are Biased

Now let's go back to the saturated condition. This time we will apply two fields added together. One is the same large cyclic field we applied in the last paragraph, but the other is a smaller field. The smaller field is applied and held constant. The large field is taken to saturation, then cycled and reduced to zero as in the erasure process. Then the small field is also reduced to zero. Now, the remanent magnetization is *not* zero. In fact, it is larger than one might expect from the application of that small field. This remanence is called anhysteretic

Figure 2: Initial magnetization curve and major hysteresis loop of a hard ferromagnetic material. See text for explanation of abbreviations.



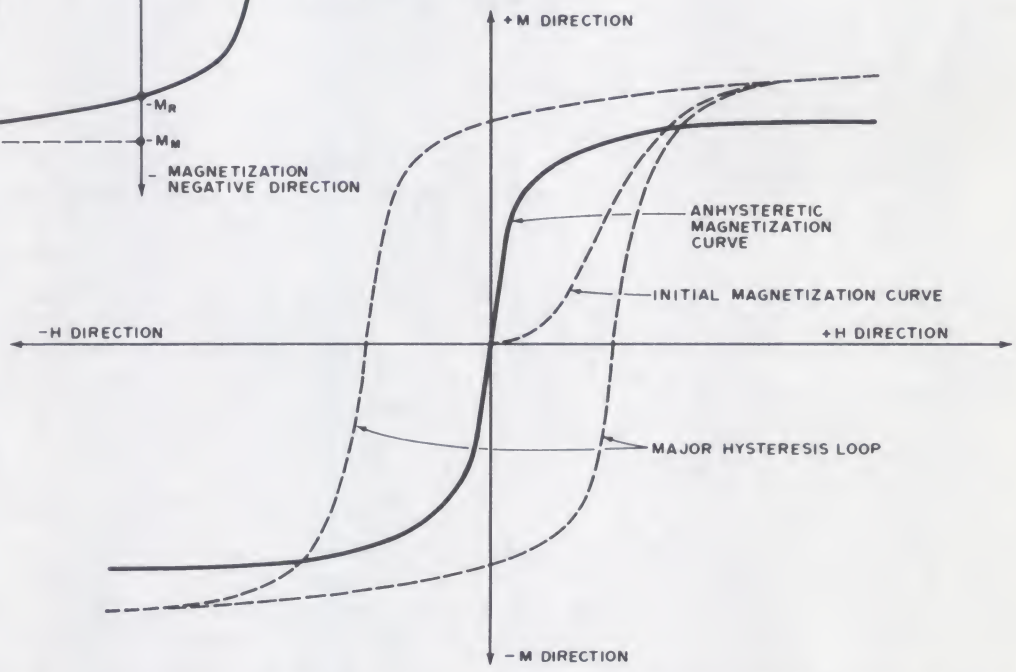


Figure 3: Anhysteretic magnetization curve of a hard ferromagnetic material, compared to the initial magnetization curve and a major hysteresis loop.

remanence. Figure 3 shows a plot of the anhysteretic remanence (solid line) plotted against the small applied field, with the major hysteresis loop shown with dashed lines. This is a transfer curve, which is measured point-by-point, and is not continuous like the hysteresis loop. Note how linear this curve is, and that it is nearly parallel to the sides of the major hysteresis loop. This anhysteretic process is similar to how biased recording works. The large cyclic field is called the bias, and the small DC field is called the "signal."

If a field is applied to an erased medium and then removed, there is some remanent magnetization. If we plot this remanence versus various values of applied field, the curve looks like the solid line in figure 4. Compare it to the linear anhysteretic magnetization curve, which is the dashed curve in figure 4. Its nonlinearity prevents it from being used for audio and other types of recording requiring a linear transfer curve. Note particularly that there is very little remanence until the maximum field is at least as large as H_C. This curve is also a point-by-point curve like the anhysteretic magnetization curve.

An Assist From Euclid

We've covered about all the magnetics you're going to need, so we'll get right into the geometry of the situation. Magnetic

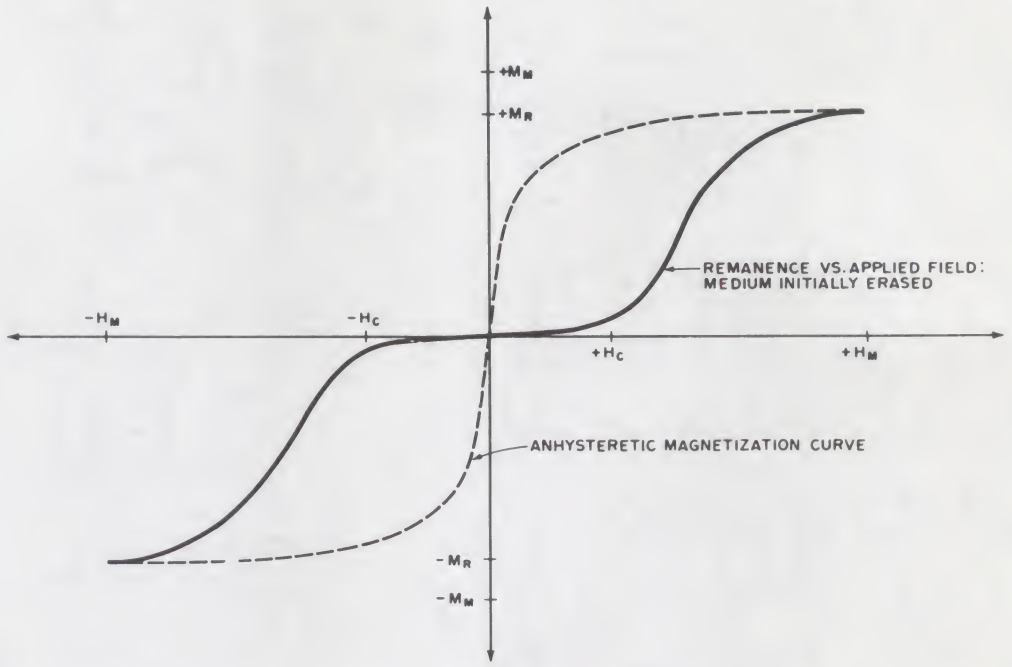


Figure 4: Remanence as a function of applied field for an initially erased hard ferromagnetic medium, as compared to the anhysteretic magnetization curve.

The principal methods of magnetic recording are hysteretic and anhysteretic magnetization.

recording is, fortunately, a two-dimensional process. This means that we can understand most of what we need to know by using only a two-dimensional picture, and the third dimension is thrown in as an afterthought. One of the two important dimensions lies along the recording surface in the direction of head-to-surface movement. The other important dimension is perpendicular to the recording surface, and measures the thickness of the magnetic medium and the head-to-surface spacing. The afterthought dimension measures the magnetic track width. It has to be considered, but it's not nearly so important as the other two.

The particular geometry we'll consider is that of a thick coating. This is the situation with floppy disks, and we'll use them as our primary example. (IBM, who invented the floppies, calls them diskettes. Another term is flexible disks.) The Philips-type cassette is also usually a thick coating (we'll use coating and medium interchangeably) situation, while rigid disks, drums, and most reel-to-reel and cartridge situations are thin media. Thick and thin refer to the ratio of the medium thickness to the write gap length, not to any absolute value of thickness. A thick medium situation exists when that ratio is greater than 0.5, and thin medium situations exist when the ratio is smaller than that. The exact size of the ratio dividing the two cases is a bit arbitrary. Probably not too many computer hobbyists have floppies as yet; but by using a thick medium as an example, we can include characteristics of thin media as a special case. Another reason for picking the floppy is that it uses a type of recording simpler than cassettes use; but, by analyzing it, we can cover all the major principles.

Heads Up!

A ring type head is shown in figure 5a. There are many other types of heads, but this one is well known and widely used, and the principles are analogous for most of the others. Note that this head is balanced: There are similar coils on both sides, and similar gaps on both the top and the bottom. A balanced head has a great resistance to pickup of stray fields, and is used where hum pickup might be a problem. A lot of digital heads are not balanced, and have only one coil, as in figure 5b. Read and write heads usually differ only in detail (gap and track dimensions), or the same head can be used for both functions. Floppy disk drives usually have only one dual purpose head.

In figure 6, I have blown up the outer edge of the top head gap, and show it contacting the magnetic medium. The actual dimensions of most floppy disk head gap lengths and the coating thicknesses of most floppy disks are about the same: 100 millionths of an inch (100 microinches or 2.54 micrometers).

We're Always Blowing Bubbles

When we create a magnetic field in the write head by passing an electric current through the head coils, the field stays inside the core until it reaches the gap, where it balloons out like a weak spot in an inner tube. Since the head gap is small, the field bubble is confined to a rather small volume. Near the corners of the gap edge, the field rises to a rather high value, even with only a small field in the head core. If the field in the magnetic medium is much higher than the coercivity of the medium, the magnetization of the medium begins to follow the field, and we say that it is being switched. Subsequently, if we allow the field to drop below the coercivity, the magnetization stays pointed in the same direction as the last applied field, and is more or less proportional to the difference between the highest applied field strength and the coercivity (up to the point where the highest applied field strength saturates the material).

Now refer back to the curve "Remanence versus applied field," in figure 4. If we set the write current at a moderate level, some part of the medium is experiencing fields from above saturation (H_m) down to nearly zero. In region A (figure 6) the fields are greater than H_m. In region B the fields are less than H_c and there is little magnetization. The part of the medium in the recording zone (figure 6) will experience a substantial amount of remanence after the field goes to zero (the part of the curve in figure 4 between H_c and H_m). The part of figure 6 labeled "Recording Zone — Low Drive" is a transition region, where some of the material is following the field, and some is not. For most materials, the boundaries are not sharp as shown, but are actually rather fuzzy.

As the medium moves away from the head gap, the part of it which has been in the recording zone has a signal impressed

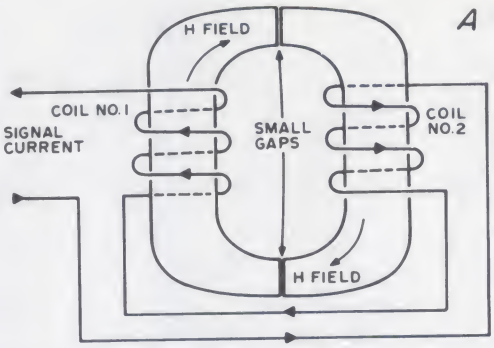


Figure 5: (a) Magnetic ring head, with balanced coils and gaps. (b) Magnetic head with single coil and gap.

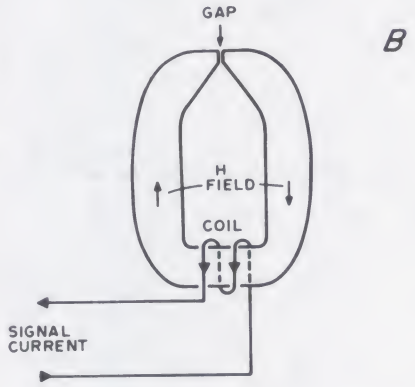
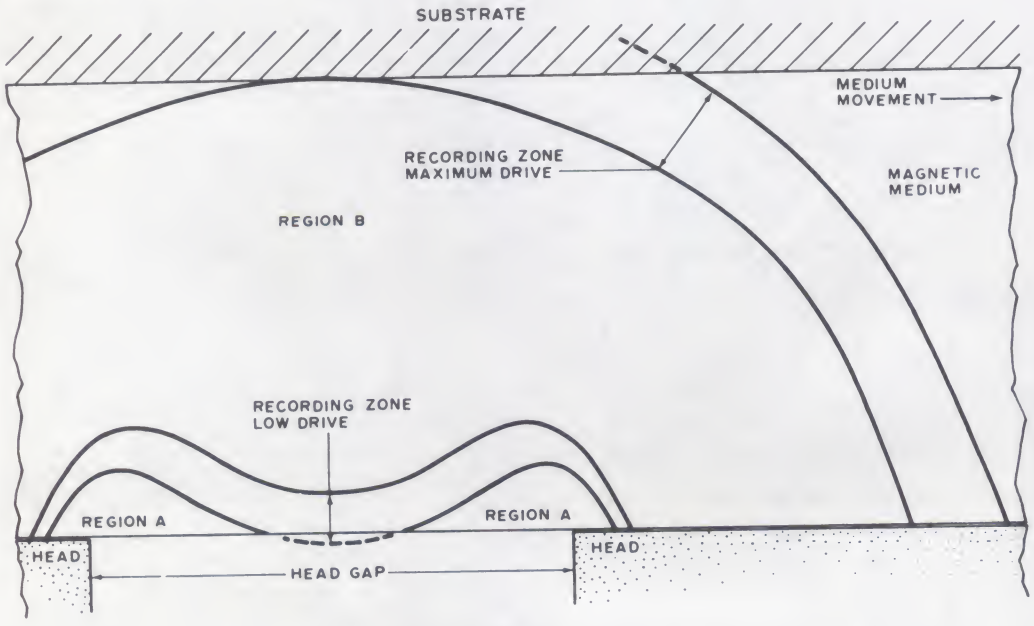


Figure 6: Write head near gap, in contact with magnetic recording medium. Total field near recording zones shown for low drive and maximum output drive.



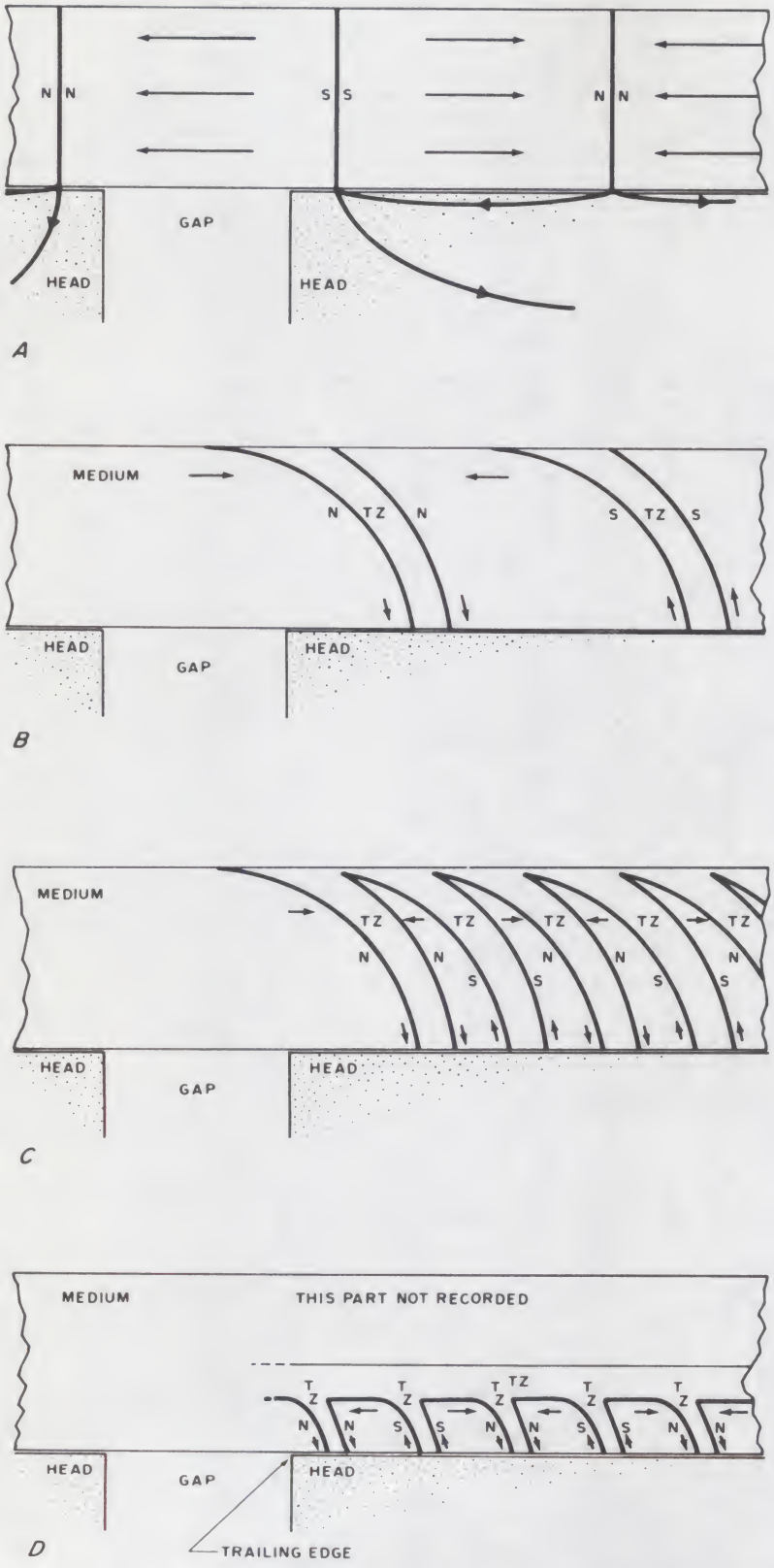


Figure 7: (a) Read head near gap, in contact with ideally magnetized recording medium. (b) Write head near gap, showing two maximum output flux transitions. (c) Same as (b), but with flux transitions extremely crowded. (d) Same as (c), except that write drive has been reduced to relieve crowding effects. Arrows show magnetization direction. N: North-seeking poles; S: South-seeking poles; TZ: Transition Zone.

upon it, while the part farthest away from the head has not seen a field high enough to leave a signal. We can record through the whole coating by increasing the drive through the head coils. With a higher drive current, the transition region is labeled "Recording Zone – Maximum Drive" (figure 6). Note how the width of the transition region has increased; this is a fundamental limitation of the medium and the head.

Things Are in a State of Flux

Figure 7 shows a series of diagrams of the magnetic flux (lines of force) patterns for a reading/writing situation similar to the floppy disk geometry. The flux intensity bounds determining the transition zones are also shown for the writing situation.

Figure 7a shows the head reading, and figures 7b through 7d show the head during a writing sequence. To simplify things in 7a, the ideal recorded flux pattern (which resembles bar magnets laid end-to-end) is shown. Actual flux patterns are similar, but more complicated. Observe that the flux from all the magnetized segments is shorted out by the head, except for the segment across the gap. In that case, the flux threads itself all the way around the head. When the next magnetized region moves into place, the flux will go in the opposite direction. The head coils have an output voltage only when the flux changes from one direction to the other; and the faster it changes, the higher the output voltage will be.

Floppies Are for Real

Now let's look at a real recording situation. The simplest coding is not used by floppy disk machines; but it illustrates all the principles, and is easiest to understand. It is called NRZ1 (Non-Return to Zero, change at 1) recording. The track is divided into small segments all the same length. If the recording is at a bit packing density of 800 bpi (bits per inch), the segments are 1/800 inches long, or 0.00125 inches (32 micrometers). The read electronics are gated so that they only read signals which come shortly before, to shortly after, the dividing line between segments. During this period, if there is a flux transition from saturation in one sense to saturation in the other sense, a pulse will appear in the gate. The presence of a pulse is a one, and the absence of a pulse is a zero. More complicated coding than this is used for floppy disks. One type is called phase modulation. It uses flux transitions between gates so that a positive pulse is a one and a negative pulse is a zero. There are dozens of other coding schemes for digital recording.

Figure 7b shows a head which has just written two maximum drive flux changes on the medium, which is moving from left to right. Several things are of note: (1) the magnetization directions, shown by the arrows, are vertical in some places and horizontal in others; (2) there is a fairly wide transition zone between saturated segments; and (3) the transition zone is spread along the length of the medium. Compare this to the ideal situation shown in figure 7a, which has: (a) all the magnetization in the longitudinal direction; (b) a zero-width transition zone between segments; and (c) the transition zone lying only in the vertical direction. Each of these discrepancies from the ideal case loses some of the signal. There is an optimum value of drive current to get maximum output for any given distance between flux transitions. If the optimum situation is shown in figure 7b, increasing the drive current would make the transition zones more vertical, but the width of the zones would increase so much that the output would go down. If the drive current is decreased, the part of the coating away from the head doesn't get recorded, and this also reduces the output even though the transition zone width decreases.

Long Bars Are Better Than Short Bars

Now look at figure 7c. Either the medium-to-head speed has been slowed, or the frequency of flux changes increased; so that the flux changes come much closer together. We know that the maximum read output would come from what looked like long bar magnets laid end-to-end (as in figure 7a, but with the magnets even longer). The shorter the bar magnets, the less flux goes through the read head and the more goes through the bar magnet itself (this is known as demagnetization). In figure 7c, there is almost as much transition zone as magnet; the magnets are very short and not at all like bars; and the saturation magnetization does not go all the way through the coating. The read output will drop off so much that reducing the drive current, as shown in figure 7d, will actually increase the output again! In figure 7d, the magnets look more like bars, and the transition zones are not such a large percentage of the magnetized part. The recorded volumes do not go all the way through the coating, but the recorded part far from the head in figure 7c was out of phase with the recorded part near the head. It was really subtracting from the signal, so loss of that part actually increases the read output.

One thing is very apparent in 7c: Half the medium is not being used. For short distances between flux transitions, then, a thick medium is a waste. It's even worse than that. The transition zone is partially recorded, and the part farthest away from the head is making a negative contribution to the read signal output. We find that if we decrease the medium thickness so that we get rid of the continuous part of the transition zone (away from the head), we get some increase in output. Decreasing it too much will diminish the output again, so there is an optimum medium thickness for any digital recording situation. Because of the rapid loss of output as the transitions are crowded closer together, transitions are never placed as close together in digital work as in other types of recording. If this crowding is overdone just a tiny amount, some transitions give such a low output that bits are lost: an intolerable situation.

The Cassette Connection

Magnetic recording theory is divided into two parts: Magnetics and geometry.

There is a lot in common between digital recording on floppy disks and digital recording on cassettes, cartridges, or other tape media; but there are some differences, too. One difference is that we have been discussing a medium which is isotropic; that is to say, its magnetic characteristics are the same in all directions. This is not true of tapes, as their particles have been oriented during the manufacturing process, so that they record more easily in the direction of head-to-tape motion, and poorly in the other two directions. This means that the longitudinal component of the field is much more effective in recording than the vertical component is. The corresponding figures for oriented media (to 7b, 7c, and 7d) would always have the transition zone going to a point which would be fixed near the trailing edge of the head gap (see figure 7d), and the zone would slant to the left for low write currents and to the right for high write currents. Even with these differences, the conclusions we have already drawn would hold to a large extent. There is some indication that the vertical part of the write head field causes a type of partial erasure of the recording on the surface near the head, when an oriented medium is employed.

Another difference may be that biased recording is used, instead of saturation recording. The situation of oriented media used with biased recording is fully discussed in reference 1. Other types of recording, including frequency or phase modulated carriers, may be used. Teletype signals transmitted over telephone lines or via radio use a frequency shift type of modulation, where one audio frequency is a one and another is a zero. This type signal can be sent directly to an audio tape recorder with good results, except that it tends to be slow.

Keep It Clean, Fella!

Looking back on what we have learned about reading and writing digital signals on magnetic media, one thing stands out: The distances involved are very small. The period at the end of this sentence is about 0.02 inches (510 micrometers) in diameter. This is huge, compared to these important dimensions in recording systems:

Item	Dimension In:	Inches	Micrometers
Coating thicknesses:	Floppy disk	0.0001	2.5
	Cassette tape	0.0002	5.1
Head gap lengths:	Floppy disk	0.0001	2.5
	Cassette playback	0.00005	1.3

On a floppy disk, a magnetized volume of material on the surface of the coating away from the head is only about 15% as effective as an equally magnetized volume of the coating next to the head; and this is due only to the increased distance from the head. And as we have seen, it's harder for a write head to magnetize the far part of the coating, making things even worse. It follows that a piece of dust, just large enough to see, between the medium and the head can cause a very large loss of output signal. Something only half as large as that period would cause the complete loss of several bytes of information. In a factory making precision tapes or disks, no smoking is allowed in manufacturing areas; hair is kept covered; and special clothing is worn so as not to get anything on the recording surfaces. Even the smoke from cigarettes, pipes, or cigars will build up on heads and recording surfaces and cause eventual signal loss. Ashes cause instant dropouts (total loss of signal). Dust from any source is to be avoided like the plague.

There's also dust and dirt which comes from the medium itself, or its substrate. Floppy disks and tapes are both made out of a long polyester plastic sheet (called a web) which is coated with a special lacquer containing the magnetic material as its pigment. The original web may be from 12 inches (30.5 cm) to 48 inches (122 cm) wide for floppies, or 6 inches (15.24 cm) to 48 inches (122 cm) for tapes. After coating and drying, the web is usually calendered (pressed between heavy rolls). This smooths the surface to a mirrorlike finish, though it was fairly smooth to start with. Tapes are slit out of the web by shearing. Floppies are cut out with a die which also shears the edges. Tapes and floppies are then cleaned by various methods, since the shearing process leaves some debris behind. If the lacquer is formulated properly, and the shearing and cleaning are done with care, normal usage will not generate very much more dust and dirt to cause problems. If manufacturing is done carelessly or the lacquer is poorly formulated or unstable, usage will cause shed (dust), or worse, a gummy build up on the heads. Both these things tend to push the head away from the recorded surface, with a serious loss of output. Even the best of coatings will eventually cause some build up on the heads, and heads should be regularly inspected and cleaned.

Cleaning methods vary, and several ways are effective. If your machine operator's manual makes any recommendations, follow them. There are some special tapes and disks which are run in the machine for cleaning. Several companies have head cleaning materials and solutions on the market. My favorite concoction is half toluene and half isopropyl alcohol; but it has to be used with care, since the toluene dissolves some plastics and media coating lacquers. Straight isopropyl alcohol does a fair job, and is available in any drugstore. Apply the cleaner to the heads (and guides of a tape machine) with cotton tipped sticks. The ones made especially for cleaning heads are best, since their sticks are stiff, but you can also use the ones made for cleaning and oiling babies. Clean until the coating color is removed, or until the cotton swab comes away clean.

Professional installations sometimes have special machines to clean and recheck their media, but this is not usually within the budget of the individual. Cleaning of tapes is often accomplished by running them across a woven, lightly oiled, soft paper wipe which is moved slowly away from the point of contact. Tapes and disks can also be cleaned in an ultrasonic bath with an air squeegee. All methods require relatively complicated machinery, making cleaning impractical except for the largest installations. There are some companies which make a business of cleaning and re-certifying media. I recommend retiring from digital use any dirty media, and substituting new.

When buying tapes and cassettes, get the best quality you can buy. This is no place to save money, as it is always at the expense of lost bits. Tapes especially made for digital use are a good buy (floppies are always made for digital use). If you can't get these, use the top line of a well known brand of audio tape. Even this is second choice, since audio tapes, even good ones, may have some bumps on the surface which cause dropouts. The loss of five cycles of that high violin in "Scheherazade" will cause only a tiny gap which you won't hear, and you can lose a whole percent or so of "Rites of Spring" and

never know it; but the loss of just a bit or two of a digital sequence can cause nothing but garbage to issue from your computer.

Making Your Media Comfortable

About 15 years ago, some people at Southwest Research Institute, with grant money from the Rockefeller Foundation, made a monumental study for the Library of Congress on storage of sound recordings (reference 2). Part of their study was concerned with magnetic tape. Not very much can be added to their findings today. Boiled down, we can almost put their findings into one sentence: If people are comfortable in an environment, tapes can be safely stored there for long periods of time with little degradation. I say almost, because there are a couple of things to add to this. One is that, other than the earth's field, no other magnetic fields should be present if information is contained on the media. Permanent magnets, wiring carrying heavy currents, power transformers, and magnetic erasers or degaussers should be kept away from the media. For most of these things, three feet (one meter) is a good rule of thumb for distance. Don't get carried away and worry about such things as shielding from the earth's field, protecting from lightning or static electricity, guarding against radiation from radio transmitters or radar sets, or storing a hundred feet away from any electric wiring. Trouble from magnetic fields, though it can occur, is rare. The other added condition is that all media should be stored under low mechanical stress. Tapes and cassettes should be wound properly from a regular run, not a fast wind. Floppies should be stored flat, with no weight piled on top. If supported so that they don't buckle, they can be stored on edge. Never remove them from the envelope if you want to use them again. Avoid large temperature or humidity changes.

Summary

What I have tried to do is give you first an overview of digital magnetic recording so that maintenance and setup instructions for your machine will make sense to you. I haven't given specific directions for maintenance or setup, because each machine is a little different. Knowing how the information is contained on the medium is also of importance to understanding why cleanliness and good storage conditions are so important to safe storage. Lastly, I collected together several guidelines for cleanliness and storage which you probably won't find in the instruction manual for your machine. I hope that all this helps you to pack away your bits for easy retrieval. Once these principles become second nature to you, your large-scale storage problems should fade into the woodwork, and you can then apply your troubleshooting talents elsewhere.■

BIBLIOGRAPHY

1. "A Primer on Choosing Tape," William A. Manly, *Audio*, Volume 58, Number 9, September 1974, pages 34-46.

2. "Preservation and Storage of Sound Recordings," A.G. Pickett and M.M. Lemcoe, Library of Congress, Washington: 1959. Superintendent of Documents, Washington DC.

GLOSSARY OF MAGNETIC RECORDING TERMS

Anhysteretic magnetization: The magnetization remaining in a ferromagnetic material after applying a constant field H_{fixed}, superimposing on it a field varying continually from $+H_{cycled}$ to $-H_{cycled}$ (which is initially large enough in amplitude to cause practical saturation in each direction, then reducing the amplitude of H_{cycled} to zero as the cycling continues).

Biased recording: Magnetic recording done by adding the signal field to be recorded, a high frequency, large amplitude field called the *bias*. The purpose of the bias is to linearize the recording process.

Bulk storage: Supplemental storage of large volume capacity. Also called external storage, secondary storage or mass storage.

Coercive field: The applied magnetic field in a given direction, necessary to reduce the remanent magnetization of a ferromagnetic material to zero, after the application of a saturating field in the opposite direction.

Curie point magnetization: Magnetization of a ferromagnetic material, acquired by applying a field, heating the magnetic material until its ferromagnetism disappears (the "Curie point"), then cooling the material while still in the field.

Demagnetized: The condition of a ferromagnetic material when the directions of magnetization of all its domains have been randomized, so that there is no external field coming from the material.

Domain: A small volume of a ferromagnetic material in which the atoms are always magnetically aligned in the same direction. The magnetic direction of a domain may be changed, but it may not be demagnetized so long as the material is ferromagnetic.

Electron: A non-nuclear part of an atom; the smallest particle of (negative) electricity. An electron is regarded by physicists as a fuzzy ball of negative electricity which has a "spin" characteristic.

→

Erasure: The process by which a bulk magnetized ferromagnetic material is placed in a bulk demagnetized condition.

Ferromagnetic: A ferromagnetic material is spontaneously magnetized into an assemblage of tiny permanent magnets called domains. A ferromagnetic material can be demagnetized only in a bulk sense, and only when it is of a large enough physical size to contain many domains.

Frequency modulation: The changing of a carrier wave's frequency in accordance with the signal being transmitted.

Hysteresis loop: A closed curve obtained by plotting magnetization for ordinates ("y" direction) and applied magnetic field for abscissa ("x" direction) as the material passes through a complete cycle between definite limits of applied magnetic field.

Hysteretic magnetization (or hysteresis magnetization): Magnetization in a ferromagnetic material acquired by the cyclic application of a single applied magnetic field; magnetization at some point on a hysteresis loop.

Initial magnetization curve: The plot of the magnetization for ordinates and the applied field for abscissa of an initially bulk demagnetized ferromagnetic material, as the applied field has its strength increased from zero to some high value.

Isotropic: An isotropic material has some property the same in all directions. This word must be modified by some adverb describing the property, such as "magnetically isotropic."

Magnetic direction: A vector on a permanent magnet pointing from the south-seeking pole to the north-seeking pole; for a magnetic field, the vector starts at the north-seeking pole of a magnet and goes toward the south-seeking pole.

Magnetization: The number of elemental magnetic dipoles per unit volume of magnetic material. A single, isolated, spinning electron can be taken as the elemental magnetic dipole. All other units of magnetization are based on this.

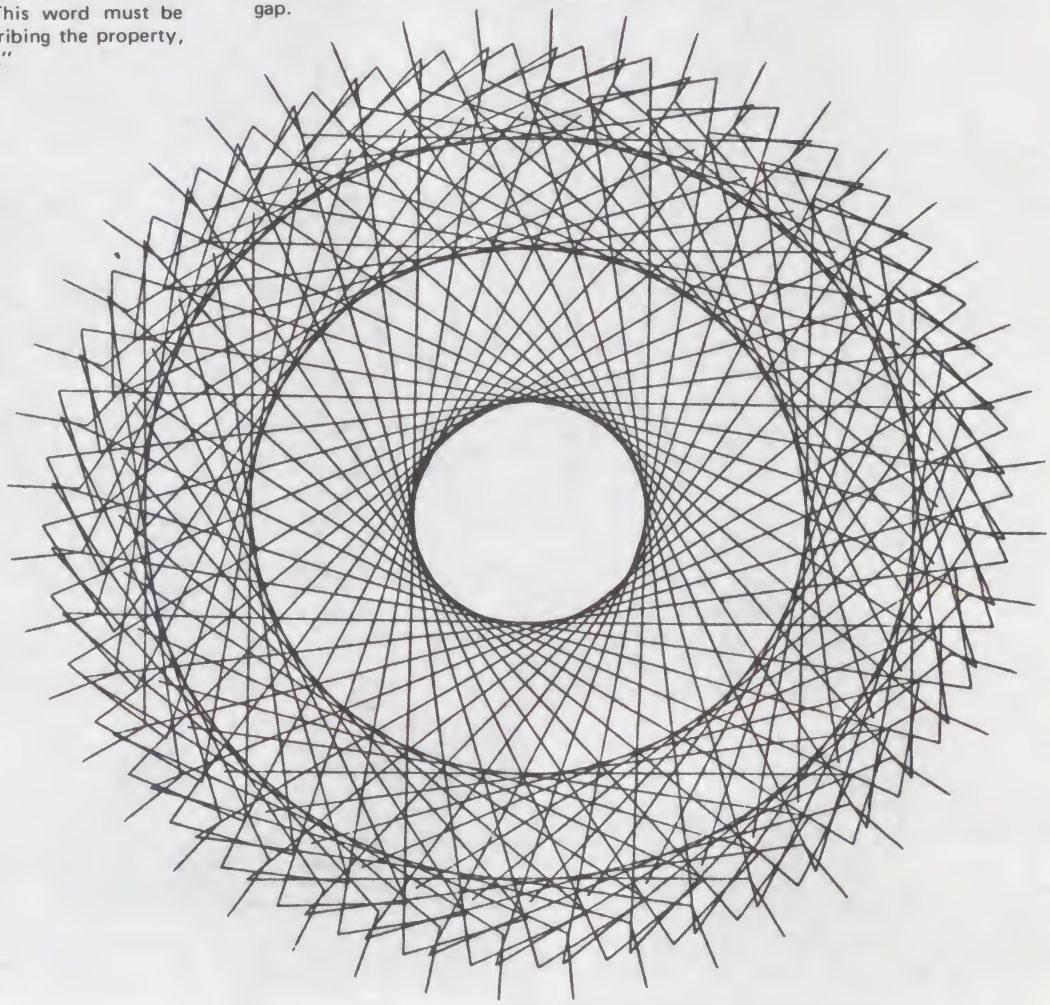
Remanent magnetization: The particular value of magnetization on a hysteresis loop when the applied field is zero; the bulk magnetization of a ferromagnetic material when there is no applied field.

Saturation magnetization: The magnetization of a ferromagnetic material when the applied field is so large that all the domains have their magnetic directions aligned with the applied field.

Spin: A representation of the rotation of an atomic or sub-atomic particle. Spin is a vector pointing along the direction of the axis of rotation.

Thick coating: A relative term referring to a magnetic coating or layer such that its thickness is greater than about half the length of the write head gap.

Thin coating: A relative term referring to a magnetic coating or layer such that its thickness is less than about half the length of the write head gap.



Computer Kits

First Person Report:

Assembling an Altair 8800

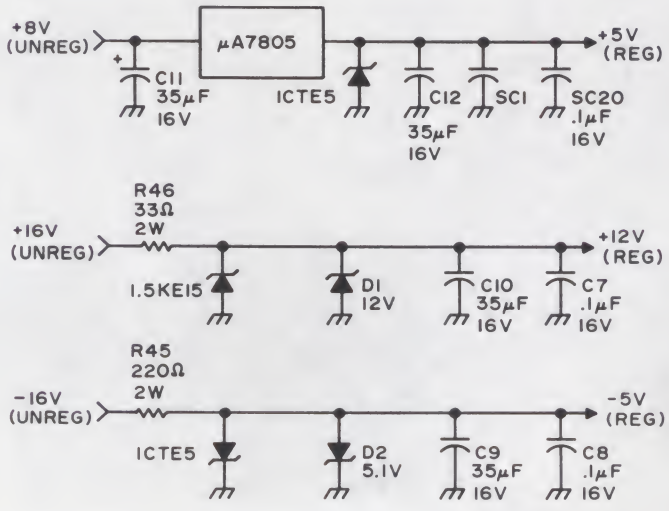
by
John Zarrella
90-9 Wakelee Rd.
Waterbury CT 06705

My adventure with microprocessors began rather late in the hobby game, at the end of 1974. It was about this time, or so it seemed to me, that micros became the topic of conversation in anything related to computers and automation. With the IMP-16, the 8080, 8008, 4004, etc., it became clear that this was what the computer market was waiting for. However, it was the article on the MITS Altair in the January 1975 issue of *Popular Electronics* which finally did it. Although inaccurate and vague, it certainly decided me — I was definitely going to own a micro. The next few months saw hurried mailings of information requests to any company which produced a product even remotely connected with a microprocessor. I immediately got out my checkbook, and mailed all my hard earned dollars to every newsletter that was published, in my frantic search for the "right" processor.

The results were both rewarding and disappointing. I found that there were some fantastic processors, but since my hardware background is not all that hot, I decided that I would have to opt for a kit with one of the most powerful micros I could find. I figured that this would enable me to get on line quickly, learn enough hardware to keep up with the state-of-the-art, and permit me to evaluate new micros as they came out, so I could build my "dream machine" when the right parts became available.

I decided to build the Altair 8800. Although the instruction set looked rather impressive, what convinced me was seeing a process control system which used the 8080; I was truly impressed with its capability.

Fig. 1. The schematic diagram of power supply circuitry, showing additional protection diodes.



I decided I would have to opt for a kit . . . this would enable me to get on line quickly.

The Order

After calling in my order to MITS, I waited nearly seven weeks for delivery. MITS did make it within the advertised 60-day delivery time. All was not roses for those seven weeks, however; it seems that either MITS or BankAmericard got their signals crossed and couldn't get a credit authorization (they both eventually declined to accept responsibility). You can imagine what it was like getting a call during dinner, explaining that my unit was

ready to ship, but unfortunately ... Luckily they agreed to ship it COD, and I quickly ran down to my bank to get a certified check. Every morning I left my wife with the admonition not to miss the delivery, and every day at lunch I called to determine whether or not my "computer" had arrived. (Did you ever try to ask your insurance agent whether you needed extra renter's insurance – "You keep a computer at home??!! What for?")

Assembly

Within a week of that call, I had the Altair in my hot little hands. "Are those little plastic parts all you get for $500.00?'', my wife exclaimed, peering over my shoulder. Undaunted, I shooed her out and locked myself in the back room all weekend soldering PC boards. It took three weekends to complete the assembly (was it my fault I came down with pneumonia in the middle?).

Ah yes, assembly. In general, I found that the MITS assembly instructions were well written. However, their additions were sometimes in the manuals in the wrong place (e.g., page 68A after 69). In at least one case (front panel control board) I had already tightened the panel in place (bolts on numerous switches), when I read that the nut on the little screw holding the voltage regulator to the board (accessible only with the panel out) had to be removed to add a grounding strap. Therefore it pays to check the manual pages carefully, and look two or three pages ahead to see if there are any little tricks sneaking up on you.

As for the parts, only one resistor was missing; however, out of all the screws and bolts supplied with the kit, I could never find the right one to fit. Maybe it was my own stupidity, but it seemed that the last bolt of any given size was always supposed to be used in at least 10 more places. I found that it pays to have a good assortment of screws and bolts (number 6, various lengths ¼" to ¾") to permit frustrationless assembly.

All soldering and component placement was easily accomplished – positions were clearly marked on the boards and in the manual. This is high praise since I hadn't built many kits before; and of these, none were this large. Of all the assembly, the worst (and easiest to mess up) part was correctly connecting the 60 bus wires between the display/control board and the chassis motherboard. I used an Ohmmeter to assure that each connection was correct and that there were no solder bridges to the other bus lines. There's got to be a better way. I hear Processor Technology, Inc., is currently marketing a 16-slot motherboard (on the Altair you have to jumper four of the four slot boards together, only one of which comes with the kit), and an improved connector for the display/control board. These will definitely be my first additions.

I made only one modification to the circuit during assembly. That modification was to add three protection zeners to the CPU board. Fig. 1 shows the electrical connections for this change. These were inserted to protect the 8080 chip (still pretty expensive in singles) from power supply failure. These zeners should ground out overvoltages at currents up to 100 Amps. ICTE5s were used for the +5 V and −5 V lines to the 8080 and a 1.5KE15 for the +12 V. The zeners on the CPU board are illustrated in Fig. 2.

I also added sockets for the 8101 RAMs, cleaned all boards with trichloroethylene solvent, and inspected the finished boards with a magnifying glass. I would highly recommend these procedures as they helped me find more than one solder splash and cold solder joint.

Of all the assembly, the worst (and easiest to mess up) part was correctly connecting the 60 bus wires between the display/control board and the chassis motherboard.

The Big Test

On the fourth weekend I got up the courage to mount the 8080 and 8101s. Then came turning on the power and checking voltages. Everything looked good, with very little ripple from the

Did you ever try to ask your insurance agent whether you need extra renter's insurance for a computer?

Fig. 2. Detail of the additional protective diodes mounted on the Altair CPU board.

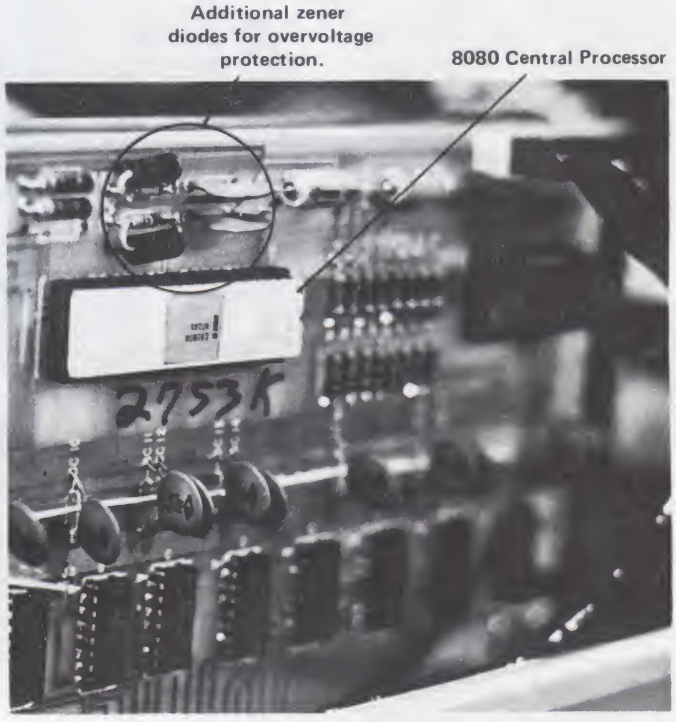
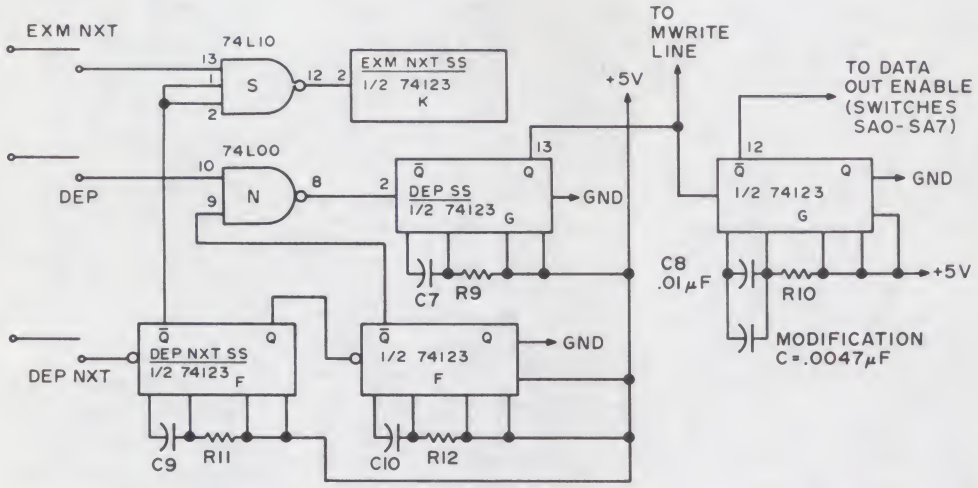


Fig. 3. Adding a parallel capacitance of .0047 uF to C8 of the Altair CPU board schematic lengthens the data out enable line time so that memory write does not extend longer than the data out time.



on-card voltage regulators. Finally the big test: Run a program. This is where the only problem finally showed up. I stopped and reset the CPU, set the switches for my spectacular program (JMP 0) and would you believe it, "deposit" wouldn't work. An hour later I had determined that all other panel switches worked correctly (including deposit next), and that the deposit switch itself was in good order. In order to initially get around the problem I had to examine location 177777 (all address bits 1), then use deposit next to get to location 0.

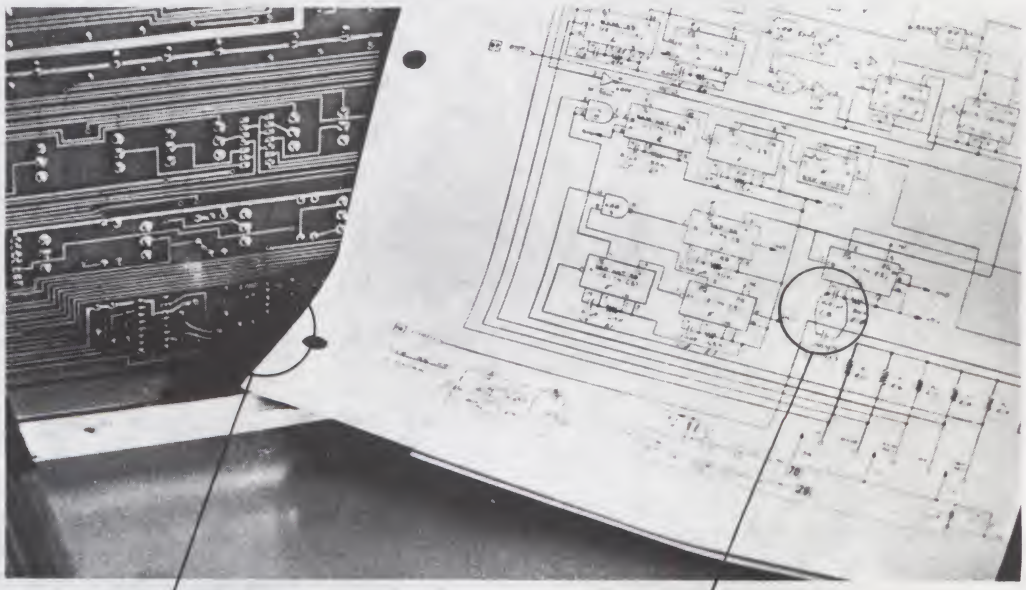
A study of the schematics showed that deposit and deposit next use the same circuitry, except that deposit next first does an examine next. You can verify this visually by loading all ones into the first 10 locations of memory. Then, if you use deposit next to change all the locations to zero, by carefully watching the data LEDs, you will notice that they all flash on as the switch is activated (examine next) and immediately go off again as the deposit is performed.

I concluded that the problem had to be in the timing, since the circuits were otherwise identical. Sure enough, when I looked at the signals on a scope, lo and behold, when a deposit was performed, the memory write line was enabled for approximately 20 ns more than the data out line. There are two oneshots in the deposit circuit; the first enables the memory write line, and the second enables the data out line. The memory write problem was cured by increasing the capacitance on the second deposit oneshot. An increase of .0047 uF (which increases the data out enable time by at least 30 ns) proved sufficient. This was obtained by adding the .0047 uF capacitor as shown in Fig. 3. When building the Altair, this means that C8 (front panel control board) should be approximately .0147 uF; if the board is already assembled into the case, a .0047 uF capacitor can easily be soldered onto the back of the board without removing any components from the case. (Be sure to unplug the computer before making the change, however.) Fig. 4 shows placement of the new capacitor and the change to the Altair schematic diagram.

I feel that the kit is reasonably well made and a good buy — at least at the current 8080 single lot prices, though the add-on options may cost somewhat more than elsewhere.

My plans for my unit currently involve addition of vectored interrupts (a 9318 or 74148 8-bit to 3-bit priority decoder is about all that's needed to translate the eight vectored interrupt lines on the bus into an RST instruction), a real-time clock, monitor clock and some type of I/O (teletype, CRT, etc.). ■

Fig. 4. The additional .0047 uF capacitor is mounted on the rear of the control panel board.



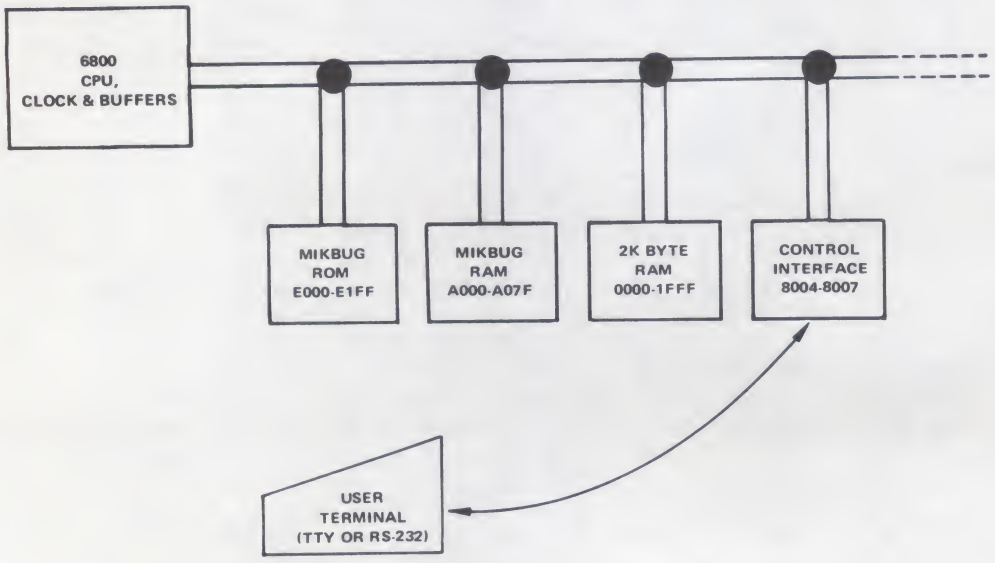
Build A 6800 System With This Kit

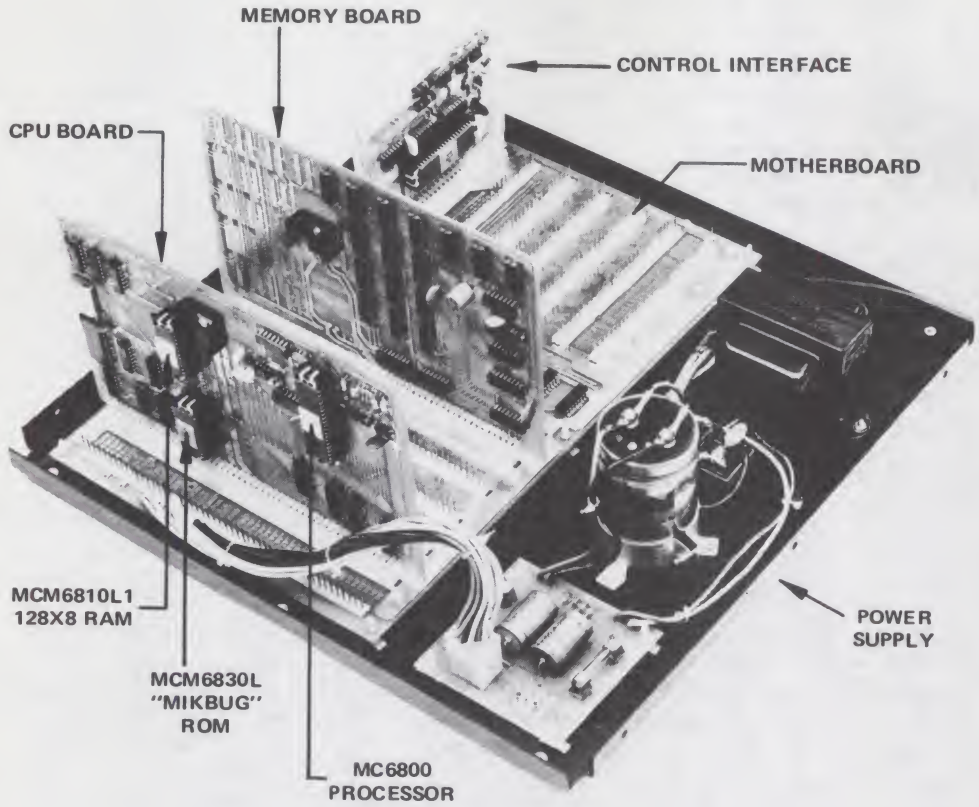
by
Gary Kay
Southwest Technical Products Corp.
219 W. Rhapsody
San Antonio TX 78216

If you are one of the many people getting ready to purchase one of the reasonably priced microprocessor system kits on the market today, you might ask yourself whether or not you will be able to start entering programs once you get it all put together. Of course you can always load programs and data through the front panel programmer's console, but most individuals aware of the front panel's slow speed and difficult readability prefer to use a Teletype or low cost video terminal such as the TV Typewriter II (February 1975, *Radio Electronics*) for data and program input/output. This is all well and good except that in order to attach a terminal, you'll have to purchase an interface for your computer if it is not supplied with the basic system. In fact you will generally need a separate interface for each I/O (input/output) device connected to your computer. This can run your system investment up considerably since such interfaces typically cost between $75 and $150 each, and there are more surprises yet to come.

So now you've got your computer, with interface, attached to your terminal; you're ready to sit down, power up and start typing in your program, right? Well, not quite. You see, in order to be able to use the terminal for either entering programs or getting data in and out of the computer you must have a program resident or loaded into memory telling the processor how and what to do. Without this software (program), you can pound on the keyboard all you want and the computer won't do anything. Computers are no smarter than their programming lets them be and without programming they're not very smart at all. How do you get this software into memory? Well, you could load it in from paper or cassette tape, that is if you have a paper tape reader or cassette tape interface (another sizable investment) or you could enter it directly from the programmer's console. The problem here is two fold. Software to give the terminal reasonable system control will probably be around 500 words in length. This is far too long to enter from the programmer's console especially when you consider it has to be re-entered every time the system is powered up or after a wayward program overwrites any of its allocated area of memory. The second problem is that few if any of the manufacturers supply a listing, paper tape or cassette tape of such a program to begin with. Their terminal control routines are contained within editor/assembler and higher

Fig. 1. Block diagram of the SWTPC 6800 system. The address allocations of the elements of the system are noted inside the blocks.





Details of the SWTPC 6800 System. This photo illustrates what you see when you remove the cover of a typical SWTPC computer system. This is an assembly of the parts which come in the MP-68 kit.

level language packages which not only must be loaded from some kind of tape reader, but require from 4,096 to 8,192 words of memory to operate. And you thought the interfaces were expensive, just check the prices on 8,192 words of memory. Many of the systems now on the market are supplied with an amount of memory with the basic unit which is considerably less than what might actually be needed for useful programming.

So what's the alternative? Well, the system presented in this article has been designed to eliminate the aforementioned problems and allow the user to have a powerful and fully functional system at a minimum cost (see Fig. 1). The entire system is built around the Motorola MC6800 microprocessor and its family of support devices. The computer itself is being made available in kit form including the chassis, cover, power supply and all circuit boards, parts and hardware necessary to build a Motorola 6800 based microprocessor including a 1,024 word ROM (read only memory) stored operating system with 128-word scratch pad memory, serial interface baud rate generator, serial interface, and 2,048 words of memory for $450. This article gives a description of the microprocessor and mother board. A future article will describe the power supply, memory and interface boards.

The Microprocessor/System Board (MP-A)

The Microprocessor/System Board (coded MP-A) is the primary logic board for the system. It is a 5 1/2" x 9" double sided plated-through hole circuit board containing the 6800 microprocessor chip, the 6830 ROM which stores the mini-operating system and the 6810, 128-word scratch pad random access memory (RAM) needed by the ROM. There is a crystal controlled processor clock driver and baud rate generator providing serial interface baud rates of 110, 150, 300, 600 and 1200 baud for all but the terminal control interface which is operable at 110 or 300 baud. Also provided is a power up/manual reset circuit which restarts the ROM stored mini-operating system whenever activated. Full I/O buffering is provided for the 16 address lines and eight bidirectional data lines with these and other connections made to the rest of the system through the mother board via a 50-pin connector. Power for the board is derived from a +5 volt regulator fed from the system's unregulated 7 volt, 10 Amp power supply. Average current consumption for the board is 0.8 Amps.

The mini-operating system stored in the 6830 ROM on this board has got to be one of the most outstanding features of this system. It is through this Motorola written software package called "MIKBUG" that the user can 1) enter program or data into memory from either the terminal's keyboard or tape (where applicable), 2) jump to and execute a program loaded in memory, 3) list programs or data stored in memory, on the terminal or tape (where applicable), 4) examine and/or change the contents of the internal CPU registers, 5) examine and/or change the contents of specified memory locations. These operations are performed using a 20 mA current loop Teletype or an RS-232C compatible serial ASCII terminal.

This ROM mini-operating system does not have to be loaded from tape and it cannot be overwritten. It is always there at your fingertips – just pressing the RESET button or simply powering the system up automatically restarts this firmware (ROM stored software). When activated, this system responds with a

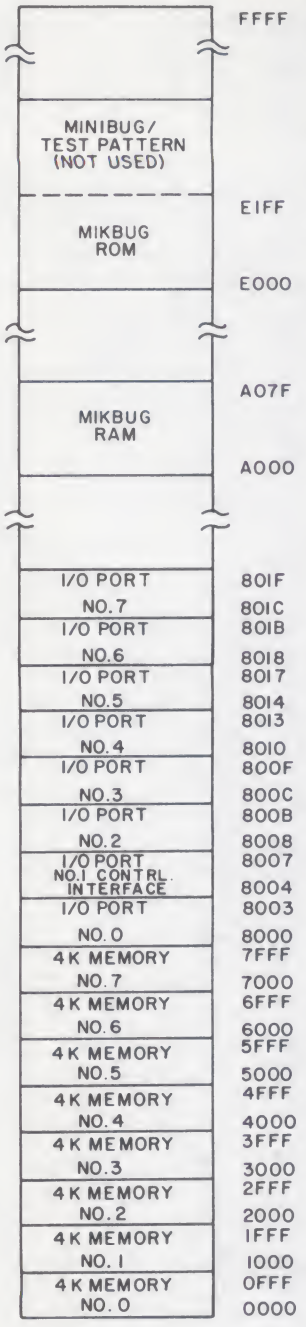


Fig. 2. SWTPC 6800 Microprocessor System memory map. The 64K address space of a 6800 CPU is divided up into the segments shown here. The first 32K locations are available for user read-write memory. The second 32K is devoted to I/O port assignments and the requirements of the MIKBUG program supplied by Motorola.

carriage return, line feed and then prints a * on the terminal at which time you may enter various single character control commands such as M for memory examine/change, L for load from tape, P for punch or list, R for examine registers or G for go to and execute a loaded program. A program debug routine can also be implemented by using the software interrupt (SWI) instruction as a "breakpoint" which forces a jump from your program to the operating system to allow you to examine the contents of memory and/or the CPU registers. All data entered or displayed through the terminal is in convenient hexadecimal (base 16) notation. This means you can type in a command to load address location $A000_{16}$ with $9E_{16}$ instead of setting 24 console switches to an address of 1010 0000 0000 0000 with data of 1001 1110 as must be done with the conventional programmer's console. Since the operating system is stored in ROM, it consumes no user RAM memory, in fact, it actually gives the user a little extra. There is a 128-word scratch pad memory utilized by the operating system for storing various addresses and data, but there are more than 54 locations within this 6810 RAM memory which are totally unassigned plus a 46-word deep push-down stack. All of this memory is in addition to the 2,048 words (expandable to 4,096 words) contained on the standard memory board.

Since the terminal and mini-operating system provide the user with complete system control, there is no need for the conventional programmer's console. Take note also that once system control is turned over to your program, the control terminal is totally available for your program input/output. In fact, since the character input/output subroutines are already stored within the operating system ROM, they can be used by your programs simply by loading or storing the characters to be handled in the proper register and executing a jump to subroutine (JSR).

The Motorola MC6800 microprocessor chip is the element around which this entire system is built. It is an 8-bit parallel processor with eight bidirectional data lines and 16 address lines giving it an addressing capability of up to 65,536 words. There is no distinction between memory and I/O addressing on this system, therefore, all input/output data transfers are handled just as are the memory transfers. This means the I/O interfaces must have their own allocated memory addresses where neither ROM or RAM memory may be located. This may at first seem to be a disadvantage until you realize that all memory handling instructions are usable for the interface data handling as well, thus eliminating the need for special data I/O instructions. The memory assignments for this system have to be made as shown in Fig. 2. User RAM may be located anywhere in the lower 32K (0000_{16} to 8000_{16}) addresses with the upper 32K addresses reserved for the operating system ROM, RAM and interface boards.

There are six registers internal to the MC6800 microprocessor element which consist of the program counter, stack pointer, index register, accumulator A, accumulator B and condition code register. The stack pointer is a 16-bit register used to store the address of the push-down stack which is located in RAM memory external to the MC6800 microprocessor element. The push-down stack itself is used to store the program counter and/or processor data during branch to subroutine (BSR), jump to subroutine (JSR), push (PHS) or interrupt routines. The index register is a 16-bit register generally used as an address pointer for many processor instructions.

There are 72 basic instructions for the 6800 microprocessor system (Fig. 3) with most of the 72 utilizing several of the seven possible addressing modes: Accumulator, implied, relative, direct, immediate, extended and indexed.

- *Accumulator* — In accumulator addressing, either accumulator A or accumulator B must be specified.
- *Implied* — In implied addressing the instruction code itself specifies the operand (stack pointer, index register, etc.).
- *Relative* — Relative addressing is used for the branch instructions and indicates the value contained in the word of memory immediately following the instruction code added to the program counter +2 with the result then loaded back into the program counter. Positive data (bit 7 = 0) generates forward jumps up to 129 words from the branch instruction while negative data (bit 7 = 1) generates backward jumps up to 125 words from the branch instruction.
- *Direct* — In direct addressing, the value contained in the word of memory immediately following instruction code is an actual memory address within the first 256 words of memory (0000_{16} to 00FF) which contains the operand of the instruction. This mode typically saves one CPU cycle of execution when compared to *extended* addressing.
- *Immediate* — In immediate addressing, the

value contained in the word, or in some cases two words of memory, immediately following the instruction code is the operand of the instruction.

• *Extended* — In extended addressing, the two words of memory immediately following the instruction code contain the address of the memory location which contains the operand of the instruction.

• *Indexed* — In indexed addressing, the value contained in the word of memory, immediately following the instruction code, is temporarily added to the contents of the index register generating a new address where the operand of the instruction is located. The jump is positive only, going from 0 to 255 words and the actual contents of the index register are not changed.

Also provided on the main processor board is an MC14411 baud rate generator which uses an external 1.8432 MHz crystal and internal oscillator and divide chain to generate serial interface clocks for baud rates of 110, 150, 300, 600 and 1200 baud. Also derived from this circuit is the 921.6 kHz clock used by the MC6800 microprocessor element. It is first, however, fed into a gating circuit generating two non-overlapping, 50% duty cycle, complementary clock signals ϕ_1 and ϕ_2.

Mother Board (MP-B)

The Mother Board (coded MP-B) is a 9" x 14" double sided, plated-through hole circuit board onto which all of the various processor boards are plugged. Provisions have been made for one Microprocessor/System Board, up to four 4,096 word random access memory boards plus two unused slots. This allows the system to be expanded to 16,384 words of memory. For those demanding even more memory, the 50-line system information bus may be paralleled onto another mother board with separate power supply expanding the system to a maximum of 32,768 words of random access memory.

The Mother Board also provides the line buffering and address decoding for up to eight interface boards. Although one of the eight must be the serial terminal, control interface, the other seven may be any combination of parallel or serial interfaces the user may choose to have. For those demanding even more interfacing capability, the 50-line system information bus may be paralleled onto another mother board with separate power supply expanding the interfacing capability to one terminal, control interface plus any combination of up to 15 serial or parallel interfaces.

The following is a brief description of each of the 50 lines on the system information bus:

The A0 — A15 lines carry address bits 0 through 15 respectively, forming a 16-bit address which is used to define either a memory location or interface address.

The BUS AVAILABLE line goes high acknowledging a processor halt, meaning the processor has stopped and that the system information bus is available for external control.

The $\overline{D0}$ — $\overline{D7}$ lines carry inverted data bits 0 through 7 respectively, forming 8-bit data words which are exchanged between the various boards within the system.

The GND line is the system's common power supply ground point.

Fig. 3. The 6800 microprocessor's instruction set. This is a list of the mnemonics available. A more complete explanation of the basic operations of the processor is found in Motorola's programming manual for the 6800 which is part of the SWTPC documentation package.

ABA	Add Accumulators
ADC	Add with Carry
ADD	Add
AND	Logical And
ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
BCC	Branch if Carry Clear
BCS	Branch if Carry Set
BEQ	Branch if Equal to Zero
BGE	Branch if Greater or Equal Zero
BGT	Branch if Greater than Zero
BHI	Branch if Higher
BIT	Bit Test
BLE	Branch if Less or Equal
BLS	Branch if Lower or Same
BLT	Branch if less than Zero
BMI	Branch if Minus
BNE	Branch if Not Equal to Zero
BPL	Branch if Plus
BRA	Branch Always
BSR	Branch to Subroutine
BVC	Branch if Overflow Clear
BVS	Branch if Overflow Set
CBA	Compare Accumulators
CLC	Clear Carry
CLI	Clear Interrupt Mask
CLR	Clear
CLV	Clear Overflow
CMP	Compare
COM	Complement
CPX	Compare Index Register
DAA	Decimal Adjust
DEC	Decrement
DES	Decrement Stack Pointer
DEX	Decrement Index Register
EOR	Exclusive OR
INC	Increment
INS	Increment Stack Pointer
INX	Increment Index Register
JMP	Jump
JSR	Jump to Subroutine
LDA	Load Accumulator
LDS	Load Stack Pointer
LDX	Load Index Register
LSR	Logical Shift Right
NEG	Negate
NOP	No Operation
ORA	Inclusive OR Accumulator
PSH	Push Data
PUL	Pull Data
ROL	Rotate Left
ROR	Rotate Right
RTI	Return from Interrupt
RTS	Return from Subroutine
SBA	Subtract Accumulators
SBC	Subtract with Carry
SEC	Set Carry
SEI	Set Interrupt Mask
SEV	Set Overflow
STA	Store Accumulator
STS	Store Stack Register
STX	Store Index Register
SUB	Subtract
SWI	Software Interrupt
TAB	Transfer Accumulators
TAP	Transfer Accumulators to Condition Code Reg.
TBA	Transfer Accumulators
TPA	Transfer Condition Code Reg. to Accumulator
TST	Test
TSX	Transfer Stack Pointer to Index Register
TXS	Transfer Index Register to Stack Pointer
WAI	Wait for Interrupt

The normally high $\overline{HALT}$ line when brought low halts the processor and frees the system information bus for external control.

The INDEX line is an unused one and is provided so the pin on each of the male connectors may be cut with the corresponding female connector pins plugged, preventing the circuit boards from being plugged on incorrrectly.

The $\overline{IRQ}$ is the maskable, single level interrupt request line feeding the processor board. If not inhibited by software it will when momentarily given a TTL zero level signal, force the processor into a push-down stack store routine followed by a program jump to a user selected address stored in the operating system RAM.

The M. RESET line, when momentarily grounded manually, indirectly resets the registers internal to the processor and interfaces, and loads the ROM stored mini-operating system. This line is normally grounded by depressing the RESET button on the system's front panel.

The $\overline{NMI}$ is the non-maskable, single level interrupt line feeding the processor board. When momentarily given a TTL zero level it forces the processor into a push-down stack store routine, followed by a program jump to a user selected address stored in the operating system RAM. The $\overline{NMI}$ is not maskable thus cannot be inhibited by the programmer through software.

$\overline{\phi_2}$ is one of the two complementary system clock outputs and is used to signal that valid data is on the data lines $\overline{D0}$ – $\overline{D7}$ when low.

ϕ_1 is the non-overlapping clock complement of $\overline{\phi_2}$.

The $\overline{RESET}$ line when low resets the registers internal to the processor and interfaces, and loads the ROM stored mini-operating system. This line is activated by one shot on the Microprocessor/System board when the system is first powered up or when M. RESET line is momentarily grounded.

The R/$\overline{W}$ line establishes the direction of data flow on the eight data lines, $\overline{D0}$ – $\overline{D7}$. It is high for a read from memory or interface and is low for a write to memory or interface.

$\overline{VMA}$ is the valid memory address line which goes low to confirm that valid memory address data is being presented on the 16 address lines, A0 – A15.

The UD1 and UD2 are user defined lines and have not been assigned a function.

The –12 and +12 points are lines to which an isolated ground –12 @ 200 mA and +12 @ 200 mA power supply should be connected. The voltages are necessary for generating the currents required by 20 mA current loop Teletype equipment on the serial interfaces.

The 7 – 8 VDC UNREG point is the line to which a +7 to 8 volt dc @ 10 Amp unregulated power supply should be attached. This voltage is then regulated down to +5 V dc by independent regulators on the various boards within the system.

The five 110b, 150b, 300b, 600b, 1200b lines carry 1758.8, 2400, 4800, 9600 and 19200 Hz clocks required by the serial interfaces for 110, 150, 300, 600 and 1200 baud communication.

Also attached to the 50-line system information bus are the interface decode and driver circuits. A considerable cost savings is made here by providing the address decoding and information bus buffering for all of the interfaces right on the mother board instead of providing it on each of the interface boards individually. Since each of the parallel interfaces require four address locations and the serial two, four addresses are provided for each of the interface positions. They are assigned as shown in the memory map, Fig. 2. Interface position 1 (8004 – 8007) is reserved for the terminal control interface. The signals carried on the interface information bus are almost identical to those on the system bus. UD3 and UD4 are here again User Defined data lines and RS0 and RS1 are Register select lines which are identical to address lines A0 and A1 respectively. Power for the address decode and buffer circuits on the mother board is provided by a separate on board regulator with a current consumption of typically 0.4 Amp. ■

(More SWTPC 6800 data is coming in BYTE.)

Once you've assembled and checked out the operation of your MP-68 kit, the result will be a product which looks like this. Note the complete absence of most of the usual control panel functions you might expect. This is achieved by using a serial communications device such as a Teletype or an RS-232C compatible terminal as the "front panel."

More on the SWTPC 6800 System

Gary Kay
Southwest Technical Products Corp
219 W Rhapsody
San Antonio TX 78216

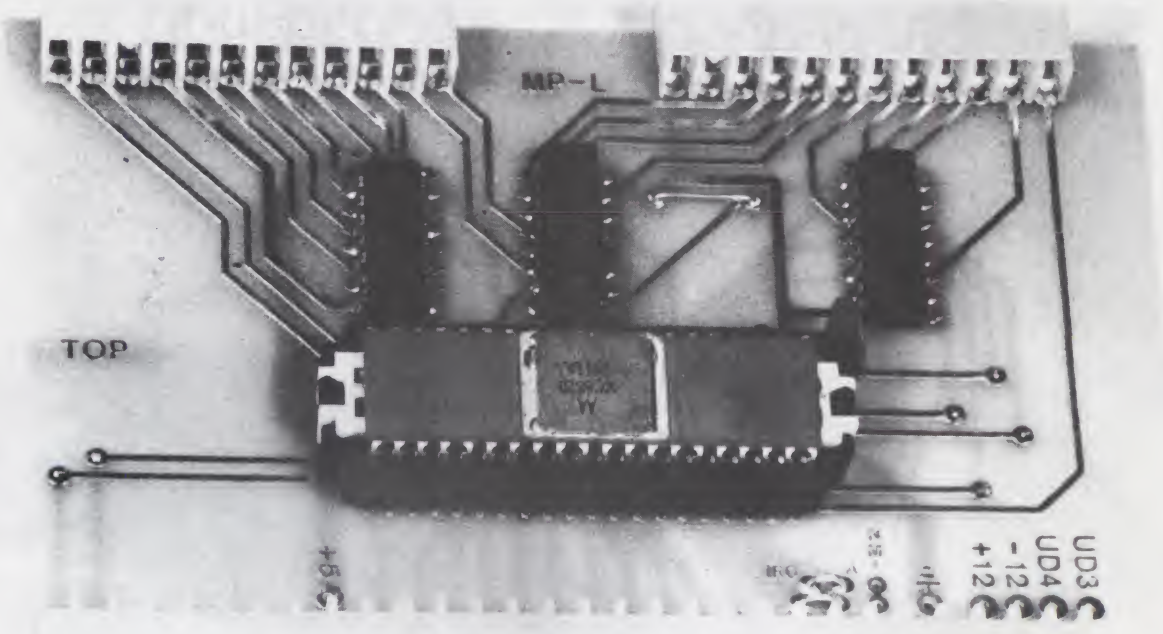
In the December 1975 issue of BYTE we talked about the microprocessor/system board (MP-A) and the mother board (MP-B) for the Southwest Technical Products 6800 microprocessor system. This article describes the serial control interface (MP-C), the 2,048 byte random access memory board (MP-M), the power supply (MP-P) and cabinet (MP-F). We will also talk about the serial interface boards (MP-S) and parallel interface boards (MP-L).

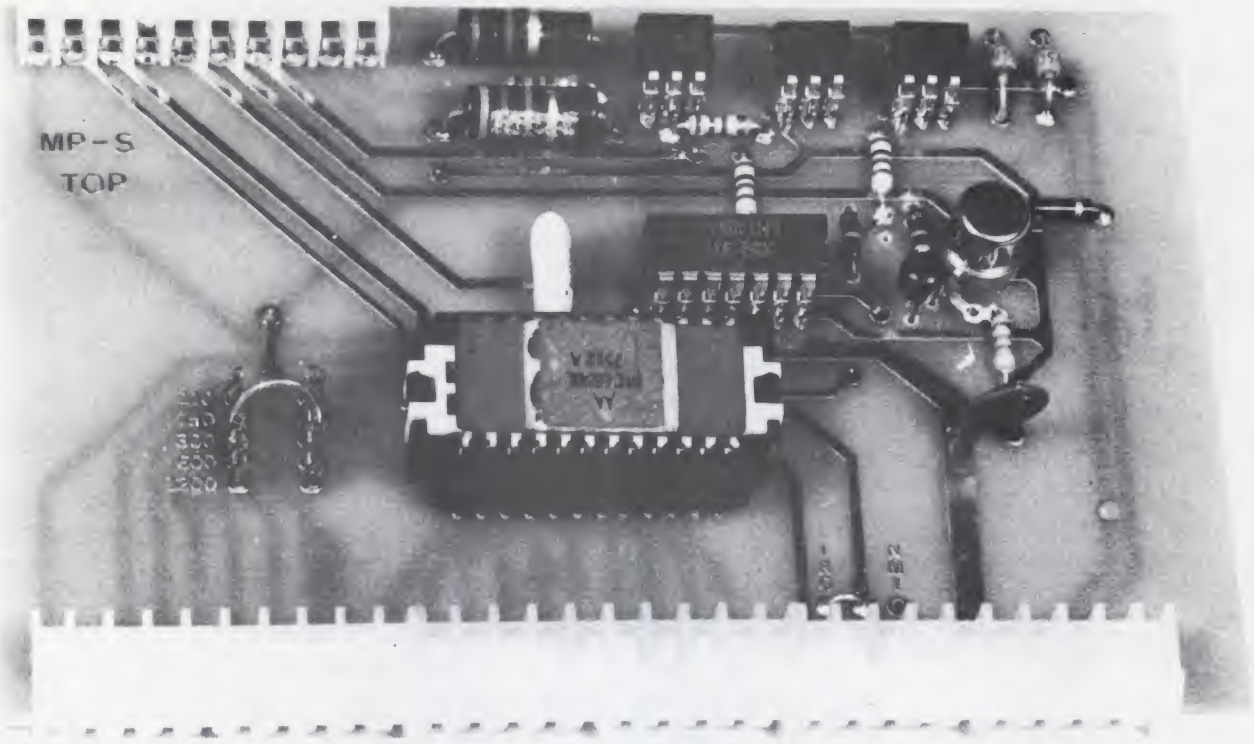
Serial Control Interface

The serial control interface (coded MP-C) is a 5.25 inch by 3.5 inch doubled sided, plated through hole board containing a 6820 peripheral interface adapter integrated circuit and circuitry which forms the serial control interface. Data rates of 110 or 300 baud are selected by a jumper wire. The interface includes software control of an input to output echo feature which is necessary in some tape reader operations. Its data input or output must be in ASCII (without parity) and either 20 mA Teletype or RS-232 compatible. A low cost terminal such as the TV Typewriter II (February 1975 issue of Radio Electronics Magazine) is ideal.

IO connections are made to the interface via a 10 pin connector along the top edge of the board. Power for the board is provided by a 5 VDC regulator at a current consumption of about 0.2 A. 12 VDC and −12 VDC sources are also used.

The Parallel Interface Board is used to latch and control the input and output of 8 bit bytes. The Motorola 6820 Peripheral Interface Adaptor (PIA) is the main component of this board, with several smaller chips acting as buffers. This board permits parallel connections to such devices as printers, laboratory breadboards, and special purpose keyboards.





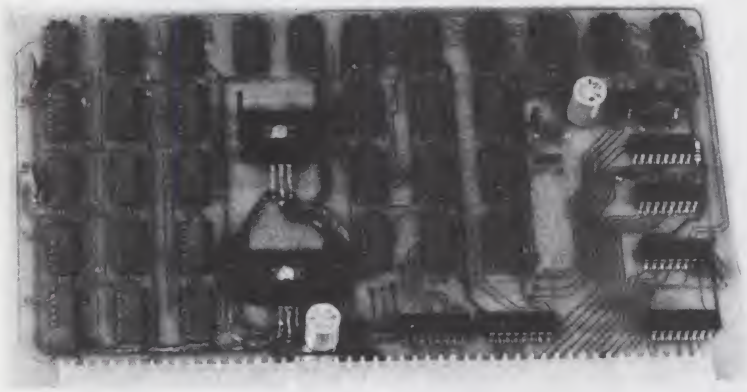
The Serial Interface Board is used to convert parallel data from the processor into serial data to a terminal (and vice versa). The major portion of this task is accomplished using the Motorola 6850 Asynchronous Communications Interface Adapter (ACIA) circuit, which is the large package in the center of the board. The remaining components on this board are used to provide both an RS-232 interface and a Teletype 20 mA current loop interface.

The board itself must only be plugged onto the first interface port position of the mother board. It is constantly polled for incoming commands by the Motorola MIKBUG software stored in the operating system ROM whenever the system is powered up, or is reset and is under operating system control. When system control is turned over to your program, the control terminal is also available for program IO. To output a character to the terminal's display, store the character in accumulator A and jump to subroutine OUTEEE, which is a character output routine written into the operating system ROM. To input a character from the control terminal's keyboard, jump to subroutine INEEE, which is a character input routine written into the operating system ROM. In this subroutine the system hangs in a loop until a character is typed at which time there is a return from subroutine with the entered character deposited in accumulator A. The use of these ROM stored subroutines greatly simplifies the job of the programmer for control terminal data input/output.

In addition to the Serial Control Interface, any combination of up to seven parallel or serial interfaces may be plugged onto the interface connectors. Since the 6800 family of chips includes both parallel (6820) and serial (6850) interface elements, interfacing is extremely flexible.

Parallel Interface

The Parallel Interface (coded MP-L) is a 5.25 inches X 3.5 inches (12.86 cm X 8.57 cm) double sided, plated through hole circuit board containing a 6820 peripheral interface adapter integrated circuit and its associated circuitry which is used to connect a parallel data device such as a printer or parallel data terminal to the computer system. The board is provided with two separate connectors along the top edge of the board. One has eight fully buffered TTL compatible high current data outputs along with one buffered "data ready" output line and one "data accepted" input line for complete handshake control. The other connector has eight TTL compatible input lines along with one "data ready" input line and one "data accepted" output line, here again for complete handshake control. The "data



The memory board, shown here with a full 4096 word complement of 2102 memory chips, is one of the more important elements of the system. The black chips at the right edge of the photo are interface devices and address decoding. The two voltage regulators on the board are in the center. The remaining integrated circuits are 32 chips of 1 K by 1 bit memory.

ready" and "data accepted" lines are under complete program control even to the extent of setting the transition polarity upon which the lines will be triggered. Interrupts are under complete software control as well.

For the user who has specialized parallel IO requirements, the TTL data buffers may be omitted from the board, and each of the sixteen data lines may be individually software programmed by the user as either all inputs, all outputs or any combination of the two. The programmer has complete software control of the four handshake lines, two of which are software programmable for input or output. Power for the board is supplied by a 5 V regulator at a current consumption of 0.3 A.

Serial Interface

The Serial Interface (coded MP-S) is a 5.25 inches X 3.5 inches (13.3 cm X 8.9 cm) double sided, plated through hole circuit board containing a 6850 asynchronous communications adapter integrated circuit and its associated circuitry which is used to interface a serial device such as a terminal to the computer system. Like the Serial Control Interface, its communication must be in ASCII form and either 20 mA TTY or RS-232 compatible. Baudot coded teletypes will not work. The data IO baud rate for each of the interfaces is jumper programmable and may be set for 110, 150, 300, 600 or 1200 baud operation. One central clock on the microprocessor/system board provides all of the various baud rate clocks simultaneously, so that each of the serial interfaces can have an independent data rate. This eliminates a good deal of duplicate circuitry and keeps the serial interface cost low.

As with the Parallel Interface, there are many functions that are under software control. Selection of one of 8 different combinations of bit count, parity, and number of stop bits is user programmable as is control of transmitter and/or receiver interrupts. Checking the interface for transmitter buffer empty, receiver buffer full, framing error, parity error, and receiver overrun are here again all done through software just by reading the data contained within the interface's internal status register. External connections to the board are made via a ten pin connector along the top edge of the board. Power for the board is supplied by a +5 V regulator at a current consumption of approximately 0.2 A. +12 VDC and −12 VDC sources are used for generating the Teletype currents and the RS-232 voltage output.

Memory Board

The Memory Board (coded MP-M) is a 5.5 inch by 9 inch (14 cm by 22.9 cm) double sided plated through hole circuit board with data bus buffering, and address decoding for up to 4,096 bytes of fast 2102 static random access memories. The basic memory board kit comes with only 2,048 words, however. To fill the board to a full 4,096 words of RAM, you must add the memory expansion kit (MP-MX) which contains another 2,048 words of memory ICs and a separate voltage regulator. Up to four of these 4,096 word boards may be plugged onto each mother board.

The 2102 static memories were chosen because of their availability, low cost and established reliability. Although the 4 K dynamic memories are becoming popular, they require refresh circuitry and slow the processor during refresh cycles. Address assignments are made on each memory board by connecting the address jumper to one of the eight possible positions, progressing on each memory board from 0 to 7. This programs the boards from 0 to 32 K words in 4 K word increments. Since each mother board will only support up to four 4,096 word memory boards, it is necessary to use another mother board with separate power supply to expand the memory beyond 16,384 words. Power for the lower 2,048 words of memory as well as the decode and buffer circuits is provided by a 5 V regulator with a current consumption of approximately 0.8 A. Power for the upper 2,048 words of memory when present is

provided by a separate 5 V regulator at a current consumption of approximately 0.6 A.

Power Supply

The MP-P power supply consists of a power transformer, high current bridge rectifier, filter capacitor, and power supply board. The low voltage transformer secondary winding, bridge rectifier and filter capacitor provide the 7 to 8 V DC at 10 A required by the complement of boards in the computer system. Since the regulation down to 5 V is provided on each of the system boards, the actual value of this voltage is not critical. It must however be maintained at no less than 7 V for proper regulator operation while not so high as to cause the regulators to generate abnormal temperatures.

The higher voltage transformer secondary winding along with the rectifiers and filter capacitors on the power supply board provide the +12 and −12 V DC at 0.5 A required by the control and serial interfaces. All connections from the power supply to the mother board are made through an easily detached connector on the power supply board. This makes mother board installation and removal a snap. The power transformer's primary may be wired for either 120 or 240 VAC operation with a current consumption of 120 VAC at 1 A or 240 VAC at 0.5 A.

Chassis and Cover

All of the boards for the 6800 computer system including the power supply are housed in a 15.125 inches wide X 7.0 inches high X 15.25 inches deep (37.05 cm wide X 17.15 cm high X 37.36 cm deep) anodized aluminum chassis with a perforated cover. The use of the perforated cover eliminates the need for a cooling fan in almost all environments. The front panel supports both the POWER on-off and RESET switches. The RESET switch initializes all of the registers in the system and loads the terminal controlled Motorola MIKBUG operating system whenever depressed. The rear panel contains an array of holes through which the interface cables and line cord may pass. Both panels along with the cover may be easily removed providing 360° access to the system for prototyping or service.

The 6800 system presented within this and the previous article, has been shown to have outstanding ease of use and is an economical package. But as many of us already know, hardware is but a small part of a "computer system." Programming, or software as it is generally referred to, is just as important as the hardware. Of course this system does have a very useful ROM stored operating system, but what else is available, and how does one load such software in memory without having to type it in through the control terminal one byte at a time? Well, first of all several diagnostic listings are provided by the manufacturer of the kit to help check out the various boards within the system. These diagnostics are typically less than 90 bytes in length and can be entered manually from the control terminal in less than five minutes. Included within these diagnostics are two programs that provide a thorough checkout of the random access memory boards, a common failure point for many systems.

Regarding some method of storing and loading in programs, a low cost audio cassette tape digital storage system is presently in the works that will be totally compatible with this computer system. You can expect to see it in a forthcoming BYTE Magazine article. Also to be available shortly is an editor/assembler software package which will be sold for the cost of the documentation and tape only to those people simultaneously purchasing 4 K of the 8 K words of memory necessary to support the package.

Another note of importance is that the ROM stored mini-operating system on the Microprocessor/System Board is exactly the same (MC6830L7) as that used on Motorola's Evaluation Module and Integrated Circuit Evaluation Kit. This means that most all programs written for the Motorola's Evaluation Module will function on the 6800 computer system presented here. Motorola also supports their more sophisticated prototyping system called the EXORcisor® (Registered trademark of Motorola Inc.). This system has a larger, more sophisticated firmware package, but it uses the same 6800 microprocessor element, therefore much of its software is compatible with the 6800 system presented here. Because of this compatibility, arrangements have been made with Motorola Inc. to allow Southwest Technical Products 6800 Computer System customers to have access to Motorola's 6800 program library. Customers will be permitted to join by either submitting an acceptable program or by paying a membership fee. Either makes them a member of the Motorola 6800 User's Group for two years with access to programs within the library plus upcoming program additions.

For those applications requiring the utmost in speed and storage capability, arrangements are in the works with ICOM Corporation to supply a floppy disk and floppy disk operating system (FDOS) that is compatible with the 6800 system described in this article. ■

The New ALTAIR 680

James B Vice
MITS Inc

The new ALTAIR 680 designed by MITS is a system based on the 6800 microprocessing unit (MPU). The MPU is available from Motorola or American Micro-Systems and adapts nicely to a minimum design configuration.

The ALTAIR 680 case measures about 11" by 11" by 4-3/4" (28 cm by 28 cm by 12 cm) making it less than one third the size of the ALTAIR 8800. The basic system is available in three configurations, depending on the intended application. These include a user programmable processor with complete front panel controls, and two smaller versions oriented towards dedicated ROM programmed applications.

The compact size of the 680 obviously precludes any significant amount of internal expansion, although additional memory and IO control are already on the drawing board. Its small physical size can be deceiving. The overall concept was to keep the machine as simple, small and inexpensive as possible; but it forms the complete central processor of a system in itself. All that is needed to make a MITS 680 system is the addition of some IO devices and software.

The Three Models

The construction of this machine is a relatively easy matter for even the most inexperienced kit builder. Almost all of the circuitry is contained on a single large printed circuit board, including memory and a built-in IO port. This single board is a full central processor with the exception of a power transformer and some control switches. This is where the main distinction between the three configurations is encountered:

- Most hobbyists will be concerned with the full front panel model. This contains all of the necessary controls for addressing and entering data besides those for controlling the processor itself.
- A turn key front panel model is also available which eliminates all controls except restarting the processor's ROM software. This could be used in applications where it is desirable to eliminate the possibility of the operator or any other person affecting the machine's memory or computing cycle. An example for such an application might be its use in controlling an intrusion detection system, or for a manufacturing machine control system.
- The third configuration is similar to the turn key version. The 680 will also be available as just the large PC board mentioned above. This board contains

everything but a power supply and controls. Its application is similar to the turn key model, except that the computer would be "buried" inside another machine.

The board only model is an excellent starter for the experimenter who wishes to purchase an absolute minimum and do a bit of his own designing. Such experimental use is aided by the considerable amount of information available on the 6800 microprocessing unit from Motorola Semiconductor Products, Inc. The 6800 MPU is also TTL compatible and requires only one 5 volt power supply.

Front Panel

In the front panel model of the 680 there is an additional printed circuit board. This board contains all of the logic circuitry necessary to reset, halt or start the processor. Also located on this board are switches and associated LED indicator lights for each of the sixteen address lines and eight data lines. The front panel printed circuit board mounts directly to the main printed circuit board via a 100 contact edge connector. This eliminates the need for a cumbersome wiring harness. The only other control is the power switch, located on the back panel of the unit for safety purposes.

On the dedicated program models, no front panel is needed because PROM or ROM software is used to store the starting address; a minimum fixed set of programs must be supplied by the user or manufacturer in this form of the system.

Functional Description

The basic ALTAIR 680 computer can be subdivided into five functional sections. These are the MPU and clock, the memory, an IO port, control and indication, and the power supply.

The first three of these sections, along with the power supply regulation components, are located on the main printed circuit board.

MPU and Clock

At the heart of the 680 system is the 6800 microprocessing integrated circuit. This is a versatile and very powerful little processor, yet it is directly responsible for the overall simplicity of the 680 design.

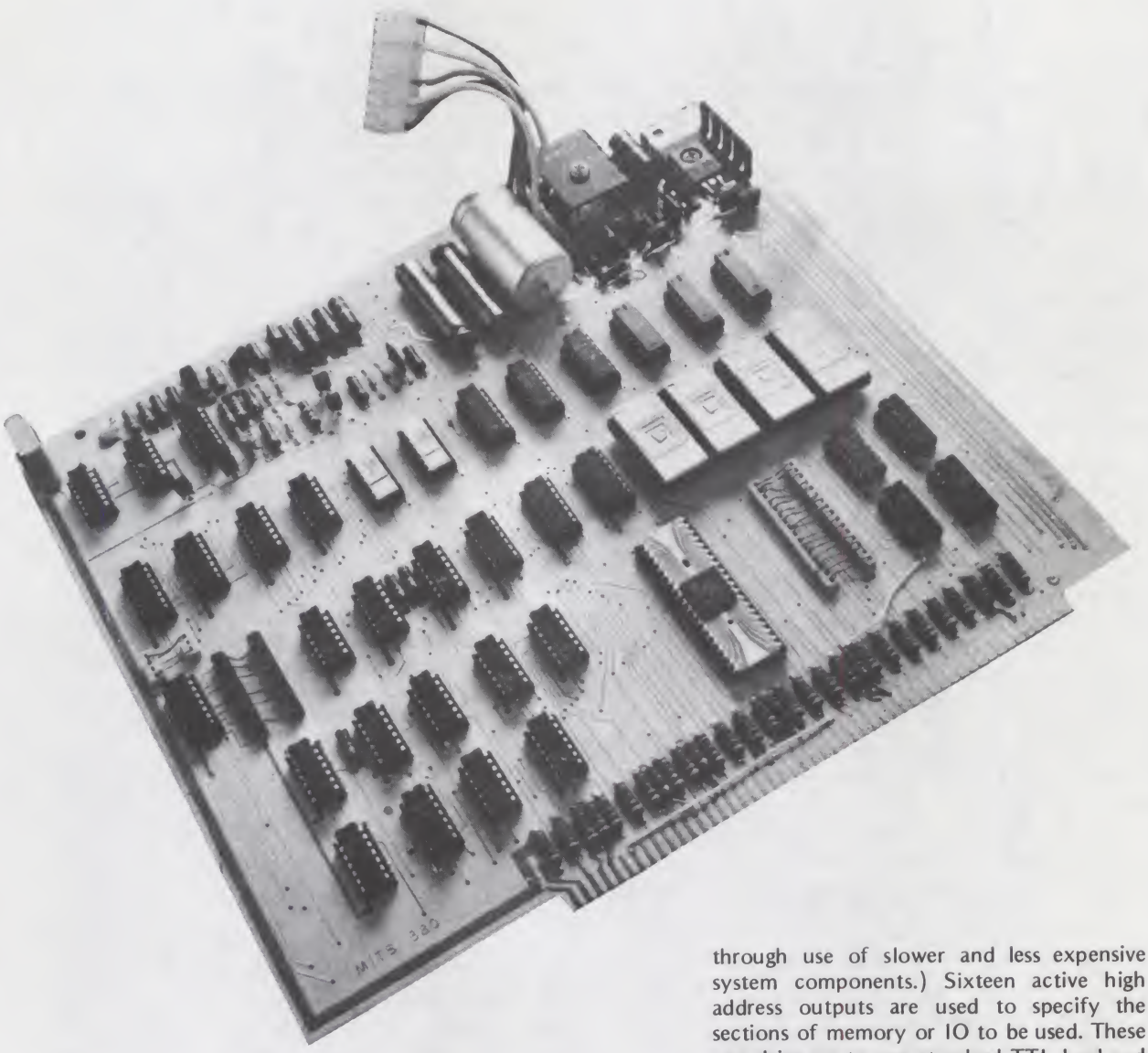
The 6800 is an 8 bit parallel processor using a bi-directional data bus and a 16 bit address bus. The address bus gives it the ability to directly address 65,536 bytes of memory. (Of course most configurations will have fewer than 65,536 bytes.) The instruction set consists of 72 basic instructions with various addressing modes giving a total of 197 different operation codes.

The 6800 has seven different addressing modes, with the available modes being a function of the type of instruction selected. The seven modes include the following:

- Accumulator Addressing – one byte instructions which specify either of the two accumulators use this mode.
- Immediate Addressing – two or three byte instructions with data specified in the instruction use this mode. In immediate mode instructions, one or two bytes of data follow the op code, depending upon the instruction involved.
- Direct Addressing – two byte instructions which allow the user to directly address the first 256 bytes of memory address space in the machine employ this mode.
- Extended Addressing – three byte instructions with a full 16 bit address in the second two bytes use this mode. There is no need to set up an on-chip register to access all of memory with the 6800. This mode is available for most data manipulation operations.
- Indexed Addressing – two byte instructions with this mode add the second byte of the instruction to the 16 bit index register to give the address of the operand.
- Inherent Addressing – certain one byte instructions imply the operands directly and thus do not need a separate address.
- Relative Addressing – all the branch instructions calculate the branch address by adding the second instruction byte to the current program counter plus two. The relative offset is treated as a signed two's complement number (8 bits) being added to the address in the program counter. This allows the user to branch to memory location +129 to −125 bytes from the location of the present instruction.

These various addressing modes may take a bit of getting used to, but once understood they allow for some very fast programs to be written.

The 6800 MPU contains three 16 bit registers and three 8 bit registers. The program counter is a two byte register which keeps track of the current address of the program. The stack pointer is also a two byte register which contains the next address in a variable length stack found in main memory. The index register is a two byte register used to store data or a memory address for indexed addressing operations. There are two single byte accumulators used



for holding operands and results from the arithmetic logic unit (ALU). The 8 bit condition code register indicates the results of an ALU operation. In this register there are two unused bits, kept at a logic one. The remaining six bits are used as the status flags for carry, overflow, zero, negative, interrupt and half carry.

There are several timing and control signals required to operate the MPU. Two clock inputs are required, phase 1 and phase 2. These must be nonoverlapping and run at the Vcc voltage level. Ordinary TTL will not drive these clocks properly. In the 680 the clock is a 2 MHz crystal controlled oscillator with logic to provide a 500 kHz two phase clock. (Although the 6800 is capable of running with a clock of up to a 1.0 MHz, MITS has set the speed of the 680 to 500 kHz in order to greatly reduce the cost through use of slower and less expensive system components.) Sixteen active high address outputs are used to specify the sections of memory or IO to be used. These can drive up to one standard TTL load and 130 pF. There are also eight bi-directional data lines with the same drive capability as the address lines. The $\overline{\text{HALT}}$ signal is an active low input which ceases activity in the computer. The RW (read or write) signal in the high state indicates that the processor is in a read condition; in the low state it indicates that the processor is in a write condition. The VMA (valid memory address) signal tells external devices that the processor has a valid address on the memory bus. The DBE (data bus enable) signal is the input which enables the bus drivers. The BA (bus available) signal indicates that the machine has stopped and that the address bus is available. $\overline{\text{RESET}}$ is used to reset and start the MPU from a power off condition. The $\overline{\text{IRQ}}$ (interrupt request) signal, when low, tells the processor to start an interrupt sequence. This can occur only if the interrupt mask bit in the condition code register is low. The $\overline{\text{NMI}}$ (nonmaskable interrupt)

signal is essentially the same as the $\overline{IRQ}$ signal except that it is not dependent on the condition code register.

Memory

The main printed circuit board on the 680 contains the basic memory for the unit also. This includes 1024 bytes of random access memory and provisions for another 1024 bytes of read only memory. The random access memory circuits being used are the 2102 static 1024 X 1 bit parts. Read only memories of the mask programmed type can be custom ordered, and are very expensive in small quantities. The 1702 type, ultra-violet erasable programmable read only memories are typically used in this system. These are 256 X 8 bit units, so four 1702As would be required to fill up the available space in the 680.

There is additional memory for the 680 on the drawing board at this time which may add up to 12 K bytes more storage to the unit.

IO Port

Also on the main printed circuit board is a built in IO port and the appropriate interface circuitry. This port may be configured as either an RS232 level port or either a 20 mA or 60 mA current loop TTY level port. This means it can be interfaced with proper software to the old Baudot type Teletypes, such as the Model 19 and Model 28 machines.

The entire design of the 680 is greatly simplified due to the 6800's use of memory address space for IO addressing. The processor uses addresses to refer to IO devices as well as memory, rather than have special IO instructions and a separate IO bus. Within the limits of practical engineering, programming and memory requirements, as many IO devices as desired can be added to a 6800 microprocessor system. No logical limitation is built into the instruction set. MITS also has additional IO interfaces on the drawing board at this time; although availability of this and the additional memory boards will be greatly influenced in their development by customer response.

Control and Indication

On the fully user-programmable version of the 680, the front panel assembly contains a RUN/HALT switch with an LED indicator for each switch position. There is a RESET switch with no indicator, and another indicator for the AC power switch which is located on the back panel of the unit. The switches for the 16 address lines and 8 data lines, and their associated indicator lights, are also located on the front panel assembly of the fully programmable model. There is also a DEPOSIT switch.

The DEPOSIT, RESET, DATA and ADDRESS switches are enabled only when the RUN/HALT switch is in the HALT position. To view the data in a particular memory address, the RUN/HALT switch must first be in the HALT position and then the ADDRESS switches may be set to the required address. The data located at that particular address will then appear on the DATA LED indicators above the DATA switches.

To write data in a desired location, once the correct address has been set on the address switches, the appropriate data should be entered on the DATA switches and then the DEPOSIT switch activated. Since the address bus is already connected to the switches by being in the HALT state, a write pulse causes the data to be written into the selected RAM address.

When the RESET switch is activated, the processor itself resets. This initiates a restart sequence, pulling the address bus to its high state and causing hard-wired data on the board to be used as the restart address.

On the dedicated program versions of the 680, most of these functions are taken care of by ROM or PROM. The only controls available to the user are the AC power and RESET switches.

Power Supply

The 5 volt supply to the computer is supplied from the power transformer through a conventional bridge rectifier and filter capacitors and voltage regulator IC. A 32 volt winding on the transformer is used to generate the unregulated ±16 volts required for a TTY interface, and a −16 volt line is fed to four zener diode regulated outputs to provide four −9 volt lines for the PROMs.

The transformer itself, along with the power switch, is located on the computer's back panel. There are also provisions for installing a cooling fan when necessary.

As far as software goes, MITS has a package available similar to the 8800's Package One. This includes an editor, PROM monitor and assembler. This all goes to make the ALTAIR 680 a rather powerful little machine. There is also the possibility for further software development.

MITS has decided to await customer response to determine the course of further 680 development in both the areas of software and hardware.

Although it's not quite as powerful as the ALTAIR 8800, the ALTAIR 680 is mighty close and costs less.■

Photo 1: When you first open your KIM-1 box, you see a thick layer of documentation, including a large wall chart of the system's hardware details, an MCS650X Instruction Set Summary card, KIM-1 User Manual, Programming Manual *and* Hardware Manual. *Also shown in this picture is the KIM monitor listing copy which must be requested separately and is a must if you are to take advantage of KIM's subroutines in applications programs.*

A Date with KIM

Richard S Simpson
314 Second Av
Haddon Heights NJ 08035

Here it is! In the November 1975 BYTE, Dan Fylstra reviewed the capabilities of the MOS Technology 6501 microprocessor chip in an article titled "Son of Motorola" (page 56). The article stated that "it will be three to six months before you see (a 6501) designed into a kit. . ." Well, MOS Technology has gone one better and introduced not a kit, but a completely assembled, tested and warranteed microcomputer with a price tag of only $250! Using the 6502 processor chip (a 6501 with an on-chip clock), the microcomputer features 1 K of RAM, 2 K of ROM containing the system executive, a complete audio cassette interface, a serial terminal interface, 15 bidirectional IO lines, a 23 key keypad and a six digit LED display. This completely assembled one board computer has all the programming features of the 6502 at a very competitive price.

If you have been hesitating over buying a microcomputer because of the difficulty of assembly and the fear that it won't work when you're finished, KIM-1 is for you. The only assembly required is to attach six self adhesive plastic feet to the back of the KIM-1 printed circuit board and attach a +5 volt, 1 ampere power supply to the 44 pin edge connector provided. You'll also need a supply of +12 V for the cassette interface; but a handful of flashlight batteries should work fine since only about 50 mA of +12 V is required, and that only when the interface is being used.

The name KIM is an acronym for Keyboard Input Monitor. The name really describes the ROM executive routines, not the whole unit, but it's a pleasant change from the manufacturer's name followed by a number. It's also significant that the system derives its name from its software.

The KIM-1 board can be operated in one of two modes: using the on board keypad and LED display, or using a serial terminal. The keypad and hexadecimal display is infinitely easier and less error prone than throwing toggle switches and reading results from binary lamps. In fact, for program entry and many simple applications, I prefer the 23 key keypad and bright LED display to my slow, noisy Teletype. The keys have a good, positive "feel" to them (MOS Tech-

Photo 2: The KIM-1 processor as it is removed from its box. The MOS Technology product comes in a neat package which has one foam padded and static protected KIM-1 board as its bottom layer.

nology should know about such things, since they are a major manufacturer of chips for calculators).

The switch in the upper right corner of the keypad puts the machine in single instruction (not single cycle) mode. When the switch is "on," each depression of the "GO" button causes a single instruction of your program to be executed. Control is then returned to the executive program in ROM and the contents of all six machine registers (PC, X, Y, S, P, and the accumulator) are stored in fixed memory locations where you can easily examine them through the keypad or terminal and then "GO" to the next instruction. This is an important capability, since if you just halt a microprocessor after each instruction there is no way of examining the registers (they're all inside the chip!).

I won't go into any detail on the instruction set (see Dan Fylstra's article for that) except to say that it is comprehensive. The variety of addressing modes makes complex programming (especially when processing lists) a lot easier. The 6502 architecture has no IO register or IO instructions, so any memory location can become an IO "port" if you build the hardware for it. KIM comes with a built-in 15 line bidirectional IO interface. TTL levels are acceptable, of course, and one of the lines can supply enough current (5 mA) to directly drive a power transistor. The manual shows how to use it to drive a small speaker for "microprocessor music" programmed in a manner similar to the Kluge Harp of October BYTE (page 14). Each line can be separately programmed for input or output by writing a status word into the correct memory location.

The cassette interface is carefully thought out and should be foolproof. Half of the executive ROM is devoted to the cassette interface software, which includes rudimentary file management and sophisticated programmed equivalents to UART operation. This software allows multiple dumps to a single cassette. A header written on each output segment allows you to say, in effect, "find me program number 34 on the tape and load it starting at location. . ." A checksum is stored at the end of each segment and the user is immediately informed if the computed checksum doesn't match when the tape is read back in. You can even record voice data between segments of digital data – the interface will ignore the voice. This feature could be used to verbally record the instructions for a game and then automatically load and run it. Both high and low level outputs are provided to interface with any type of cassette recorder. It's not a vital feature, but it indicates the care with which the entire system has been thought out.

The TTY interface is for a standard 20 mA current loop (figure 1 shows how I modified it for an RS-232 interface). A unique feature of the software is automatic data rate detection. As soon as the system is powered up, the user types a RUBOUT character on his terminal. The software

If you have been hesitating over buying a microcomputer because of the difficulty of assembly and the fear that it won't work when you're finished, then KIM-1 is for you.

KIM-1 derives its name from the software, a significant indication of the importance of good user support programs.

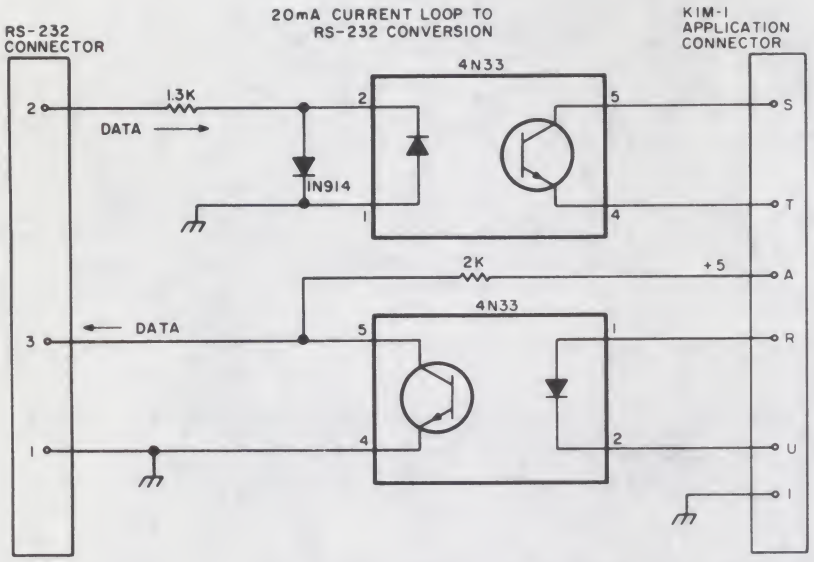


Figure 1: One way to interface KIM-1 with an RS-232 compatible terminal is illustrated in this diagram. Opto isolators are used to accomplish the coupling. The RS-232 pins 1, 2 and 3 will be sufficient for terminals which do not involve handshaking; on some terminals, pins 5, 6, 8 and 20 of the standard RS-232 plug may have to be tied together to bypass handshaking signals.

calculates the data rate (anything from 110 to 1200 baud is acceptable) and automatically adjusts all further conversation to that rate. No additional timing standards or switches are required for the interface.

The real beauty of the terminal interface is in the software, not the hardware. On request, MOS Technology supplies a complete listing of KIM. All the executive ROM software subroutines are documented and available to the user referencing this well-commented listing. Thus, to print the contents of the accumulator in hex on the terminal requires a simple one-instruction subroutine call. Those readers who have had to invent their own terminal interface software will have a deep appreciation for this capability. Similar subroutines are provided for reading characters from the terminal or keypad, printing one or a string of ASCII characters, or writing digits in the LED display.

To round out the terminal interface, software is provided in ROM to read and punch paper tape if your terminal is so equipped. Again, care has been taken to provide checksums on the punched tape which is automatically verified when the data is reloaded. This kind of attention to detail reflects the high caliber of the MOS Technology offering. One reason for this is the fact that MOS Technology sells a sizeable portion of the KIM units to industrial users. This policy of building to industrial rather than consumer standards is also evident in the quality of the PC board, the PC artwork, and the fact that the board is coated with a solder mask, a plastic coating which protects the printed wiring. To further emphasize their faith in KIM, MOS Technology gives you a 90 day warranty on the entire KIM system, not just the components. Mail-in repair service is available even after the warranty expires.

Interval Timer

Another feature of KIM which is finding its way into more and more microprocessors is the inclusion of a program controlled interval timer. The KIM board actually contains two programmable timers, but one is dedicated to control the keypad and cassette interface. Any count from 1 to 256 can be loaded into the timer by writing to the timer's memory location. The user can control the scale of the timer by programming it to count every clock pulse or to count every 8th, 64th, or 256th clock pulse. This prescaling of the counter is done by decoding the last two address bits for the timer. Thus, the time scale is controlled by which memory location is loaded with the count. You might consider using a similar scheme whenever you have to write more than eight bits to control an external device: Just use the least significant address bits as data.

When the timer has counted down to zero, a software interrupt is generated, notifying the program that "time has run out." As soon as the interrupt is issued, the timer continues to count past zero (into negative numbers) at the clock rate. If the program is servicing other interrupts, it can read the counter register to determine how long ago (in machine cycles) the timer interrupt occurred.

Memory Expansion

If you are interested in expanding the KIM memory beyond the 1 K provided, you'll be glad to know that all the decoding for the first 4 K is provided right on the KIM board. All you need to provide is 4 K more of RAM chips and some buffers.

There are two connectors on the KIM board; one called the expansion connector is for adding memory and bus oriented devices. The second connector, called the application connector, interfaces directly to the outside world. The expansion connector has all the address, data, and memory control signals. The application connector terminates the lines for the audio cassette, the terminal send and receive signals, and the 15 IO lines. Connections are also provided so that the keypad can be removed from the KIM board and mounted elsewhere, a useful feature if

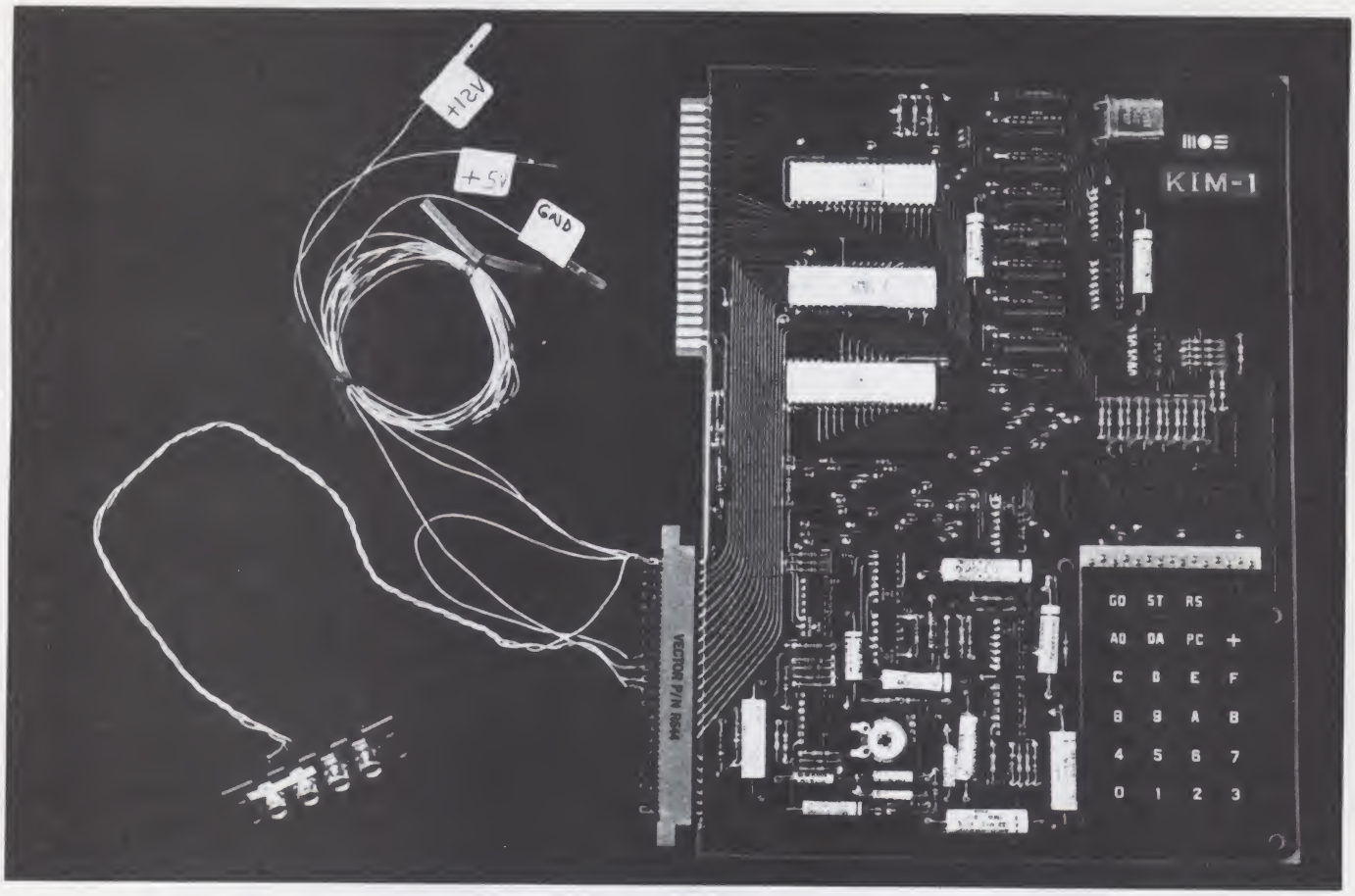


Photo 3: Wiring for Stand Alone Use. With due respect to the instructions in the KIM-1 user's manual, and addition of some miscellaneous parts, the results will be a wiring harness similar to that shown here. Wires have been attached and labelled for GND, +5 volts and +12 V. The audio cassette interface has been brought out to an RCA-style phono jack assembly purchased at a retail electronics store, along with interconnection cables for the recorder input and output. This setup enables the user to enter and test out programs through the KIM-1 control panel and LED display.

you want to wrap up the KIM printed circuit board in sheet metal along with a power supply.

Documentation

The documentation which comes with KIM is thorough and comprehensive. Any regular reader of BYTE should have no trouble following the details of the 200 page programming manual. There are plenty of examples; and the explanation of the operations which occur in each machine cycle of multicycle instructions, while not essential, is very instructive. Special sections of the manual are devoted to interrupt handling and use of the stack pointer. This is vital information often glossed over in other manuals.

I have to admit that I have not yet digested all the information in the 150 page hardware manual which came with my KIM, since my main interest is in programming my system as soon as possible. However, the manual seems to have a solid emphasis on IO interfacing and usage of the control lines.

The third manual provided is the actual KIM user's manual. This 100 page document explains how the keypad, cassette interface and terminal interface are to be used. It gives a few basic programming examples, including an example which goes through the entire design of a simple application using the IO lines. My only complaint is that no sample program was provided for the use of the programmable timer or the ROM executive subroutines. Also, the listing of KIM should have been supplied as a standard item.

Also included in the package is a pocket reference card for the instruction set and a wall size schematic of the entire KIM board. Two other useful documents are available from MOS Technology on request. One is the manual for the 6500 cross-assembler, which is available on several commercial time-sharing systems. The other is the well-commented listing of the executive programs stored in ROM as mentioned earlier.

In summary, the KIM is an excellent microcomputer requiring no assembly and which is very attractively priced. The only auxiliary equipment required is a power supply and a cassette recorder. The manuals are among the best available and the built-in keypad and display make KIM easy to get started with. The terminal interface and ease of memory expansion make it easy to upgrade as your requirements increase. Make a date with KIM — you'll enjoy it! ■

True Confessions:

How I Relate to KIM

Yogesh M Gupta
118 E Main St
New Concord OH 43762

I recently purchased a KIM-1 microcomputer card from MOS Technology (See "KIM-O-Sabee?" in the April BYTE, page 14 and "A Date With KIM" in the May issue, page 8). In my opinion, KIM-1 offers one of the best bargains to a computer experimenter for the price ($245 for the card + $4.50 for shipping and handling). However, the hobbyist may be faced with a few problems, as I was. The intent of this article is to solve some of these problems.

Clock Stretch and Random Access Memories

The cheapest random access memory in experimenters' markets today is the standard 2102 static memory which averages approximately 0.15¢ per bit. During a write cycle, the inexpensive slow versions of this device require the data to be stable for 800 ns before the trailing edge and data hold time of 100 ns after the trailing edge of the write pulse. Even if the MOS Technology 6502 processor is slowed down to 250 kHz to obtain the data stability, there is still not enough data hold time for the slow chips.

I solved this problem by implementing the circuit shown in figure 1. This circuit allows the 6502 processor to use a mixture of fast and slow 2102 memory devices in the same system. The processor cycle is main-

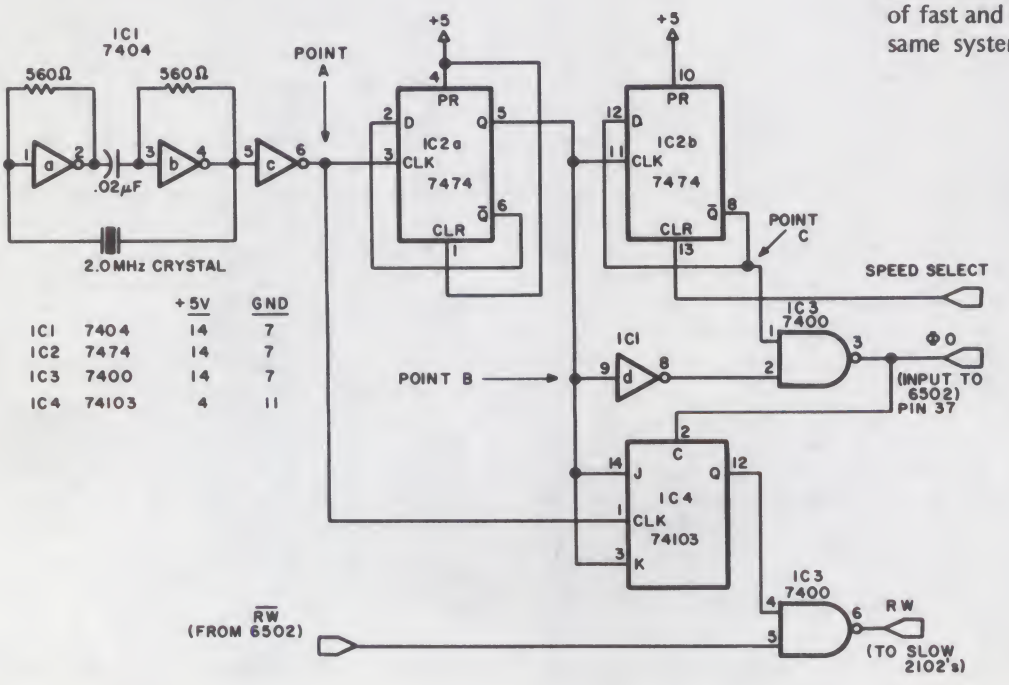


Figure 1: A circuit which creates an alternative slow clock cycle for the 6502 processor on the KIM-1 board under control of a "SPEED SELECT" line generated by slow memories. SPEED SELECT = 0 for fast cycles, SPEED SELECT = 1 for slow cycles. This circuit requires a 2.0 MHz crystal.

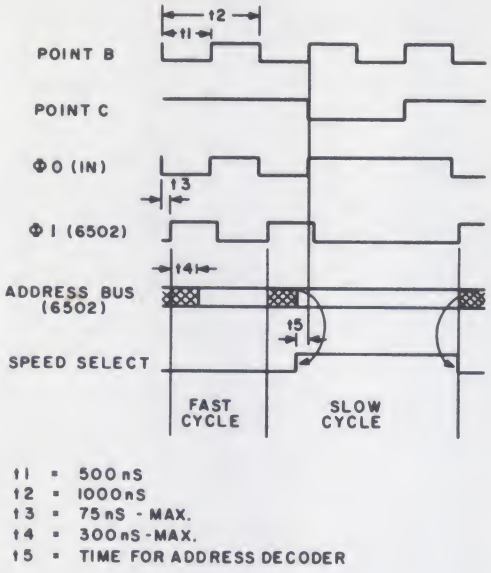


Figure 2: Method 1 SPEED SELECT Discipline. In this method, fast cycles are the rule, slow cycles are the exception. Refer to figure 1 for points B and C. Invalid data on the address bus is indicated by the cross-hatched areas.

tained at 1.0 μs for the fast memory access, while for the slower 2102s, the cycle is automatically stretched to 2.0 μs.

Sometimes integrated circuits behave in ways that are not predicted by, or are overlooked by, their manufacturers. This modification of KIM-1 to enable the clock stretching function is accomplished by removing the usual KIM-1 6502 clock generation circuitry, and simply driving the $\phi 0$ pin of the 6502 directly from a TTL clock source which is external to the chip. This mode of operation is not documented in the *6502 Hardware Manual* of MOS Technology, but it worked quite satisfactorily in my system . The intention of the designers of the 6502 was that the clock generation logic on the chip would be used with external components setting the frequency of the oscillator.

The SPEED SELECT signal to stretch the

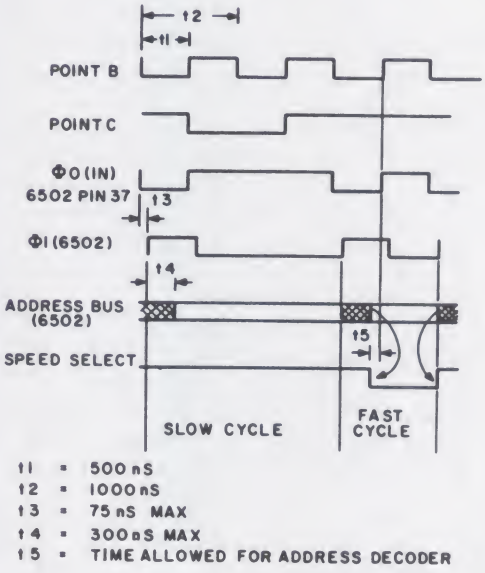


Figure 3: Method 2 SPEED SELECT Discipline. In this method, slow cycles are the rule, fast cycles are the exception. Refer to figure 1 for points B and C of the timing diagram. Invalid data on the address bus is indicated by the cross-hatched areas.

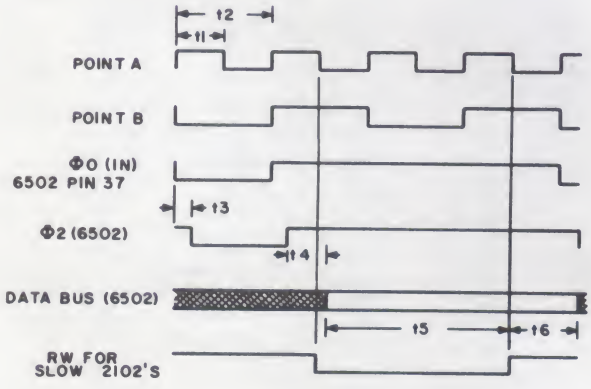


Figure 4: Write Cycle for Slow 2102 Memories. The timing requirement is that valid data be present on the bus when the RW signal to the memory changes from 0 (write state) to 1 (read state). The crosshatched areas indicate when data is invalid on the data bus.

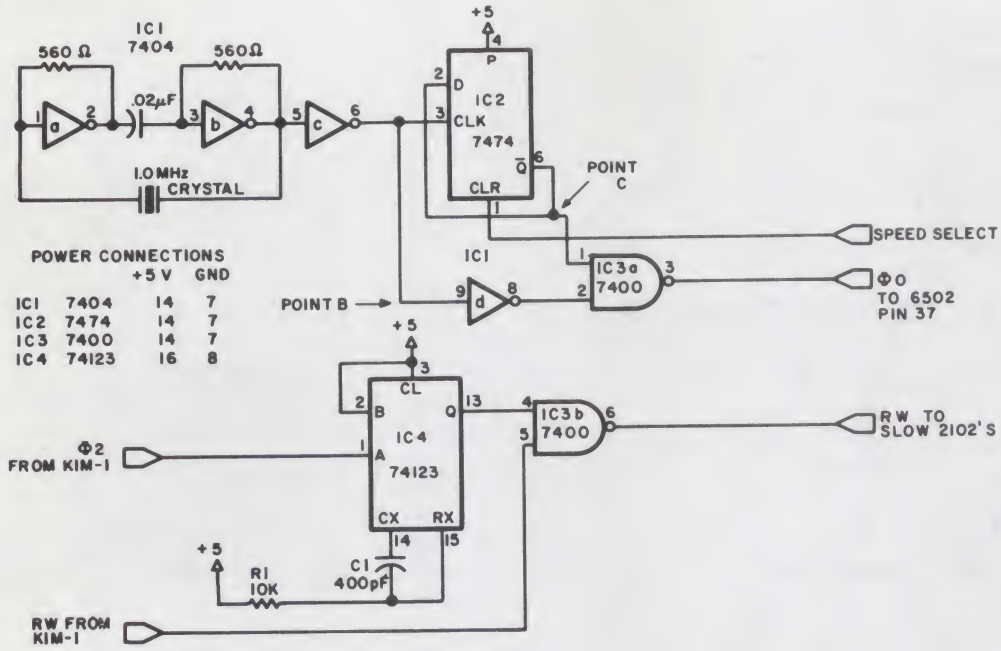


Figure 5: Alternate Slow Clock Generation Circuit. In this circuit, the original KIM-1 crystal can be used, since a digitally controlled timing cycle is replaced by the 74123 oneshot.

cycle is generated by address bus decoding logic using one of the following two methods.

Method 1: Normally the SPEED SELECT signal is kept low so the processor cycle is 1.0 μs. However, this signal goes high when the processor addresses the slow memory region causing the cycle to stretch to 2.0 μs. See figure 2 for the timing relationships.

Method 2: Normally the SPEED SELECT signal is kept high so that the processor cycle time is 2.0 μs to access slow memory. However, this signal goes low when the processor addresses the fast memory devices causing the cycle time to be only 1.0 μs. See figure 3 for the timing relationships.

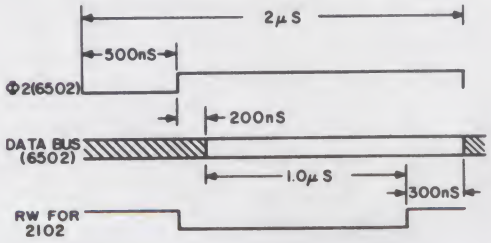


Figure 6: Write Cycle for Slow 2102 Memories using the circuit of figure 5. The output pulse width of RW is adjusted to 1.2 μs nominally. (Check the results on your scope even if you use other than precision parts of the values shown for R1 and C1 in figure 5.)

The circuit shown in figure 1 will allow a data stability of 950 ns before the trailing edge and data hold time of 350 ns after the trailing edge of the Write Pulse for the slow 2102s. See figure 4 for the timing relationships.

However, the KIM-1 board comes with a 1.0 MHz crystal. Figure 5 shows an alternative circuit using a 1.0 MHz crystal. The timing relationships to control SPEED SELECT signal are the same as shown in figures 2 and 3. The RW signal for the slow memory is generated in this case by using a 74123 oneshot. The value of the RC constant for the 74123 is chosen to provide a nominal output pulse width of 1.2 μs. This allows a data stability of 1.0 μs before the trailing edge and data hold time of 300 ns after the trailing edge of the write pulse for the slow memories. Figure 6 shows the resultant timing relationships. It should be noted that the output pulse width of the 74123 can only tolerate a $\pm$ 16.66% variation, and still permit successful operation of the 2102 memory devices. This tolerance may require selection of precision parts for the external resistor and capacitor of the oneshot.

Bus Expansion

The 6502 bus is only capable of driving one standard TTL load. If more drive capability is needed, the tristate drivers such as the 8T97 or DM8833 parts may be used.

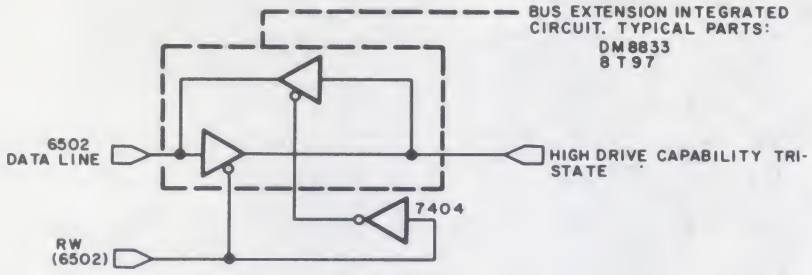


Figure 7: Use of a Bus Extension Integrated Circuit. In order to tie in extra memory or peripherals, a bus extension is required. The typical logic diagram of a simple attempt which will not always work is shown here. (Conflicts can arise.)

However, you must be very careful when using an extended data bus. If you enable the drivers by RW signal as shown in figure 7, then during read mode, the drivers for the existing KIM-1 memory (eg: 74125s) can be turned on simultaneously. The low level output current of 74125s is only 16 mA and is not sufficient to pull down a turned on 8T97 type driver to logic 0 level. Therefore, during read mode the bus extension tristate drivers should be turned off when the existing on board KIM-1 memory (RAM, 6530-002 and 6530-003) is being accessed as shown in figure 8. In actual implementation the DECODE ENABLE signal may be the same as the one needed on the application connector of the KIM board (when more than 8 K memory is needed).

Interrupt Prioritizing Logic

The *KIM-1 Hardware Manual* (Section 2.3.3) describes a few approaches to implement interrupt priority logic; but I found them either inefficient (software time) or expensive (use of ROM). The circuit shown in figure 9 provides a cost effective compromise. The interrupts from the peripheral devices are latched in by the $\phi 2$ signal. This inhibits the priority encoder from generating a false vector (if the interrupts from the peripherals are changing while the 6502 is fetching the vector). In response to IRQ, the 6502 fetches the vector from hexadecimal locations FFFE and FFFF. During these fetch cycles, the 6530-002 is disabled by letting the decode enable signal go high on the application connector. Therefore, the vector generated by this circuit is fetched by the 6502 instead, and the program goes to one of the locations from 0200 to 021C. This segment of memory serves as a vector table with pointers to the individual interrupt service routines as follows:

```
0200        JMP VEC0
0204        JMP VEC1
0208        JMP VEC2
020C        JMP VEC3
0210        JMP VEC4
0214        JMP VEC5
0218        JMP VEC6
021C        JMP VEC7
```

The actual service routines will reside in locations VEC0 through VEC7 for the respective interrupts. It should be noted that each vector in the table requires 4 locations. (Only 3 locations are needed for a jump but

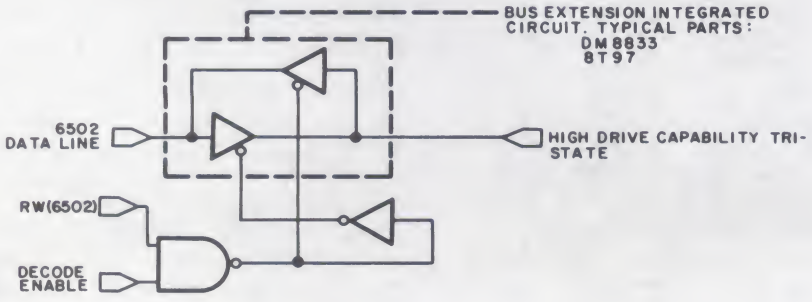


Figure 8: Adding a gate to the bus extension control resolves a potential conflict through the use of a decode enable signal which is high if external memory is referenced, low if memory on the KIM-1 board is referenced.

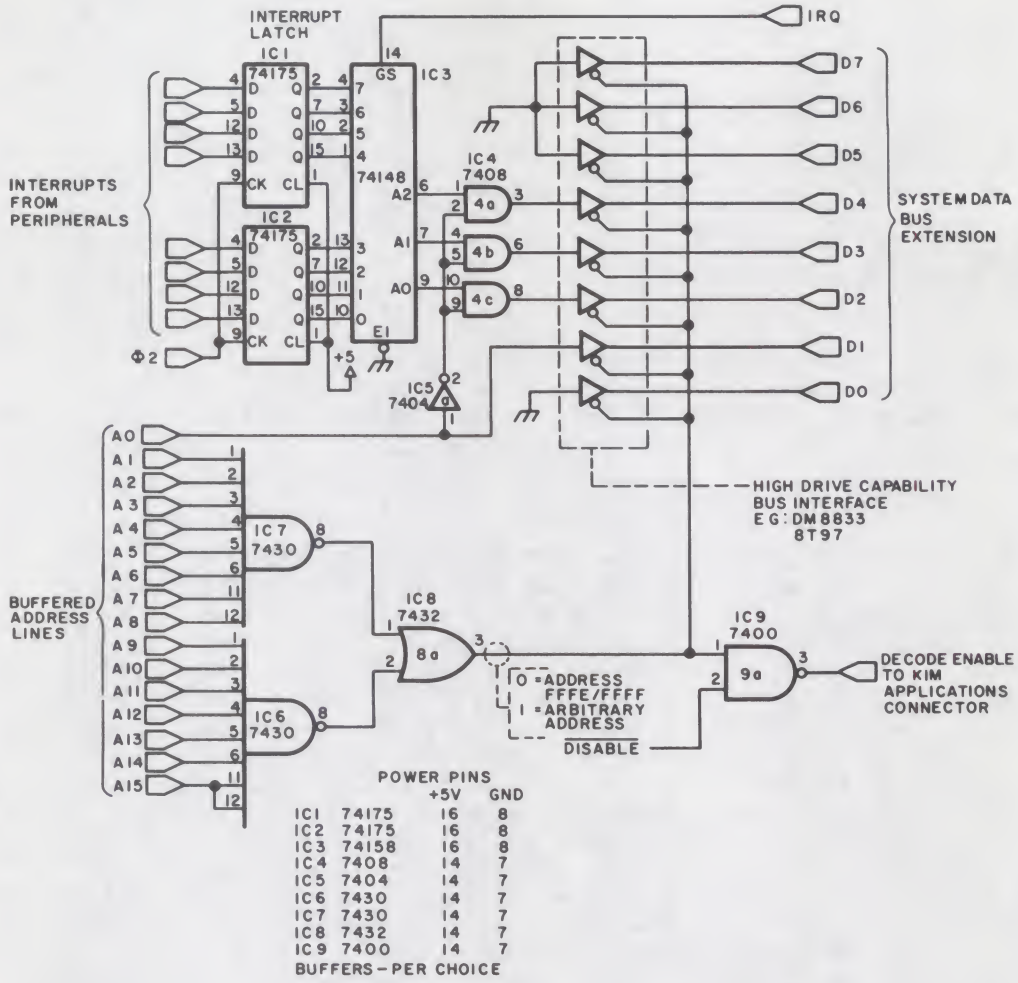


Figure 9: By disabling normal address decode through the DECODE ENABLE pin for the KIM-1 applications connector, an alternate source of the interrupt vector at locations FFFF and FFFE can be created which accomplishes interrupt prioritizing functions.

the extra location is a requirement of a simple hardware design.) JMP instruction takes only three locations, so your software might use the fourth location to save the accumulator, eg:

0200	PHA
0201	JMP
0202	VEC0 (LOW)
0203	VEC0 (HIGH)

This architecture will re-map the 1 K resident RAM on the KIM board as follows:

0000 through 00FF	Page 0
0100 through 01FF	Stack
0200 through 021F	Vector Table
0220 through 03FF	Applications

The disable signal in figure 9 will deselect existing KIM-1 memory when low. This is implemented for memory expansion as described earlier. However, if memory expansion is not desired the signal may be fixed to a logic 1 level.

Halt?

Another problem one faces is how to debug the software when the processor does not have a HLT instruction. You can single step the program instructions on KIM-1, but this feature does not help the programs which involve multiple levels of loops or critical peripheral timing controls. The obvious solution is to use the BRK (software interrupt) instruction. However, this would require software overhead in every interrupt service routine to determine whether it was a hardware or a software interrupt. On the KIM-1 system, I found the sequence JSR 05 1C (Jump to subroutine at location 1C05) more useful for this purpose instead. The execution of JSR causes the program to jump to an input monitor loop and display of the address (PC + 2) on the KIM board. PC is the location where the JSR was executed.■

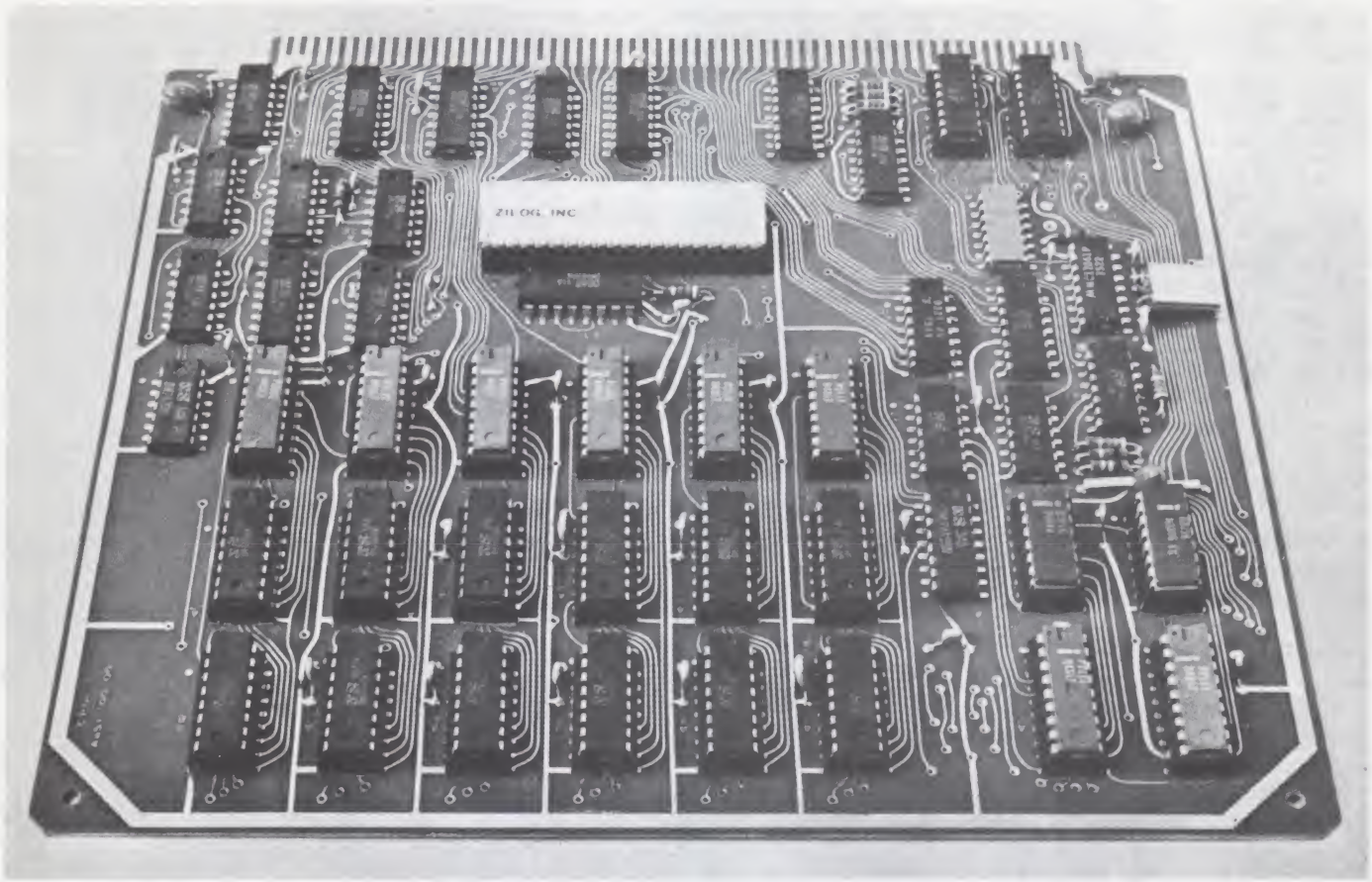


Photo 1: The Z80 microprocessor evaluation board.

Microprocessor Update: Zilog Z80

Burt Hashizume
PO Box 172
Placentia CA 92670

One feature of the Z80 not found in other 8 bit microprocessors is a built in dynamic MOS memory refresh algorithm which employs unused memory cycles to do hidden (from software timing) refresh operations.

Zilog, a fairly new company in Los Altos CA, has been sampling an 8 bit microprocessor, the Z80, since early this year. The Z80 is a "third generation," single chip, NMOS microprocessor, which is completely software compatible with Intel's 8080A. Its 158 instructions include the 8080A's 78 instructions as a subset. Because the 8080A is probably the most widely used 8 bit microprocessor on the market today and because of the Z80's upward software compatibility, this article evaluates the Z80 in comparison to the 8080A.

Physical and Electrical Characteristics

The Z80 processor is packaged in the standard 40 pin dual in line package; however, even though the Z80 is software compatible with the 8080A, it is most definitely not pin compatible. (See figure 1 and table 1 for pinout definitions.) There are numerous differences between the two processors as far as electrical characteristics are concerned.

The 8080A requires three voltage levels, +12, +5, and −5 V. A high voltage two phase clock is also required. Maximum speed is a 480 ns clock period. Finally, some sort of system controller is needed to separate the system control signals from the data bus. This all makes for a fairly complex system design around the 8080A.

On the other hand, it is very easy to design a system around the Z80. It requires only a single +5 V power supply because the

technology used is of the same type used by Motorola in its 6800 microprocessor, which also requires a single 5 V power supply. The Z80 requires a single phase 5 V clock. Maximum frequency is 2.5 MHz for a 400 ns clock period. System control signals, such as memory read and write, have separate pins from the processor and are not time shared with the data bus. An additional feature not found on any other microprocessor at the time of this writing is the capability to refresh dynamic memory.

Because the Z80 is upward software compatible with the 8080A, the internal architectures are similar. (See the register configuration in figure 2.) Both have 16 bit program counters and stack pointers as well as a register array of six general purpose registers, (B, C, D, E, H and L), an accumulator (A), and a flag register (F).

The Z80 has numerous additional characteristics. It has an additional duplicate register array consisting of 8 registers (A', F', B', C', D', E', H' and L'). These can be switched with the primary register array for fast interrupt processing. There are also two 16 bit index registers (IX and IY) for increased addressing capability and easier data manipulation. An 8 bit interrupt vector register (I) expands the capability and increases the power and speed of interrupt handling by the processor. Finally, an 8 bit memory refresh register (R) automatically increments after every instruction fetch and refreshes memory while the processor is not using the bus. Thus the execution time of the system is not increased due to refresh overhead.

Software

Now that we have seen the hardware aspects of the Z80 and how it compares to the 8080A, let's take a look at its instruction set. The fact that the Z80 has 158 instructions versus the 8080A's 78 gives only a small indication of its technological superiority in this area. The instruction set can be broken up into two aspects, addressing modes and instruction groups.

Since the Z80 is software compatible with 8080A, it necessarily has the same addressing modes as the 8080A. The modes in common are register addressing, register indirect addressing, direct addressing, and immediate addressing.

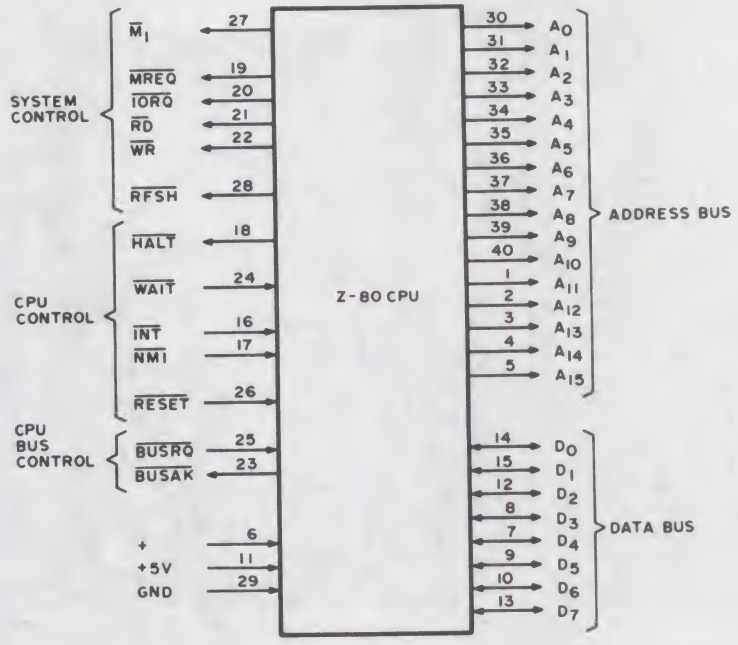


Figure 1: Pin configuration of the Z80 processor. Of particular note to custom hardware hackers is the "M_1" line which gives users the possibility of identifying instruction cycles.

Table 1: Signal list for the Z80 processor. This table lists each active pin of the Z80 with a short explanation of its purpose.

A_0–A_{15} (Address Bus)	Tri-state output, active high. A_0–A_{15} constitute a 16 bit address bus. The address bus provides the address for memory (up to 64 K bytes) data exchanges and for IO device data exchanges. IO addressing uses the 8 lower address bits to allow the user to directly select up to 256 input or 256 output ports. A_0 is the least significant address bit. During refresh time, the lower 7 bits contain a valid refresh address.
D_0–D_7 (Data Bus)	Tri-state input and output, active high. D_0–D_7 constitute an 8 bit bidirectional data bus. The data bus is used for data exchanges with memory and IO devices.
$\overline{M_1}$ (Machine Cycle one)	Output, active low. $\overline{M_1}$ indicates that the current machine cycle is the OP code fetch cycle of an instruction execution.
$\overline{MREQ}$ (Memory Request)	Tri-state output, active low. The memory request signal indicates that the address bus holds a valid address for a memory read or memory write operation.
$\overline{IORQ}$ (Input/Output Request)	Tri-state output, active low. The $\overline{IORQ}$ signal indicates that the lower half of the address bus holds a valid IO address for a IO read or write operation. An $\overline{IORQ}$ signal

Table 1 (continued).

	is also generated when an interrupt is being acknowledged to indicate that an interrupt response vector can be placed on the data bus. Interrupt Acknowledge operations occur during M_1 time while IO operations never occur during M_1 time.
$\overline{RD}$ (Memory Read)	Tri-state output, active low. $\overline{RD}$ indicates that the processor wants to read data from memory or an IO device. The addressed IO device or memory should use this signal to gate data onto the processor data bus.
$\overline{WR}$ (Memory Write)	Tri-state output, active low. $\overline{WR}$ indicates that the processor data bus holds valid data to be stored in the addressed memory or IO device.
$\overline{RFSH}$ (Refresh)	Output, active low. $\overline{RFSH}$ indicates that the lower 7 bits of the address bus contain a refresh address for dynamic memories and the current $\overline{MREQ}$ signal should be used to do a refresh read to all dynamic memories.
$\overline{HALT}$ (Halt state)	Output, active low. $\overline{HALT}$ indicates that the processor has executed a HALT software instruction and is awaiting either a non maskable or a maskable interrupt (with the mask enabled) before operation can resume. While halted, the processor executes NOPs to maintain memory refresh activity.
$\overline{WAIT}$ (Wait)	Input, active low. $\overline{WAIT}$ indicates to the Z80 processor that the addressed memory or IO devices are not ready for a data transfer. The processor continues to enter wait states for as long as this signal is active. This signal allows memory or IO devices of any speed to be synchronized to the processor.
$\overline{INT}$ (Interrupt Request)	Input, active low. The Interrupt Request signal is generated by IO devices. A request will be honored at the end of the current instruction if the internal software controlled interrupt enable flip flop (IFF) is enabled and if the $\overline{BUSRQ}$ signal is not active. When the processor accepts the interrupt, an acknowledge signal ($\overline{IORQ}$ during M_1 time) is sent out at the beginning of the next instruction cycle. The processor can respond to an interrupt in three different modes that are described in detail in the Zilog documentation.
$\overline{NMI}$ (Non Maskable Interrupt)	Input, active low. The non maskable interrupt request line has a higher priority than $\overline{INT}$ and is always recognized at the end of the current instruction, independent of the status of the interrupt enable flip flop. $\overline{NMI}$ automatically forces the Z80 processor to restart to location 0066 hexadecimal. The program counter is automatically saved in the external stack so that the user can return to the program that was interrupted.
$\overline{RESET}$	Input, active low. $\overline{RESET}$ forces the program counter to zero and initializes the processor. The processor initialization includes: 1) Disable the interrupt enable flip flop 2) Set Register I = 00 3) Set Register R = 00 During reset time, the address bus and data bus go to a high impedance state and all control output signals go to the inactive state.
$\overline{BUSRQ}$ (Bus Request)	Input, active low. The bus request signal is used to request the processor address bus, data bus and tri-state output control signals to go to a high impedance state so that other devices can control these buses. When $\overline{BUSRQ}$ is activated, the processor will set these buses to a high impedance state as soon as the current processor machine cycle is terminated.
$\overline{BUSAK}$ (Bus Acknowledge)	Output, active low. Bus acknowledge is used to indicate to the requesting device that the processor address bus, data bus and tri-state control bus signals have been set to their high impedance state and the external device can now control these signals.

- *Register addressing.* The opcode itself specifies a register or register pair in which the data is contained. An example would be to load the data in register B into register D.
- *Register indirect addressing.* The opcode specifies a register pair which contains a 16 bit address. This address points to the data in memory or is an address to be loaded into the program counter (PC). An example would be to load the accumulator with data in memory pointed to by the HL register pair.
- *Direct addressing.* The opcode is followed by two bytes of operand. These two bytes are either a 16 bit address pointing to data in memory or a 16 bit address to be loaded into the PC. For example, in a jump instruction, the two bytes indicate an address to which program control is transferred.
- *Immediate addressing.* The opcode is followed by one or two bytes of operand. This operand is the data itself to be used. An example is load accumulator immediate which moves an 8 bit operand into the accumulator.

To these addressing modes, the Z80 has added three more powerful modes. These are indexed addressing, relative addressing, and bit addressing. The first two are somewhat similar to index and relative addressing in the Motorola 6800 microprocessor.

- *Indexed addressing.* The opcode is followed by an 8 bit displacement. This displacement is a *signed* two's complement number to be added to the contents of one of the two index registers. The result is a 16 bit effective address. The contents of the index register are unchanged.
- *Relative addressing.* The opcode is followed by an 8 bit signed two's complement number. The number is added to the contents of the program counter and the result placed back in the PC. This results in being able to execute program jumps within a range of +129 to −126 bytes using only a two byte instruction. Since most programs have a lot of jumps to locations relatively close to current locations, using relative addressing will significantly reduce program size. Another advantage is the ability to write relocatable code using relative addressing.
- *Bit addressing.* Three bits in the opcode itself specify one of eight bits in a byte to be addressed. This byte

could be the contents of a register or of a memory location. An example would be to set bit 6 in memory pointed to by index register, IX, displaced by −20.

The Z80 instruction set's increase of 80 instructions over the 8080A's didn't come from just increasing the number of addressing modes. There are instructions which don't exist in any other microprocessor. The instruction set will be broken up into groups by their function.

Load and Exchange Instructions

This group includes all the instructions that move data to and from registers, such as load B from D, load C from memory, store HL into memory, push IX into stack, and exchange AF with A'F'. The 8080A has most of the same instructions.

Block Transfer and Search Instructions

This group has several useful and unique instructions. The load and increment instruction moves one byte of data from memory pointed to by HL to another memory location pointed to by DE. Both register pairs are automatically incremented and the byte counter, BC, is decremented. This instruction is extremely valuable in moving blocks of data around.

Another instruction repeats the load and increment instruction automatically until the byte counter reaches zero. Thus, in one instruction, a block of data, up to 64 K bytes in length, can be moved anywhere in memory. Each byte of data transferred requires only 8.4 μs.

In the compare and increment instruction, the contents of the accumulator are compared with that of memory pointed to by HL. The appropriate flag bits are set, HL is automatically incremented, and the byte counter is decremented.

The instruction compare, increment, and repeat repeats the above instruction until either a match is found or the counter reaches zero.

The 8080A has no analogy to these instructions. It would have to execute three to ten separate instructions to achieve the same result. The number of bytes would be several times larger and the execution time would be several times longer.

Arithmetic and Logical Instructions

These instructions include all the adds and subtracts, increments, compares, exclusive-ors, etc. What the Z80 has added to the 8080A instructions is the indexed addressing mode and double precision add with carry and subtract with carry.

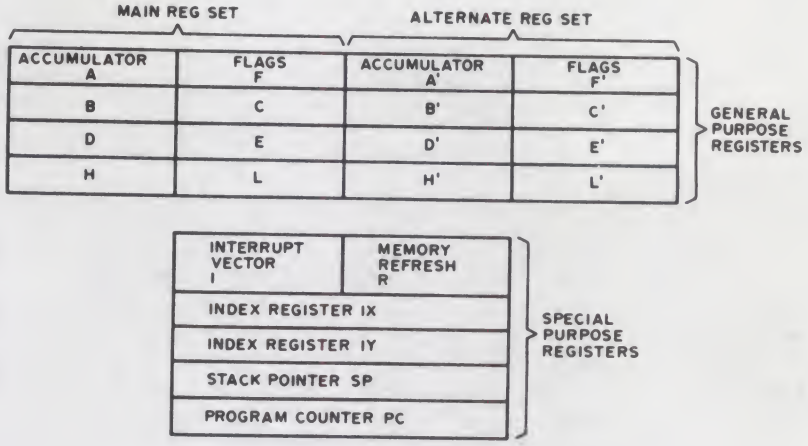


Figure 2: Programmable registers of the Z80. Considerable improvement over the 8080 design is found in the alternate register set, and the addition of two index registers, interrupt vector and memory refresh registers.

Rotate and Shift Instructions

Here the Z80 has taken the four 8080A rotate accumulator instructions and increased the possible addressing modes as well as included logical shifts and arithmetic shifts. On top of this there are a couple of rotate digit instructions. With these a digit (4 bits) can be rotated with two digits in a memory location, which is great for BCD arithmetic.

Bit Manipulation Instructions

There are three basic operations, test bit, set bit, and reset bit. With the various addressing modes, a powerful group of instructions is generated. For instance, if several memory locations are used for IO devices, status bits can be individually tested and control bits individually set or reset. The 8080A (nor any other 8 bit microprocessor) has no such capability to manipulate bits.

Jump, Call, and Return

Both the 8080A and Z80 have numerous conditional and unconditional jumps, calls, and returns. In addition, the Z80 has several jump relative instructions using relative addressing. One of special interest decrements the B register, and jumps relative if B is not zero. This is especially useful in program loop control; it would take the 8080A two instructions to perform the same task.

Input/Output Instructions

The 8080A has two IO instructions, input and output to and from the accumulator. The device address is in the second byte of the instruction, which means that each

The Z80 should be a natural for string manipulation software with its pair of full 16 bit index registers and powerful multi-byte operations such as block move, memory search and block IO instructions.

In addition to expanding operations upward to the level of blocks, the Z80 refines its addressing downward to the bit level with a group of bit manipulation instructions which are quite unique.

device must have its own IO routine. One standard routine can't be used in common because each device has a different address and therefore different instruction. The Z80 has resolved this by including IO instructions that use the C register to contain the IO device address. Therefore one IO routine can be used with the device address placed in register C before entering the routine. Also instead of being restricted in inputting or outputting to and from the accumulator only, any register can be used.

If this isn't enough, the Z80 has eight block transfer IO instructions which are similar to the memory block transfer instructions. HL is the only memory pointer, C is the device pointer, and B is the byte counter. Therefore, an IO block transfer can handle up to 256 bytes. Essentially these commands are a processor implementation of direct memory access (DMA), invoked by a software sequence.

Miscellaneous Features

These instructions include no-operation, halt, enable and disable interrupts, decimal adjust accumulator, set carry, and complement carry. The Z80 can also select one of three interrupt modes.

Interrupts on the Z80

The 8080A has one input for interrupts; the Z80 has two. One is a nonmaskable interrupt (similar to the Motorola 6800 or MOS Technology 6502) which cannot be disabled by the software. The other is a maskable interrupt which can be selectively enabled or disabled by the program. The maskable interrupt is analogous to the single 8080A interrupt.

The Z80 simplifies the hardware required to implement a system as compared to the original 8080 design. Aside from the instruction enhancements, here is a way to get an 8080 instruction set with the ease of interfacing until now only available (in 8 bits) with processors like the 6800 and 6502.

A nonmaskable interrupt will be accepted at all times by the Z80 processor. When one occurs, the processor will execute a restart to hexadecimal location 0066. The nonmaskable interrupt is used for very important functions that must be serviced immediately, such as a power failure routine.

The Z80 has three programmable modes for processor response to a maskable interrupt. There are three instructions that will select these three modes.

Mode 0 is identical to the 8080A single interrupt response mode. The interrupting device places an instruction on the data bus, and the processor executes it. The instruction will often be a restart. This mode is also the default mode for the Z80 upon a reset.

In mode 1, the processor will respond to an interrupt by executing a restart to location 0056. The response in this mode is similar to the response to a nonmaskable interrupt except for the restart location.

In mode 2, a table of 16 bit starting addresses for every interrupt routine must be maintained. This table can be anywhere in memory. When an interrupt is accepted, a 16 bit address is formed from the contents of the 8 bit I register and the 8 bits on the data bus. The I register contains the upper 8 bits of the address and the 8 bit data on the data bus from the peripheral device constitutes the lower 8 bits of the address. This 16 bit address points to a location in the interrupt vector table. The processor fetches the 16 bit address found at the selected table location (in two bytes) and loads the program counter with its value. This whole process takes 19 clock periods, or just 7.6 μs.

The peripheral devices in the Z80 microcomputer family all have daisy chain interrupt structures. They automatically supply a programmed vector to the processor during interrupt acknowledge. Only the highest priority device interrupting the processor sees the interrupt acknowledge because of the daisy chain structure. With these devices, IO interfacing becomes quite a simple task, and is as powerful as the IO techniques used in many minicomputers.

Conclusion

What does the Z80 have going for it? It's easy to interface; one chip does the job of several 8080A family chips. It's as easy, if not easier, to design an entire system around than any other microprocessor on the market today, and the Z80 is software compatible with the 8080A, the most widely used and known 8 bit microprocessor. Its instruction set is much more powerful than the 8080A's or any other 8 bit microprocessor's instruction set.

Is there anything negative about the Z80? As of this writing (March), it is not yet in production and therefore not readily available to the personal computing experimenter. The price tag for unit samples is $200, but there are numerous price breaks with larger quantities. For instance, the price is $80 for quantities of 25 – 99. This is still more expensive, however, than either the 8080A, 6800 or 6502, and is about the same as 16 bit microprocessors.

The result is a tradeoff of cost versus performance. Much of the cost difference relative to other 8 bit processors is made up by the Z80's better memory utilization and (with respect to the 8080A) by the fact that fewer parts are needed to get a minimum system going. Although the Z80 processor is priced higher than the 8080A, when the cost of all the support devices the 8080A requires are included, the costs are comparable.■

For more information on the Z80 CPU and other Z80 parts contact Zilog Inc, 170 State St, Ste 260A, Los Altos CA 94022, (415) 941-5055.■

A New Mini-Microcomputer System

The Digital Equipment Corporation LSI-11

Robert W. Baker
34 White Pine Dr.
Littleton MA 01460

Digital Equipment Corporation has a new addition to the microcomputer market. Designated the LSI-11, it is a complete 16 bit microcomputer system on a single 8.5 inch by 10 inch (21.6 cm by 25.4 cm) printed circuit board, combining the instruction set of a PDP-11/40 with an under $1000 price.

A 3.5 inch H by 19 inch W by 13.5 inch D (8.9 cm by 48.3 cm by 34.3 cm) boxed version of the LSI-11 is designed as an off-the-shelf microcomputer system. Designated the PDP-11/03, it consists of an LSI-11 microcomputer, serial line interface, power supply, and a mounting box designed to mount in a standard 19 inch cabinet. Removing the front panel exposes the LSI modules and cables allowing replacement or installation of a module from the front of the PDP-11/03. The power supply has three front panel switches and indicators accessible through a cutout in the front panel. The lights and switches are still attached to the power supply and functional when the front panel is removed. Input power of the PDP-11/03 is typically 190 Watts at full load.

LSI-11 Evolution

The processor, memory, device interfaces, backplane and interconnecting hardware of the LSI-11 are all modular in design to allow custom tailoring necessary for specific application requirements. It was not intended to be a low end minicomputer, but to provide minicomputer capability to the new microcomputer applications.

To accomplish this goal, the LSI-11 was designed to optimize system costs rather than component costs. A four-chip microprogrammed central processor was selected to emulate the PDP-11 instruction set, allowing the inclusion of automatic dynamic memory refresh without additional cost. The microprogrammed processor also makes feasible user microcode and an ASCII console which will be discussed later.

Central Processor

The central processor module consists of the microprogrammed processor and 4096 words of memory, together with the bus transceivers and control logic. The four chip microcomputer controls the time allocation of the LSI-11 bus for peripherals and performs all arithmetic and logic operations as well as instruction decoding. Eight 16 bit, general-pupose registers can be used as accumulators, address pointers, index registers, stack pointers, or other desired functions. Arithmetic operations can be from one register to another, from one memory location or device register to another, or between a memory location or a device register and a general register. Data transfers between IO devices and memory on the bus occur without disturbing the processor registers.

Bus

The bus, which is implemented on the H9270 card guide backplane assembly, is the data path which enables a complete system to be configured. This bus was designed to allow low cost peripheral interfaces for microcomputer applications, rather than to support the wide range of peripheral configurations common to large minicomputer systems. The processor module is capable of driving six device slots along the bus without additional termination, as provided with the H9720 backplane. Devices or memory can be installed in any location along the bus, as most bus control and data signals are bidirectional, open-collector lines that are asserted when low. The bus signals include 16 multiplexed data/address lines, 6 data transfer control lines, 6 system control lines, and 5 interrupt and direct memory access (DMA) control lines.

Any communication between two devices on the bus is in the form of a master-slave relationship. Only one device, the bus master, can have control of the bus at any point in time. The master device controls the bus while communicating with another device on the bus, the slave. Since the LSI-11 bus is used by the processor and all IO devices, there is a priority structure to determine which device gets control of the bus. Every device on the bus capable of becoming bus master has a specific priority associated with its position along the bus. When two devices request use of the bus simultaneously, the higher priority device will receive control. All data transfers on the bus are interlocked so that communication is independent of the physical length of the bus and the response time of the slave so long as a bus timeout does not occur. Asynchronous operation allows each device to operate at the maximum possible speed.

Interrupt System

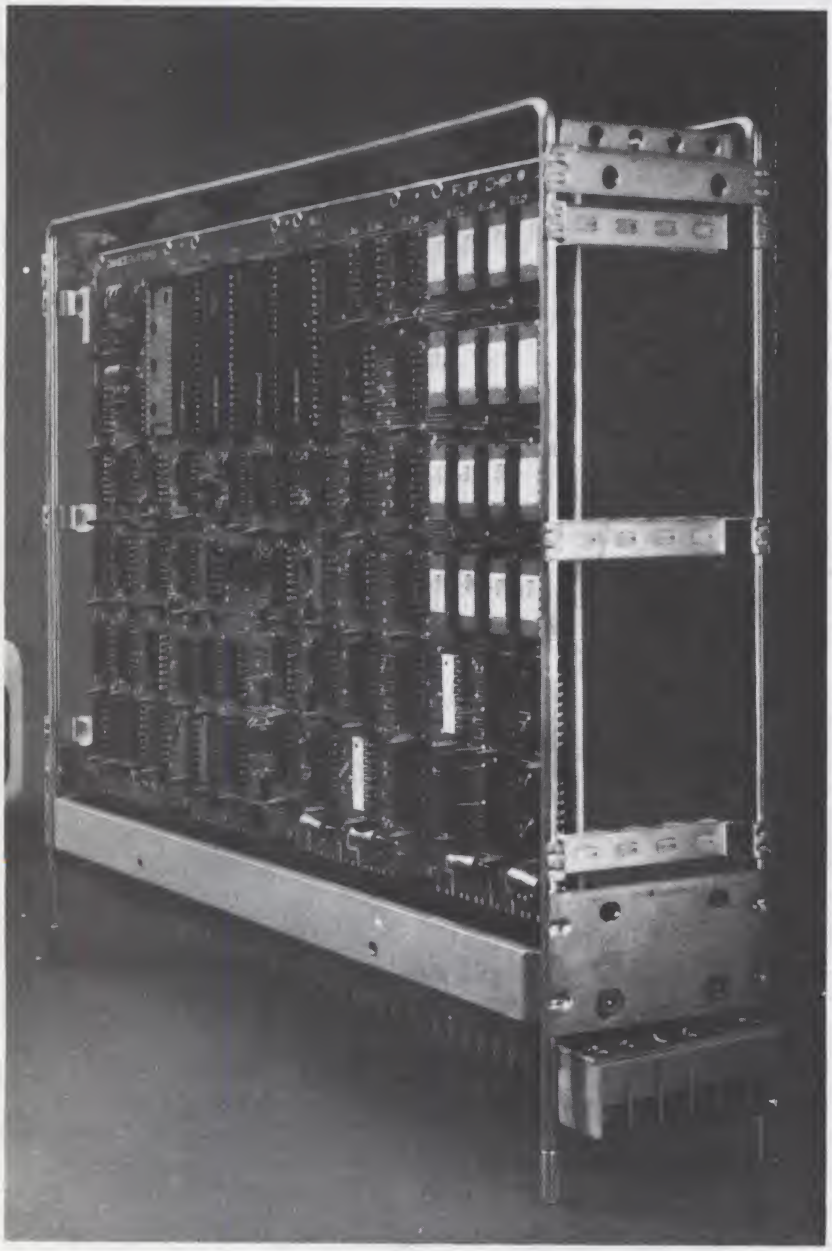
Interrupt and DMA handling incorporates two daisy-chained grant signals. This method eliminates device polling to service interrupt requests and establishes an interrupt priority. The highest priority device is the module located electrically closest to the microcomputer module. Only when a device is not asserting a request does it pass grant signals to lower priority devices. When an interrupting device receives a grant, the device passes to the processor an interrupt vector which points to a new processor status word (PSW) and the starting address of an interrupt service routine for the device. The current value of the PSW and program counter (PC) are stored on the stack.

The processor operates with the interrupt mask (PSW bit 7) set (1) or cleared (0). When PSW bit 7 is equal to 1, no external device can interrupt the processor with a request for service. The processor must be operating at PSW bit 7 equal to 0 for the device's request to be effective. Interrupts can occur only between processor instructions since they change the state of the processor. DMA operations, on the other hand, may occur between individual bus cycles since these operations do not change the processor state.

One signal line on the bus functions as an external event interrupt line to the processor module. When connected to a 60 Hertz line frequency source, this signal line can be used as a real-time clock interrupt. When automatic interrupt dispatch (vectoring) is not needed, this line may be used as a common interrupt signal. Although this necessitates device polling (as in earlier computers, such as the PDP-8), device interfaces may now be slightly less complicated. A single connection on the processor module enables or inhibits the external event interrupt. When enabled, the device connected to this line has a higher interrupt priority than any device connected to the daisy-chained grant signals.

Power Fail/Restart

To further increase the system flexibility, several power fail/restart options are avail-



able. The power fail sequence is initiated upon sensing a warning signal from the power supply signaling an impending AC power loss. The current PSW and PC are pushed on the processor stack and a new PC and PSW are taken from a vector at location 24. Normally, with non-volatile memory, this routine would save processor registers, set up a restart routine, and halt. When only volatile memory is used, the registers cannot be saved but the power fail trap does allow an orderly system shutdown to occur.

When AC power is restored, one of the four jumper selectable power-up options is initiated. The first option is loading a programmed PSW and PC from the vector at location 24. This would be used with non-volatile memory to continue execution of the program at the point where the power fail occurred or to restart the program at an arbitrary address with ROM program storage. If the BHALT line on the bus (the halt switch) is asserted during this power-up sequence, the ASCII console microcode will be entered immediately after loading the PSW and PC. The second power-up option causes an unconditional entry to the ASCII console routines. The processor can then be started by an ASCII console command allowing remote system starting without controlling the bus halt line. (More on the ASCII console later.) Alternately, the last two options allow program execution to begin at a specified address in either macrocode or microcode.

Memory

The 4096 word memory on the basic CPU module consists of sixteen 4096 bit dynamic RAMs. This memory logically appears on the external bus while being physically on the CPU module. Being accessible to the bus allows external DMA transfers to take place to and from the basic 4096 word memory. Also, an optional jumper allows the CPU module memory to occupy either the first or second 4096 word block of the bus address space.

Various memory modules are available for applications requiring more storage than the standard 4096 word MOS memory on the processor board. Those offered include a non-volatile 4096 word core memory, a 1024 word static RAM, read-only memory (PROM/ROM) with a maximum capacity of 4096 words per board in 512 word increments or 2048 words in 256 word increments, and a 4096 word dynamic MOS RAM.

A common disadvantage of using dynamic MOS memory is the necessity of refreshing the contents of memory at specific intervals. The refresh operation is required to replace the stored charge in each

memory cell which has been lost through leakage currents. To eliminate most of the control circuitry normally necessary to perform this memory refresh, the LSI-11 CPU microcode features automatic refresh control.

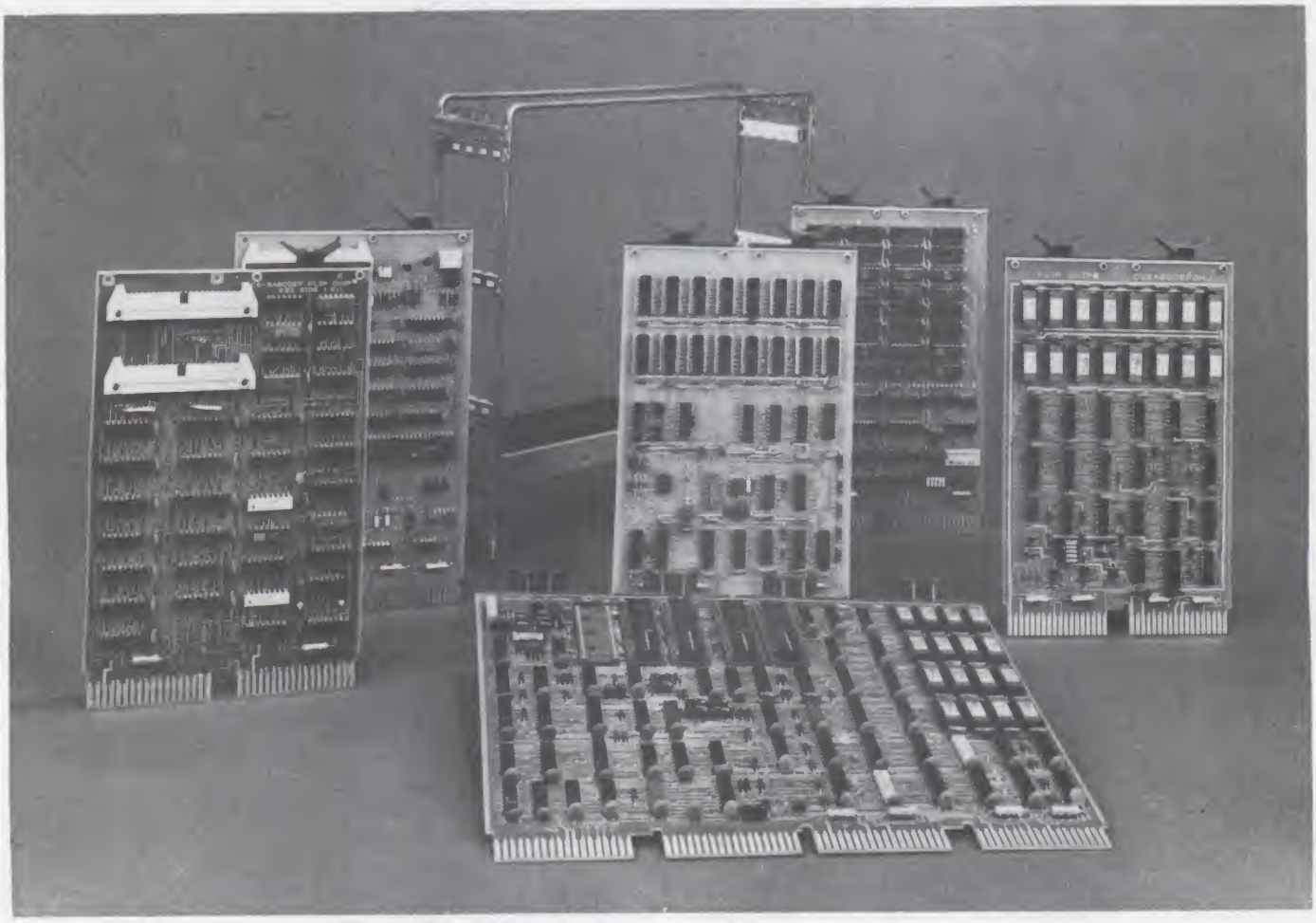
When enabled by an optional jumper, the CPU refresh control causes execution of a microcode subroutine approximately every 1.6 milliseconds; this operation refreshes all dynamic MOS memory in the system, not just the memory contained on the CPU module. While asserting a bus signal causing all dynamic memories to cycle at the same time, the CPU performs 64 memory references to refresh their contents. During the burst refresh time, external interrupts are locked out while DMA requests are still possible.

Maximum memory size of the 16 bit LSI-11 is 65,536 bytes or 32,768 words. Usually the top 4096 words of memory on members of the PDP-11 family are reserved for peripheral device control and data buffers, so the nominal maximum main memory size is 28,672 sixteen bit words. However, the user is not required to dedicate the entire upper 4096 word space to IO, but may implement only what is needed. Octal addresses 000 to 376 are usually reserved for trap and device interrupt vector locations. Several of these are reserved in particular for software generated interrupts (TRAPS) as shown in Appendix A.

Instruction Set

All operations are accomplished with one set of instructions rather than the conventional collection of memory reference instructions, operate/accumulator control instructions, and IO instructions. Single and double operand address instructions for words or bytes are used with a wide range of addressing modes, providing efficiency and flexibility in programming. The various addressing modes include sequential forward or reverse addressing, 8-bit byte addressing, 16-bit word addressing, and stack addressing. Using variable-length instruction formatting allows a minimum number of words to be used for each addressing mode.

Each processor instruction requires one or more bus cycles. The first operation fetches an instruction from the location specified by the program counter (PC). If no further operands are required for executing the instruction, no further bus cycles are used. If memory or an IO device is



referenced, however, one or more additional bus cycles are required.

A special maintenance instruction is included in the LSI-11 instruction set to aid in hardware checkout. This instruction stores the contents of five internal registers in a specified block in main memory. A diagnostic program may then be used to examine the information and determine the internal operation of the micro-level processor.

The basic instruction set is that of the familiar DEC PDP-11/40 without memory mapping. Included are several operations normally not found even in other small PDP-11 processors, such as exclusive-or (XOR), sign extend (SXT), or subtract one and branch (SOB). There are also two new instructions used to explicitly access the processor status word (PSW). With the optional extended arithmetic chip, full integer multiply/divide and floating point arithmetic are also available. The instruction set is more comprehensive than that of the PDP-11/05 while the execution times are a little slower. Refer to Appendix B for a complete list of the LSI-11 instruction set and Appendix C for typical timings.

The branch instructions make use of the condition codes (PSW bits 0 to 3) which are set after execution of every arithmetic or logical instruction. This allows more efficient use of memory by eliminating extra instructions and temporary storage locations typically used to check results of various operations. The result of every operation is directly accessible and can be modified under software control by using any of the Condition Code Operator instructions. A list of the four condition codes along with a brief definition of each is listed in Appendix D.

Software

Since the LSI-11 uses standard PDP-11 software, there is an extensive library of programs available from DEC including diagnostic programs to check out your system after it is built. There is also a DEC Users Society (DECUS) which makes available a complete library of various PDP-11 programs at reasonable prices. Every LSI-11 owner automatically becomes a member of this organization.

ASCII Console

The conventional front panel lights and switches are replaced by an ASCII console/ODT package that operates with any standard terminal device communicating through a serial interface at a specific device address at any available baud rate. The functions available are very similar to those used by the familiar PDP-11 Octal Debugging Technique and are shown in detail in Appendix E. These include examining and changing the contents of memory and registers, calculation of effective addresses for relative and indirect addressing, and the functions of halt, single-step, continue and restart. By examining the contents of an internal CPU register, it is possible to determine which of the five methods of entering the console routines was used.

Upon entering the console routine, the location of the next instruction to be executed will be printed followed by @. The console routine will then wait for one of the 14 legal command characters. Thus, the user retains all the direct hardware control of a conventional lights and switches front panel and gains the ability to boot load from a specified device in byte transfer mode.

Interfaces

The LSI-11 system includes several standard interface modules to handle a variety of applications. Currently both a serial and a parallel IO interface is available, each as a single 8.5 inch by 5 inch (21.6 cm by 12.7 cm) PC board. The DLV-11 handles a single asynchronous serial line between 50 and 9600 baud, while the DRV-11 provides a full 16-bit parallel interface complete with two interrupt control units. The use of the two standard interface modules makes it very simple to connect any desired device to the LSI-11 bus. Standard devices such as teletypes, line printers, analog to digital converters, etc., can be connected directly to the interface modules with no additional circuitry. A simple cassette recorder interface can be made using the DRV-11 parallel interface, a UART chip, and a simple speed independent recorder interface circuit such as that shown in Don Lancaster's article *Serial Interface*, page 30, in the September issue of BYTE. ■

Are you interested in buying one?

This article has described the details of the LSI-11 computer by Digital Equipment Corporation. For those interested in purchasing the board version of this computer, the Southern California Computer Society is organizing a group purchase for amateurs. This purchase will involve an original equipment manufacturer (OEM) quantity of 50 or more machines, on a basis of cost plus 2% minimum contribution to SCCS. For further information contact Hal Lashlee of The Southern California Computer Society, at 213-682-3108. SCCS is organizing quantity purchases of other computer equipment, and is interested in making such offerings available through other computer clubs.

APPENDIX A: TRAP VECTORS

Location	Vector
ØØØ	(Reserved)
ØØ4	Time out & other errors
Ø1Ø	Illegal & reserved instructions
Ø14	BPT instructions
Ø2Ø	IOT instructions
Ø24	Power Fail
Ø3Ø	EMT instructions
Ø34	TRAP instructions
Ø6Ø	Console Input Device
Ø64	Console Output Device
1ØØ	External event line interrupt
244	FIS option

APPENDIX B: LSI-11 INSTRUCTION SET

MNEMONIC	INSTRUCTION
Single Operand - General:	
CLR	Clear word
CLRB	Clear byte
COM(B)	Complement (1's)
INC(B)	Increment
DEC(B)	Decrement
NEG(B)	Negate (2's complement)
TST(B)	Test
Rotate & Shift:	
ROR(B)	Rotate right
ROL(B)	Rotate left
ASR(B)	Arithmetic shift right
ASL(B)	Arithmetic shift left
SWAB	Swap bytes
Multiple Precision:	
ADC(B)	Add carry
SBC(B)	Subtract carry
SXT	Sign extend
Processor Status (PSW) Operators:	
MFPS	Move byte from PSW
MTPS	Move byte to PSW
Double Operand - General:	
MOV(B)	Move
CMP(B)	Compare
ADD	Add
SUB	Subtract
Logical:	
BIT(B)	Bit test (logical AND)
BIC(B)	Bit clear
BIS(B)	Bit set (logical OR)
XOR	Exclusive OR
Branches:	
BR	Unconditional branch
BNE	Branch if not equal to Ø
BEQ	Branch if equal to Ø
BPL	Branch if plus
BMI	Branch if minus
BVC	Branch if overflow is clear
BVS	Branch if overflow is set
BCC	Branch if carry is clear
BCS	Branch if carry is set
Signed Conditional Branches:	
BGE	Branch if greater or equal to Ø
BLT	Branch if less than Ø
BGT	Branch if greater than Ø
BLE	Branch if less or equal to Ø
Unsigned Conditional Branches:	
BHI	Branch if higher
BLOS	Branch if lower or same
BHIS	Branch if higher or same
BLO	Branch if lower
Condition Code Operators:	
CLC	Clear C Condition Code Bit
CLV	Clear V
CLZ	Clear Z
CLN	Clear N
CCC	Clear all condition code bits
SEC	Set C Condition Code Bit
SEV	Set V
SEZ	Set Z
SEN	Set N
SCC	Set all condition code bits
Jump & Subroutines:	
JMP	Jump
JSR	Jump to subroutine
RTS	Return from subroutine
MARK	Mark (aid in subroutine return)
SOB	Subtract 1 & branch if not Ø
Trap & Interrupts:	
EMT	Emulator trap
TRAP	Trap
BPT	Breakpoint trap
IOT	Input/Output trap
RTI	Return from interrupt
RTT	Return from interrupt
Miscellaneous Instructions:	
HALT	Halt
WAIT	Wait for interrupt
RESET	Reset external bus
NOP	(no operation)
Optional EIS:	
MUL	Multiply
DIV	Divide
ASH	Shift arithmetically
ASHC	Arithmetic shift combined
Optional FIS:	
FADD	Floating add
FSUB	Floating subtract
FMUL	Floating multiply
FDIV	Floating divide

Continued from page 19

APPENDIX C:
TYPICAL INSTRUCTION TIMING

INSTRUCTION	TIME (usec)	COMMENTS
ADD R1,R2	3.5	Register addressing
MOV R3,RØ	3.5	
MOV TAG1,1Ø (R2)	11.55	Relative & index addressing
TSTB (R3)+	5.25	Auto-indexed
BMI TAG2	3.5	Conditional branch
JSR PC,2(R2)	8.Ø5	Subroutine call
JMP (R4)	4.2	Jump indirect
RTI	1Ø.5	Return from interrupt
Optional EIS & FIS Instructions:		
MUL	24 - 64	Multiply
FADD	42.1	Floating add
FMUL	52.2 - 93.7	Floating mult
FDIV	151 - 232	Floating divide

APPENDIX D:
PSW CONDITION CODES

CODE	PSW BIT	CONDITION WHEN SET = 1
N	3	If result were negative
Z	2	If result were zero
V	1	If operation resulted in an arithmetic overflow
C	Ø	If operation resulted in a carry from the msb (most significant bit) or a 1 was shifted from the lsb (least significant bit)

APPENDIX E:
ASCII CONSOLE/ODT COMMANDS

Command	Function
< CR >	Close opened location and accept next command.
< LF >	Close current location; open next sequential location.
Up-arrow	Open previous location.
Back-arrow	Take contents of opened location as a relative address, and open that location.
@	Take contents of opened location as absolute address and open that location.
r/	Open word at location r.
/	Reopen the last location.
$n/ or Rn/	Open general register n(Ø-7) or S (PSW).
r;G or rG	Go to location r and start program
nL	Execute bootstrap loader using n as device CSR. Console device is 17756Ø.
;P or P	Proceed with program execution.
RUBOUT <DEL>	Erases previous numeric character. Response is a backslash (\).

Continued from page 22

BUS — A collection of parallel data paths and power lines used to interconnect the various elements of the system, including the central processor, memory, and all peripherals.

BUS TIMEOUT — Bus timeouts or bus errors occur whenever the controlling device on the bus (the bus Master) does not receive a response from the addressed device (the Slave) within a certain length of time. In general, these are caused by attempts to reference non-existent memory or peripheral devices. Bus error traps cause processor traps through the trap vector address 4.

DMA — Direct Memory Access. For high speed devices, memory may be accessed directly through the bus without the use of program controlled data transfers.

JUMPER SELECTABLE OPTIONS — The CPU module contains locations for six wire jumpers to control the various operating options as follows:

1. Two wires select which of the four possible power-up options is desired. These are normally set to restart through vector location 24 (so the LSI-11 acts as a standard PDP-11).

2. One wire jumper enables the external event (or real-time clock) interrupt feature when inserted.

3. One wire jumper enables the automatic dynamic memory refresh feature when inserted.

4. Two wire jumpers determine the addressing of the 4K RAM memory located physically on the CPU module.

Each wire jumper consists of a short length of bare copper wire soldered between two designated holes in the PC board.

MACROCODE — The instruction set which the programmer sees and actually uses to implement his program, such as the PDP-11 instruction set in this case.

MICROCODE — The low level instruction set used in a microprogrammed processor to "emulate," or execute, the macrocode. Microcode is more primitive in function, but executes at a higher speed than the macrocode.

PC — Program Counter. A register which contains the address of the next instruction to be executed.

POWER-UP SEQUENCE — Two wire jumpers on the CPU module select one of the four possible power-up modes:

OPTION	POWER-UP
0	PC at 24, PSW at 26, or HALT
1	ODT — ASCII Console
2	PC = 173000, or HALT
3	Special processor microcode

The power-up sequence is initiated upon supplying power to the processor module or on restoration of power after a temporary power fail has occurred.

PSW — Processor Status Word. Contains information on the current status of the processor including the priority mask (bit 7) and the condition codes (bits 0 to 3).

STACK — An area of memory set aside by the programmer for temporary storage or subroutine/interrupt sevice linkage. The stack uses the "Last In — First Out" concept; thus various items may be added to a stack in sequential order and retrieved or deleted from the stack in reverse order. Stack starts at the highest location reserved for it and expands linearly downward. In the LSI-11, register 6 is reserved as the hardware stack pointer and must be initialized by the software. However, registers 0 to 5 may be used for various program defined stacks as needed.

TRAP — Software generated interrupt.

VECTOR — A unique address which points to a reserved set of locations (2 words) for interrupt or error handling. The first word contains the starting address of a service routine (a new PC) while the second holds the new PSW to be used by the service routine.

VOLATILE MEMORY — Volatile memory, such as RAM, will not retain useful information without power applied continuously. Non-volatile memory, such as core or ROM, will always retain its information with or without power applied.

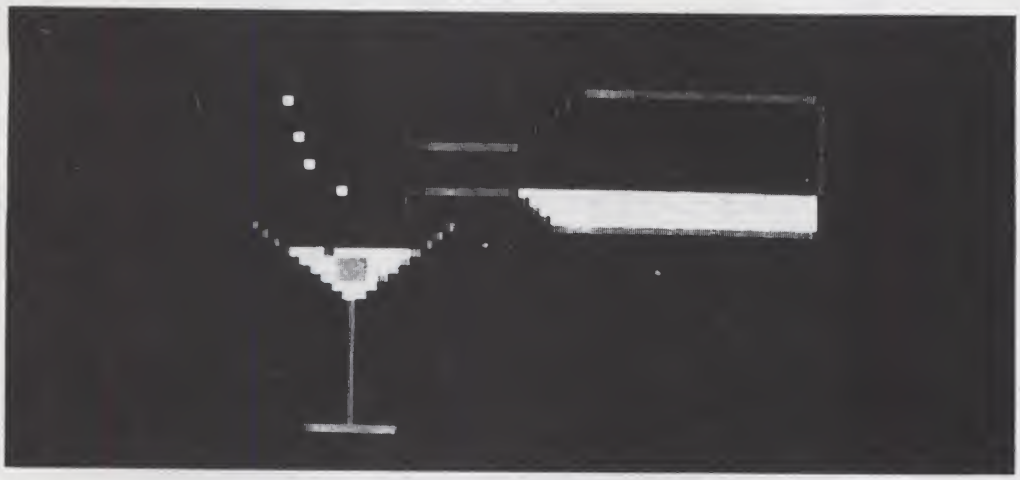


Photo 1: Here is a black and white reproduction of a single frame of a wine pouring animation sequence created by Steve Dompier using his Dazzlemation program. The colors of the original reproduce as shades of grayness in this black and white picture.

Cromemco TV Dazzler

[This short account is based upon materials supplied by Harry Garland of Cromemco. . . . CH]

Imagine being able to look inside your computer memory, actually being able to see the individual bits. With this sort of X ray vision your computer memory could also serve as your computer display. Messages could be spelled out by lighting some bits and darkening others. Games could be played with clusters of bits forming game pieces and markers. Space War might be played with miniature rocket ship patterns zooming in, out and around the visible region of memory address space. The key element of hardware required to actually achieve this imagined result is a memory module which has provisions to map its contents onto a television screen. This is precisely what Cromemco has done in creating its TV Dazzler product, the results of which were used to create this month's cover.

The TV Dazzler hardware features two modes of operation providing high resolution and low resolution generation of a television picture. Through software selection the TV Dazzler can be programmed either as a 128 x 128 point black and white display, or as a 64 x 64 point colored display. The points of the display grid are tiny square regions on the screen which map into segments of the 2 K byte memory of the TV Dazzler module.

In the high resolution "bit mapped" mode, TV Dazzler uses its 2 K byte memory as a means of storing 2^{14} = 16,382 bits required to generate a unique "on" or "off" value for each location of a 128 x 128 grid. This high resolution black and white mode is very effective for alphanumeric displays and detailed computer controlled images.

In the low resolution "nybble mapped" mode, TV Dazzler uses its 2 K byte memory as a means of storing 2^{12} = 4096 four bit nybbles of data needed to generate a color display on a 64 x 64 grid. Each nybble determines the color and intensity of the corresponding picture element on the grid. The most significant bit sets either high or low intensity, and the next three bits independently select the blue, green and red channels of the color TV signal.

Like a metaphorical beachball, (see January 1976 BYTE editorial), the Dazzler provides the hardware for an incredible variety of applications. This variety is realized through the software for games and other purposes developed by people who buy and use this type of peripheral. One particular application of the peripheral is a program called Dazzlemation which was written by Steve Dompier. The purpose of Dazzlemation is to record an animated sequence of TV frames in color, then play these back. In order to make such a sequence, Dazzlemation is used to color in the appropriate regions of single frames which are stored in memory. Steve's standard demonstration sequence shows a carafe of red wine being poured into a wine glass. One frame of the carafe sequence is illustrated by photo 1. This is just one of an endless variety of computer generated animated displays which is made possible by programs like Dazzlemation.

A second application of the Dazzlemation hardware was used to generate the pattern which forms the main portion of the cover. This is a program called Dazzler-LIFE which was written by Ed Hall. John Conway's

fascinating game of LIFE gains a new dimension when it is displayed in color. Watching the patterns evolve can be intoxicating in black and white, but becomes truly addictive when color is used to illustrate the game board. In the Dazzler-LIFE program, the game begins in a drawing mode which allows the user to draw an initial colony of cells on the screen using controls from the ASCII keyboard. Then the evolution process is initiated with each succeeding generation being displayed on the screen with colors marking the health of each cell. Cells that are too crowded, or too remote, turn a flaming red color, then wither away. Newborn cells first appear in green, then grow up to a mature blue color. The kaleidoscopic result is fascinating to watch. One frame of a colorful LIFE history was photographed for the cover.

Still another application of the Dazzler is as a hardware game board for sophisticated computer automated games. One example of such an application is the Tic Tac Toe software written by George Tate. Dazzler Tic Tac Toe is written in BASIC, and demonstrates how very well suited the MITS BASIC is for creating colorful creations. George's program is one of a class of "man versus computer" game applications, and is reputed to be extremely competent at Tic Tac Toe. A sample of the output is reproduced here in black and white as photo 2.

A useful utility program for the Dazzler, which demonstrates the bit mapped mode of operation is the Dazzlewriter software created by Ed Hall. This program turns your ASCII keyboard/computer/Dazzler combination into a TV typewriter by generating the 5 x 7 dot matrix display for each keyboard character. A sample of Dazzlewriter activity is shown in photo 3. Since the main memory of the computer is used to store the character generation information, there is no need for any additional hardware beyond the memory requirements of Dazzlewriter.

Another delightful application of the display is an "idling" program you'll probably want to leave in the computer system when you're not using it for another purpose. This program is Li-Chen Wang's colorful Kaleidoscope program. The program is surprisingly short, just 127 bytes long, yet it generates an unending sequence of captivating patterns.

These programs were created by some of the first individuals who had access to the Dazzler hardware. They are written for the 8080 instruction set (except George Tate's BASIC Tic Tac Toe) and are available in paper tape form from Cromemco at $15 each.■

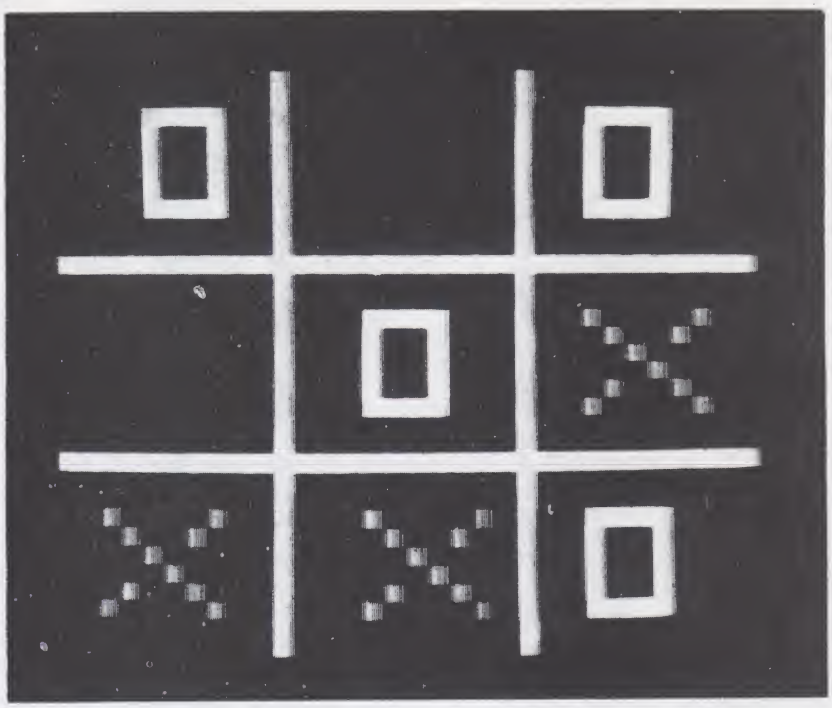


Photo 2: Here is the game board of George Tate's Tic Tac Toe application, written in MITS Altair BASIC with the TV Dazzler as its display peripheral.

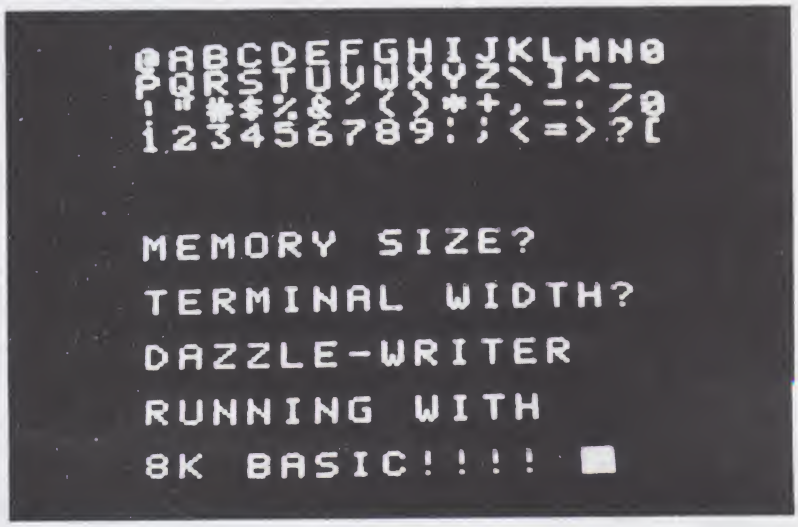


Photo 3: Here is a sampling of outputs generated using Ed Hall's Dazzlewriter program to turn the TV Dazzler/computer/keyboard combination into the logical equivalent of a TV typewriter style display.

Hardware

Flip Flops Exposed

by
William E. Browning
516 N. 95th E. Ave.
Tulsa OK 74155

One of the important building blocks in working with transistor-transistor logic is the flip flop. It is important to understand this building block if you desire to use it in projects of your own.

The two most common types of flip flops are known as the JK flip flop and the D flip flop.

JK Flip Flop

The JK flip flop has four or five inputs and one or two outputs. The input pins are labeled J, K, CLOCK, CLEAR and PRESET, and the output pins are labeled Q and $\overline{Q}$ which is often pronounced as "Q bar" or "not Q". A typical block diagram of a JK flip flop is shown in Fig. 1.

The outputs (Q and $\overline{Q}$) can be in one of two states: High (logic 1) or low (logic 0). In general, if the Q output is high then the $\overline{Q}$ output is low, and vice-versa, if the Q output is low then the $\overline{Q}$ output is high. We will often refer to the Q output only since we know that $\overline{Q}$ will be the opposite. So if we say that the output is high it means that Q is high and $\overline{Q}$ is low. However, this is not always so; on some flip flops you may find a Q output only, and, as you will see further below, both outputs may be high or low under specific conditions.

Asynchronous Inputs

Now that we know about the output states, let's discuss the inputs to give us the desired outputs. The PRESET and CLEAR pins are known as asynchronous inputs. Asynchronous means that these inputs do not depend upon the timing derived from the clock pulses.

With almost all flip flops a low PRESET will take Q to high, and a low CLEAR will take the $\overline{Q}$ to high. With both PRESET and CLEAR low both Q and $\overline{Q}$ will be high. This is the only time when Q and $\overline{Q}$ are not opposite from each other. If both PRESET and CLEAR are high, then the control of the output is given to the J, K and CLOCK inputs. The relationship between the asynchronous inputs and the output is shown in the truth table of Fig. 2.

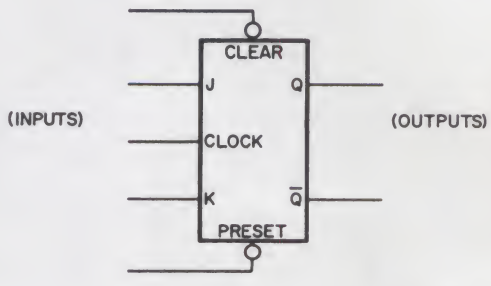


Fig. 1. The JK flip flop block diagram.

PRESET	CLEAR	Q	$\overline{Q}$
L	L	H	H
L	H	H	L
H	L	L	H
H	H	NO CHANGE	

Fig. 2. Truth table of a JK flip flop responding to preset and clear.

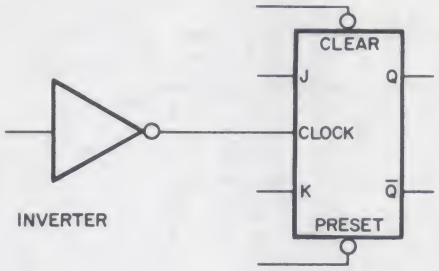


Fig. 4. Inverting the clock input converts rising edge-triggered operation into negative edge-triggered operation and vice versa.

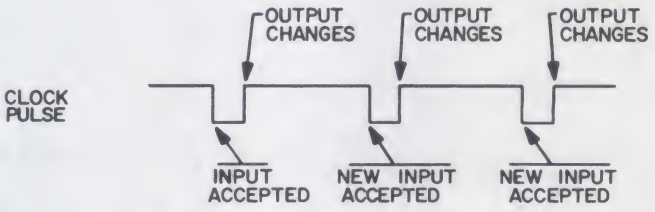


Fig. 5. Timing of a negative clock pulse master/slave flip flop.

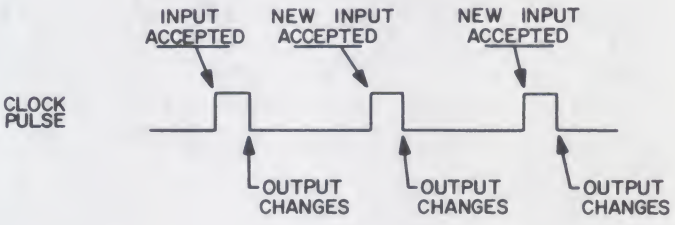


Fig. 6. Timing of a positive clock pulse master/slave flip flop.

Synchronous Inputs

The J and K inputs are synchronous inputs. Synchronous means that they depend on the CLOCK for operation.

For most JK flip flops a simple set of rules apply to the synchronous inputs, but not all flip flops are standard. There are several variations on the timing when the chip accepts inputs and when the output changes.

The most common input-output relation is known as the rising-edge triggered flip flop. The rising-edge triggered flip flop derives its name from the fact that it changes its output states only when the CLOCK level rises from low to high. Of course, the output states depend on the configuration of the J and K inputs. Fig. 3 illustrates the changes of the Q pin depending on the levels on the J and K pins: When both J and K are low, Q does not change; when both are high, Q changes into its opposite state; with J being low, and K being high, Q assumes a low level, and when J is high, but K is low, Q will take on a high level. Remember, though, that this change can happen only if the CLOCK input rises from low to high, and, as was shown in conjunction with Fig. 2, when both PRESET and CLEAR are high.

The less frequent type of input-output relation is known as the falling-edge triggered flip flop. This flip flop resembles the first, except that the output changes to the condition selected by the J and K inputs when the CLOCK level falls from high to low. Everything else remains the same.

It is easy to change the operation of a falling-edge triggered flip flop to that of a rising-edge triggered flip flop. All that needs to be done is to invert the input to the CLOCK. The inverter shown in Fig. 4 changes a high level to a low level, and vice versa; a rising-edge triggered flip flop can be changed to a falling-edge triggered flip flop, and vice versa, by means of the inverter.

t_n		$t_n + 1$
J	K	Q
L	L	Q
L	H	L
H	L	H
H	H	$\overline{Q}$

Fig. 3. Truth table of a JK flip flop for synchronous (clocked) operation.

A third type of input is known as the negative clock pulse master/slave flip flop. With this flip flop the inputs are applied to the J and K pins when CLOCK goes low and change their outputs after the rising edge of the clock pulse (see Fig. 5).

The clock pulses should be made as short as possible and the time between clock pulses as long as possible. This will increase immunity to alternating current noise and accommodate all ripple delay between clock pulses.

A fourth type of input is the positive clock pulse master/slave flip flop. This flip flop accepts inputs when CLOCK goes high and changes output when CLOCK goes low (see Fig. 6). The pulses should be made as short as possible for the above mentioned reasons.

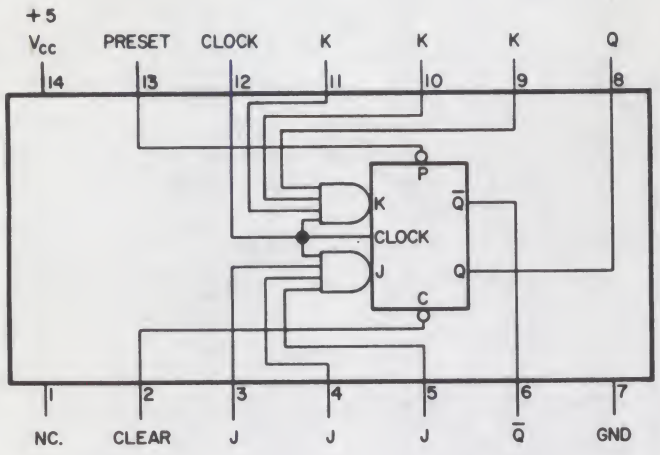


Fig. 7. Pinout of the 7472 IC, a positive clock pulse master/slave flip flop. (Top view)

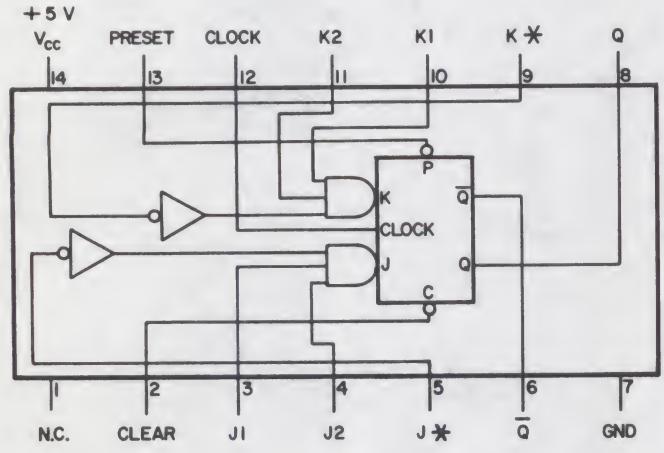


Fig. 8. Pinout of the 7470 IC, a positive edge triggered flip flop. (Top view)

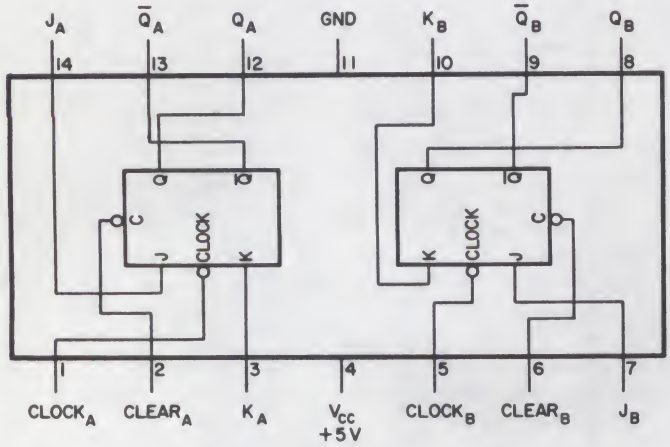


Fig. 9. Pinout of the 7473 IC, a dual positive clock pulse master/slave flip flop. (Top view)

Variations on the Four Flip Flops

One of the confusing aspects with flip flops is that they differ in the logic controlling the J and K inputs. Some have AND/OR logic circuits accepting several distinct inputs, and others have an inverter before the J or K inputs.

If we look at some diagrams of flip flop ICs, their operation will be easier to understand.

The 7472 in Fig. 7 is a single JK flip flop of the positive pulse master/slave type.

The 7472 uses gates on the J and K inputs. In order to get a high level on the J input of the flip flop a high on all three J inputs of the IC (pins 3, 4 and 5) and the clock (pin 12) is required. In a similar manner, a high on the K input of the flip flop is obtained with a high on all three K pins (pins 9, 10 and 11) and the clock of the IC.

If you have only one J input, connect all three J pins together, or connect two of the J inputs to high, and use the remaining input for the data input. The same holds true for the K inputs.

The 7470, shown in Fig. 8, is a single JK flip flop of the positive-edge triggered type.

The 7470 has identical gates and inverters on the J and K inputs. For example, to get a high level on the J input of the flip flop a high on J1 and J2 (pins 3 and 4) and a low on J* (pin 5) must be received. A low level on pin 5 is inverted to a high on the input to the gate.

If you do not need J* or K*, connect them to ground. If you do not need J1, J2, K1 or K2 connect them to high.

The 7473 IC is a dual JK flip flop of the positive pulse master/slave type. Fig. 9 shows that this IC does not use gating on the J and K inputs.

The two flip flops operate separately from each other having only the +5 volt Vcc and ground in common with each other. You may also have noticed that Vcc is on pin 4 and ground on pin 11. This is not the same as on most 7400 ICs which have Vcc on pin 14 and ground on pin 7. So check your connections before you apply power.

There are no PRESET inputs to the chip, in order to allow a 14 pin package.

If you do need the PRESET on a dual JK flip flop, use the 7476 available in a 16 pin package. It is shown in Fig. 10. It has independent CLEAR and PRESET pins, whereas in the 74H78 of Fig. 11 the CLOCK and PRESET pins, respectively, are connected. The 74H78 comes in a 14 pin package.

In many circuits the same clock operates many flip flops and several flip flops are cleared at the same time. The 74H78 is well suited for these needs because it saves on the number of external connections, saves on the size of the IC package, and the number of pins.

D Flip Flop

Let's make one small modification to the JK flip flop: An inverter connects the K input to the J input as shown in Fig. 12.

If J is high then K will be low, and if J is low, K will be high. Since there is only one synchronous input instead of the two, let's call it the data input or D input.

Some flip flops have this modification built into an IC and are referred to as D flip flops. A typical block diagram of a D flip flop is shown in Fig. 13.

The truth table of the D flip flop is shown in Fig. 14. Remember that the PRESET and CLEAR must be high, as discussed in connection with Fig. 2. The truth table for asynchronous inputs applies also to the D flip flop.

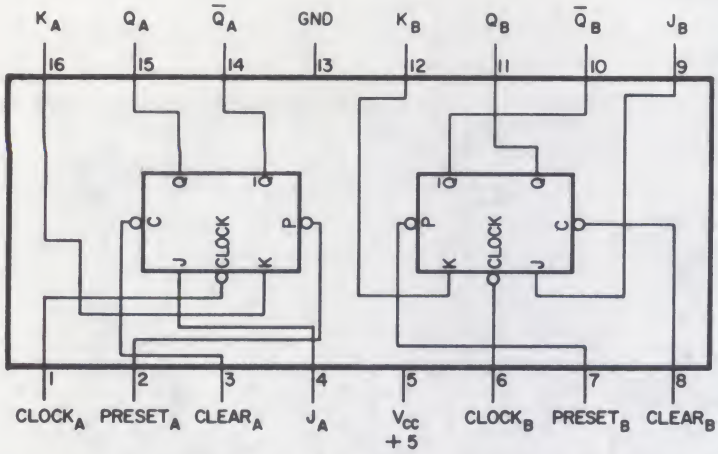


Fig. 10. Pinout of the 7476 IC, a dual flip flop similar to the 7473 but having both PRESET and CLEAR. (Top view)

Several ICs employ the D flip flop. One of these is the 7474 dual D flip flop. Since only one pin is needed for data entry to each flip flop both preset and clear capability can be provided in a 14 pin package. The diagram of the 7474 is shown in Fig. 15.

The 7475 IC uses a D flip flop which is called a latch because the CLEAR and PRESET pins are absent. The reduction in pins has been carried one step further by combining two CLOCK pins each. Therefore it is possible to put four latches on one 16 pin IC; the 7475 quad latch is shown in Fig. 16.

This short introduction to flip flops, latches and integrated circuits should help your understanding of this building block. With a thorough understanding of these circuits, you will be well on your way to designing your own equipment. ■

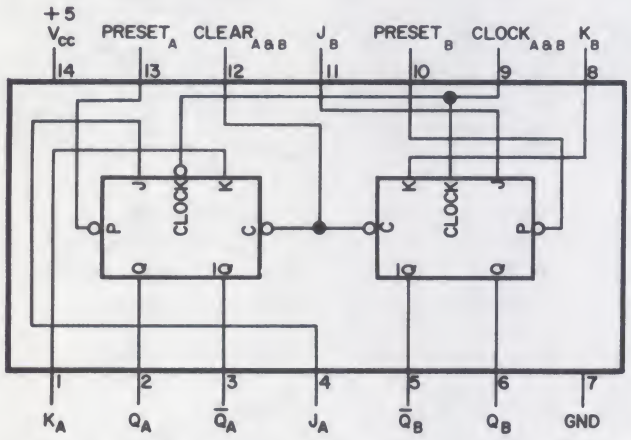


Fig. 11. And still another combination - the pinout of the 74H78 has 14 pins with common CLEAR and CLOCK, and separate PRESET pins.

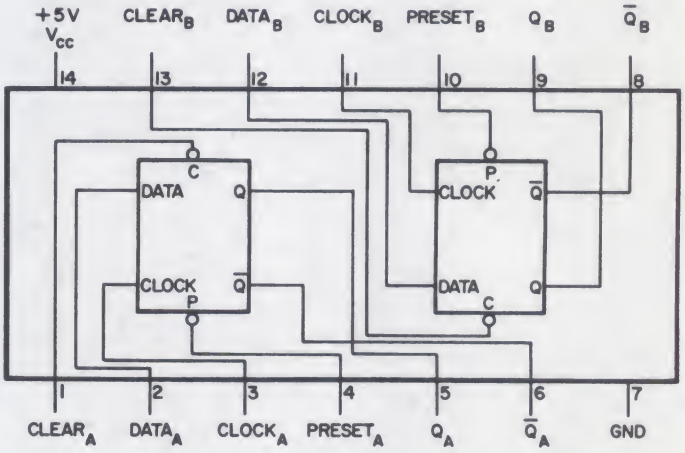


Fig. 15. The pinout of the 7474 dual D flip flop. (Top view)

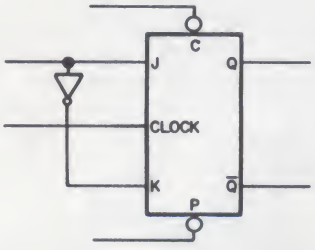


Fig. 12. Making a "D" flip flop out of a "JK" with an inverter. (Top view)

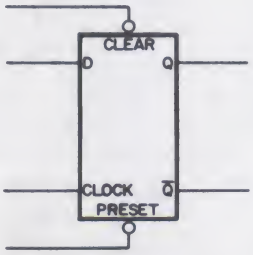


Fig. 13. The block diagram of a D flip flop.

t_n	t_n+1
D	Q $\bar{Q}$
L	L H
H	H L

Fig. 14. Truth table of a D flip flop.

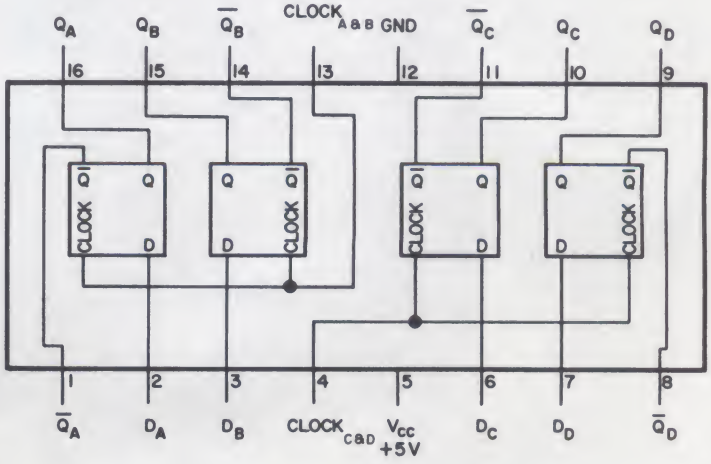


Fig. 16. Pinout of the 7475 latch (or register) circuits. (Top view)

Recycling Used ICs

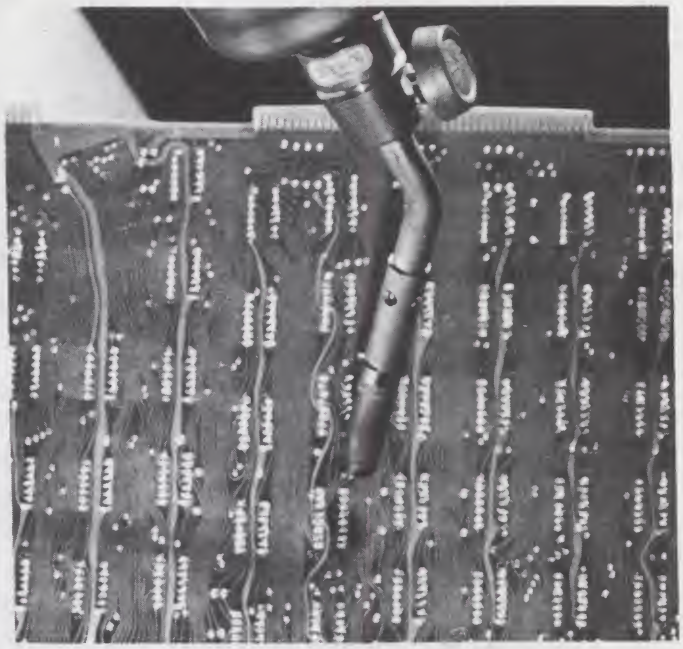
by
Carl Mikkelsen
35 Brookline St., No. 5
Cambridge MA 02139

The surplus market is saturated with used printed circuit boards from early computer systems which offer a very inexpensive per chip source of ICs. Used boards typically contain 50-200 chips of small scale or medium scale integration, usually with many simple two input gates and four bit data registers. Common part numbers include 7400, 7402, 7404, 7408, 74126, 74174, 74175, etc. Through careful shopping, I have found boards with large numbers of multiplexors such as 74151, 74153, and even scratch pad registers — 7489. After removing chips from the boards and eliminating any non-functional units, cost per chip is from 3 to 8 cents, resulting in an overall cost of about one fourth to one tenth of the individual chip cost through other surplus outlets.

Removing chips from boards offers advantages over purchasing chips surplus which makes them attractive for reasons other than price. Primarily, the companies which originally built the boards used top-quality, fully spec'ed components. All chips have already been tested, and most have already served in equipment.

Given that you've found a serendipity of well soldered chips, it's necessary to unsolder them without either burning them or cracking their cases. Desoldering individual leads can be done, but usually the chip is made unnecessarily hot by the prolonged application of heat. Also, pulling each lead out separately results in bent, often broken leads. Devices are available which will heat all 14 or 16 pins of a small IC, but again a long time is needed to melt the solder since the total amount of energy available is limited to a small soldering pencil heating element. Most available boards are two sided and four layer boards aren't uncommon. Multi-layered boards make the required amount of energy even higher.

Sweep the blow torch over the IC's pins—one complete sweep once or twice a second.



When a board is built, the ICs are positioned in place with all other components, and the board is soldered by a three step process.

1. The underside is washed by hot, bubbling, liquid flux.
2. The clean board is passed over a small fountain of solder, so that the board just touches it.
3. After cooling, the board is immersed in FREON gas to remove any remaining flux.

As you can see, the board is subjected to high temperatures during the soldering phase, which takes around 5-10 seconds.

The blow torch method of IC removal duplicates conditions during board soldering by heating all pins simultaneously; removing the IC is a single step.

Equipment Needed

To use this technique, you will need:

A torch. Non-oxygenated propane and acetylene gas has been used.

Clamps or a vise to hold the board fairly rigid during chip removal.

A way to grip the chips, depending on how they are packed next to each other. Components, small vise grips, a small screw driver and a fine point awl should be all that are needed.

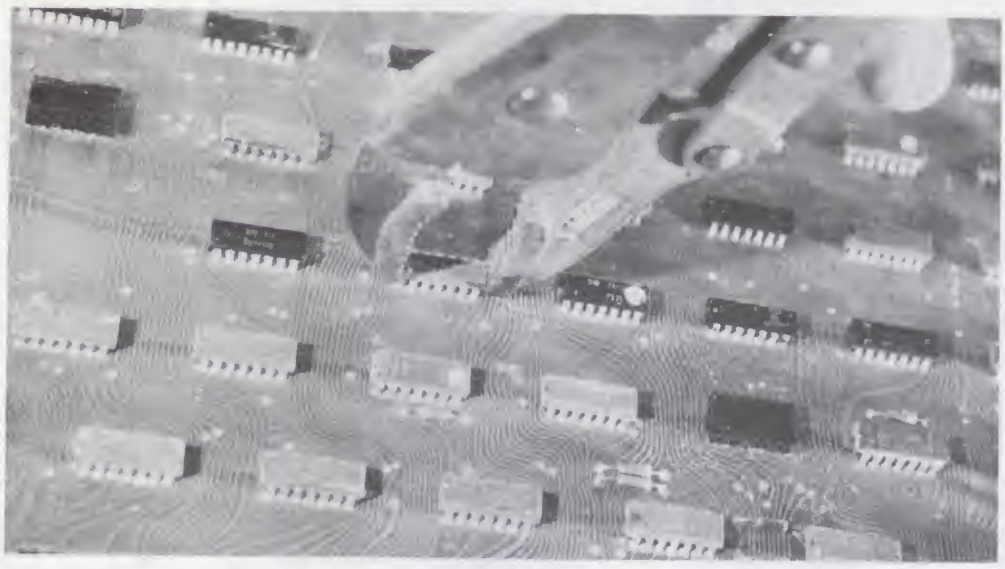
A place where splashed solder will not be serious.

Some form of eye protection.

WARNING 1

Using this method involves heating PC boards to high temperatures. Some boards release Hydrogen Chloride (HCl), which becomes hydrochloric acid in your lungs. Do this only in a well ventilated area, and stop to allow air to clear if irritation develops.

Grip the IC a second after removing the flame and rock it away from the board. It should come free in a couple of seconds.



WARNING 2

When an IC is pulled from a board, the board often snaps back to its original position. This is especially true if it isn't fixed very rigidly in place. When the board flips, solder is often sprayed away from the back side of the board. I ruined a pair of pants by not considering this before I started. I, therefore, wear old clothes and if you don't want solder on the floor, cover it with newspapers.

Enough warnings . . . following is how I pull ICs from boards:

First I clamp the board to my bench so that I can get my vise grips on about half the ICs (this is with a 10" x 14" board). I adjust the vise grips so I can grip a 14 pin IC without the vise grips locking and then light the torch. The flame on my Benzo-matic torch with the narrow tip is about an inch long.

Beginning with the lowest IC I can reach, I heat it with the torch by sweeping the torch over its pins (you obviously heat the non-component side). Especially when using a torch with a narrow flame it is necessary to move the flame over the pins. One complete sweep should be done once or twice a second. After a second or so, the IC should be gripped, and rocking tension away from the board applied. It helps to rock the IC, especially if corner pins have been bent over to hold the IC in place during assembly. The IC should very rapidly become loose, and in another couple of seconds should come free of the board.

When the IC is removed, quickly drop it on the bench and move the torch and pliers to the IC above the one removed. Heating the lower IC pre-warms the board above, making the next removal easier. Also, the board position just heated will cool faster, thereby reducing the amount by which the board will be damaged.

As each column of ICs is removed, the next is done. When all ICs on one half have been removed, reposition the board so the other half is accessible. I've found that the half-way point often can be a good excuse to let the room ventilate and drink a beer.

No matter how carefully and rapidly I've worked, I always burn the board at least once because I have trouble removing an IC, or my pliers slip, or for some other reason. If you consistently burn each board position, your flame is probably too hot. If, however, it takes longer than 5 to 10 seconds to remove an IC, your flame is too cool.

A certain amount of care is necessary when gripping the ICs. Too much pressure may crack them. Too little pressure will let the pliers slip, costing time to reposition them and marring the cases.

When attempting to remove the larger ICs such as 74181s and 74154s, which come in 24 pin DIPs, I have trouble gripping them, so I remove them as a two step process. First, I place an awl under the middle of one side, say between pins 6 and 7. I heat that pin row and, with the awl applying leverage, pull out that row. I then grip the IC on its thinnest dimension, heat the remaining pins, and remove the IC.

So far, by using this technique, my friends and I have removed about 1000 ICs from surplus boards which have about 80-100 ICs each. I tend to break 2% of the chips I pull by applying too much force with the pliers. But a friend has never broken one, so it clearly is an individual matter. Of those chips removed unbroken, we have tested around 250, and have never found a bad chip.

As an unrecommended demonstration of the ruggedness of ICs, I accidentally grossly overheated one, so that when I gripped it in vise grips, the chip was bent in a curve. The plastic case must have softened significantly. After allowing it to cool several minutes to the point where I could handle it by hand, I plugged it into a circuit, expecting it to have failed totally. It worked, although I didn't check out its ac characteristics. Out of general paranoiac distrust for a device so intensely mistreated, I discarded it.

After removing ICs from boards it is usually necessary to clean and straighten the pins. Boards with plated through holes often lose their plating around the IC lead.

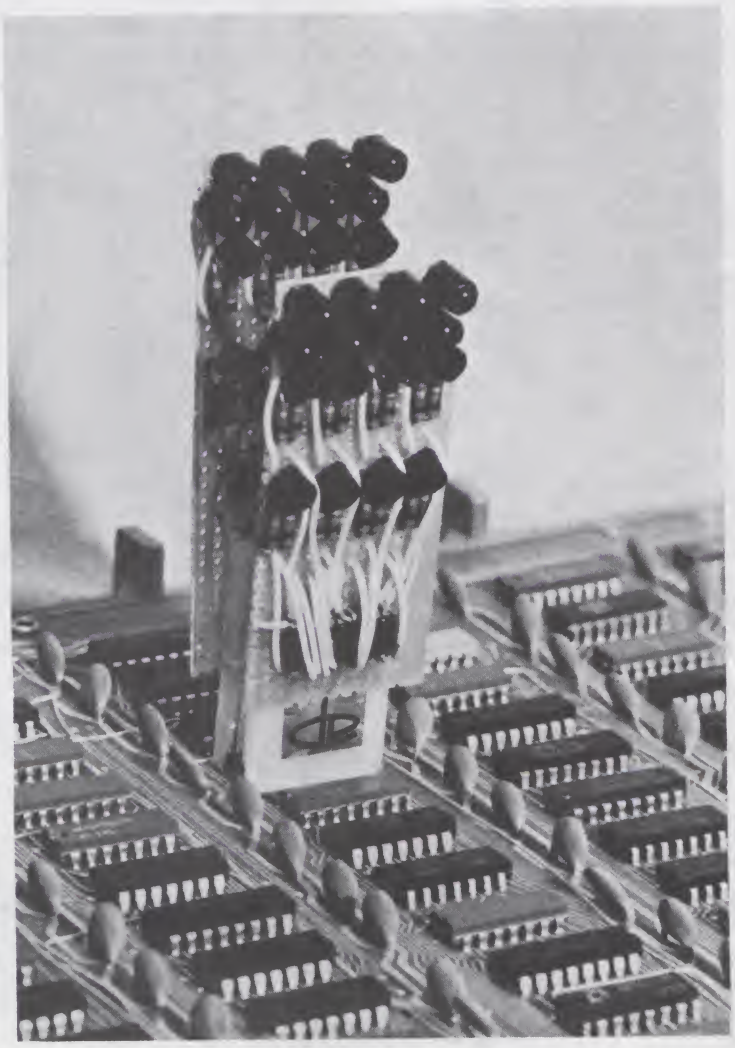
I have found this method useful as a means of quickly building a stock of ICs ready to use in any project. It is limited mainly by the availability of exotic surplus chips, but most standard 7400 series TTL is easily available. The price of 4 cents/chip can't be beat, and the time required – about 10 to 20 minutes/80 chip board – is rather small.

This technique provides a fast, cheap, safe means of removing chips. I hope it proves as effective for you as it does for me.

Powerless IC Test Clip

circuit by
John Errico
5 Richard Rd.
Hudson MA 01749

written by
Robert Baker
34 White Pine Dr.
Littleton MA 01460



This test clip operates like the expensive, commercially available clips selling for $85 or more without requiring batteries or external power. All types of ICs may be tested (TTL, DTL, MOS, etc.) and LEDs are used to indicate the logic state of each pin being tested.

The heart of the test clip is a Texas Instruments TID125 diode array which costs about $3.75. Two diode arrays are used to determine the pin with the highest voltage (Vcc) and the pin with the lowest voltage (ground). These pins are then used to power the LEDs on the test clip itself, thus taking power from the IC on the board and eliminating the need for an external or *separate supply*. The circuit is straight forward and may be expanded to make a 24- or 40-pin test clip. The larger test clip, however, may be difficult to use due to the size of the LED display.

The basic IC clip is a standard item available from AP Products Inc., Box 110-Z, Painesville OH 44077. The 16-pin clip is part number 923700 (TC-16) and sells for $5.75 each.

The diode arrays are 14-pin dip packages and were chosen to make the test clip more compact. To cut down the cost, 16 general purpose silicon diodes may be used in place of each diode array IC. The transistors used to drive the LEDs may be any NPN transistor capable of handling the LED current. Any small size LED may be used; however, the 1k resistance value may have to be changed. Choose a value which gives about 2 mA current through the LED; this should give sufficient brightness without loading down the circuit supply.

Construction is very simple and parts layout is not critical. Use a small piece of 0.1" grid perforated board bolted to each side of the IC clip to mount components on. Try to keep the overall physical size of the boards as small as possible to make the finished test clip easier to handle. The LEDs should be mounted along the top edge of the perforated boards so they are visible from above the clip when it is attached to an IC. I would suggest wrapping a small piece of dark tape or using a short piece of dark tubing around each LED to improve visibility of the finished LED display. One of the TID125 diode arrays is mounted on each piece of perforated board along with the associated resistors and transistors, positioned wherever convenient. Remember to run two wires between the two perforated boards to connect the Vcc and ground outputs of the diode arrays together. These wires should be stranded to withstand the movement of opening and closing the test clip when in use.

Using the test clip is the simplest part of all. Just clip it over the desired IC. Don't worry about how to position the test clip on the IC; pin 1 may be at either end and the test clip will still work properly. With the test clip installed on an IC package the LEDs will indicate the logic level of each pin:

ON = Logic 1 (HIGH) or Vcc pin
OFF = Logic 0 (LOW) or ground pin

On 14-pin ICs disregard the two pins not attached.

Who said building an IC test probe is hard? ■

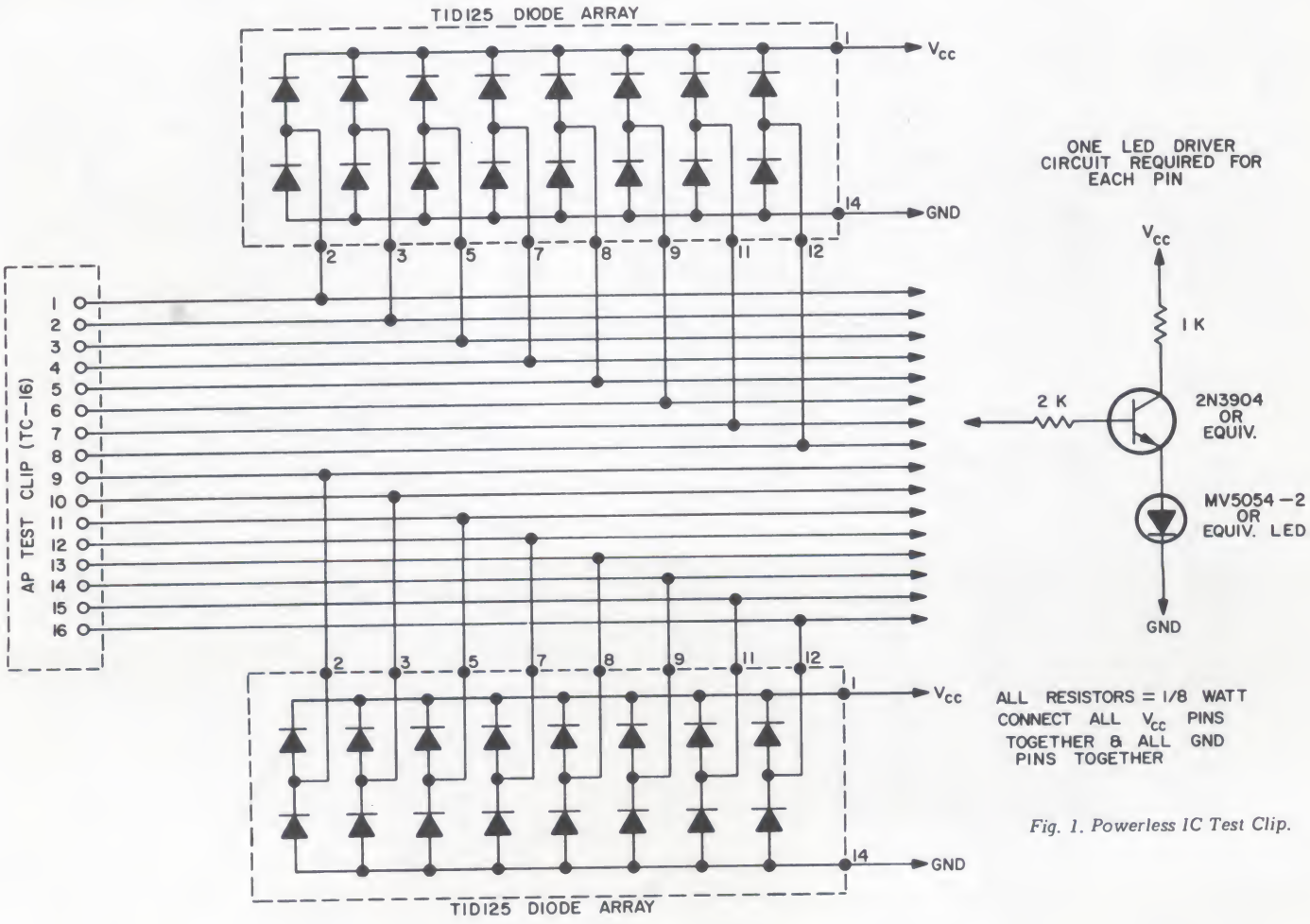


Fig. 1. Powerless IC Test Clip.

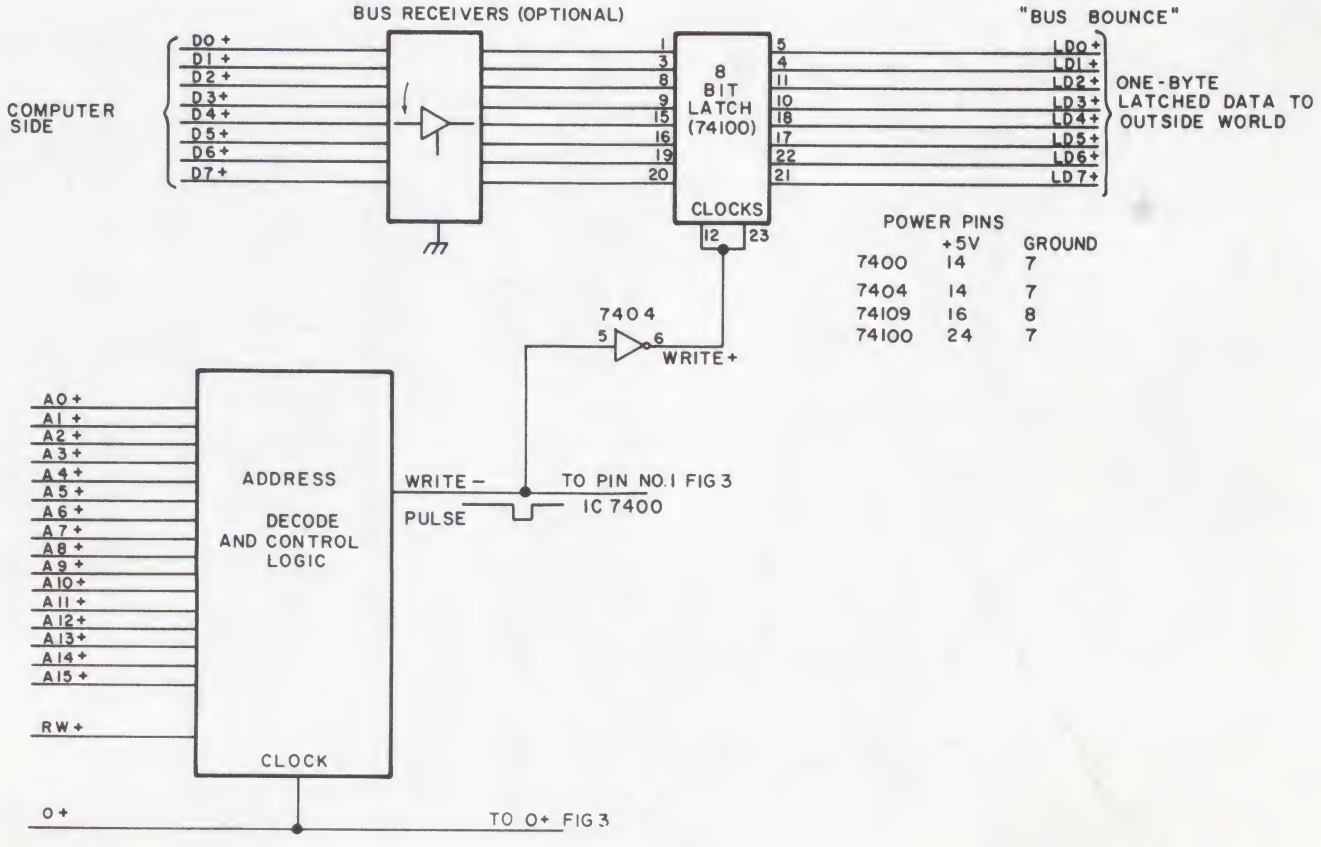
notes on parallel output interfaces

by
Carl Helmers
Editor, BYTE

One way to connect an extra output port for a teletype or other peripheral to your CPU is to make the interface simulate a memory address during the writing operations. This method is the one which is used for both the input and output functions in computers such as the PDP-11 of DEC, or the Motorola 6800 microcomputer. The method can even be used to overlap a usable main memory address since the CPU could care less whether or not the addressed port is connected in addition to the proper main memory location! The same method can even be used on computers such as the Altair 8800 which split the CPU bus into two parts and thus complicate the interface picture.

All of the microcomputers I have seen to date for the home brew computer market operate with a degree of parallelism at the bit level. Whether the chip is 4-bit, 8-bit or a 16-bitter, the concept of "parallel" data is built in. Data is parallel in nature if each bit has one line assigned to it and transfers of a group of such bits are always made simultaneously. Thus for example, the address lines used to select memory words are usually done in

Fig. 1. 8-bit bus output latch.



in memory address space

parallel in CPU designs of practical utility. With the bus oriented computer chips likely to be used for homegrown systems, it is possible to grab data from the busses by latching it in a register which listens to the bus continuously but is only written when the proper address is found. This article concerns such latching of output data and suggestions about several applications of the technique.

The basic idea of the bus output is illustrated in Fig. 1. Fig. 1 shows a set of data lines (denoted D0+ to D7+ to indicate a positive logic definition) constituting an 8-bit data bus. In a 16-bit computer, this set of lines might be one or the other half of the 16-bit data bus, or the logic might be extended to 16 bits. Connected directly to the bus pins of the interface I have noted a set of "bus receivers". This circuit should be put in if necessary to maintain consistency of bus loading with all the other bus interconnects. For instance, with a tri-state 8833 circuit as the bus definer, up to 100 high-impedance PNP-input receivers (input side of 8833) can be connected to the bus. But put a TTL load on, and the fanout will be reduced considerably. (For an Altair 8800, the data bus is split into two components: in and out. The principle of minimizing the loading of the Altair drivers (TTL) would indicate use of a low power (74Lxx) device as the bus receiver. A non-inverting receiver is to be preferred in order to keep the same logical sense of the data to be stored in the latch.)

Following the bus receiver, a latch is shown. The latch illustrated with its pinouts is the 8-bit, 24-pin package called a 74100. Alternate circuits for this function include a pair of 7475s, or even four dual master slave flip flop packages, such as 7473s. In general, it will pay to use the larger scale of integration from a power-budget standpoint. Consider the specs for two 7475s (64 mA) versus four 7473s (80 mA). For a sixteen bit output, all that is required is to double the number of bits used for the latch. The latch is used for only one purpose — to hold the data after it is stored, until updated by a later write to the same location.

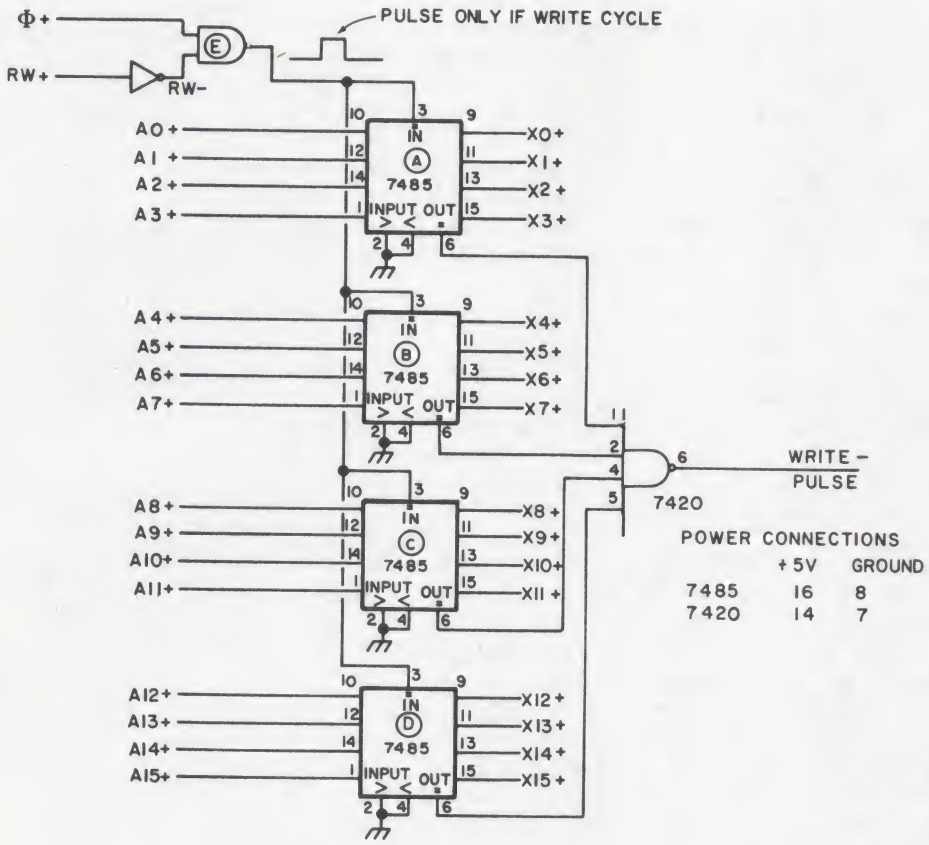
A latch is required to buffer the output logically in many instances of I/O devices. A primary example of such a case is an output which needs stable data for a much longer period than the short CPU-cycle during which data is stable at the output of bus receivers. If you interface your computer bus to one of the Burroughs SELF-SCAN display devices with the memory option, for instance, your data must be stable for a long period of time (about 60 microseconds). This requirement is necessitated by the need to wait for the shift register memory to cycle around to the proper position for entry of new data. If your interface is to a digital to analog converter (DAC) presenting a gain control voltage to a computerized audio mixing panel, then you would want the control signal to stay stable for all time until explicitly altered by the CPU.

Fig. 1 is completed by the notation of a big mostly-blank box. Big blank boxes with labels in them are a way of saying "here is a function, but I haven't told you what it is in detail." In this case the function is address decode and control logic for grabbing output data. I have drawn the box with inputs indicated from 16 bits of addressing, an "RW+" signal and a "Ø+" signal. The logic of this box will respond to a specific address in order to generate a negative logic (WRITE-) pulse which is inverted and used to latch the data at the correct time. The definition of the specific address desired and the decoding are both considered a bit later when Fig. 2 is discussed. The "RW+" signal controls the direction of the CPU's data transfer. If it is logic "1" (high level) then the CPU is attempting to read data from the bus and no clock pulse is allowed to reach the latch, even if the address bits A0+ to A15+ match the desired address. If "RW+" is low, then the CPU is sending data out and a clock pulse is allowed through the address decode and control logic. The clock pulse is taken from the CPU supplied clock Ø+ and is

"Big blank boxes with labels in them are a way of saying 'here is a function, but I haven't told you what it is in detail.' "

"The method can even be used to overlap a usable main memory address since the CPU could care less . . . "

Fig. 2. Single-address 16-bit decode with 7485. "Xn" (n=Ø to 15) is logical 1 or Ø defining desire address.



a positive logic signal indicating that valid data is present.

Assuming you actually want to grab some data off the bus at a specific location when it is referenced, how can you implement the address decode and control function? Fig. 2 is a suggestion of one method to accomplish this function for a *specific location* at a considerable price in hardware: using more than one memory location defined in this way would rapidly lead to a large parts count for 7485s as decoding logic. The basic idea of Fig. 2 is to use the 7485 comparator circuits to check for equality between the address lines A0+ to A15+ and a set of "desired address" definition lines, X0+ to X15+. The comparison is done as four groups of four bits, and a parallel logical product (AND) of the results of all four bit-group comparisons is performed by the 7420. The comparators' cascading input for equality is used to enable the comparison: the AND gate "E" detects a CPU write operation as the simultaneous occurrence of the clock Ø+ and a low state of RW+.

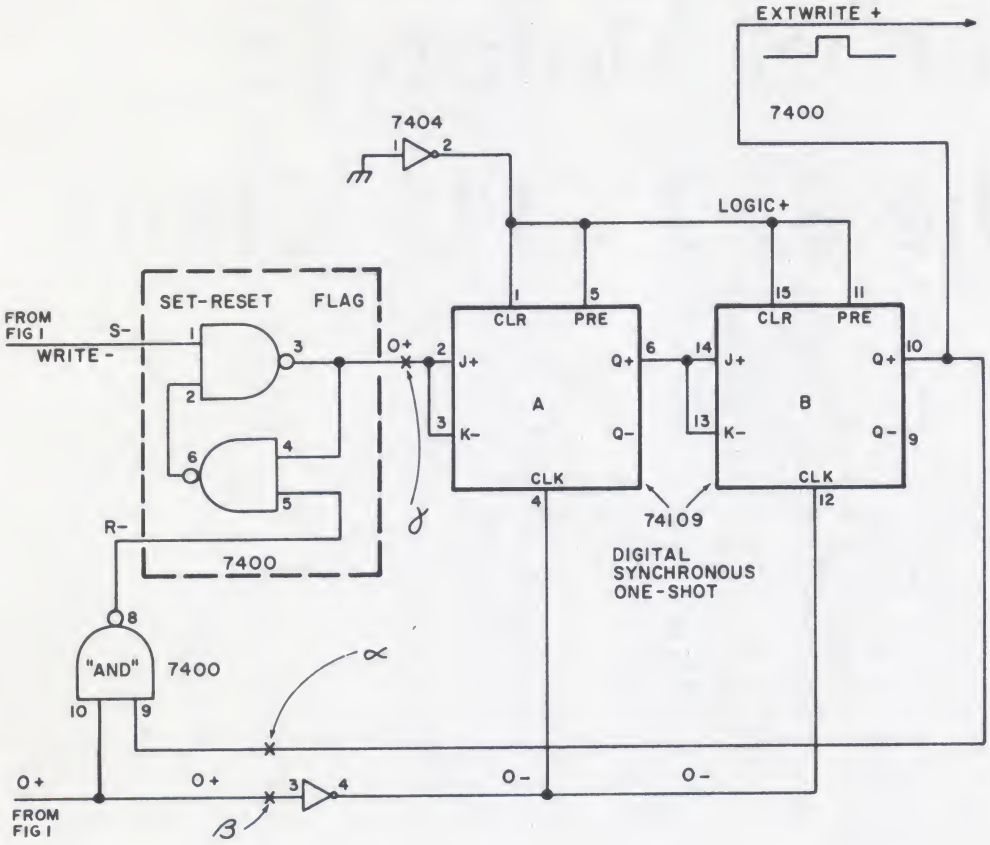
A Hardware Memory Contents Monitor

A particular application for which single-address decoding might be useful is as a debugging tool based on this circuit, used to monitor the last content written into a specific location. Such a debugging tool can be built by defining the X0+ to X15+ address lines as the outputs of a set of four hexadecimal switches or six octal encoded switches, hand set from the panel of the debugging instrument. Then the outputs of the latch circuit might be routed to a set of hex or octal LED drivers so that a display of each number written might be obtained. A more general variation of the same theme would be attainable as a bus monitor device if the gate E of g Fig. 2 is eliminated entirely and the clock Ø+ is simply used as the enable condition of the comparators (pins 3 get Ø+). Then the "memory contents monitor" always shows the contents of the memory bus at the time it was last used with the desired address.

Adding a Longer Clock

It is often necessary to obtain a clock signal which is longer than the original latching clock. In such cases the longer clock must also occur during a time when the latched data is stable, i.e., after the CPU is finished with its addressing of the output latch. One way to generate such a delayed longer clock is to use the analog timing elements called "one shots" — such as the 74122 or 74123 circuits. In order to do so, however, you will have to calculate a bunch of resistor

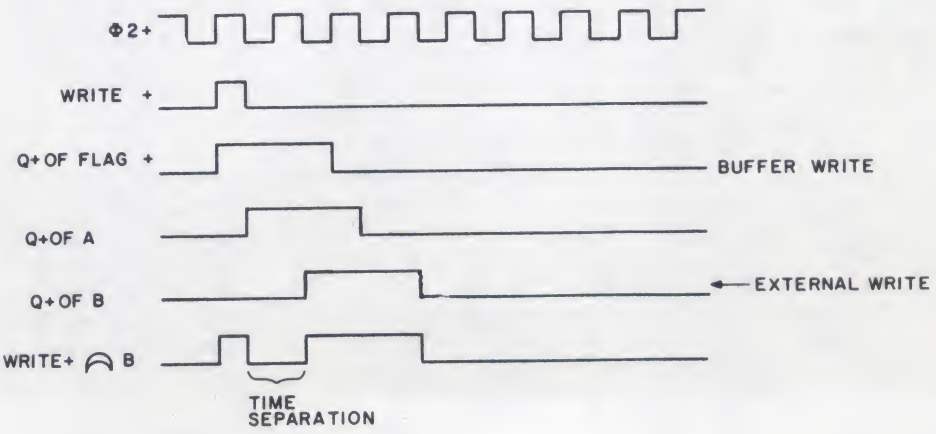
Fig. 3. Generating a longer clock digitally.



and capacitor values for the delays, put in your nearest approximations and cross your fingers. A better way to achieve a deterministic system is to use entirely "synchronous" logic concepts and delays implemented with gates and flip flops.

Fig. 3 and its corresponding timing diagram Fig. 4 is a detail of one method to cue a long but delayed clock pulse. The basic idea is to set a flag (the SR flip flop formed by the two NAND sections and labelled "flag") when the I/O write occurs. This flag becomes data which will get clocked synchronously into flip flop A, then into flip flop B. The output of flip flop B is used to enable a reset pulse to the flag, which brings the system into a stable quiescent state until the next output WRITE- pulse occurs. The timing diagram of Fig. 4 illustrates how the synchronous operation produces an auxiliary pulse (Q+ of flip flop B) which is 2 clock periods in length. This clock is delayed with respect to setting the flag by the original WRITE- pulse, but the delay is fixed and synchronous due to the fact that the actual clock (or its inverted e derivative) is used to cause all state changes of the flip flops.

Fig. 4. Timing: external write vs. buffer write.



Son of Motorola (or, the $20 CPU Chip)

Would you believe – another microprocessor? You bet. The calculator firm, MOS Technology of Norristown, Pennsylvania, has just recently announced a new microprocessor which combines plug in compatibility with the Motorola 6800 and a new instruction set to come out with yet another option for microprocessor users – but at a price of $20 in single quantities. Here comes the under $200 processor kit? Not quite yet, but maybe within a year or two. (It's already to the point where the sheet metal and transformer iron of a home computer often cost more than all the silicon products which make it work . . . this new low on CPU prices just compounds the problem.) It may be three to six months before you see one of these new MCS6501 processors designed into a kit, so Dan Fylstra in his article covers quite a few details of the Motorola 6800 by way of comparison with "Son of Motorola."

by
Daniel Fylstra
Associate Editor, BYTE
25 Hancock St.
Somerville MA 02144

We thought that the "age of the affordable computer" had arrived when you could buy a microprocessor chip for $150. But the potent combination of new technology and free enterprise has brought about developments beyond our wildest expectations.

So now you can buy your microprocessor brand new, in single quantities, for $20. The new offering is from MOS Technology, Inc., and is pin-compatible, but software-incompatible with the Motorola 6800 microprocessor. Although it will be a while before the new chip finds its way into ready-to-build kits for the hobbyist (after all, the first Motorola 6800 kits have just been announced), the news should be of interest to nearly every home brew computer experimenter. So here's a comparison of the Motorola 6800 and the MOS Technology 6500 series, based on the information presently available. If you aren't already familiar with the Motorola microprocessor, don't worry – we'll cover its major features in the course of the comparison.

Hardware Comparison

Both the Motorola 6800 and the MOS Technology chip are TTL-compatible devices, operating from a single five volt power supply. Like earlier microcomputers, such as the Intel 8008, 8080 and National PACE, these processors make use of a bidirectional data bus, to which both memory and input/output devices may be connected. However there are no special input/output instructions in the instruction repertoire of either the Motorola or MOS Technology microprocessors. Output of a character, for example, is accomplished by storing a value into a certain memory location, which is in reality a special register inside an external I/O interface chip, connected to the data bus just like any other RAM or ROM chip.

Motorola supplies a Peripheral Interface Adapter (PIA) chip which connects to the data bus for 8-bit parallel I/O, and an Asynchronous Communications Interface Adapter (ACIA) for bit-serial input/output. (The ACIA is simply a type of UART, as discussed in Don Lancaster's September article on serial interfaces. It may be used to connect a teletype or CRT terminal to the microcomputer system.) MOS Technology plans to supply a similar set of chips.

Most of the time, data is being transmitted between the microprocessor and the memory chips over the data bus. But the processor can also disconnect itself from the bus, enabling, for example, a data transfer to take place directly between an I/O device and memory. Both the Motorola 6800 and the MOS Technology chip have three-state buffers for the eight data lines, enabling them to disconnect from the bus in this fashion. But the Motorola also has three-state buffers on its 16 address lines, whereas the MOS Technology chips do not.

This would be used, for example, in a floppy disk controller which is capable of transferring a whole block of many bytes of data in response to a single command from the CPU. The controller would present a series of addresses on the 16 address lines, and data bytes on the data lines, causing the bytes to be stored in a series of locations in some RAM chip on the bus; all this would take place in the intervals when the CPU itself was disconnected from the bus.

As a practical matter, however, small systems do not require this kind of direct memory access (DMA) capability, and larger systems with more devices on the bus will require buffers on the

Ready or not, here I come: 6800 to 6501.

address lines to supply the necessary power – and these buffers may as well have three-state outputs.

The other major hardware difference between the Motorola 6800 and the MOS Technology 6500 series is that the MOS Technology chip has an 8080-style Ready line, whereas the Motorola 6800 does not. The Ready line is used to make the microprocessor wait for a variable length of time before going on with the execution of an instruction. This feature makes it easy to use the less expensive memory chips, especially for Programmable or Erasable Read-Only Memory (PROM or EROM) which are not as fast as the CPU itself. It is possible to use such devices with the Motorola 6800, of course, by stretching out one of the clock phases to as long as five microseconds. But the availability of the Ready line on the MOS Technology chip is certainly a convenience, and allows you to use extremely slow memories if you wish.

The MCS6501, first in the MOS Technology 6500 series, requires the same type of external clock as the Motorola 6800. But for $25 you can have the MCS6502, which includes an on-the-chip clock, driven by an external single phase clock or an RC or crystal time base input. As the manufacturer suggests, it is probably cheaper in an original design to use the MCS6502 than to provide the external logic to generate the two-phase clock.

To sum up, both the Motorola 6800 and the MOS Technology have comparable features with some differences. In terms of hardware differences, the MOS Technology Ready line is probably more valuable than the three-state address line buffers available on the Motorola 6800.

A final hardware advantage possessed by the MOS Technology chip is speed. The Motorola 6800 cycle time is one microsecond (1 MHz clock rate), and a typical instruction takes about three clock cycles. While the cycle time of the MOS Technology chip is nominally the same, the company has hinted broadly that the chip can be run at clock rates of 2 or even 3 MHz. Of course, one would have to use faster and more expensive memory chips to take advantage of this increased speed.

In addition, certain critical instructions take fewer cycles on the MOS Technology chip. An STA (store accumulator) instruction referencing an arbitrary location takes 4 cycles, versus 5 for the Motorola, and a JSR (jump to subroutine) instruction requires 6 cycles, as opposed to 9 on the 6800. Conditional branches take 4 cycles on the Motorola microprocessor, while they take 2 cycles if the condition is false and 3 if it is true on the MOS Technology chip. Because these instructions are so frequently executed in most programs, the 6500 series should enjoy a performance edge over the Motorola 6800 even at the same clock rate.

Table I. Functionally equivalent instructions for both the Motorola 6800 and MOS Technology MCS6501 microprocessors. The mnemonics are Motorola's. Of course, these instructions operate on the A accumulator only in the MCS6501, but can address either accumulator in the Motorola 6800. The BIT instruction (*) has a different effect on the V and N processor flags in the MCS6501.

ADC	DEX
AND	EOR
ASL	INC
ASR	INX
BCC	JMP
BCS	JSR
BEQ	LDA
BIT*	LDX
BMI	LSR
BNE	NOP
BPL	ORA
BVC	PSH
BVS	PUL
CLC	ROL
CLI	RTI
CLV	RTS
CMP	SBC
CPX	SEC
DEC	SEI
	STA
	STX
	TSX
	TXS

Software Comparison

We can treat the instruction set architecture of the two processors in two stages, first considering the facilities for manipulating *data* and then dealing with the facilities for manipulating *addresses*. Both features are important to the overall effectiveness of the processor design.

Data Manipulation

The instructions for manipulating data are quite similar on the two processors. There are two major differences: First, the Motorola 6800 has two 8-bit accumulators, A and B, while the MOS Technology chip has only one accumulator, A. Second, in addition to conditional branches for unsigned comparisons, the Motorola 6800 has special branch instructions for signed comparisons, but the MOS Technology chip does not. (The signed comparisons treat the two values as positive or negative numbers in two's complement notation, in the range −128 to +127. For example, −1 is represented as 2^8-1 = 11111111. An unsigned comparison would treat this quantity as the largest possible (8-bit) value, whereas a signed comparison would treat it as smaller than, say, zero.)

Table I lists the instructions which are the same for both processors, while Table II lists instructions on the Motorola 6800 which must be replaced by more than one instruction on the 6500 series microprocessors.

We thought that the "age of the affordable computer" had arrived when you could buy a microprocessor chip for $150. But the potent combination of new technology and free enterprise has brought about developments beyond our wildest expectations.

Some of the instructions omitted on the MOS Technology chip are merely incidental; others are more serious. The lack of signed comparisons represents a real inconvenience in many applications. The lack of a simple ADD instruction means that an operation such as A = B + C on one-byte operands must be coded with a "Clear Carry" (CLC) as in this example:

```
CLC
LDA  B
ADC  C
STA  A
```

on the MOS Technology chip. On the other hand, a computation such as A = B + C − D could be coded as

```
CLC
LDA  B
ADC  C
SBC  D
STA  A
```

assuming that the inclusion of "carry" in both operations is indeed desired.

Less serious but still irritating are the absence of the ROR (rotate right), NEG (negate) and COM

Table II. Motorola 6800 instructions which have no direct equivalent in the MCS6501. The information in this table is taken from MOS Technology documentation on the 6500 series.

Motorola 6800 Instruction	Equivalent 6500 Series Sequence
ABA	No B accumulator
ADD	CLC, ADC
BGE loc	BMI *+6, BVC loc, BVS *+4, BVS loc
BGT loc	BMI *+6, BVC *+6, BVS *+6, BVC *+4, BNE loc
BHI loc	BCS *+4, BNE loc
BLE loc	BEQ loc, BMI *+6, BVS loc, BVC *+4, BVC loc
BLS loc	BCS loc, BEQ loc
BLT loc	BMI *+6, BVS loc, BVC *+4, BVC loc
BRA	JMP
BSR	JSR
CBA	No B accumulator
CLR [loc]	LDA #0, [STA loc]
COM [loc]	[LDA loc], EOR #$FF, [STA loc]
DAA	Replaced by SED
DES	Use PHA
INS	Use PLA
LDS loc	LDX loc, TXS
NEG [loc]	EOR #$FF, ADC #1 [or LDA #0, SBC loc]
ROR [loc]	[LDA loc], PHP, LSR, PLP, BCC *+4, ORA #$80, [STA loc]
SBA	No B accumulator
SEV	LDA #1, LSR
STS loc	TSX, STX loc
SUB	CLC, SBC
SWI	BRK saves state without transferring control
TAB	No B accumulator
TAP	PHA, PLP
TBA	No B accumulator
TPA	PHP, PLA
TST	BIT #0
WAI	JMP *
op disp, X [indexed addressing mode]	LDY #disp, op @loc, Y [indirect indexed addressing mode]

(complement) instructions, as well as single-byte instructions to increment and decrement the accumulator. Probably the least significant difference is the omission of the B accumulator on the MOS Technology chip. This is more than made up for by the availability of an extra index register (see below).

All in all, the Motorola 6800 comes out ahead when considering facilities for manipulating data, the most important point in its favor being the availability of the signed comparisons. Generally speaking, however, the basic instructions available on the two processors are quite similar.

Address Manipulation

The greatest architectural differences between the two processors lie in their facilities for manipulating addresses, or their "addressing modes" — and here the MOS Technology chip has much more to offer.

The two microprocessors are the same in one respect: both have special "short forms" of most instructions for referencing the first 256 bytes of memory. This is called "direct addressing" on the Motorola 6800, and "zero page addressing" on the MOS Technology chip. As an example, the most general LDA (load accumulator) instruction is three bytes long; the second and third bytes form the effective address (0-65535), which can reference any byte in memory. The short form of the LDA instruction, however, is two bytes long; the second byte forms the effective address (0-255) of a byte in the first "page" of memory. The "short form" instructions generally take one fewer clock cycle to execute, since only two rather than three instruction bytes must be fetched from memory.

The major differences between the two processors lie in the important area of indexed addressing. The Motorola 6800 has a single 16-bit index register, called X. Essentially all instructions have an indexed addressing form, in which a one-byte displacement (0-255) is added to the address in the index register to form the effective address. The MOS Technology chip, on the other hand, has two 8-bit index registers, called X and Y. All of the computational instructions have indexed addressing forms in which either a one- or two-byte base address is added to the contents of either the X or the Y register to form the effective address.

Which approach is the better one? For the purpose of accessing elements of arrays, or tables of many identical elements, the MOS Technology chip comes out way ahead. This is partly due to the lack of certain critical instructions on the Motorola 6800, such as an instruction to add the contents of an accumulator to the index register, or even to transfer the value in the accumulators to the index register.

Suppose that we wish to add the Ith element of an array, S_I, to another variable, T. In general, the array may be located anywhere in memory, and the subscript I may be the result of some calculation done in the accumulators. Letting S denote the address of the zeroth element (the base address) of the array, and assuming that the value of the subscript I is already in the A accumulator, consider the instructions necessary to accomplish this operation on the two processors.

The biggest difference is in the area of addressing modes, an area where the 6500 series devices far outshine the Motorola 6800.

On the Motorola 6800, our first try yields the following:

```
SHI   EQU S/256*256
      CLR B          ]
      ADD A #S-SHI   ]  Calculate the
      ADC B #S/256   ]    indexed address
      STA A TEMP+1   ]
      STA B TEMP     ]
      LDX TEMP       ]
      LDA A 0, X     ]  Perform desired
      ADD A T        ]    computation
      STA A T        ]
```

This instruction sequence requires 19 bytes, counting the two-byte temporary TEMP and assuming that TEMP and T are located in the first 256 bytes of memory. Since the array S could be anywhere in memory, we were unable to use the displacement field of an instruction with indexed addressing for the array base address, and instead we had to add the array base to the index (in double precision), store the result in memory, load it into the index register, and finally reference the array element S_I.

We can improve on this with the aid of a little lateral thinking. Noticing that the 6800 is actually capable of adding a one-byte quantity to a two-byte address, but only in a storage reference with indexed addressing, we will split up the base address into two parts to arrive at a better solution:

```
SHI     EQU S/256*256
        STA A TEMP+1
        LDX TEMP
        LDA A S-SHI, X
        ADD A T
        STA A T
          .
          .
          .
TEMP    FDB SHI
```

This instruction sequence requires only 12 bytes, under the same assumptions.

Even so, we can't match the simplicity of the solution to the same problem on the MOS Technology chip:

```
TAX
LDA S, X
ADD T
STA T
```

This instruction sequence requires only seven bytes. Only four bytes were needed to reference the element S_I, versus eight for the Motorola 6800.

How important is this improvement? It is certainly significant, since arrays and tables are used so frequently in programs of any size. On the other hand, in many applications it is only necessary to reference each element of an array in turn; it is not necessary to access elements randomly based on a computed subscript. In this case, we can obtain better code on the Motorola 6800 by first loading the array base address into the index register, and then referencing each element directly (i.e., with a zero indexed address displacement), incrementing the address in the index register using the INX instruction to proceed from element to element. We are therefore using the 6800's index register to hold a pointer or indirect address rather than an index.

An even more important difference between the two microprocessors in that the MOS Technology chip possesses *two* (8-bit) index registers, X and Y, whereas the Motorola 6800 has only *one* (16-bit) index register X. As we shall see, two index registers are far more valuable than two accumulators. This is because programs frequently manipulate two (or more) tables, or other indirectly addressed variables, at the same time. As an example, we will consider perhaps the simplest operation of this type, the problem of moving a string of bytes from one area of storage to another. Assume that 20 bytes, starting at the location denoted by the symbol FROM, are to be moved to the area starting at the location denoted by the symbol TO.

On the Motorola 6800, we can write the following routine:

```
LOOP   LDX FRPTR  ]  Fetch
       LDA A 0, X ]  FROM
       LDX TOPTR  ]  Move
       STA A 0, X ]  TO
       INC FRPTR  ]  change
       INC TOPTR  ]  pointers
       DEC COUNT  ]
       BNE LOOP      Test continuation
         .
         .
         .
FRPTR  FDB FROM
TOPTR  FDB TO
COUNT  FCB 20
```

This routine requires 24 bytes, including the working storage locations, and executes in 820 clock cycles. This routine can move up to 256 bytes.

On the MOS Technology chip we have the following solution:

```
       LDX #0
       LDY #0
LOOP   LDA FROM, X
       STA TO, Y
       INX
       INY
       DEC COUNT
       BNE LOOP
         .
         .
         .
COUNT  FCB 20
```

Two index registers are far more valuable than two accumulators.

This routine requires 17 bytes, and executes in 404 clock cycles. The improvement in speed clearly depends on the number of bytes to be moved; each pass through the loop in the Motorola 6800 routine takes 41 clock cycles, while each pass through the loop in the MOS Technology routine takes 20 cycles. (The MOS Technology routine is also limited to moving at most 256 bytes.)

Once again the degree of improvement is substantial, and the improvement is

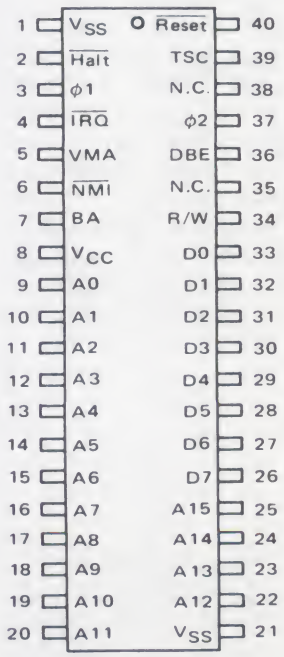


Fig. 1. The pin assignments of the Motorola 6800 (and by implication, the MOS Technology MCS6501). V_{SS} is ground (0 volts) and V_{CC} is +5 volts. The A lines are address outputs, and the D lines are bidirectional tristate data bus lines. For details see the Motorola and MOS Technology documentation of these parts.

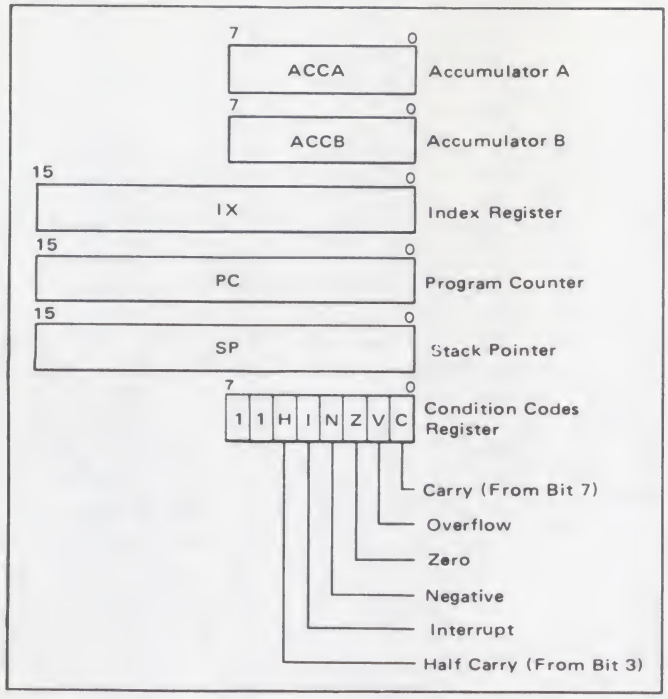


Fig. 2. The programmer's view of the 6800 CPU. This diagram, excerpted from the Motorola 6800 documentation, shows the various registers of the CPU including the processor's condition code register. Note the similarity to the MCS6501 in Fig. 3.

significant because this type of problem arises so frequently in large programs.

The MOS Technology chip has some additional addressing modes not possessed by the Motorola 6800. First, there is a "short form" for instructions with simple indexed addressing if the array base address is in the first "page" (256 locations) of memory. This feature is of somewhat limited use except in very small programs, since only a few small arrays can actually be placed in the first 256 locations. Of greater interest is the so-called "indirect indexed" addressing mode. Instructions with this type of addressing are two bytes long; the second byte specifies the address of a two-byte constant in the first page of memory. This two-byte constant then becomes the "array base address," and the contents of the Y register are added to this constant to form the effective address. This addressing mode is very useful: In a program with many references to a particular array or table which is too large to place in the first page of memory, one can *trade space for time* by placing the array base address in the first page of memory, and then referencing elements of the array using indirect indexed addressing. Each element reference takes less space (two bytes instead of three) but more time (five cycles instead of four) than would be required for ordinary indexed addressing.

There are two other addressing modes on the MOS Technology chip which are somewhat less useful. The first is called "indexed indirect" addressing: Here the contents of the X register are added to a one-byte base address to obtain the address of a two-byte constant in the first page of memory. The contents of this two-byte constant then becomes the effective address. Unfortunately this addressing mode is not available for the JMP instruction, where it would be most useful: It could be used to implement a "jump table," or a "computed GO TO" or "CASE statement" in some high-level languages.

Finally, two other addressing modes are used with branch instructions:

One unfortunate feature of the MOS Technology chip's many addressing modes is that they do not apply consistently to all instructions.

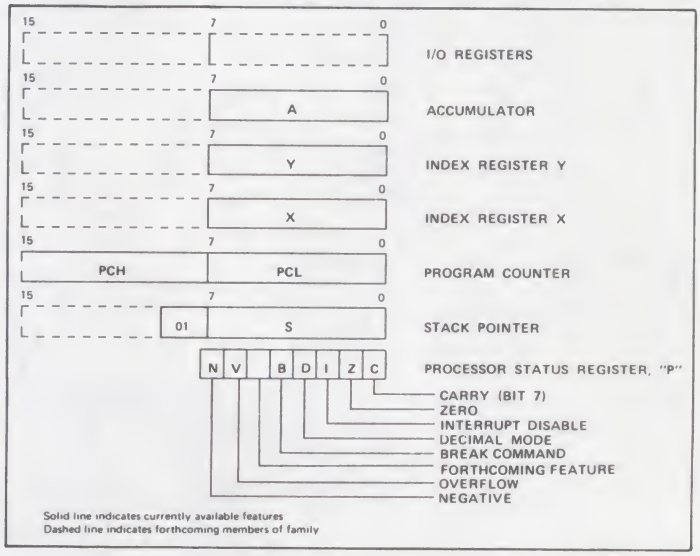


Fig. 3. The programmer's view of the MCS6501 CPU. This diagram, excerpted from the MOS Technology 6500 series preliminary documentation, shows the various registers of the CPU. Note the similarity to the Motorola 6800 diagram in Fig. 2.

Table III. Instructions, addressing modes and execution times for the Motorola 6800 processor. Execution times are in "machine cycles" which for a 1.0 MHz clock take 1.0 microsecond apiece. This table is excerpted from Motorola documentation on their processor.

	(Dual Operand)	ACCX	Immediate	Direct	Extended	Indexed	Implied	Relative
ABA		•	•	•	•	•	2	•
ADC	x	•	2	3	4	5	•	•
ADD	x	•	2	3	4	5	•	•
AND	x	•	2	3	4	5	•	•
ASL		2	•	•	6	7	•	•
ASR		2	•	•	6	7	•	•
BCC		•	•	•	•	•	•	4
BCS		•	•	•	•	•	•	4
BEA		•	•	•	•	•	•	4
BGE		•	•	•	•	•	•	4
BGT		•	•	•	•	•	•	4
BHI		•	•	•	•	•	•	4
BIT	x	•	2	3	4	5	•	•
BLE		•	•	•	•	•	•	4
BLS		•	•	•	•	•	•	4
BLT		•	•	•	•	•	•	4
BMI		•	•	•	•	•	•	4
BNE		•	•	•	•	•	•	4
BPL		•	•	•	•	•	•	4
BRA		•	•	•	•	•	•	4
BSR		•	•	•	•	•	•	8
BVC		•	•	•	•	•	•	4
BVS		•	•	•	•	•	•	4
CBA		•	•	•	•	•	2	•
CLC		•	•	•	•	•	2	•
CLI		•	•	•	•	•	2	•
CLR		2	•	•	6	7	•	•
CLV		•	•	•	•	•	2	•
CMP	x	•	2	3	4	5	•	•
COM		2	•	•	6	7	•	•
CPX		•	3	4	5	6	•	•
DAA		•	•	•	•	•	2	•
DEC		2	•	•	6	7	•	•
DES		•	•	•	•	•	4	•
DEX		•	•	•	•	•	4	•
EOR	x	•	2	3	4	5	•	•

	(Dual Operand)	ACCX	Immediate	Direct	Extended	Indexed	Implied
INC		2	•	•	6	7	•
INS		•	•	•	•	•	4
INX		•	•	•	•	•	4
JMP		•	•	•	3	4	•
JSR		•	•	•	9	8	•
LDA	x	•	2	3	4	5	•
LDS		•	3	4	5	6	•
LDX		•	3	4	5	6	•
LSR		2	•	•	6	7	•
NEG		2	•	•	6	7	•
NOP		•	•	•	•	•	2
ORA	x	•	2	3	4	5	•
PSH		•	•	•	•	•	4
PUL		•	•	•	•	•	4
ROL		2	•	•	6	7	•
ROR		2	•	•	6	7	•
RTI		•	•	•	•	•	10
RTS		•	•	•	•	•	5
SBA		•	•	•	•	•	2
SBC	x	•	2	3	4	5	•
SEC		•	•	•	•	•	2
SEI		•	•	•	•	•	2
SEV		•	•	•	•	•	2
STA	x	•	•	4	5	6	•
STS		•	•	5	6	7	•
STX		•	•	5	6	7	•
SUB	x	•	2	3	4	5	•
SWI		•	•	•	•	•	12
TAB		•	•	•	•	•	2
TAP		•	•	•	•	•	2
TBA		•	•	•	•	•	2
TPA		•	•	•	•	•	2
TST		2	•	•	6	7	•
TSX		•	•	•	•	•	4
TSX		•	•	•	•	•	4
WAI		•	•	•	•	•	9

NOTE: Interrupt time is 12 cycles from the end of the instruction being executed, except following a WAI instruction. Then it is 4 cycles.

"Relative" addressing, available on both the Motorola and the MOS Technology processors, is used with the conditional branch instructions, which are two bytes long. The second byte of such an instruction specifies a positive or negative displacement in two's complement notation (−128 to +127). The destination address of the branch is taken to be the algebraic sum of the address of the byte immediately following the branch instruction and this displacement. Of course, this means that it is possible to branch directly to a location within only a certain limited distance from the branch itself; but, more often than not, the range of −128 to +127 bytes is adequate, and a space savings is realized in comparison to processors such as the Intel 8080 which have only three-byte branch instructions. If necessary, a conditional branch can always transfer to a three-byte unconditional JMP instruction, which can jump to any location in memory. On the MOS Technology chip, a JMP instruction can also specify "absolute indirect" addressing: In this case, the second and third bytes of the instruction specify the address of a two-byte constant anywhere in memory, and the contents of this two-byte constant becomes the destination address for the jump.

One unfortunate feature of the MOS Technology chip's many addressing modes is that they do not apply consistently to all instructions. For example, the binary arithmetic instructions are available with essentially all addressing modes, but the unary arithmetic instructions are missing the Y-register and indirect modes, and the BIT instruction is missing several others as well. This not only makes programming more difficult, since one must constantly check to see which instruction forms are legal, and program around the exceptions; it also makes the design of an assembler or compiler more complicated. A compiler, in particular, would require complex logic to determine when it could and could not take advantage of the addressing modes.

Which processor comes out ahead overall? To a great extent it depends on your point of view: Systems programs are better on the MOS Technology machines; applications programs would tend to come out ahead on the Motorola 6800.

In summary, the MOS Technology chip comes out ahead when considering facilities to manipulate addresses, and in many cases the advantage realized due to the availability of the extra addressing modes is substantial. The greatest failing of the 6500 series design is the inconsistent availability of the addressing modes from instruction to instruction.

Which processor comes out ahead overall? This is very difficult to judge. It depends partly on whether the programs being executed on the microcomputer are "system" programs, such as compilers, interpreters and I/O controllers, which tend to make heavy use of address

Table IV. Instructions, addressing modes and execution times for the MOS Technology MCS6501 processor. Execution times are in "machine cycles" which for a 1.0 MHz clock take 1.0 microsecond apiece. This table is excerpted from MOS Technology documentation on their processor.

	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect, X)	(Indirect), Y	Absolute Indirect
ADC		2	3	4		4	4*	4*			6	5*	
AND		2	3	4		4	4*	4*			6	5*	
ASL	2		5	6		6	7						
BCC										2**			
BCS										2**			
BEQ										2**			
BIT			3			4							
BMI										2**			
BNE										2**			
BPL										2**			
BRK													
BVC										2**			
BVS										2**			
CLC									2				
CLD									2				
CLI									2				
CLV									2				
CMP		2	3	4		4	4*	4*			6	5*	
CPX		2	3			4							
CPY		2	3			4							
DEC			5	6		6	7						
DEX									2				
DEY									2				
EØR		2	3	4		4	4*	4*			6	5	
INC			5	6		6	7						
INX									2				
INY									2				
JMP						3							5

	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect, X)	(Indirect), Y	Absolute Indirect
JSR						6							
LDA		2	3	4		4	4*	4*			6	5*	
LDX		2	3		4	4		4*					
LDY		2	3	4		4	4*						
LSR	2		5	6		6	7						
NØP									2				
ØRA		2	3	4		4	4*	4*			6	5*	
PHA									3				
PHP									3				
PLA									4				
PLP									4				
RØL	2		5	6		6	7						
RTI									6				
RTS									6				
SBC		2	3	4		4	4*	4*			6	5*	
SEC									2				
SED									2				
SEI									2				
STA			3	4		4	5	5			6	6	
STX			3		4	4							
STY			3	4		4							
TAX									2				
TAY									2				
TSX									2				
TXA									2				
TXS									2				
TYA									2				

* Add one cycle if indexing across page boundary.
** Add one cycle if branch is taken. Add one additional if branching operation crosses page boundary.

In favor of the 6500 series are price and speed; in favor of the 6800 are availability and very good Motorola documentation.

manipulation facilities; or application programs, which make greater use of data manipulation facilities. One would expect better results in the former case with the MOS Technology chip, and in the latter case with the Motorola 6800. One would also expect the MOS Technology chip to enjoy an advantage on large programs, since larger programs inevitably tend to make use of tables, subroutines with parameters, and other forms of address manipulation.

All in all, the Motorola 6800 comes out ahead when considering facilities for manipulating data . . . but nevertheless the two processors are quite similar.

Against these factors one must weigh the availability of an excellent applications manual, proven software, and kits for the hobbyist for the Motorola 6800 microprocessor. At the same time, the MOS Technology chip's price can't be beat, and its speed advantage may be important for some purposes.

At the time that this article is being written (late August), the MOS Technology chip is just a promise: The chip should be available for purchase at the Western Electronics Conference (Wescon) in San Francisco, September 16-19. By the time you read this, the chip itself should be in the hands of at least a few hobbyists. Let's have some letters to BYTE describing initial experiences with the new microprocessor! Send your comments to the author or to the editor of BYTE. In the meantime, we'll be waiting to see what new surprises the semiconductor houses and kit manufacturers have in store for us. And BYTE will try to keep you up to date on the latest developments in the world's hottest, fastest-moving hobby – home computers!

More information on the 6500 series microprocessors is available from:

MOS Technology, Inc.
Valley Forge Corporate Center
950 Rittenhouse Rd.
Norristown PA 19401
1-215-666-7950

Information on the Motorola 6800 microprocessor is available from many local distributors, and from:

Motorola Semiconductor Products Inc.
Box 20912
Phoenix AZ 85036

Table V. MCS6501 microprocessor instructions, listed in alphabetical order by mnemonics. The instructions with asterisks are similar to the same mnemonics in the Motorola 6800 processor.

*ADC	Add with Carry to Accumulator	*JSR	Jump to New Location Saving Return Address
*AND	"AND" to Accumulator	*LDA	Transfer Memory to Accumulator
*ASL	Shift Left One Bit (Memory or Accumulator)	*LDX	Transfer Memory to Index X
		LDY	Transfer Memory to Index Y
*BCC	Branch on Carry Clear	*LSR	Shift One Bit Right (Memory or Accumulator)
*BCS	Branch on Carry Set		
*BEQ	Branch on Zero Result	NØP	Do Nothing - No Operation
*BIT	Test Bits in Memory with Accumulator	*ØRA	"OR" Memory with Accumulator
*BMI	Branch on Result Minus	*PHA	Push Accumulator on Stack
*BNE	Branch on Result not Zero	PHP	Push Processor Status on Stack
*BPL	Branch on Result Plus	*PLA	Pull Accumulator from Stack
*BRK	Force an Interrupt or Break	PLP	Pull Processor Status from Stack
*BVC	Branch on Overflow Clear	*RØL	Rotate One Bit Left (Memory or Accumulator)
*BVS	Branch on Overflow Set		
*CLC	Clear Carry Flag	*RTI	Return From Interrupt
CLD	Clear Decimal Mode	*RTS	Return From Subroutine
*CLI	Clear Interrupt Disable Bit	*SBC	Subtract Memory and Carry from Accumulator
*CLV	Clear Overflow Flag	*SEC	Set Carry Flag
*CMP	Compare Memory and Accumulator	SED	Set Decimal Mode
*CPX	Compare Memory and Index X	*SEI	Set Interrupt Disable Status
CPY	Compare Memory and Index Y	*STA	Store Accumulator in Memory
*DEC	Decrement Memory by One	*STX	Store Index X in Memory
*DEX	Decrement Index X by One	STY	Store Index Y in Memory
DEY	Decrement Index Y by One	TAX	Transfer Accumulator to Index X
*EØR	Exclusive or Memory with Accumulator	TAY	Transfer Accumulator to Index Y
*INC	Increment Memory by One	*TSX	Transfer Stack Register to Index X
*INX	Increment X by One	TXA	Transfer Index X to Accumulator
INY	Increment Y by One	*TXS	Transfer Index X to Stack Register
*JMP	Jump to New Location	TYA	Transfer Index Y to Accumulator

GLOSSARY

BYTE's Board of Resident Inexperts (BRI) has ruled the following terms to be worthy of further explanation. This list is probably not complete – readers who would like further explanation of terminology are invited to write a letter to the editor identifying terms which need such treatment.

8-Bit Bidirectional Bus – a "data bus" which simultaneously transmits eight separate signals corresponding to one byte's worth of information. The bidirectional aspect means that either tristate, open collector or similar form of output stage is used, so that multiple drivers can be tied in common with only one such driver active at any time. A given board, CPU, output terminal or other logic circuit can then interface to the bus (with some addressing and master timing control intelligence) for both sending and receiving data.

Effective Address – whenever the computer's CPU addresses memory, it must send out 16 bits (for Motorola 6800, MCS 6501 or other similar chips). The way in which these 16 bits are derived can often be a fairly elaborate procedure, as well as a simple absolute expression. Whatever the method of derivation, however, the result is a 16-bit value which is used to address memory, called the effective address because it is what actually does go out to memory regardless of the details of the internal codes of the program.

Instruction Repertoire – the repertoire of a musician is the set of all pieces he or she can play well in concert. Well, the repertoire of a computer – its instructions – is the list of all the instructions it can perform and their definitions.

Subscript – in typical high order languages, a means is provided to specify elements of arrays of data. This is done by subscripts to indicate the "nth" element for subscript "n". Use of such notation presents the problem of calculating the *effective address* of the actual data being referenced. In the context of evaluating a CPU, attention spent on the problem of calculating effective addresses from subscripts is very fundamental.

Time Base – whenever it is necessary to examine the relative timing of different signals, it is necessary to have a reference point and a scale for making the measurement. This is the "time base" of the reference.

TTL compatible – one of the largest families of integrated circuits is the line of "transistor-transistor logic" devices, TTL for short. A TTL compatible line of some non-TTL device can "drive" one or more TTL loads if it is an output, or can receive a TTL device's output if it is an input. There are various cautions to be observed – probably worthy of a BYTE article – when different types of logic are interfaced, but the phrase "TTL compatible" usually means that the compatible device can be wired directly to TTL interconnection pins safely in at least one configuration.

Unary – this term is derived from the Latin roots of "oneness." A unary operation is an operation which has but one operand, for example the complement operation of a Motorola 6800 CPU.

Data Paths

Gary Liming
3152 Santiago Dr
Florissant MO 63033

Data transmission usually brings to mind terminals, telephone lines, satellites, and large computer centers. Computer links in retail stores, banks, airlines and government agencies are becoming more and more widespread. Such large scale operations can easily cost millions of dollars and are thus out of the range of the hobbyist. The prospect of linking home systems across distances for program swapping and interactive games will undoubtedly become more a possibility as the technology improves.

However, data transmission in a broader sense doesn't have to mean large networks of computers and remote terminals. It is defined as the process of sending error free bits from one place to another, and applies to all digital systems regardless of complexity. In this article we illustrate some data transmission principles applied to hobby system design, over distances ranging from the length of printed circuit foil runs on a circuit card to the extremely long distances involved in phone or radio links.

Data transmission in a broader sense doesn't have to mean large networks of computers and remote terminals.

Communication Theory

Data transmission is part of the broader subject of communication theory which is used to analyze communication systems. Any communication system has three parts: a message *source*, a *medium*, and a *receiver*. To communicate, information of some kind must be transferred. Information is defined simply as an orderly representative signal. Orderly means that the signal is sent in a known format which can be interpreted and decoded by the receiver. Representative means that there is agreement between source and receiver upon what the signal will mean. A signal could be a series of printed characters, a bell, a whistle or even a color. The smallest unit of information is the bit, representing only an on off or yes no condition. One or a series of these fundamental bit signals makes up the message in digital communications.

Any medium that can transfer a message has limits, and the medium within these limits is called the *channel*. The limits which define a channel might be physical properties such as the technologically available bandwidth, or human defined limits such as an arbitrary FCC ruling that a radio station is allocated a particular set of frequencies with a prescribed bandwidth for its signals. *Noise* is defined as any signal that interferes with the message, like radio static or dirt on a camera lens.

A communication that works one-way, or does not allow information to be mutually exchanged is called a *simplex* transmission or communication. If information can be exchanged, it is called a *duplex* system. There are two kinds of duplex systems: If information can be sent between two points simultaneously, it is called a *full duplex* system; if the information can be transferred in both directions but not at the same time, it is called – you guessed it – *half duplex*. Figure 1 illustrates the various kinds of communications exchanges.

Let's apply this to a simple example – consider the page you are looking at. The author is the message source, you are the receiver, and paper and ink are the medium. The size of the page sets the channel limits, and ink blots or printing errors comprise noise. Communication is simplex. When a reader replies, it has become half duplex.

This point of view can be applied at different levels to your system design. Integrated circuits, printed circuit boards, peripherals and terminals can all be considered sources and receivers. They all use the bit as the common unit of information.

An important factor in data communications is the *data or transmission rate* at which the bits are transferred. This is measured, naturally, in bits per second (abbreviated b/s). It is on this simple point that many newcomers first get into trouble by using the term baud. Baud has a different meaning which can be ambiguous, as we will see when we look at modulation methods and modems.

Another important parameter of information transmission is the *error rate*, measured by the number of bits in error out of the

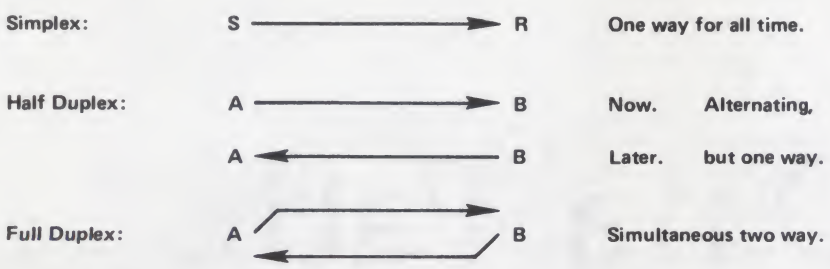


Figure 1: The terminology describing modes of communication between sender and receiver depend upon who is sending data and when the data is sent.

Data transmission is the process of sending error free bits from one place to another.

total number transmitted. If your computer processes instructions at 500 kilobytes per second which equals 4 megabits per second, an error rate of one in 10 million (10**–7) will give you an error on the average of every 2.5 seconds. Clearly, what is a tolerable error rate depends on the transmission rate. A central processor which has errors every 2.5 seconds is not very usable.

Medium Characteristics

To transfer data, the hobbyist can use any medium that will support a bit stream. There isn't any reason why you couldn't take a serial interface and hook it up to modulate a laser beam. However, since most of the transmission done by the hobbyist uses conductors, let's first look at small gauge wire used as a communication channel.

As the data rate through the wire increases, the bit stream begins to look like an AC signal passing through a transmission line, and must be treated as such. This *is not* necessarily related to a reversal of current flow, like AC, but is due to the fast rise and fall times of the pulses.

Therefore, the channel must have a fairly wide band of frequencies it can pass (*bandwidth*). It must pass an AC signal with approximately the same rise and fall times as the pulse and the flat portion of the pulse, which is essentially AC at zero frequency.

Conductor properties such as resitance, capacitance, and inductance degrade the quality of the pulses. The voltage drop due to the resistance of the wire lowers the voltage of the received pulse. Capacitance between the signal wire and the ground wire shunts some of the voltage, and inductance and capacitance both provide impedance to the flow of the pulses. Noise induced from the environment and the power supply will further degrade signal quality.

Another problem amateur radio operators will be familiar with is the skin effect, where high frequency current tends to concentrate in the outer layers of the wire, increasing the effective resistance. Also, the propagation time of the pulses should be taken into account. Even though the pulses travel at near the speed of light, for 22 AWG (0.79 mmϕ) wire, the delay is about 1.5 ns/ft (4.9 ps/cm); a 100 foot coaxial cable introduces a transmission delay which is nearly a whole machine cycle delay in some high speed systems. Indeed, such transmission lines are often used as delay elements in oscilloscopes.

All these phenomena depend on the length of the wire and the frequencies of the signal. They can combine to ruin the shape of the pulse to the extent that the logic gates can become confused as to whether they are seeing a zero or a one. We conclude then that the longer the wire and the higher the transmitted frequencies, the harder it is to get an acceptable error rate.

Microtransmissions

To communicate, information of some kind must be transferred.

Armed with these characteristics and definitions, let's look at how conductors affect data transmission in a typical processor. To date hobby systems have been predominantly designed with 7400 series TTL, which can handle clock frequencies up to around 35 MHz, but are commonly clocked at around 1 MHz. At 1 MHz, characteristics like resistance and capacitance of wires are not significant for short transmissions such as chip to chip or board to board transfers. The big problem *inside* systems is induced high frequency noise due to changes of logic states. The typical TTL transition time of ten nanoseconds has a significant harmonic content well into the VHF range of 50 to 200 MHz. (This is the reason your computer can generate some powerful television interference if it is not properly shielded.) The current surges at the power connection of a TTL gate which is changing state induces a noise signal, since the power bus is typically a poor conductor of VHF.

Thus one common source of noise is a poorly designed power supply and distribution system. Because of its high speed characteristics, TTL logic is very sensitive to

changes in its supply voltage. The power surges of one gate changing state can momentarily drop the level of a local power distribution wire, affecting its neighboring integrated circuit and thereby giving birth to a *glitch* in the system. Detecting a glitch is a real hassle for the pros, and the best policy is to use sound design practices from the start. The design of well regulated power supplies is a significant subject in its own right, and will not be covered in this article. Home brew computer experimenters can often find excellent high current logic power supplies in surplus stores.

Noise spikes in the power wiring can also occur between chips on the power paths and can spread to other chips and boards. These noise spikes in the power wiring are induced due to the inductance of the printed circuit foils or wire wrap wires as the gates change state and draw a lot of current. Using wide flat power supply runs in the PC artwork will lower the high frequency impedance of the conductors. Problems can be further minimized by placing many small ceramic decoupling capacitors of approximately 0.01 uF between the positive power supply bus and ground. Use one decoupling capacitor for every five to ten TTL integrated circuits. Using an integrated circuit voltage regulator on each board will also help provide isolation of power supply noise between boards.

What is a tolerable error rate depends upon the number of bits per second transferred.

A well grounded case will greatly help reduce environmental noise. The case will also shield you from your neighbor's complaints about interference with his television reception. Another benefit of a well grounded system case or chassis is protection from static electricity. In a dry house in winter, shuffling across the room to turn on the system can wipe out some MOS chips, as I know from bitter experience.

These may not sound like important data transmission problems, but they are direct results of the same high frequency transmission characteristics which affect long wire links. Troubles that start with an improperly designed power distribution and layout scheme are hard to spot and correct, but will certainly show up in transmissions over long wires.

Macrotransmissions

Macrotransmission problems occur between central processors and peripherals. The transmission line characteristics become important: If the length of the wire approaches the order of magnitude of the wavelength of the signal, transmission line effects are a potential source of problems. This phenomenon occurs in short wires at high frequencies, and in longer wires at lower frequencies. As mentioned previously, the frequency characteristics of TTL logic circuits changing state — VHF components in the 100 MHz range — are what tend to dominate the transmission line properties of long wires carrying TTL signals. Using the usual radio formula,

$$\lambda = 300/f \quad (\lambda \text{ in meters, f in MHz})$$

gives wavelengths for the high frequency components of a TTL state transition which are in the vicinity of three meters at 100 MHz. Thus cables with lengths of one or two meters should exhibit many of the properties of transmission lines when they carry standard TTL signals. Note that this property primarily depends upon the transition time, and is independent of the actual number of transitions per second. By slowing the transition time by a factor of 100 to one microsecond or more, transmission line effects will not begin to occur until cables of 100 meters or more are considered. Given some arbitrary length of cable, the alternatives open are to take into account transmission line behavior through impedance matching techniques, or to slow down the signals so that transmission line effects are no longer a consideration. Since the latter option produces a non-TTL signal because it changes state too slowly, let's turn attention to methods of compensating for transmission line behavior.

As a simple example, consider two parallel wires. Each wire has the properties mentioned before, and is represented in figure 2. In order for the pulses to travel through the conductor with minimal losses in signal quality, each end of the cable must be terminated properly. Termination of the

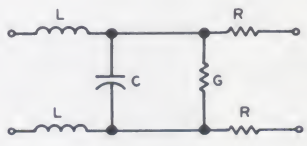


Figure 2: Symbolic representation of parallel wire transmission. The system is symmetric, so it does not matter whether the left terminals are at the source and the right terminals are at the receiver or vice versa. The symbols used in the diagram are as follows:

L, inductance of the wire.
R, resistance of the wire.
C, capacitance between the wires.
G, high resistance leakage path between the wires.

Photo 1: Coaxial cable consists of a central conductor, an outer conductive braid, and a protective coating. It is bulky and expensive, but it has good characteristics as a transmission line for data.

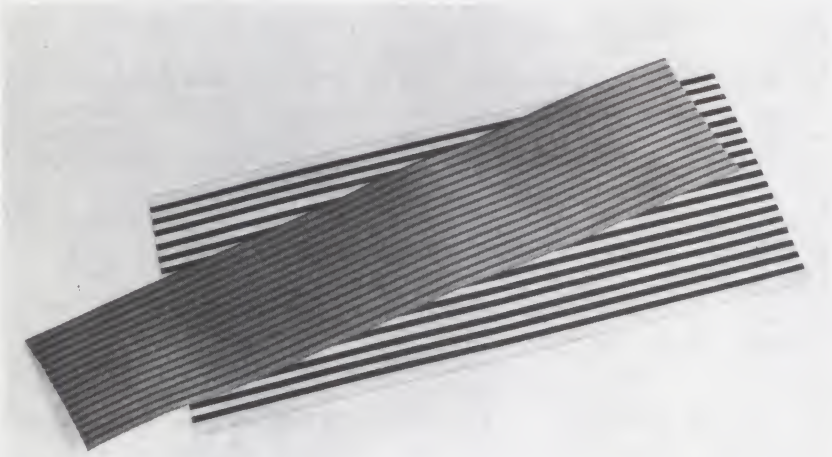


Photo 2: Two examples of ribbon cable. The lower example is a surplus item consisting of flat copper conductors (similar to PC lamination) embedded in a plastic carrier. The upper example is a more conventional cable intended for assembling to a special dual in line package (DIP) plug.

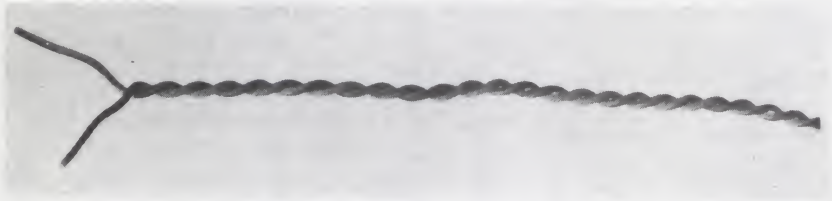


Photo 3: Twisted pair cable is the only good data transmission line which can be easily fabricated at home. Here is an example made using an electric drill to do the twisting.

line involves matching the *characteristic impedance* of the wire with the *impedance* of receiver and transmitter.

As a pulse is sent to the other end, the energy of the pulse is dissipated by the termination of the wire. If the wire is not terminated properly, a *reflection* of the pulse will travel back to the source, and a condition called *ringing* will occur.

It is for this reason that flip flops should never be used to directly drive a line of significant distance. Ringing or noise spikes could occur on the line and enter the flip flop circuit and change its state.

Typical 7400 gates have an impedance of 100 O in the high state and nearly 0 O in the low state. Almost all newer small scale TTL integrated circuits are diode clamped, preventing most ringing on the inputs. This allows wires to go about 5 feet between gates without using external impedance matching techniques, and assumes a relatively high speed and constant impedance line. If a standard TTL gate is used as the transmitter in a data link, fan out rules must be observed to supply sufficient current. To raise the output voltage of the pulse, a 2.2 kO resistor can be connected between the output and the 5 V source. This pullup resistor raises the output pulse to a full 5 V and reduces the chances of noise affecting the line.

For longer runs at high speeds, a TTL line driver chip like the 74128 can be used to provide more current to the line. For even longer runs or in critical applications special chips like the Signetics 8T13 and 8T16 are used as drivers and receivers to insure a low error rate. The maximum length for these transmissions depends on the type of wire being used.

Coaxial cable is one of the best cables to use for long distance transmission of digital data. It has a center conductor set in a non conductor with a metallic braid or foil (the shield) wrapped around it. The shield is used as the ground return and for protection from external noise. Photo 1 illustrates a typical coaxial cable, cut so its construction can be seen. Cables with a nominal characteristic impedance of around 100 O are normally used in order to match gate terminations. Coaxial cable has the disadvantage of being inflexible and bulky, especially if many wires have to be terminated in a small area. An even worse disadvantage is its high cost. Coax is usually used when other wires aren't suitable.

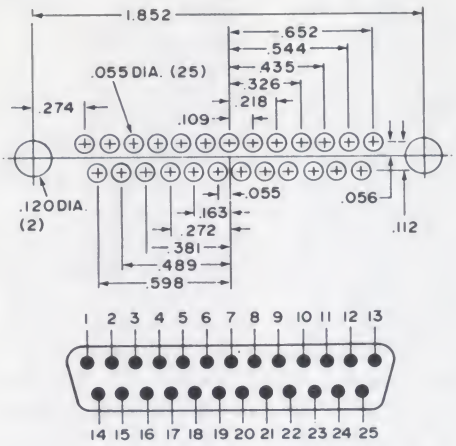
Flat ribbon cable, which usually has every other conductor grounded, provides a constant impedance and reduces the chance of wires inducing signals on each other. Ribbon cable for the hobbyist is still rather expensive, and special connectors generally must be used. Photo 2 shows two types of ribbon cable: flexible copper strips in plastic, and multiple stranded wires.

Twisted pair wire is the most cost effective transmission line for long runs in hobby systems. This kind of cable can be fabricated at home using an electric drill. In multipair cable, each pair should be used as a single signal path, with one wire grounded. The rise time characteristics of the pair are determined by the conductor size and tightness of the twist. For a 100 O cable, the wire should be 22 to 24 AWG (stranded) with about three turns to the inch. Multipair wire is available at many surplus houses, and is generally a bargain. Photo 3 illustrates a typical home made twisted pair.

Good old hookup wire is the most susceptible to noise and usually has a highly

A well grounded case . . . will help shield you from your neighbor's complaints about interference with his television reception.

When wires get long enough to look like transmission lines, termination and impedance matching become important.



RS-232 PIN ASSIGNMENTS

Pin	Name	Function
1	FG	Frame Ground (not switched)
2	TD	Transmit Data
3	RD	Receive Data
4	TRS	Request To Send
5	CTS	Clear To Send
6	DSR	Data Set Ready
7	SG	Signal Ground
8	DCD	Data Carrier Detect
9		Positive DC Test Voltage
10		Negative DC Test Voltage
11		Unassigned
12	(S)DCD	Secondary Data Carrier Detect
13	(S)CTS	Secondary Clear To Send
14	(S)TD	Secondary Transmit Data
15	TC	Transmit Clock
16	(S)RD	Secondary Receive Data
17	RC	Receive Clock
18		Receiver Dibit Clock
19	(S)RTS	Secondary Request To Send
20	DTR	Data Terminal Ready
21	SQ	Signal Quality Detect
22	RI	Ring Indicator
23		Data Rate Select
24	ETC	External Transmit Clock
25		Busy

Figure 3: The commonly used RS-232 electrical interconnection for data transmission is shown here in the form of pin assignments for the typical D connector. A typical part number for the connector is DB-25P (plug) and DB-25S (socket) made by Cinch.

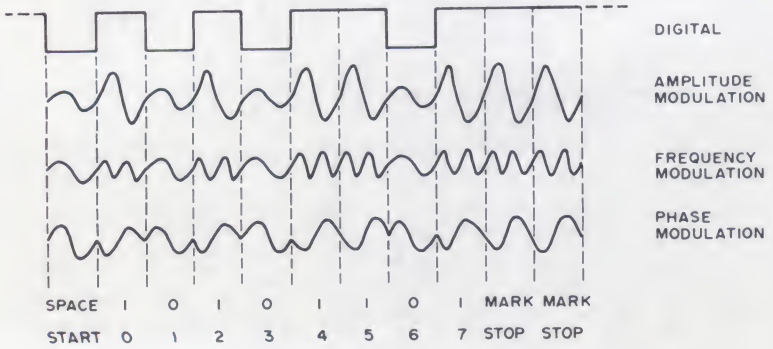


Figure 4: Two level asynchronous modulation, shown for the ASCII character "5" along with typical modulated waveforms for different methods of modulation.

unpredictable characteristic impedance. Hookup wire should be used only for short runs between boards and as on board jumpers.

The Long Haul

At distances greater than your next door neighbor's house, it begins to become impractical to use your own wire between systems. This is primarily due to a very important conductor property – cost. Copper is not cheap, and stringing wire all over the place will give hobbyists a bad name. However, the phone company has already done this and will provide service to you if you pay the price.

Another alternative is using a different medium, like radio waves. It will be interesting to see how many people suddenly want FCC ham licenses in order to play interactive TV games and exchange software.

Both of these methods have a major drawback; they will not directly pass a digital signal. The telephone system was designed to carry audio frequencies. Phones just cannot carry DC level signals. Radio frequencies are at the other end of the spectrum and certainly won't support a usable DC coupled logic signal.

However, the pulses of digital transmission can be superimposed on an AC signal that is within the bandwidth of the channel being used. The process of superimposing one signal on another is called *modulation*. A device that will translate the digital bit stream to an encoded analog signal for transmission and analog to digital for reception is called a *mod*ulator-*dem*odulator, or *modem*.

Since it is impractical to have eight telephone lines or eight radio frequencies transmit in parallel, a conversion to serial must be done. This can be accomplished by writing the conversion in software, but is more cost effective by using an integrated circuit called a UART (Universal Asynchronous Receiver-Transmitter). This chip takes the data in parallel form and converts it to serial at a rate specified by external components. For a more detailed discussion of the serial interface and UARTs, see "The Serial Interface" by Don Lancaster in the September 1975 issue of BYTE.

There is a standard for interfacing serial data transmission between peripherals, systems, and modems. The Electronic Industries Association (EIA) of America has, by consent of various manufacturers and users, standardized a 25 pin connection with appropriate signal levels called the RS-232 interface. There is also a new EIA standard that has been introduced, called the RS-422 standard that is more suitable to TTL.

However, since almost all modems and peripherals now available and all surplus items are likely to use the RS-232 standard, it will remain the most significant to hobbyists for some time.

A commonly used RS-232 connector along with pin assignments is shown in figure 3. The signals that appear on the signal pins must be bipolar with 3 V to 25 V representing a logical one and −3 V to −25 V being a logical zero. There are chips available to do this conversion from TTL, the 1488 and 1489. If you are going to use commercial RS-232 equipment, you should expect to provide this interface; and you'll also need the positive and negative supply voltages. Keep in mind that in common practice many of the pins in figure 3 are not used. The most important lines on the RS-232 interface plug are the grounds, the transmit data (TD) and receive data (RD). In many instances it is sufficient to use only these lines, especially if you are just experimenting with an RS-232 peripheral. Note, however, that some terminals require inputs for one or more additional pins, many of which can simply be wired to the RS-232 logic zero or logic one lines (−12 V or +12 V, for example).

Data Modulation

There are three basic techniques for modulating an analog signal and many different variations of these. An analog signal that is to be modulated by the data is called a *carrier*, and the carrier has three basic characteristics that can be varied. If the amplitude, frequency, or phase is varied in step with the bit stream, modulation occurs. An example of each technique is shown in figure 4.

Amplitude modulation is seldom used in modems because of its high susceptibility to noise and attenuation, but the technique is used in some magnetic tape encoding schemes. Frequency modulation is a more common technique. The example shown in figure 4 uses one frequency to represent a 0 bit and a higher frequency to represent a 1 bit. This particular method is called *frequency shift keying* (FSK).

The phase modulation example in figure 4 shows a two level coding scheme with each 180° phase shift triggering a logical state change.

The number of times the signal is varied each second is called *Baud* or *Baud rate*. Suppose you were designing a modem using phase modulation. You could, for example, divide the possible phase shifts into 45° each, having eight possible phase shifts for each signal change. A 45° shift would represent a group of three bits, namely '000'. 90° would represent '001', 135° = '010', 180° = '011', etc., up to 360° for '111'. In this case a signal changing 100 times a second, or at 100 Baud, would actually transfer data at 300 b/s. This method of one signal change representing more than one bit is called *multilevel encoding*. It is in principle the way some commercial high speed modems function.

In order to obtain a fair amount of accuracy in transmitting data from a transmitter to a receiver, it is necessary to keep the two systems in step with each other. There are two common methods to do this, called *synchronous* and *asynchronous* transmission.

Asynchronous transmission is also called start-stop transmission because each character is sent as it is created at the transmission interface. To synchronize the receiver, each character carries its own timing in the form of additional bits called start and stop bits. These give the receiver the ability to decode each bit reliably. The format of a single character is shown in figure 4 along with data for the digit 5 encoded in ASCII.

Synchronous transmission is usually associated with blocks of data, where groups of characters are sent together. A fixed speed of transmission is set by clocks or oscillators, and data bits are transferred at this rate. To provide character synchronization, usually two special synchronization characters precede the actual block of data. An end of block character follows the data to signal the receiver that all of the data in that block has been sent. An additional error detecting character may also be sent as shown in figure 5.

A baud is not necessarily a bit per second – although it might be in special cases. A baud is a signal change per second.

Figure 5: Synchronous transmission is block oriented and assumes highly accurate common clocking of both the sender and the receiver.

Finally, there is one other complication in using radio or telephones for your communication channel – the FCC. The FCC regulates the telephone industry by tariffs that specify the costs and types of devices that can be used with Ma Bell. They also regulate radio frequency allocation and power output and information codes that can be transmitted by radio.

There are good reasons for having these regulations followed, but they do tend to make life difficult for the hobbyist. A dedicated 100 mile phone line with a Bell modem will exceed most other system costs

for a home computer in a very short time. A ham license and suitable radio equipment are not cheap, either. Long distance real time data transmission is presently out of reach to many hobbyists.■

GLOSSARY

Asynchronous transmission: Transmission where data is sent a character at a time with synchronizing bits added. See Synchronous Transmission.

b/s: Bits per second.

Bandwidth: The width of the frequency spectrum that a channel can pass, measured in Hz.

Baud: The number of signal changes per second.

Carrier: The analog signal that is modulated by the information that it is to carry. Also, the provider of a communications channel, i.e., Ma Bell.

Duplex: Communication system that allows information to be exchanged. See simplex.

FSK: Frequency shift keying, a type of frequency modulation for digital data.

Full duplex: Communication system that allows simultaneous information exchange. See half duplex.

Half duplex: Communication system that allows information to be exchanged, but not simultaneously. See full duplex.

Modem: Modulator-demodulator. A device used to convert digital to analog signals and vice versa.

Modulation: The process of superimposing information on a carrier. See **carrier**.

Multilevel Encoding: The process of using a signal change to represent more than one bit of information.

Noise: Unwanted signals that interfere with the message.

Ringing: A condition in transmission lines where "reflections" of pulses oscillate due to an impedance mismatch.

Simplex: Communication system that does not allow information to be exchanged. See duplex.

Synchronous Transmission: Transmission where the bit rate is clocked. Usually associated with block transmission. See asynchronous transmission.

William A Walde
28 Tuttle Dr
Acton MA 01720

Build a TTL Pulse Catcher

While checking out the operation of some oneshots on the address latch board during the construction of my Mark-8 microprocessor, I discovered that my first homemade logic probe could not detect very short TTL logic pulses. Since my old probe would not work, I needed a quick and easy way to tell whether a short TTL pulse had arrived. I dreamt up this circuit which solved the problem by adding memory in the form of an RS flip flop wired from a NAND gate. This pulse catcher will detect pulses as short as the combined gate delays of the two NAND sections used to form the flip flop, approximately 10 to 30 ns. The circuit works by changing state at the start of a pulse, with an LED monitoring the flip flop output. There is no indication of the end of the pulse and there is no way to tell if multiple pulses have occurred. After a pulse has been detected, the circuit must be reset in order to detect the next pulse. The TTL pulse catcher is designed to work with either a positive or a negative going pulse as selected by a switch.

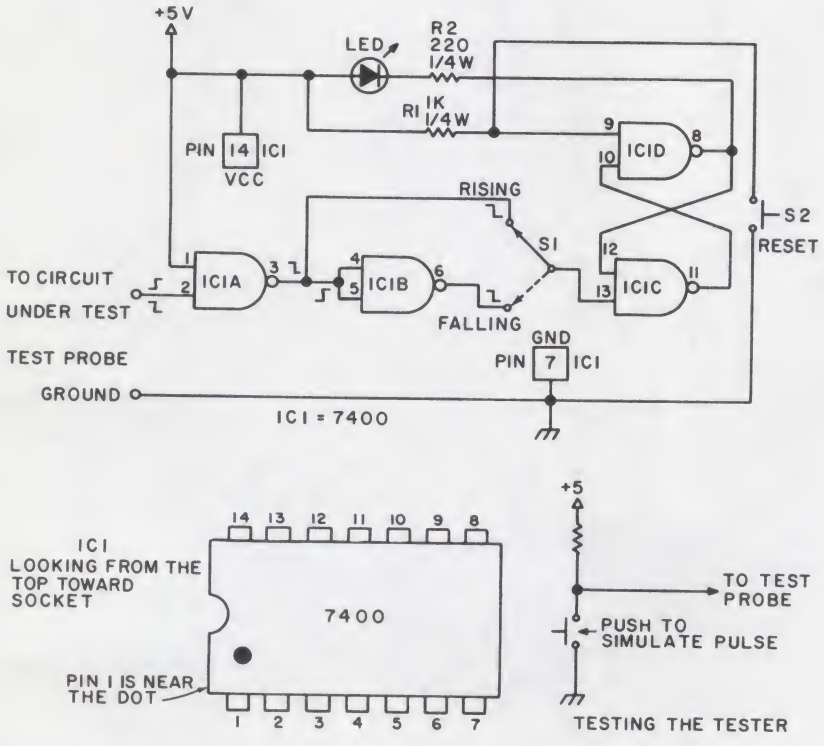


Figure 1: The circuit diagram and parts list for the TTL pulse catcher design.

Construction

The original version of this test instrument was built using perforated board and flea clips. The circuit could also be made using a small printed circuit board. Lead dress is not critical, but the polarity of the LED must be observed. Select a value of the resistor R2 in the range 47 to 470 O such that good illumination is achieved with a current of approximately 20 mA. A typical resistance value is 220 O. The tester can be tested using a resistor, a bounceless push-button switch and the test circuit shown in figure 1. Label switch S1 to identify which setting is the rising edge mode and which setting is the falling edge mode. A clever approach would be to build the pulse catcher inside a cylindrical enclosure, such as a pen or thin tube.

Using the Pulse Catcher

Connect the 5 VDC and ground terminals to a suitable power supply. If you use a power supply separate from the main system supply of the computer or logic device you

are testing, be sure to tie the grounds of the two supplies together. Connect the input probe to the line being tested and select the rising or falling edge mode via switch S1. Depress the reset button to extinguish the LED and arm the latch. If you are unable to cause the LED to stay off after releasing the reset, the following information is learned about the line being tested:

It may have a steady state condition which is inconsistent with the mode of the test. Change S1 and try again.

You may be observing a line which has regular clock transitions. The light will never go out when testing such a line.

In normal operation, once the pulse catcher is reset, the LED will remain out until the first pulse comes along, after which it will stay on until reset by pushing the S2 button.

Theory of Operation

Gates IC1c and IC1d are cross-coupled to form the familiar RS flip flop that is used to remember when a pulse has occurred. Momentarily depressing SW2 grounds pin 9 of gate IC1d, causing the flip flop to go into the reset mode. In this mode both sides of LED1 are high and it will not light up. In a similar manner, falling pulse at pin 13 of gate IC1c will cause the flip flop to go into the set mode with gate IC1d now becoming a current sink for LED1, causing it to illuminate.

Switch SW1 is used to select between either a rising or a falling input pulse. The RS flip flop always needs a falling pulse to operate. In the case of a rising pulse, it is inverted by gate IC1a, which is also used as a buffer. Gate IC1b is used to invert a falling pulse a second time to put the pulse back into its original form. Note that pin 1 of gate IC1a is held high so that the input (pin 2) will only present one standard TTL load to the circuit under test (as compared to gate IC1b where the 2 inputs are tied together).

R1 is used as a pull up resistor to the pin 9 input to gate IC1d for noise immunity. Resistor R2 is used as a current limiting resistor for LED1.■

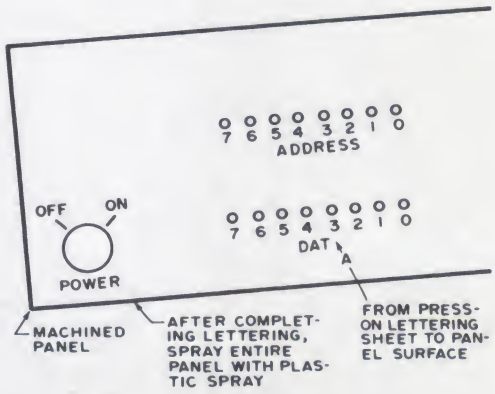
Dressing up Front Panels

Don R. Walters
3505 Edgewood Dr
Ann Arbor MI 48104

To dress up panels of equipment use press on lettering (available from stationery shops, college and university book stores, and from graphics arts supply shops) to label the various functions performed by the components which will be mounted on the panel (see figure 1). The lettering can be pressed on, wire brushed, chemically etched, or painted surfaces. The only caution is that the surface be free of dirt and grease before applying the lettering to the panel.

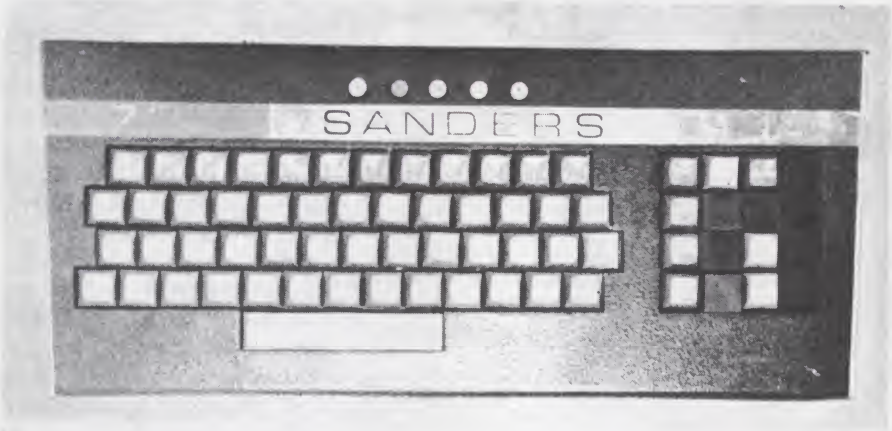
When applying the lettering, if a mistake is made the mistake can be removed by gently scraping the lettering off the panel.

After all the lettering is completed and you are satisfied with the job, lay the panel on a flat surface and carefully spray the panel with several (5 to 10) light coats of a clear plastic spray. Allow the surface to dry before applying the next coat. Also allow the sprayed surface to dry completely (overnight) before installing the components on the panel. The plastic spray protects the lettering from being rubbed or scraped off easily.



A word of caution; let the spray dry thoroughly before mounting the components. Secondly, on each spraying put down only a light coat. Otherwise the letters will tend to move (float?) a little out of place. Also be careful when mounting components onto the panel so that the lettering and/or the panel's surface is not scratched. The end result of lettering and care will be a very good looking piece of equipment.

DECIPHERING



MYSTERY KEYBOARDS

Did you ever wonder about the use of surplus keyboards for use in your system? Here is an article describing one way to analyze such a keyboard – illustrated by a particular model which is available through one of BYTE's advertisers. Do you use a surplus keyboard already? This is one of the most common and usable of surplus subsystems – I'd like to see a few reader submitted articles on use of various keyboards available in surplus channels. . . . CARL

by
Carl Helmers
Editor, BYTE

One of the best sources of input data for your home brew computer system is the typewriter style keyboard device. A decent keyboard will give you the ability to enter parallel character data 8 bits at a time. The typical keyboard input devices will also include a flag of some sort to indicate that a key has been pressed. It might also include an "acknowledge" line to be pulsed after the computer had read the data. The parallel interface of a typical keyboard is illustrated in Fig. 1. Fig. 1 is a typical interface of a keyboard, and is used only as a guide to the analysis of an actual keyboard later on in this article.

The manual input of the keyboard is its most important feature. It is the human operator's depression of a selected key which communicates some information to your system. When the key is depressed, it causes the keyboard input device's logic to generate an encoded binary pattern for the key. This encoded binary pattern is typically an ASCII character code presented on the data lines D0 to D6. In addition to the encoding function, the keyboard has logic which produces a "flag" signal to indicate that some key has been depressed. This flag is either a pulse (see timing diagram example in Fig. 1) or a level state, depending upon the particular keyboard design involved. It is often the case (but not required) that the keyboard is designed for interactive control by the computer processor. In such cases, an "acknowledge"

signal must be generated by the computer and sent back to the keyboard to reset the logic of the keyboard input device.

The encoding pattern of the keyboard input device depends upon the manufacturer's design and must be determined for a surplus keyboard before you can use it. For many keyboards, the ASCII pattern of Table I is applicable – each key maps into one of the 7-bit patterns listed. Unless stated by the dealer, you will have to approach the analysis of the surplus keyboard without any assumptions: it is likely to be ASCII but... you could wind up with a Univac "Fieldata" encoded keyboard; you could wind up with an IBM EBCDIC keyboard, etc. Many non-standard encoding schemes for alphanumeric keyboards are derivatives of ASCII. Thus the example in this article is chosen with an ASCII encoding scheme in mind. (IBM surplus is rarely in usable form and the number of EBCDIC keyboards by non-IBM manufacturers is an unknown but assumed small number.) In Table I, the common character codes are shown in a typical graphic form as well as in binary, octal and hexadecimal representations.

Now a new keyboard fully encoded for ASCII and/or EBCDIC is one option you have for implementing a keyboard input device. For example, a new commercial keyboard will typically sell in the $50 to $150 range depending upon options – a keyboard with a standard typewriter style layout and an LSI encoding method. As a second example, Southwest Technical Products Corp. used to sell a hobby quality keyboard at about $40 in kit form. The advantages of new keyboards are obvious: you get the complete description of the hardware along with the product – and an interface which will be similar to the one described in Fig. 1. With the newer LSI encoded boards, you will probably get a keyboard with an "n" key rollover feature to decipher multiple key strokes which overlap in quick succession. This is all well and good, but is there a less expensive alternative? The answer of course is "Yes", and the remainder of this article concerns the techniques involved.

Using Surplus Keyboards

The alternative to new equipment is "pre-owned" equipment, to borrow a term from standard used car dealers' lexicon. Since computers have been in use for a number of years there is a fairly wide selection of equipment in the "surplus" market, as you can find out by reading the advertising pages of BYTE. An item which is frequently found in surplus vendors' offerings is the keyboard input device. Prices for keyboards vary considerably – from $10 for real "junk" to about $40 for premium keyboards. The use you can get out of such a surplus board ranges from a complete subsystem ready to hook up – to a mere array of key switches which must have a new set of encoding logic to make it work.

The keyboards you employ for this purpose must be selected and analyzed on an individual basis – there is no stock formula applicable to all such keyboards. Several rough guidelines will help you keep out of too much trouble:

1. Always look for a unit which is in sound physical condition. Get one which has the cleanest possible key tops, smoothly working keys, little sign of "hack" modifications to PC circuits, etc. Verify that the keyboard is a "switch" type – Hall effect or capacitive keyboards exist and should be avoided without proper documentation.

2. The most desirable keyboard will be one in which the encoding logic is readily decipherable. This will invariably be the case with diode matrix keyboards (see text below) – and may be possible if an LSI chip with a standard part number is utilized.

3. The most desirable keyboard will be one on which the PC layout people have made notations of nice little comments like "+5V", "-12V", "VCC", "A", "$", etc. These are great aids to figuring out the operation of the devices.

If you (at a minimum) satisfy the first criterion above, the keyboard will ultimately be usable, provided it uses actual keyswitches, since you can always construct a switch scanner and/or diode matrix to encode the switches as ASCII binary information.

Diode Matrix Keyboard Analysis – An Example

To illustrate what can be done with surplus keyboards, the remainder of this article concerns the analysis of a particular keyboard input device. The keyboard in question has been advertised recently, and is a fairly typical diode matrix encoded

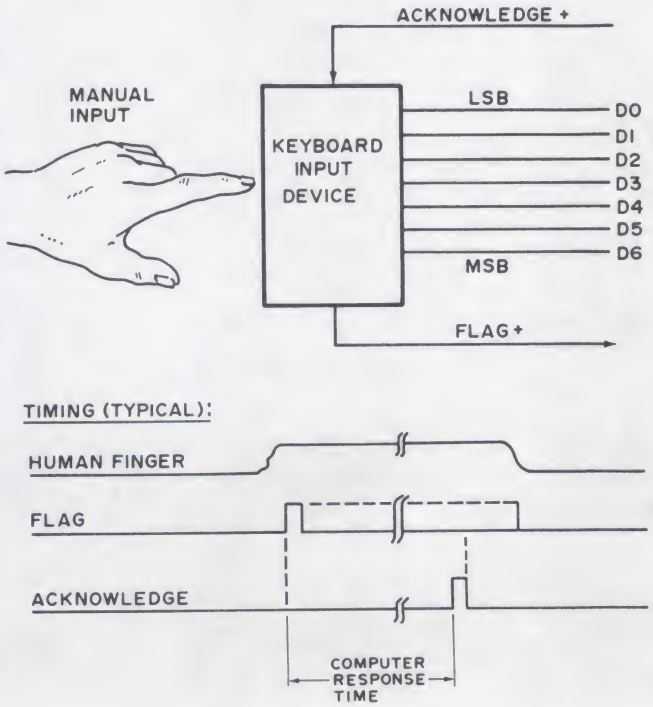
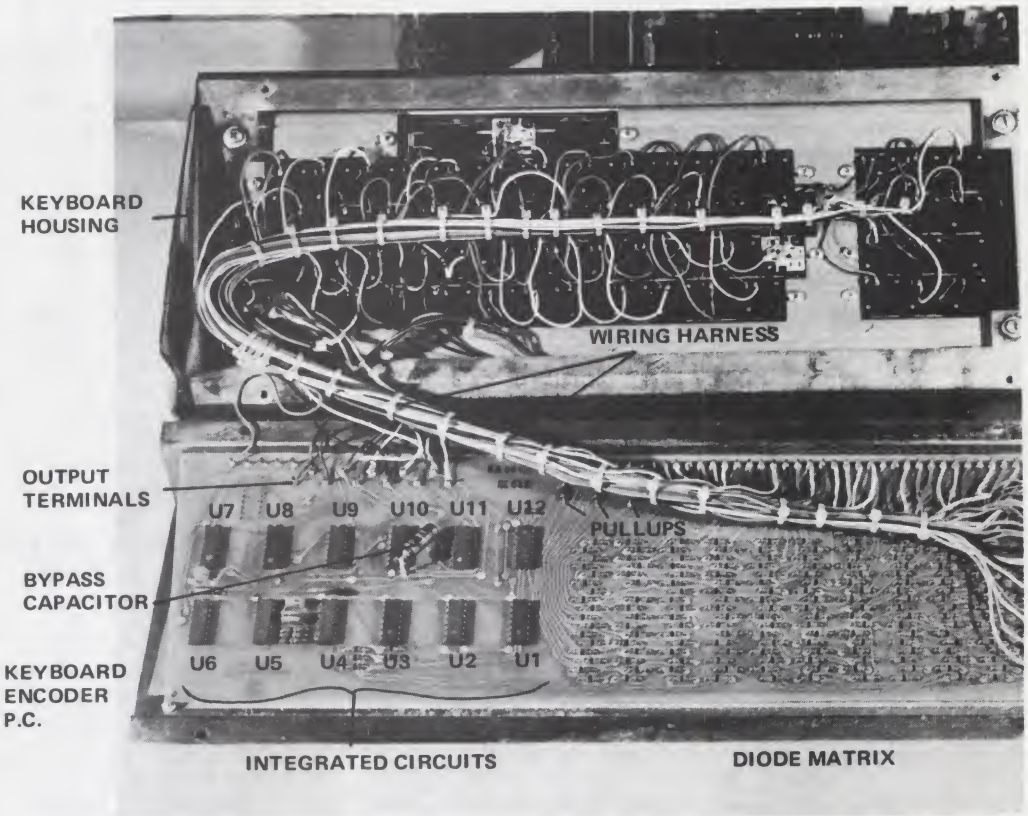


Fig. 1. Typical keyboard functions.



The keyboard, with bottom plate removed and encoder board out in the open. The encoder printed circuit is separated from its mounting on the bottom plate but is still attached by its wiring harness.

keyboard of the 1966-1970 vintage. This keyboard is a surplus Sanders Associates Model 722-1 subsystem, which comes enclosed in a metal housing with a fairly typical Teletype style key layout. On the right hand side of the keyboard is a set of special function keys, which obviously had some meaning in the original system using the device.

The keyboard and housing can be used "as is" in your system — with the only necessary modifications being the substitution of an interface plug and cable which can mate with your own equipment. The example of analyzing and figuring out this keyboard can be used as a guide to similar work with other surplus keyboards.

Start at the Beginning

The object of this project is to determine the details needed to make the Model 722-1 keyboard work — but without any original design documentation from the manufacturer, since it is surplus. The first step is to put on your Sherlock Holmes cap, crank up your deductive powers and begin disassembling the keyboard. In order to analyze the circuit, a likely place to start is the bottom cover plate. In the case of the 722-1, four screws hold the cover plate to the bottom of the housing. Upon opening the cover plate, the 722-1 will be found to have a printed circuit board attached to the plate — a thin plastic sheet glued to the cover plate prevents inadvertent shorting of PC conductors. The PC should be removed from the cover plate by unscrewing the four nuts securing it. The result will be a PC board hanging out the back of the housing/keyboard assembly by its wiring harness.

The actual process of analysis of a keyboard such as this will probably take you an evening or so. The key features to look for in a diode matrix encoder keyboard are identified in the photo.

Keyboard Encoder PC. The typical diode matrix keyboard will have a printed circuit board containing a large number (approximately 100-200) of computer diodes and several integrated circuits, with individual wires running from keyswitches to the PC. Sometimes the functions of encoding and control logic will all be mounted on the same printed circuit as in this example. Occasionally, the logic will be split up into smaller chunks on separate boards.

Wiring Harness. A keyboard is easy to figure out if you can get at it "live" (under power). In this case, a wiring harness allows considerable room for extension so that the key switch matrix and housing can be separated from the encoder board.

Diode Matrix. The way to tell a diode matrix board is by the regular array of diodes found at some point. In this example, the array is at the lower right in the photo. While the array is regular, the actual printed wiring is fairly random — although it will ultimately condense down into a set of bit busses.

Integrated Circuits. This particular keyboard has a bunch of integrated circuits in the left hand portion of the encoder board. The photo illustrates arbitrary reference numbers U1 to U12 for the purposes of this article, since no references were built into the printed circuit board.

Pullup Resistors. In diode matrix boards, a set of negative logic "wired or" busses is used to generate each bit of the encoded binary word. One pullup resistor (typically 1000 Ohms) is associated with each bus line.

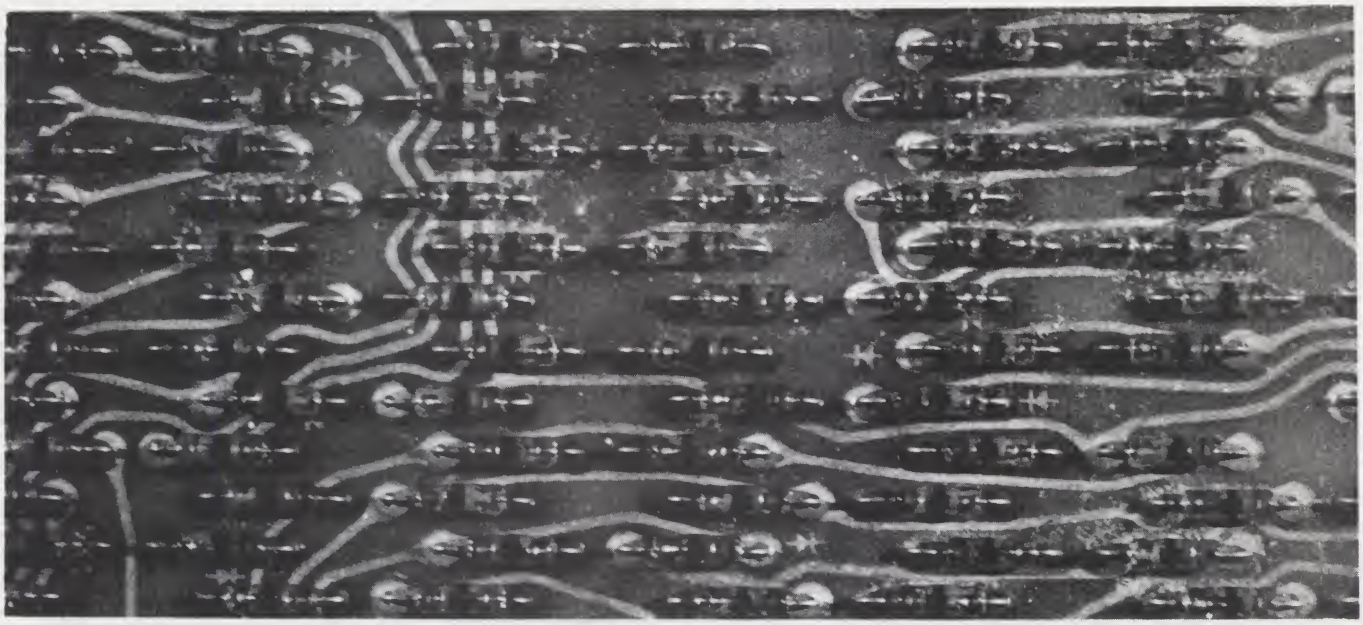
Identifying the Power Requirements

One of the most critical items to be determined in figuring out a keyboard is to identify the power requirements. The best way is

Table I. Binary, Octal and Hexadecimal ASCII Codes. This table contains common symbols for keyboard characters and the corresponding ASCII codes.

Binary	Octal	Hex	Common "Graphics"*
0000000	000	00	NUL character
0000001	001	01	
0000010	002	02	
0000011	003	03	
0000100	004	04	
0000101	005	05	
0000110	006	06	
0000111	007	07	Bell — Ring the Bell!
0001000	010	08	
0001001	011	09	
0001010	012	0A	LF — Line Feed
0001011	013	0B	
0001100	014	0C	
0001101	015	0D	CR — Carriage Return
0001110	016	0E	
0001111	017	0F	
0010000	020	10	
0010001	021	11	
0010010	022	12	
0010011	023	13	
0010100	024	14	
0010101	025	15	
0010110	026	16	
0010111	027	17	
0011000	030	18	
0011001	031	19	
0011010	032	1A	
0011011	033	1B	ESC — "Escape"
0011100	034	1C	
0011101	035	1D	
0011110	036	1E	
0011111	037	1F	
0100000	040	20	SP — Space
0100001	041	21	! — Exclamation
0100010	042	22	" — Quotes
0100011	043	23	# — Number Sign
0100100	044	24	$ — Dollar Sign
0100101	045	25	% — Percent
0100110	046	26	& — Ampersand
0100111	047	27	' — Apostrophe
0101000	050	28	(— Left Paren
0101001	051	29	) — Right Paren
0101010	052	2A	* — Asterisk
0101011	053	2B	+ — Plus sign
0101100	054	2C	, — Comma
0101101	055	2D	- Minus Sign (hyphen)
0101110	056	2E	. — Decimal (period)
0101111	057	2F	/ — Slash
0110000	060	30	0
0110001	061	31	1
0110010	062	32	2
0110011	063	33	3
0110100	064	34	4
0110101	065	35	5
0110110	066	36	6
0110111	067	37	7
0111000	070	38	8
0111001	071	39	9
0111010	072	3A	: — Colon
0111011	073	3B	; — Semicolon
0111100	074	3C	< — Less than
0111101	075	3D	= — Equality
0111110	076	3E	> — Greater than
0111111	C77	3F	? — Question Mark

Binary	Octal	Hex	Common "Graphics"*
1000000	100	40	@ — "at"
1000001	101	41	A
1000010	102	42	B
1000011	103	43	C
1000100	104	44	D
1000101	105	45	E
1000110	106	46	F
1000111	107	47	G
1001000	110	48	H
1001001	111	49	I
1001010	112	4A	J
1001011	113	4B	K
1001100	114	4C	L
1001101	115	4D	M
1001110	116	4E	N
1001111	117	4F	O
1010000	120	50	P
1010001	121	51	Q
1010010	122	52	R
1010011	123	53	S
1010100	124	54	T
1010101	125	55	U
1010110	126	56	V
1010111	127	57	W
1011000	130	58	X
1011001	131	59	Y
1011010	132	5A	Z
1011011	133	5B	[— Left bracket
1011100	134	5C	\ — Reverse slash
1011101	135	5D	] — Right bracket
1011110	136	5E	
1011111	137	5F	— Underscore
1100000	140	60	
1100001	141	61	a
1100010	142	62	b
1100011	143	63	c
1100100	144	64	d
1100101	145	65	e
1100110	146	66	f
1100111	147	67	g
1101000	150	68	h
1101001	151	69	i
1101010	152	6A	j
1101011	153	6B	k
1101100	154	6C	l
1001101	155	6D	m
1101110	156	6E	n
1101111	157	6F	o
1110000	160	70	p
1110001	161	71	q
1110010	162	72	r
1110011	163	73	s
1110100	164	74	t
1110101	165	75	u
1110110	166	76	v
1110111	167	77	w
1111000	170	78	x
1111001	171	79	y
1111010	172	7A	z
1111011	173	7B	{ — Left brace
1111100	174	7C	
1111101	175	7D	} — Right brace
1111110	176	7E	≈
1111111	177	7F	DEL — Delete



Diode matrix – this system of generating the ASCII code is used in older keyboards.

of course to get a keyboard which has power requirements listed on its encoder printed circuit in no uncertain terms. However, "the best" is often a matter of luck and judicious choice of equipment in surplus circles . . . you can make do with less than perfect documentation by employing some knowledge of common design practices. Figuring out power voltages requires the analysis of one circuit power line for each level of voltage involved to completely establish the requirements of the system.

One of the least ambiguous ways to identify power lines is to look up the power pinouts of the integrated circuit components used in your keyboard. This method requires a supply of reference books and a keyboard encoder circuit which uses standard part numbers. For keyboards which are manufactured by the smaller companies in the computer field, parts are usually standard items so that this method can be employed. One of the main justifications for home brew computer clubs is the nice informal arrangement which provides for an exchange of information of this type. In the case of the Sanders keyboard, the two integrated circuit designs used were labelled "ST659A" and "ST680A". The only problem is that no direct reference could be found in literature I had available. However, don't give up with an initial failure to find a reference. What I did after striking out on these two numbers was to look for a similar number differing only in the alphabetical information. I did find references to two DTL integrated circuits "SP659A" and "SP680A," an expandable 4-input NAND gate and a quad 2-input NAND gate. Both these gate designs have package power connections of Pin 8 for power and Pin 1 for ground.

The gate references gave me a high probability determination of the power connections by tracing down ground to the I/0 pin labelled 7 and tracing down power (+5 for DTL) to I/0 pin 5. Being a cautious type of person, I then looked for some independent confirmations of this power pinout identification.

Another method of identifying power and ground connections is to look for color coding on wires. This kind of a confirmation is only possible for boards manufactured with hand wiring. If the harness is one of the multiconductor ribbon cables, color coding is not likely. In the keyboard I analyzed I found that the ground terminal of the decoder was routed via a black wire to the connector on the case, and that the +5 volt terminal was *routed to* the connector via a red wire. This is consistent with the industry conventions which are used for such wiring – power (positive) is red, ground (negative) is black.

Still another method for determination of power connections is to examine the polarity of electrolytic capacitors mounted on the board for local power supply filtering. These bypass capacitors are often (not always) connected between

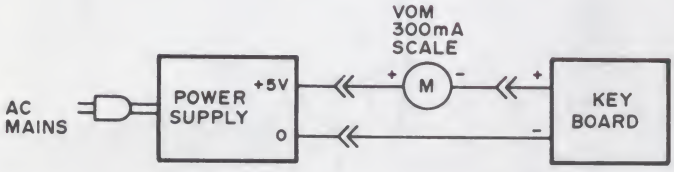
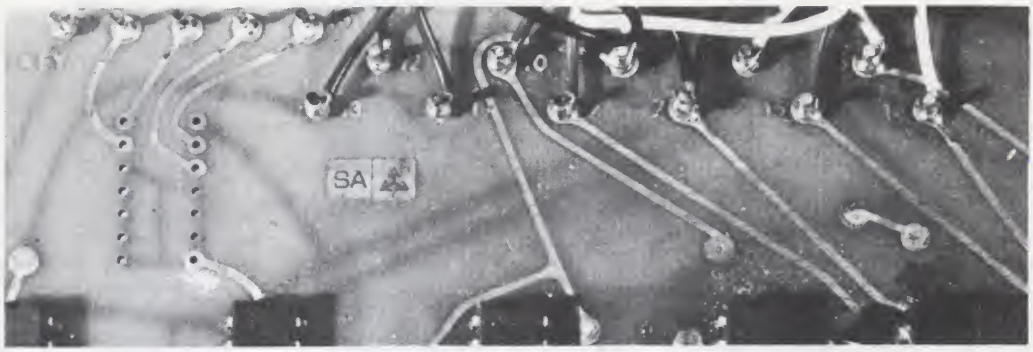


Fig. 2. Turn it on and cross your fingers.



Detail of the output pins. This keyboard is one of the more desirable types – it has labeling of many key features etched along with the printed wiring.

the positive supply and ground, with markings of (+) for the supply side and (-) for the ground side. In the disassembled keyboard photograph accompanying this article, the bypass capacitor is labeled. Using clip leads, the bypass capacitor often provides a handy way to apply power when first testing the board.

Multiple power supply keyboards often occur with later equipment, especially where MOS encoders are employed. This will complicate the analysis problem – often to the point where it might be wise to avoid such boards unless adequately labeled with voltage designations, part numbers and other comments.

Turn It On and Cross Your Fingers?

Now that you think you have the power connections straight, your next step in analysis is to apply a little bit of power to the circuit and see what happens – using a milliammeter. Connect the keyboard using the circuit of Fig. 2. If the power leads have been correctly identified, the current read on the meter should be approximately 100 milliamperes. Remove power ASAP if the meter movement is "pinned" on a 300 or 1000 milliampere scale, since that indicates either a short circuit or incorrect polarity for the power. If a reasonable current (under 300 milliamperes) is drawn, then you can safely trust your power connection determination and proceed by removing the meter from the circuit.

Does It Have a Flag?

The next thing to look for is a "flag" indicating that the keyboard has been activated by a finger and data is present. The term "flag" means a logic line generated in the keyboard encoder which may be either pulsed or steady state. This test requires a method of catching pulses – either an oscilloscope with about 10 MHz bandwidth, or one of a number of logic probes available which "flash" when a state change occurs. Check each of the several I/O connection terminals while pressing a key. If the keyboard is working at all, you will find at least one terminal which changes state – with a pulse or a level change – as keys are activated.

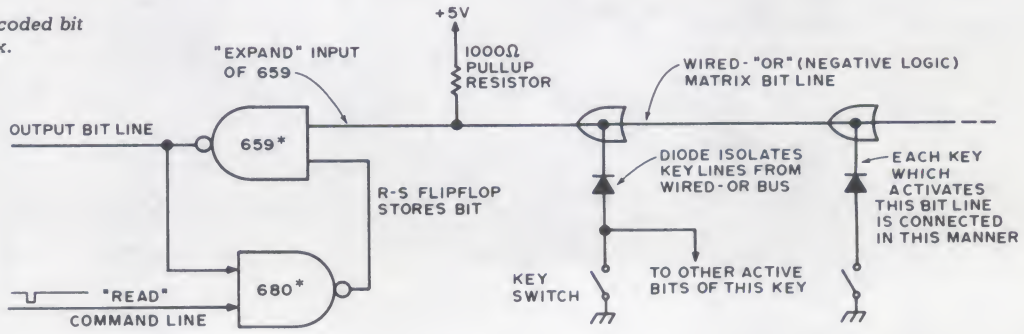
When you have found a pin which changes state, the next test is to see whether it changes the same way for every normal key on the keyboard. If the effects vary from key to key, then the line in question is a data line – if the tentatively identified "flag" pin pulses or changes its level consistently for all keys (with one or two possible exceptions) then it is probably the flag desired. In the Sanders surplus board analyzed here, the flag pin was found to be I/O connection terminal 8.

The exception possible to the "same behavior on every key" statement is evidenced in the Sanders board – the flag interconnection terminal is a pulsed output of 2 microseconds in width for all keys except one: the "Repeat" key causes the flag to change its state. The flag is normally high in this board, but when repeat is depressed it is held low.

Where's the Data?

Now, having found a flag to indicate when data is present, the next problem immediately presents itself – you now turn to examine the other pins of the interconnection to the

Fig. 3. The typical encoded bit line for a diode matrix.



**DTL gates used in surplus "Sanders 720" keyboard; TTL might be used in variations on this theme, e.g.: 7400 series.*

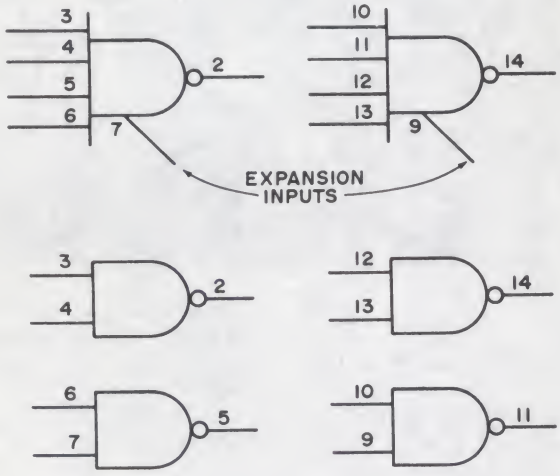


Fig. 4. Pinouts for the DTL gates in the Sanders keyboard. (Unused inputs are assumed logic 1 without external pullups.)

decoder and find no change whatsoever in levels regardless of the key pressed. Ah! The frustration! It's enough to drive you to tracing down the logic of the keyboard, at least for one of the low order data bits. That's exactly what happened in analyzing this example of a keyboard. Fig. 3 is the result of that tracing operation – using the pinouts of Fig. 4 which were obtained from an old (late sixties) data reference for the DTL gates.

As can be seen in Fig. 3, an R-S flip flop is made out of two NAND gate sections for each bit-line of the keyboard. This storage of the state of the diode matrix outputs explains the lack of change seen when first examining the board's outputs for possible data – in order to read (or get ready to read) a key, the R-S flip flops of all diode matrix outputs must be reset. The "read" command line performs this reset. After resetting, the first negative going pulse on the matrix bit line into the 659's expander input sets the flip flop, thus debouncing the contact closure. There is one bit line for each possible bit of "raw data" – and some logic is used to superimpose the shift key and control key information as required.

So, in order to find out which interface terminal corresponds to the "read" command line which resets all the flip flops, a bit more circuit tracing is required. Fig. 5 illustrates the effective logic resulting from the tracing for "Read" – which it turns out is commanded by a negative logic pulse from the computer via interconnection terminal pin 6. In Fig. 5, the R-S flip flop (A) is used to control the computer interface. The receipt of an acknowledge command from the computer resets that flip flop potentially allowing a read, but the NAND gate (B) inhibits recognition of any new keystroke until after the previous key is released. Thus this keyboard has zero-key rollover since all keys must be released before a new key can be recognized.

Figuring Out the Coding

Once the problem of locked up outputs is solved by identifying the "Acknowledge" signal line, the next problem is to identify the bit lines at the interconnection interface. To do this requires the following procedure (by hand) when testing the state of individual bit lines as keys are depressed . . .

1. Short the Acknowledge line to ground.
2. Press a key whose code is to be examined.
3. Look at the outputs on a scope or logic state indicator (the latter is an LED driven by a gate section).

To identify your coding, make the following reasonableness hypothesis initially:

Keys with an identifiable sequential order (eg: alphabetical order) will be consecutive integer numbers in any reasonable binary coding scheme.

You can identify the low order bits in ASCII, for instance, if you make this assumption.

Fig. 5. Keyboard "Read" and Acknowledge logic.

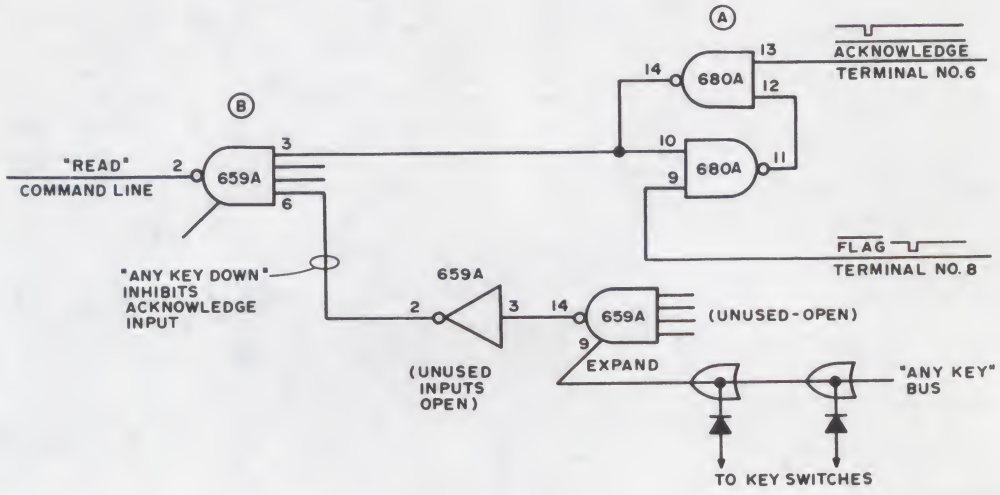


Table II. Terminal Connections for the Sanders surplus keyboard.

Terminal I.D.
#5 Power (+5 volts)
#6 Acknowledge (–)
#7 Ground
#8 Flag (–) (pulse unless REPEAT key held down)
#9 Bit 0 (+) ASCII LSB
#10 Bit 1 (+)
#11 Bit 2 (+)
#12 Bit 3 (+)
#13 Bit 4 (+)
#14 Bit 5 (+)
#15 Bit 6 (+)

So, pick two neighboring keys with identical ASCII high order bits, and test first one then the other (using the three steps above) for each potential bit line until you find a bit line which alternates with your key strokes. Thus, for instance, if you alternately press @ and A on the Sanders board of this article (acknowledging between each look) you will find the state of interface terminal 9 alternating. This can only be the low order bit of the ASCII code. Now pick two keys in alphabetical order which are at a change in bit 1. For example, pick "A" and "B". This will result in all high order bits of the code remaining identical down to the ASCII bit 1 line. Examine the terminals of the encoder while alternately looking at A and B until you find the line which changes.

This procedure can be repeated for the third ASCII low order bit (bit 2) by picking the letters C and D. The bit 2 terminal is found to be 11 by this test for the Sanders board. Continuing once more, test the bit 3 output by looking at G and H alternately (ignore the previously identified pins – all high order pins will remain the same).

By the time terminal 12 is found to be ASCII bit 3, a trend has been established for this keyboard – ascending terminal identifications from 9 are the bits of the ASCII code. In many cases this will be the order of terminals – but you have to identify several of the least significant bits first before you can make a conjecture. This conjecture of ordering can be verified for the Sanders board being analyzed by looking at typical codes (see Table I, and look, for instance, at the output for "line feed" using the terminal identifications listed in Table II).

Now a major input to this identification process is the assumption of ASCII coding – if this assumption gives "funny" results, you have no choice but to use a slightly different method: take each key in turn, depress it, and look at all possible output bits lines of the encode. Record the results in a table similar to Table I, but with the key you find, instead of the standard ASCII. You may find you have inverted data, a completely non-ASCII code set such as EBCDIC, or a modified ASCII.

Now You've Sorted the Bits – So What's Next?

When you have figured out the equivalent of Table II for your own surplus keyboard, the next step is to make a systematic identification in a table similar to Table I. One of the best ways to do this is to use your computer with an input port devoted to the keyboard, and a display or hard copy device for output. A program written to implement the flow chart of Fig. 6 can be used to selectively examine keys on the keyboard. The program accepts a key input, unpacks the bits into a binary and octal form, then displays the bits on your output (TV, character generator, or printer) as a binary and an octal number. If you have a printer output (eg: a Teletype or line printer) then you should write the symbol on the key next to each code after the code is printed. If you only have a display output, then you should note the code on paper along with the key symbol. After you have completed this bit of research, your keyboard is now thoroughly documented so that its input codes can be interpreted by programs. ■

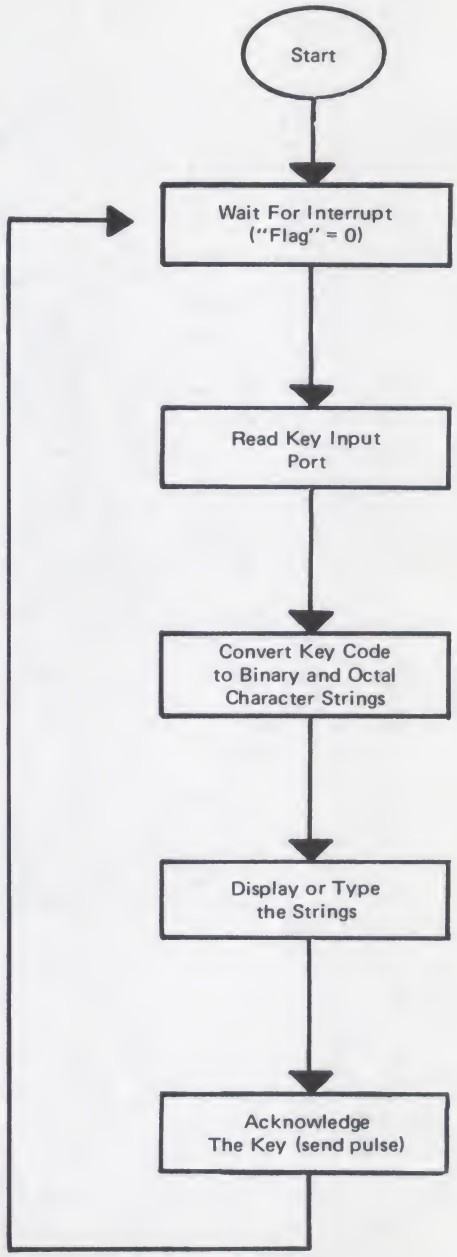


Fig. 6. Keyboard Test Program Flow Chart.

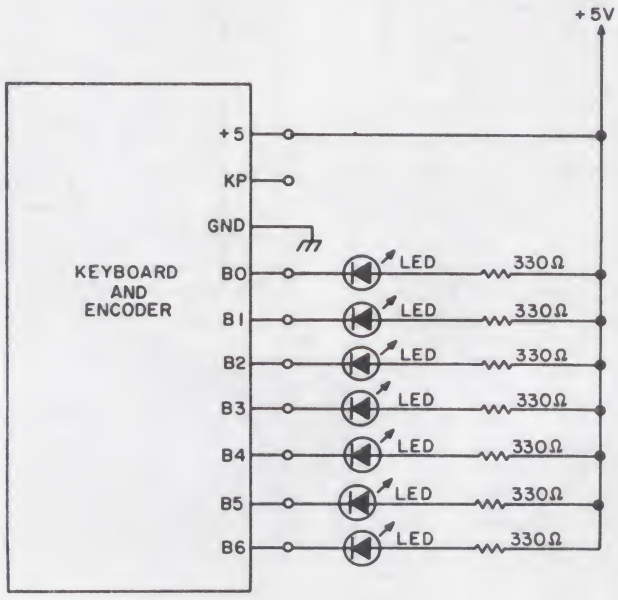
A QUICK Test of Keyboards

This indicator circuit can be used to advantage when analyzing keyboards using techniques described in BYTE #1, "Deciphering the Mystery Keyboard," page 62.

by
Don R. Walters
3505 Edgewood Dr.
Ann Arbor MI 48104

After completing the assembly of a keyboard late one night, I wanted to check the keyboard out for proper operation. So I picked up my VOM and started looking at the voltage levels on the output pins of the keyboard, since I do not have a CRT terminal or any other ASCII device available. Well, being a software type, I kinda felt a little frustrated since I am generally used to being able to see all the bits of a bit pattern at the same time. The solution was very simple, inexpensive, and quickly allowed the bit pattern on the keyboard output pins to be viewed as a bit pattern. Fig. 1 shows the system used. The LEDs are lit or unlit depending on the key pressed and held. The pattern produced by the LEDs will display the bits of the character generated by the key pressed on the keyboard. Keyboards which generate ASCII, EBIDIC, or whatever could be checked out quickly with this system.

Fig. 1. Examining Keyboard Outputs with LED Indicators. A TTL-compatible output can drive the typical LED with about 10 milliamperes in an active low state.



Example

A keyboard which generates ASCII coded characters has the "A" key pressed and held. The LED bit pattern would look like this:

0-LED on, logic level high
●-LED off, logic level low

0 ●●● ●●0 ASCII character code for "A"
bit Ø 123 456
1 000 001

It should be pointed out that this test method will work without modification with diode encoded keyboards such as Southwest Technical Products KBD-2 keyboard (which is the keyboard I assembled and tested with the above method). However, some keyboards may generate an inverted code which shouldn't be a problem. Some keyboards (surplus and perhaps new) with more sophisticated debouncing techniques may not work with this test method without some additional components. For example, some keyboards have a bus-oriented tri-state MOS output without sufficient drive to light the diode lamps; you would need a buffer gate in this case, as well as an output data strobe. Other keyboards require an active "read" operation in which a pulse is supplied to reset flip flops acknowledging CPU acceptance of data.

Keyboard Modification

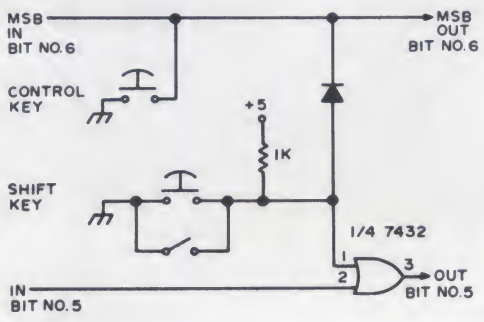
George Macomber
1422-18th Ave
Seattle WA 98122

I read your article in the September 1975 BYTE on surplus keyboards with interest. I have made some simple modifications to produce lower case codes on RTL and DTL keyboards. I have a Southwest Technical Products keyboard which I have modified. I have also modified a Sanders 720 owned by a friend.

Control Key: On keyboards with RTL or DTL outputs (Sanders 720), simply grounding the most significant bit (MSB) converts the upper case letters to the corresponding control codes. "M" becomes "carriage return" and "J" becomes "line feed," etc. Most keyboards have some control codes, but this simple modification gives all 32 possible codes 0000000-0011111.

As an example, on a Sanders 720, the "repeat key" is wired to terminal 8 (yellow wire), which is the flag output (see "Deciphering Mystery Keyboards," September 1975 BYTE). The repeat key simply grounds the flag output. Moving the wire to terminal 15 converts the repeat key to a control key.

Lower Case: A somewhat more complicated modification which works on both the Sanders 720 and the Southwest Technical Products keyboards allows the production of lower case letter codes 1100000-1111111.



In order to get lower case codes from a keyboard which produces only upper case, it is necessary to make the fifth bit high. The code for "A" 1000001 becomes "a" 1100001, and "[" 1011011 becomes " { " 1111011. The circuit shown adds lower case and control to any RTL or DTL output keyboard.

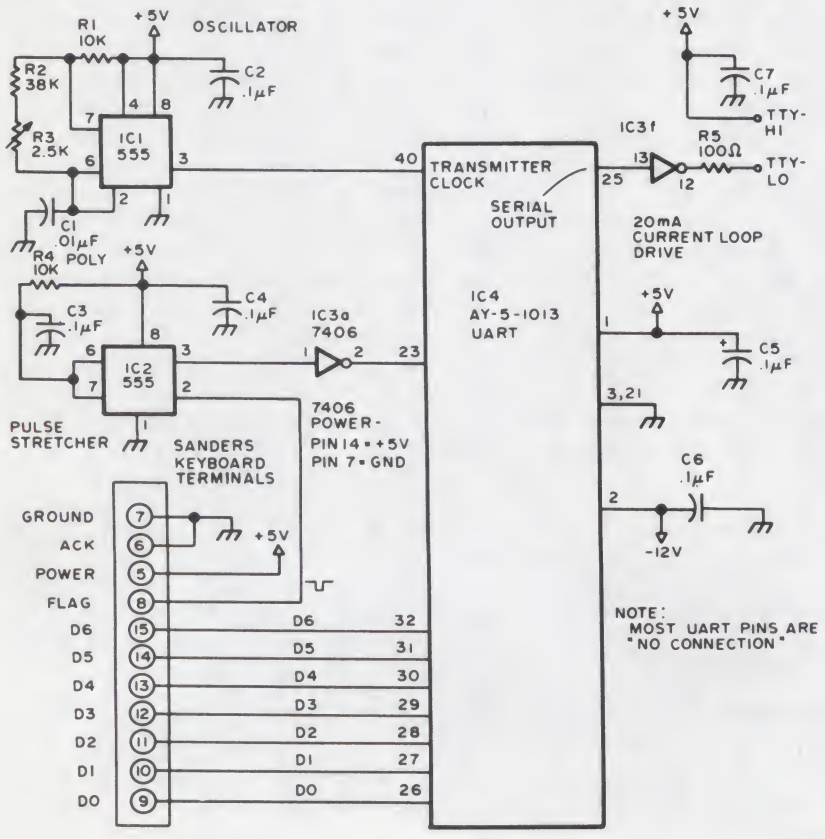
The control key has already been mentioned and is not required if the keyboard already has one (Southwest Technical Products). Both a shift key and a toggle switch are shown. You will probably want both. When the toggle or key switch is closed, the keyboard behaves as it did before modification. When both are open, the keyboard generates lower case, but the numbers and other shifted keys (i.e., 1 → !) are unaffected. A convenient key to use on the Sanders is one of the shift keys, leaving the other shift key for numbers and some other symbols (i.e., [, \ ,], _).

What happens when both shift keys are pressed? Shift' has no effect on the numbers since bit no. 6 is Ø, which forces the upper case or shift' function. But the old shift changes the letter codes, either by forcing the fourth bit to Ø, or by inverting the fourth bit (Sanders). Inverting the fourth bit allows the generation of some additional codes ([, \ ,], _), and their equivalent lower case ({ , } , ≈, DEL) and control codes. These will not be available if your keyboard forces the fourth bit, unless it has separate keys for these codes. ■

Serialize Those Bits From Your Mystery Keyboard

Dr George L Haller
1500 Galleon Dr
Naples FL 33940

Figure 1: Parallel ASCII to Serial ASCII Converter. The output of an ASCII keyboard can be converted from parallel to asynchronous serial format using a UART and two 555 timers. The result can be used to drive the 20 mA current loop of the Teletype print mechanism.

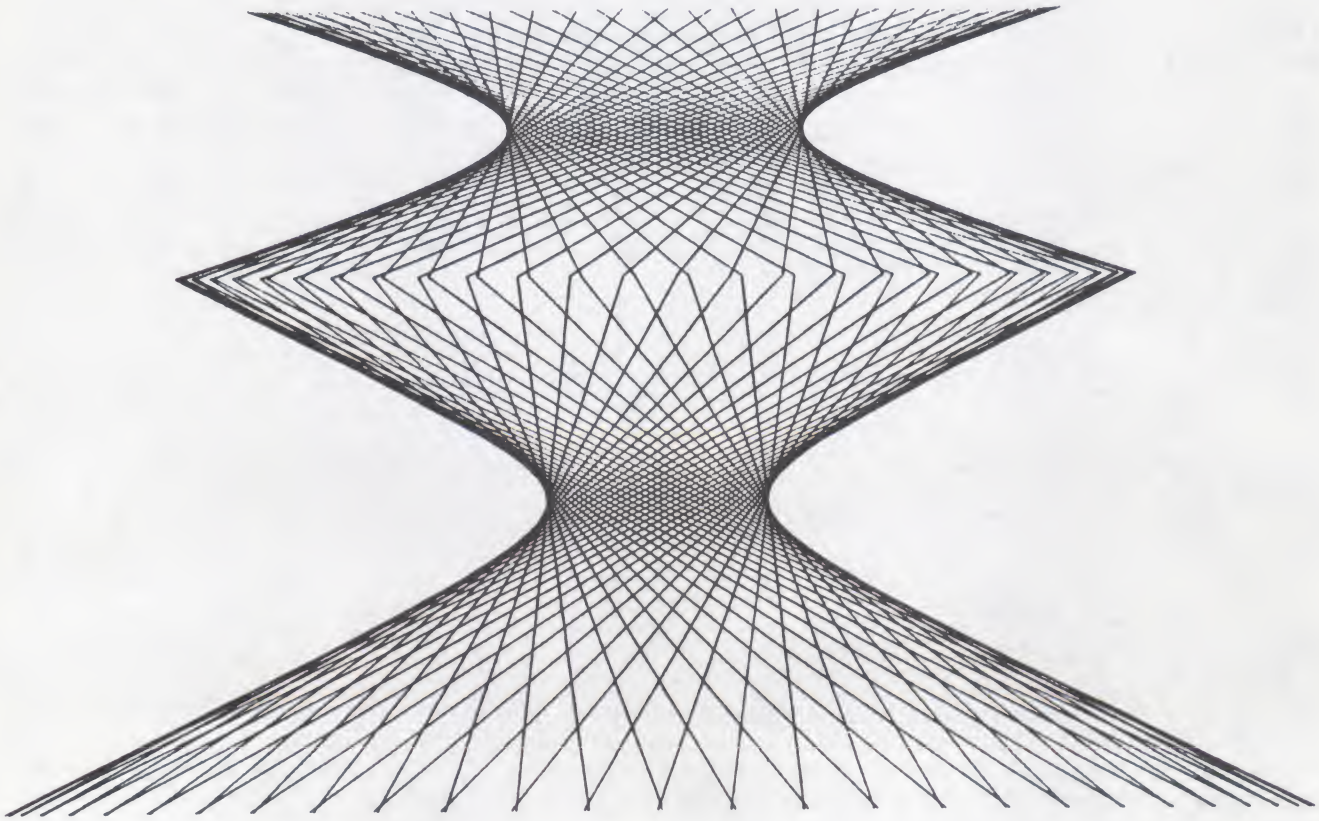


Now that you have deciphered your mystery keyboard, (page 62, September 1975 BYTE) and have determined which terminals are for the power supply, data bits, and flag pulse, what are you going to do with it? Well, one good use is to make it part of a Teletype style terminal. The Teletype models 33KSR or ASR, which are complete with printer and mechanical keyboard, are still quite expensive, usually over $1000 new; but the model 33R0, which consists of the printer only, can be bought for less than one half of that price. Now, mate the model 33R0 Teletype with your electronic keyboard and you have the equivalent of the 33KSR for your computer terminal. The ASR is the same with the addition of paper tape punch and reader. The computer terminal is usually specified as a full duplex terminal which merely means that while both the printer and the keyboard operate with serial data, they are not connected together except through the computer. The following is a description of a small adapter which will convert your electronic keyboard from a parallel to a serial output device which will then be the keyboard half of your full duplex terminal. The cost of the parts for this adapter, exclusive of power supply, is less than $10.

The main component of this adapter is, of course, the UART which has been used for several years in communication circuits for series to parallel and parallel to series conversion. An excellent explanation of the UART was given in the very first issue of BYTE. (Don Lancaster's "Serial Interface," page 22, September 1975 BYTE.) In order to use the UART, we write in 8 bits of parallel data whenever a key is struck. The key pressed pulse sent to the UART must be negative going and have the correct width to drive the UART strobe. A clock frequency

of 1760 Hz must be applied in order to get a 110 baud data rate out of the UART. The output will produce a high level mark and a low level space. Note that we are only using one half of the UART. The adapter shown here was made for the Sanders keyboard, but it should be applicable to any keyboard if considerations are made to insure that the start pulse sent to the UART is negative going, and data is in true form (logical 1 is a high level). Looking at figure 1, we find that the power is applied to UART pins 1, 2, and 3. The power requirement is about 200 mA at 5 volts (pin 1) and 10 mA at −12 volts (pin 2). The data bits are wired directly from the keyboard to the UART as shown. Terminal 6, the acknowledge function to the keyboard, is grounded. Terminal 8 of the keyboard is the key pressed flag. In the Sanders keyboard, this flag is a negative going pulse which is too short to operate the UART directly. This pulse is first stretched in a 555 timer circuit (IC2). This particular stretcher requires a negative input. After stretching, it is reinverted in a section of the 7406 and applied to the UART. The clock circuit is also a 555 (IC1). The output frequency at pin 3 of IC1 should be adjusted to 1760 Hz. This can be determined by using a frequency counter or by adjusting the potentiometer until good copy is obtained while the keyboard and adapter are connected directly to the 33R0. The frequency should be held to an accuracy of about 1%, but this is no problem with a good polystyrene condenser shown as 0.01 μF. Most of the other terminals on the transmission side of the UART should be a high level input, which means that they can be left unconnected, since they have internal pull ups. The exception is terminal 21 which is grounded. The serial output is connected through the inverter with an external pull up resistor which provides the loop with a mark current of 20 mA and a space current of zero.

Another slight modification of the Sanders keyboard will make it more useful. As received, the keyboard has no "line feed" key. It is a simple matter to convert the TAB key to an LF key. We must change the code for this key from an 013 to an 012 octal, which means we must change the zero bit from a 1 to a 0. Find the terminal at about the center of the rear of the diode matrix labeled "VT". A yellow wire connects this terminal to the TAB key. On top of the matrix board this terminal is connected to a single diode. Either end of this diode should be disconnected. This is the zero bit diode. There are two other diodes still connected under the board which will leave the code 012 octal.■



Build a Television Display

C W Gantt Jr
6 Fieldpoint Rd
Aurora IL 60538

As a small system expands and becomes more sophisticated, the limiting factor is often the speed of input and output (IO). In addition to being noisy, mechanical, and paper consuming, the slow clacking of a TTY may account for a large percentage of system time. Among the alternatives, the display of characters on a standard TV set is among the simplest and most economical methods.

This TV display (TVD) is designed to take data from 512 bytes of memory and convert it into a video signal with 16 lines of 32 characters. This can be used to feed a black and white or color TV. The data in the TVD memory is in a six bit ASCII subset and is updated by the CPU to create the desired display. The processor addresses the TVD memory just as it does any other portion of memory and can actually execute instructions from the TVD memory if so programmed. Of course, some provisions must be made to prevent the CPU and TVD from simultaneously accessing the TVD memory (more about this below).

As designed, the TVD is strictly a display device with the central processor of your system doing all housekeeping (entering characters, etc). This approach simplifies the hardware at the expense of extra software, but also allows the user to take advantage of the flexibility offered by software data manipulation and formatting.

At present one TVD is up and running in my system, but the memory and central processor interfaces are incomplete. The remainder of this article therefore emphasizes the TVD design and only offers some basic ideas on interfacing to processors. Although simple items such as the power supply and oscillators have been omitted, the information furnished should be sufficient for the more experienced readers to assemble a working version. The straight forward TVD design allows easy modification to meet individual system requirements.

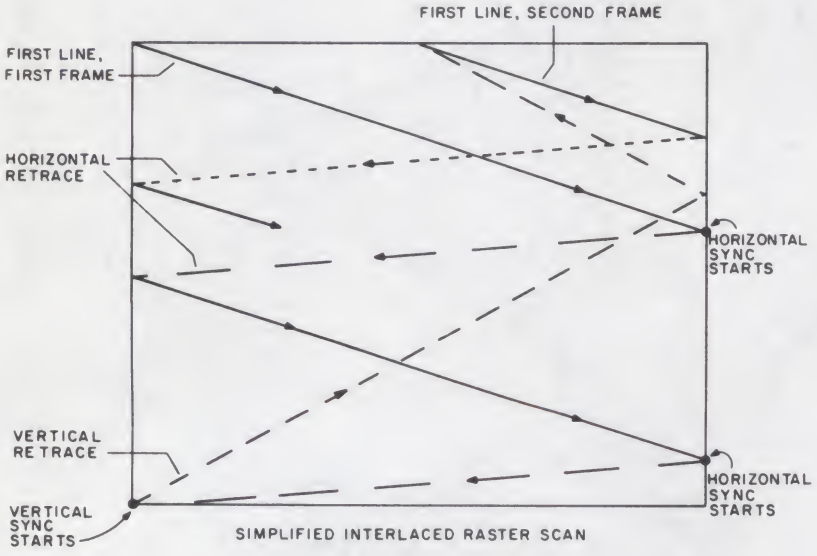


Figure 1: This shows how the electron beam is moved during an interlaced scan in a television monitor. The dashed lines are quick retrace motions which are normally invisible. The solid lines are periods during which the display presents video information controlling brightness on the tube face.

Television Raster Scanning

Before going too deeply into operation of the TVD, a review of the basic television scanning system will clarify some terms with which pure digital designers may not be familiar.

A television picture is formed by scanning an electron beam across the face of the picture tube. A TV line is one sweep of the electron beam from the left of the picture tube to the right (as viewed from the front of the set) and is initiated by the horizontal sync (see figure 1). The horizontal sync pulse causes termination of a line, horizontal retrace of the electron beam back to the left side of the screen, and the start of a new line. During the time of retrace the beam is blanked so that the retrace will not be seen. The time allotted for each complete line (including retrace) is 63.5 microseconds. Of this about 16% is taken by retrace, leaving 53.5 μs of usable line. Video information in the form of a voltage fed to the picture tube

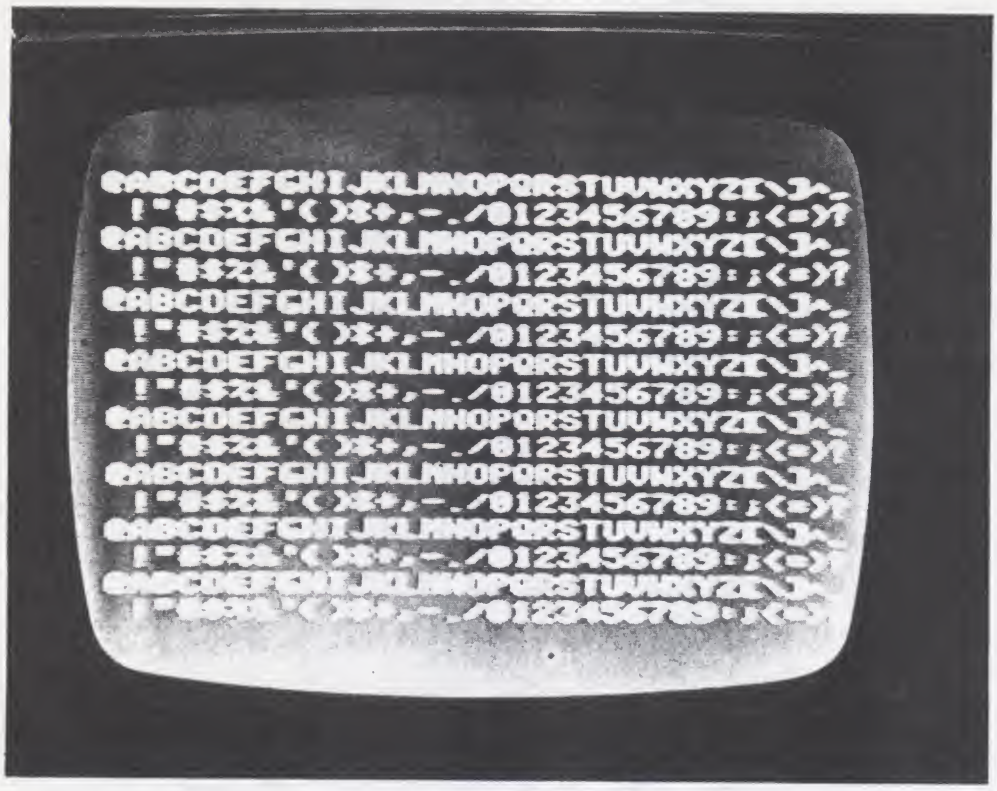


Photo 1: This is a test display pattern generated by connecting the low order outputs of the character and line counters to the character generator ROM's 6 input bits. The result presents every combination of the character set, so every character pattern is visible on the screen.

controls the brightness of the beam as it is swept across the screen.

To trace out a frame, the electron beam is slowly deflected from the top of the screen to the bottom as it rapidly sweeps horizontal lines. This vertical sweep is allotted 16.67 milliseconds (60 Hz) so there are 262½ lines in one frame. In a manner similar to the horizontal sync, the vertical sync causes the beam to be returned to the top of the screen to start a new frame. The beam is blanked during vertical retrace which takes about 1250 μs. This leaves 242 usable lines in each frame.

A complete picture is formed by two consecutive frames that are interlaced with each other. Interlacing means that the horizontal lines of one frame fit in between the horizontal lines of the other frame. The result is 30 complete pictures every second of about 484 usable (525 total) lines each. Because of the interlacing, however, the screen is illuminated at a 60 Hz rate. This eliminates an objectionable "flicker" that would be seen if the screen were only scanned at a 30 Hz rate.

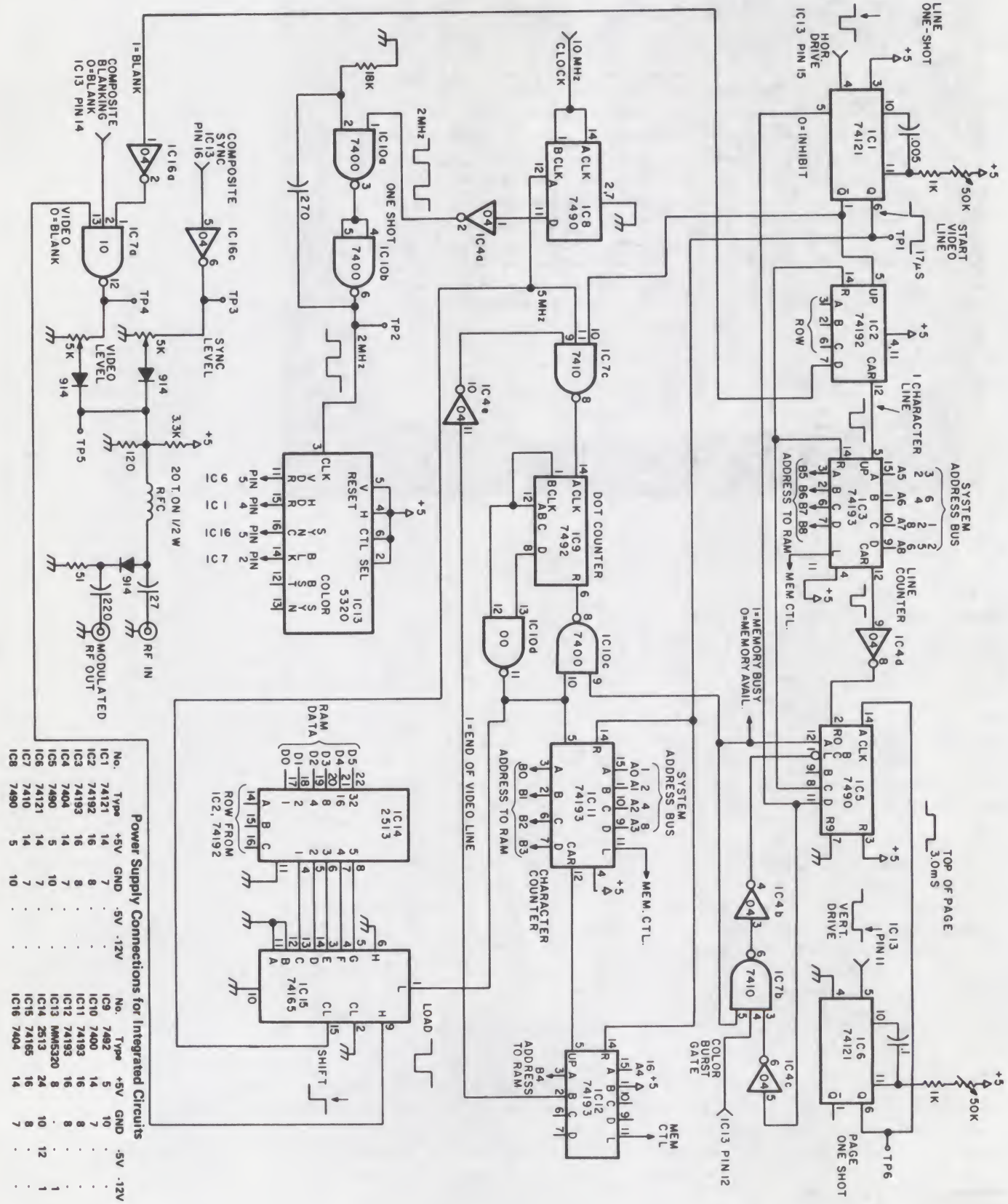
The TV signal received at the antenna terminals contains the information needed to generate the vertical and horizontal sync, blanking, and video. The TVD simulates a TV signal by supplying a composite wave form containing the same information normally present except sound. The full schematic of this TVD design (except memory) is shown in figure 2.

Character Generation

The scanning nature of the TV raster requires that the video (or brightness) information be sent in serial form to control the electron beam as it sweeps lines across the screen. Suppose, for example, that the character "H" is to be displayed as shown in figure 3. The first line can be represented as 10001, ones signifying light spots (dots) and zeros signifying dark spots. The remaining six lines can similarly be represented as a series of dots and dark spaces. When the seven lines are displayed one above the other, the character "H" is seen.

The tedious job of deciding where to put the dots (ones versus zeros) to generate a given character is done by the 2513 read only memory, IC14. It has been mask programmed at the factory with the bit patterns required for 64 separate five by seven dot matrix characters. The 2513 supplies five bits of parallel output data representing one line of a given character. It requires the six bit ASCII subset code of the character and the three bit line number as inputs. The five bit parallel output of the 2513 is converted to serial data by the 74165 shift register, IC15. To produce one line of video, five bits are required for each character in the line, plus spacing bits. Note

Figure 2: Schematic Diagram of the TV Display. This diagram includes details of the time base generation circuitry and control logic for television display generation. It omits detailed wiring of the memory circuits shown conceptually in figure 4.



Power Supply Connections for Integrated Circuits

No.	Type	+5V	GND	-5V	-12V
IC1	74121	14	7	-	-
IC2	74192	16	8	-	-
IC3	74193	16	8	-	-
IC4	7404	14	7	-	-
IC5	7490	5	10	-	-
IC6	74121	14	7	-	-
IC7	7410	14	7	-	-
IC8	7490	5	10	-	-
IC9	7492	5	10	-	-
IC10	7400	14	7	-	-
IC11	74193	16	8	-	-
IC12	74193	16	8	-	-
IC13	MM5320	8	-	-	1
IC14	2513	24	12	1	-
IC15	74165	16	8	1	-
IC16	7404	14	7	-	-

that the TVD generates two identical interlaced frames to make a complete picture. The result is that each character is actually 14 lines high.

Sync Generator

The MM5320 sync generator chip, IC13, uses a single 2.0 MHz input to produce all the sync and blanking signals needed for a 525 line interlaced raster. The same logic could be wired using TTL but would require considerably more hardware and probably cost just as much. (The 5320 runs $4.95 ppd from NEXUS Trading Co, Box 3357, San Leandro CA 94578.) The only disadvantage found thus far with the 5320 is that it prefers a square wave 2 MHz source. To this end the 100 nanoseconds pulse from the 7490 "D" output is squared using two 7400 sections of IC10 as a oneshot.

Line Generation

Horizontal drive (coincident with horizontal sync) from the 5320 triggers a 74121 oneshot, IC1, to delay the start of each line and establish the left hand margin on the screen. The output of the oneshot serves three purposes:

1. Triggers the 74192 row counter, IC2.
2. Resets the 74193 character counters, IC11 and IC12.
3. Inhibits the dot counter, IC9, until the start of the line.

When the line oneshot output pulse ends, the dot counter starts counting at 5 MHz. It resets itself every seventh count to allow for the five dots of the character plus a two dot space between characters. When the dot counter resets, it also loads the next character into the 74165 shift register, IC15. (The very first character of each line is all zeros since the 74165 is not loaded until the dot counter resets the first time.) The 74165 shifts out the two dot space and the five dot character at a 5 MHz rate. As each character is loaded, the 74193 character counter increments by one to change the address for the RAM to the next character. When the 32nd load pulse occurs, the 5 MHz input to the dot counter is inhibited using the "B" output of the second 74193 character counter. The 74165 continues to shift out the 32nd (last) character and then shifts out a steady zero. When the character counters are reset at the start of the next line, the process repeats itself.

Line Counter

The 74192, IC2, counts each video line displayed. It counts to 10 for the seven lines of character information plus a three line space.

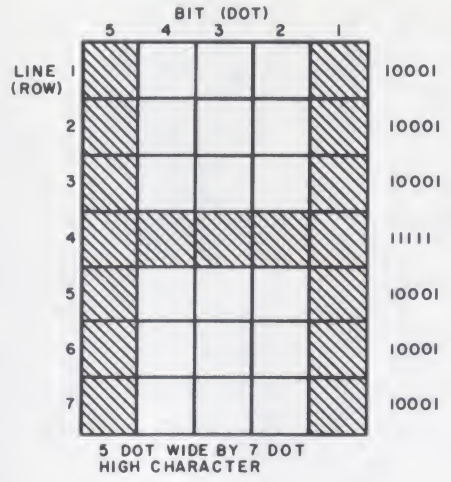


Figure 3: An example of a dot matrix pattern generated by the television display.

The "A", "B", and "C" outputs control the row inputs to the 2513 character generator chip. The first video line is all zeros since the row input to the 2513 is zero. Lines 9 and 10 are blanked using output "D" of the 74192, resulting in a total of three lines blanked. At the end of each complete line of characters, the 74193 line counter, IC3, increments by one until, at the end of the 16th line, a carry pulse is produced. This carry pulse resets the 7490, IC5, and signifies the end of a page. Output "A" of the 7490 is used to inhibit the 7492 dot counter and prevent the first line from being repeated at the bottom of the page.

Page Control

The 7490, IC5, stays reset until the top of the next page. Output "A" can be used to tell the memory control circuits that the TVD is not using the memory so that any required updates may be made by the CPU. Output "A" also inhibits the "B Clock" input via the 7410. The "D" output inhibits the line oneshot.

When a vertical drive pulse (coincident with vertical sync) triggers the 74121 page oneshot, IC6, the TV set syncs to the top of the next frame. The page oneshot delays the start of the first line to establish the top margin. At the end of the oneshot's output, IC5, bit "A" is clocked to a one. This tells the memory control that the TVD needs to resume control of the memory address and also enables the IC5 "B Clock" input via the 7410, IC7. The "B" section of the IC5 then proceeds to count color burst gate pulses to give the memory time to complete any access already in progress. The color burst gate was used only because it was convenient and occurs at the same rate as the horizontal drive – the horizontal drive could be used at the expense of a buffer since the 5320 can

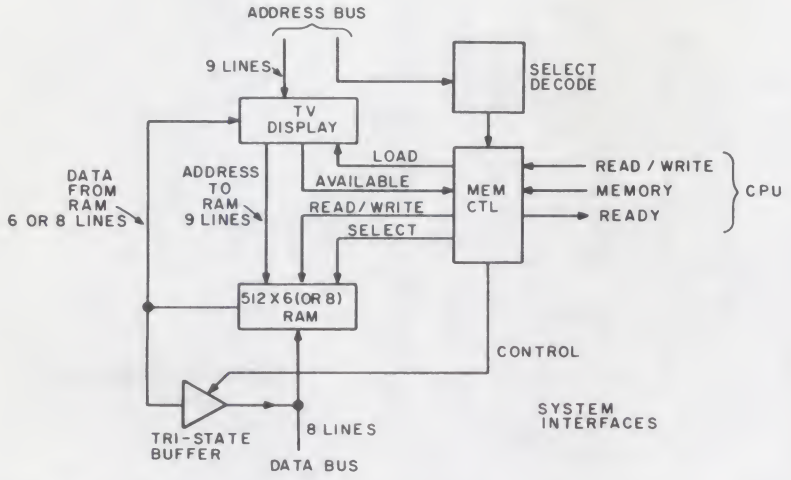


Figure 4: System Diagram. This figure details how the TV display fits into a central processor's memory address space. The low order 9 lines of address go directly to the line and character counters of the TVD; the memory array is addressed by the outputs of the counters, which are connected logically to the address bus when the load line demands central processor access. The high order bits of the processor's address are decoded separately and are used to enable processor access if the TV display portion of address space is referenced.

only drive one TTL load per output. When output "C" of the IC5 goes high, the line counters are reset. When output "D" goes high the "B clock" is inhibited via IC7b, and the line oneshot is enabled. This allows the first line to start.

Composite Video Generation

The video and sync are independently adjusted and then added to produce composite video. This can be piped directly into a set (be sure not to touch a hot chassis!) or used to modulate a low power RF source. A signal generator works fine for tests. (See "Television Interface" by Don Lancaster, page 20, October 1975 BYTE, for a thorough discussion of the various tricks to improve the interface.)

Memory Interface

Figure 4 illustrates how the TVD fits into a larger system. It is intended that the address outputs of the 74193 character and line counters (IC3 and IC11) be hard wired to the address lines of a 512 or 1 K by 6 static random access memory using 2102s or similar parts. The data outputs of low order 6 bits of this memory are the ASCII character select inputs to the 2513 character generator, IC14, and can be gated back to your system's data bus if you want the CPU to be able to read from the RAM. (Of course, a 512 by 8 memory would be needed if the CPU is to be able to use the RAM for other tasks.) The data inputs of the RAM tie to the data bus to allow the CPU to write into the RAM.

To avoid breaking up the picture on the display during access, the memory control logic must use the "A" output of IC5 to tell when the CPU can use the RAM and when it must signal a busy to the CPU at the start of a page. There is more than ample time between the "A" output and the line counter reset to finish any access in progress. To use this feature, the memory busy line must be wired to your processor's "memory ready" line (possibly through an inverter if the logic of your particular computer requires it). This method will work well for any processor, like the 8008 or 8080, which allows unlimited "memory busy" delays. However, for dynamic processors such as the 6800, the maximum processor delay time of about 5 μs dictates use of an alternative approach. One simple approach is to ignore the effect of memory access on the display. The result will be a short glitch in the display corresponding to each computer access. The nature of the glitch will be a resetting of the line and character counters to a new location, causing a scrambling of the display for the remainder of the current frame. A second approach is to wire the memory ready line into a single bit input port which can be tested as a status flag: If the line indicates a retrace, then the memory access software for the display will allow an update to occur.

The CPU addresses the RAM through the character and line counters (IC3 and IC11) by tying their data inputs to the system address bus and using the load control of pin 11. The 74193s can also be used as temporary storage for the address in a system with a common address and data bus. Note that the TVD does not interfere with CPU access to the remainder of the system's memory at any time and only delays the CPU by one of the techniques discussed above if it tries to access the TVD RAM while a page is being displayed. The CPU has the entire vertical retrace to make updates at once every 16.67 milliseconds.

Lacking a memory for my initial testing, the 2513 data inputs were temporarily tied to the 74193 address outputs (2513 PIN 17 to character counter PIN 3, etc) to display the complete 2513 repertoire every two lines as in photo 1. The 74193 load lines must also be connected to a "one."

Modifications and Adjustments

1. There is one known bug so far and no doubt more will show up when the TVD is integrated into a system. The 7490 can, on power up, hang in state with both the "C"

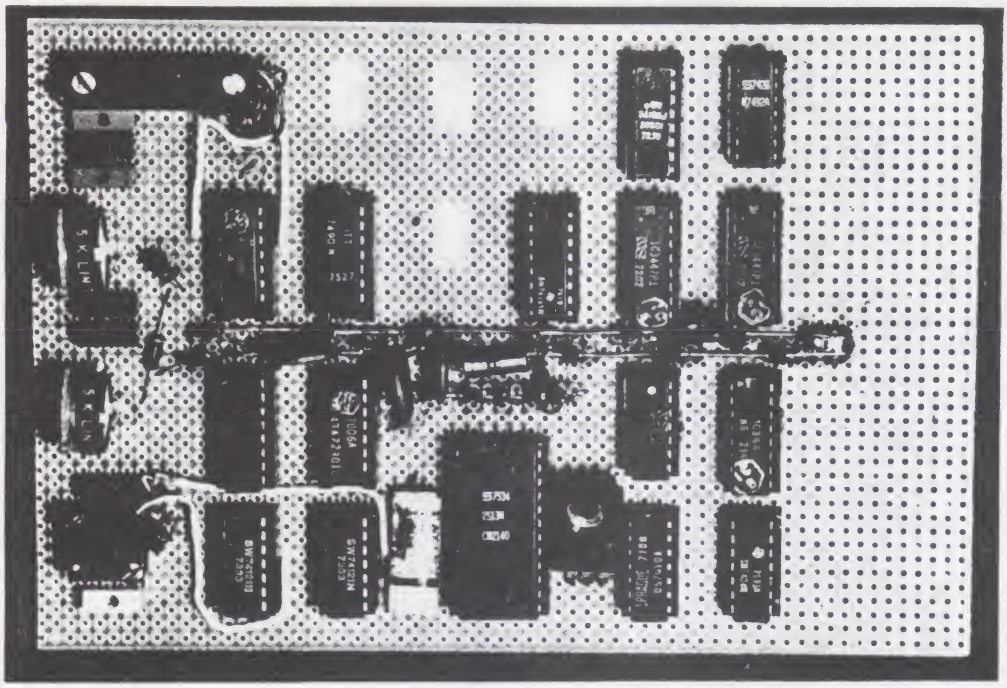


Photo 2: Prototype Circuit. The large socket is for the character generator. Test points are the 6 small rectangular objects along the left hand side of the board. A 7812 regulator in a plastic package is to the right of the character generator, and is used to provide the −12 V bias for the ROM. A zener diode with a dropping resistor is used to create the −5 V bias required for the ROM.

and "D" outputs a one. This state continuously resets the 74192 row counter and is nonrecoverable. A cure would be to power-on-reset the 7490 "R9" which does produce a recoverable state.

2. As mentioned before, the 5320 likes a square wave input, so check the 7400 imitation of IC10a and b oneshot, or better yet use a 74121.

3. Using a separate oscillator for character generation (5 MHz) would allow adjustable character width, but watch out for any interaction with the 2 MHz – it shows up as a torn, garbled display, as will most sync or jitter problems. A crystal is best. (It is possible to use 12 MHz and a 7492 in place of IC8 to get 6 MHz for the characters and 2 MHz for the 5320.)

4. The prototype is wire wrapped on a 4½ by 6 inch (11.43 by 15.24 cm) vector board (see photo 2) with room to spare, although a slightly larger board would accommodate more interface goofs. Fulp's corollary says things like this always get bigger. Also, the 44 pin connector planned for the prototype is not large enough counting the additional RAM address and data lines.

5. The modulation levels for the radio frequency modulator are fairly critical and misadjustment of sync or video levels will cause a torn display. Try setting video level control for 1/2 of maximum and sync for 3/4 of maximum.

6. Harmonics from the 10 MHz tend to leak into the TV so pick a higher channel (5 or 6) if herringbone is noticeable on your display.

7. There are many causes for ghosting and smeared characters including VSWR (voltage standing wave ratio) on the cable to the TV, misadjusted fine tuning, or a narrow band width TV.

8. Character and line spacing can be altered by modifying the dot and row counters, respectively, to reset at different counts. Be careful though, or the display will not fit on the screen.

9. The unused bits C and D of the second 74193 character counter, IC12, may be used for the 512 and 1024 bits if a two or four page RAM is desired. Some method of controlling these bits during display time is needed to select the page.

10. A light pen could strobe the present RAM address into a latch to be read by the CPU via the data bus.

11. The 5320 provides color sync gating so how about color characters? The extra 2 bits available with an 8 bit wide RAM can provide software control of many goodies (like brightness, color, blinking, underlining, black on white, etc).

The TVD as described can be used to display one or more pages of ASCII characters and opens up many possibilities of modernizing the IO portion of a small system. My home brew computer will be a complete microcomputer chip based system, designed with the TVD as the main man machine interface.■

The "Ignorance Is Bliss" Television Drive Circuit

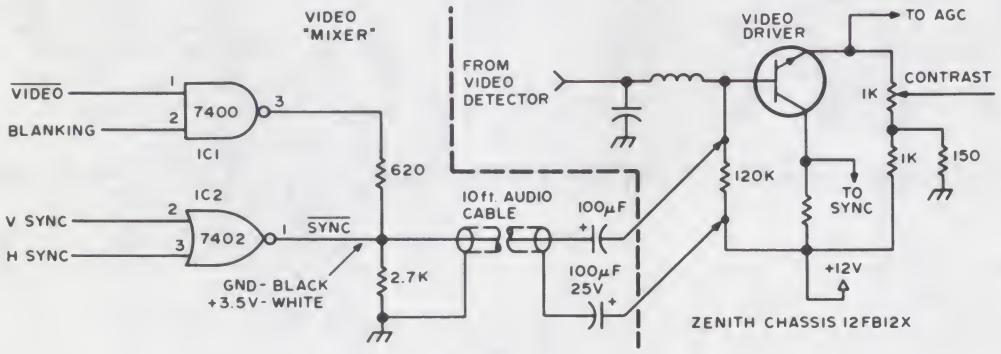


Figure 1: The "Ignorance Is Bliss" Television Drive Circuit. The components to the left of the dashed line were added as part of the interface. The components to the right of the dashed line are part of the Zenith 12FB12X chassis which was used for the television display.

Ken Barbier
PO Box 1042
Socorro NM 87801

I had not yet heard of BYTE magazine, or hams building such hardware, when I built my CRT terminal (a computerhead's term for "TV typewriter") in the fall of 1975. I didn't even own a TV set! Two situations resulted: I had to buy a new all solid state TV, and I didn't have any idea how to interface with it. I knew approximately what it took to create horizontal and vertical sync, but had no idea whether levels, pulse widths, and frequencies would be noncritical. I was delighted, therefore, when my sync generator worked just fine the first time I patched its output across the video driver base resistor using the circuit as shown in figure 1. My big fat TTL level pulses swamp the AGC circuity so effectively that normal signals and noise from the TV IF just disappeared and I had nothing to switch off!

Not having any idea how to mix in my video (character generator output) with the sync, I just hooked up a 2 K pot where the 620 ohm is shown, and started reducing video until it stopped interfering with the sync, and there I was at 620 ohms. All was fine, until I erased my character memory and started typing in one character at a time. Contrast went all to pot! I had provided no DC restoration. And I never did. At least not in the TV set circuitry.

My terminal design produces 24 lines of 64 characters each, with a total of 270 scan lines per frame. Vertical sync is the 10 scan lines that would have been character line 26. To eliminate the need for the type of DC restoration as detailed in "Television Interface" (page 20, BYTE, October 1975), I generated a black level blanking signal covering what would have been character lines 25 thru 27. This signal enters the blanking gate, IC1, at pin 2 in figure 1. Now, when I turn on my system and erase the memory, my TV field shows a nice white area with a black border top and bottom. My character generator output produces black on white characters which I find preferable to the usual white on black.

Simply by turning off the logic power I can be instantly flooded with the inanities emanating from the vast TV wasteland. With this design, I have no need to pull plugs or throw switches. Sometimes ignorance can be bliss.■

Build a TV Readout Device for Your Microprocessor

Dr Robert Suding
Research Director, The Digital Group Inc
PO Box 6528
Denver CO 80206

A television set readout for your microprocessor has many attractive advantages. The TV readout is vastly faster, quieter, and even lighter, than the usual Teletype based design. Since it is an electronic rather than mechanical device, less service and maintenance are required. Much more data may be contained on a television screen than on front panel readouts.

The precise design of the television driving circuitry (interface) can take on a considerable number of forms. Some considerations are:

- Number of horizontal characters.
- Number of vertical characters.
- Upper case only, or upper and lower case text.
- Character generation format:
  row or column scan.
  5 x 7 dot matrix, 7 x 9 dot matrix or?
- Alphanumeric only, or alphanumeric and graphic formats?
- Converted home TV set or commercial TV monitor?
- Separate TV buffer memory or TV buffer shared with main memory?
- Shift registers for memory, or programmable RAM?
- Multiple video pages or single display?
- Interlaced scan or non interlaced?
- Hardware or software cursor, or no cursor?

Rather extensive list, isn't it? Understandably, a large number of designs have appeared recently, and many more will be seen. Every design has some advantages, some disadvantages.

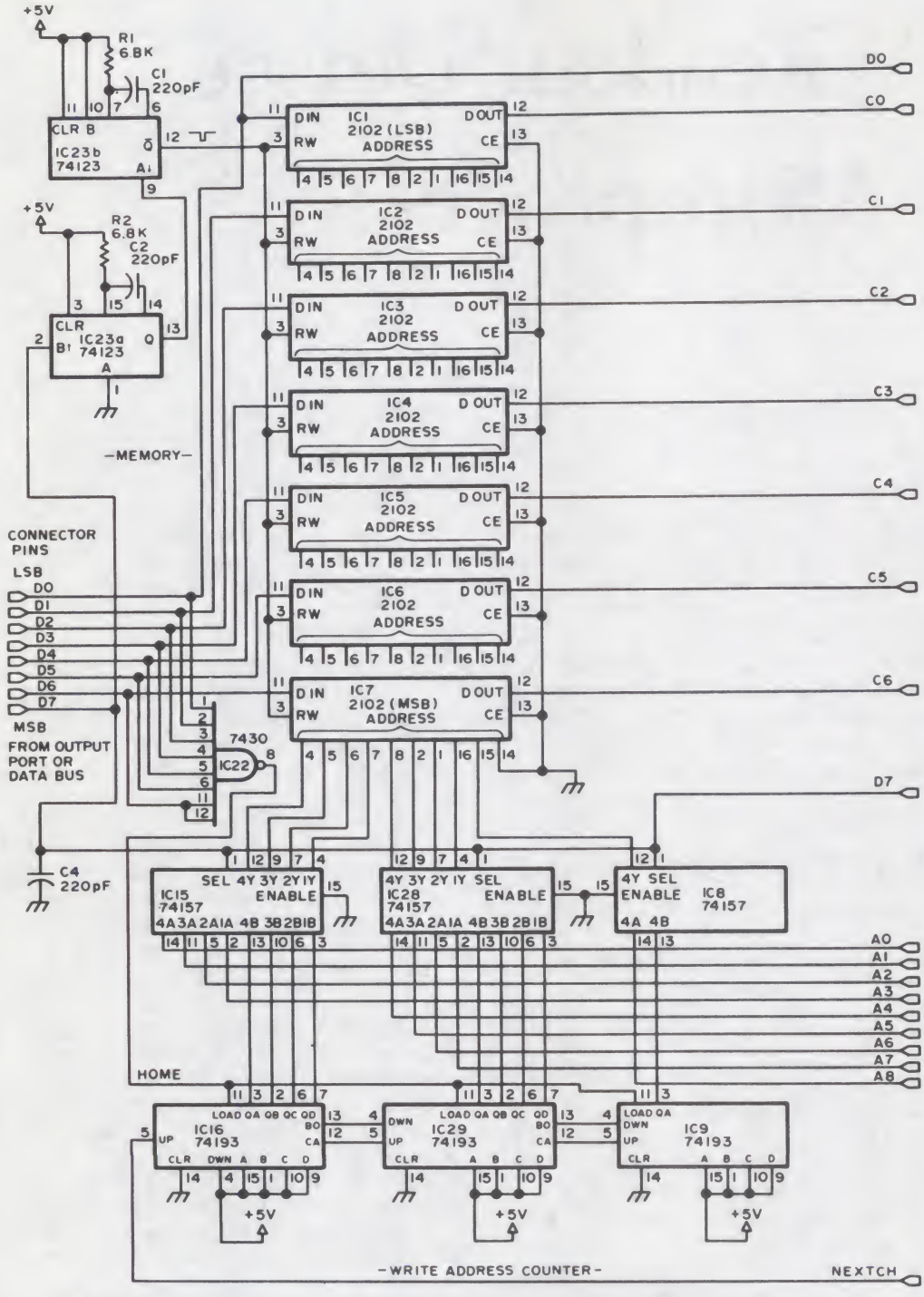
The 5 x 7 dot television display circuit in the June 1976 BYTE [*page 16*] is an example of a number of the above design alternatives. The 5 x 7 dot 2513 is a rugged, low cost character generator. The MM5320 is a fairly easy way to generate an interlaced signal. Programmable random access memory provides a random and faster screen update capability compared to the shift register "TV typewriters" of a few years ago.

Major Features

The television display design shown in this article has several major departures from previously published designs. The June 1976 BYTE article on "A Systems Approach to a Personal Microprocessor" [*page 32*] stresses the need to keep various system elements independent in order to avoid unnecessary obsolescence. By using a simple parallel interface and making refresh memory part of the design, this television display achieves independence from a particular computer and bus design. This same display is also useful in such items as terminals, TV typewriters, and large computers.

A Motorola MCM6571L character generator is used as the heart of the Digital Group as well as several other video display systems. This character generator provides a 7 x 9 dot matrix character with automatic character shift for lower case characters such as g,j,y, etc, which extend below the base line, making an effective *7 x 13 dot matrix.*

Thirty-two characters per line by 16 lines give a total of 512 characters on the screen. Endless arguments can result when screen formats arise. The 32 x 16 format was chosen to achieve the clearest and simplest (hence lowest cost) system. The more characters per line, the more television bandwidth is required. This system requires a TV monitor with better than 6 MHz bandwidth. A system with 64 characters per line would require a 12 MHz monitor, etc. Since the system was designed to minimize costs, a



+5 POWER SUPPLY PINS

Number	Type	+5V	GND
IC1	2102	10	9
IC2	2102	10	9
IC3	2102	10	9
IC4	2102	10	9
IC5	2102	10	9
IC6	2102	10	9
IC7	2102	10	9
IC8	74157	16	8
IC9	74193	16	8
IC10	74193	16	8
IC11	7404	14	7
IC12	74193	16	8
IC13	7400	14	7
IC14	7400	14	7
IC15	74157	16	8
IC16	74193	16	8
IC17	74193	16	8
IC18	7410	14	7
IC19	7404	14	7
IC20	7420	14	7
IC21	74193	16	8
IC22	7430	14	7
IC23	74123	16	8
IC24	74193	16	8
IC25	7430	14	7
IC26	7401	14	7
IC27	74L00	14	7
IC28	74157	16	8
IC29	74193	16	8
IC30	MCM6571L	2	13
IC31	74165	16	8
IC32	74193	16	8

Figure 1: Memory and write address counter logic for the TV readout design. The inputs to this circuit are at the left, labelled D0 to D7 corresponding to the data lines of a typical latched output data port. The connections to figure 2 include D0 and D7, memory outputs C0 to C6, and video timing chain address lines A0 to A8.

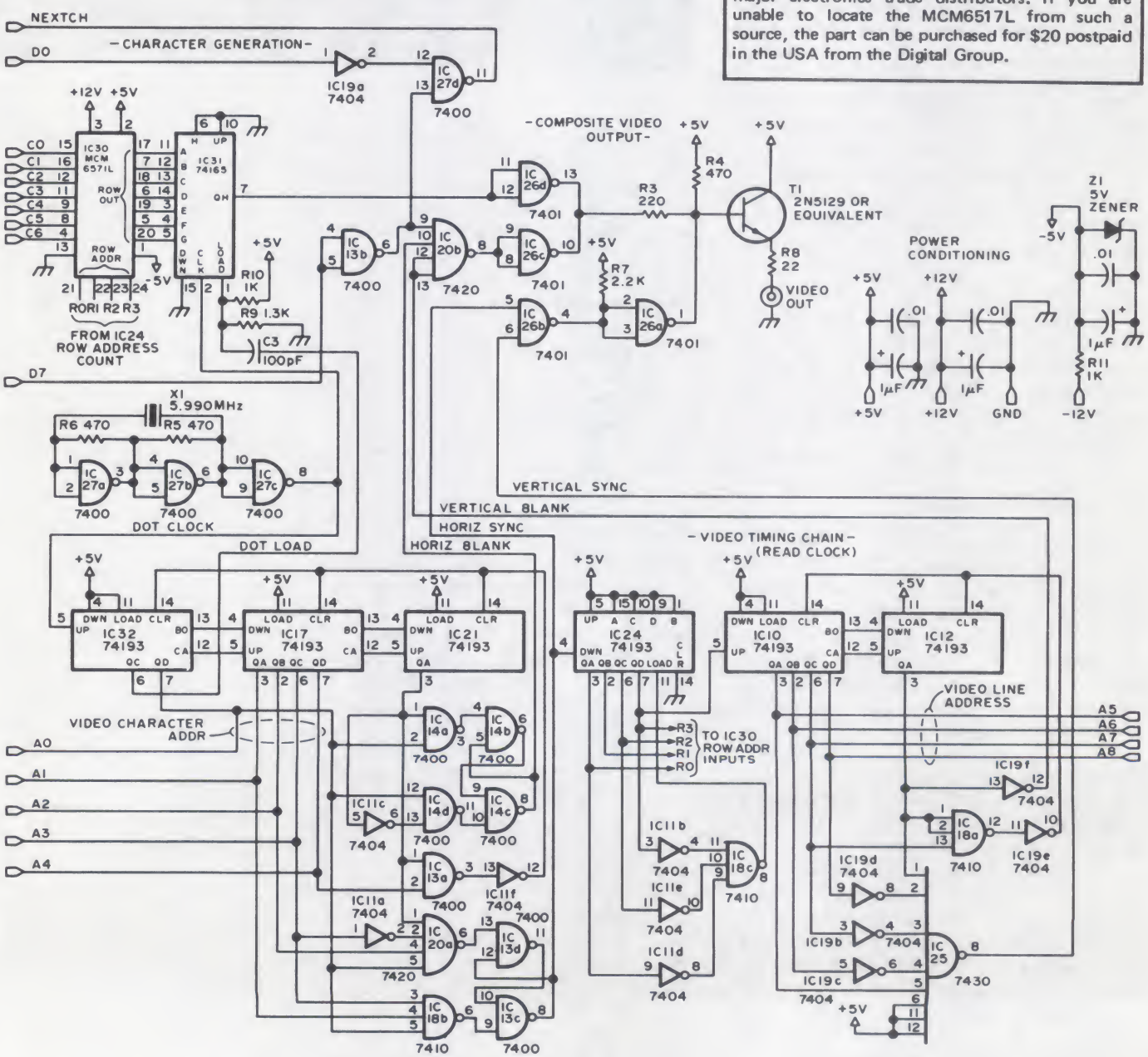
Figure 2: Character generation, composite video output and video timing chain logic for the TV readout design. The output of the TV readout is the composite video signal which drives a monitor or modified standard television set through a coaxial cable. The character generation logic consists of a read only memory, IC30, to translate character patterns into horizontal rows of dots, and the shift register, IC31, which sequences the bit by bit output of the row of dots. The video timing chain is a series of counters driven by the 5.990 MHz crystal, which cycles through the memory section of figure 1 and controls operation of the display.

A Note About Construction

The circuit shown in figures 1 and 2 is complete, and can be constructed in any well equipped home hardware laboratory using point to point soldering, home brew printed circuits, Vector wiring pencils, or wire wrap as an interconnection technique.

For those who wish to take advantage of construction using a circuit board and a complete set of electronic parts, Dr Suding's TV readout and the cassette interface described in his article on page 46 of July BYTE are available in a combined kit form for $130, postpaid in the USA. Contact the Digital Group Inc, PO Box 6528, Denver CO 80206, for information on this product.

For home brewers, the only part which might be difficult to find in surplus markets is the Motorola MCM6517L character generator chip. This package is available over the counter at many major electronics trade distributors. If you are unable to locate the MCM6517L from such a source, the part can be purchased for $20 postpaid in the USA from the Digital Group.



home black and white television set can be easily modified [*page 20, October 1975 BYTE*] and will satisfactorily meet the 6 MHz requirement. Sixteen rows of characters allow use of a non-interlaced sync system for lower cost. My own preferred TV display formats are either a 32 x 16 character system or the 80 x 24 character format. However, the 80 horizontal characters will require an expensive monitor to achieve the 15 MHz TV bandwidth and critical corner focus requirements.

The character memory can be of several formats, but this system uses a self contained programmable memory buffer which is loaded sequentially from the driving 8 bit output port of the microprocessor, or an ASCII keyboard. Some systems permit data readback from the TV readout system, but a greater cost is involved, and a mirror image buffer in the computer's programmable memory will produce the same result. Use of programmable random access memory in the TV readout permits very fast loadings, as fast as the system can output data. The typical update time for a total of 512 characters is under 5 ms. How far under 5 ms depends on the driving software and microprocessor used.

Cursors and cursor control may be performed in hardware or software. The approach of this system is to use software for the most part, which results in lower cost hardware. Cursor inserting subroutines are then used as needed.

So much for system design alternatives.

TV Readout Description

This TV readout consists of five interacting sections. They are memory, character generation, composite video output, video timing chain, and write address counter. The memory section (figure 1) consists of seven 2102A-2 or faster 1 K memories. Only one half of each memory is used, giving a possible storage of 512 seven bit ASCII characters. The microprocessor, keyboard, or some attached circuit writes the characters one by one into the 2102s, and then the TV readout continuously displays these characters until either more characters are entered, or the circuit is turned off.

The character generation circuit (see figure 2) consists of two integrated circuits, the MCM6571L character generator, IC30, and the 74165 shift register used to convert from parallel to serial. The 6571 takes the seven bit ASCII character coming from the memories and outputs 7 dots making up a character row for each of 13 potential rows making up each character. The 74165 loads all 7 dots into its internal memory, and then outputs these dots one at a time for serial transmission to a TV set. For more information on TV character generators, I would suggest reading an excellent article by Don Lancaster in June 1974 *Radio Electronics* [*pages 48-52*], or the June 1976 BYTE magazine article by C W Gantt [*page 16*].

Photo 1: A test demonstration of Dr Suding's TV readout, shown in schematic form as figures 1 and 2. The test pattern consists of the four lines in the center which cycle through the possible binary combinations of characters. The differences in line width between the top line and the other lines are caused by non-linearities in the monitor used for this photograph.

The video output section uses a 7401 open collector NAND gate and a driver transistor to produce a low impedance composite video signal. The output is around 3 V peak to peak with about a 1/2 V horizontal and vertical sync and blanking pedestal.

The read clock (see figure 2) is the source of master control for the various sections. Starting from an initial frequency of 5.990 MHz, a countdown chain of three 74193s (ICs 32, 17, and 21) produce an 8 μs horizontal sync when gated by IC11a, IC20a, IC18b, IC13c and IC13d. A 41 μs horizontal blanking circuit prevents loss of characters at the edges of the screen, and is produced by the gating action of IC14, IC11c, IC11f and IC13a. The resultant horizontal frequency is 15,598 Hz, somewhat lower than the standard 15,750 Hz, but usually only requires trimming horizontal hold slightly, if at all.

The vertical countdown chain uses three more 74193s (ICs 24, 10 and 12) to obtain a final vertical frequency of 60 Hz, the same frequency as the AC line to avoid hum roll and wobble problems on low cost televisions. IC19b, IC19c, IC19d and IC25 produce an 820 μs vertical sync pulse, IC18a and IC19e detect state 20 of IC10 and IC12, counting lines 0 to 19 and giving four line periods for vertical retrace. The inverter IC19f produces a 3.5 ms vertical blanking

pulse during states 16 to 19 of the counter IC10 and IC12.

As if these operations weren't enough, part of the video timing chain, counter IC24, tells which of the 13 lines in a character is being currently accessed. The counter IC32 keeps track of shifting and loads the 74165 when the row of 7 dots is available from the 6571. The 5.990 MHz signal then shifts out 8 dot periods (the 8th one is a horizontal space between characters) before the next dot load command occurs. All of these timings are very critical during the design phase; but since the circuit is digital, the builder should have no problems, since no adjustments are needed. The video timing chain counters develop a 9 bit address that controls which of 512 characters is currently being presented to the 6571 for dot encoding. This is routed to memory through 74157 multiplexors IC15, IC28 and IC8 except during write clock time.

I thought you'd never ask about the write clock. Well, it controls the entry of the

Figure 3:

Check Out Notes

The TV readout should be assembled according to your preferences (see "A Note About Construction") using sockets for all integrated circuits. These notes suggest a procedure for orderly testing of the new TV readout.

1. **Power supply.** Start checkout after all wiring has been completed, but before any integrated circuits have been inserted into sockets. Measure the resistance between ground and the other voltage supply pins. A very low resistance indicates a bad bypass capacitor, a solder bridge, or some other form of short circuit between the supply voltage and ground.

2. **TTL integrated circuits.** Insert all the integrated circuits of the TV readout except the memories (2102s, IC1 to IC7) and the character generator (MCM6517, IC30). Measure the resistance between the ground and the +5 V supply pin, noting its value; reverse the ohmmeter leads and remeasure. A shorted reading in either direction indicates a bad integrated circuit, and nearly equal readings in both directions indicates that at least one integrated circuit has been plugged in in reverse.

3. **Initial power up.** Temporarily ground the most significant bit input pin (D7 in figure 1), and connect the video output to a commercial TV monitor, or a TV set which has been modified to act as a monitor. Turn on the +5 V power. You should see 32 white vertical columns on the screen. (Refer to the "Diagnosis of Ailing Readouts", section 2, if this does not happen.) Turn off +5 V power.

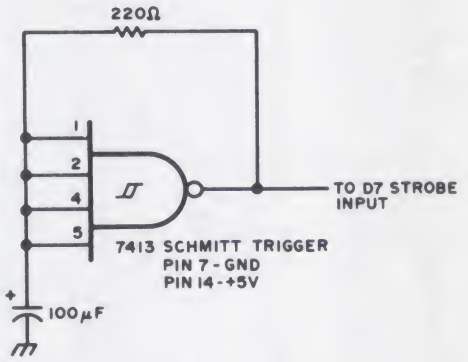
Connect up the +12 V and -12 V power supplies, then turn on all power again. Verify the proper voltages on the MCM6517L socket, IC30: Pin 1 should have -5 V, pin 2 should have +5 V and pin 3 should have +12 V. Turn off power again.

4. **Now plug in the MOS parts:** The seven 2102 memory integrated circuits and the MCM6517 character generator read only memory. (The temporary ground jumper for the D7 input, and the video monitor output are still attached.) This time, when power is turned on, you should see a random display of 512 characters on the screen. The actual character at each location is determined by the chance power on initialization of each bit location, and cannot be predicted in advance.

5. **Testing:** Complete testing is now possible under computer control or by using a breadboard input device. If you use microprocessor control, simply wire the inputs to the TV readout to an 8 bit output port, load the software of listing 1 (if you have an 8080 or Z-80; write equivalent programs for other processors if necessary), and write some simple programs to generate known data and load that data into the display.

If it is desired to test the TV display without a microprocessor, the oscillator of figure 3 can be used to drive the input strobe pin, D7. Then temporarily tie all the other data pins to the +5 V supply through a 1 k resistor. Verification of the operation of the display can be obtained by grounding bits D0 through D6 of the input (the 1 k pullup resistors protect the power supply). The following table gives the characters which should fill the screen for each case:

Pin to Ground	Character	Octal Code	
D0	~	376	
D1	}	375	
D2	{	373	
D3	w	367	
D4	o	357	
D5		337	(underscore)
D6	?	277	



A test setup for manual verification of the display. The Schmitt trigger integrated circuit, a 7413 NAND function, has an RC feedback network to cause oscillations. This logic oscillator is used to drive the strobe input continuously, so that memory will be filled with a constant character pattern if that pattern's ASCII code is presented on input pins D0 to D6.

characters from your external source into the 2102 memory bank. Several alternatives in character entry are possible, yet give the user a very capable unit, particularly when using a microprocessor, or even mini, midi, or maxi processors.

A sequential entry system is utilized. A home reset control signal (denoted "■") is developed by IC22 when it detects all of the 8 input lines high ("1"). The write address counter of IC16, IC29 and IC9 is then preset so that the next character to be entered will result in its being displayed as the top leftmost character on the screen. The second character will be viewed to the right of the first, . . . until on the 33rd character a new

Table 1: Character graphics, octal codes and binary codes for the TV readout.

Char	Octal	Binary*	Char	Octal	Binary*
α	200	10 000 000	@	300	11 000 000
β	201	10 000 001	A	301	11 000 001
γ	202	10 000 010	B	302	11 000 010
δ	203	10 000 011	C	303	11 000 011
ε	204	10 000 100	D	304	11 000 100
ζ	205	10 000 101	E	305	11 000 101
η	206	10 000 110	F	306	11 000 110
θ	207	10 000 111	G	307	11 000 111
ι	210	10 001 000	H	310	11 001 000
κ	211	10 001 001	I	311	11 001 001
λ	212	10 001 010	J	312	11 001 010
μ	213	10 001 011	K	313	11 001 011
ν	214	10 001 100	L	314	11 001 100
ξ	215	10 001 101	M	315	11 001 101
ο	216	10 001 110	N	316	11 001 110
π	217	10 001 111	O	317	11 001 111
ρ	220	10 010 000	P	320	11 010 000
σ	221	10 010 001	Q	321	11 010 001
τ	222	10 010 010	R	322	11 010 010
υ	223	10 010 011	S	323	11 010 011
φ	224	10 010 100	T	324	11 010 100
χ	225	10 010 101	U	325	11 010 101
ψ	226	10 010 110	V	326	11 010 110
ω	227	10 010 111	W	327	11 010 111
Ω	230	10 011 000	X	330	11 011 000
√	231	10 011 001	Y	331	11 011 001
→	232	10 011 010	Z	332	11 011 010
←	233	10 011 011	[	333	11 011 011
↑	234	10 011 100	\	334	11 011 100
÷	235	10 011 101	]	335	11 011 101
Σ	236	10 011 110	↑	336	11 011 110
≅	237	10 011 111	←	337	11 011 111
blank	240	10 100 000	`	340	11 100 000
!	241	10 100 001	a	341	11 100 001
"	242	10 100 010	b	342	11 100 010
#	243	10 100 011	c	343	11 100 011
$	244	10 100 100	d	344	11 100 100
%	245	10 100 101	e	345	11 100 101
&	246	10 100 110	f	346	11 100 110
'	247	10 100 111	g	347	11 100 111
(	250	10 101 000	h	350	11 101 000
)	251	10 101 001	i	351	11 101 001
*	252	10 101 010	j	352	11 101 010
+	253	10 101 011	k	353	11 101 011
,	254	10 101 100	l	354	11 101 100
-	255	10 101 101	m	355	11 101 101
.	256	10 101 110	n	356	11 101 110
/	257	10 101 111	o	357	11 101 111
0	260	10 110 000	p	360	11 110 000
1	261	10 110 001	q	361	11 110 001
2	262	10 110 010	r	362	11 110 010
3	263	10 110 011	s	363	11 110 011
4	264	10 110 100	t	364	11 110 100
5	265	10 110 101	u	365	11 110 101
6	266	10 110 110	v	366	11 110 110
7	267	10 110 111	w	367	11 110 111
8	270	10 111 000	x	370	11 111 000
9	271	10 111 001	y	371	11 111 001
:	272	10 111 010	z	372	11 111 010
;	273	10 111 011	{	373	11 111 011
<	274	10 111 100	}	374	11 111 100
=	275	10 111 101	~	375	11 111 101
>	276	10 111 110	"Home"	376	11 111 110
?	277	10 111 111		377	11 111 111

*The low order 7 bits of the binary representation map into the ASCII graphics where such graphics are defined. The high order bit is always a "1" value to act as a strobe in the software of TVOUT shown in listing 1.

Diagnosis of Ailing Readouts

1. Troubles – General
   - One of the more difficult troubles to find is an IC pin which was bent under the integrated circuit when it was inserted. Any unusual pressure when inserting an integrated circuit should be investigated.
   - Check continuity. Your wiring should be correct. If soldering is used, as in printed circuit assembly, check to make sure all joints are in good shape.
   - When troubleshooting with an oscilloscope probe, measure from the *top* side of the integrated circuit, not the bottom, to eliminate the possibility of being misled by a pin which is bent under or a defective socket.
   - Before ever plugging in any integrated circuits, always measure the voltages at the terminals of the display board and at the power pins of the more expensive integrated circuits, like the MCM6571.
   - When handling integrated circuits, avoid static charges. Run your house humidity high, and ground yourself by touching a grounded chassis before touching the integrated circuits.

2. When initially checking out, if no white columns appear on the screen at step 3, the following may be a cause of the problem.
   - Bad connection between TV output connector pin and TV.
   - Temporary jumper from input D7 pin to ground not connected.
   - Crystal not oscillating. Check for pulses at pin 1 of IC27.
   - Horizontal countdown chain defective. Successively measure output at pin 3 of IC32, IC17 and IC21. Each should be progressively lower in frequency.
   - Vertical countdown chain defective. As above, but measure pin 3 of IC24, IC10 and IC12.
   - Defective video mixer. Look for pulses at pins 1 and 13 or IC26.

3. Initial checkout pattern (step 3) is poorly defined or lacking synchronization. In this case the following comments might apply.
   - TV could be overloaded by the ≅ 3 V of video. Cut the level by adding a series resistor of 10 ohms to see if sync and video stabilize.
   - Check for horizontal and vertical sync and blanking pulse at connector pin 16. A 75 ohm load should be attached. The pattern should look like this:

   a. If horizontal sync is defective, check IC11, IC20, IC18 and IC13.
   b. If vertical sync is defective, check IC19 and IC25.
   c. If horizontal blanking is defective, check IC11, IC13 and IC14.
   d. If vertical blanking is defective, check IC19.

4. No characters at step 4 of the checkout procedure. Look for:
   - Missing voltages at the MCM6571 (IC30).
   - Defective character generator.
   - Defective 74165 (IC31).
   - Defective logic signals to and from IC30 and IC31. All inputs and outputs should be pulsing at valid TTL levels (0 to 0.8 V = low; 2 to 5 V = high).

5. Wrong character(s) in display when driving from computer or manual testing of step 5 in checkout.
   - Miswired or misjumpered input.
   - Defective memory IC. Note the bit difference between the intended character. IC1 is the memory for the Least Significant Bit (LSB) of the character . . . and IC7 is the Most Significant Bit's (MSB) memory.
   - Defective 74157(s), IC8, IC15 and IC28.

6. "Twinkling" characters on TV. The source of this problem could be:
   - Slow memories. 650 ns or faster 2102s must be used.
   - Overheated memories. Access times increase with heat.
   - Wrong pulse levels at pin 1 of 74165 (IC31). A base level of about 2.5 V with short positive and negative going spikes should be seen.
   - Defective character generator, IC30.
   - Incorrect timing components on 74123, IC23.

7. Won't write characters into memory of TV readout. Look for:
   - Missing strobe pulse, or continuous level on D7 input.
   - No write pulse from 74123. Measure at pin 12 of IC23, looking for an ≅ 600 ns negative going pulse. Connecting the D7 input to a ≅ 50 kHz TTL clock will permit viewing on lower cost oscilloscopes.
   - Write clock not toggling. With above temporary oscillator inputting to D7, look for pulses at pin 3 of IC16, IC29 and IC9.
   - Defective memory address multiplexers, IC15, IC28 and IC8.

8. Extraneous characters can be caused by:
   - Noise on the input lines to the memory, particularly on the D7 line. A 220 pF condenser (C4) is used on D7 to suppress most noise sources. More or larger condensers may be required in extreme cases. This trouble often shows up as an α appearing on the screen when another port is addressed.
   - Data sent to the TV character generator faster than it can handle. Data must be valid for 1.5 us following the rise of D7 strobe. Faster data rates can be handled by reducing the value of the condensers in the 74123 write strobe singleshot. Alternatively, a data hold loop in your program, consisting of NOP instructions, can slow the data output to the readout.
   - Defective or slow memories. Look at the bit pattern of the extraneous character to determine if a single memory is bad in a single or several data locations.
   - More bypassing required. Power supply conditioning is shown in figure 2. Look at the power supply with a high speed scope – if excessive voltage glitches are present, add capacitance.

line appears, displaying the 33rd character. Up to 512 characters are thus sequentially entered and displayed. If a 513th and following characters are entered, the address wraps around so that an overwrite condition results: New characters start appearing at the top left corner of the screen. The display address may be reset to the home position at any time. Screen erase is accomplished either by loading 512 or more ASCII "spaces" (octal 240) followed by the home reset (octal 377), or by issuing the home reset followed by exactly 512 ASCII spaces, the latter being preferable.

Memory writing occurs when the MSB goes high. The memory address multiplexors (IC15, IC28, and IC8) then use the write address counter to control the memory address lines, interrupting normal display activity. 600 ns later, a 600 ns strobe pulse writes the new character into memory.

An excellent idea was suggested by Phil Mork in the *Digital Group Clearinghouse* to utilize a parallel logic path to step the write address counter without writing a character. Using a cycle of 511 write address steps, a blank, 511 write address steps and a non-blank character, a blinking "pseudo cursor" effect is obtained without the usual expense of a number of comparators. This software "blink" may be easily implemented with a final result indistinguishable from a hardware cursor. The write address stepping logic consists of IC19a and IC27d which detect the presence of a "1" in the least significant bit while the most significant bit is held low. This toggles the write address counter without firing the 74123 write strobe (IC23b). Disable the "pseudo cursor" when using a direct keyboard input. Do this by disconnecting pin 12 of IC27 from IC19, and tying pin 12 to +5 V (logical 1).

8080/Z80 Driving Software

This television display can be driven by a microprocessor's 8 bit output port. In the Digital Group systems, we use port 0 for this function. Listing 1 shows code for the routines CLEARTV, SPACE, and TVOUT to show how the software drivers are designed.

The main subroutine is labeled TVOUT and is located at <0> 372. The programmer merely loads the A register with one of the characters from the list in table 1 and calls the TVOUT subroutine. The codes in table 1 include all the standard upper and lower case ASCII codes, but have the high order bit of an 8 bit word set to "1". For those characters in table 1 which have ASCII graphics, subtracting 2 from the leftmost digit will give the equivalent 7 bit ASCII code (with the high order eighth bit assumed to be zero).

The instruction at <0> 370 will load the "space" character for you, so to get a space on the screen, merely call SPACE at address <0> 370.

Before attempting to write any character on the screen, the user must know where on the screen the character will appear. A third included subroutine starting at <0> 343 called CLEARTV will reset the write address counter to the home position and clear the 512 character screen. The next character entered after this subroutine will appear at the top leftmost position on the screen.

Listing 1: Utility software for driving the TV readout with an 8080 or Z80 system. This listing gives the CLEARTV, SPACE and TVOUT functions, a total of 28 bytes. The CLEARTV operation simply homes the display, then writes 512 spaces leaving a blank screen and the write address counter pointing to the upper left corner of the screen. The SPACE subroutine simply loads a space code into the accumulator (see table 1) then falls through into TVOUT. TVOUT simply outputs the value in the accumulator, then clears the accumulator and outputs all zeros so that the write strobe (D7) is turned off completing the write operation. This routine assumes a latched output port.

```
Split
Octal
Address      Octal Code      Label    Op.    Operand   Commentary

<0> 343   076 377          CLEARTV  MVI    A,377     A := '■' [set up home reset character];
<0> 345   315 372 <0>              CALL   TVOUT     write character [resets write address];
<0> 350   006 000                  MVI    B,0       } BC := 2000 [set loop count to split
<0> 352   016 002                  MVI    C,2       }       octal equivalent of 512];
<0> 354   315 370 <0>      CLEAR   CALL   SPACE     write one space on screen;
<0> 357   015                      DCR    C         C := C - 1 [low order count];
<0> 360   302 354 <0>              JNZ    CLEAR     if not C = 0 then reiterate the loop;
<0> 363   005                      DCR    B         B := B - 1 [high order count];
<0> 364   302 354 <0>              JNZ    CLEAR     if not B = 0 then reiterate the loop;
<0> 367   311                      RET              return with screen clear, write address
                                                      counter pointing to home position;
<0> 370   076 240          SPACE   MVI    A,240     A := ' ' [load one blank character code];
<0> 372   323 000          TVOUT   OUT    0         (port 0) := A;
<0> 374   257                      XRA    A         A := 0 [turns off strobe pulse in bit 7];
<0> 375   323 000                  OUT    0         (port 0) := A;
<0> 377   311                      RET              return from SPACE or TVOUT;
```

Entry points:

CLEARTV: Called with no parameters when TV display screen is to be cleared completely and left in the "home" (upper left) position. Uses registers A, B and C.

SPACE: Called when a space (ASCII 040, 240 from table 1) is to be sent to the TV display. Uses register A.

TVOUT: Utility output routine to transfer contents of A (high order bit assumed "1") to the TV display and increment the write address counter. Uses register A as input parameter, destroys its value leaving 0.

Conclusion

This television display design provides a versatile and essentially self contained circuit to provide the key output device of a small and inexpensive computer system. It can be built from scratch in the typical experimenter's laboratory or from a kit provided by Digital Group. Due to its use of an extended character set with 127 symbols including upper and lower case, special characters and Greek, the display will prove quite useful in a variety of applications.■

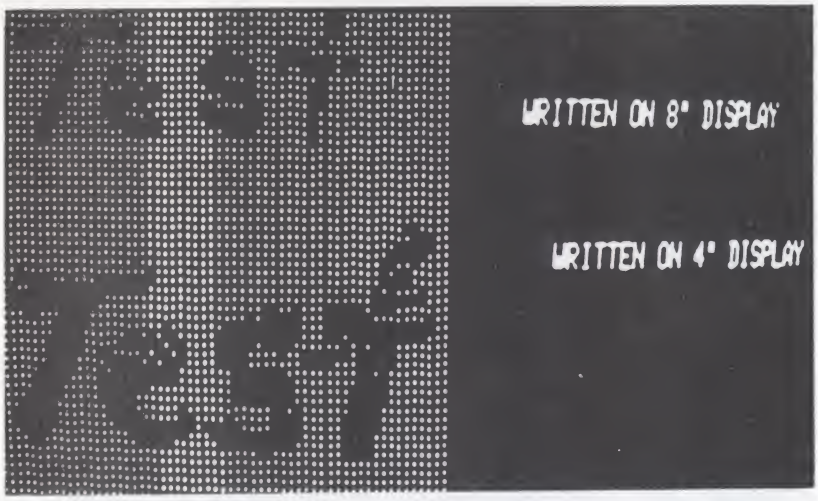


Photo 1: The ease of removing just one dot from a full field display depends upon the display size. The author's X-Y display has an adjustable size control which was used in preparing this picture.

Let There Be Light Pens

Sumner S. Loomis
Loomis Laboratories
Route 1 Box 131A
Prairie Point MS 39353

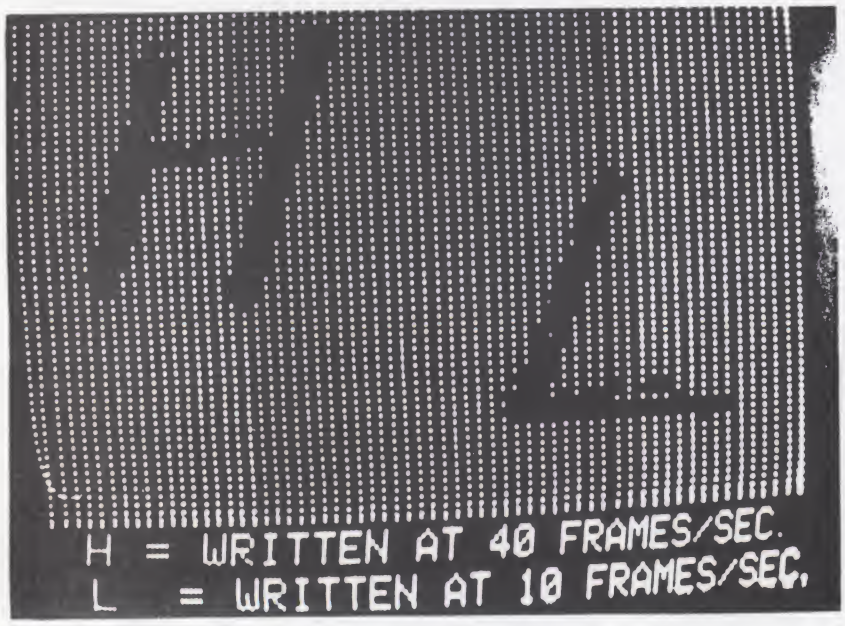


Photo 2: The light pen can be used in an "erase" mode by filling the screen with "on" dots then selectively removing dots with the light pen. The titles added to this picture (and all the pictures in this article) were created with a separate character generator which is not described.

With only a few components and a few hours of construction you can add a versatile light pen to the oscilloscope graphics interface which has been described in the October 1975 issue of BYTE, page 70 ff.

By holding the light pen to the face of the cathode ray tube (CRT), a point may be added or removed. This eliminates the awkward and time consuming effort required when using a program or manual switches to change the dots on the screen.

The resolution and capability of the light pen are dependent on two characteristics of the CRT. The brightness and the size of the display tube will determine how easily you may add or remove one dot. An idea of the effect of display size may be had from photo 1. The word Test was written twice on a 12 inch (30.5 cm) black and white TV picture tube configured as an XY display like an oscilloscope. The top word was written with the display adjusted to an 8 inch (20.3 cm) size, and the lower word was written with a 4 inch (10.2 cm) display. Each letter was written with only one stroke of the light pen without touch up or corrections. With some practice, and possibly several passes, one dot may be added or removed if the display measures 8 inches (20.3 cm) or more. Further improvements to the pen are required with smaller display tubes. An advanced circuit that greatly improves the capability of the pen with small displays is also described in this article.

The light pen can erase or draw depending on the setting of a switch. Examples of the two actions may be seen in photos 2 and 3. If the oscilloscope interface is adjusted for a high repetition rate, some smearing or carry over into the neighboring dot positions occurs. The author's system has a front panel control permitting ten repetition rates. A small improvement in resolution can be noted at the lower writing rates, as shown in photo 2. A frame consists of 64 by 64 dots.

Theory of Operation

The light pen operates on the principle that brightness is quite intense during the actual interval that a particular dot is being written by the CRT's electron beam. Although phosphor will continue to emit light for some time, the brightness decays in an exponential manner after the writing beam has moved on to the next dot.

Figure 1 illustrates the simple light pen circuit. With proper adjustment of the sensitivity control (and possibly the brightness control), the photocell in the tip of the light pen will sense the moment in time when a dot is written at the particular location of the light pen. At this instant, the photocell will conduct, biasing the PNP transistor which causes a short pulse to be conducted through capacitor C1 to the base of the NPN transistor Q2. If the pulse is greater than .6 V, this transistor will be driven into saturation, and the light pen output will fall to .3 V. This output line is the connected to pin 5 of the oscilloscope graphics unit which writes a 1 or a 0 bit (dot or no dot) at precisely the instant that the dot position touched by the pen was addressed.

The above procedure works quite well if the dot to be changed is illuminated at the time. With proper adjustment of the sensitivity control, we can usually use an illuminated dot just above the point of action (it must precede the dot in scan sequence) to create a new dot in the next space. This action of extending a line can be quite useful for drawing bar graphs on the CRT. This mode of entry is possible because screen persistence allows the light pulse to be carried over into two or three subsequent dot positions depending on the frame speed.

How can the photocell sense the dot's position if there is not any illumination to trigger it? This is accomplished by the flood circuit which is shown in figure 2. This circuit overrides the normal Z-axis control and floods the screen with light by feeding a logical one signal to the Z axis of the display unit. With this arrangement the pen is placed at the required dot position, the footswitch is actuated to flood the screen with light, and the photocell is energized when the

Photo 3: The light pen can be used in an "enhance" mode by using a footswitch control to flood the screen momentarily when the light pen is in position.

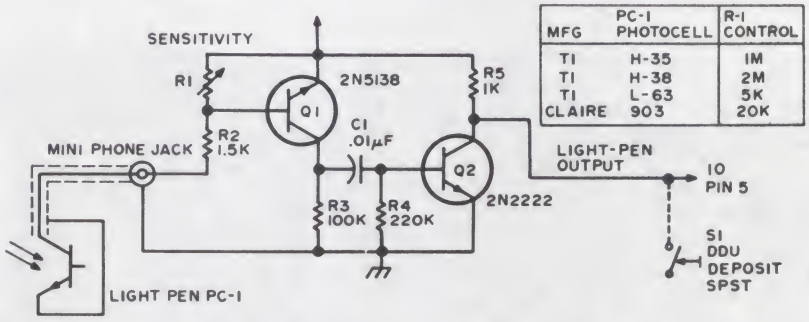


Figure 1: The simple version of the light pen can be constructed according to this schematic. All resistors ¼ W.

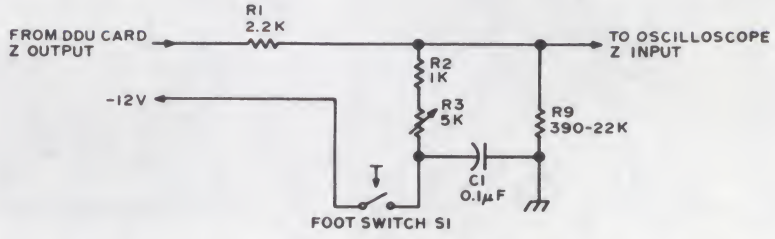


Figure 2: The foot switch control used to flood the screen for "enhance" mode operation is given in this circuit which modifies the Z-axis signal to the oscilloscope driver.

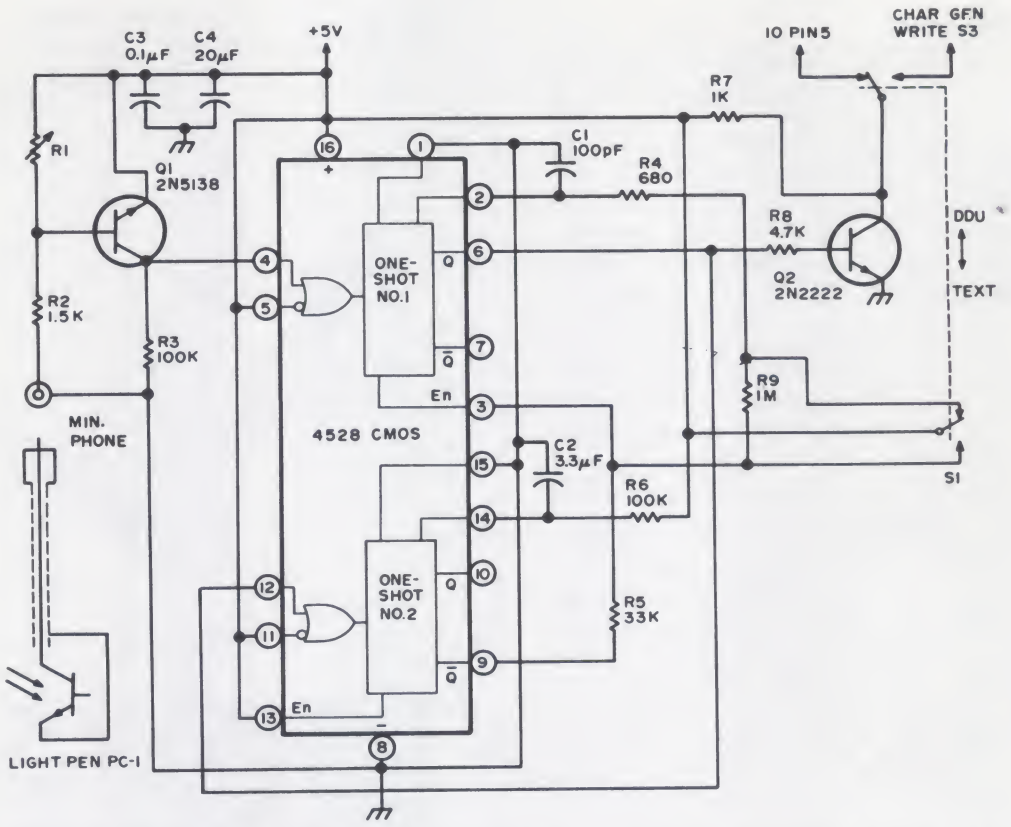


Figure 3: By adding a pair of oneshots to the circuit, the ability to draw pictures is improved through a short data lockout period which avoids smearing.

writing beam reaches that particular dot position. Releasing the footswitch removes the flood and allows the data to be examined.

The circuits just described will probably suffice if you wish to use the light pen only for occasional correction of data. If you plan extensive and detailed work, such as cartooning or statistical data entry, a modification of the circuit will allow you to tailor the light pen's response to your own particular needs and system speed. The circuit shown in figure 3 is similar to the one shown in figure 1. However, it includes two oneshot multivibrators (contained in one CMOS DIP). The first one produces a constant amplitude pulse of approximately 200 nsec duration which is sufficient to bring about the storage of a 1 or 0 bit in most versions of the 2102 memory (ICs 11 to 14 in the oscilloscope graphics interface). The second one delays the generation of another write command for .25 sec, giving the operator sufficient time to withdraw the pen from the screen or move to a new location, before a double or multiple dot can be drawn. Once the two pulses have been timed in accordance with a given system speed and the operator's writing speed, it becomes very easy to draw detailed images with the light pen.

In figure 3 resistor R4 and capacitor C1 control the length of the write pulse, and resistor R5 and capacitor C2 control the wait time. For the 4528 CMOS oneshot, the time of the pulse (T) measured in microseconds is a function of resistance (R) and capacitance (C) measured in ohms and microfarads, respectively, as follows:

```
T = 2.5 * R * C ** .85;
```

where a single asterisk denotes multiplication, and a double asterisk exponentiation.

The circuit shown in figure 3 also includes a switch and connections for using the light pen with the author's text display and editing system. Exact details for this connection are not given here, as they will differ with the type and construction of the text display system. I found, however, that the shift register type memories commonly used in these systems require a much longer write pulse than is necessary for the 2102 memories. It was also desired to eliminate the holdoff circuit (second oneshot) for this application. These changes are accomplished with switch S1 and resistors R9 and R5. If these features are not desired, it is recommended that R9 be replaced by a wire, R4 changed to 4.7 kΩ and C1 to 20 pF.

Construction

As is shown in the table accompanying figure 1, several different types of photocells are suitable for use in the light pen. The Texas Instrument (TI) type H-35 or H-38 is a very small device with a built in lens. These were originally designed for use in punched tape and card readers, thus the small size. Their size, sensitivity, and restricted field of view make them ideal for this application. The high impedance of these devices, however, makes them somewhat slow for this application, particularly at low brightness levels. The slow response time limits their use at the faster scan rates, and complicates the smearing mentioned earlier. Another device, the L-63 type which is available from Radio Shack (276-140 infared detector), was found to be considerably faster. Being a much larger device, however, it has a larger field of view, and much of its speed advantage is lost to optical smearing. Models of both photocell types were built and tested by me, with only slight preference for the H-35. With some careful masking, and possibly the addition of a small, short focal length lens (e.g., Edmund Scientific number 12050 cylinder lens, or a small drop of clear epoxy), this photocell will probably perform better than the H-35 for this application. The Claire types 903 and 903-L were tried with only fair results.

Any ball-point pen or felt-tipped marker can be reworked to make a housing for your light pen. Take a tour of the local stationery store to find likely candidates. The L-63 photocell was found to fit nicely into the end of a Graphi-100 marker pen which can easily be disassembled with diagonal cutters. An example of the construction with the L-63 is shown in photo 4, and the H-35 assembly is shown in photo 5.

Secure the photocell in place with epoxy adhesive after attaching the shielded cable. The cable can also be secured against damage from pulling by filling the entire pen with silicone rubber adhesive or ordinary household bathtub caulk. It is wise to keep the cable short, especially with the H-35 or H-38 photocells, to obtain maximum possible response speed. I used an 18 inch (45.7 cm) long miniature coaxial cable leading to a miniature phone plug.

If you are using the simple circuit of figure 1, the parts can be assembled on a small turret terminal board available at most electronic supply houses. This assembly is shown in photo 6. The circuit of figure 3 can be assembled in the same manner with the addition of a 16 pin DIP socket. R4, R5, C1 and C2 should be mounted in such a manner that they can be changed easily (Cambion 601-1512 component clips are useful here).

Photo 4: This shows a pen based on the TI type L-63 photocell, built using a marking pen case.

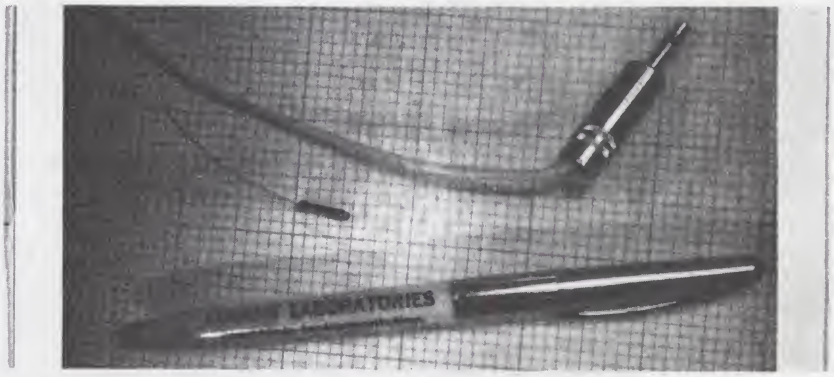


Photo 5: This picture shows an assembled light pen using a TI type H-35 (or H-38) photocell with a standard ballpoint pen housing.

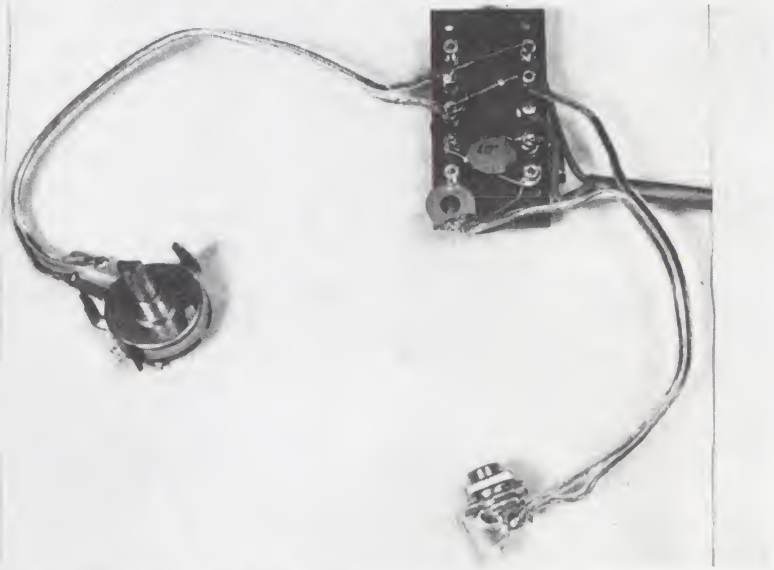


Photo 6: This photo illustrates how the circuit of figure 1 can be assembled using a small turret terminal board. The transistors and R5 are mounted out of sight on the rear side of the board.

Photo 7: Using the improved circuit of figure 3 reduces much of the over-writing of multiple dots which occurred using the original circuit of figure 1. This is an enhance mode picture.

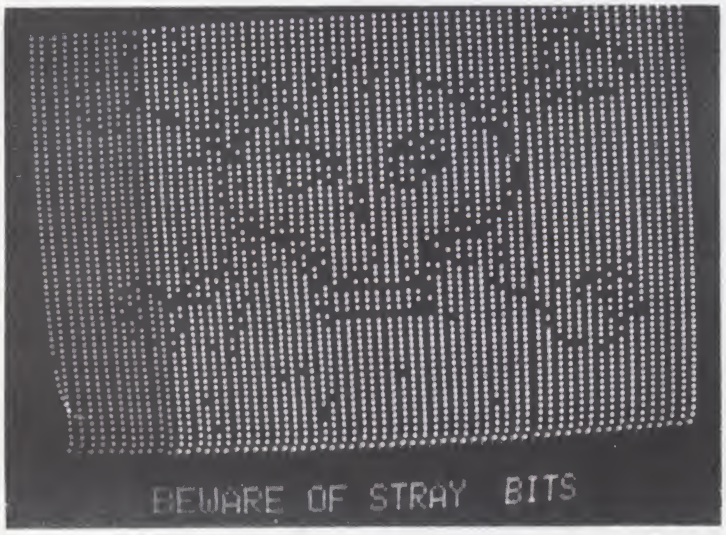


Photo 8: This illustrates a cartoon drawn using the erase mode of operation with the improved circuit of figure 3.

The sensitivity control may be conveniently mounted on the front panel.

The operation of the light pen requires control of the inputs to the oscilloscope graphics unit. I have found that one of the most convenient ways in my system is through a set of manual data switches. This type of input was illustrated as a test fixture for the oscilloscope graphics interface in figure 5 on page 75 of the October 1975 issue of BYTE. In my system, these data input switches are shared with a Mark-8 minicomputer front panel by means of an 8 pole double throw toggle switch. It is also possible to set up input codes to the oscilloscope graphics unit using software in the microcomputer system which drives it.

In order to enter data with the light pen, a deposit switch is pressed whenever the pen is in the proper position for data entry. The deposit switch should be mounted in a convenient location near the display tube and light pen. In my system the light pen deposit switch was mounted next to the original deposit switch of the Mark-8 computer.

Using The Light Pen

To illustrate the use of the light pen, we will cover the procedure necessary to draw a simple figure on the screen in the erase mode using manual controls. Set the switch register to 1000 0110 binary (turn scan on) and depress the deposit switch. This should produce random dots on the screen. Set the switch register for 1000 0010 binary (set Z on) and depress the deposit switch again. The screen will show a full field of dots. (If the Z axis polarity of your display tube is reversed, you will have to use the "set Z off" command (1000 0011 binary) to illuminate the screen.) Set the switch register for 1000 0010 (set Z on), but *do not* activate the deposit switch. Now bring the light pen in contact with the display CRT, and note that the dot or dots within its field of view are erased. To erase the entire screen and start over, simply press the deposit switch and repeat the above procedure.

To write in the enhance mode (screen dark, writing illuminated dots), reverse the above procedure by wiping the screen clean with the "set Z off" command (while the scan is on), and after setting the switch register to "set Z on" without the deposit switch, proceed to write dots with the light pen. In this mode, the flood foot switch must be periodically activated to provide the required illumination. Examples of the light pen's drawing capability can be seen in photos 7 and 8. ■

Build an Oscilloscope

Ever wonder how to make a computer draw pictures for output? One way is to use an oscilloscope – which many readers have on general principles for debugging the logic circuitry. Jim Hogenson provides a practical circuit for accomplishing that end in his "Oscilloscope Graphics Interface" design. This graphics device was conceived by Jim as a neat idea to add to the 8008-oriented computer system he was building for a high school science fair. He first mentioned it to me in a letter late last year. I suggested to him (or was it the other way around?) that it might be appropriate to turn it into an article for the *ECS Magazine* I was publishing at the time. After a fair amount of time spent researching the various options – plus one lengthy phone conversation with me – Jim settled on the design shown in this article, which is reprinted here from its original publication in the last issue of *ECS Magazine*. The interface is very simple, and can be adapted to virtually any computer with a minimum of 8 parallel TTL output lines and a clock pulse line which is active when output data is stable. Arrangements have been made for a PC version of this design (see the parts list, Fig. 6) so you won't have to wire wrap the thing like Jim did in his first version.

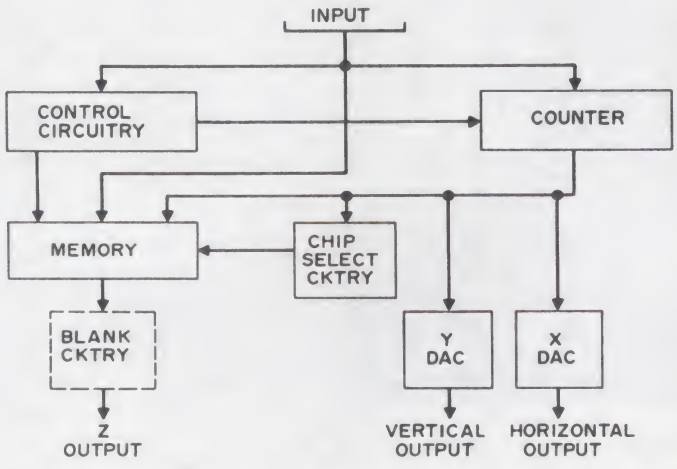
. . . CARL

by
James Hogenson
Box 295
Halstad MN 56548

Many members of the large family of alphanumeric computer output devices may be readily used in the home computer system. But there are as yet few devices of a graphic orientation which are economically acceptable in the home computer system. The oscilloscope graphic interface project presented here provides one unique, inexpensive and uncomplicated solution to the graphic output problem in small scale systems. It turns an essential test instrument – the oscilloscope – into a versatile output device.

The oscilloscope graphic interface is programmed and operated through a parallel 8-bit TTL compatible input. An image is represented by a pattern of dots which is organized according to the computer's instructions. During the scan cycle, the digital dot pattern is converted to analog waveforms which reproduce the image on an oscilloscope screen. The graphic interface stores the dot pattern within its own internal refresh memory. Therefore, once the pattern has been generated and loaded into the graphic interface memory, the computer is left free to execute other programs.

Fig. 1. Oscilloscope graphics display block diagram.



Principle of Operation

The raster begins its scan in the upper left-hand corner, scanning left to right and down. The full raster contains 4096 dots, 64 rows of 64 dots each. The horizontal scan is produced by a

Graphics Interface

Fig. 2. Oscilloscope graphics interface instruction codes.

Op Code Binary	Octal	Mnemonic	Explanation
00dddddd	Odd	STX	Set X
01dddddd	1dd	STY	Set Y
10xxx000	2x0	DCY	Control – Decrement Y
10xxx001	2x1	TSF	Control – Turn off scan
10xxx010	2x2	ZON	Control – Set Z on
10xxx011	2x3	ZOF	Control – Set Z off
10xxx100	2x4	ZNI	Control – Set Z on with increment
10xxx101	2x5	ZFI	Control – Set Z off with increment
10xxx110	2x6	TSN	Control – Turn on scan
10xxx111	2x7	DCX	Control – Decrement X
11xxxxxx	3xx	CNO	No Op

d = data x = null

stepping analog ramp wave. Each of the 64 steps in the ramp produces one dot. The vertical scan is similar. It is a stepping ramp wave consisting of 64 steps. However, there is only one step in the vertical wave for each complete horizontal ramp wave. The result is 64 vertical steps with 64 horizontal steps per vertical step, or 64 rows of 64 dots each.

The timing of horizontal and vertical sweep waveforms originates in a 12-bit binary counter, the operational center of the entire circuit. The six least significant bits of the counter are connected to a digital-to-analog converter (DAC) which converts the digital binary input to a voltage level output. The output of the least significant DAC is the horizontal ramp wave. The six most significant bits are connected to a second DAC. This DAC produces the vertical ramp wave. Incrementing the 12-bit counter at a frequency of around 100 kHz results in a raster on the screen of the oscilloscope.

The contrast in the pattern of dots needed to represent a picture is dependent upon the intensity of each dot. From this point, it is assumed that a dot can be either on or off. An "on" dot will show up on the screen as a bright dot of light. An "off" dot will be a dim dot of light.

When a particular dot is addressed by the counters, it may be set to either the "on" or the "off" state. The on-off control is represented by a single bit. It is this bit which is stored in the internal memory of the oscilloscope graphic interface. There is one bit in the memory for each of the 4096 dots in the raster. When displaying the image, the 12-bit counter which produces the raster addresses the appropriate dot status bit in the memory as that dot is produced on the screen. The on-off dot status bit taken from the memory is converted to a Z-axis signal which controls the intensity of the dot on the screen.

The major portion of the circuitry is taken up in the 12-bit counter, the DACs, and the memory. Fig. 1 shows a block diagram of the oscilloscope graphic interface. The remaining circuitry is the control circuitry which decodes the 8-bit input word and allows for completely programmed operation.

Programming

The programming instruction format is shown in Fig. 2. Bits 7 and 6 of the input word are the high-order instruction code. It is assumed that the addressing of dots is done on the basis of X and Y coordinates. The X coordinate is the 6 bits in the least significant or horizontal section of the 12-bit counter. The Y coordinate is the 6 bits in the most significant or vertical section of the counter. In programming from an 8-bit microcomputer source, all 12 bits of the counter cannot be set at once. The counter is set one half or 6 bits at a time. It is for this reason X and Y coordinates are assumed in programming.

When the instruction code (bits 7 and 6) is set at 00, the data on bits 0 through 5 of the input word is loaded into the least significant counter section as the X coordinate. When the instruction code is set at 01, the data on bits 0 through 5 is loaded into the most significant counter section as the Y coordinate. In effect, the Y coordinate will select a row of dots, while the X coordinate will select one dot in the selected row. The coordinates loaded into the counter will address

the memory and select the desired dot status bit for programming.

After loading the coordinates of the dot selected for programming, the status of the dot (on or off) is set using the ZON, ZOF, ZFI or ZNI control codes. Setting the instruction code at 10 directs the control circuitry to decode the three least significant bits of the input word for further instruction. The three least significant bits are called the "control code."

Since the 12-bit counter must store selected coordinates during programming, the raster scan must be disabled before programming. Control code "1" will stop the scan. Control code "6" will restart the scan. When the scan is on, the 12-bit counter will be incremented at a high frequency and the programmed image is displayed on the scope screen.

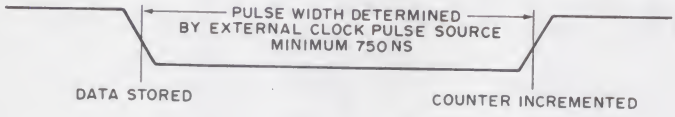
Control code "2", "set Z on", will program a bright dot to appear at the dot location presently stored in the counter. Control code "3", "set Z off", will program a dim dot or blank to appear at the dot location presently stored in the 12-bit counter.

Control codes "4" and "5" set Z in the same manner as control codes two and three. However, after setting Z, these instructions will increment the counter by one thus advancing to the next dot location in the raster scan pattern. This will allow programming of the entire raster using only a repeated "set Z" instruction.

Control code "0" will not set Z, but will decrement the most significant or vertical section of the counter only. In effect, control code "0" will decrement the stored Y coordinate. Control code "7" will not set Z, but will decrement the entire 12-bit counter by one. This, in effect, will decrement the stored X coordinate. Since the X and Y counter sections are cascaded, Y will automatically be incremented or decremented once for every 64 executions of an increment or decrement X control code.

The increment and decrement control codes are very useful in constructing lines in an image since lines require repeated "set Z" instructions, often on the same axis. An effective method of clearing an image from the screen is repeating a "set Z with increment" control code in a programmed loop. This method allows the option of using either a light or dark image background.

Fig. 3. Timing pulse input to the interface. The 8 data lines must be stable during this pulse.



Circuit Operation

Once the data word on the microcomputer parallel output interface is stable, one clock pulse is used to execute the instruction. This clock pulse is taken from the microcomputer output interface. The instruction code is decoded by the 7410 triple three-input NAND gate and two inverters. The clock pulse is enabled by the NAND gate to the appropriate counter section, or to the strobe input of the control code decoder.

The 12-bit counter consists of two 6-bit counting sections. Each section consists of two cascaded TTL 74193 presettable binary counters. Bits 0 through 5 of the data input are common to both sections of the counter. The set X instruction will pulse the load input of the least significant or horizontal section, while the set Y instruction will pulse the load input of the most significant or vertical section of the counter. A pulse on the load input will cause the data on bits 0 through 5 to be loaded into the proper counter section.

Four TTL counters must be used to provide independent loading capabilities for each 6-bit section. The counters within each section are cascaded in the normal fashion. The two sections are cascaded by connecting the upper data B output of the X counter section (IC 8, pin 2) through inverter "a" of IC 2 to the count up input (IC 9, pin 5) of the Y counter section. The inverter is needed to provide proper synchronization in high frequency counting.

The control code is decoded by a 74155 decoder connected for 3 to 8 line decoding. Bits 0 through 2 are decoded by the 74155. The control code is enabled by the pulse coming from the 7410 instruction decoder only when the instruction code is set at 10 on bits 7 and 6.

Decoder lines 1 and 6 are connected to an R/S flip flop

Fig. 4. PC artwork of the graphic interface, by Andrew Hay.
(a) Component side.

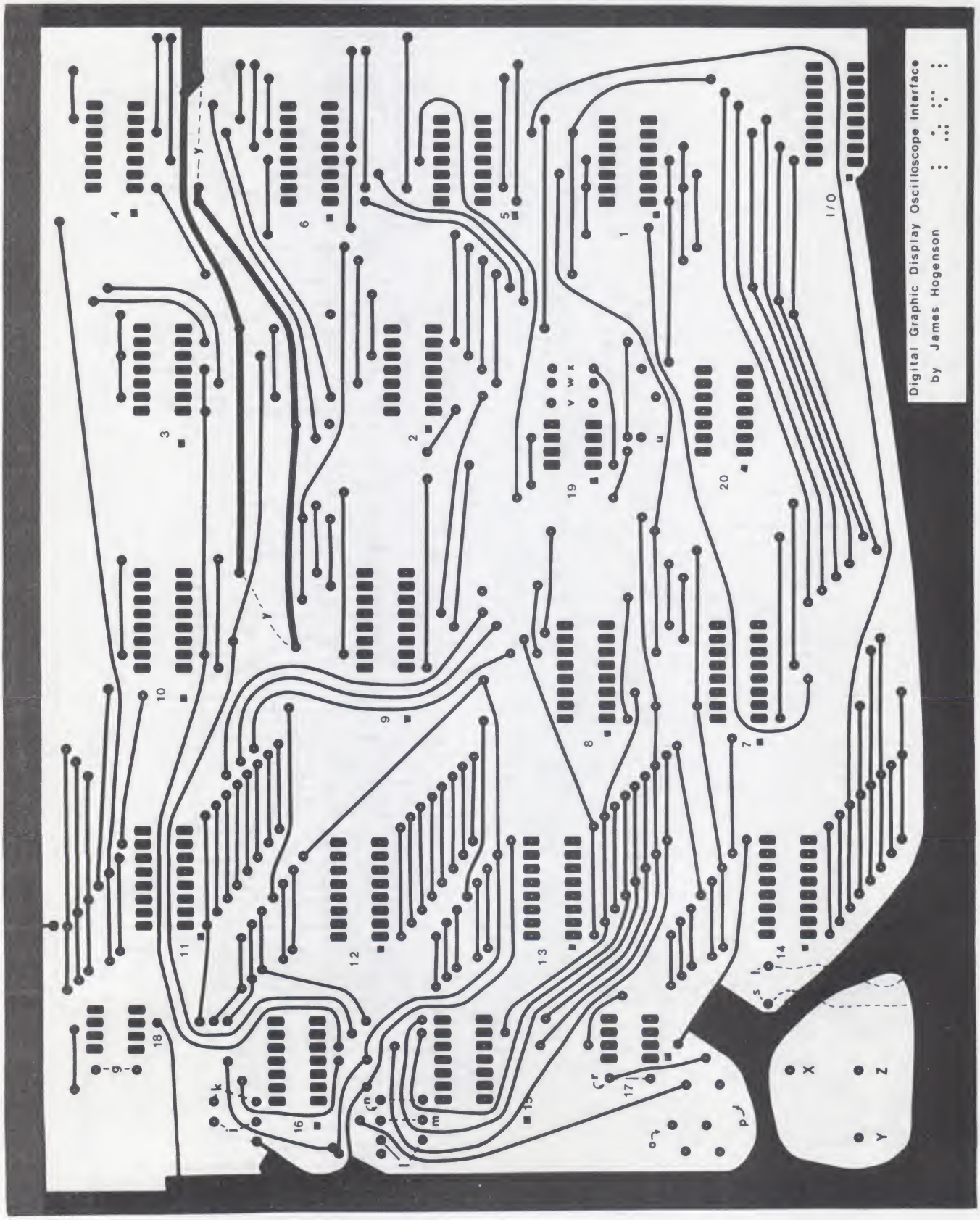
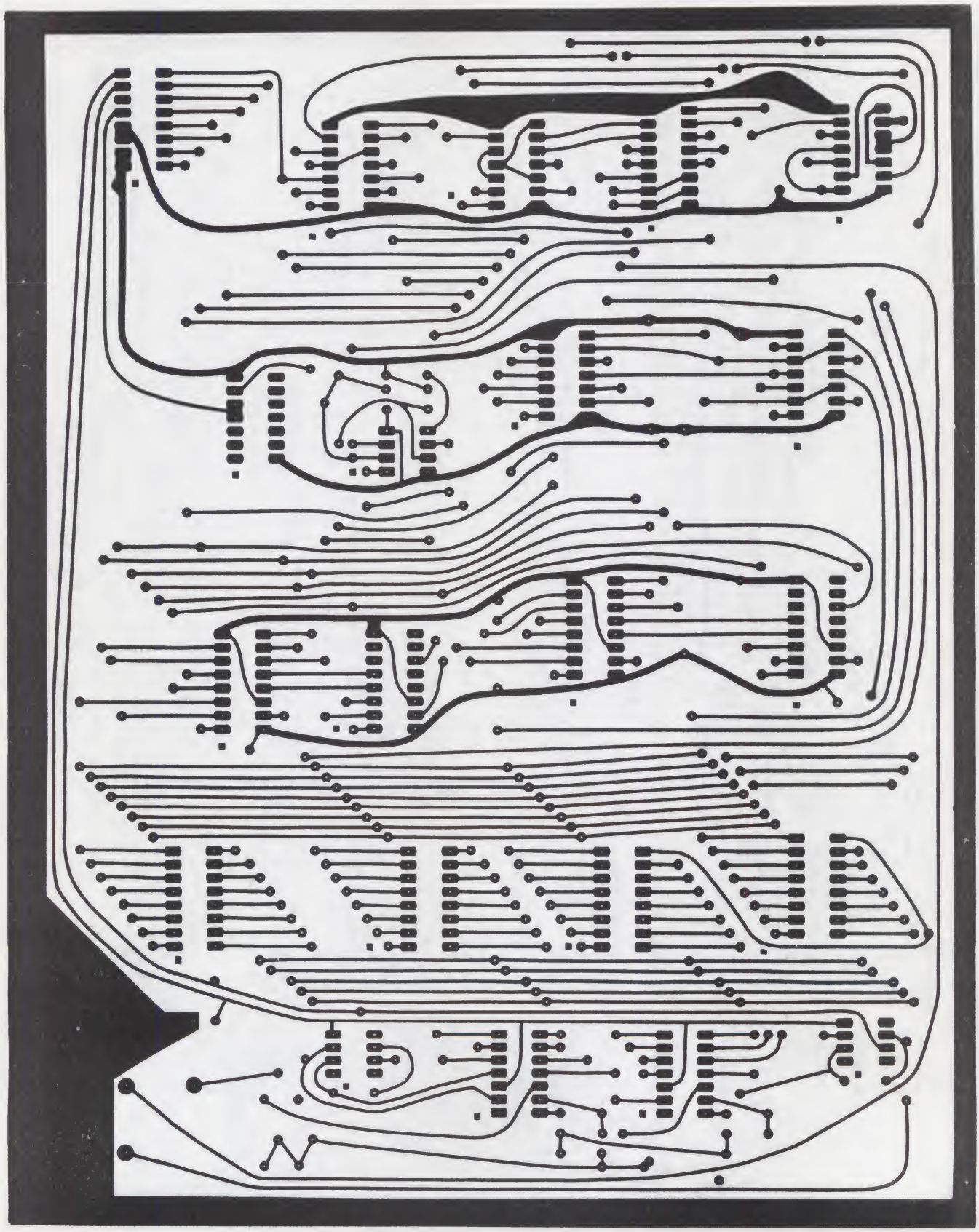


Fig. 4. PC artwork of the graphic interface, by Andrew Hay.
(b) Solder side.



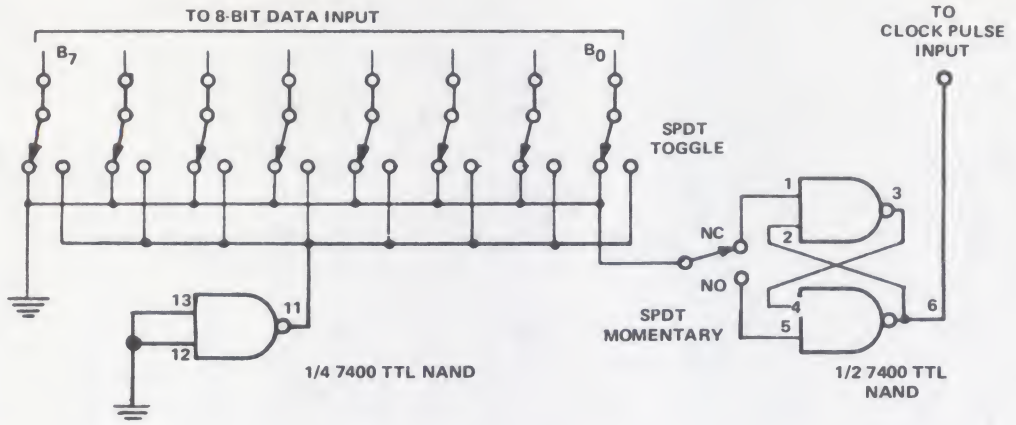


Fig. 5. A test circuit for manual operation. The set-reset flip flop of the 7400 circuit generates a debounced clock pulse which will perform the operation set into the toggle switches. If you haven't got a computer up and running yet, the manual interface can be used in order to test out the display.

which provides the scan on/off control. The flip flop enables the system clock to provide the high frequency square wave which increments the 12-bit counter.

Control codes 2 through 5 define the "set Z" instructions which perform a data write operation. Decoder lines 2, 3, 4 and 5 are connected to a group of AND gates (IC 5a, b, c) functioning as a negative logic OR gate. The output of this gate is the Read/Write control line for the memory. When this line is in the low state, the data present on the data input line of the memory will be written into the memory location presently stored in the 12-bit counter.

The data input of the memory is connected directly to bit 0 of the 8-bit input word. This bit is stored in the memory only when a set Z command is executed. The Z-axis circuit configuration will require a high state pulse for a blank or dim dot. As shown in the binary instruction format, Fig. 2, bit zero will be binary zero for "set Z on" instructions and binary one for "set Z off" instructions. The backward appearance of this binary format will be overlooked when programming in octal notation.

The high frequency system clock controlled by the R/S flip flop and decoder lines 4 and 5 are negative logic ORed. The resulting pulse increments the counter according to control commands.

The same clock pulse taken from the computer output interface is used to write data into the memory and increment the counter in control commands 4 and 5. The data is written into the memory on the leading edge of the pulse. The counter is incremented on the trailing edge. Fig. 3 shows the waveform timing.

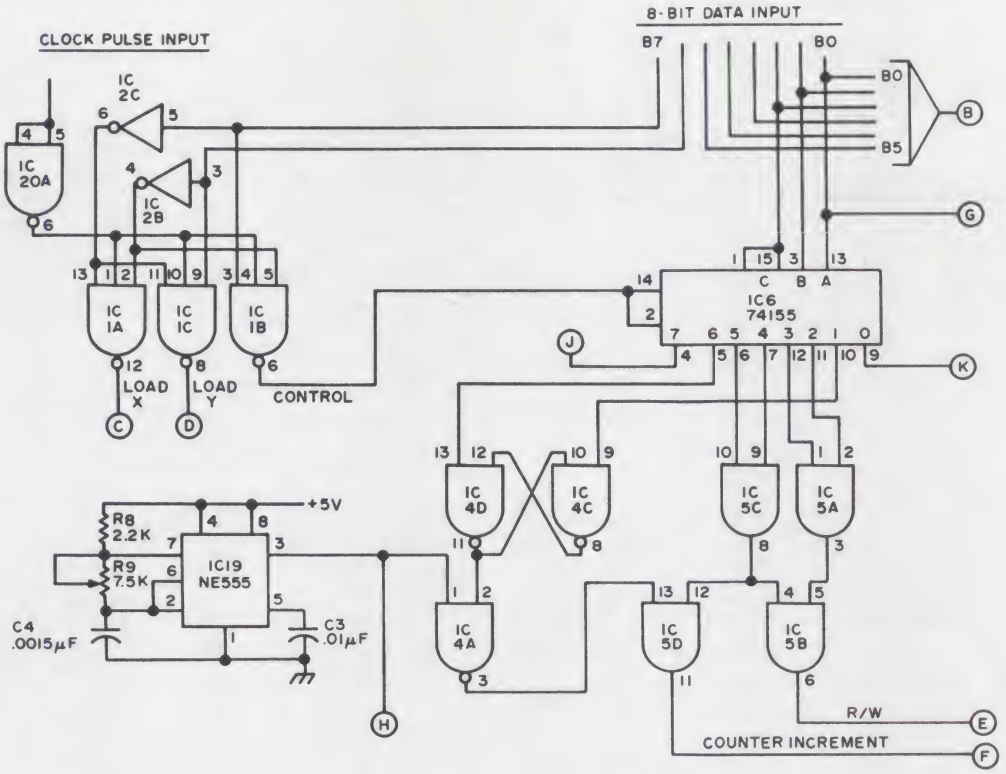
Output bits 0 through 9 of the 12-bit counter are connected to the address inputs of the memory. The memory uses four MM2102 type 1k x 1 bit MOS RAMs (Random Access Memories). Bits 10 and 11 of the counter output are connected to the chip select circuitry which

Fig. 6. Parts list.

C1, C2	20 pF	disc capacitor
C3, C5, C6-C11	.01 mF	disc capacitor
C4	.0015 mF	disc capacitor
C12	25 mF	electrolytic capacitor
IC 1	7410	TTL triple 3-input NAND gate
IC 2	7404	TTL hex inverter
IC 3, IC 4, IC 20	7400	TTL quad 2-input NAND gate
IC 5	7408	TTL quad 2-input AND gate
IC 6	74155	TTL dual 2-to-4-line decoder
IC 7 – IC 10	74193	TTL presettable 4-bit binary counter
IC 11 – IC 14	2102	NMOS 1024-bit static RAM
IC 15, IC 16	MC1406	Motorola 6-bit DAC
IC 17, IC 18	741	Op amp
IC 19	NE555	Oscillator (timer IC)
R1, R2	3.3k Ohm	resistor
R3, R4	5.6k Ohm	resistor
R5, R6	10k Ohm	miniature potentiometer
R7	1k Ohm	resistor (all resistors ¼ Watt, 10%)
R8	2.2k Ohm	resistor
R9	7.5k Ohm	miniature potentiometer

A printed circuit board using the masks of Fig. 4 is available for $29.95. Write to M. F. Bancroft, CELDAT Design Associates, Box 752, Amherst NH 03031.

Fig. 7. Oscilloscope graphics interface circuit diagram. (a)



enables one memory chip at a time for addressing and data input/output operations. The chip select circuitry uses 2 inverters and a TTL 7400 Quad two-input NAND gate.

The data outputs of the RAMs are OR-tied and connected to an AND gate. The data output is synchronized with the high frequency clock for better blanking performance. The output of this gate is connected to the Z-axis blanking circuitry. The blanking circuitry converts the TTL level signal to a scope compatible signal which may be varied over a wide range of output voltages to best match the scope being used.

Bits 0 through 5 of the 12-bit counter are connected to the X coordinate DAC. Bits 6 through 11 are connected to the Y coordinate DAC. The DACs are Motorola MC1406 ICs. The DACs operate on voltages of +5 and -5 to -15. A current output is produced by the DACs. The current output is converted to a voltage output and amplified by the 741 op amps. The output from the X coordinate amp is connected to the horizontal input of the scope. (The scope should be set for external horizontal sweep.) The output from the Y amp is connected to the vertical scope input.

Although the scope used does not need dc-coupled inputs, triggered sweep, or high frequency response for this project, a Z axis or intensity input is required. The Z axis output provided on the interface PC pattern is TTL compatible only. Most scopes will need some type of blanking circuitry to amplify the TTL level pulses. The design of the blanking circuitry will be of the builder's choice, allowing the builder to best suit his scope. A suggested method which is simple and effective is the use of the circuit shown in Fig. 13.

Construction

This project may be wire wrapped, the PC artwork in Fig. 4 may be used to fabricate a double-sided printed circuit board, or the printed circuit board product mentioned in the parts list may be employed. The PC pattern is designed for easy soldering. The components need be soldered on the bottom side only.

Remember that the memory ICs are MOS devices and should be handled as such. Static electricity will easily puncture the thin MOS transistor junctions.

Bypass capacitors should be connected between supply voltages and ground. A minimum of a 10 mF electrolytic or tantalum capacitor should be used for all supply voltages. For the +5 logic supply, one .01 mF disc capacitor should be used for each 2 to 5 integrated circuits. The large

electrolytics will filter out low frequency noise and voltage transients while the small disc capacitors will filter out high frequency noise which could falsely trigger flip flop and counter circuits.

Set-up, Testing and Operation

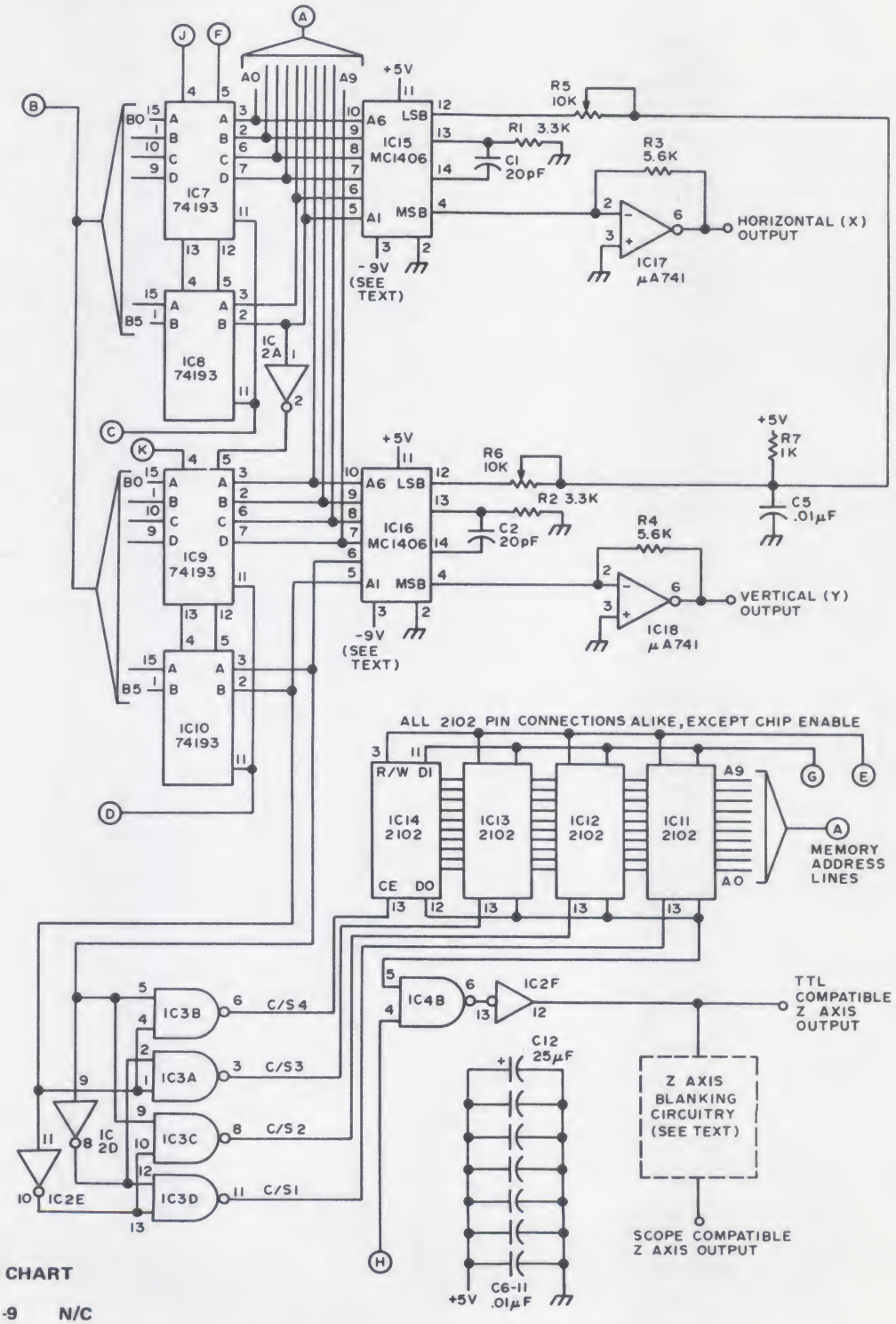
The system requires a +5 volt, 400 mA power supply and a dual polarity supply of from ±9 to ±15 volts at 10 mA. The wide range of analog supply voltages allows use of existing power supplies for the graphic interface.

The clock pulse derived from the computer parallel I/O interface should be active in the low state. If a device operating with an active high pulse is used, one of the free gates of IC 20 may be used to invert the clock pulse or IC 20 may be omitted.

When ready for testing, be certain of voltage supply polarities, then apply power. If the scan does not come on at random, execute a "turn on scan" command. Using the 10k Ohm pots, R5 and R6, adjust the DAC voltage references to eliminate any distorted concentration of dots in the raster.

The system clock consists of a 555 timer IC connected as an astable multivibrator.

Fig. 7. Oscilloscope graphics interface circuit diagram.(b)



IC POWER AND PIN CONNECTION CHART

IC	+5	GND	+9	-9	N/C
1,2,3,4,5	14	7			
6	16	8			9,4
7, 9	16	8, 14			
8, 10	16	8, 14			6,7,9,10,12,13
11,12,13,14	10	9			
15,16	11	2		3	1
17, 18			7	4	1,5,8
19	4,8	1			
20	14	7			1,2,3,8,9,10,11,12,13

2102 MEMORY ADDRESS PIN CONNECTIONS

A-0 – pin 8 : A-1 – pin 4 : A-2 – pin 5 : A-3 – pin 6
A-4 – pin 7 : A-5 – pin 2 : A-6 – pin 1 : A-7 – pin 16
A-8 – pin 15 : A-9 – pin 14

Fig. 8. CLEAR Program flow chart.

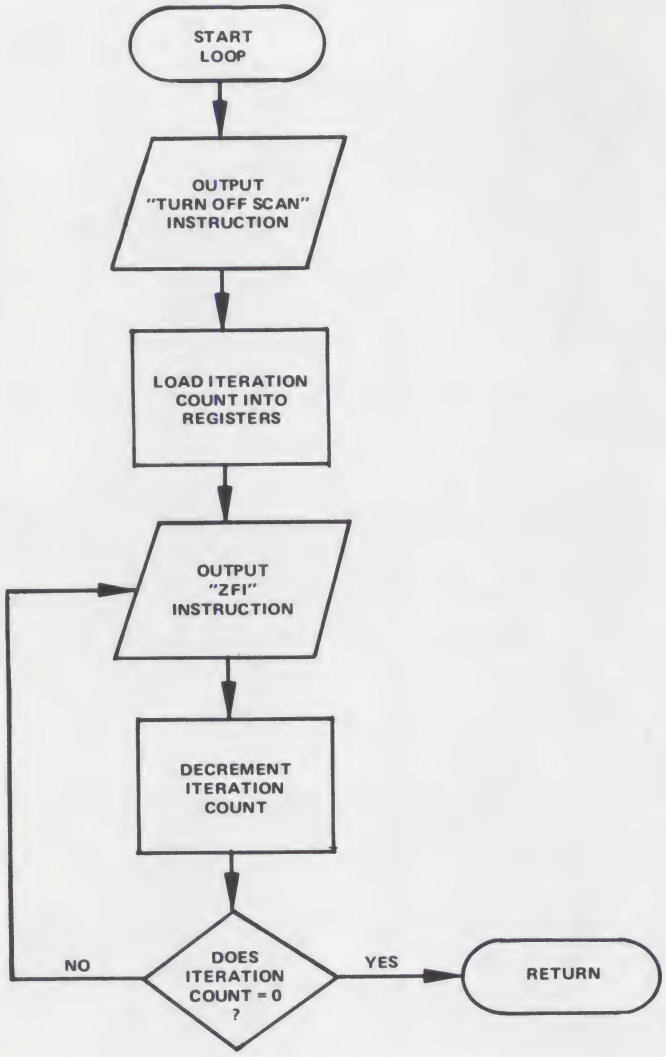


Fig. 9. Listing of 8008 code for the CLEAR program.

```
START
                00/344 = 006    LAI
                00/345 = 201    (TSF)
                00/346 = 121    OUT 10
                00/347 = 006    LAI
                00/350 = 205
                00/351 = 016    LBI
                00/352 = 377
                00/353 = 026    LCI
                00/354 = 021
                00/355 = 121    OUT 10
                00/356 = 011    DCB
                00/357 = 150    JTZ
                00/360 = 365
                00/361 = 000
                00/362 = 104    JMP
                00/363 = 355
                00/364 = 000
                00/365 = 021    DCC
                00/366 = 110    JFZ
                00/367 = 355
                00/370 = 000
                00/371 = 377    HLT
```

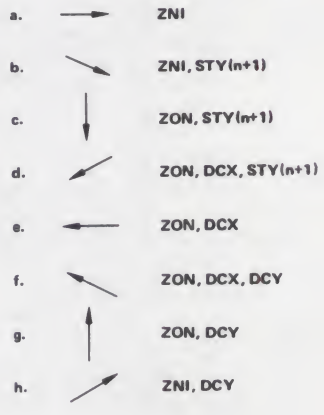
Adjusting the frequency may be necessary to obtain a stable raster. The frequency is adjusted using R9, the 7.5k pot. The frequency of the system clock should be approximately 100 kHz, but is not critical. The only requirement is appearance of the raster.

If the raster is evenly distributed over the screen, but is severely chopped up, check the digital inputs to the DACs. Use the scope to check the vertical and horizontal ramp waves individually. If the wave is not an even ramp, two or more of the DAC inputs may be reversed. Note that DAC input A1 is the most significant bit while input A6 is the least significant bit. Reversed inputs may also cause incomplete raster formations. Slight gaps or overlapping between some dots is caused by non-linearities in the manufacturing of the DACs.

If no raster at all appears, first check for a square wave output at pin 3 of the 555 timer IC. Then check for square wave outputs at each TTL 74193 counter. These square waves will be binary submultiples of the oscillator frequency. If the counter is operating, check all connections to the DACs and op amps.

Applying power will produce a random pattern of on and off dots. Adjust the amplitude of the Z axis signal for best contrast. Since most scopes will have an ac-coupled (or capacitor coupled) Z axis input, both amplitude and frequency of the signal will affect

Fig. 10. To construct a line segment in the direction shown by the arrow, alternately execute the commands shown.



performance. Charging the capacitor within the scope with too much voltage at a given frequency will cause the blank pulse to carry over into the next dot. This could cause more dots than desired to be blanked out or dimmed.

After a satisfactory raster is obtained, each instruction should be executed to verify its operation. First, clear the screen. The flowchart for a simple CLEAR program is shown in Fig. 8. The method outlined is to simply send out a "set Z off with increment" instruction 4096 times.

Fig. 9 shows the program listing for an 8008 system. This example used the B and C registers to keep track of the iteration count. The register contents are decremented once for each output ZFI instruction. The RETURN instruction may be substituted with a HALT if the CLEAR program is not to be used as a called subroutine. The CLEAR subroutine as listed in Fig. 9 begins by turning off the scan (which must be done before any programming, as stated), but does not turn the scan back on after the interface memory is cleared. The course of operation is left to the programmer once CLEAR has been called.

The chart in Fig. 10 may be used in testing the various control commands. The chart shows the commands to be used to construct a line segment in the direction shown by the arrow. Lines moving in a downward direction require that Y be reset with (n+1) for each dot programmed, "n" being the

Fig. 11. CHECKERBOARD Test Pattern Program flow chart.

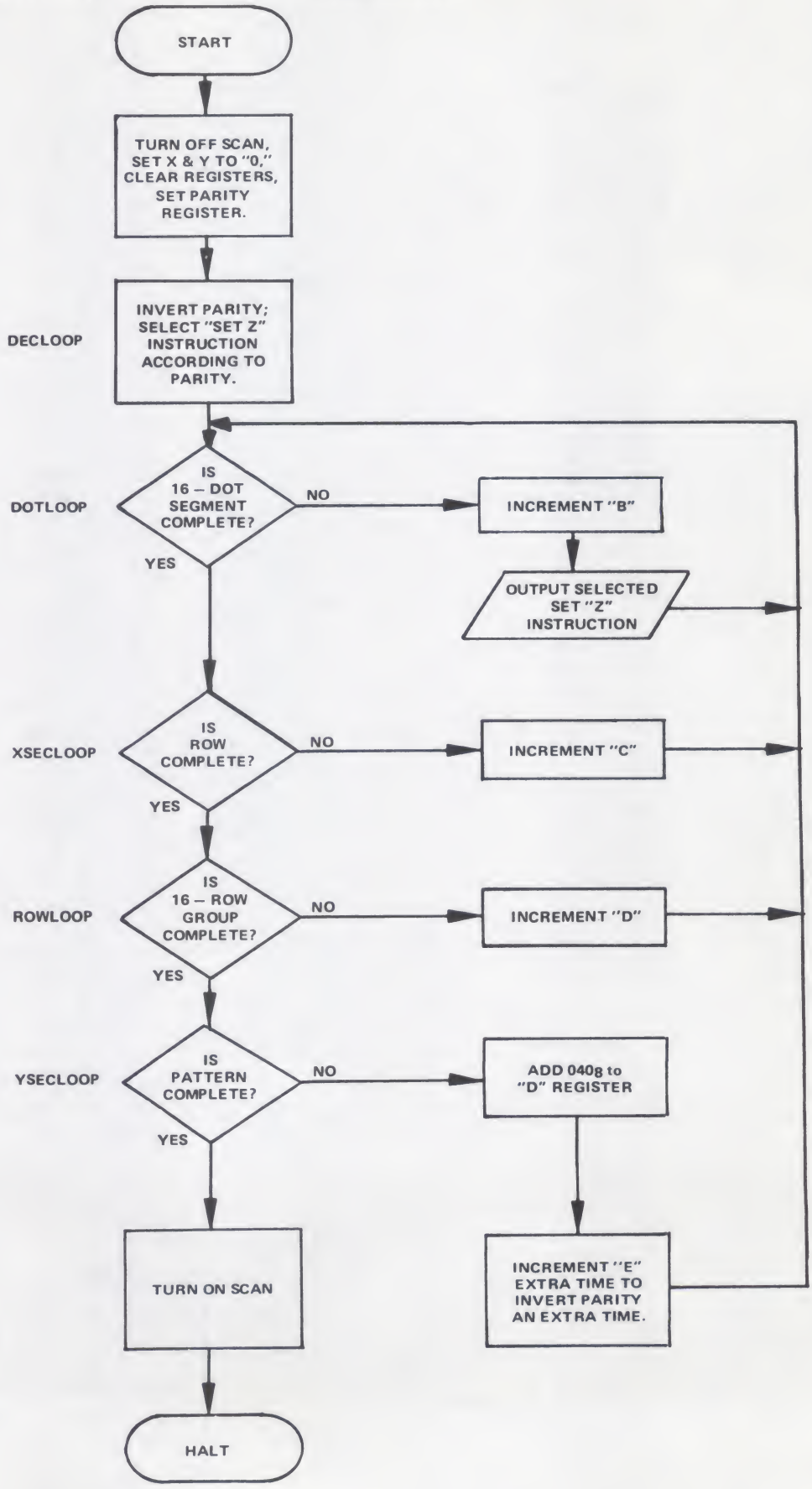


Fig. 12. Listing of 8008 code for the CHECKERBOARD program.

```
START       00/200 = 006    LAI
            00/201 = 201    (TSF)
            00/202 = 121    OUT 10
            00/203 = 006    LAI
            00/204 = 000    (STX)
            00/205 = 121    OUT 10
            00/206 = 006    LAI
            00/207 = 100    (STY)
            00/210 = 121    OUT 10
CLEAR       00/211 = 016    LBI
REGISTERS   00/212 = 000
            00/213 = 321    LCB
            00/214 = 331    LDB
            00/215 = 351    LHB
            00/216 = 361    LLB
            00/217 = 046    LEI
PARITY REG  00/220 = 000
DECLOOP     00/221 = 040    INE
            00/222 = 304    LAE
            00/223 = 044    NDI
            00/224 = 001
            00/225 = 150    JTZ
            00/226 = 246
            00/227 = 000
            00/230 = 066    LLI
            00/231 = 332
DOTLOOP     00/232 = 301    LAB
            00/233 = 024    SUI
            00/234 = 020
            00/235 = 150    JTZ
            00/236 = 253
            00/237 = 000
            00/240 = 010    INB
            00/241 = 307    LAM
            00/242 = 121    OUT 10
            00/243 = 104    JMP
            00/244 = 232
            00/245 = 000
DECLOOPJMP  00/246 = 066    LLI
            00/247 = 333
            00/250 = 104    JMP
            00/251 = 232
            00/252 = 000
XSECLOOP    00/253 = 016    LBI
            00/254 = 000
            00/255 = 302    LAC
            00/256 = 024    SUI
            00/257 = 003
            00/260 = 150    JTZ
            00/261 = 267
            00/262 = 000
            00/263 = 020    INC
            00/264 = 104    JMP
            00/265 = 221
            00/266 = 000
ROWLOOP     00/267 = 026    LCI
            00/270 = 000
            00/271 = 303    LAD
            00/272 = 044    NDI
            00/273 = 037
            00/274 = 024    SUI
            00/275 = 017
            00/276 = 150    JTZ
            00/277 = 305
            00/300 = 000
            00/301 = 030    IND
            00/302 = 104    JMP
            00/303 = 221
            00/304 = 000
YSECLOOP    00/305 = 303    LAD
            00/306 = 044    NDI
            00/307 = 340
            00/310 = 330    LDA
            00/311 = 024    SUI
            00/312 = 140
            00/313 = 150    JTZ
            00/314 = 326
            00/315 = 000
            00/316 = 303    LAD
            00/317 = 004    ADI
            00/320 = 040
            00/321 = 330    LDA
            00/322 = 040    INE
            00/323 = 104    JMP
            00/324 = 221
            00/325 = 000
END         00/326 = 006    LAI
            00/327 = 206    (TSN)
            00/330 = 121    OUT 10
            00/331 = 377    HLT
            00/332 = 204    (ZNI)
            00/333 = 205    (ZFI)
```

present Y coordinate. Use the STX and STY instructions to select a starting point. The dot whose coordinates are X=00, Y=00 will be in the upper left corner, the point where the scan begins its cycle.

The flow chart for a CHECKERBOARD TEST PATTERN program is shown in Fig. 11, with an 8008 listing in Fig. 12. The pattern produced will be 16 alternating light and dark squares. The 64 rows of dots are divided into 4 groups of 16 rows each. Each row is divided into 4 segments. The segments are alternately light and dark. The 4 groups also alternated to reverse the pattern between each group.

The set Z with increment instructions is used. The least significant bit of the E register is used in DECLOOP to alternate between "set Z on" and "set Z off." To obtain the complement of the entire pattern on the screen, place a 001 in location 00/220 instead of 000.

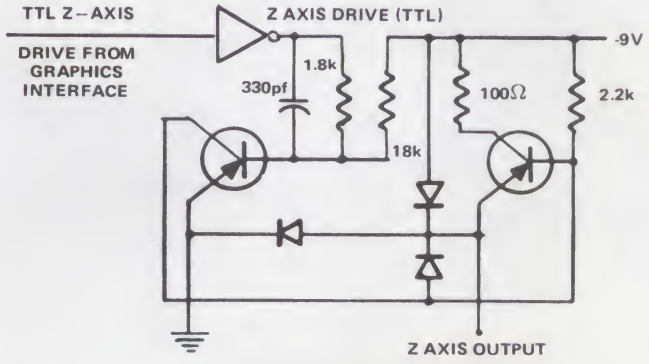


Fig. 13. A Z-axis drive circuit used to control blanking in the author's original version of the design. The transistors are 2N5139s and the diodes are silicon switching diodes such as the 1N914 part or its equivalent.

An Introduction to Addressing Methods

John Zarrella
90-9 Wakelee Rd
Waterbury CT 06705

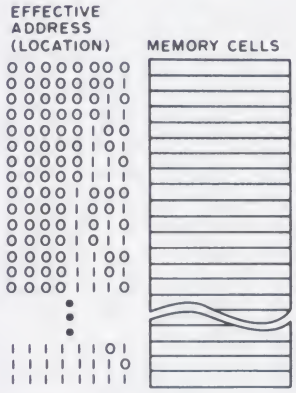


Figure 1: Memory Addresses. The effective address is the object of memory address calculations. It identifies a location in memory address space for the particular cell involved in some operation.

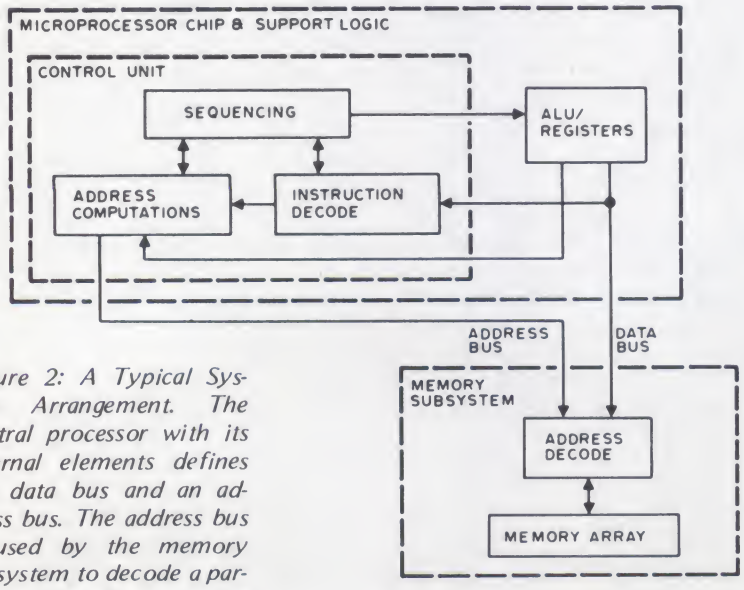


Figure 2: A Typical System Arrangement. The central processor with its internal elements defines the data bus and an address bus. The address bus is used by the memory subsystem to decode a particular location in the memory array which will be connected to the data bus.

An address is an identifier which describes the location of a particular piece of information within a computer's memory system. This information, when presented to the central processing unit for use in a computation, is usually referred to as an operand. In all microprocessor systems and in most other computer systems, an address is a binary number which is decoded to reference one computer word of information somewhere in the memory subsystem. Figure 1 illustrates how unique addresses are typically associated with memory cells.

It is interesting to note that this identifier need not be a number. There are some experimental computer systems in which memory locations are actually referenced by name or a combination of a name and a numeric index during execution. In these systems, there is hardware which translates the name directly into the location of an appropriate memory cell or group of cells.

In a similar manner, when writing programs in either assembly language or a higher level language such as FORTRAN, a programmer uses names to reference information. In this case, however, the names are generally mapped into numeric addresses by the language processing program and are not actually implemented in hardware as named references.

Instruction Cycles

Figure 2 illustrates typical interconnections among the control unit, arithmetic and logic unit (ALU), registers and memory subsystems of a general purpose processor. A brief review of the typical instruction fetch and execute cycle of such a CPU will be useful for the discussion which follows. The instruction fetch begins when the control

unit requests the next instruction by transmitting its address to the memory subsystem via the address bus. The current instruction address is usually maintained in a register called the program counter (or PC), and is updated to point to the next instruction when the current instruction is completed.

The information returned is treated as an instruction which specifies what function is to be performed by the processor. This instruction is analyzed in the instruction decode section of the processor. The execute portion of the instruction cycle then performs the functions which are specified by the decoded instruction.

Most instructions require data operands from the memory subsystem before execution can be completed. Thus a memory address must be created and sent to memory. This address is created using information contained in the decoded instruction in conjunction with information contained in various registers of the processor. The process of determining a data address is called address formation or address computation and is performed by the address computation section of the central processor. The result of address calculation is called an effective address.

A number of address formation capabilities are provided in the various designs of computers which are available. The typical contemporary microprocessor only provides a portion of the address calculation options to be described below. However, each mode, when available, can be utilized advantageously by the programmer. An understanding of addressing modes is useful when evaluating the instruction set of a computer. In order to clearly define the variety of addressing methods, an analogy will be used in the following discussion.

Immediate Addressing

In many ways memory addressing may be likened to the postal system. Imagine that you are writing a book on atomic physics and that Dr J Smith is to be a consultant. He currently lives in a small apartment complex called Apple Valley at 15 Grove St. There are five apartments at this location, each of which has its own street number—from 15 (manager) to 19. The mailboxes are arranged as shown in figure 3.

While researching the book, you attempt many of the necessary calculations yourself. These calculations involve multiplication, addition, transcendental functions and so on. Many times in these calculations you use fixed numeric factors, such as 18, which approximates $2\pi^2$. In doing this, you are treating 18 as a simple integer constant for the purposes of the approximation. In computer addressing terminology, this constant might be referenced with what is called immediate addressing by simply putting the number in a field of the computer instruction which follows the operation code. Here the effective address of the data is derived from the current program counter, and the actual instruction contains no addressing information.

Direct Addressing

An effective address is the goal of address calculation techniques.

Many times when performing calculations, you find that the results obtained are perplexing and need explanation. Therefore, you decide to ask your consultant for help. Since Dr Smith does not believe in telephones, you must send him a note, addressed to:

Dr J Smith
18 Grove St

In this case, the value of 18 is being used as an address. When delivering the letter, the mailman uses this address to determine where the letter belongs on Grove St. In its computer form, addressing with a single number such as 18 is called direct addressing or absolute addressing. In a computer, this number forms the address field which follows the instruction code in the program. This address field contains all the information needed by the memory subsystem in order to reference the required information, in the same manner that 18 Grove St contains all the information needed to locate Dr Smith on Grove St.

The problem of computing a result often reduces to the problem of organizing the reference of operands in memory through addressing techniques.

Note the contrast of this use of 18 as an address with its previous use as a constant. The number 18 which follows the instruction code is the same in either case; the intended use differs according to the instruction being executed. To know whether to use a number following the instruction code as an address or as a constant, its context must be known. In the typical computer, this is accomplished by building a special set of instructions called immediate instructions which use the number following the instruction code as a constant. A second set of instruction codes will be devoted to the absolute addressing mode, in which the field following the instruction code is an address. In general, for each possible addressing mode, a set of instructions exists which uses

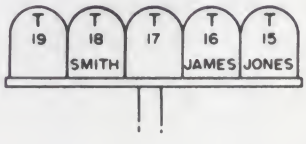


Figure 3: The concept of a memory address can be likened to that of a post office address.

that mode and interprets the information following the instruction code according to that mode.

Addressing With Registers

Suppose that you did not know Dr Smith's street address and sent the letter anyway. When the letter is received at the post office, the postmaster, knowing Dr Smith very well, would have to tell the postman: "I can't remember Dr Smith's address, but he lives in Apple Valley apartments at 15 Grove St and his mailbox is the fourth from the right in front of the complex." This specifies Dr Smith's address relative to a base address, 15 Grove St. In a computer, such a base address might typically be stored in an index (or general purpose) register as shown in figure 4. The displacement or address modifier in this case would be 3, which added to 15 gives the actual address of 18 Grove St. A computer with this single register indexed addressing method carries out the same form of calculation to produce the effective address: It adds the displacement or modifier field to the contents of the index register identified in the instruction.

An absolute or direct address specifies an operand location as a fixed number embedded in the instruction sequence.

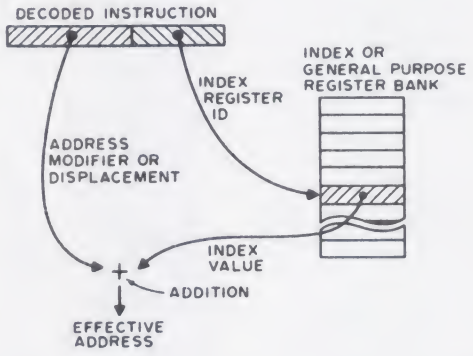


Figure 4: Indexed Addressing. One common mode of addressing is called indexed addressing, in which an index register specifies one numeric value which is added to an address modifier to produce the effective address. If the index register contains a base address value, then the modifier specifies a displacement or offset which is added to the base; if the index register contains an offset or displacement, then the modifier field is interpreted as a base address. In either case the result is an effective address.

Use of registers for address components enables one to employ base and index address concepts.

In the most general case, the index register may contain either an actual base address such as the first address of a table of values, or a displacement value. The corresponding contents of the modifier would be a displacement value or a base address, respectively. In some presently existing microprocessor designs, the index register is not large enough to contain a full base address. For instance, this occurs if the microprocessor uses a 16 bit address space and contains only an 8 bit index register. This case would require using the index register to contain a displacement with the base address becoming the instruction's modifier field.

Other options which sometimes occur include the choice of a second register as a component of effective address generation. In such cases, the instruction specifies one register which is intended as a base register, and a second register which is intended as an index register, as shown in figure 5. This form of double register addressing is sometimes combined with a modifier field as shown in figure 5. At this time, however, the microcomputers commonly available do not have such a powerful addressing mode.

One of the advantages of using a base register as well as an index register is that the base register can be used to locate a segment of memory, while the index register is used to access various places in that segment according to the program. Since all addressing is specified relative to the base register, relocating the program or data being referenced can be accomplished without modifying any code except the instructions which load the base register. The example of figure 6 shows the case of a computer which specifies a jump instruction effective address as the sum of a base register (register 0) and a displacement. Loading the same binary code at location 100 or 1125 is possible, provided the base register is initialized at the start of the program. The problem of relocation thus consists of redefining the constant which will be loaded into register 0 at the start of the program.

Program Counter Relative Addressing

Program counter relative addressing is very similar to indexed addressing except that the base address is implicitly specified using the program counter. In a typical machine which allows program counter relative addressing for data as well as program control purposes, the instruction contains a modifier relative to the current contents of the program counter as shown in figure 7. In some microcomputers, such as the 6800, program counter relative addressing is only allowed for branch instructions, and is specified relative to the next address following the end of the current instruction.

In terms of the postal analogy, this corresponds to the mailman coming upon a letter with no street address as he is working along his route. He therefore calls the postmaster and explains his dilemma. Since

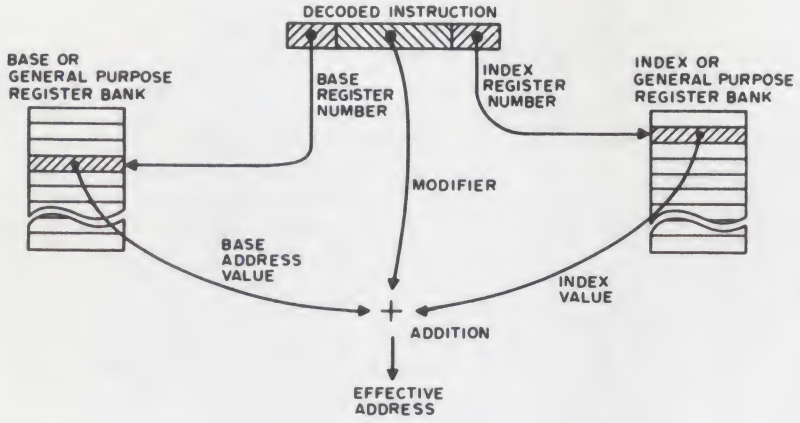


Figure 5: Combining Two Index Registers. A more general address calculation uses one register as a base register, a second register as an index register, and a modifier. The effective address is then the sum of the values found in the two registers and the value of the modifier. The order of calculation and detailed significance of the registers depends upon the processor design which uses this type of address calculation.

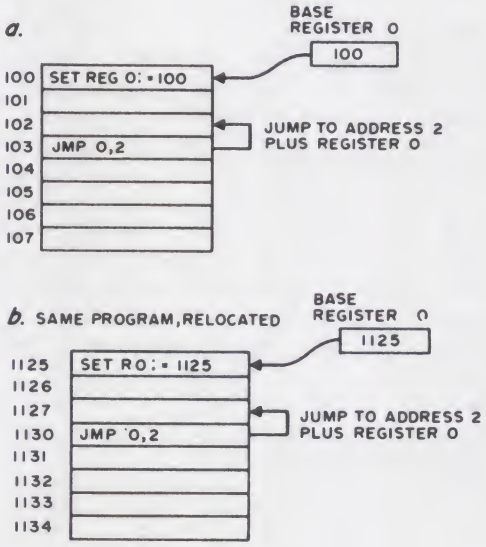


Figure 6: A base register scheme allows convenient relocation of code. In this example, the target address of a JMP (jump) instruction is specified as a base address register and a displacement. The value of the displacement is shown as two words from the start of a block of memory in which the program resides. With the base register loaded to the starting address, it does not matter where the block is located. At (a) it is located at octal address 100; at (b) the block is located at address 1125. With base addressing schemes, the first operation on entry to a program or block of code is to establish the value in the base register, as illustrated in these examples.

there is only one phone booth on the route, the postmaster gives him directions, such as: "Walk down the street directly in front of you and deliver the letter to the fourth mailbox in the apartment complex." Note that the base address is implicitly specified since the postmaster knows the location of the phone booth.

Indirect Addressing

To illustrate still another method of addressing, assume that Dr Smith recently had a post office box, #35. Since then he changed his mind and asked to have all his mail forwarded to his Grove St address. In order to remember the change when mail comes to the old address, the postmaster might mark Dr Smith's Grove St address on box 35. Then, when the mailman attempts to insert a letter for box 35 into that box, he sees the note that tells him to forward the letter to 18 Grove St. Thus, the box is not the final destination of the letter; in fact, it contains only an address to which the letter is to be forwarded. We call this method of locating the effective address (18 Grove St)

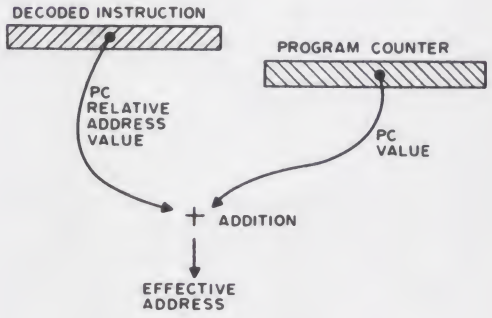


Figure 7: Program Counter Relative Addressing. Some computers provide a means to address memory in terms of an address displacement relative to the current program counter value. The instruction contains the displacement which the processor adds in the current program counter value for this type of effective address calculation.

indirect addressing. Figure 8 illustrates how the effective address is used to retrieve a second effective address in the computer form of indirect addressing. In the simplest form of indirect addressing, only one such level of indirection is involved.

We could easily extend this notion to multiple levels. In the postal analogy, imagine that Dr Smith moves out of 18 Grove St. The change of address order to the post office would result in a note to the postman on the 18 Grove St route, giving the new address of Dr Smith. Then, if a

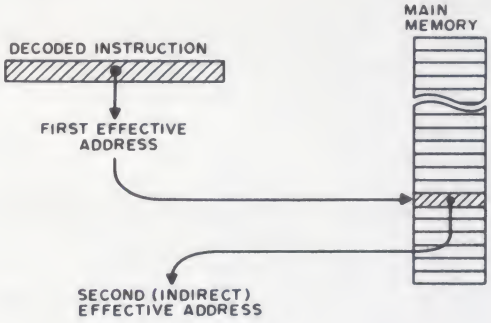


Figure 8: Indirect Addressing. In this form of addressing, the first effective address developed is used to address memory to find a pointer which will become the final effective address used for the instruction.

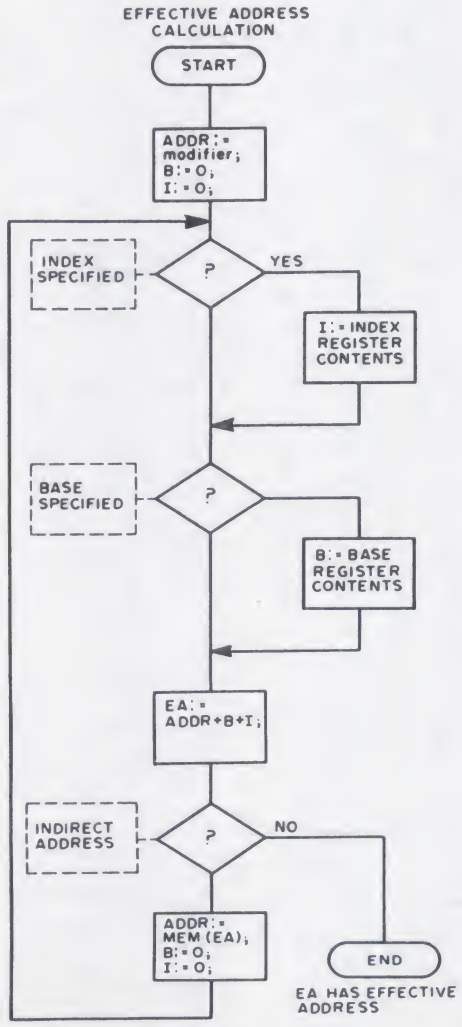


Figure 9: A General Address Computation Algorithm. This flow chart shows a typical address calculation algorithm of a modern general purpose computer. The typical microcomputer design circa early 1976 does not employ such a powerful addressing algorithm, but future improvements in chip designs should yield addressing techniques which approach the power of a good general purpose computer's addressing.

letter came to the original post office box 35 address, the postman would look up the 18 Grove St address. At the 18 Grove St address, the postman would in turn find the pointer to a new address for Dr Smith. The letter in this case would reach Dr Smith after two levels of indirection. This might happen a number of times if Dr Smith has a habit of frequently moving. In a microprocessor, the current chip designs offer only a very limited version of this mode, if indirect addressing is permitted at all. In minicomputers and large scale systems, indirect addressing is often allowed to continue to an indefinitely large number of levels.

General Address Evaluation Algorithm

Indirect addressing is often combined with the other addressing modes in computers which feature the most powerful effective address calculations. For instance, the indexed addressing mode might be used to develop the effective address for the first indirect address in a chain of indirect addresses. Once the chained indirect address lookup is begun, the processor might continue through multiple levels of indirection until a chain termination condition is detected. A general address evaluation algorithm which combines base register, index register and the possibility of indirection is shown in figure 9. Such an algorithm is typical of a good minicomputer, but is only partially implemented for most presently available microcomputer chip designs.

Summary

These methods of addressing are usually referred to as the addressing modes of the computer. To recap, the typically available modes are:

1. Immediate Addressing, in which the data being referenced forms a part of the actual instruction.
2. Direct or Absolute Addressing, in which the address of the operand is actually given as part of the instruction.
3. Indexed Addressing, in which one or more registers are specified, possibly including a modifier field. The effective address is a sum of the contents of the addressing registers and the modifier.
4. PC Relative Addressing, in which the program counter acts as a base address with an offset specified by the instruction.
5. Indirect Addressing, in which one of the other modes develops an effective address at which a pointer to data will be found.■

Interface an ASCII Keyboard to a 60 mA TTY Loop

Jay A Cotton
Bldg 844, Apt 2H
Gov Island NY 10004

I recently purchased a Sanders 720 electronic keyboard. This keyboard is identical to the Model 722-1 keyboard which was described in BYTE, September 1975, page 62, except for the key layout and the line feed code. My version of the keyboard had no line feed, but had a vertical tab key which produced an octal 013 code. In order to convert this to an octal 012 line feed code, some form of transformation logic was required. I also wanted to drive my Teletype's 60 mA current loop directly from the keyboard. By combining the special case code conversion, a UART for parallel to serial conversion, a clock and a current loop driver, I achieved the desired function of sending characters to my Teletype. Figure 1 shows the schematic of this conversion.

The Circuit

I chose to detect the octal code 013, then to use this special case to alter the data on the low order bit of the parallel code presented to the UART. By changing the low order bit of the octal 013 code from a logical one to a logical zero, the number is converted from 013 to 012. The 013 code is detected using inverters and the 7430 NAND gate shown in figure 1. The low order bit is selectively changed for this one code by using the exclusive OR function of one section of the 7486 integrated circuit. When the input at pin 2 is low (the normal case without the 013 code input), the exclusive OR normally passes line 0's value directly to the UART pin 26 input; when the input at pin 2 of the exclusive OR is high (as is the case when 013 is detected), the exclusive OR function inverts the value of line 0, thus transforming 013 at the keyboard into 012 at the UART.

The UART is programmed to generate the standard Teletype compatible format of a start bit, seven ASCII data bits, least significant first, then parity and stop bits. The key pressed signal from the keyboard unit is used as the data strobe to start transmission, and the transmitter end of character output of the UART is used to acknowledge completion of transmission. A 555 circuit is used to generate the clock. The clock should be adjusted to a 1760 Hz square wave; the circuit shown has about a 15% adjustment range for this purpose. The output of the UART is buffered by two inversions which protect the UART from excessive current drain. The buffered output in turn drives a relay through the quasi-Darlington coupled transistors. The relay used must be capable of switching the 60 mA current loop in times on the order of one millisecond. It must also be capable of sustained operation at high rates of change. If your junk box is not equipped with such a relay, other alternatives include use of an opto isolator and use of a high power interface circuit such as the 75451 driver chip.■

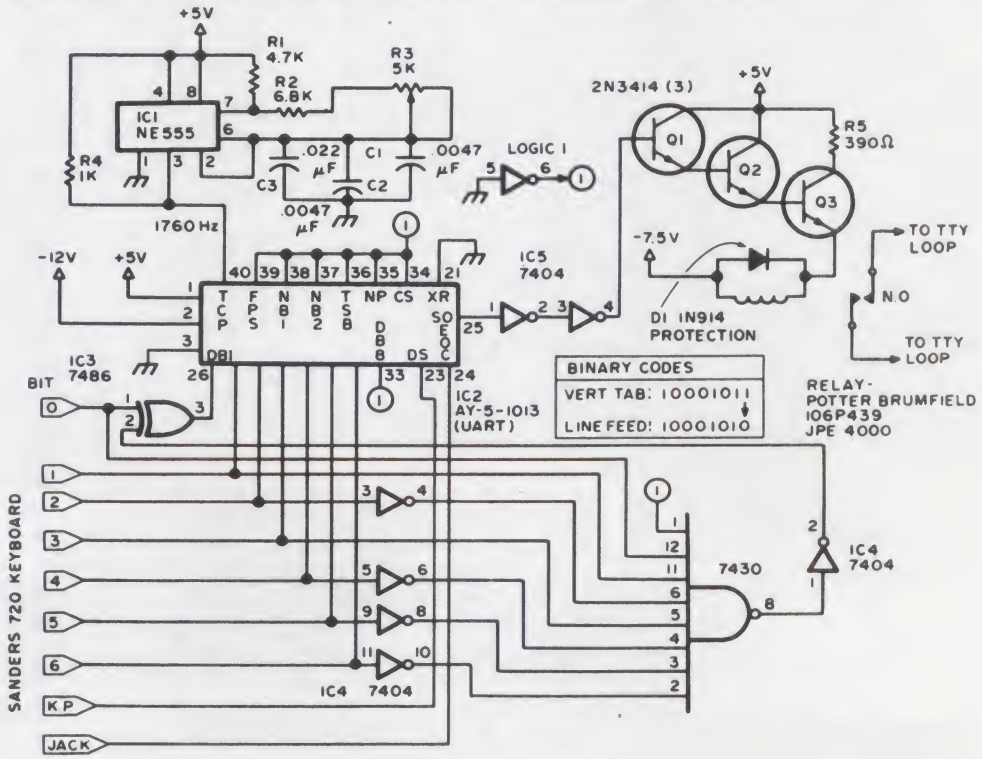


Figure 1: Using a UART and special case logic to convert and serialize the output of a keyboard for a 60 mA current loop.

Interfacing the 60 mA Current Loop

Walter S King
451-145th Place NE
Bellevue WA 98007

Generally the older Teletype units such as model 15s, 19s and 28s require a 60 mA loop to operate the printer. These older machines are not as attractive looking as the newer model 32s and 33s, but for the Altair computer hobbyist, looks are probably second to costs. The 60 mA interface circuits shown below are simple, straightforward, and do an effective job.

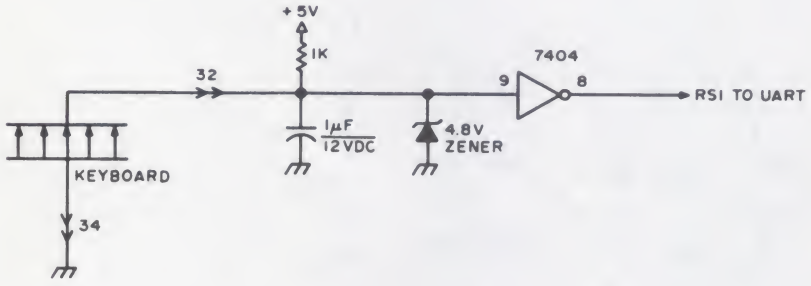


Figure 1: Input Circuit. The Teletype generates Baudot codes mechanically by activating switch contacts according to the code being generated. To condition the inputs for the UART, this circuit will debounce the signal and convert it to a TTL level.

Circuit Notes

The loop keying transistor, 2N5655, is a 250 V power tab purchased at two for $1 at a surplus house. In the mark state, this transistor is fully saturated. The collector dissipation is 0.7 V x 0.060 A or 0.042 W. In the space state, with no collector current, the dissipation is zero. Heat sinking is not required. The 0.1 μF and the 470 ohm resistor protects the keying transistor from voltage spikes generated by the inductance of the printer magnet. The 10 K ohm resistor in base circuit limits the current supplied by the UART TSO output gate to a safe value when in the mark state. The variable resistor in the loop should be adjusted with a milliampere in the circuit. Set the loop current to 60 mA. A pull up resistor, 1 k ohm, is connected to the keyboard and +5 V to generate a TTL level keying signal. The 1 μF capacitor in parallel with the keyboard is used to smooth out any contact bounce. The 4.8 V zener diode clamps the space signal to 4.8 V (logic 1). Also, hopefully, it will act as a crowbar

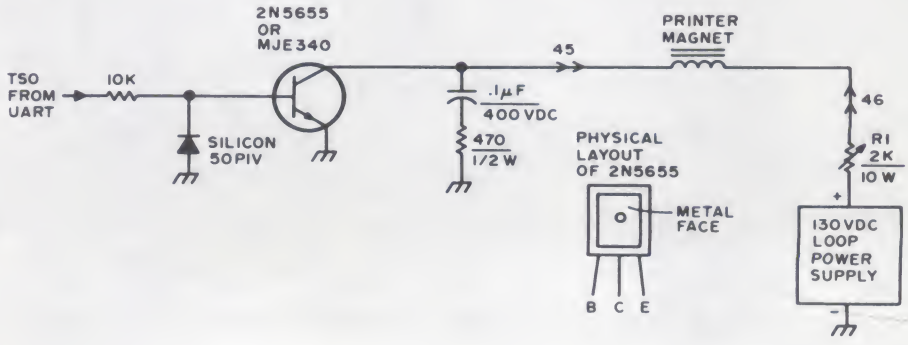


Figure 2: Printer Drive Circuit. The 60 mA current loop is a circuit which normally passes 60 mA through all the printer magnets and keyboard contacts of Teletypes which are "in the loop." This circuit drives the printer mechanism only by using a TTL level signal from the UART to control a transistor switch.

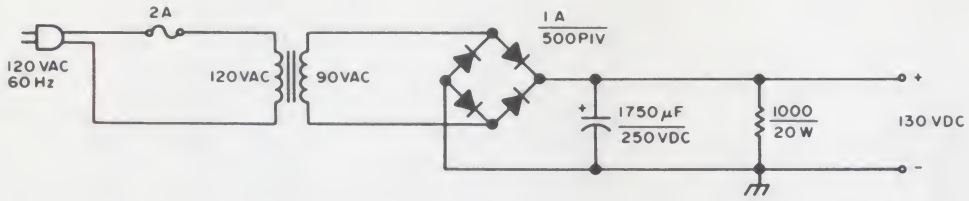


Figure 3: To complete the adaptation of a 60 mA TTY to your UART, this simple power supply will provide the necessary voltages. The transformer should have a secondary with at least 90 volts AC input to the bridge rectifier. The actual value could be 90 to 120 volts or so depending upon what you can find in the junk box of your home laboratory. The capacitor value of 1750 μF is also not critical. The voltage rating should be higher than the output peak of the bridge rectifier and the value should be greater than 500 μF.

circuit (short to ground) if high voltage were to appear in the keying circuit by accident.

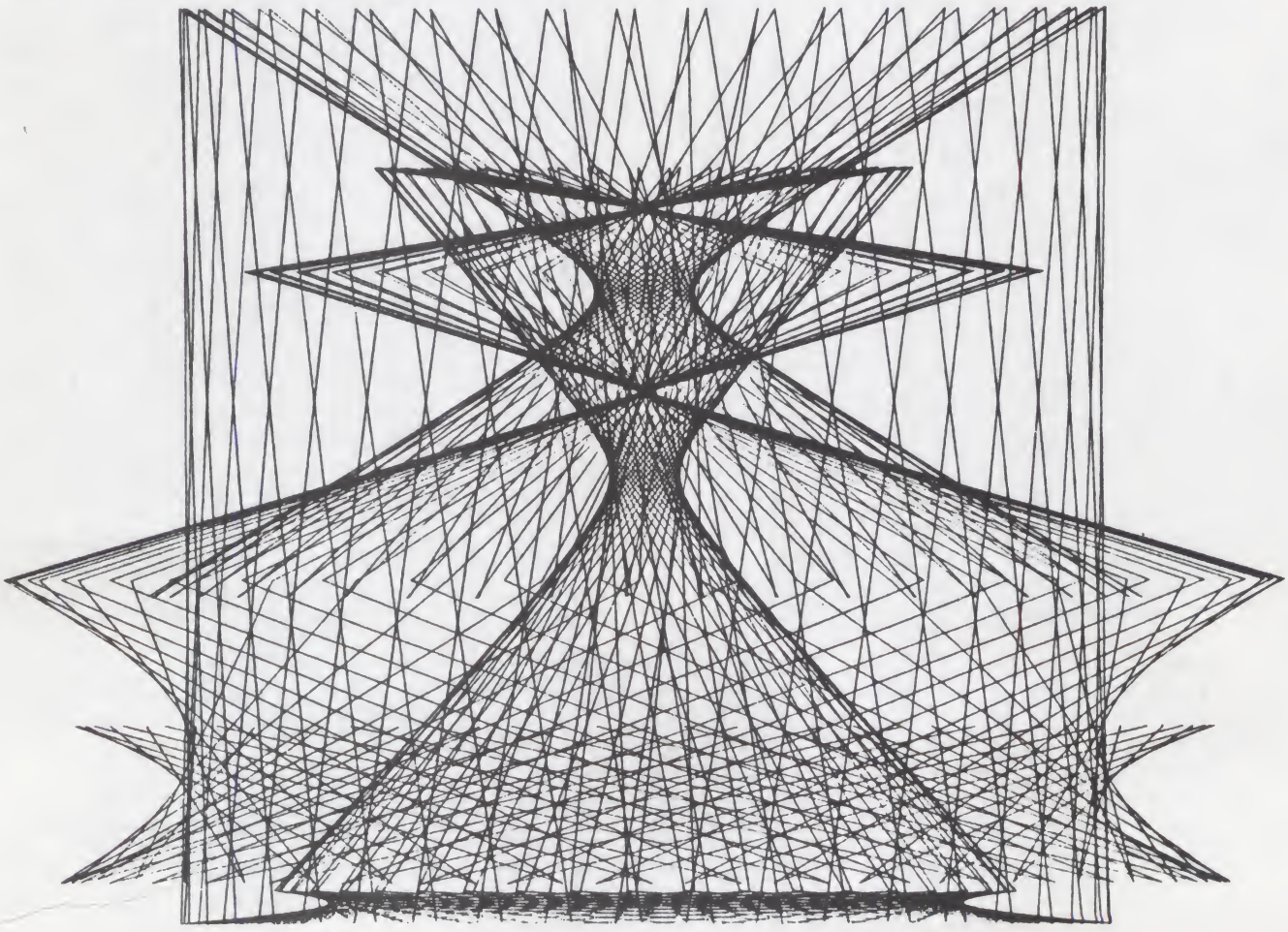
External Connections

It is a good idea to mount the keying transistor on a perforated board separated from the serial IO board. An inadvertent short circuit to the high voltage loop could wipe out the serial IO board integrated circuits. The keyboard contacts on a Model 15 or 19 are usually terminals number 32 and 34. The printer magnet terminals are numbers 46 and 45. If there is a line relay in the machine, remove it and discard it.

UART Connections

If you are using a Model 15 or Model 19, the Baud rate is 45. The UART clock preset count for 45 Baud is 2454 in octal. The Model 28 with 100 word per minute gears runs at 74.2 Baud. The preset count for 74.2 Baud is 4553 in octal. Since these older Teletypewriter machines use only five data bits, the UART jumpers NDB1 and NDB2 must be wired to GND. The NSB jumper is connected to logic 1 which selects 1-1/2 stop bits when NDB1 and NDB2 are grounded.

If read errors begin to occur on the keyboard, it is probably due to an oil film on the keyboard switch contancts. Use a little carbon tetrachloride solvent on them or carefully pull a piece of paper between the contacts to clean them.■



The COMPLEAT Tape Cassette Interface

Jack Hemenway
151 Tremont St, 8P
Boston MA 02111

The software of a tape cassette interface provides open, data transfer and close operations for both input and output.

Mass storage is one of the most important functions in the small computer system design. Mass storage can typically be used as the medium of a text editor, as the input and output of a full fledged language translator program, and as a means of saving working and debugged software you've created. One of the least expensive ways to accomplish mass storage is the audio cassette storage method.

What is involved in the use of audio cassettes for mass storage? Here's an answer which works quite well in my Motorola 6800 microcomputer system. The COMPLEAT Tape Cassette Interface consists of tape input and output software, the Lancaster speed independent audio interface (see BYTE, September 1975), a Motorola asynchronous communications interface adapter (ACIA), a transmit clock, and the circuitry needed to start and stop the tape recorder's motor under program control. The hardware of the interface is shown in block diagram form in figure 1. The software consists of an open, data transfer and close subroutine for each direction of transfer, input and output. The hardware and software described in this article can be used as the stepping stone to a more complete cassette tape information management system, or it can be used alone whenever a program requires cassette input or output functions.

What Is an ACIA?

The Motorola MC6850 asynchronous communications interface adapter is a specialized version of the familiar universal asynchronous receiver transmitter (UART). The ACIA is designed specifically to interface the Motorola 6800 central processor; however, its use is by no means limited to the 6800. An ACIA can be used conveniently in any computer system with data paths 8 bits or more in width.

The ACIA differs from the conventional UART in the way it is controlled. All control, status and data transfers are made over a single 8 bit bi-directional bus. The integrated circuit contains a control register which may be set by the microprocessor; it is a location in memory address space. The ACIA contains a status register which may be tested by looking at the same location. The ACIA also contains transmitter and receiver data registers which are treated as a memory location via the bus structure and selection logic. In contrast to the UART with its separate input and output data buses, hardwired option selections and 40 pin package, the ACIA design fits into a 24 pin package with several pins left over for use as address selection and modem control functions.

The ACIA options are normally selected by storing a bit string into the control

register when the computer system is first initialized (at power on time) or later when the reset operation is performed manually. However, since the ACIA has the control register, these options can be changed at any time by a program which runs the interface. This capability is used to advantage in the COMPLEAT Tape Cassette Interface: The tape cassette motion is controlled through the RTS line of the ACIA (pin 5, IC1) by setting an appropriate two bit code into the transmitter control bits of the control register (bits 5 and 6); whenever the tape motion is changed (on to off, or off to on), these bits of the control register are redefined.

The ACIA is interfaced to the system data bus either directly, or by means of an appropriate 8 bit bus buffer. The interface is controlled by means of the read write line (RW) and address selection logic. In the hardware of this article, a full address decode is avoided by wiring the chip select lines to appropriate system address bits and using the low order address bit as the register select line (RS, IC1 pin 11). The ACIA has four registers, but only two memory address space locations are required. The apparent inconsistency is resolved by the read write line of the system interface. Two of the internal registers are read only registers (receiver data and status registers), and two of the internal registers are write only registers (transmitter data and control registers). Table 1 shows the system addresses and register access used by the interface of figure 2. (Note that any pair of neighboring locations in memory address space can be used conveniently with appropriate decoding.)

The enable line (E, pin 14 of IC1) is used to synchronize the ACIA status and control changes to the processor, and to condition the ACIA's internal interrupt circuitry. The interrupt request line (IRQ, pin 7 of IC1) is used in systems which employ interrupts to coordinate IO operations. If used, it signals the microprocessor whenever the ACIA is requesting an interrupt. In the simple interface presented here, interrupts are ignored and the software is coordinated using the status register flags of the ACIA.

For a full description of the ACIA control and status registers, the specifications of the Motorola MC6850 ACIA integrated circuit should be consulted. See also pages 3-22 to 3-25 of the Motorola *M6800 Microprocessor Applications Manual*. The software shown in this article makes use of the status register bits for timing and error detection, and sets up the control register for a standard 8 bit asynchronous data format

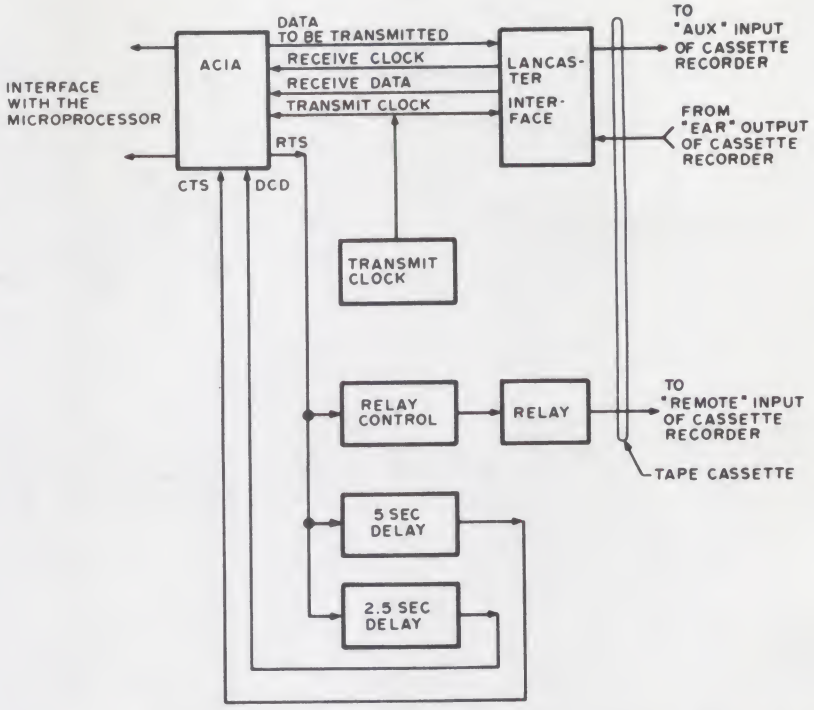


Figure 1: Block diagram of the COMPLEAT Tape Cassette Interface. This illustrates the major elements of the interface; see Don Lancaster's BIT BOFFER article in this issue for details of the hardware of the cassette modem.

with one start bit, one stop bit, odd parity and a division ratio of 16 for the clocks used with the ACIA.

On the peripheral side of the ACIA there are three lines which are used to control and test the tape recorder interface. An output line called request to send (RTS, pin 5 of IC1) is used for tape motion control. An input line called clear to send (CTS, pin 24 of IC1) is used for a tape output delay timer, and an input line called data carrier detect (DCD, pin 23 of IC1) is used for a tape input delay timer. Serial data generated by the ACIA is sent to the Lancaster tape interface modem over the transmit data line (TXDATA, pin 6 of IC1), and serial data received from the Lancaster tape interface

An ACIA is Motorola's version of a UART.

Table 1: ACIA addresses. This table shows how the four ACIA registers are referenced, using two memory locations. The secret is that two of the registers are input only, and two of the registers are output only. Thus at each address, the register referenced in the ACIA depends upon whether the CPU is reading data from that address or writing data to that address.

Address	Operation	Symbol	ACIA Register	Typical Code	
8010	read	ACIACTRL	status	LDAA	ACIACTRL
8010	write	ACIACTRL	control	STAA	ACIACTRL
8011	read	ACIADATA	receiver data	LDAA	ACIADATA
8011	write	ACIADATA	transmitter data	STAA	ACIADATA

modem is presented to the receive data line (RXDATA, pin 2 of IC1). During transmission, the data rate is set by the transmitter clock, generated by IC3 and IC4, and during input operations the receiver clock (RXCLK, pin 3 of IC1) is recovered from the tape data by the Lancaster interface, locking the ACIA to the actual tape speed.

Hardware Software Interfaces

In this interface, software is synchronized to hardware, using the technique of testing status bits in wait loops.

The ACIA is controlled by the microprocessor software which views it as the two adjacent memory locations shown in table 1. The interfaces between hardware and software can be controlled by one of two different methods. The interrupt method of IO synchronization relies upon the ACIA to generate an interrupt in the processor through the IRQ line of the system. Because the central processor is interrupted (and its state is saved) only when IO service is required, the processor can be busy with some other task while waiting for the slow IO device to complete its operation.

In contrast, the programmed transfer method employs a wait loop in a program to monitor ACIA status register bits which indicate the progress of the data transfer operations. When the status bits indicate that the ACIA is ready for a transfer to or from the data location, the interface program can then proceed to carry out the transfer. The programmed transfer method is employed in the software of the COMPLEAT Tape Cassette Interface illustrated here, primarily because of its simplicity.

A flip flop toggled by a clock produces an output clock which is a perfect square wave at one half the input frequency.

Reading is accomplished by testing the status register repeatedly until the receiver data ready flag (bit 0 of the status register) is high, indicating the presence of data. When the data is available, the program loads the ACIA data location into an accumulator, an operation which resets the status flag; for input the status register is also tested for the three kinds of error conditions, and a condition code is returned from the input routine in the other accumulator. Similarly, writing is accomplished by testing the status register repeatedly until the transmitter buffer empty flag (bit 1 of the status register) is high, indicating an empty buffer which can receive the output character. The output program then stores a character into the ACIA data location causing the ACIA to begin an output operation and resetting the flag status bit.

Hardware for COMPLEATness

Software timing loops and hardware oneshots can accomplish the same goal: delaying execution.

The circuit of the interface is shown in figure 2, including all details except the Lancaster tape cassette modem circuit. Note that with appropriate clock frequencies, any modem circuit could be used which accepts the asynchronous data format and clock information. (See Don Lancaster's article "Build the BIT BOFFER" in this issue, and "BYTE's Audio Cassette Standards Symposium," page 72 in the February 1976 BYTE.) The modem interface consists of four signals:

TXDATA: This signal is the output data generated by the ACIA at a baud rate equal to the TXCLK frequency divided by the ACIA divide ratio. In this article, division by 16 is used, as set in software by the control codes to the ACIA. This line is shown wired directly from the ACIA, so it can drive the equivalent of one TTL load.

TXCLK: This signal is the output data clock, which is fed to the ACIA and to the tape interface. The Lancaster interface modem uses this signal to synchronously generate the two data frequencies which are recorded on the tape according to TXDATA.

RXCLK: This signal is the input data clock, which is recovered from data by the tape interface. Since this signal is derived from the tape input data, it is locked to any variations in tape speed. Thus the ACIA's input circuitry will not make errors due to differences between the transmitter clock frequency and the variations in tape speed which are called "wow and flutter" in audio tape recorder specifications.

RXDATA: This signal is the input serial data recovered from the Lancaster audio interface modem. Since RXDATA is locked to RXCLK, speed variations of data relative to clock cannot occur.

Transmit Clock

The transmit clock is provided by a 555 oscillator (IC3) followed by a flip flop (7473, IC4) which divides the oscillator frequency by 2. The 555 is wired with components selected for a frequency of 9600 Hz. When the interface is constructed, the potentiometer R1 should be adjusted so that the frequency is 9600 Hz, using a frequency counter or an oscilloscope to make the measurement. For those using oscilloscopes, 9600 Hz is a period of 104.2 μs, or a 5.21 cm trace on a scope set for a 20 μs/cm horizontal time base.

The division by 2 which follows the oscillator is provided by a JK flip flop set up to toggle. This means that both the J and the K inputs are connected to logical one (IC4

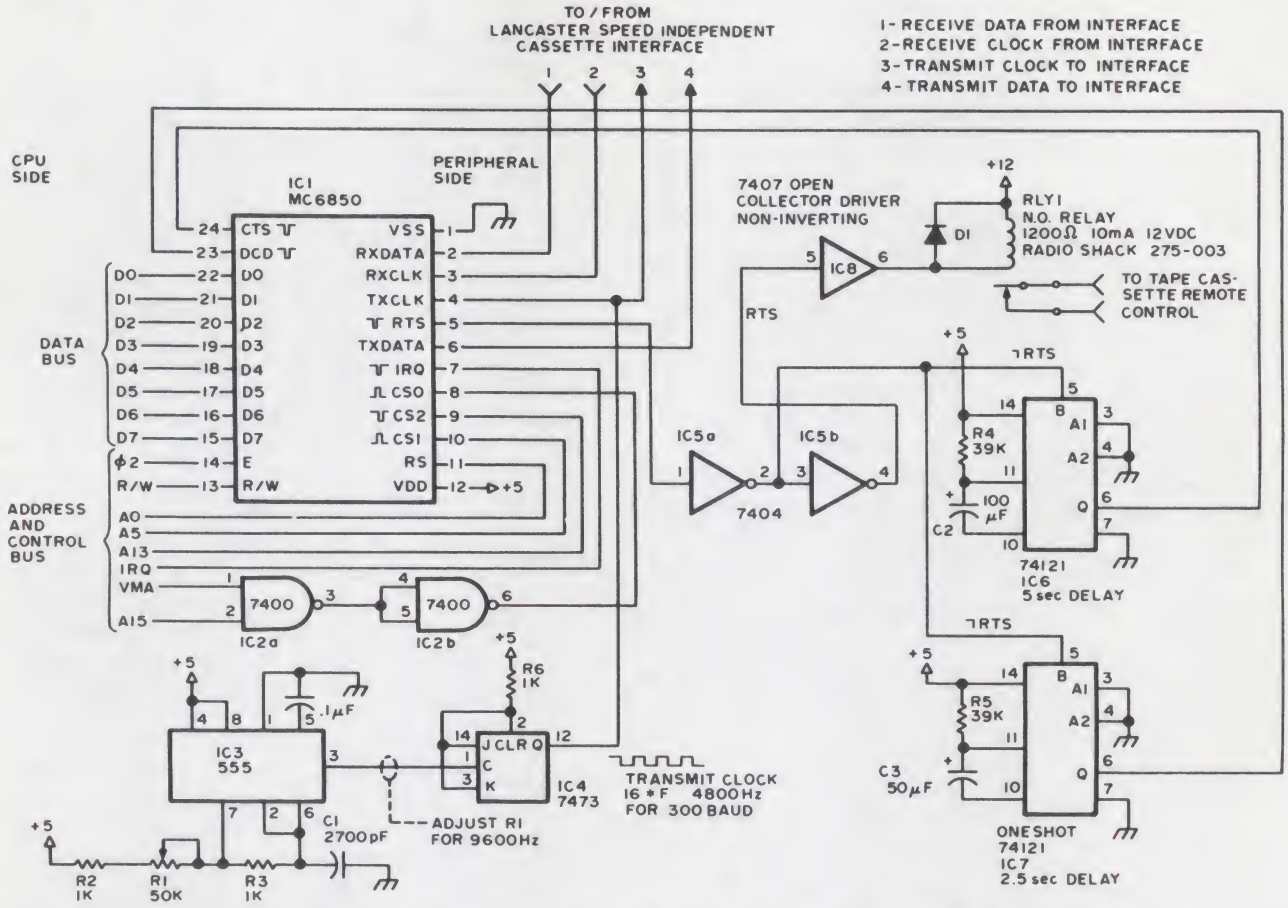


Figure 2: Motorola 6800 ACIA and control circuitry for the COMPLEAT Tape Cassette Interface.

pins 14 and 3). The purpose of the division stage is to produce a perfect square wave clock signal, which is a requirement for the Lancaster cassette interface.

Tape Motion Control

The request to send line (RTS) is used to control the tape recorder's motor, as mentioned earlier. Whenever the ACIA is set up with a control register code for a low value of RTS, the signal presented to the 7407 (IC8) section used as a relay driver is low (after two inversions in IC5). This signal is buffered by the driver, producing a low state at its output (IC8, pin 6) which places 12 volts across the relay coil, closing the contacts and turning on the motor. When RTS is set high, using a different ACIA control code, the input to IC8 is high, so the relay coil has zero volts across it and the relay contacts are open. Note that diode D1 is placed across the relay coil to guard against inductive back EMF which can blow out integrated circuit drivers such as IC8.

Tape Motor Start Delays

Two oneshots are provided in this design in order to give hardware delays of 2.5 and 5.0 seconds following tape motor turn on. The long delay is used prior to output operations so that a long leader at the mark frequency will be recorded. The short delay is used during read operations so that reading will start 2.5 seconds prior to the first actual data byte. Since the asynchronous data format is used, the solid mark tone for about 2.5 seconds will not cause any data to be input; it provides tolerance for manual tape positioning to selected blocks using a tape position counter which is built into many cassette recorders.

The system as designed and illustrated in this article uses hardware to generate the time delays of 2.5 and 5.0 seconds after motor start. It should be noted, however, that the timers IC6 and IC7 could be omitted and replaced by software. To make such a change, the input and output initialization routines would have to be altered to use software timing loops to create the required delay. Examples of such timing loops have appeared in previous issues of BYTE (see "Add A Kluge Harp to Your Computer," October 1975, page 14, and "Can Your Computer Tell Time?", December 1975, page 82). This is an example of a

Always program with commentary — if you want to communicate what you mean to yourself (five years from now) or to your neighbor.

hardware software tradeoff: If you find it easier to program timing loops and you don't mind having an idle computer wasting time in such loops, then omit the hardware timers and use software; if you plan to use interrupts, with the central processor turning to other tasks while waiting for IO operations, then the hardware timers would be preferable.

Software COMPLEATness

No tape cassette interface is complete without software to run it. The software of the COMPLEAT Tape Cassette Interface gives facilities to perform several operations. In order to understand the use of the software provided, the three operations of opening, transferring data and closing a file should be defined:

The larger a block of data on a cassette, the less tape is wasted in "inter record gaps" as the motor starts and stops. At 300 baud, a 4096 byte block can be recorded in 150 seconds, so the 5 to 10 seconds of inter record gap waste less than 7% of the available tape on a cassette.

Opening a file. The first operation in a data transfer to a device such as the COMPLEAT Tape Cassette Interface is opening the file. This operation is minimal for the simple system discussed here: The tape motion is started and a wait loop is entered until the motor start delay is complete. For output, the subroutine T1OTZ performs this operation. For input, the subroutine T1INZ performs this operation. In a more sophisticated software system, opening a file can have a much more general meaning and effect. For the COMPLEAT Tape Cassette Interface, the tape recorder must be set up manually for playback (read operation) or record (write operation) and the tape should be positioned at the beginning or at a location specified by the location counter of the recorder prior to opening the file.

Data Transfer. Once the file is opened, the motor is on; and data transfer can occur for as long as software desires. A data transfer operation is the input or output of one character from the computer. The software of the interface provides an input routine called T1GET which reads the next character into accumulator A with an error condition code in accumulator B. The software of the interface provides an output routine called T1PUT which takes a character from accumulator A and stores it in the ACIA for conversion and output to the cassette modem. Note that the program which calls the data transfer routines must keep track of how many bytes to transfer. One convenient way to do this is to arbitrarily decide to always output a fixed number of bytes, such as 256. Another way to keep track of block size is to decide to send the block length as an 8 (or 16) bit number which is always recorded as the first byte (or first two bytes) of the block. These are by no means the only software data formats possible.

Closing a file. When the software which requires an IO interface completes sending all the data required for one block on the tape, the last step of the IO operation is to close the file. In this simple cassette interface, the file closing operation consists of turning off the motor. (In the case of output, the last character transmission must be completed so the close routine also includes a wait loop.) In the more general case of an information management system, closing a file might include other operations such as recording check sums for error detection. The output file closing routine is T1OSTP, and the input file closing routine is T1ISTP.

The Listings

Listing 1 starts a detailed presentation of the software in the absolute machine language and the symbolic assembly language of the Motorola 6800 microprocessor. (While the interface is shown oriented toward my 6800 system, the listings are given with ample commentary to document function and facilitate conversion to other microprocessors.) Listing 1 contains three statements which set up information used by the assembler. Lines 1 and 2 use the EQU pseudo operation to set the addresses of the labels ACIACTRL and ACIADATA which are used to reference the two memory address space locations associated with the COMPLEAT Tape Cassette Interface hardware. Line 3 is an ORG statement which is used to set the location counter of the assembler to hexadecimal 1000 which will become the starting point of the first routine. While the listings of the COMPLEAT Tape Interface routines in this article show a hexadecimal starting address of 1000, the subroutines can in fact be relocated to any starting address without changing the absolute machine code. If you do choose to relocate the code, however, you should figure out the relocated addresses of the subroutine entry points so that they can be referenced by software which uses these routines.

Listing 2 describes the output initialization routine T1OTZ which is used to prepare for an output operation. When T1OTZ is given control, an assumption is made that the cassette recorder has been manually prepared for recording but with motor

power removed. The ACIA is first reset using the control register code of hexadecimal 5F (lines 6 and 7). Then the ACIA control code for normal operation, hexadecimal 1D, is set up by execution of lines 8 and 9. This turns on the tape recorder motor and triggers the oneshots of IC6 and IC7 in the interface. The output of oneshot IC6 is monitored as bit 4 of the control register (the CTS line into pin 24 of IC1). The initialization routine falls into a loop at lines 10 to 12 until this time delay signal ends and bit 4 of the control register becomes zero. T1OTZ has no parameters and uses the stack to preserve the contents of accumulator B, so that upon return none of the internal registers of the processor have been altered.

Listing 3 describes the output data transfer routine T1PUT. The purpose of this routine is to put the contents of accumulator A into the output data stream. T1PUT first tests the transmitter buffer empty flag of the status register (bit 1) in a wait loop at lines 17 to 19. Then it simply stores the output character of accumulator A into the ACIA data location, which automatically initializes an output operation for that character. T1PUT has one parameter, a character code passed in accumulator A. It preserves the content of accumulator B in the stack.

Listing 4 gives the code for the output close routine, T1OSTP. This routine uses a wait loop at lines 24 to 26 in order to ensure completion of the last ACIA output conversion. Following completion of the last character, T1OSTP loads the ACIA control register with the hexadecimal control code 5D in order to turn off the tape recorder motor. It then returns to the caller. T1OSTP has no parameters. As shown, T1OSTP uses accumulator B as a temporary data storage area but does not preserve its value in the stack; addition of PSHB and PULB operations (after line 23 and before line 29 respectively) could be done to preserve these registers if required.

Listing 5 gives the code of the input open routine, T1ITZ. This routine is identical to T1OTZ in all areas except one: It tests the DCD status line (bit 2 of the control register) instead of the CTS status line. Thus the input initialization routine waits for the 2.5 second delay produced by oneshot IC7.

Listing 6 shows the input data transfer routine, T1GET. This routine is called once for each character of input expected by the program which uses the tape cassette interface. Its first action is to enter a loop (lines 41 to 44) waiting for the receiver data available flag in the status register to become high. When a character is ready and indi-

Listing 1: Global symbol equates and origin of the COMPLEAT Tape Cassette Interface software. This listing sets up the symbolic addresses ACIADATA and ACIACTRL, and sets the location counter to start at hexadecimal 1000. Note that a common statement number sequence is used for all the tape cassette interface software in listings 1 through 8, and that symbols with up to 8 characters (Motorola allows 6) are used in these listings.

```
1  0000  80  10      ACIACTRL  EQU   $8010     set up address of ACIA control
2  0000  80  11      ACIADATA  EQU   $8011        and then ACIA data;
3  1000  10  00                ORG   $1000     start program at 1000 hexadecimal;
```

Listing 2: Output initialization routine T1OTZ. This subroutine is called after *the tape recorder has been readied manually for a write operation. T1OTZ resets the ACIA and turns on the tape recorder motor, then waits for the end of the output initialization delay. The delay is ended when CTS is found to be zero five seconds after the motor turned on.*

```
4  1000  10  00      T1OTZ     EQU   * this routine initializes and starts output;
5  1000  37                    PSHB                push B into stack to save it;
6  1001  C6  5F                LDAB  #$5F          ACIACTRL := control code for
7  1003  F7  80  10            STAB  ACIACTRL         master reset, RTS high;
8  1006  C6  1D                LDAB  #$1D          ACIACTRL := control code for
9  1008  F7  80  10            STAB  ACIACTRL         RTS low, normal operation;
10 100B  F6  80  10  T1OTZW    LDAB  ACIACTRL      B := ACIA status register;
11 100E  C5  08                BITB  #$08          test status of CTS (bit 4);
12 1010  26  F9                BNE   T1OTZW        if not ready then keep looping;
13 1012  33                    PULB                else pull B from stack to
14 1013  39                    RTS                    restore it, then return;
```

Listing 3: Character PUT routine T1PUT. This subroutine is called whenever it is desired to write a character on tape. After waiting for a go ahead from the transmitter buffer empty status bit, the routine transfers the contents of accumulator A to the ACIA transmitter buffer register.

```
15 1014  10  14      T1PUT     EQU   * this routine writes one character of output;
16 1014  37                    PSHB                push B into stack to save it;
17 1015  F6  80  10  T1PUTW    LDAB  ACIACTRL      B := ACIA status register;
18 1018  C5  02                BITB  #$02          test status of transmitter (bit 1);
19 101A  27  F9                BEQ   T1PUTW        if not ready then keep looping;
20 101C  B7  80  11            STAA  ACIADATA      else transmit a byte from A;
21 101F  33                    PULB                pull B from stack to restore it;
22 1020  39                    RTS                 return to the caller;
```

Listing 4: Output close routine T1OSTP. This subroutine is called following output of a series of characters (a "block"). T1OSTP waits for the completion of the last output operation, then shuts down the tape recorder motor and returns.

```
23 1021  10  21      T1OSTP    EQU   * this routine stops tape after writing a block;
24 1021  F6  80  10            LDAB  ACIACTRL      B := ACIA status register;
25 1024  C5  02                BITB  #$02          is transmitter data register empty?
26 1026  27  F9                BEQ   T1OSTP        if not keep waiting for empty;
27 1028  C6  5D                LDAB  #$5D          else ACIACTRL := control code for
28 102A  F7  80  10            STAB  ACIACTRL         RTS high, motor off;
29 102D  39                    RTS                 return to caller;
```

Listing 5: Input initialization routine T1INZ. This subroutine is called after *the tape recorder has been readied manually for a read operation. T1INZ resets the ACIA and turns on the tape recorder motor, then waits for the end of the input initialization delay. The delay is ended when DCD is found to be zero 2.5 seconds after the motor is turned on.*

```
30 102E  10  2E      T1INZ     EQU   * this routine initializes and starts input;
31 102E  36                    PSHA                push A into stack to save it;
32 102F  86  5F                LDAA  #$5F          ACIACTRL := control code for
33 1031  B7  80  10            STAA  ACIACTRL         master reset, RTS high;
34 1034  86  1D                LDAA  #$1D          ACIACTRL := control code for
35 1036  B7  80  10            STAA  ACIACTRL         RTS low, normal operation;
36 1039  B6  80  10  T1INZW    LDAA  ACIACTRL      B := ACIA status register;
37 103C  85  04                BITA  #$04          is DCD(bit 2) low?
38 103E  26  F9                BNE   T1INZW        if not then keep waiting;
39 1040  32                    PULA                else pull A from stack to
40 1041  39                    RTS                    restore it, then return;
```

Listing 6: Character GET routine T1GET. This subroutine is called whenever it is desired to read a character from tape. After waiting for a go ahead from the receiver data available status bit, the routine transfers the input data to the accumulator A before returning. If errors occur, the error status bits are returned in accumulator B. Note that once the tape is started for input, the processing performed between T1GET calls must on the average be completed **before** *the next character is ready, if an over run error is to be avoided. For a 300 baud transmission rate, this gives 33.33 milliseconds, or 33,000 Motorola 6800 instruction cycles at 1 MHz, assuming that the output routine was called at an IO limited rate.*

```
41  1042  10  42      T1GET     EQU     * this routine reads one character of input;
42  1042  F6  80  10            LDAB    ACIACTRL  B := ACIA status register;
43  1045  C5  01              BITB    =$01      is receiver data ready (bit 0)?
44  1047  27  F9              BEQ     T1GET     if not then keep looping;
45  1049  C5  70              BITB    =$70      are there any errors (bits 4-6)?
46  104B  27  01              BEQ     T1GETR    if not then go read character;
47  104D  39                  RTS               else return with condition in B;
48  104E  B6  80  11  T1GETR  LDAA    ACIADATA  A := ACIA data register;
49  1051  5F                  CLRB              B := 0 (clear condition code in B);
50  1052  39                  RTS               return with character in A;
```

Listing 7: Input close routine T1ISTP. This subroutine is called following the last input of a series of characters (a "block"). T1ISTP immediately turns off the motor, since the T1GET routine is assumed to have been executed for the last character prior to T1ISTP. Note that the determination of the length of a block is intentionally omitted from the software of this package.

```
51  1053  10  53      T1ISTP    EQU     * this routine stops tape after reading a block;
52  1053  36                  PSHA              push A into stack to save it;
53  1054  86  5F              LDAA    =$5F      ACIACTRL =: control code for
54  1056  B7  80  10          STAA    ACIACTRL      RTS high and motor off;
55  1059  32                  PULA              pull A from stack to restore it;
56  105A  39                  RTS               return to caller;
```

Listing 8: Test Routines. The programs READ and WRITE are shown in this listing. WRITE should be called first to output 256 bytes of arbitrary data located at hexadecimal addresses 400 to 4FF in memory. Once the block is dumped to tape, the tape cassette can be rewound and set up for a playback operation. Then READ can be called to transfer the data back into the computer where the terminal monitor program (for example, Motorola MIKBUG) can be used to examine the data to verify that the interface restored it properly.

```
57  0100  01  00              ORG     $0100  start programming at location 0100;
58  0100  10  00      T1OTZ   EQU     $1000      here is a set of
59  0100  10  14      T1PUT   EQU     $1014          equates used to
60  0100  10  21      T1OSTP  EQU     $1021          tell the assembler
61  0100  10  2E      T1INZ   EQU     $102E          where the COMPLEAT
62  0100  10  42      T1GET   EQU     $1042          Tape Cassette Interface
63  0100  10  53      T1ISTP  EQU     $1053          is located;

64  0100  01  00      WRITE   EQU     * here begins the output test routine;
65  0100  BD  10  00          JSR     T1OTZ     call the output open routine;
66  0103  CE  04  00          LDX     =$400     X := starting address of block;
67  0106  A6  00      WRITELP LDAA    0,X       A := memory (X);
68  0108  BD  10  14          JSR     T1PUT     output ;= A (call the put routine);
69  010B  08                  INX               X := X + 1;
70  010C  8C  05  00          CPX     =$500     is X = last address plus one?
71  010F  26  F5              BNE     WRITELP   if not then reiterate;
72  0111  BD  00  21          JSR     T1OSTP    call the output close routine;
73  0114  39                  RTS               return to monitor;

74  0115  01  15      READ    EQU     * here begins the input test routine;
75  0115  BD  10  2E          JSR     T1ITZ     call the input open routine;
76  0118  CE  04  00          LDX     =$400     X =: starting address of block;
77  011B  BD  10  42  READLP  JSR     T1GET     A := input (call get routine);
78  011E  C1  00              CMPB    =0        are there errors?
79  0120  26  08              BNE     ENDREAD   if so then stop prematurely;
80  0122  A7  00              STAA    0,X       memory(X) := A;
81  0124  08                  INX               X := X + 1;
82  0125  8C  05  00          CPX     =$500     is X = last address plus one?
83  0128  26  F1              BNE     READLP    if not then reiterate;
84  012A  BD  10  53  ENDREAD JSR     T1ISTP    call the input close routine;
85  012D  39                  RTS               return to monitor;
```

cated by the flag, T1GET transfers the input data from the ACIA receiver data register to the accumulator A at line 48, after verifying that there are no errors in a test at lines 45 to 46. A premature return with the error code in bits 4 to 6 of accumulator B occurs at line 47 if a parity, over run or framing error was detected. The program using the cassette interface is responsible for checking any errors and possibly taking some form of corrective action. If the data had no detected errors, the normal return at line 50 is taken after clearing the error indication code in accumulator B at line 49. T1GET has two parameters which are returned in the CPU accumulators. Accumulator A contains the character which was received (or undefined garbage if in error). Accumulator B contains 0 if there were no errors, and an error condition code in bits 4 to 6 if an error occurred:

- bit 4 is 1 if there was a framing error;
- bit 5 is 1 if there was an over run error;
- bit 6 is 1 if there was a parity error.

Listing 7 completes the tape utility routines with the input close routine T1STP. Since input operations do not have to wait for completion of the last character, simply turning off the cassette motor suffices to complete the input operation. The motor is turned off by storing the hexadecimal code 5F in the ACIA control register.

A Test and Example

In order to illustrate typical use of the COMPLEAT Tape Cassette Interface, a demonstration program was written and is shown in listing 8. This demonstration program has two routines: The routine named WRITE at hexadecimal location 100 should be called from your terminal monitor program (such as Motorola MIKBUG) to copy the contents of hexadecimal memory locations 400 to 4FF onto tape as a single block of characters. (Remember to set up the tape recorder before calling WRITE.) Then the routine named READ at hexadecimal location 115 can be called to read the information back in from tape starting at location 400. (Be sure to rewind the tape and set it up for a playback operation first.)■

REFERENCES

1. Motorola Corporation: *M6800 Systems Reference and Data Sheets, M6800 Microprocessor Applications Manual, M6800 Microprocessor Programming Manual,* all 1975.

2. Lancaster, Don: "Serial Interface," BYTE, September 1975, pp 29–32.

Digital Data on Cassette Recorders

(The Demise of an Overworked Carry-Corder)

Harold A Mauch
Pronetics Corp
PO Box 28582
Dallas TX 75228

Nearly everyone has a portable cassette recorder. If you don't have one, chances are your kid does ("Hey, Mom, Dad stole my tape recorder!"). These recorders range from the under $20 "bare bones" variety to multi centibuck units with nearly every feature imaginable. Fortunately it should be possible to use nearly *any* cassette recorder available if it is clean and in good working condition. Pawnshops and similar outlets are good sources of used cassette recorders. Used recorders are often quite dirty and may need repair. Take along a couple of test cassettes when you go shopping and check out the units' operation before buying.

Watch out for bent capstans and broken cassette holders since these often are not repairable and indicate excessive abuse.

Some dictating and "pocket secretary" cassette recorders do not use a capstan drive system. While these recorders are usable, it may not be possible to exchange programs recorded on these machines with a friend. Stick with the capstan driven recorders.

While nearly any cassette recorder is usable for storage of digital information, some units have features which improve performance or convenience.

Tops on the convenience list is a *digital tape counter*. Next to destroying a valuable recording, nothing is more frustrating than not being able to find a desired program on a cassette with several programs. The tape counter solves this problem. Merely reset the counter with the cassette fully rewound and write the counter reading of the start of each program on the cassette label. Some of the newer cassette recorders also have cue and review capability. While occassionally useful, these features are not really necessary.

A recorder with an AC *bias and erase oscillator* will produce the most reliable performance and highest quality recordings. Unfortunately most of the under $100 cassette recorders now available erase and bias the tape with DC.

DC erased and biased recordings have more low frequency noise and residuals and poorer high frequency response than AC bias recordings. Cassette recorders designed for music recording usually have circuitry to erase and bias the tape with a 50 to 100 kHz signal. These same recorders usually have drive motors which are speed controlled by the power line frequency. The result is more precisely driven and recorded tape. Since

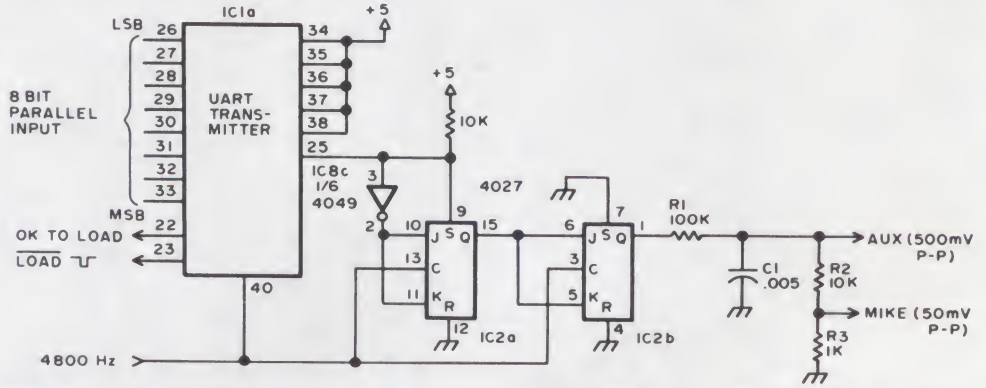


Figure 1: Cassette digital modulator. This circuit converts 8 bit parallel data from a computer into a series of 2400 Hz and 1200 Hz tones using a UART. Filtering provided by C1 and R1 is used to turn the square wave outputs of IC2b into a closer approximation of a sine wave (see figure 2).

these are normally stereo recorders, be sure to bulk erase the cassette first to remove the residual signals between the stereo tracks. If you apply the signal to be recorded to both channels, the resulting recording will be usable on any of the portable cassette players.

The cassette tape unit you select must have an auxiliary (AUX) or microphone input and a line or earplug output. This is the only reasonable way to connect the cassette tape unit to the necessary modulator/demodulator circuitry. Acoustic coupling through the microphone and speaker is totally unsatisfactory.

Pause controls are nice but not necessary.

Use the cassette tape unit available to you, but remember you only get what you pay for and these days even that costs more.

Choice of Cassette and Tape

The choice of cassette cartridge and tape has more effect on performance than *all* other factors combined. This is no place to save a penny or even a buck. Get the very best tape you can buy. Do not even consider anything less than the super tapes. If your recorder can record the chromium dioxide tapes, use them. Anything less than the best will result in much frustration. Avoid using the C90 and C120 cassettes. The tape is too thin and fragile. C60 and shorter tapes are much more rugged.

If a cassette is not in use it should be stored in its container in a dust free location. Keep the cassette tape unit spotlessly clean and do not smoke in the room in which the cassette equipment is used or stored.

It is impossible to adequately stress the importance of buying the very best quality tape and then keeping it and the tape unit clean. Tape quality and cleanliness is much more important in digital applications than in the more conventional speech or music applications.

Getting the Digital Information onto the Cassette

There are many ways to record digital information on audio cassette tapes. Many of these techniques work quite well as long as the data is played back on the same machine as was used to make the initial recording. Rather than debate the merits and deficiencies of the various techniques, the author has chosen to support the proposal suggested for evaluation by the BYTE sponsored symposium on audio digital cassette recording. I feel the proposal adequately accommodates the limitations imposed by conventionally available audio cassette tape units.

Digital information from your computer is generally available as 8 bits parallel from an IO port or data bus. The recording on tape must be serial with start and stop delimiting bits. The transmitter portion of the UART is ideal for converting the parallel data to this serial format. Figure 1 is a circuit implementing such a converter or modulator.

The serial output of the UART is said to be NRZ (non return to zero). It means that a logic one bit is a high level and a logic zero bit is a low level. A logic one causes the modulator to generate a 2400 hertz output signal and a logic zero generates a 1200 hertz signal. Normal output from the modulator is a string of square waves. The sharp edges of the square wave signal do not usually record well on recorders with DC recording bias. The designers of such recorders "roll off" the amplifier low frequency response and boost high frequency response in an attempt to diminish the drawbacks of DC biased recording. This causes a square wave to be abnormally "peaked" on the rising and falling edges and the flat portions to be "tilted." Refer to figure 2.

Such signals are more likely to cause errors during playback. Ideally the modu-

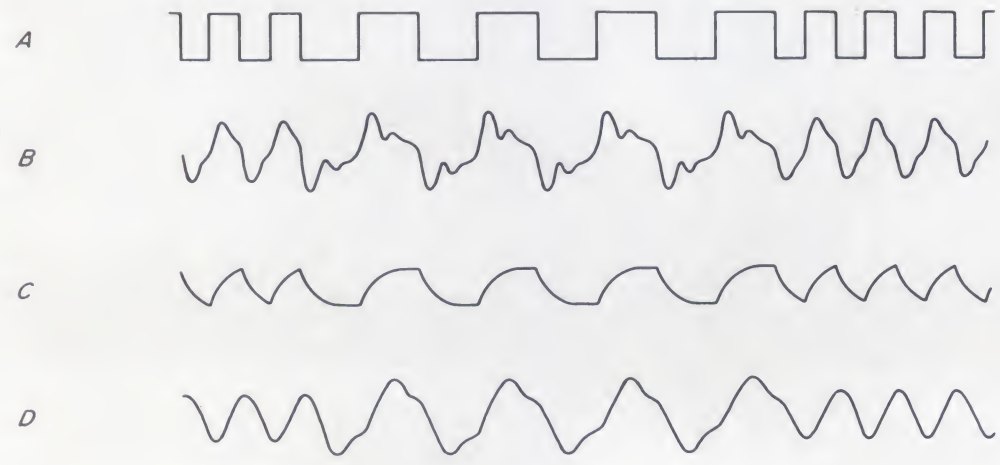


Figure 2: If a square wave signal such as waveform A is recorded on a low cost cassette recorder, the playback response may look like waveform B, which is very difficult to demodulate. If the square wave is filtered with a low pass filter before recording (waveform C), the playback response will look like waveform D, which is a usable signal.

lating signals should be sine waves but generating and switching sine wave signals digitally is somewhat complicated. "Rounding the square wave corners" with a low pass filter (R_1 and C_1 in figure 1) is not totally effective but does provide a usable waveform.

The AUX output is a 500 mV peak to peak signal. This signal level will overdrive a microphone input and should only be connected to the recorder auxiliary input (50 kOhm or greater input impedance). The MIKE output is 50 mV peak to peak and will drive most cassette microphone inputs.

The 4800 Hz signal should be as precise as possible and capable of driving 2 TTL loads. Ideally it should be obtained from a crystal oscillator and divider string or a phase locked loop (PLL) locked to the power line frequency. If such stable sources are not available the circuit shown in figure 3 is satisfactory but it must be accurately adjusted with a frequency counter.

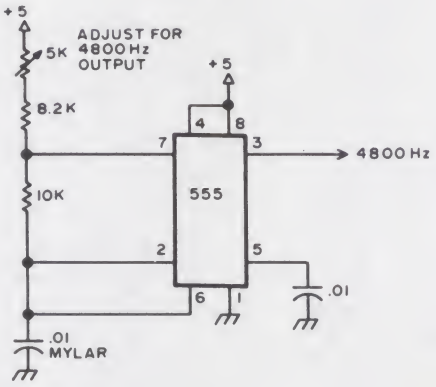


Figure 3: Circuit of a 4800 Hz oscillator. This oscillator, using the 555 precision timer circuit, can be used if a crystal controlled or line frequency derived timing source is not available.

If the available digital information to be recorded is already in serial form with the necessary start and stop bits (2 stop bits are required) and is being sent at 300 baud, the UART transmitter is not necessary. However, the 4800 Hz clocking signal should be synchronous with the serial digital information (16 clock pulses per bit). If the information is serial but at some rate slower than 300 baud, it will be necessary to use a UART receiver to first convert the information to parallel form. It is then loaded into the UART transmitter as described earlier.

When the UART transmitter is ready to accept a parallel byte of data, the OK TO LOAD line will be high. Data on the eight parallel input lines is loaded into the UART transmitter buffers by pulsing the LOAD line low for at least 1 microsecond or until the OK TO LOAD line goes low. The transmitter will start transmitting the byte or character when the LOAD line is returned to the high state.

If the UART is not transmitting any data, its serial output line is high, causing the modulator to generate the 2400 Hz signal.

Playback of the Recorded Data

Since the signal recorded on tape is basically a standard FSK (frequency shift keyed) signal, it is possible to recover the digital signal with a phase locked loop (PLL) or FM discriminator. In fact, users of the Suding cassette system (wide shift audio FSK) should be able to recover the NRZ data signal by readjusting their demodulators. However, data recovery by these means is not as precise nor as insensitive to tape speed variations as digital recovery techniques which extract speed insensitive timing pulses from the recorded signal and use these pulses to retime the NRZ data.

Figure 4 is a complete schematic of the playback recovery circuit or demodulator.

The cassette earplug output signal is conditioned by the operational amplifier Schmidt trigger IC3. IC4 is a retriggerable one shot with a period of 555 microseconds. As long as the 2400 Hz signal is being received, the one shot is constantly retriggered and does not time out. This causes flip flop IC5a to remain at the high state interpreting the data as a logic one. When the 1200 Hz signal is received, its period is long enough to allow the one shot to time out. Flip flop IC5a is immediately reset. It stays at the low state as long as the 1200 Hz signal is being received, because the one shot is timed out whenever the next triggering edge occurs. When the 2400 Hz signal returns, the one shot output stays high, thereby permitting the flip flop IC5a output to switch to its high state. The output of flip flop IC5a is the recovered NRZ serial data.

Under ideal circumstances, the recovered data would be sufficiently stable to drive a 300 baud teleprinter or TV typewriter directly. However, if the tape speed varies in excess of approximately ±6 percent (a common occurrence), errors will result. Since the 1200 and 2400 Hz signals carrying the digital information on tape will vary in frequency directly with tape speed variations, it is possible to use these signals to accurately retime the recovered data. Flip flops IC6a and IC6b extract this timing information.

When the 1200 Hz signal is received, IC6a is preset with a pulse generated by C8 and R15 every time the one shot times out. The effect is to cause IC6 to act as a division by two. When the 2400 Hz signal is being received, the one shot does not time out and IC6 acts as a divide by four. The result is a double clock rate at the output of IC6b.

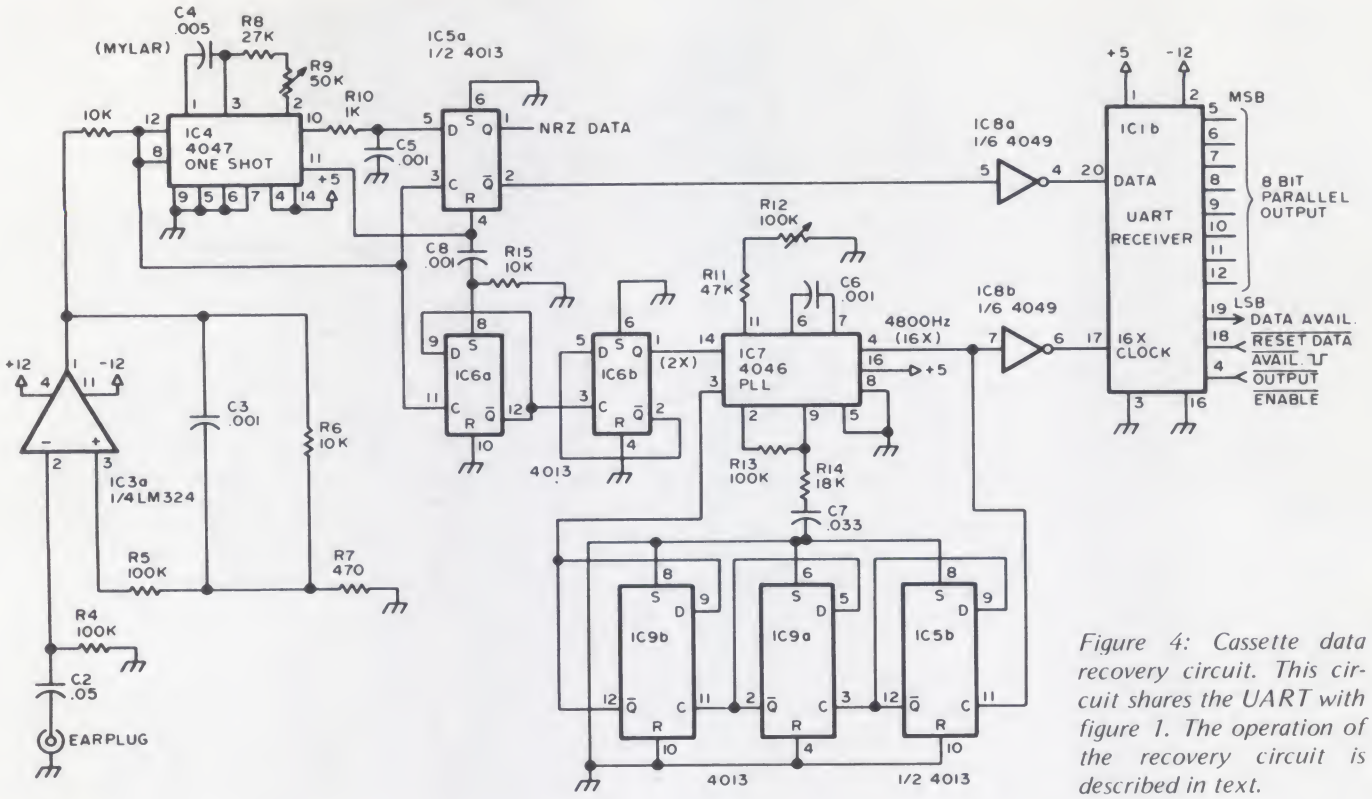


Figure 4: Cassette data recovery circuit. This circuit shares the UART with figure 1. The operation of the recovery circuit is described in text.

Instead of clocking the data into a shift register, it may be more desirable to use the receiver portion of a UART, since the UART receiver has built in circuitry to identify the beginning and end of each byte or character automatically. Furthermore, the UART parallel data outputs are 3-state, which permits convenient direct connection to most IO ports or data buses. (For a more detailed discussion of the UART, you may wish to read "Serial Interface" by Don Lancaster in BYTE, September 1975).

However, the UART requires a clock at 16 times the data rate. This problem is solved by phase locking an oscillator at 4800 Hz to 600 Hz (2X) ouput of IC6b. The phase locked loop (PLL) oscillator is adjusted for 4800 Hz in the absence of any input signal. IC5b and IC9 divide the PLL oscillator output by eight and drive one of the PLL phase detector inputs. The other phase detector input is driven by the 2400 Hz clock output of IC6b.

When the UART receiver recognizes that it has received a complete character, it raises its DATA AVAILABLE output line to logic one (high level). Since the UART outputs are 3-state, it is necessary to drive the RECEIVED DATA ENABLE input to logic zero (low level) to read the parallel output data. After the parallel data has been read, it is necessary to pulse the RESET DATA AVAILABLE line to prepare the UART to output the next byte or character. The pulse must remain at logic zero for a minimum of one microsecond or until DATA AVAILABLE drops to logic zero.

Circuit Adjustments

As already stated, the 4800 Hz signal used to drive the UART transmitter and modulate the tape recorder should be obtained from a very stable and accurate source for best results. No other adjustments are necessary on the recorder modulation circuits.

The data recovery one shot and the phase locked loop oscillator in the playback data recovery circuits must be accurately adjusted for best results. The most critical adjustment is the period of the data recovery one shot. An easy way to adjust the period is to connect a well calibrated audio oscillator to the earplug input of the data recovery circuit and a high impedance voltmeter to the NRZ data output (IC5a pin 1). Set the audio oscillator for 1800 Hz and the output level for 1.5 to 3.5 volts RMS. Adjust R9 until the voltmeter reading just changes (use the 5 to 15 volt scales). Get the adjustment as close to the point of change as possible.

The PLL oscillator is adjusted for 4800 Hz (R12) with no connection to the earplug input. If a frequency counter is not available, compare the PLL oscillator output (IC7 pin 4) to the 4800 Hz signal used to drive the UART transmitter.

Operating Procedure

The playback data recovery circuit will operate best with an earplug output signal of between 4 to 10 volts peak to peak. This is within the range of most portable cassette recorders. It may be necessary to put a low gain amplifier ahead of the data recovery circuit if you are using a cassette tape deck not capable of driving a speaker directly. It may be necessary to turn down the playback tone control if the tape was recorded on a DC biased recorder.

To comply with the BYTE Symposium Standard, the recorded block of data on tape must have a minimum of five seconds of the 2400 Hz tone before data is recorded. This is easily obtained by permitting the recorder to run in the record mode for five seconds or longer before sending data to the UART transmitter. When the UART is idle the modulator is generating 2400 Hz.

During playback it is recommended that you wait until the playback is one or two seconds into the 2400 Hz "leader" before allowing the computer to accept the UART receiver output. This is to avoid reading "trash" caused by turning the cassette tape unit on and off.

It is possible to turn the cassette tape unit on and off with a relay under computer program control using the cassette tape unit remote control input. However, the cassette will record and playback "trash" during the startup and stop intervals which may take as long as 3 to 5 seconds. The 2400 Hz signal recorded on tape before each block of data gives the computer a "trash free" interval in which to prepare itself for the data to follow.

Circuit Design Considerations

It will be some time before enough information has been learned about the use of audio cassette recorders for storage of digital information to permit truly optimum designs of the necessary modulator/demodulator circuits. Therefore the author would like to present his design considerations to provide other experimenters and designers a starting point for additional experimentation and optimization. The comments are somewhat technical and are intended for the advanced experimenter or designer.

Modulator Waveform

The nonlinearity and skewed frequency response of most low cost cassette recorders impose serious limitations on the waveform of the recorded signal. In severe cases, the waveform recovered from a square wave input may be so seriously "tilted" and "peaked" and filled with overshoots that data recovery is impossible. Obviously a better modulating signal would be a sine or triangular waveform. On the other hand, "doctoring" the square wave with filters is attractive from an economic viewpoint. Such filtering can only be carried so far before the resulting differential amplitude of the two modulating frequencies produces "pumping" of the recorder automatic level control circuits and begins to diminish the signal-to-noise ratio and signal drop out margins of the higher of the two modulating frequencies. Economical generation of a better modulating waveform will go a long way toward improving data recovery reliability with simple recorders.

Modulator Signal Level

The signal level applied to the recorder appears to be relatively uncritical. However, I feel the level should be standardized; but I am not prepared to recommend a preferred level at the present time.

Demodulator Signal Conditioning

Many experimenters have used simple zero crossing comparators to condition the playback signal. While these circuits have tremendous immunity to signal drop out, they are quite sensitive to "drop in noise" and tend to "chatter" at low signal levels or in the absence of an input signal. I prefer a circuit with sufficient hysteresis to provide some margin against the drop in noise and residuals and to prevent chatter. The ideal

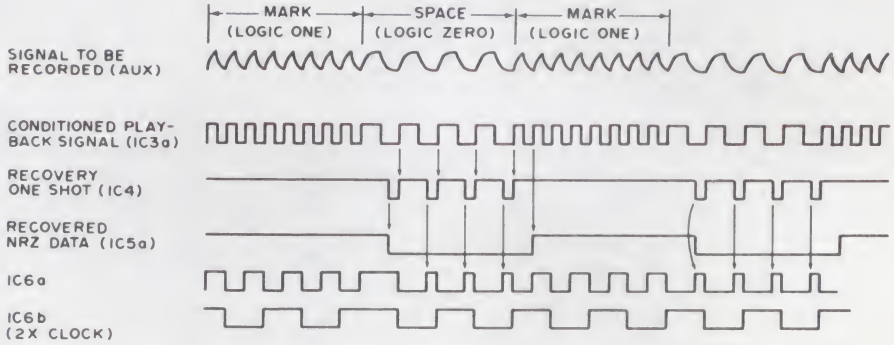


Figure 5: Cassette modulator demodulator wave forms. The signal presented to the tape recorder is a filtered square wave, shown at the top. The timing of data recovery is shown relative to the conditioned playback signal in the remaining five traces.

trip points for such a circuit is probably in the range of 20 to 30 percent of peak signal. The trip points of the circuit described in this article are approximately ±0.5 volt. Best performance will then be obtained from 3.5 to 5.0 volt peak to peak input signals.

Demodulator One Shot

If the one shot is properly adjusted, the data is recoverable with tape speed variation in excess of ±30 percent from nominal speed. I have found the speed distribution of the portable cassette tape units to be skewed roughly 5 percent negative. If a tape is played on the same unit as was used to make the recording the problem is negligible. If, however, the tape was prepared on precision tape recording equipment (such as may be used for mass production of cassettes for widespread distribution), then played on a consumer quality tape player, the tolerance of the recovery circuit to a decrease in tape speed will be diminished. This may provide some argument for increasing the period of the one shot 5 percent.

A characteristic of the data recovery circuit used is that it, causes an approximately 6 percent marking bias in the recovered waveform. This is not too important if the data is recovered by a shift register or clocked into a UART receiver. A purist approach would delay the space to mark transition 6 percent of the nominal bit cell duration.

Some experimenters filter the recovered data waveform to provide an additional immunity to error. I have not found it to be necessary and have found it creates more problems than it solves.

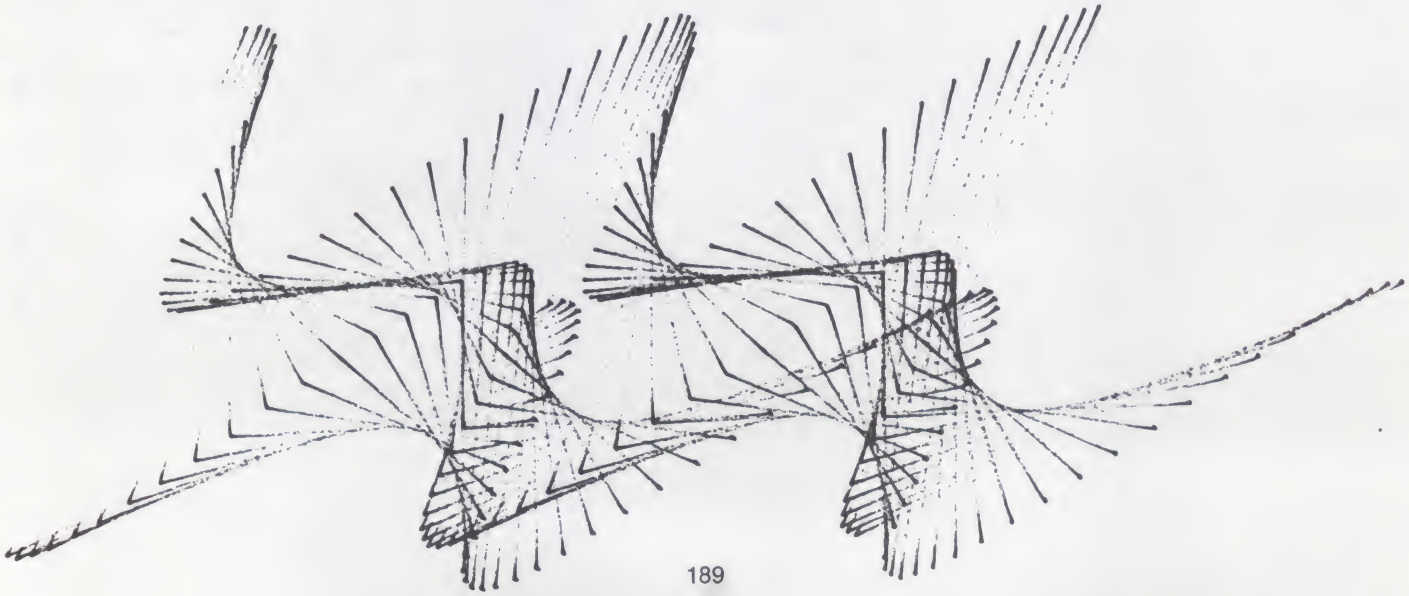
Demodulator Phase Locked Oscillator

The PLO is only necessary because the UART requires a clock at 16 times the data rate. The phase detector output is filtered with a lag-lead network. The filter was designed to permit capture of signals ±15 percent from nominal speed with a 0.707 damping factor. Consequently, the oscillator will remain locked during ±15 percent step changes of the input signal frequency. Once locked, the oscillator will track the input signal over a ±70 percent range. The sum frequency component of the phase detector output does modulate the oscillator slightly but was not considered to be a problem. This modulation can be diminished by increasing the loop filtering; however, this reduces the capture range which is undesirable.

Conclusion

The use of hardware to modulate and demodulate the cassette tape simplifies the programming problems associated with using the cassette for program loading and storage. In some circumstances it may be possible to connect the cassette hardware interface directly to your panel switches and display drivers and "let it rip." Other systems may require peripheral interface adapters or other similar circuitry to get the data onto and off the computer data bus.

The cassette interface described in this article is manufactured by Pronetics Corporation. It is available fully assembled and tested on a 4.5 x 6.5 inch circuit card with connections through a standard dual 22 pin gold plated card edge connector. Price, availability, and other information may be obtained by writing: Pronetics Corporation, PO Box 28582, Dallas TX 75228. ■



Why Wait?

Build a *FAST* Cassette Interface

Dr Robert Suding
Research Director for Digital Group Inc
PO Box 6528
Denver CO 80206

This cassette interface does not have a ±30% speed tolerance. The design requires ±12 V and +5 V to run. A good quality recorder must be used, along with excellent quality tapes. Careful adjustments are required.

So why use it? Well, it works! It's dependable. And it's fast. In contrast, the proposed BYTE standard cassette interface runs at 300 Baud. A Teletype paper tape reads @ 110 Baud. I have 24 K on my system. How long would it take me to completely load my system (not including any Bootstrap Loader operations)?

Teletype @ 110 Baud — 40 minutes 58 seconds

Proposed BYTE standard @ 300 Baud — 15 minutes 1 second

The system to be shown in this article has been running for almost a year at 1100 Baud (with an upper limit of 1750 Baud with critical tuning).

Suding system @ 1100 Baud — 4 minutes 6 seconds

Past issues of BYTE have included several articles on cassette interface proposals and circuits. I would suggest re-reading these articles. You will find one common element. Slow. If you get the impression that I'm impatient, you're right. I'll bet you are too. Imagine reading 300 Baud for 15 minutes to discover a noise pulse had destroyed data, requiring re-reading. Ugh!

Thus the proposed standard of the BYTE Kansas City conference in 1975 has a major disadvantage: The use of a redundant Manchester format with a 1200 Hz low frequency critically restricts the user to slower data rates. A related disadvantage for those who use filters or phase lock loops as an input detection method is the fact that the Manchester code employs harmonically related frequencies; this leads to design problems in detectors based upon frequency discrimination techniques.

The system shown in this article avoids the above pitfalls. It uses the non-harmonically related tones of 2125 Hz — Mark and 2975 Hz — Space. The second harmonic of 2125 Hz occurs at 4250 Hz, well down on the passband of a 2975 Hz detector. Sufficient space exists between the two frequencies to allow for reasonable recorder speed discrepancies. The higher frequencies involved permit increasing the data rate.

Several approaches are possible in cassette interfacing, as seen in past BYTE articles. However, their emphasis on wide cassette speed tolerance made them slower. My

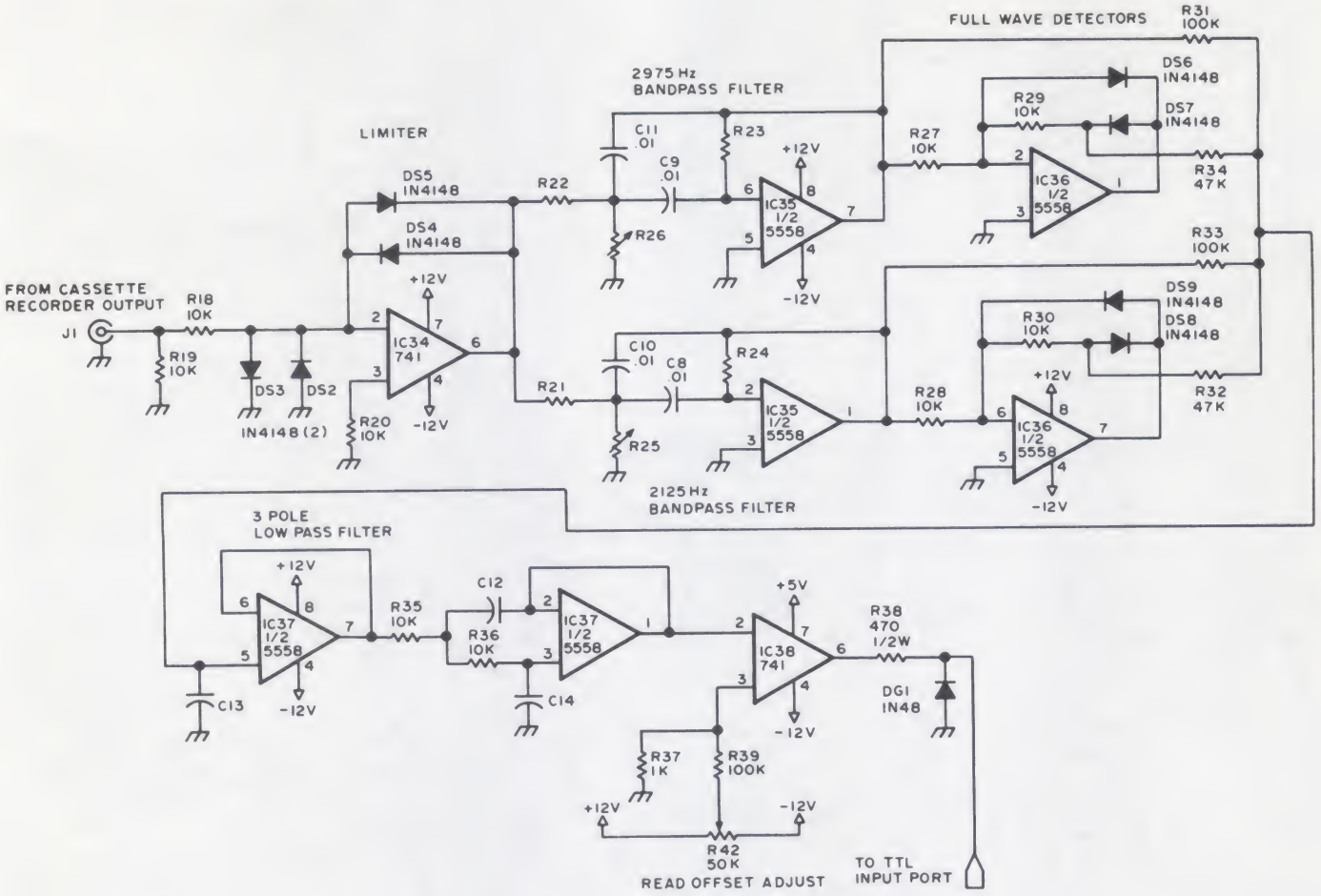


Figure 1: The schematic of the Suding cassette input interface as found in the Digital Group systems. This interface amplifies and clips the cassette output with limiting amplifier IC34, discriminates the two data frequencies (see table 1) with bandpass filters followed by full wave detectors, passes the detected signal through a 3 pole active low pass filter, then converts the result to a TTL level which is read by a single bit input port. One example of software (see listing 1) to drive this input interface uses a programmed simulation of UART input algorithm; an actual UART or ACIA device could be substituted if desired.

approach to "out of specification cassette speed" is – "put it in the specification, or get a good recorder." More of that later.

Theory of Operation

The 1100 Baud Digital Group system uses the circuits of figures 1 and 2. The cassette receive circuitry detects the prerecorded frequency shift keying and produces a "1" or a "0" output as a result of a detected 2125 Hz or 2975 Hz tone at the input. A 741 operational amplifier, IC34, is used as a clamped limiter which prevents variations in cassette amplitude from affecting the detection process. The output of the limiter should be about .6 V peak to peak, roughly a square wave with rounded edges of the incoming frequency, constant in amplitude regardless of tape volume setting or minor tape "dropout" problems.

Two bandpass active filters (IC35) then amplify a tone five times when actually tuned to their respective frequencies of 2975 Hz for the top filter, and 2125 Hz for the lower filter. The further off the tuned frequency the tone is, the less amplification the filter will produce. The gain, bandwidth, and tuned frequency are set by the three resistors and two condensers in each filter. Each filter may be exactly tuned to frequency by carefully setting the variable resistance value (which may be either a potentiometer or selected fixed values).

Full wave active detectors produce rectified full wave pulses at the summing junction, pin 5 of IC37. The 2975 Hz tones are rectified to a positive voltage, and the 2125 Hz tones are rectified to a negative voltage. As received tones depart from either exact frequency, a value less positive or

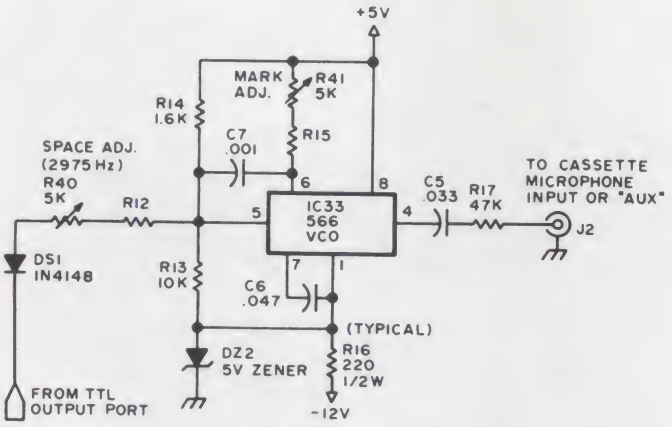


Figure 2: The schematic of the Suding cassette output interface as found in the Digital Group systems. The output interface is a simple audio frequency shift keyer made up of a 566 voltage controlled oscillator with two frequency states controlled by a single TTL data line. The TTL level which drives the output modulator is a single bit derived from an output port. The software (see listing 2) to drive this output interface is shown as a programmed simulation of a UART output algorithm; an actual UART or ACIA device could be substituted if desired.

negative is produced until approximately midway (2550 Hz) a summed voltage of 0 results.

A three pole lowpass active filter then removes the remaining traces of pulsating DC from the summed signal with almost no effect on the data pulses up to a speed of 1000 bits per second. If lower data rates were to be utilized, an improved signal to noise ratio could be obtained by multiplying the values of C12, C13, and C11 by the reciprocal of the data rate ratio. Table 1 shows some component values for alternative frequency designs.

The final receiver section is a 741 operational amplifier, IC38, connected as a slicer. This operational amplifier detects whether the voltage at its pin 2 is positive or negative with respect to the constant voltage at its pin 3. The output voltage will then swing either to nearly −12 V or to nearly +5 V. Notice that this operational amplifier has +5 as its positive supply voltage, pin 7. A forward biased germanium diode prevents the actual output voltage from going less

Tune Up Notes

The cassette interface must be carefully tuned to achieve proper performance. Careless tuning has been the most frequent cause of cassette system failure.

1. Plug in the six integrated circuits of the cassette interface.
2. Connect a calibrated audio oscillator between the limiter input and ground. A digital frequency counter driven by the audio oscillator is highly recommended. The oscillator should cover the desired range of 2 − 3 kHz, with a sine wave output of .5 or so, although the precise level is not at all critical.
3. Apply +5 and ±12 voltages to the circuit. Measure the output at pin 6 of the 741 limiter (IC34) with an oscilloscope. The wave shape should be a rounded square wave of about .6 V peak to peak.
4. Set the audio oscillator to 2125 Hz. Measure the output at pin 1 of the 5558 active bandpass filter. Slowly turn R25 until the signal peaks. Be sure that you are peaking at 2125 Hz, not a harmonic. Vary the oscillator frequency a few decades to insure 2125 Hz is the tuned frequency.
5. Similarly, set the oscillator to 2975 Hz and measure the output at pin 7 of the 5558 (IC35). Slowly turn R26 until the signal peaks. Vary the oscillator to insure a 2975 Hz peak.
6. Measure the detected voltages at pin 5 of IC37. When the oscillator approaches 2125, the voltage should go negative. When approaching 2975, the voltage should go positive. Trouble in this area would most likely be caused by reversed or defective diodes, or shorts between adjacent lines.
7. Measure the voltage at the cathode (bar) end of the output clamping germanium diode (G1). Sweeping the frequency between 2125 and 2975 Hz should result in a clean voltage jump somewhere between 2125 and 2975 Hz. Measure the output swing to insure that it does not exceed +5, −.3 V.
8. Remove the audio oscillator and short input connector J1 temporarily to ground. Measure the output at pin 6 of IC34. A stable condition (no oscillation) should be seen. Connect the oscilloscope to the cathode of G1 again. Adjust the balance potentiometer (R42) so that the output voltage is a negative level. Slowly turn the potentiometer until the output voltage jumps to a positive level and leave the setting at this point.
9. Disconnect the temporary jumper from the input connector and reconnect the audio oscillator. Perform step 7 again. The crossover threshold should be close to 2550 now. If all proceeds well at this point, the cassette interface is ready to receive data.
10. Connect the oscilloscope to pin 4 of the 566 voltage controlled oscillator (IC33). A triangular wave output should be seen.
11. Connect a temporary jumper between the TTL input going to DS1 and +5 V. Connect a frequency counter to pin 3 of the VCO (IC33). Adjust potentiometer R41 for a resultant output frequency of 2125 Hz.
12. Remove the jumper from +5 V and connect the jumper from DS1's input to ground. This time adjust R40 for 2975 Hz output.
13. Remove the jumpers, and you are ready for final tune in the driving circuit. Connect the cassette interface to the driving output port, and program the driving processor to send a TTL high level ("1") output to the cassette interface. Adjust R41 to 2125 Hz. Then have the processor send a "0" level. This time adjust R40 for 2975 Hz output. The cassette interface is now ready for use.

than $\simeq -.2$ V, so that valid TTL levels are not exceeded. An offset adjusting potentiometer allows the output to be placed in a "Mark Hold" condition when no tone input is being detected.

The cassette recording section (figure 2) uses a single integrated circuit, a 566 voltage controlled oscillator, IC33. A logic level from the computer's output port controls the resultant audio frequency output to the cassette recorder microphone input. A high input ("1") produces a 2125 Hz output, and a low input ("0") results in 2975 Hz. The output wave shape is a symmetrical triangular wave. Should the user object to using a triangular wave, a more nearly sine wave can be obtained by connecting a pair of back to back 1N914 diodes between ground and the output side of the coupling capacitor C5.

Exact values and high quality components will result in a trouble-free voltage controlled oscillator. The 47 K (R17) resistor in series with the output is a typical value to be used when coupling to the low level, low impedence external microphone inputs of most cassette recorders. Using the "AUX" input of your cassette recorder generally gives better results.

Construction

The cassette interface is available as a part of a printed circuit board kit from the Digital Group. The printed circuit board is shared by a television display circuit to be described in the next article in this series. A kit of the cassette interface only is also available from the Digital Group for $30, which includes all parts and the printed circuit board. The experienced builder can build the circuit in an evening or two by hand wiring components on standard .1 inch grid Vectorboard. All the circuitry can be contained in an area of approximately 3 inch by 5 inch (about 8 cm by 13 cm).

Be sure to use only high quality components, particularly in the active bandpass filters and voltage controlled oscillator. Some strange "frequency jump" problems have been traced to surplus 566s which were temperature sensitive. Lay out the receive circuit to avoid feedback paths from output to input, particularly in the limiter, active bandpass filters, and slicer areas. Different op amps could be used, but may result in instability or degradation of final performance due to suboptimization.

Modifying Your Cassette Recorder

It is very helpful to listen to the data from the cassette so that the beginning of the data burst may be detected, as well as hearing the end of the data. When the cassette read cable is plugged into most cassette recorders' earphone output jack, the speaker output is usually cut off. However, since a closed circuit jack is all that is involved, a quick solution is to connect a jumper on the jack so that the speaker is not disconnected. Even better, use a 100 ohm ¼ watt resistor instead of the jumper, and the data howl won't be so loud. A 10 ohm, ¼ watt resistor from the amplifier lead to jack, to the jack frame will prevent potential damage to the output driving transistor(s).

Alternative Frequencies and Applications

The cassette interface design may be used with the proposed BYTE standard should you so desire. Table 1 has appropriate component values calculated for two alternative possibilities: the simple way (less desirable) and the "right way". The simple way permits using a switch on the bandpass active filters to select the frequency pairs. The right way involves setting the circuit to the optimal values, and using separate interfaces for each frequency pair.

Amateur radio (ham) radioteletype (RTTY) generally uses 2125 – 2295 Hz frequency shift keying for 170 Hz shift. The existing cassette interface can be used by "straddle tuning," but improved performance may be obtained by selecting a second R26 which will tune the high filter to 2295. The cassette read cable may then be attached to the short wave receiver and the microprocessor, programmed as a radioteletype video terminal, which can replace the noisy Teletype machine. Of course, a cassette interface specifically designed for this 170 Hz shift at 100 WPM will give superior performance under marginal conditions.

The cassette interface may be used as a stand alone radioteletype terminal unit and audio frequency shift keying if desired, and works quite nicely in this application.

Software

I would suggest using software for your cassette read and write timings. Sample 8080 software is included as listing 1. Timings at locations <0>/116, <0>/133, <0>/241, and <0>/260 are based on an 8080 system with a 500 ns T time and no wait states. Slower systems will require proportionately decreased loop timings.

A UART could be used instead of the "software UART" system shown. However, several disadvantages arise. First, a slightly greater cost and complexity. More important, however, is a degradation in total

	Low Filter			High Filter			Low Pass Filter			VCO	
	R21	R24	R25 *	R22	R23	R26 *	C13	C12	C14	R12	R15
2125-2975 Hz 1100 Baud	6.8 k	68 k	938	4.7 k	47 k	697	.0056 μF	.01	.015	2.7 k	1.3 k
1200-2400 Hz 300 Baud (Simple)	6.8 k	68 k	4173	4.7 k	47 k	1162	.0056 μF	.01	.015	470 k	2.7 k
1200-2400 Hz 300 Baud (Correct)	12 k	120 k	1668	5.6 k	56 k	906	.015 μF	.033	.047	470 k	2.7 k
2125-2295 100 Baud (Simple)	6.8 k	68 k	938	4.7 k	47 k	1301	.0056 μF	.01	.015	47 k	2.7 k
2125-2295 100 Baud (Correct)	36 k	360 k	156	27 k	270 k	179	.056 μF	.1	.15	47 k	2.7 k

*** means that the value so indicated is the typical calculated value. The precise value is dependent on component tolerance.**

Table 1: Theoretical values of components for alternate frequencies. This table gives values of components to be used with the circuits of figures 1 and 2 in order to make this cassette interface work with several alternate specifications. See the text for a definition of the various comments at the left of the table.

Potential Troubles

Knowing about potential problem areas is a first step to minimization of their effects. Troubles seem to break down into *six* classes.

- Cassette recorders and the cassettes used: A marriage between your $1000 microprocessor and junior's $20 cassette recorder, which has been using 30¢ cassettes for the last five years, will not produce happy offspring! I have been using a Superscope C-104 for the past year, and can report no failures except for defective cassette tapes. The C-104 has several attractive features. Besides the usual conveniences such as index counter, cuing, etc, it has a variable readback speed control, dandy for out of spec cassettes from friends. Inside, another special motor speed control potentiometer is located near the speaker which allows precisely setting the record/write speed. Quality control seems good overall, and the list price of $120 (cheaper at discount stores) is worth the investment. Don't waste your money on cheap cassettes. Sony Low Noise C-45s have been generally good. Some $2 — $4 Data Certified Cassettes are superior, but not needed.
- Microprocessor caused problems: Some microprocessor designs will not work directly with this interface system. This interface was designed to be connected directly to a single bit IO port, with the processor handling all of the bit timings through timing loops. If your processor must periodically catch its breath for such things as dynamic memory refreshing, you may be unable to directly use the "Software UART" system. What a shame! However, a hardware UART will permit using the system even with a system of this nature.
- Cabling problems: It is possible to connect your cassette recorders with the read and write cables reversed. Enough crosstalk from the write line to the read limiter existed to give the appearance of data being read, but so many errors resulted that the programming would not run.
- Tuning problems: Circuit tuning is the most common problem. *Carefully* tune the active filters!
- Cassette Crashes: Cassette damage is frequent on tapes which have always worked before, but now mysteriously fail. The most common cause of this is removing a cassette from the recorder without completely rewinding. The exposed oxide then gets damaged, and is no longer usable.
- Miscellaneous circuit problems:

Defective level output from cassette read limiter.
1. None at all: Check for ±12 V to IC34, and IC34.
2. Too high output level: Diodes (DS4 and DS5) open, or one is reversed.

Bandpass active filters don't filter.
1. Off frequency
2. Bad 5558
3. Check for shorts or out of tolerance condensers C8, C9, C10, or C11. Disk ceramics are a "no-no" in tuned circuits.
4. Resistors improperly wired or inserted.

Full wave detector does not work as described:
1. Diodes open, reversed or shorted.
2. Defective IC36.

Low pass active filter fails to work:
1. Shorted or out of tolerance condensers.
2. Defective IC37.

Output slicer (IG38) fails to produce TTL levels:
1. Reversed, open or not Germanium diode at DG1.
2. Too heavily loaded output. This circuit should drive no more than one TTL load (standard for most IO ports).

VCO won't oscillate.
1. Defective 566 (IC33).
2. Shorted condenser C6.

VCO has parasitic oscillation (high frequency):
1. C7 not connected.
2. Defective 566.
3. C6 is open, producing a very high frequency.

VCO won't tune to frequency or stay there:
1. Out of tolerance or defective C6. You really didn't use a disk ceramic here, did you?
2. Defective 566.
3. Non-TTL levels used to drive VCO.
4. Defective potentiometers R40 or R41.
5. DS1 or DZ2 reversed or defective.

Listing 1: Stand Alone Suding Cassette Input Program. This program is a self contained data transfer routine which will transfer a block of data from cassette to split octal memory locations xxx/xxx through yyy/000. This program assumes that MEMTOCAS (see listing 2) was used to create the tape being read. A more generally useful input facility would be modelled on this program and linked to a system monitor as a subroutine.

	Split Octal Address	Octal Code	Label	Op.	Operand	Commentary
	<0>/100	041 xxx xxx	CASTOMEM	LXI	H,xxx/xxx	Load starting address in HL pair;
	<0>/103	021 010 000	STARTBYT	LXI	D,000/000	Load E, clear D;
	<0>/106	333 001	SYNCHLOO	IN	1	Port 1 bit 0 read for input;
	<0>/110	346 001		ANI	1	Mask all but bit 0;
	<0>/112	302 106 <0>		JNZ	SYNCHLOO	If not start bit then reiterate loop;
*	<0>/115	006 300		MVI	B,300	Time delay to middle of first data bit*;
	<0>/117	005	WSYNCH	DCR	B	Decrement synch wait count;
	<0>/120	302 117 <0>		JNZ	WSYNCH	If not done then keep waiting;
	<0>/123	333 001	GETDATA	IN	1	Read port 1 bit 0 again;
	<0>/125	346 001		ANI	1	Mask all but bit 0 again;
	<0>/127	202		ADD	D	Sum old bits with new bit;
	<0>/130	017		RRC		Rotate new and old into next position;
	<0>/131	127		MOV	D,A	Save result back in D;
*	<0>/132	006 200		MVI	B,200	Time delay between bits;
	<0>/134	005	WDATA	DCR	B	Decrement data wait count;
	<0>/135	302 134 <0>		JNZ	WDATA	If not done then keep waiting;
	<0>/140	035		DCR	E	Decrement data count loaded at 0/103;
	<0>/141	302 123 <0>		JNZ	GETDATA	If not done then repeat for next bit;
	<0>/144	162		MOV	M,D	Save received data in memory;
	<0>/145	043		INX	H	Point to next available location;
	<0>/146	174		MOV	A,H	Move high order address to A for end check;
✓	<0>/147	376 yyy		CPI	yyy	Has high order address reached end?
	<0>/151	302 103 <0>		JNZ	STARTBYT	If not then reiterate for next byte;
	<0>/154	166		HLT		End input;

Notes:

- Input is assumed to be wired to bit 0 of port 1, from output of IC38 pin 6 via resistor R38 and shunted by diode DG1.
- Loading proceeds from split octal address xxx/xxx to address yyy/000. Enter this program by jumping to location <0>/100 after setting up constants of address.
- "*" indicates a timing constant for the "software UART" inputs.
- "✓" indicates the end of transfer comparison mentioned in text.
- <0> indicates an arbitrary page location for this program, to be replaced by a real memory page number when actually loading the program at byte 100 of some page.

Listing 2: Stand Alone Suding Cassette Output Program. This program is a self contained data transfer routine which will transfer a block of data from split octal memory locations xxx/xxx through yyy/000 onto cassette tape after a five second leader output delay. This program assumes that CASTOMEM (see listing 1) will be used to read the tape being created. A more generally useful output facility would be modelled on this program and linked to a system monitor as a subroutine.

	Split Octal Address	Octal Code	Label	Op.	Operand	Commentary
	<0>/200	041 xxx xxx	MEMTOCAS	LXI	H,xxx/xxx	Load starting address in HL pair;
	<0>/203	076 001		MVI	A,1	Start port output in high state;
	<0>/205	323 001		OUT	1	Send initial state out;
	<0>/207	026 012		MVI	D,012	Outer leader delay count;
	<0>/211	006 377	LEADER5S	MVI	B,377	Outer leader delay loop return;
	<0>/213	016 377	LEADER5X	MVI	C,377	Middle leader delay loop return;
	<0>/215	015	LEADER5Y	DCR	C	Inner leader delay loop return;
	<0>/216	302 215 <0>		JNZ	LEADER5Y	If inner loop not done then reiterate;
	<0>/221	005		DCR	B	Middle leader delay count;
	<0>/222	302 213 <0>		JNZ	LEADER5X	If middle loop not done then reiterate;
	<0>/225	025		DCR	D	Outer leader delay count;
	<0>/226	302 211 <0>		JNZ	LEADER5S	If outer loop not done then reiterate;
			*			
			* Upon reaching this point, 5 seconds of mark (high) state have			
			* been output to the cassette interface.			
			*			
	<0>/231	016 011	BYTEOUT	MVI	C,011	Define output bit count (decimal 9);
	<0>/233	257		XRA	A	Clear carry (start bit level is 0);
	<0>/234	176		MOV	A,M	Move current byte to A;
	<0>/235	027		RAL		Rotate bit into position (carry=0 first);
	<0>/236	323 001	WNEXBIT	OUT	1	Send current LSB to output port;
*	<0>/240	006 200		MVI	B,200	Time delay between bits;
	<0>/242	005	WOUTLOOP	DCR	B	Decrement delay count;
	<0>/243	302 242 <0>		JNZ	WOUTLOOP	If time left then reiterate;
	<0>/246	037		RAR		Rotate new bit into position;
	<0>/247	015		DCR	C	Decrement output bit count;
	<0>/250	302 236 <0>		JNZ	WNEXBIT	If data left then reiterate;
	<0>/253	076 001		MVI	A,001	Stop bit state defined
	<0>/255	323 001		OUT	1	then sent out to port;
*	<0>/257	006 377		MVI	B,377	Stop bit value set;
	<0>/261	005	WIBDELAY	DCR	B	Decrement stop bit counter;
	<0>/262	302 261 <0>		JNZ	WIBDELAY	If time left then reiterate;
	<0>/265	043		INX	H	Increment memory address;
	<0>/266	174		MOV	A,H	Move high order address to A for end check;
✓	<0>/267	376 yyy		CPI	yyy	Has high order address reached end?
	<0>/271	302 231 <0>		JNZ	BYTEOUT	If not then continue output process;
	<0>/274	166		HLT		End output;

Note:

- Output is assumed to be wired from bit 0 of port 1 to DS1 in figure 2.
- See notes to listing 1 for listing conventions.

system flexibility. The "software UART" allows the timing constants to be dynamically modified (if desired) by detecting the variations in the stop bit timing, thereby compensating for wow and flutter. Digital integration of the incoming data bits is possible by setting a register to octal 200 at the beginning of each bit time. During the bit time, repeated sampling either adds or subtracts from the register (depending on whether 1 or 0) and a "branch minus" instruction system effectively eliminates receive problems. This digital integration detection is utilized by the Digital Group Z-80 cassette read software.

Versions of this "software UART" system have been written for 8008, 8080, Z-80, 6502, and 6800. All work satisfactorily.

Operation

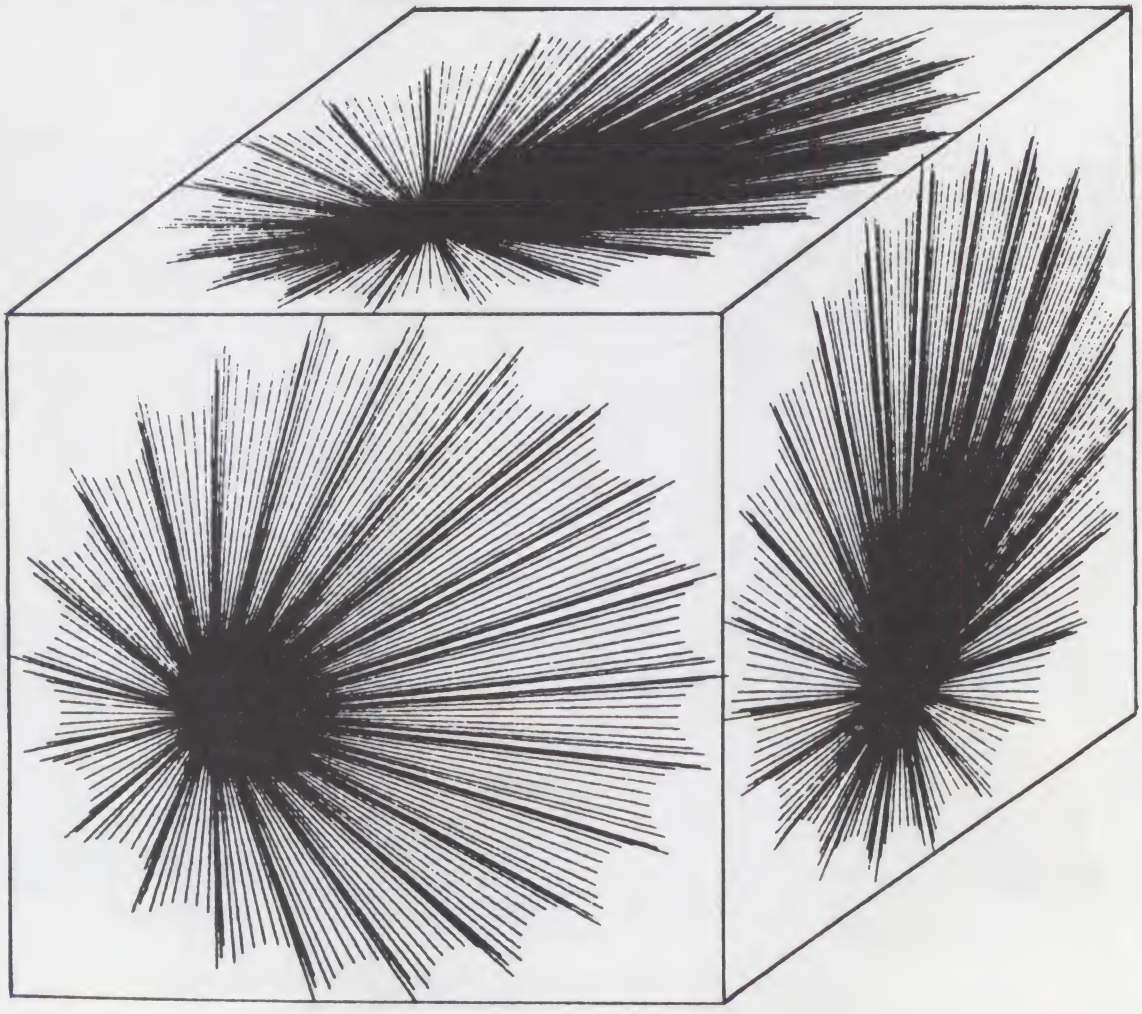
This cassette system is utilized by first turning on the cassette recorder and waiting until the lower tone 5 second leader tone is heard. At this point, restart the system to the beginning address of the "Cassette to Memory" software.

Cassette writing is accomplished by restarting the system to the beginning of the "Memory to Cassette" programming. Be sure to set the appropriate start and stop addresses prior to beginning the read or write operations. The monitor programs in the various Digital Group systems automatically set the start and stop addresses. The check marks in the listing (√) indicate the points where start and stop addresses may have to be modified.

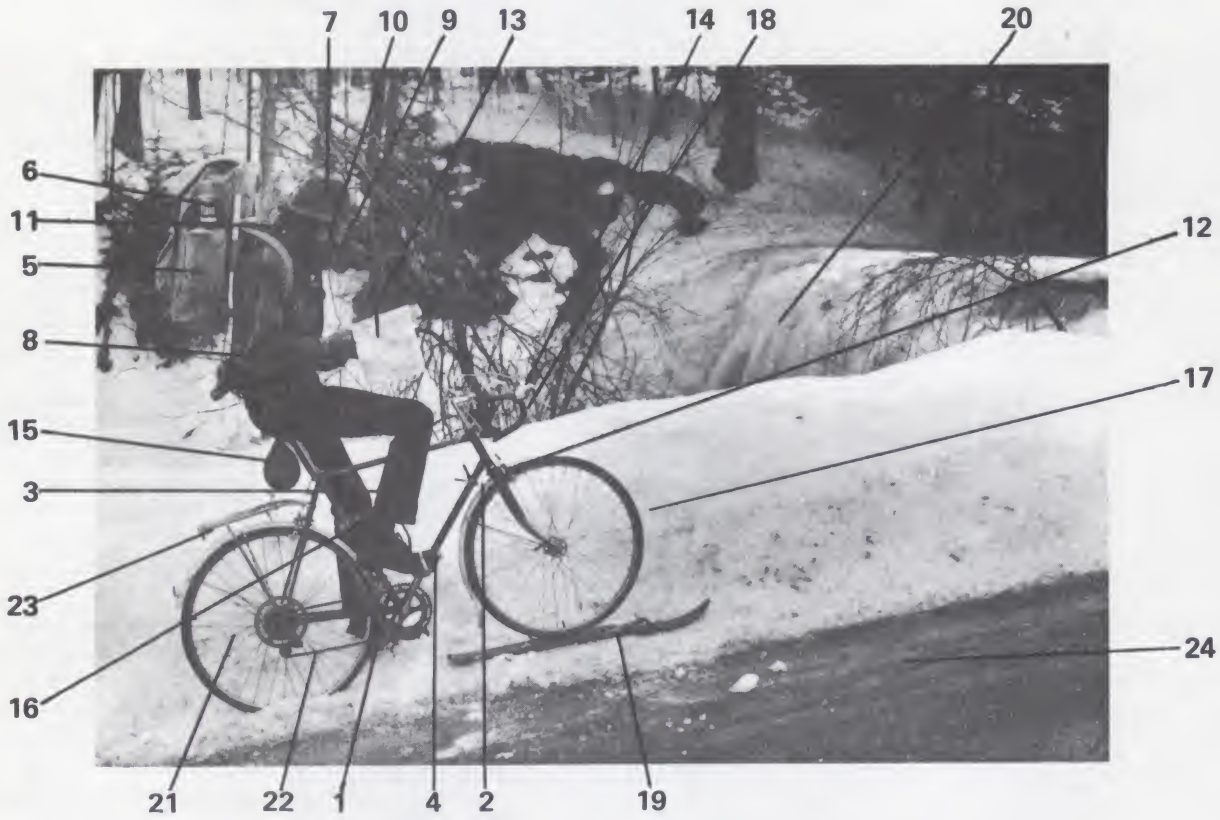
The software may be adjusted to run at different data rates by changing the values at the addresses mark with an asterisk (*). Note that the constants at <0>/133 and <0>/241 are the same. The constant at <0>/116 is 50% greater and the constant at <0>/260 is twice the value of the constant at <0>/241.

Summing It Up

This cassette interface represents a simple but fast and dependable way to store programs and data for the serious hobbyist. It does not seek to be all things to all users, but a number of applications can be run using the same basic design. The detail interface design has independence from other components in the system, allowing various processors to use the same cassette system (with appropriate software).■



Technology Update



BYTE always searches far and wide for the latest in the technology of computing systems. This month in the hills of New Hampshire, we discovered an example of computer technology in the form of the first practical Touring Machine, shown here complete with a unary relocatable based operator (in IBM OS PL/1 parlance).

For those individuals having less than a passing acquaintance with computer science, the Turing Machine is a famous mathematical construction first formulated some decades ago by Alan Mathison Turing, and which can be shown to be logically equivalent to any digital computer implementation. A Turing Machine is to computing what a Carnot Cycle is to thermodynamics. (The fact that this particular Touring Machine implementation looks like a CarNot Cycle is purely incidental.) But Turing machines have been notoriously impractical in terms of everyday computer usage until this new product rolled into town.

This newly released virtual Touring Machine, version 27 chain level 1, incorporates numerous state of the art features which make it one of the better examples of the form. These features include:

1. SHIFT (micro instruction).
2. 10 speed clock controls.
3. 2 phase clock drive.
4. clock conditioner.
5. LCS (large cookie store).
6. global debugging mechanism.
7. flying head with head crash padding.
8. access arm.
9. audio output peripheral.
10. visual input scanner.
11. audio input scanner.
12. local debyking mechanism.
13. relocatable memory mapping software.
14. HLT (halt instruction).
15. system maintenance package.
16. competing access lockout feature.
17. nomadic road interfaces.
18. tape.
19. SHIFT (macro instruction).
20. EXCP (executing channel program).
21. sectored disk drive.
22. transmission links.
23. unallocated stowage.
24. machine environment (circa January 30 1976).

What's in a Video Display Terminal?

Don R. Walters
3505 Edgewood Dr
Ann Arbor MI 48104

Let's look at the video display terminal as a black box which is connected to a computer system (somehow) as depicted in figure 1. Since the computer system has already been explained (at the block diagram level) in BYTE ("The State of the Art" by Carl Helmers, November 1975, page 6), we will concentrate on what smaller black boxes make up a video display terminal.

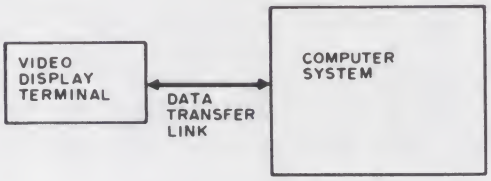


Figure 1: Two black boxes: the video display terminal and the computer system.

Figures 2 and 3 illustrate subassemblies typically combined to form the video display terminal. We see that the video display terminal is actually made up of some more familiar subassemblies, such as a keyboard, video display controller, video display, and a parallel (figure 2) or a serial (figure 3) interface. Let's take a closer look at each subassembly and see what its function is. The keyboard is a man-machine interface which is used to enter data (alphabetic commands, instructions, and/or numbers) into the computer system. When a key is pressed, the equivalent electrical code assigned to the character is generated and is available in parallel form. Thus all bits of the character's code are available at the same time at the output of the keyboard.

Now that the electrical codes of characters can be easily generated using the keyboard, how will that data be transferred to a computer system? Since the data from the keyboard is already in a parallel form, the data could be transferred to the computer system through the parallel interface in figure 2. The parallel interface handles the buffering of the data between the keyboard and the computer system (which must also have a parallel IO interface). The parallel data from the keyboard could also be sent to a computer system in serial form by using the serial interface of figure 3. Serial interfaces are usually used when the data path between the video display terminal and the computer system is longer than five feet, which would be the case when the video display terminal is to be connected to an acoustical coupler. The coupler is a device which changes serial data into frequency shifted tones to transmit the data over voice grade telephone lines so that a terminal can be used with a remote

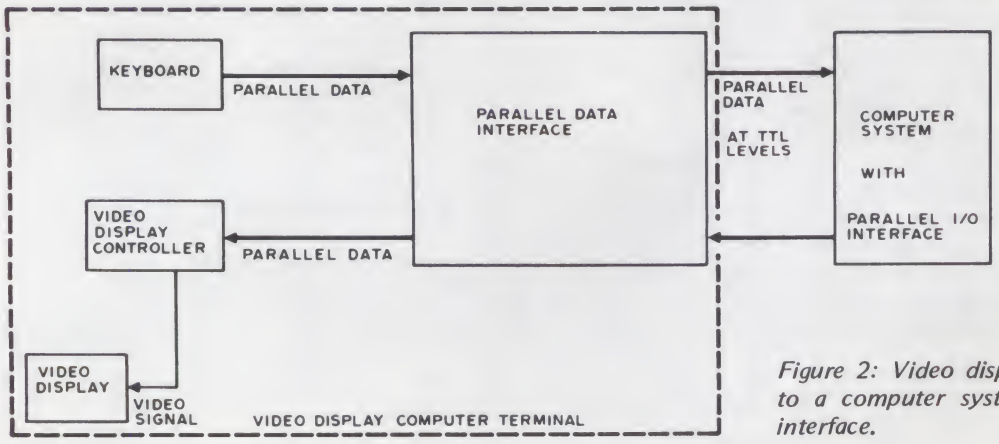


Figure 2: Video display terminal interfaced to a computer system through a parallel interface.

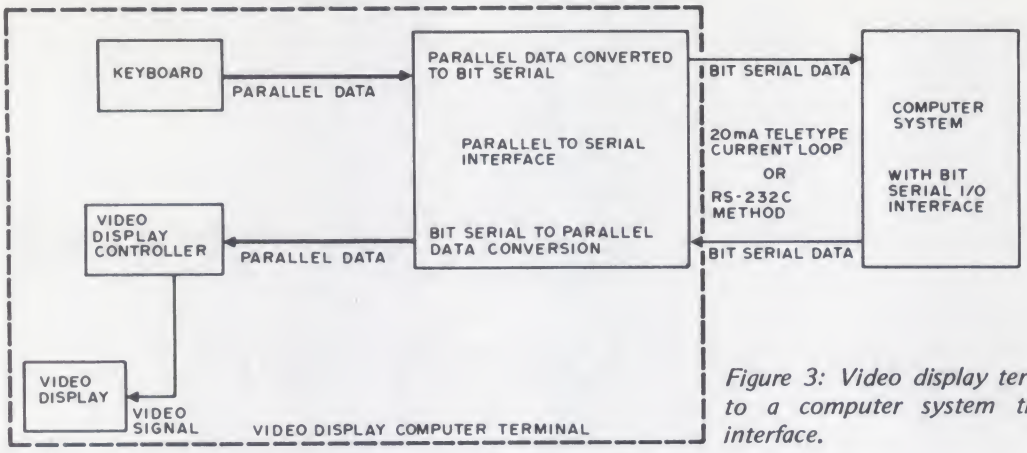


Figure 3: Video display terminal interfaced to a computer system through a serial interface.

computer system via the telephone. The serial interface converts the parallel data from the keyboard to a bit serial form. In this form, the bits of the character are sent one bit at a time until the entire character has been sent. Of course the computer system must also have a serial IO interface.

We now have traced the data path from the keyboard of the video display terminal to the computer system. Let's trace the data path from the computer system to the video display terminal. Data is sent from the computer system to the video display terminal in parallel or serial form with the same type of interface (parallel or serial) as is used between the keyboard and the computer system, except that each data path must have its own interface electronics.

The data from the interface (parallel or serial) is fed to the video display controller in parallel form. The video display controller converts its parallel data input to a composite video signal which causes the video display to show the desired characters.

The video display portion of the terminal is essentially a TV set without the RF tuner, IF amplifiers, and mixer circuits, but with the necessary circuitry to display a video signal on a CRT screen.

As you can see, the video display terminal is not a very complicated black box after all.■

John M Schulein
Homebrew Computer Club
P O Box 626
Mountain View CA 94042

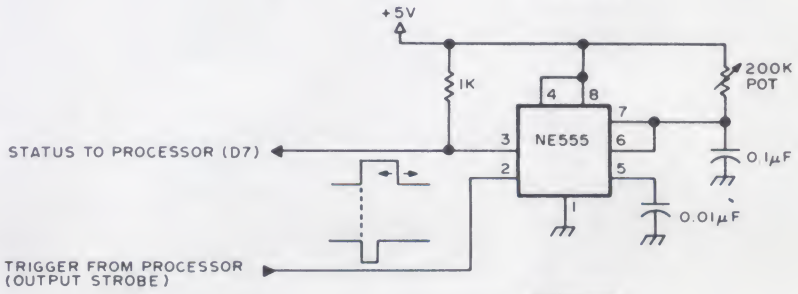
Pot Position Digitizing Idea

A scheme to convert the position of a potentiometer arm into a digital value, using a cheap commonly available timer IC (NE555) and a few bytes of program in an 8008 or 8080 microprocessor, is shown in figure 1. The software is organized as a subroutine and uses the flags and the A and B registers. The NE555 is triggered by the OUT TRIGGER instruction and then the program monitors the output pin of the NE555 in a loop that increments the B register. When the NE555 times out, the program exits from the subroutine and the B register contains a digital representation of the pot position.

The hardware and software shown in figure 1 was run on an 8008 system with a 2.5 μs clock and the B register digital output varied from 2 to 65 Hex. The values of the pot and/or the timing capacitor can be modified (see the NE555 data sheet) to suit your processor's speed and the desired range of the digitized output.■

Figure 1: Pot Position Digitizing Idea.

Hardware



Software

```
POTPOS:  MVI   B,0
         OUT   TRIGGER
CONT:    INR   B
         IN    STATUS
         ANA   A          ;Sets sign flag
         JM    CONT
         RET
```

NOTES: 1. Software written as a subroutine for the 8008 or 8080 microprocessors.
2. The flags and registers A and B are affected by the subroutine.
3. Register B contains the pot position on exit.

Read Only Memories in Microcomputer Memory Address Space

Dale Eichbauer
Digitech
PO Box 6838
Grosse Pointe MI 48236

The important advantages of a ROM in microcomputer use are nonvolatility and write protection for whatever data it holds.

In an earlier BYTE (see "Read Only Memory Technology," page 64, December 1975), Don Lancaster introduced the use of read only memories as a tool for design at the hardware level. This application is but one of a multitude of uses for ROMs, especially when you consider a ROM as part of the main memory address space for your computer. The important advantages of a ROM in microcomputer use are nonvolatility and write protection for whatever data it holds. It relieves the user from the chore of reentering frequently used programs each time his machine is fired up or after data is accidentally modified. To put it simply, your data is *always* in the machine whenever you need it.

A bootstrap or absolute loader is a simple program which just transfers data from an input device to memory. To keep it in your machine, it should ideally be in ROM.

The two most common and well known uses of ROMs are for holding loaders and system programs. There are two basic types of loaders: the bootstrap (or absolute) and the more complex relocating loaders. The bootstrap or absolute loader is a short program which is used to load the machine following a power interruption or any other type of catastrophic failure which wipes out the main programmable memory. (Unless your machine's programmable memory is of a special design, it is volatile, meaning that its data is lost if power to the memory is lost for more than a very short time.) This loader program requests input from a peripheral device such as a paper tape reader or cassette drive which contains programs needed for machine operation and stores this input data in programmable memory. After toggling all your data in from the front panel following power interruption, one can easily see both the convenience and versatility of such a bootstrap loader.

System monitors are prime targets for ROM technology.

The relocating loader takes the input data from the peripheral device, converts its addresses from a relocatable form into absolute binary and stores it in memory at selected addresses. It might typically perform some error checking and turn over execution to the loaded object program.

Monitors and Debuggers

System programs suitable for or, preferably, in ROM include such things as system monitors, assemblers, device drivers, software debugging programs, hardware fault testing and diagnostics. The system monitor (which is often available from the computer or CPU manufacturer) is a program which handles and coordinates machine operations at a basic level. A monitor allows the user to control the entire system's operation with simple, powerful commands. A typical monitor might have commands for the creation, modification, and deletion of files, device independent IO (from the user's point of view), automatic assembly and execution of programs, relocation of programs and data, and so forth. Device drivers (short programs which handle the software end of peripheral interfacing) are rarely changed once debugged and are needed for almost all IO operations, making them a natural for ROM storage. Software debugging programs, often manufacturer supplied, provide a means of detecting and correcting programming faults. The many forms and features which they possess are too extensive for any detail in this article. One rather unusual but potentially useful application of ROM storage is in storing hardware testing and diagnostic routines. Testing of the microcomputer often can be done by simple programs which execute an algorithm and compare the results with the correct answer. It can also be done by complex programs which execute all functions of the machine, often in certain critical combinations peculiar to the

machine under test. At first it would seem that there is no need to put these routines in memory of any type until needed except for convenience, since it would be an infrequently used task. Consider, however, the case where a fault which is to be located is in some way related to or impeding the input or the programmable memory's storage functions. If this is the case, then the testing or diagnostic routine may never get into the machine in usable form to do its job.

Simulation and Emulation

Simulation is another use of ROMs in microcomputers which will become more common as CPU capabilities increase, machines proliferate, and users demand more of their machines. Simulation is the technique of interpretively executing an instruction set for one computer design using a program running on a second "host" machine. For example, a host machine with an 8080 CPU could execute object programs from another machine which uses a 6800 or PACE CPU (or even IBM 360/370 software for those with delusions of grandeur). A ROM could contain the simulator program to execute the foreign instruction set. With an appropriate general purpose simulator program it might even be possible to change the instruction set of a machine by referencing a different ROM data table for each simulated machine. Of course all such simulations run much more slowly than the actual speed of the computer in question.

A related technique is emulation, in which microprogrammed hardware implements an instruction set directly. Some microprocessors are internally microprogrammed, but the user typically will not see this fact externally. Microprogrammed computers are fairly widespread in contemporary technology. And with nearly every microprogrammed computer, there is a control store implemented in some form of ROM. But the majority of microprocessor chips currently available do not give the user a facility to use microprogramming techniques. The instruction set is typically committed by the manufacturer during the design stage; so, to perform the software of a foreign machine, a software simulator must be used as described above.

With such simulations, the slowness of operation is due to the fact that a series of instructions (a subroutine) must be executed on the host computer in order to achieve the effect of a single instruction of the simulated machine. Even though a simulated computer may be 10 to 50 times slower than the real machine, such slowness is often tolerable when compared to the time it would take to hand translate the program. Use of ROM to store the simulator makes the simulation mode virtually a part of your hardware, protected from destruction due to power loss or accidental modification during program execution.

Subroutines

Another excellent use of ROMs is the storage of subroutines. Multiply, divide, double precision, floating point, conversion formulas and other algorithms, plus additional software implemented functions are in the machine as soon as power is applied. When they have been implemented in ROM, such subroutines act as if they were really hardware instructions.

If you plan to do a lot of simulation, the simulator program might be a logical choice for ROM. With diligent software preparation, your humble 8008 could simulate a mighty 360/370 (although much much more slowly in execution).

Security Data

Anyone assembling a multi user computer system, especially one with remote access, should consider using a ROM for maintaining data pertinent to the various users of the system. This data might include such things as access codes, what devices and memory segments are authorized for use by which individuals, the particular user's system priorities (for job and device scheduling by the operating system), and so forth. The operating system constantly needs such information to make decisions concerning the handling of tasks for the current users. A ROM protects this information from modification or destruction, whether accidental or malicious.

A library of often used subroutines is another item which would make a good candidate for ROM storage.

Tables

An excellent use for ROMs is the storage of tables of values. There are many tables, such as logarithmic, sine, cosine, and tangent values, which could be of use to almost any computer hobbyist. A program needing one of these values then has to merely look up the desired value in the appropriate ROM table. Such tables can also be used to speed up high precision calculations by giving an approximate starting value. Those faced with interfacing a non-ASCII encoded terminal or other peripheral (such as EBCDIC, Selectric, Baudot, or Hollerith) to their microcomputer may find that a character conversion table, implemented in ROM, is part of the solution, as Don Lancaster points out in BYTE #4. However, while his conversion scheme uses a ROM which does its conversion of data apparently at the peripheral itself, in many cases it would be useful or desirable to perform this conversion in the machine. Such a conversion method would even make it possible for two terminals, whatever their coding scheme, to commu-

Data tables for character code conversion via software can be stored in ROM if they are used a lot.

nicate with each other using the microcomputer (and its ROM) as a sophisticated interpreter. And, if data rates, character lengths, and line lengths are different, then such a setup offers the added advantage of using software and memory as a buffer to compensate for these differences.

Waveforms

If your machine is equipped with a DA converter (digital to analog converter), then a ROM can contain a set of values which, when output through the DA, will produce a custom waveform. In many cases special waveforms may be generated in this fashion which would be impractical to generate, using any other method. Both the frequency and amplitude of the waveform may be controlled completely by software. With an 8 bit word and a DA with 10 volts full scale output, resolution of 0.04 volts per bit is obtainable. The maximum generated frequency is dependent on the speed of the microcomputer and the number of outputs per cycle required for a suitable waveshape.

If you want to use your computer as a low frequency (audio) waveform generator, you could burn a set of standard waveform patterns into ROMs, using software to drive a DA conversion device at various frequencies.

Error Checking and Arithmetic

Two other possible uses for ROMs which may be implemented either in main memory or as processor add-ons are a parity generator/checker and a fast multiplier/divider. A table for all possible combinations of a word can be referenced to generate the parity bit or a flag check bit. Multiplication and division may also be done as table functions. Several of the IC fast multipliers currently available are actually modified and specially programmed ROMs.

The article in BYTE #4 also introduced Programmable Read Only Memories (PROMs), which are the most useful type of ROM for computer hobbyists, since a custom pattern costs very little to have programmed or the user can do it himself.■

Bibliography on ROMs and PROMs

These articles are found in engineering publications, which should be available in well stocked corporate or university libraries.

"PROMpting a minicomputer" by Robert Hightower of Motorola in the February, 1973, *Electronic Engineer/Systems Engineering Today*. This is a description of a bootstrap (or absolute) and a relocating loader for a PDP-11 which is stored in ROM.

"PROMs, Proms, Promises" by Jerry Metzger in June 16, 1975, *Electronics Products Magazine*. This is a good introductory article on PROMs and includes a wall chart of all PROMs available, both bipolar and MOS, as of its publication.

"PROMs – a practical alternative to random logic" by Dave Uimari of Signetics in the January 21, 1974, *Electronic Products Magazine*. Here is an excellent article on PROM theory and use which also includes lengthy discussions on programming, such as how it is done, best place to have it done, typical large and small scale equipment, etc.; lists PROM programming services and equipment manufacturers.

"Designer's Guide to Semiconductor Memories – Part 1" by Robert J Frankenberg of Hewlett-Packard Data Systems in August 5, 1975, *EDN* magazine. This is a good introduction to all types of memories, ROMs and PROMs included; it also includes an excellent list of references.

"Read-Only-Memories in computers – where are they headed?" by Roger R Dussine of Compagnie Honeywell Bull and Robert M Zieve of Honeywell Information Systems in the August 1, 1972, *EDN* magazine. The authors provide an overall survey of ROMs, their use in computers, mentions use for fault location, bootstrap, some unusual types of ROMs, and things to come in ROM technology.

"Programmable ROMs offer a digital approach to waveform synthesis" by Karl Huehne of Motorola in the August 1, 1972, *EDN* magazine. This is a detailed description of ROM waveform synthesis.

"Large Bipolar ROMs and p/ROMs Revolutionize Logic and System Design" by Joe McDowell of Monolithic Memories, Inc in the June, 1974, *Computer Design*. Here you'll find a short survey of the current bipolar ROM technology and some examples of use, including a ROM controlled timing pulse generator under microcomputer command.

"Mixing Memories in Minicomputer-based Control Systems" by Richard A Farwell of Data General in the February, 1973, *Control Engineering*. This is a discussion of how various memories are used in Data General minicomputers and the costs and tradeoffs involved; a section on ROMs lists a number of uses outlined in this article.

Manufacturer's data sheets on particular devices contain a wealth of information and are free for the asking. As an example, the data sheets below contain listings of ROM and PROM lookup tables of values.

From AMI:

- A 256 word sine and cosine table in the S8614 supplemental note.
- An arctan table in the S8771 supplemental note.
- A 512 word sine and cosine table in the S8772 data sheet.
- A Hollerith to USASCII conversion table in the S8457 data sheet.
- A USASCII to Hollerith conversion table in the S8539 data sheet.

From Nitron:

- A Hollerith to ASCII conversion table in the NCM 1112 data sheet.
- A Selectric to ASCII to Selectric conversion table in the NCM 1151 data sheet.
- A 512 word sine and cosine table in the NCM 1141 data sheet.

From Computer Microtechnology:

- ASCII to EBCDIC and EBCDIC to ASCII conversion tables in the CM 2850 supplemental note.

More Information on PROMs

Roger L Smith
4502 E Nancy Ln
Phoenix AZ 85040

Have you ever wanted to program your own read only memories automatically so that you could copy programs into a permanent storage device? This article concerns one kind of erasable read only memory, the Intel 1702A integrated circuit and its pin compatible equivalents the National MM5202AQ and MM5203Q. These memories store 256 eight bit bytes of data using a method which allows total erasure and reprogramming many times. The method of programming is complex while erasure can be accomplished simply by exposure to an ionizing radiation (such as ultraviolet light). When you need to store large tables of data or programs, use of such read only memories is a very attractive alternative to more elaborate types of memory provided a method of programming is available. These erasable read only memories are economical as well, since typical prices at the time of this article are in the $20 range.

Why PROMs?

A few years ago, it became apparent that the different users of read only memories (ROMs) had many special applications which required only one or two copies of any given data pattern. The technology of *mask programmed* read only memories is only cost effective for large production runs of parts so an alternative had to be found. A means was needed for the user of read only memories to inexpensively field program one or two copies of a data pattern. This is where Harris Semiconductor, a division of Harris Intertype Co., entered the picture and coined the term PROM for programmable read only memory, a Harris trademark that has become almost generic through widespread use. A PROM then was simply a ROM that could be programmed in the field. While production read only memories are manufactured from specific masks provided weeks in advance by the user, a PROM can be programmed in seconds automatically by the user reducing turn-around time to a minimum.

Types of PROMs

Let's examine some of the different PROMs in use today. There are a number of options for the memory elements used in making programmable read only memories including nichrome fuse links, diode matrices, stored charge devices, amorphous semiconductors, polycrystalline silicon fuses, etc. Note that all these memory elements can be electrically altered in order to store data. A few can also be restored to the original condition; these are used in erasable read only memories (EROMs).

Figure 1 illustrates how the basic PROM operates. The first thing to notice is a decode circuit. This decodes the address to select one of the 32, 64 (or whatever) word gates in the memory matrix. The decoder is simply an array of multiple input gates with one input for each address bit and one gate for each memory word.

Each decoder gate drives a multiple emitter word driver transistor. In series with each emitter is a memory element which in this case is a fusible link. In this example, we have a 4 bit word so each word driver transistor contains 4 emitters, each connected to a fusible memory element. The memory elements then connect to the appropriate bit sensors and output buffers (4 in this example).

When a particular word is addressed, its decoder and word driver transistor turn on. If the fuse link is intact, the bit sensor turns on and the output line for that bit goes low (logical zero). If the fuse link is open, the

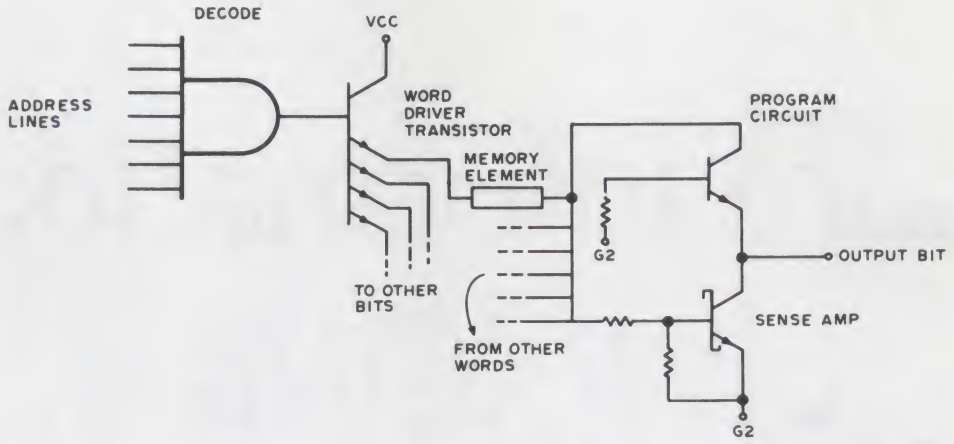


Figure 1: This partial schematic of a PROM shows the circuit for one word and one bit. This PROM would be the nichrome fuse link type.

sensor and buffer circuit remains off and the output is high (1 bit).

Not shown in the diagram are the chip select (or chip enable) lines. The chip select lines are typically connected to the higher order address bits. When many PROMs are utilized, an external decoder circuit (such as 74154 or 7442) might be used to decode several high order address bits and decide which PROMs to enable or select. Essentially, the chip select inputs are used to turn on the output bit sensors and buffers when the PROM is selected. PROMs use open collector or tri-state output buffers so that they can be bused. The buffers are in the high impedance state until enabled.

The nichrome fusible link type of programmable read only memory is manufactured by Harris, Signetics, Texas Instruments, and Motorola. From this basic nichrome fuse PROM, other types have evolved. The next natural step was to polycrystalline silicon fuses, as made by Intel and Advanced Micro Devices. These are easier to build in the semiconductor fabrication process because the fuse links are also made out of a semiconductor material. The silicon fuses are burned open in the same manner as the nichrome fusible link type. Due to the semiconductor structure of the memory elements, these PROMs often require a more elaborate programmer than the nichrome fuse type.

Another development in memory elements is the Avalanche Induced Migration (AIM) device patented by Intersil. Fabrication of these elements is similar to TTL logic which simplifies the manufacturing process. The elements are basically NPN transistors arranged in a matrix with common collectors on the X-lines and common emitters on the Y-lines. In programming a logical one, a high current is forced through the desired transistor from emitter to collector. The emitter to base junction is forced beyond normal avalanche and into secondary breakdown. Aluminum flows into the junction causing a base to emitter short that in effect leaves a base to collector diode. These PROMs are programmed using 2.5 us pulses of 200 mA current that are alternated with sense pulses. After a number of pulses, a change is sensed and the programmer moves on to the next bit.

Erasable ROMs

A memory element used by Intel and National Semiconductor is a stored charge type called a FAMOS transistor. FAMOS stands for *floating-gate avalanche-injection* MOS charge-storage device. It is similar to a P-channel silicon gate field-effect transistor with no contact on the gate. Programming the FAMOS type of memory element requires a pulse more negative than −30 volts applied to the drain or source P-N junction. High energy electrons are injected into the floating silicon gate. With this negative charge on the gate, there is current conduction between the source and drain of the FAMOS transistor.

The primary advantage of this stored charge type of memory element is that the charge can be removed later by exposing it to a high intensity, short wavelength ultraviolet light. The radiation creates an ionizing action that causes the charge on the floating gate to leak back to the substrate. These erasable ROMs (EROMs) are provided with a transparent quartz lid to allow exposure to the radiation. More about erasure later.

For the really dedicated computer hobbyist who wants all of his system monitor, resident assembler, text editor, etc. in PROMs because they are all working as desired (at least this week), erasable ROMs

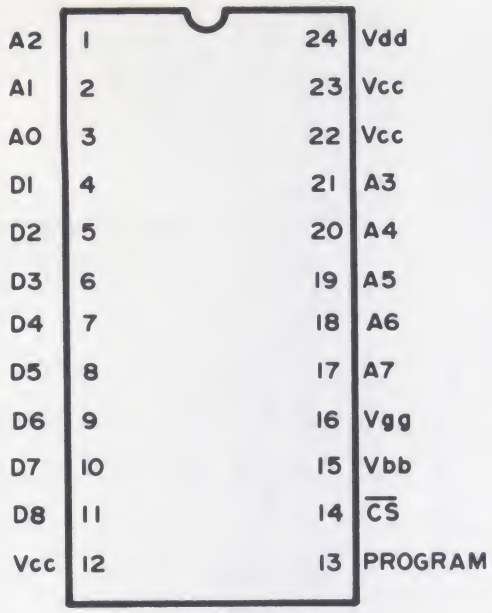


Figure 2: Pin-out diagram Intel 1702A EROM. A0 — A7 = address inputs; D1 — D8 = data output (for READ mode), data input (for PROGRAM mode); $\overline{CS}$ = chip select.

are the logical choice. Currently available for around $20 are the 2 Kb Intel 1702A and National MM5202AQ and MM5203Q. All of these EROMs use the FAMOS stored charge memory elements and can be erased with ultraviolet light. These EROMs have one definite advantage over regular ROMs; they have been tested before delivery.

Intel 1702A EROM

The Intel 1702A EROM is produced in a 24 pin dual in line package with a transparent quartz lid. Intel also makes a 1602A ROM which is identical to the 1702A except that it has a metal lid and is not erasable. All chips undergo complete programming and functional testing on each bit position prior to shipment. The 1702A and 1602A are both 256 word by 8 bit, entirely static MOS ROMs with no clocks required. All inputs and outputs are TTL and DTL compatible, but the outputs are tri-level to allow output busing capability. Memory expansion is simplified by use of a chip select input which disables the chip when high (logical one). Figure 2 shows the Intel 1702A pin connections while table 1 shows the voltage inputs for the read or program modes.

Erasure Methods

To erase EROMs such as the 1702A, Intel recommends using the Model S-52 ultraviolet lamp available from Ultra-Violet Products Inc., San Gabriel CA (cost is about $170) or through Intel distributors. An inexpensive eraser can be built for about $15 using a General Electric ultraviolet lamp #G8T5, a ballast transformer, single pole switch, a push button starter switch, and mounting hardware. The lamp is mounted in an enclosure and the EROM is placed under it at a distance of 0.25 inch. The lamp is turned on for about 6 minutes for complete erasure, but use caution not to expose anyone to the ultraviolet rays.

Table 1: Intel 1702A EROM input voltages.

Pin	Read Mode	Program Mode
12 Vcc	5 V	ground
13 PROGRAM	5 V	Program pulse (–46 V to –48 V)
14 $\overline{CS}$	ground	ground
15 Vbb	5 V	12 V
16 Vgg	–9 V	Pulsed Vgg input (–35 V to –40 V)
22 Vcc	5 V	ground
23 Vcc	5 V	ground
24 Vdd	–9 V	Pulsed Vdd input (–46 V to –48 V)

CAUTION: When using an ultraviolet lamp, you should exercise extreme care not to expose your eyes or skin to the rays. Short wave ultraviolet light can cause sunburning of the eyes and skin.

According to a National Semiconductor engineer, the ultraviolet erasable EROMs cannot be indefinitely erased and reprogrammed. After about 52 cycles of reprogramming, the device will not work properly unless it is reconditioned by baking in an oven at 400°F for 45 minutes. After reconditioning, the program-erase cycle can be repeated another 52 times, although the National Semiconductor engineer recommends only 35 cycles between reconditioning.

EROM Programming

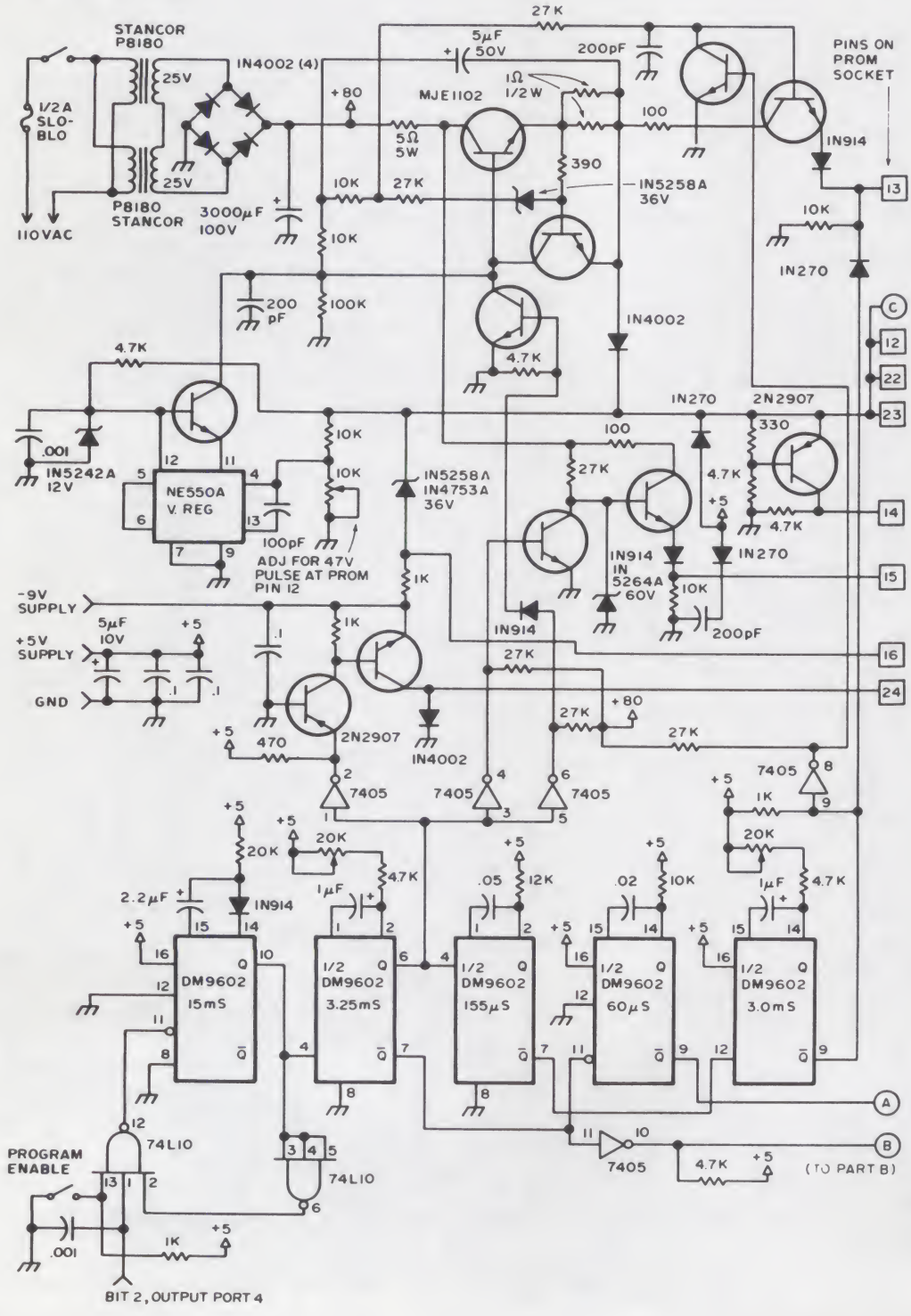
We'll describe two possible methods of programming these EROMs. The first method is *highly* recommended, will prove least expensive, and is extremely simple — order it programmed or send it in to be programmed!

Many EROMs are not simple to program. The 1702A type EROM requires a series (over 32) of 47 volt programming pulses of 3 ms duration with 20 percent duty cycle for each word. Also, at the beginning of each pulse, the address must be complemented.

Manual programming is out, and the cost of an automatic programmer may not be justified. Remember also that in order to erase programs you must buy some type of ultraviolet lamp. A PROM/EROM programmer could, however, prove to be a very interesting and fund raising activity for an industrious computer club.

If this isn't convincing enough, or if you plan on going into the business, or if you're just plain curious, you may want to try the circuit of figure 3 that can be built to program the 1702A, 5202AQ, etc. The programmer is a simplification of the Intel MP7-03 programmer and is designed to work with the 8080 program of table 3. Crowbar

Figure 3A:



Figures 3A and 3B: Computer controlled PROM programmer for stored charge PROMs. Unless otherwise noted, transistors are MPS-A06 or 2N3722 or equivalent. Pin 14 of ICs to 5 V, pin 7 to ground.

Figure 3B:

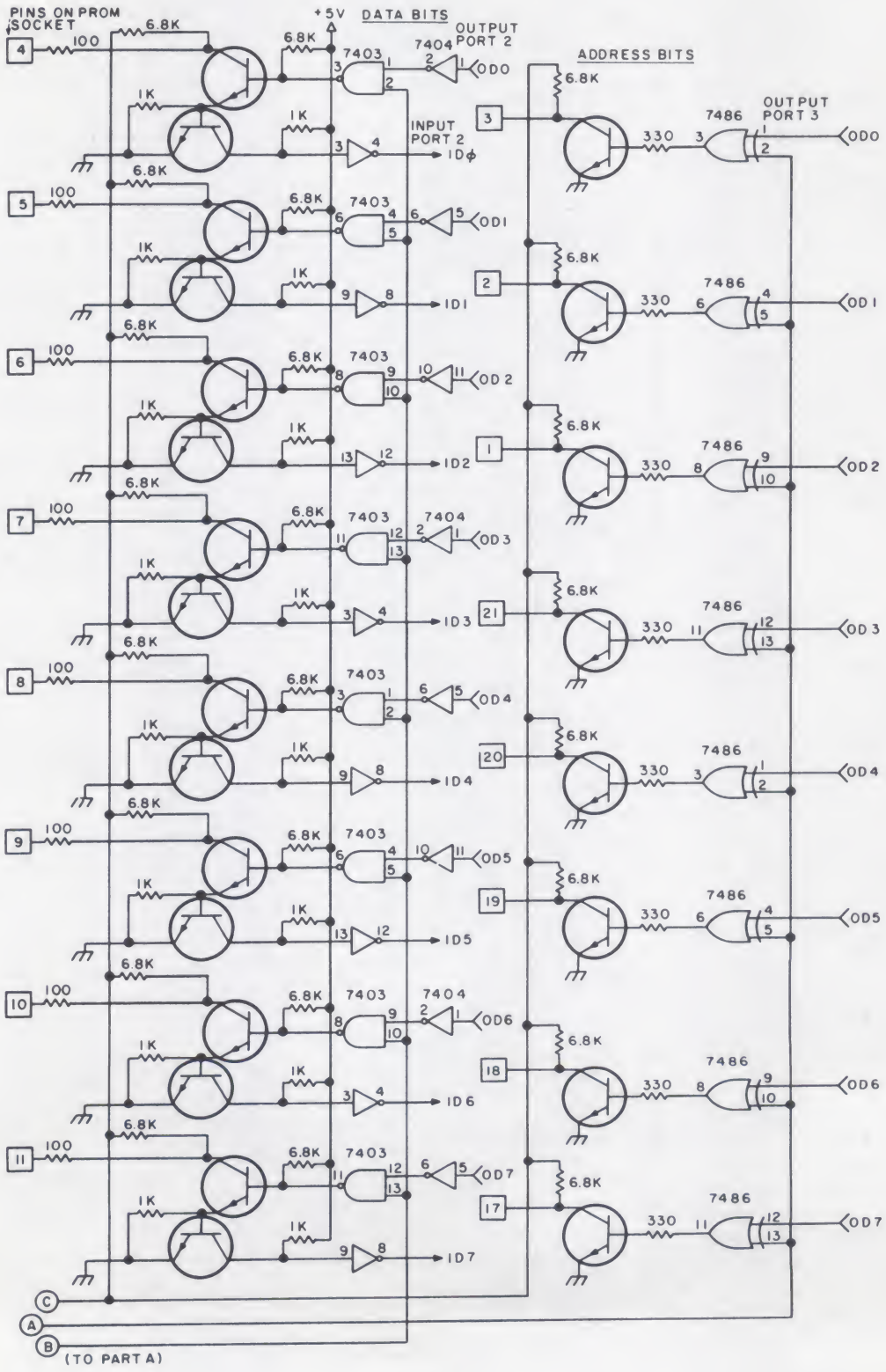
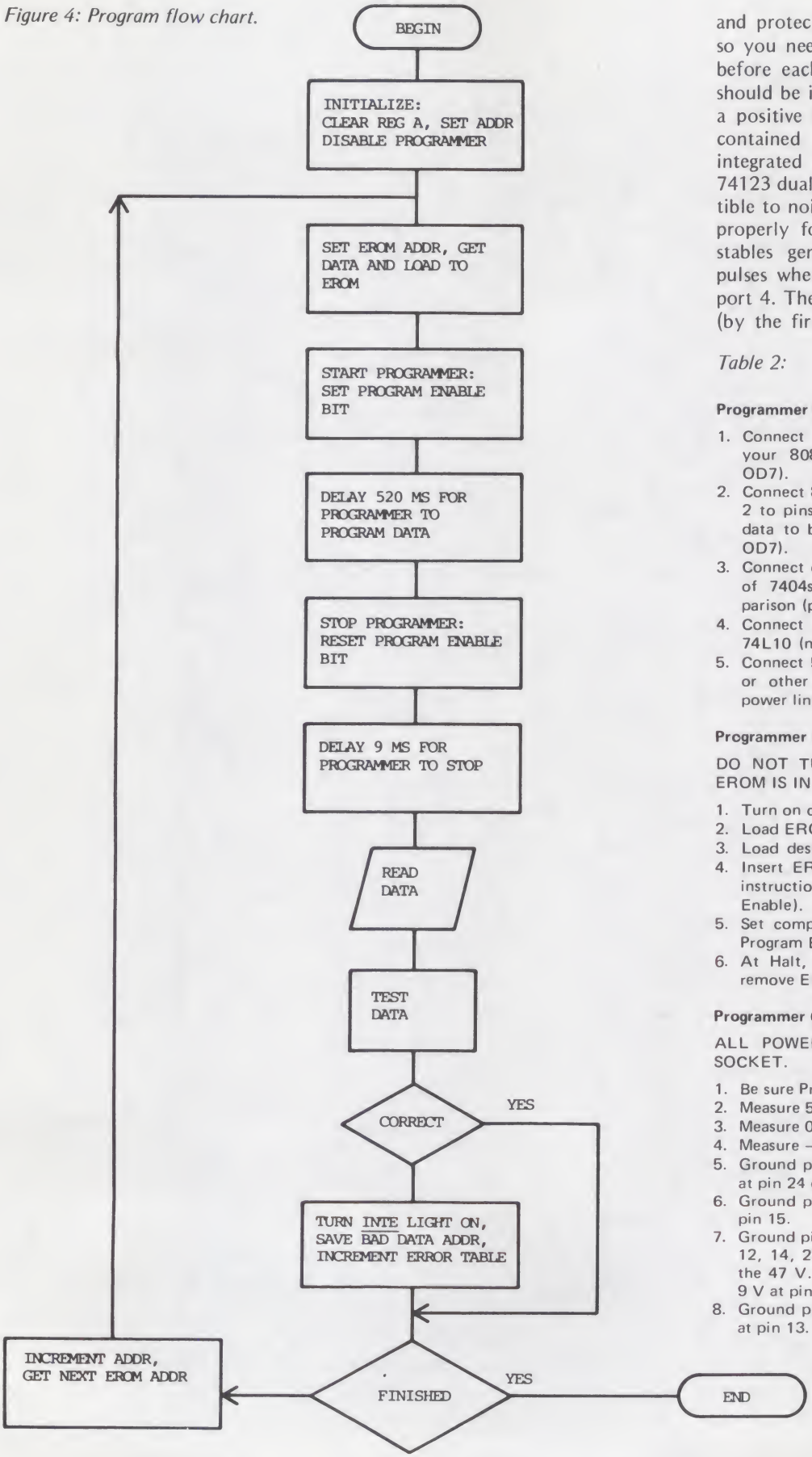

PINS ON PROM SOCKET
+5V
DATA BITS
OUTPUT PORT 2
ADDRESS BITS
OUTPUT PORT 3
INPUT PORT 2
6.8K
100
1K
330
7403
7404
7486
OD0
OD1
OD2
OD3
OD4
OD5
OD6
OD7
ID0
ID1
ID2
ID3
ID4
ID5
ID6
ID7
(TO PART A)

Figure 4: Program flow chart.



and protective features have been removed, so you need to check your circuit voltages before each use. Input data and addresses should be in positive logic (i.e., logical one is a positive level). The five monostables are contained in three 9602 dual one shot integrated circuits. Do not substitute the 74123 dual monostable which is very susceptible to noise and therefore may not operate properly for this application. These monostables generate the proper programming pulses when enabled by bit 2 of the output port 4. The pulses are repeated every 15 ms (by the first 9602) and the length of time

Table 2:

Programmer Connections

1. Connect 8 Address lines to output port 3 of your 8080 computer (port 3 lines OD0 to OD7).
2. Connect 8 Output Data lines from output port 2 to pins 1, 5, and 11 of the three 7404s for data to be programmed (port 2 lines OD0 to OD7).
3. Connect computer input port 2 to pins 4, 8, 12 of 7404s for reading EROM data for comparison (port 2 lines ID0 to ID7).
4. Connect bit 2 of output port 4 to pin 1 of 74L10 (near the PROGRAM ENABLE switch).
5. Connect 5 V and −9 V supplies from computer or other source and connect the 110 VAC power line.

Programmer Operation

DO NOT TURN POWER ON OR OFF WHILE EROM IS IN SOCKET.

1. Turn on computer and programmer.
2. Load EROM program at location 001/000.
3. Load desired EROM data at location 002/000.
4. Insert EROM into socket. Single step first five instructions of the program (to disable Program Enable).
5. Set computer at address 001/000, switch on Program Enable switch, and start computer.
6. At Halt, turn off Program Enable switch and remove EROM.

Programmer Calibration Test

ALL POWER ON, AND NO EROM IN THE SOCKET.

1. Be sure Program Enable switch is OFF.
2. Measure 5 V at pins 12, 13, 15, 22, and 23.
3. Measure 0 V at pin 14.
4. Measure −9 V at pins 16 and 24.
5. Ground pin 2 of the 7405 IC and measure 0 V at pin 24 of the EROM socket.
6. Ground pin 4 of 7405 IC and measure 58 V at pin 15.
7. Ground pin 6 of 7405 and measure 47 V at pins 12, 14, 22, and 23. Adjust pot on NE550 for the 47 V. Pin 13 should remain at 5 V. Measure 9 V at pin 16.
8. Ground pins 6 and 8 of 7405 and measure 47 V at pin 13.

each word is programmed is determined by the software. This will be 520 ms with the program given here, so about 35 pulses are generated per word. Note that all bits in a word are programmed at one time.

At first glance it would appear that the voltages generated by this programmer don't seem to agree with the voltages specified for programming the 1702A in the manufacturer's documentation. However, if you look at the voltages with respect to Vcc (pin 12), they do agree. Follow the programming instructions of table 2 exactly and *be sure* you don't turn power on or off while an EROM is in the socket. The monostables could be triggered and program unwanted bits in the EROM. Also, turn the Program Enable switch to off when inserting or removing EROMs. Table 2 gives a complete list of programmer connections and a calibration test.

The author will program the 1702A EROM for $5. All he needs is the PROM, an octal or hexadecimal listing of the desired program and $5.

A program for use in your 8080 type computer to program the 1702A (or equivalent) is shown in table 3 and the accompanying flow chart is shown in figure 4. This program is loaded into your computer starting at octal location 001/000, and the data to be written into the EROM is loaded starting at location 002/000. If any errors in programming are detected, the Interrupt Enable (INTE) light will light and the EROM addresses of the invalid words will be stored consecutively starting at octal location 000/000. The program runs for about 2.25 minutes and then halts. If you have an 8008 system, you can translate the operations and change the timing loops accordingly. For users of other machines, comments and a flow chart are provided to aid in translating the 8080 code for your machine. This EROM program can be entered and operated entirely from front panel switches or, for those with a teletype, punched tape, TV typewriter, etc., the program can be modified to suit. In any event, check that the timing loops are correct for your particular system before attempting to use the programmer to actually program EROMs.

Good luck with the PROM/EROMs for your computer.■

Table 3: 8080 program for EROM programmer.

```
001/000  257                          XRA   A       ;CLEAR REGISTER A
   /001  117                          MOV   C,A     ;SET ERROR TABLE ADDRESS
   /002  137                          MOV   E,A     ;SET FIRST EROM ADDRESS
   /003  323  004                     OUT   4       ;DISABLE PROGRAMMER
   /005  041  000  002                LXI   H,DATBK ;SET ADDRESS FOR DATA TABLE
   /010  323  003           START:    OUT   3       ;SET EROM ADDRESS
   /012  176                          MOV   A,M     ;GET DATA FROM TABLE
   /013  323  002                     OUT   2       ;LOAD DATA TO EROM
   /015  076  004                     MVI   A,004   ;SET BIT 2 = 1
   /017  323  004                     OUT   4       ;ENABLE PROGRAMMER
   /021  006  260                     MVI   B,260   ;SET DELAY COUNT
   /023  026  000                     MVI   D,000
   /025  025           DELAY1:       DCR   D       ;DELAY FOR 520 MSEC.*
   /026  302  025  001                JNZ   DELA1
   /031  005                          DCR   B
   /032  302  025  001                JNZ   DELAY1
   /035  257                          XRA   A       ;CLEAR REGISTER A
   /036  323  004                     OUT   4       ;DISABLE PROGRAMMER
   /040  006  003                     MVI   B,003   ;SET DELAY COUNT
   /042  025           DELAY2:       DCR   D       ;DELAY 9 MSEC.*
   /043  302  042  001                JNZ   DELA2
   /046  005                          DCR   B
   /047  302  042  001                JNZ   DELA2
   /052  333  002                     IN    2       ;READ PROGRAMMED DATA
   /054  276                          CMP   M       ;CHECK DATA – IS IT CORRECT?
   /055  312  064  001                JZ    ALLOK   ;JUMP IF OK
   /060  373                          EI            ;BAD DATA – TURN INTE LIGHT ON
   /061  173                          MOV   A,E     ;GET DATA ADDRESS FOR DATA
   /062  002                          STAX  B       ;STORE BAD DATA ADDRESS
   /063  014                          INR   C       ;INCREMENT ERROR TABLE ADDRESS
   /064  076  377           ALLOK:    MVI   A,377   ;CHECK IF DONE PROGRAMMING THIS CHIP
   /066  273                          CMP   E       ;AT LAST ADDRESS?
   /067  312  100  001                JZ    STOP    ;HALT IF DONE
   /072  034                          INR   E       ;INCREMENT EROM ADDRESS
   /073  043                          INX   H       ;SET NEXT DATA ADDRESS FOR DATA TABLE
   /074  173                          MOV   A,E     ;GET NEXT EROM ADDRESS
   /075  303  010  001                JMP   START   ;CONTINUE – PROGRAM NEXT WORD
   /100  166           STOP:         HLT           ;HALT – ALL DONE

002/000  (EROM DATA)   DATBK:  BLK   256.
```

*Note: Time delay loops are shown for an ALTAIR 8800 with a clock frequency of 2 MHz. Total loop time = 11.5 usec for an apparent frequency of 1.3 MHz.

Getting Inputs from Joysticks and Slide Pots

Carl Helmers

Have you ever wondered how to get inputs from joysticks and slide pots for interactive game control purposes? A joystick is a two dimensional potentiometer control of the kind often seen in model aircraft radio control rigs. A slide pot (or conventional pot) is just a one dimensional version of the same concept of interactive control. To use the information obtained from such a potentiometer in the computer it must be converted into two binary integers. An inexpensive oscillator, two counters, a four bit output latch and one NAND gate section are needed in addition to a standard 8 bit bus IO interface and a simple set of software routines. The ideas in this article can be adapted to any computer, although sample subroutines are shown for the 8080 and the 6800 microprocessors.

The Method

The problem to be solved is turning a mechanical signal into the corresponding value of a digital word used by the program. The mechanical signal is the position of the joystick, slide pot or conventional shafted potentiometer. The electronics can immediately measure this position by measuring the resistance of a potentiometer. The problem thus evolves into looking for a way to convert a resistance into a binary measurement.

There are many different ways to accomplish this task. The particular method chosen here is to convert the measurement into a frequency through an oscillator. The frequency is measured under direct control of the computer program using an 8 bit counter with the CPU clock as a time base. To

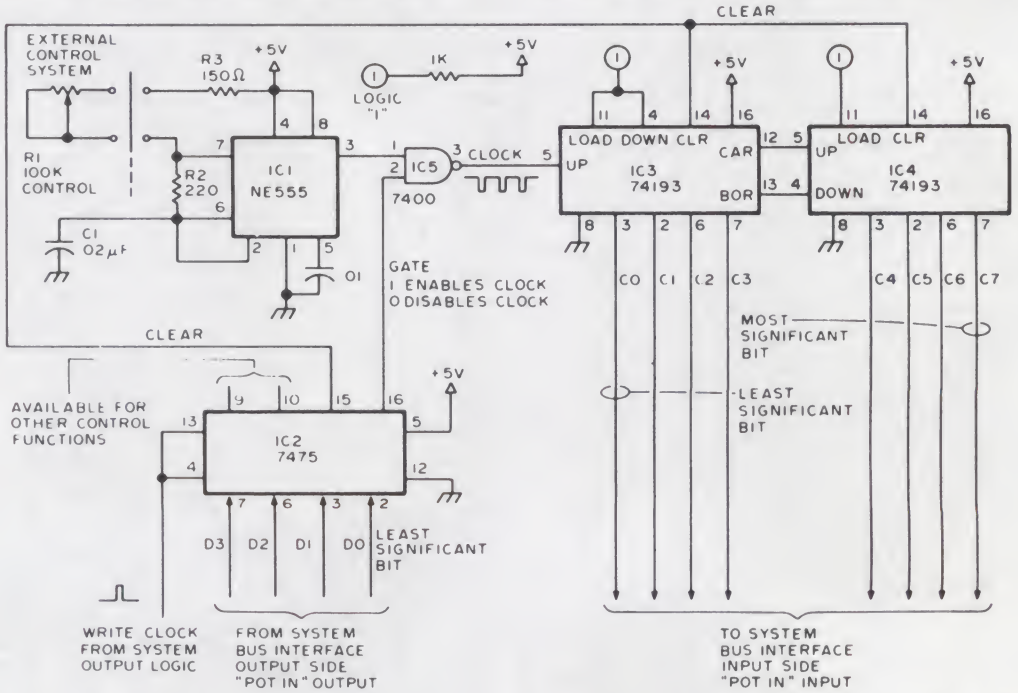


Figure 1: The hardware of an input device for interactive manual controls consists of an oscillator, two counter circuits, an output latch and a NAND gate section. This hardware must be driven by a suitable program.

accomplish this conversion, the processor must execute a simple five step process:

1: Clear the counter,
2: Turn on the counter,
3: Wait 2 milliseconds,
4: Turn off the counter,
5: Read the count.

The result is the number of cycles during a two millisecond period. For the circuit of figure 1 this number will range from 1 to about 240. The relationship of frequency to control position depends primarily upon the resistance to frequency conversion function of the oscillator and the linearity of the potentiometer. The accuracy of the conversion will not necessarily be high – but the intended application as an interactive control input more than makes up for that failing. For a game control application such as Space War or Pong, accuracy of the conversion function is not a paramount concern, so long as it is reasonably repeatable within limits of human perception.

The Hardware

The resistance to frequency conversion is performed by IC1, an NE555 timer integrated circuit which comes in an 8 pin mini DIP package (see figure 1). The timer is set up as an oscillator with frequency (f) of oscillation determined by R1, R2, R3 and C1:

f = 1.44/((R1 + R3 + 2*R2)*C1), where

f is the frequency measured in kHz,
R1, R2, and R3 are all measured in kO, and
C1 is measured in uF.

The resistances and capacitance used are chosen so that the frequency will range from about 0.75 kHz to about 122 kHz as the control R1 is varied from 100 kO to 0 kO.

A linear potentiometer is not recommended because of the relationship between changes in frequency and resistance as illustrated in table 1. A linear potentiometer provides for resistance changes proportional to the position of the shaft. The relationship between resistance and frequency, on the other hand, is not linear. Therefore, the relationship between position of the shaft and frequency is not linear if a linear potentiometer is used. To solve this problem to some extent, a logarithmic potentiometer may be used; it is often called an "audio-taper" because of the relationship between the position of the shaft and the sensitivity of the ear. This potentiometer will not perfectly compensate for the nonlinearity of the resistance and frequency relationship. However, it is quite an improvement.

Table 1: Frequency (rounded to nearest 0.5 kHz) versus Resistance of R1. (R2 = 440 Ohms, R1 = 150 Ohms, C1 = 0.02 uF)

kilohm	kilohertz
100	0.5
90	1.0
80	1.0
70	1.0
60	1.0
50	1.5
40	2.0
30	2.5
20	3.5
10	7.0
9	7.5
8	8.5
7	9.5
6	11.0
5	13.0
4	15.5
3	20.0
2	27.5
1	45.5
0.5	66.0
0	122.0

A control register is provided by the 7475 circuit (IC2). This circuit is connected to the output side of the IO port (POTIN). The two lines D0 and D1 of the output side of POTIN are used to control the circuit. The GATE line is used to control whether or not the oscillator output is allowed to reach the counter. A 1 bit output enables counting. The CLEAR line is used to reset the counters prior to beginning a measurement. This line is connected directly to the asynchronous clear inputs of the counter circuits IC3 and IC4. A 1 bit output clears the counters.

The counters used to measure the frequency are 74193 circuits (IC3 and IC4) which are wired for 8 bits. Following a clear operation, a 2 millisecond GATE signal will result in a measurement.

Not shown in figure 1 is the specific bus interface circuit required to connect this peripheral to your computer. The software of this article assumes only that hardware of your system can decode the required output operation to the 7475 (IC2), and can read the 8 bits coming out of IC3 and IC4.

The Software

Table 2 presents a subroutine called POTREAD written for the Motorola 6800 processor design, and table 3 performs the equivalent program on an Intel 8080. Both listings are done in a symbolic assembly language format with comments to explain the operations. In both listings assumptions are made about the IO operations involved. For the 6800, the POTREAD procedure assumes that the memory address space location POTIN is implemented as the inter-

Table 2: Symbolic assembly code of POTREAD implemented for a 6800 instruction set. This procedure assumes that the potentiometer input device of figure 1 is located at POTIN in the memory address space of the 6800 computer. It also assumes that ALPHA is the memory location which is to receive the latest input, and that a subroutine MILLI exists which implements a 1 millisecond wait.

```
POTREAD  LDAA   #2      binary '00000010' is the clear command
         STAA   POTIN       which is sent to the device register;
         DECA           binary '00000001' is the count enable command
         STAA   POTIN       which is sent out to start measurement;
         JSR    MILLI   call on MILLI for a one millisecond wait;
         JSR    MILLI   call MILLI to wait once more;
         CLR    POTIN   turn off the counter with binary '00000000';
         LDAA   POTIN   read the count via input side of interface;
         STAA   ALPHA   save it in ALPHA and
         RTS                return to the caller;
```

Table 3: Symbolic assembly code of POTREAD implemented for an 8080 instruction set. This procedure assumes that the potentiometer input device of figure 1 is located at a parallel interface decoded for device address POTIN. It also assumes that ALPHA is a memory location which is to receive the latest input and that there exists a subroutine called MILLI which implements a 1 millisecond wait.

```
POTREAD  MVIA   2       binary '00000010' is the clear command
         OUT    POTIN      which is sent to the device register;
         DCRA           binary '00000001' is the count enable command
         OUT    POTIN      which is sent out to start measurement;
         CALL   MILLI   call MILLI for a one millisecond wait;
         CALL   MILLI   call MILLI to wait 2 ms total;
         MVIA   0       binary '00000000' is the stop command
         OUT    POTIN      which is sent out to end measurements;
         IN     POTIN   read the count via input side of interface;
         LXIH   ALPHA   set up address of ALPHA;
         MOVM   A       save count in ALPHA;
         RET            return to caller;
```

Table 4: The MILLI procedure specified in symbolic assembly language for the 6800 processor. The timing calculation is shown in the left hand columns; the JSR which calls MILLI from the main program is shown for purposes of the timing calculation. When the return instruction (RTS) is completed, exactly 1.000 ms will have elapsed between the completion of the instruction preceding the JSR and the beginning of the instruction following the JSR, assuming that the CPU has a 1.000 MHz crystal controlled oscillator.

Note: Not all 6800 systems have 1.0 MHz CPU clocks. To adjust timing pick a new constant instead of 162, and possibly balance with NOP or nullbranch instructions.

Time (μs≡cycle)	Time Total	Label	Op.	Operand	Commentary
9	9		JSR	MILLI	main program calls MILLI;
4	13	MILLI	PSHA		save A register in stack;
2	15		LDAA	#162	decimal 162 loop count;
162 x 2	339	MILOOP	DECA		count decremented and tested
162 x 4	987		BNE	MILOOP	to keep loop going on;
4	991		BRA	NEXT	time wasting null branch;
4	995	NEXT	PULA		restore A from stack;
5	1000		RTS		back to the caller;

face to the peripheral of figure 1. For the 8080, the procedure assumes that the IO device with a symbolic code POTIN is implemented as the interface. It is an interesting exercise which is left to readers to perform, comparing the number of bytes required and the execution time required on the two machines, assuming comparable operation near the highest possible clock frequency. The result will be found to be similar.

The subroutine MILLI is intended to be a 1000 microsecond delay implemented either by reference to a hardware realtime clock, or as a timing loop with constants adjusted to the clock frequency of your computer. The concepts of creative time wasting described by Jim Hogenson in his article "Can Your Computer Tell Time" (in the December issue of BYTE) can be applied to the problem of writing such a program for your particular computer. An example for a 6800 is shown in table 4.

What's Next?

This article has illustrated a simple analog to digital conversion input device which can be implemented inexpensively. The uses to which you put this idea are up to your own imagination. The electronic music person can use this kind of input to control parameters like tempo and timbre variations. The model railroad buff could use this conversion to input engine speed information. The amateur radio operator could use such an input as one way to control the speed of machine generated Morse code transmission. The space war freak can use this type of a device for the input of heading and velocity information taken off a joystick. This is by no means an exhaustive catalog of applications which can take advantage of a simple conversion of this kind.■

Logic Probes - Hardware Bug Chasers

by
Alex. F. Burr
Physics Dept. Box 3D
New Mexico State University
Las Cruces NM 88003

While an oscilloscope or voltmeter can be used with digital circuits, a logic probe is much less expensive if built from an appropriate kit.

Digital logic, whether used in an 8080 microprocessor or as the TTL chips that can be used to make a processor, is, at least in theory, clean and simple because only two states are possible. Any point in even the most complicated circuit is either HIGH or LOW. However this very simplicity encourages the design of large and complicated circuits. While the chance of anything going wrong at any one point is small, the accumulated chances of many points means that sooner or later the experimenter is going to have to hunt for sources of trouble.

In the case of analog circuits, when trouble develops, you get out the oscilloscope or voltmeter and start looking for places which have waveforms or voltages not meeting the specifications. These instruments can be used to troubleshoot digital circuits too. The oscilloscope is particularly useful if you have timing problems, but usually they give you too much information and may just confuse the issue. The voltmeter may tell you that the voltage on pin 8 is 3.9 but, because most IC failures show up as a node stuck either HIGH or LOW, really all you need to know is that on pin 8 there is a HIGH. That single bit (literally) of information can be obtained with an instrument a lot smaller and less complicated than a voltmeter.

That instrument is the *logic probe*. In its simplest form it is just a state indicator with a sharp point. When the point is placed on any pin of an IC, the probe will indicate whether a LOW or a HIGH is present at that point. And with digital logic that is usually all the information you need.

Logic probes can detect a surprising number of different defective conditions. Fig. 1 illustrates some of the uses to which a probe can be put. Of course, just as voltmeters come with a variety of capabilities and prices, so do logic probes.

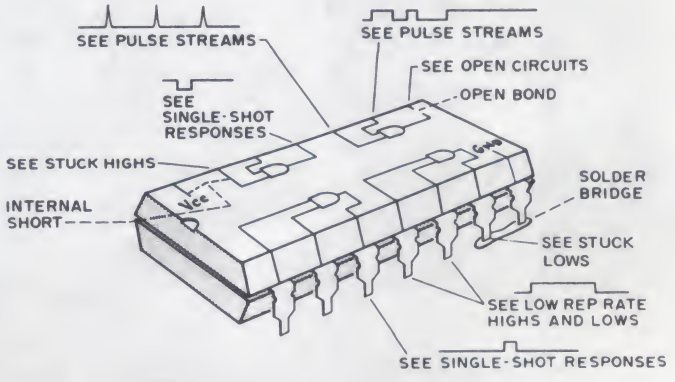
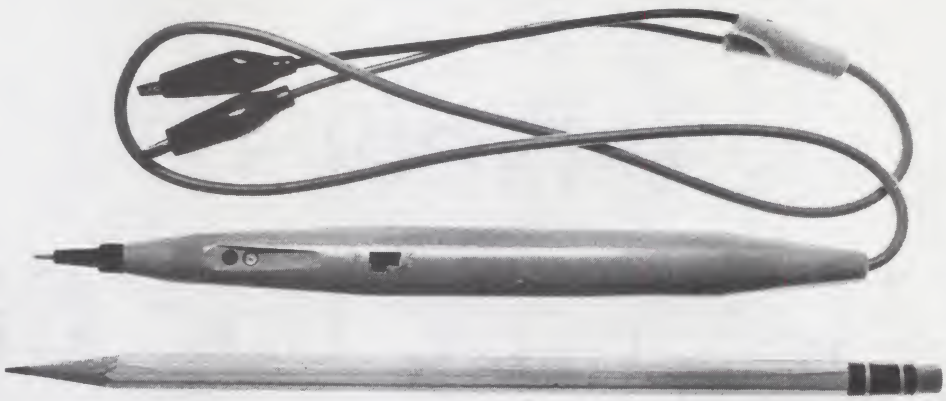


Fig. 1. Some of the uses of logic probes and the malfunctions which they can detect.

Commercial Logic Probes

One of the first developed was the Hewlett Packard 10525T logic probe. It is a marvel of compactness and versatility, all carefully human-engineered. Basically it consists of a white light which goes out when the probe is placed on a LOW and comes on when the probe is placed on a HIGH.

Simple -- yes indeed; but it does much more. What if the point tested is open circuited, or the level is just plain bad, neither HIGH or LOW? Then the light glows at half intensity. What if a pulse comes along that is too short to excite the indicator light? Then a pulse stretcher takes over. Pulses with a width of between 10 ms and 0.05 seconds are stretched to 0.05 seconds in length. What if the pulses come so fast that the eye cannot distinguish one from the next? All pulse streams with a repetition rate between 10 Hz and 50 MHz cause the lamp to blink at a 10 Hz rate. All this capability is enclosed in a probe about six inches long and one-half inch in diameter. The light is placed near the tip in such a way that it can be seen no matter how the probe is rotated. Thus you can easily see both the point of the probe and the indicator at the same time. Power is supplied to the probe by a well protected single cord which is attached to a source of 5 V dc at 60 mA.



The E & L Instruments logic probe is compact, with the two indicator LEDs visible toward the left in this photo.

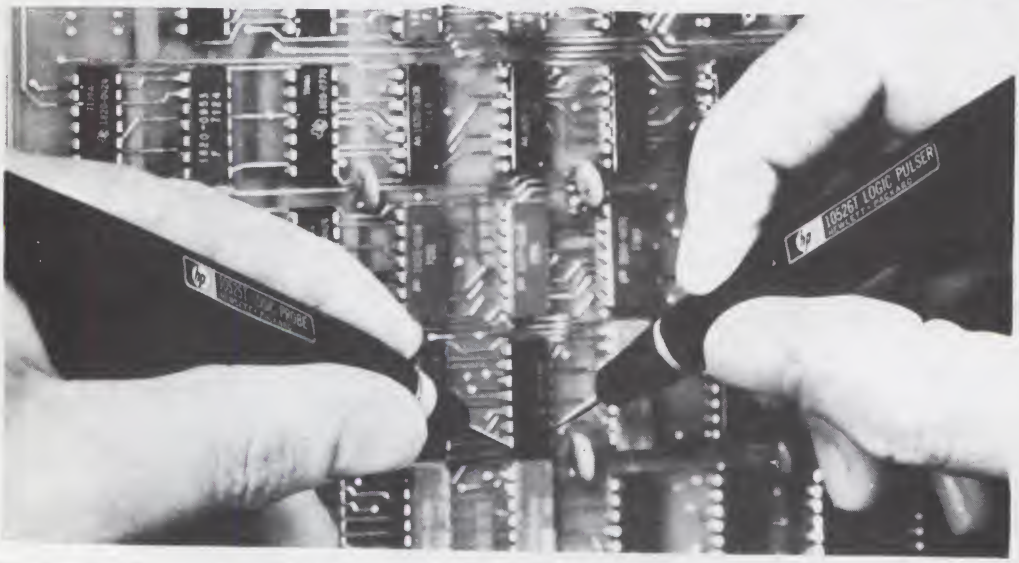
The input impedance is greater than 25k Ohms in both the HIGH and LOW state (less than one low power TTL load). The input is well protected against operator error. The probe will stand ±70 volts continuously and ±200 volts intermittently as well as 120 V ac for 30 seconds. The power input is internally protected from +7 to −15 V dc as well as power lead reversal. The only catch is the price, which even with a recent reduction is $65.

There are, however, other less expensive probes. Two of these are distributed by E and L Laboratories. Their model 340 is a logic probe and pulser combined into one instrument. The model 320 is a logic probe only, designed to give maximum information about the state of the node being tested. Both probes are well constructed, a little over 6 1/2 inches in length and half an inch in diameter. Both come with two different probe tips and handy carrying cases.

The model 340 has two LED indicator lights. In operation the two leads from the probe are connected to the 5 V dc supply and the probe tip applied to the IC lead to be tested. If that node is HIGH, the red LED lights up brightly. If that node is LOW, neither LED is lit. The green LED is used to detect pulses which do not last long enough to give a useful indication on the red LED. It lights for 0.1 seconds (just long enough to see) whenever a single pulse wider than 50 nanoseconds is applied to the probe tip. In the probe tested for this article, when a voltage increasing from zero was applied to the tip, the red LED lit when the voltage was greater than 1.5 V; just what the specifications called for.

The model 320 is a little more versatile as an indicating instrument, but it lacks the ability to generate pulses that the model 340 has. It too has two LED indicators, one red and one green. In operation, the two power leads are connected to the 5 V dc power supply and the tip to the node under test. The specifications say that if the voltage at the node is less than 0.7 volts the green LED will be lit. If the voltage is greater than 2.4 volts, the red LED will be lit. In practice the specifications are closely followed. The LEDs may glow dimly at voltages just a few tenths higher or lower than the specified voltages. But the dividing line between lit and not lit states is remarkably sharp.

A special feature of this probe is the pulse storage



The Hewlett Packard 10525T logic probe and 10526T pulser.

capability brought into play by a small switch near the tip. When the pulse storage feature is on, a short pulse (either HIGH or LOW) is stretched so that it turns on the appropriate LED to full brightness even if it is as short as 50 nanoseconds. Square and sine waves appearing at a tested node will cause both LEDs to have equal brightness.

The main difficulty noted with this probe is with the green LED. It is somewhat dimmer than the red LED and the lens diffuses the spot of light generated less well so that in bright room light it is sometimes hard to tell whether or not the green LED is lit. This fact would make the determination of the duty cycle of a chain of pulses by a brightness comparison between the LEDs much more difficult than the instruction booklet suggests.

Even the E and L Laboratories probes are expensive ($35 and $25); although they are more convenient than, and certainly in the same price range as, a good voltmeter. Nothing, however, can beat the cost effectiveness of two probe kits which have been fairly widely advertised.

"Nodes" are places in a circuit — such as the pin of an IC — where you might want to test the logic level using the probe.

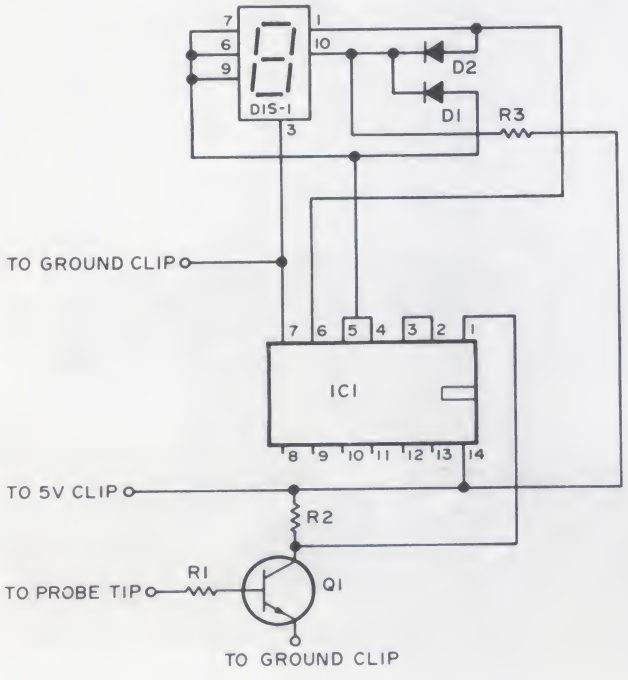


Fig. 2. Circuit diagram for the James logic probe kit.

Logic Probe Kits

One of these kits is manufactured by Chesapeake Digital Devices. This kit allows one to easily construct a probe which uses red, green and yellow LEDs to signal the presence of logic levels in digital circuits.

The kit goes together in a very short time with the aid of very complete assembly instructions. The whole probe fits into a well constructed case, a little over six inches long and slightly less than one inch in diameter. There are only three resistors, three LEDs, one transistor, and a 74S00 integrated circuit to solder onto the clearly marked printed circuit board.

In operation the green LED is brightly lit on a LOW, the red LED is brightly lit on a logic HIGH, while the yellow LED lights on an open circuit or a level between a true HIGH or LOW. A slow pulsing condition will be indicated by alternate flashing of the red and green LEDs. A fast pulsing condition will be indicated by the simultaneous activation of the red and green LEDs. The dividing line between these last two conditions is about 20 Hz, depending on the eye of the user.

The biggest difficulty with the kit was the circuit board. The copper leads had not been tinned and were oxidized, making them a bit difficult to solder; especially if the builder was concerned that he not use so much heat for so long as to damage the components. The clear plastic tube into which the circuit board with its LEDs slide did crack on assembly and the green LED was open but these difficulties were easily remedied and the result was a handy logic probe at a price significantly less than any assembled probe.

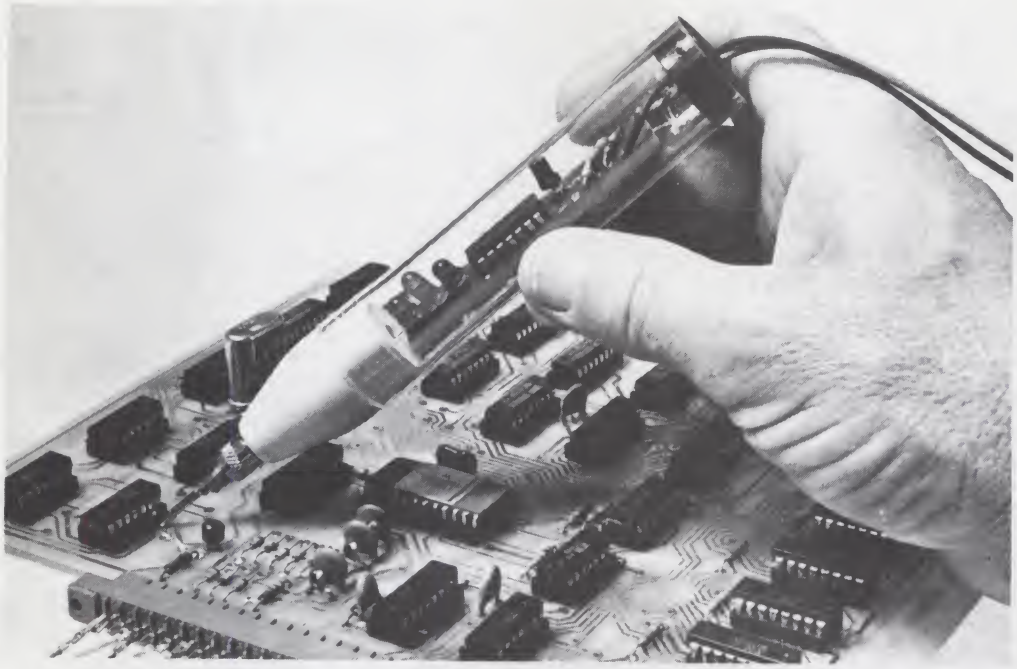
A Unique Probe

A particularly inexpensive kit is the one sold by James Electronics for $9.95 including postage and case. It is unique in that it uses a MAN 3 seven segment readout which gives a 1 for a HIGH a 0 for a LOW and a P for a pulse train — all this in a compact package measuring five inches long and one inch in diameter.

The circuit diagram for this intriguing probe is given in Fig. 2. The 2N2222 input transistor drives the chip, IC1, which in turn causes the appropriate segments of the MAN 3 to light. The chip was custom made for James Electronics by National Semiconductor and contains a proprietary circuit which was laid down by a $500 master mask.

The kit comes in a very impressive package which was carefully designed to protect the contents from rough handling by the U.S. Postal Service. The parts, which include the case and a custom glass epoxy printed circuit board, are of high quality and are not your usual cheap imports. Because most of the parts are in the 14-pin chip which is the heart of the probe, the kit goes together quickly and easily for the experienced builder (about one hour to solder all the parts to the board). There are no explicit devices for overload or reverse voltage protection. The probe draws 65 mA from any convenient 5 V point on the circuit under test.

The inexperienced builder is going to have trouble because the complete assembly instructions say, "Assemble the Logic Probe according to the schematic diagram and board layout shown below." The end. One has to have pretty sharp eyes



A kit logic probe shown in action testing a printed circuit board.

A logic clip is like 16 binary voltmeters in a neat little package.

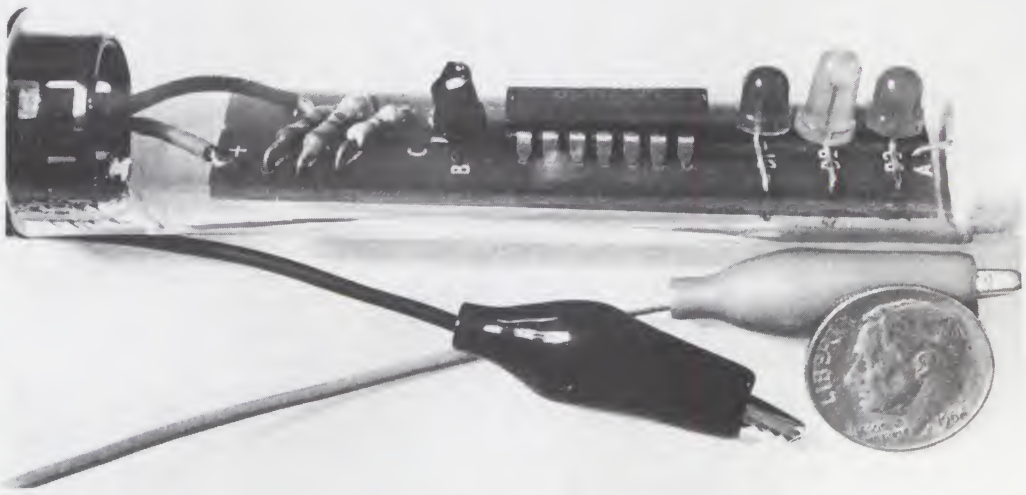
to orientate the IC, transistor and readout correctly. Even then you might miss the two jumpers that go on the circuit board. The circuit board also could have been laid out more efficiently so that the drastic bending of the MAN 3 leads would have been avoided.

There is one serious defect. It is more serious from the theoretical than the practical point of view. That defect concerns the input level at which the indicator switches from 0 to 1. That level is 0.65 volts; but the specifications for TTL logic say that the maximum voltage that the logic is guaranteed to interpret as LOW is 0.8 volts. Thus the probe would indicate a HIGH on a node which the logic would interpret as a LOW. This defect is of lesser practical importance because it is the unusual LOW which will have a voltage greater than 0.6 V. Indeed the usual gate input is only a very few tenths of a volt above ground at the most when it is LOW. Nevertheless it is a bit disconcerting to have the probe give a wrong reading even if it does so only under unusual circumstances. After all, it is under unusual circumstances that the probe is most often used.

Logic Clips?

The probes which have been discussed so far all investigate one pin of the IC at a time. There are some instruments which will do much more. These are called logic clips and are not really probes but will give you the same type of information. They are extremely handy service and design tools. They clip onto TTL DIP ICs and instantly display the logic states of all 14 or 16 pins. Each of the clip's 16 LEDs independently follows level changes at its associated pin: A lighted diode corresponds to a HIGH.

The logic clip's real value is in its ease of use. It has no controls to set, needs no power connections, and requires practically no explanation as to how it is used. The clip has its own gating logic for locating the ground and +5 volts Vcc pins and the buffered inputs reduce circuit loading. Simply attaching the clip to a TTL dual inline package makes all



A detail of the Chesapeake Digital Devices logic probe board. The three LEDs are at the right in this picture, with the 74S00 IC in the center.

the logic states visible at a glance. The clip is, in effect, 16 binary voltmeters. When used with some means of pulsing a complicated circuit slowly, sequential logic states like shift registers come alive — each state change is immediately visible.

The most popular clips are made by Hewlett Packard and Circuit Specialties. Unfortunately they have one big drawback — price. They cost from $75 to $85 each and will not be discussed further here.

Summary

Table 1 summarizes all the information that has been given here and presents some new facts about each of the logic probes discussed. By scanning this table you ought to be able to determine which probes have the features you need and the ones you can afford. The following comments are based on personal experience with each of these probes, but that experience has been rather limited.

The Hewlett Packard probe is obviously the best. It should be; it certainly costs significantly more. It will work under a wide range of conditions and it is carefully made. For the extra money you get wide frequency range, tight specifications, and vastly superior handling of pulse trains. The construction is first class and includes such extras as a compact BNC plug on the power cable (which, of course, is not so good if your breadboarding system does not have a BNC jack to supply that power).

The E and L probes (340 and 320) are imported from Japan. They are very well constructed and have the little extras like plastic carrying cases and different probe tips that the better Japanese manufacturers like to include with their products. The 320 is a better logic probe than the 340. It is less expensive and it handles pulse trains and logic levels in a better and more revealing way. Of course, it does not have the pulse generating capabilities of the 340.

The professional logic designer will want to get one of these three probes. They may be a bit expensive for the serious hobbyist. In that case one of the two kits would be satisfactory.

Both kits went together easily and rapidly. The CDD kit is much more revealing about the state of the logic under test and has superior assembly instructions. The James kit has better quality parts and is cheaper.

In any case the serious worker in digital logic and computers, whether a professional or a serious hobbyist, will find one of these probes a valued addition to his collection of test equipment. ■

Table 1. Characteristics of logic probes.

Probe	HP 10525T[1]	340[2]	320[2]	CDD[3]	James[4]
Operating Voltage	5 ± 10% V	5 ± 10% V	5 ± 10% V	5 ± 10% V	5 V
Current	60 mA	100 mA	80 mA	40 mA	65 mA
Frequency Response	50 MHz[5]	12 MHz	12 MHz		
Input Impedance	>25 kΩ	50 kΩ	100-600 kΩ		
Min. pulse width	10 ms[6]	50 ms	50 ms		See[11]
Levels OPEN	half intensity	no lights[7]	no lights[9]	yellow[10]	
HIGH	on >2±0.2 V	red >1.5 V	red >2.4 V	red >2.5 V	1>0.7 V
LOW	off $<0.8^{+0.2}_{-0.4}$ V	no lights	green <0.7 V	green <1 V	0<0.7 V
Size	6" x 0.5" dia.	6.6" x 0.6" dia.	6.6" x 0.6" dia.	6" x 1" dia.	5 x 1" dia.
Overvoltage protection	excellent	reasonable	reasonable	none	none
Price	$65	$35[8]	$25	$15[12]	$10[12]

Notes:

[1]Hewlett Packard, Palo Alto CA 94304.

[2]E and L Instruments Inc., 61 First St., Derby CT 06418.

[3]Chesapeake Digital Devices, Inc., Box 341, Havre de Grace MD 21078.

[4]James Electronics, Box 822, Belmont CA 94002.

[5]Pulse trains faster than 10 Hz cause the lamp to flash at a 10 Hz rate.

[6]Pulses between 10 ms and 0.05 seconds are stretched to 0.05 seconds.

[7]Short pulses indicated by green LED.

[8]Single pulse generator contained in probe.

[9]Switchable pulse stretcher for short pulses.

[10]Yellow LED is also lit if voltage is between HIGH and LOW.

[11]Indicator reads P for pulse trains > 20 Hz.

[12]Kit price.

Controlling External Devices With Hobbyist Computers

Robert J Bosen
Box 93
Magna UT 84044

There is an almost infinite variety of uses to which a hobbyist computer system may be applied besides calculating or data processing, and many of these can bring a great deal of satisfaction to the proud owner. For example, hobbyist microcomputers are invariably advertised with a long list of possible applications such as home security systems, light controllers, process controllers, or automated drink mixers. I have personally had several opportunities to use my computer in a variety of related ways, including controlling stage lighting and sound effects for a large bicentennial celebration, and automating a spook alley. These and other applications inspired me to build the module described here to interface my computer with virtually any electrical or electronic device. If you build this interface as I did, you'll be able to control up to 16 channels of electrical outlets or switches of any kind, and only your imagination will limit the applications.

The basic principle behind any computer interface is to change computer compatible signals to device compatible power levels, and this interface accomplishes that goal with a great deal of flexibility, allowing the user to hook up virtually any type of

Photo 1: The author's computer setup includes the two CRT terminals shown on the table, plus a rack cabinet presently containing his central processor.

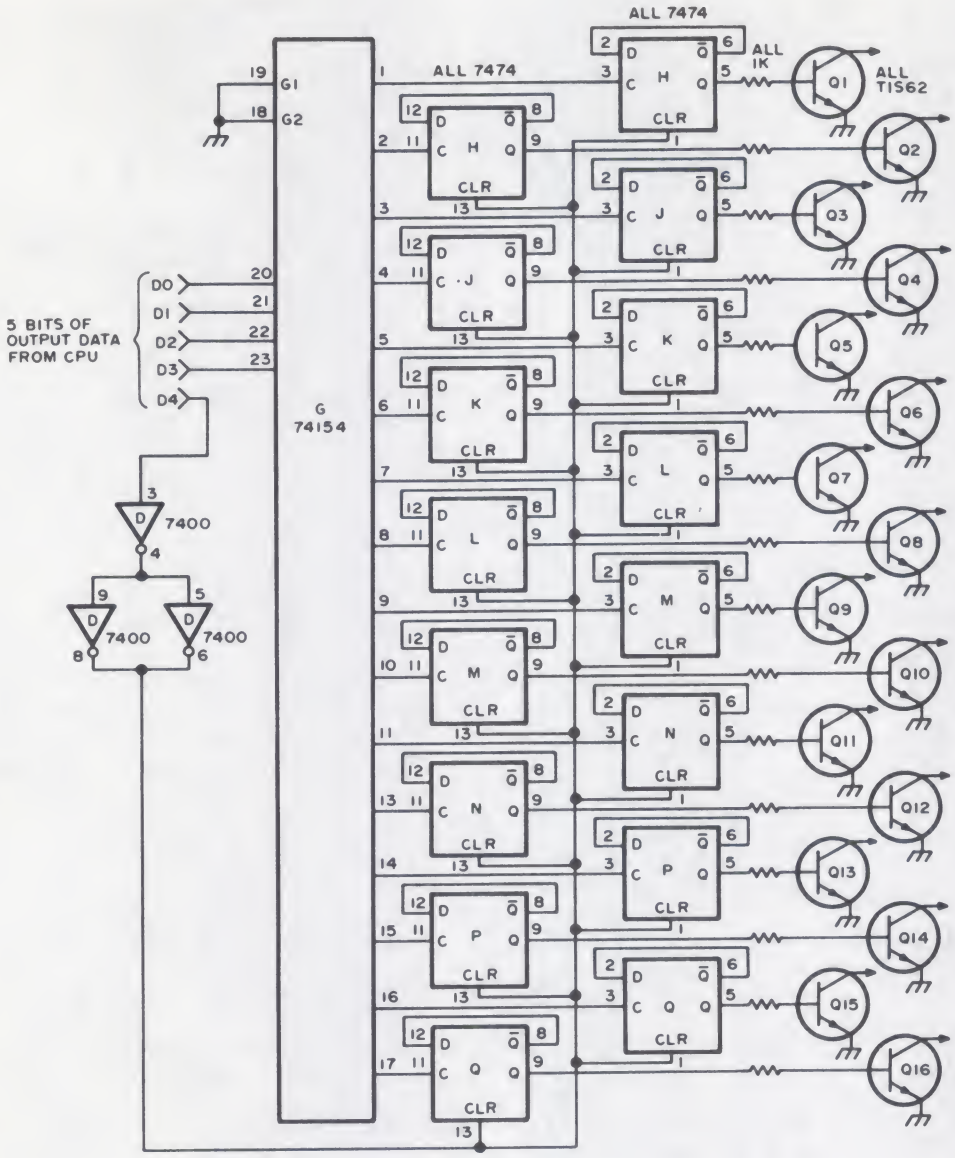


Figure 1: This is all you'll need to build if you already have a parallel output port you can use to control the interface card. If not, lines D0 through D4 should be joined with the corresponding points in figure 2. Transistors Q1 through Q16 can be any economical NPN with reasonable Beta. Due to varying configurations (you may not want to build up all 16 channels or use different transistors), I suggest the card be wirewrapped.

transistor, relay, or small electrical device to its open collector outputs. I used 16 surplus relays and wired them to 16 AC outlets and 16 sets of "five-way" binding posts. But this is by no means the only way to utilize the 16 output channels provided. All in all, the system described allows the programmer a great deal of flexibility over what he will control and how he will do it.

This interface may be used with virtually any 8 bit computer, and could be modified to work with a 4 bit machine as well. The circuit consists of four parts: A parallel output port, a 16 channel demultiplexer, a 16 bit memory, and 16 single transistor driver amplifiers. It can be built on a single small circuit board and total cost for all the solid state parts will be under $35 if a little shopping around is done. If you already have a spare parallel output port you can dedicate to this purpose, you can save about half of that cost.

Here's how it works: A byte of data is sent out of the computer to the parallel output port where it is latched. The four low order bits are applied to the four inputs of

the 74154 demultiplexer which selects one of 16 output pins and pulls it low. If, for example, the four bits are 0000, the demultiplexer will select channel zero and pin 1 will go low. There are 16 possible combinations of data that may be received, and for each of these combinations one of the pins of the 74154 will go low. Each of the 16 outputs of the demultiplexer then goes to a D flip flop which it toggles. Since we are trying to exercise control over 16 channels continuously, but the 74154 can only process one channel at a time, these D flip flops are needed to store the status of all inactive channels. Toggling the flip flops causes them to reverse their state and alternately turn on or off the transistors they drive each time a particular channel is selected. The fifth bit of the data byte is buffered (IC D) and then runs to the reset inputs of all 16 D flip flops, providing a reset signal to turn all the channels off simultaneously. (The three high order bits are unused.)

Hardware. The circuit provides 16 transistors in an open collector configuration, which may be viewed as open switches when off, and as switches shorted to ground when on. Each transistor can handle about 30 V and 30 mA. These may be used to control bigger transistors, or relay coils may be energized through them, or small electronic devices (sirens, light bulbs, etc.) may be powered directly with them by placing a voltage source in series with the device and the transistor. This is shown in several variations in figure 3. A word of caution is in order here if inductive loads such as relay coils are used: The collapsing magnetic field of the relay coil as it is turned off can generate large voltage spikes which may damage the transistors. Relay coils (see figure 3a) should therefore be protected with shunt diodes to short out these spikes when they approach dangerous levels. Relays may also oscillate at high frequencies if selected frequently in a program, so small capacitors may be necessary across the windings to short these oscillations to ground. From my own experience I found about half the surplus relays I tried exhibited this problem, but tinkering with various small capacitors clears it up.

Software. The software must provide data bytes containing the right information to select the right device at the right time. This will require a little forethought from the programmer because of the nature of the D flip flops used to store the status of each channel. Returning to the preceding discussion on circuit operation, it will be recalled that the D flip flops toggle (reverse states) each time they are selected. However, simply selecting the same channel over and over again will not toggle it on and off as it might

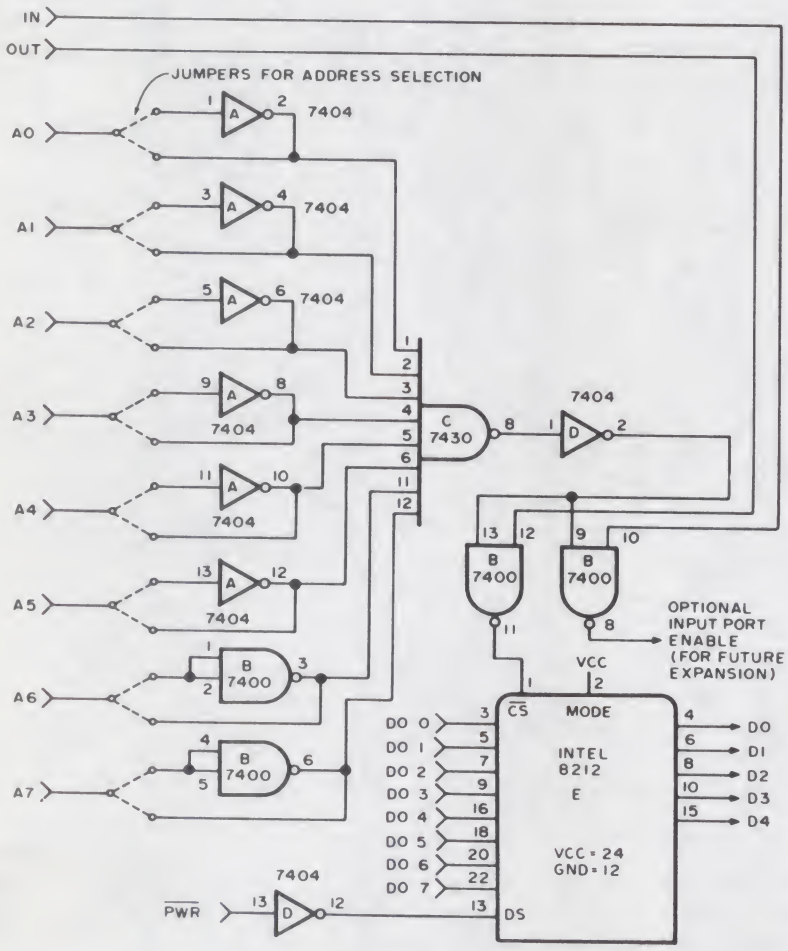


Figure 2: This is a standard parallel output port, capable of responding to any output address between zero and 255. The address is specified by the eight jumpers coming off the address lines. You may want to use low power chips (74L series) for IC A, IC B and IC C, to save on address bus loading. Incidentally, this addressed output port could be used in any application requiring a parallel output. All eight data lines are available at the various outputs of the 8212 chip. The IN and OUT and PWR inputs are for Altair 8800 and similar computers. The OPTIONAL INPUT PORT ENABLE line coming from pin 8 of IC B may be used to enable another 8212 chip with the CS pin to function as an input port and place data on the input bus when the IN line is active and the specified address is enabled.

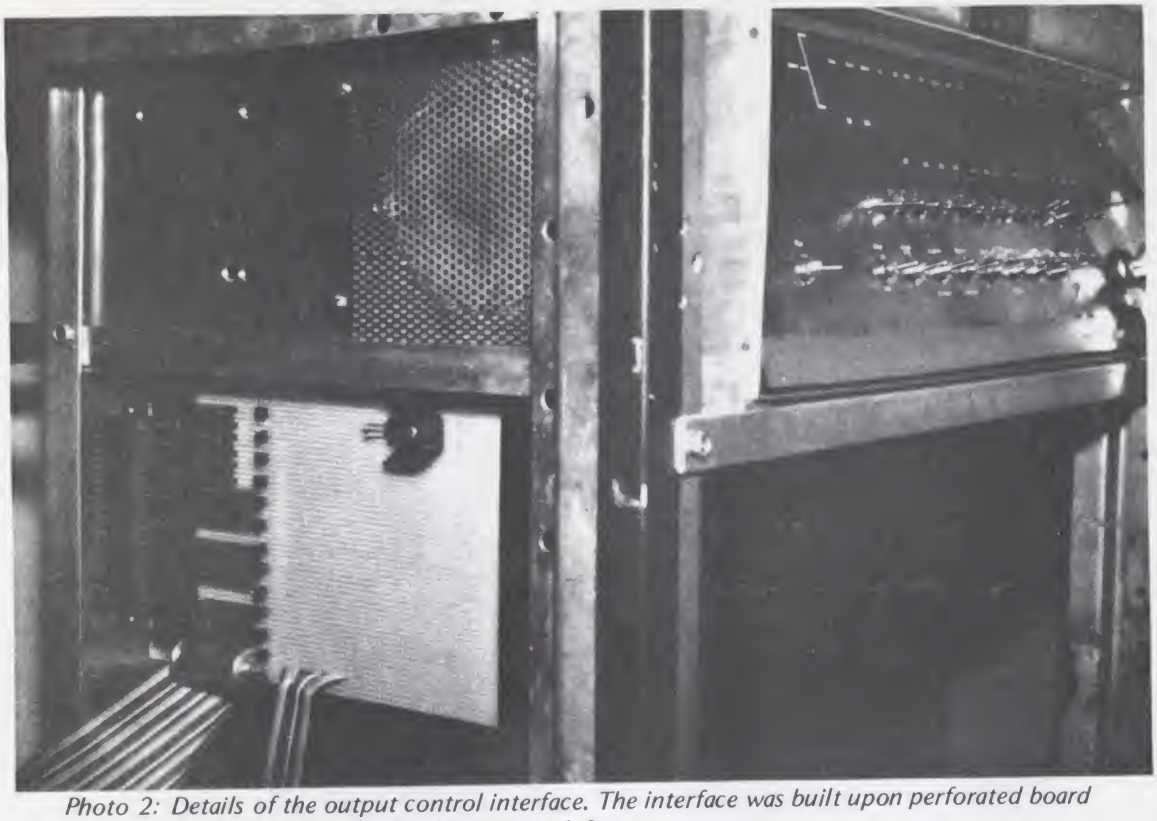
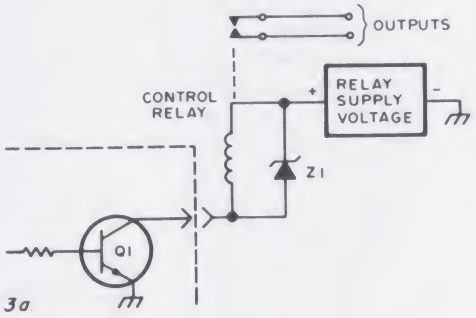
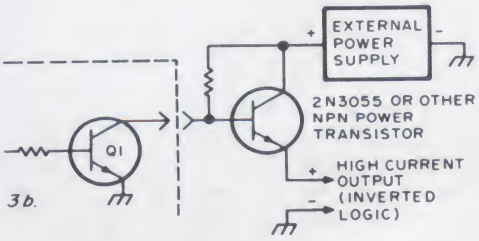


Photo 2: Details of the output control interface. The interface was built upon perforated board mounted at the side of the rack cabinet at the left.

be expected, because the D flip flops only toggle on *rising* edges from the demultiplexer, and a rising edge only occurs *after* a channel has been selected when the multiplexer *changes* to select (ground out) a different channel. So, turning a channel on and then off is accomplished by first selecting the desired channel with a data byte, then selecting a different channel (This might be an unused channel or the next sequential channel in your program), then waiting the delay needed for the first channel to switch on, then selecting it again to reset it. This may seem a little complicated at first, but it's easy to get used to.

Applications. Software and hardware will of course be determined by the application needed, and this will vary widely from instance to instance. The following ideas have occurred to me and you will undoubtedly think of many more: Light shows, computer music, industrial process control, computerized games, industrial robots, stage lighting, spook alleys (Electro-Spook?), slide presentations, darkroom automation, chemical mixing, remote controls of any type, or a fully programmable electrically operated teeter-totter. Try it – you'll like it!■





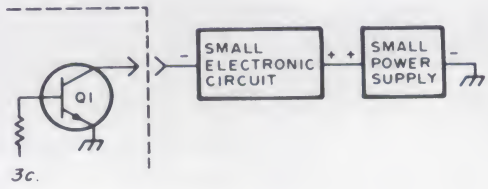


Figure 3a: Interfacing relays. Z_1 is used to protect Q_1 from spikes. Z_1 should have a breakdown voltage just higher than the relay voltage.

Figure 3b: Power transistor interface, suitable for powering tape recorders or other small appliances.

Figure 3c: Small load (<=30 mA) direct interface.

Microprocessor Based Analog

Roger Frank
1801 E Girard #247
Englewood CO 80110

Analog input and output capabilities, when added to a microcomputer, can greatly expand the power of the home or hobby computer. Inherently, the microprocessor is a digital device, ideal for control of discrete (on or off) input and output levels. However, many analog signals can also be processed with only minimal additional hardware. With this addition, such devices as temperature sensors or photocells can be monitored, and output peripherals such as oscilloscopes and audio amplifiers can be added to the microprocessor.

Taking traditional approaches to analog to digital conversion can be very expensive to the hobbyist. Hundreds of dollars could be spent, but this would yield only high speed or resolution. For the amateur, typically eight bits of accuracy is sufficient, and speed is not a critical factor. The brightness of the sun, the temperature of the room, or the moisture of the front lawn do not change very rapidly. By allowing the microprocessor to do most of the work involved in the conversion, a simple, inexpensive circuit can convert an analog input to a digital word in less than a millisecond. The overall cost can be kept under $20 for four channels of analog input.

An analog signal is typically a voltage level . . . which corresponds to measurement of some physical variable.

Two techniques of analog to digital conversion are easily accomplished by a microprocessor: the ramp and successive approximation methods. In each case, the task is to generate a digital word, apply it to a digital to analog converter (DAC), and compare the analog output of the DAC to the analog input to be converted. Based on the results of the comparison, the next digital word to the DAC is generated.

Analog signals can be processed with only a minimal addition of hardware to a system.

Traditionally, several gates, up-down counters, and clock generators are used to achieve the conversion. This approach is much more expensive than using the microprocessor to implement the same functions, using no external TTL logic in the conversion at all.

The Ramp Technique

The simplest approach is the ramp technique. It has the advantage of needing the least code in the microprocessor, but the disadvantage of being the slowest, some 15 times slower, on the average, than the successive approximation approach discussed later. For many applications, where speed is not critical, this approach may be best. Since the ramp technique is conceptually easiest to understand, it will be examined closely first.

Figure 1 shows the block diagram of the AD conversion system. Unlike hardware approaches, the identical components can be used for successive approximation, ramp, or tracking conversion algorithms. The hardware can be tailored, by software, to meet speed or accuracy requirements of the overall system.

To understand the circuit, assume in figure 1 that the analog input to the + input connector of the comparator is 2.00 V, and that all zero bits are applied to the DAC's digital inputs. The DAC's output will be 0 V at the comparator's – input connector. The comparator's output will be a 1 bit, which is applied to the microprocessor through an input port. The software, by reading and testing the input port, knows if the digital word applied to the DAC is too large or too small. In this case, the 1 bit read at the input port means "too small" and the microprocessor will increment the digital word at the input to the DAC. The output of the DAC increases by a small amount each time the comparator says "too small," until the DAC generated analog voltage just exceeds the "unknown" input voltage. At that moment, the comparator output will be read as a 0 bit, and the digital equivalent of the analog input voltage will be present at the input to the DAC.

This sequence, using an eight bit DAC, generates a ramp voltage at the input to the comparator with each step 1/256th of the

/Digital Conversion

A challenge: Write a program to send data to the DAC at regular intervals, connect the DAC output to a high fidelity amplifier, and play music with the DAC as a waveform generator.

full scale voltage. In this application, a five volt full scale is typical, so each step would be about 19.5 millivolts. Using the Motorola MC6800 microprocessor, a routine to accomplish this simple conversion would be as shown in listing 1.

Note that with the MC6800, IO is treated as a memory location, so it is simple to directly implement the algorithm. For the Intel 8008, a similar sequence could be used, as shown in listing 2. In this example, Register B will have the eight bit digital equivalent of the analog input when the sequence is complete.

The Successive Approximation Method

A faster technique, which always takes the same number of passes through the decision making loop, is the successive approximation method. The hardware is exactly the same, but instead of changing the least significant bits in incrementing fashion (19.5 millivolts per step), this method changes the most significant bits, one at a time, and very quickly homes in on the correct digital word.

Using the same example, with 2.00 V applied to the "unknown" input of the comparator, the sequence is like this. First, the most significant bit, bit 7, is set to a one in the DAC. The output of the DAC immediately goes to half scale, or 2.5 volts. (Remember that bit 7 represents 2**7 or 128 times the least significant bit's weight of 19.5 mV, which is about 2.5 volts.) Right away, the microprocessor knows that in the final digital word, bit 7 will be a zero, since the comparator is already saying "too high" with that bit only set in the DAC. The microprocessor removes bit 7 from the DAC and sets bit 6 to a one. Now the DAC output of 1.25 V is compared to the 2.00 V "unknown" input to the comparator, and the processor quickly learns that bit 6, by itself, is "too low," since 1.25 V is less than

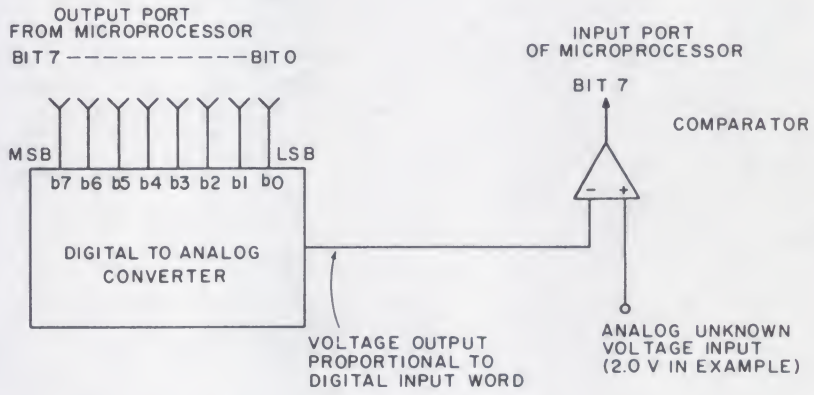


Figure 1: The microprocessor controlled analog digital conversion system consists of an 8 bit DAC output which is compared against the unknown input.

```
1  RAMP   CLR  DAC    start conversion at zero;
2  RLOOP  INC  DAC    increment output voltage;
3         TST  COMP   test comparator input of bit 7 (N);
4         BMI  RLOOP  back for more until done;
5         RTS         return to caller;
```

Listing 1: The ramp method of conversion, specified as a symbolic assembly language program for the Motorola 6800 central processor.

```
1  RAMP   XOR  A     clear the accumulator with XOR;
2         LBA        clear B from A;
3  LOOP   INC  B     increment DAC input word by one;
4         LAB        move to accumulator for output;
5         OUT  DAC   output to DAC device code;
6         INP  COMP  input from comparator device code
7         JTS  LOOP  using sign bit for comparator;
8         RET        return when done;
```

Listing 2: The ramp method of conversion, specified as a symbolic assembly language program for the Intel 8008 processor.

```
1   SUCAPPRX    CLR     A          result will be in A;
2               LDAB    #$80       rotating mask, most significant first;
3   NEXTBIT     ABA                apply trial bit to A with addition;
4               STAA    DAC        send it to the output DAC latch;
5               LDAA    COMP       read the comparator output;
6               ANDA    #$80       check sign bit with comparator output;
7               BNE     RETAIN     if low then retain trial bit;
8               LDAA    DAC        recover the DAC word;
9               SBA                restore zero to last trial bit;
10              BRA     MSHIFT     then go shift the rotating mask;
11  RETAIN      LDAA    DAC        keep the trial bit as logical one;
12  MSHIFT      ROR     B          rotate the mask;
13              BCC     NEXTBIT    on eighth rotate, carry set
14              RTS                   so return from the conversion;
```

Listing 3: A successive approximation conversion, specified as a symbolic assembly language program for the Motorola 6800 processor. This program was adapted from a Motorola application note on the subject. Note that for fast processors or slow operational amplifiers (such as the 741), a delay loop should be inserted between lines 4 and 5 of this program to allow the output to settle.

```
1   TESTPGM     LDAA    #$00       load test value for DAC;
2               STAA    DAC        and store it in the DAC;
3               RTS                then return to caller;
```

Listing 4: A test program which can be used to load the immediate value of 0 into the DAC output port. The symbolic location DAC is assumed to be the output port address.

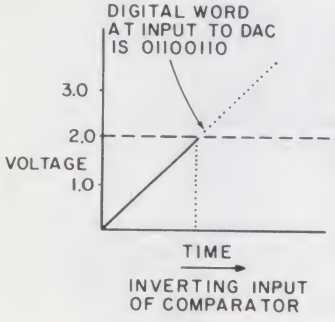


Figure 2: A ramp conversion starts at zero voltage output and increases the voltage until it equals or just exceeds the unknown input. For larger input voltages, conversion takes longer since the program must cycle through all the intermediate values from zero to the final binary word.

2.00 V. In this case, the processor leaves bit 6 on and adds the bit with lesser significance, bit 5. With bit 6 and bit 5 on, the DAC output voltage is 1.87 V, still too low. Thus, bit 5 also is left on and the next bit in line is tried.

The algorithm is this: simply try a bit, starting at the most significant. If the DAC generated voltage exceeds the "unknown," remove that bit only, else keep it. Try the next bit, repeating the process until all bits have been determined. In this case, eight passes through the loop will result in the complete digital equivalent of the unknown analog voltage input in a matter of milliseconds.

This faster technique has been implemented with the MC6800 microprocessor with the sequence shown in listing 3. A sustained rate of 1000 conversions per second has been achieved.

An actual circuit to implement these techniques is shown in figure 2. The circuit uses an inexpensive Motorola MC1408L-8 digital to analog converter, which converts digital inputs to a current output at pin 4. Current output, which is subsequently converted to a voltage, is typical with DACs.

The circuit to the left of the DAC is a simple zener diode voltage regulator. The zener maintains a constant voltage drop across the resistor R1, since the right side of the resistor is at virtual ground. The current through R1 is the reference current, which is either absorbed internally or steered out the DAC's pin 4. How much current leaves the DAC is a function of the digital input word applied on pins 5 through 12.

The current cannot be compared to the unknown analog input voltage without some conversion. Dig out your operational amplifier articles and you'll realize that the LM301 is functioning as a current to voltage converter, which changes the 0 to 2 mA output of the DAC into a 0 to 5 V voltage. This voltage, after a little filtering, is then applied to an LM311 comparator.

The LM311 has the useful feature of having an analog comparator input, but a TTL compatible (open collector) output. The LM311 output can be directly applied to an input port of the microprocessor for program controlled evaluation. Resistors R6 and R5 add a little hysteresis to the comparator and, like the filtering components C1, C2, C4 and R7, are recommended, though not absolutely essential to the operation of the circuit. Similarly, a 741 type opamp can be used in place of the LM301, but the circuit will take longer to convert the current output of the DAC into a stable voltage at the input to the LM311.

Circuit calibration is simple and consists of only one adjustment. First apply all zeros to the digital input to the DAC. The voltage at pin 6 of the LM301 should be very nearly zero volts. If it isn't, check your circuit carefully. If off by only a few millivolts, a small offset current could be injected into the input of the LM301 to make it exactly zero volts, but for eight bit accuracy this should not be necessary. Now apply all 1 bits to the DAC input. The output of the current to voltage converter should now be adjusted to 5.00 V with resistor R4. With this setting, you have calibrated to the 19.5 mV/b specification used in the examples.

Expansion of this circuit, once the single channel version is complete, is straightforward and very inexpensive. For example, each additional channel of analog to digital conversion can be added with only an additional comparator. Each added LM311 has its output connected to a separate input port bit, up to eight channels per port for an 8 bit processor. Then in software, choose the channel of interest by logically masking out

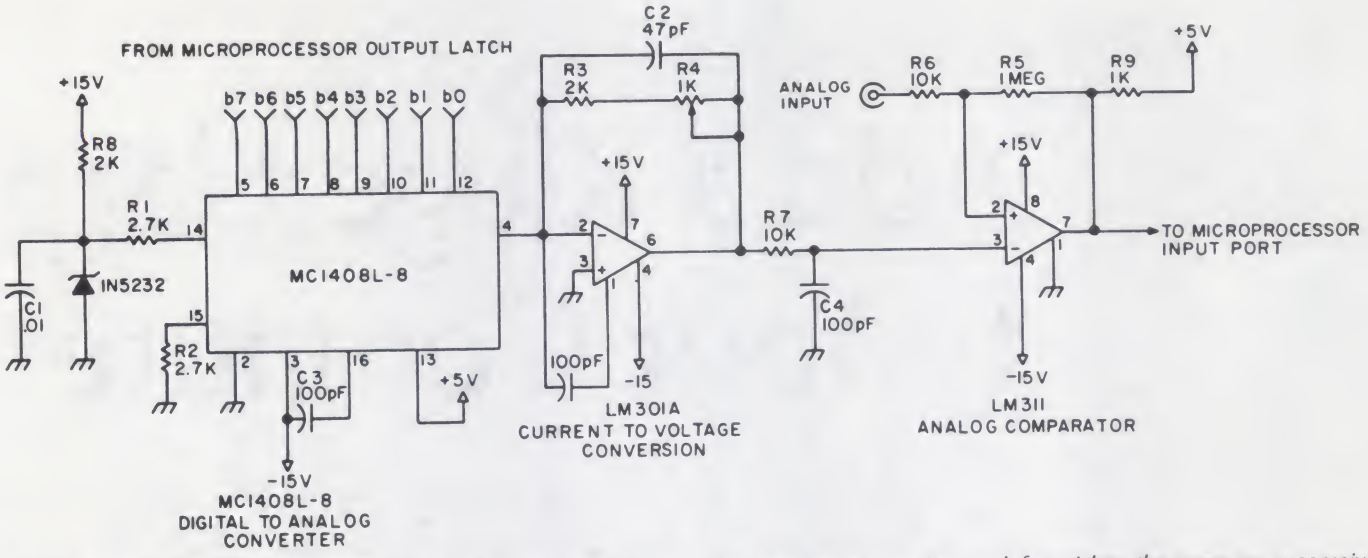
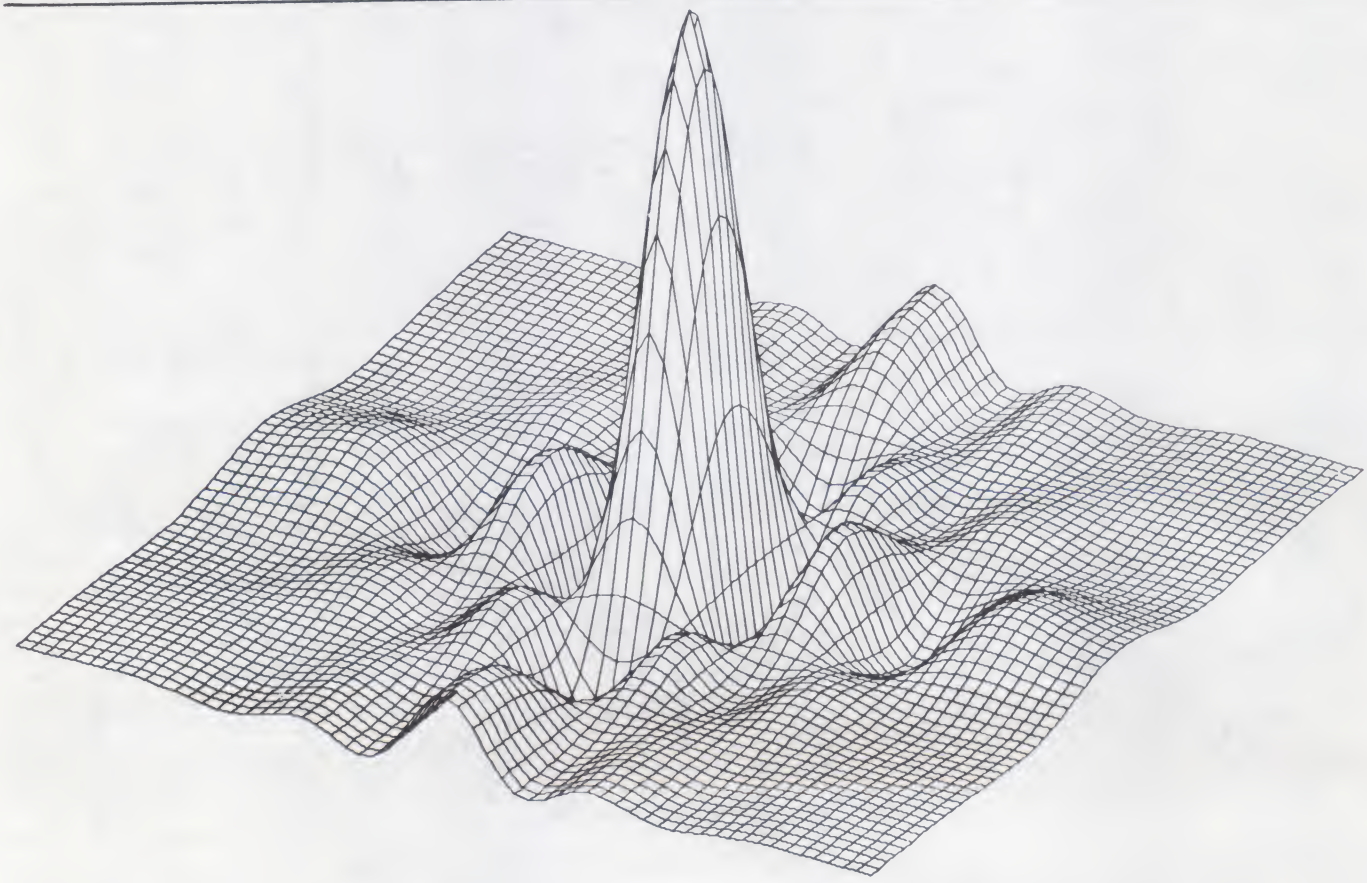


Figure 3: Schematic of the circuit used for 8 bit conversions. This hardware can be used for either the ramp or successive approximation methods described in this article.

all the other channels. Here the LM339 can be used to have four comparators, and four channels of AD, in one package. Similarly, at no charge, this circuit can be used as a source of digitally programmed analog voltage to deflect an oscilloscope trace or act as a computer controlled function generator, producing extremely complex waveforms, if desired. Another use could be a keyboard controlled power supply with suitable current gain added to the DAC output.

These techniques and this inexpensive circuit open a wide world of analog interfacing to the microprocessor hobbyist. Now the home computer can go beyond the number crunching, logic control functions and talk to the real world on its own analog terms.■



Add a Kluge Harp to Your Computer

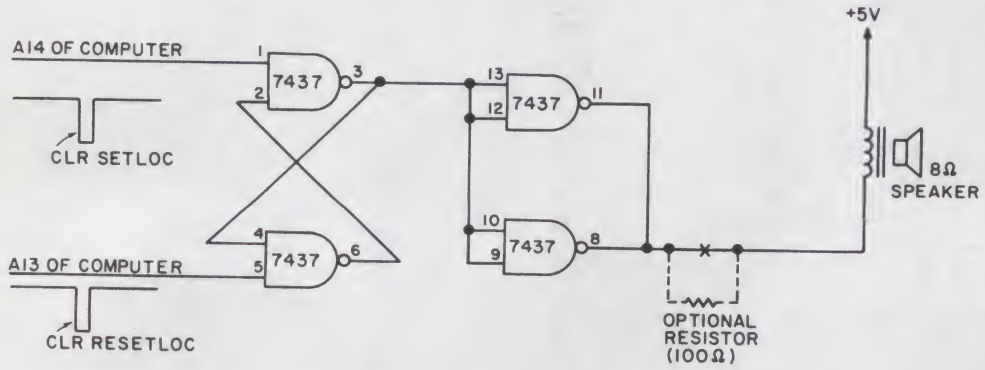
by
Carl Helmers
Editor, BYTE

One of the most interesting computer applications is that of electronic music. This is the use of software/hardware systems to produce sequences of notes heard in a loud speaker or recorded on magnetic tape. The idea of generating music — if well done — is of necessity complex. If I want to put my favorite Mozart piano sonata into an electronic form, I'd have to record a very large number of bits in order to completely specify the piece with all the artistic effects of expression, dynamics, etc . . . The magnitude of the problem can be intimidating. But, never let a hard problem get in the way of fun! Simplify the music problem to one channel of melody, and you can use a virtually bare CPU with a very simple peripheral to play music.* The combination of the CPU with this simple peripheral is what I call the "Kluge Harp" — a quick and dirty electronic music kluge.

I invented this electronic music kluge to answer a specific problem: I had just gotten a new Motorola 6800 system's CPU, memory and control panel up and running. The next problem (since I wasn't using the Motorola ROM software) was to make a test program which could be loaded by hand. By combining a little imagination, my predilections for computer music systems and an evening getting the whole mess straightened out, the Kluge Harp resulted. While the program and schematic are specific to the system I was using, the *idea* can be applied to your own system just as well.

(*ALTAIR owners: Write an 8080 version of this program and your machine can do more than blink its lights.)

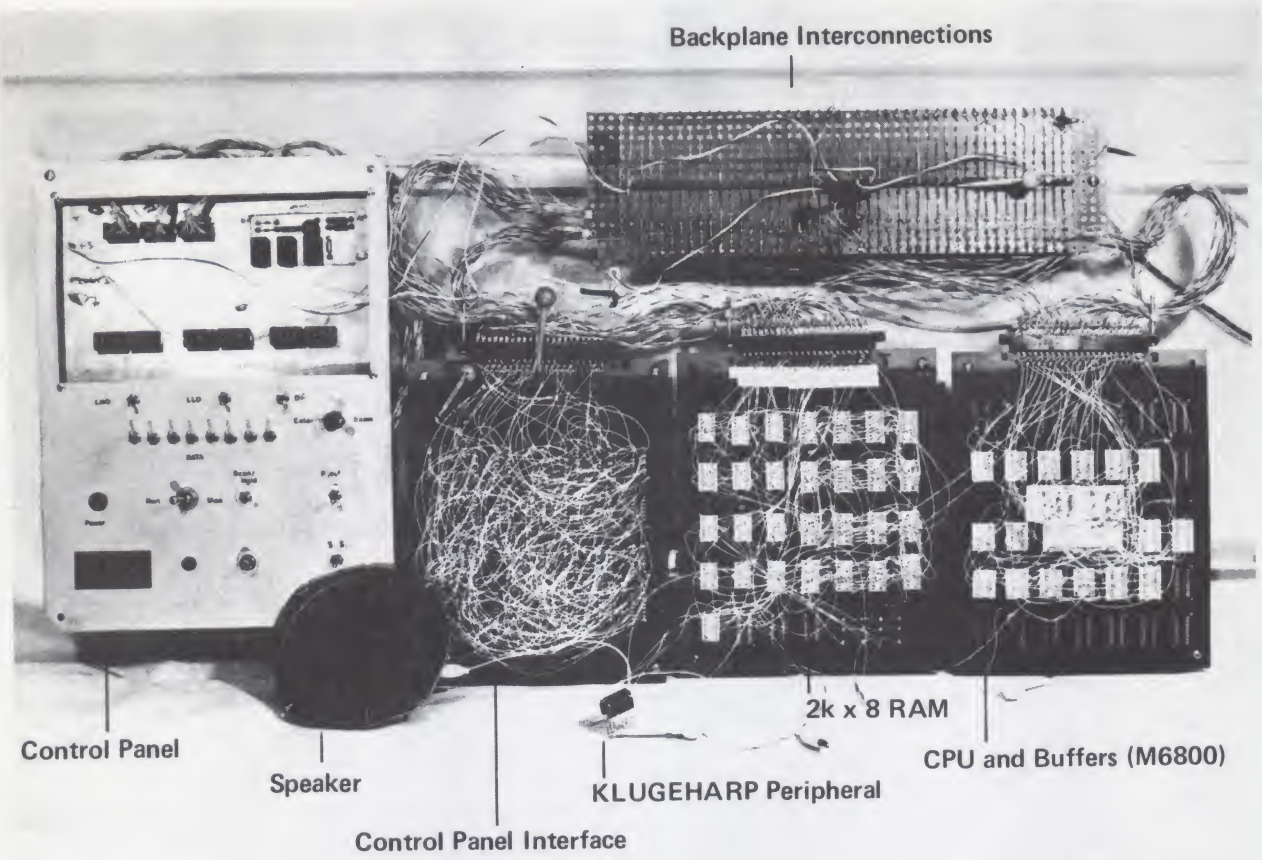
Fig. 1. The Kluge Harp Circuit . . . minus computer.



The Kluge Harp Hardware

The hardware of a Kluge Harp is simplicity at its essence. The peripheral is driven off two "un-used" high order address lines (I used A14 and A13), and consists of a set-reset flip flop. A program running in the computer alternately will set and reset the flip flop by referencing one or the other of two addresses. These addresses are chosen so that the address lines in question will change state, actuating the set or reset side of the flip flop. A "note" at some pitch consists of a delay loop in the program followed by instructions to change the state of the flip flop. Since the same count is used for the two halves of a complete cycle of the note, this will produce a perfect square wave. The actual music program organization is a bit

Michael Gipe



The Kluge Harp peripheral and the KLUGEHARP program were concocted in order to test out a Motorola 6800 system's operation. This photo shows a test bench mounting of the three main cards and control panel. The Kluge Harp peripheral, such as it is, is the single isolated wire wrap socket in the foreground, with wires dangling from connections on the CPU card.

more complex and is described in detail below.

Fig. 1 illustrates the hardware as implemented in my system. The 7437 circuit is used to form the NAND gate flip flop. This flip flop in turn drives a parallel combination of the two remaining 7437 gates, acting as a buffer. The output of this buffer is used to drive the speaker; an 8 Ohm 5" speaker produced more than adequate volume. (A 100 Ohm resistor in series will limit the volume level to spare the ear drums.)

Generating Music With Program Loops

Fig. 2 illustrates the basic concept of the one-channel music generator, expressed in a procedure-oriented language for compactness. The main program loop begins at line 2 of the listing – "DO FOREVER" means repeat over and over again all the lines of code down through the "END" at the same margin, found at line 17. This is the main loop used to cycle through the SCORE stored at some point in memory as pairs of note selection/length data bytes.

Lines 3 to 4 compute the "next" pointer to the SCORE – incrementing NOTER by 2. Then LNGTH is set equal to the second byte of the current pair, SCORE (NOTER+1). The length codes are taken from Table I along with note codes when you set up a SCORE, and represent a fixed interval of time for the note in question, measured as the number of cycles.

Line 6 begins a note length loop which extends to line 14. This "note length" loop repeats the generation of the note a number of times indicated by the length code just retrieved. The note generation is accomplished by delaying a number of time units (CPU states) set by the pitch code found at SCORE(NOTER), then changing the state of the output flip flop and repeating the process. The loop at lines 8-10 counts down the pitch code and has a fixed delay multiplied by the pitch code to give the time for one half cycle of the desired frequency. Lines 11 to 15 change the state of the Kluge Harp output device (0 to 1, 1 to 0) – remembering in the software location IT what the previous state was.

Generating Codes

Table I is a reference table of 21 notes "roughly" spaced at equal intervals on the well tempered scale. The integer numbers in the "divide ratio" column were determined using the prime number 137 as an arbitrary starting point and calculating the integer closest to the result of the following formula:

$$r_n = e^{(\ln(137)+ n \ln(2)/12)}$$

Where e is the usual mathematical number 2.717 . . . and the natural logarithm of x (base e) is indicated by ln(x). This is the standard mathematical calculation of the musical "well tempered" scale – the 8-bit approximation used by the Kluge Harp is not perfect by any means, but comes close enough for the purposes of this project.

The length count columns are determined based upon the assembly language generated code for this

Fig. 2. The KLUGEHARP program specified in a procedure-oriented computer language.

```
1   KLUGEHARP:  PROGRAM;
2       DO FOREVER;
3           NOTER = NOTER + 2;
4           IF NOTER = NOTEND THEN NOTER = NOTESTART;
5           LNGTH = SCORE(NOTER+1); /* SECOND OF TWO BYTES */
6           DO FOR I = LNGTH TO 1 BY -1;
7               PITCH = SCORE(NOTER); /* FIRST OF TWO BYTES */
8               DO FOR J = PITCH TO 1 BY -1;
9                   /* COUNT DOWN THE PITCH DELAY */
10              END;
11              IT = IT + (-127); /* SWITCH SIGN BIT OF IT */
12              IF IT 0 THEN
13                  SETLOC = 0; /* SET FLIP FLOP WITH MEMORY REF */
14              ELSE
15                  RESETLOC = 0; /* RESET FLIP FLOP WITH REF */
16          END;
17      END;
18  CLOSE KLUGEHARP;
```

routine, so that for each pitch, the corresponding length count column will measure a nearly identical interval of time. The formula is:

Lc_n = time / (oh + dt# pc_n) where:

Lc_n = n^{th} length count.

time is the total number of states for one "beat" of the music (e.g., the shortest note).

oh is the overhead of the length counting loop.

dt is the number of states in the pitch count innermost loop.

pc_n is the pitch count for the n^{th} frequency.

Table I shows the divide ratio in decimal, a hexadecimal equivalent note pitch code, and seven

Table I. Kluge Harp Synthesizer pitch/length specification codes (HEX).

n	divide ratio	hex note code	Note Length Codes (second byte of pair) 1	2	4	6	8	16	32
-10	77	4D	19	32	64	96	C8	–	–
-9	81	51	18	30	60	90	C0	–	–
-8	86	56	17	2D	5A	87	B4	–	–
-7	91	5B	16	2B	56	81	AC	–	–
-6	97	61	14	29	51	7A	A2	F3	–
-5	102	66	13	27	4D	74	9A	E7	–
-4	108	6C	12	25	49	6E	92	DB	–
-3	115	73	11	23	43	68	8A	CF	–
-2	122	7A	10	21	41	62	82	C3	–
-1	129	81	10	1F	3E	5D	7C	BA	F8
0	137	89	0F	1D	3A	57	74	AE	E8
1	145	91	0E	1C	37	53	6E	A5	DC
2	154	9A	0D	1A	34	4E	68	9C	D0
3	163	A3	0C	19	31	4A	62	93	C4
4	173	AD	0C	18	2F	47	5E	8D	BC
5	183	B7	0B	16	2C	42	58	84	B0
6	194	C2	0B	15	2A	3F	54	7E	A8
7	205	CD	0A	14	28	3C	50	78	A0
8	217	D9	09	13	25	38	4A	6F	94
9	230	E6	09	12	23	35	46	69	8C
10	244	F4	08	11	21	32	42	63	84

Data assumed by KLUGEHARP:

NOTER: 16-bit (two-byte) address value. Initialize to point to the address of the first byte of SCORE.

SCORE: An array of data in memory containing the code sequence of the music (see Table II). Initialize with the music of your heart's desire or use the example of Table II.

NOTEND: 16-bit address value, the address of the last byte of SCORE (must be an even number).

NOTESTART: 16-bit address value, the address of the first byte of SCORE (must be an even number).

SETLOC: An unimplemented address location which if referenced turns off one bit among the high order address lines, bit 14 in the author's case.

RESETLOC: An unimplemented address location which if referenced turns off one bit among the high order address lines, bit 13 in the author's case.

Data used but not initialized:

LNGTH

PITCH

IT

I, J

Fig. 3. Motorola 6800 Code for KLUGEHARP program.

Address	Data	Label	Opcode	Operand	
F800	FE	KLUGEHARP:	LDX	NOTER	
F801	FA	3:			
F802	00				Add 2 to location in score
F803	08		INX		by incrementing and then
F804	08		INX		saving 16-bit new address
F805	FF		STX	NOTER	
F806	FA				
F807	00				
F808	8C	4:	CPX	#NOTEND	compare against immediate
F809	FC	NOTEND:	(last address of		
F80A	80		SCORE plus 2)		
F80B	26		BNE		Skip if not at end . . .
F80C	03		*+3+2		
F80D	CE		LDX	#NOTESTART	
F80E	FC	NOTESTART:	(first address of		otherwise recycle
F80F	00		score . . .)		
F810	FF		STX	NOTER	save in either case . . .
F811	FA				
F812	00				
F813	FE		LDX	NOTER	This is superfluous!
F814	FA				
F815	00				
F816	E6	5:	LDAB	1,X	
F817	01				
F818	5A	LENGTH:	DECB		
F819	26	6:	BNE		Skip if length remains . . .
F81A	03		*+2+3		
F81B	7E		JMP	KLUGEHARP	Restart piece
F81C	F8				
F81D	00				
F81E	A6	7:	LDAA	0,X	
F81F	00				
F820	4A	FLOOP:	DECA		
F821	26	8:	BNE	FLOOP	
F822	FD		*+2-3		
F823	86	11:	LDAA	#80	
F824	80		(-127)		
F825	BB		ADDA	IT	
F826	FA				
F827	02				
F828	2B	12:	BMI		
F829	05		*+2+5		
F82A	7F	13:	CLR	SETLOC	
F82B	B0		(address with bit 14 off . . .)		
F82C	00				
F82D	20		BRA		
F82E	03		*+2+3		
F82F	7F	15:	CLR	RESETLOC	
F830	D0		(address with bit 13 off . . .)		
F831	00				
F832	B7		STAA	IT	
F833	FA				
F834	02				
F835	7E	16:	JMP	LENGTH	
F836	F8				
F837	18				

Data allocations for KLUGEHARP:

FA00-FA01 = Current pointer to SCORE, NOTER, which should be initialized to FC00 before starting the program.

FA02 = IT — an arbitrary initialization will do.

FA03-FFF7 = memory area available for SCORE — the example uses FC00 to FC7F and puts the relevant initializations into locations F809-F80A (NOTEND) and F80E-F80F (NOTESTART).

NOTE: In the label column, the numbers followed by colons (e.g., "6:") are used to indicate corresponding places in the high level language version of the program of Fig. 2.

In the system for which this program was written, all active memory is found at addresses F800 to FFFF. Thus for all normal program activity, bits A14 and A13 at the back plane of the system are logical "1". When the location SETLOC (B000) is cleared, the high order address portion changes and bit A14 goes to negative for a short time, setting the Kluge Harp flip flop. When the location A13 is cleared (D000) on an alternate cycle, address bit A13 goes to logical 0 for a short timer resetting the Kluge Harp flip flop . . .

columns of hexadecimal length codes weighted to 1, 2, 4, 6, 8, 16 and 32 unit intervals of time. A note is placed in the score by picking a note code, putting it in an even numbered byte, then placing a length code from the same line of the table in the odd numbered byte which follows it. The actual pitches you'll get from these codes depend upon the details of the algorithm in your own particular computer and the clock rate of the computer. For the 6800 system on which Kluge Harp was first implemented, the lowest note (code F4) is approximately 170 Hz with a 500 kHz clock — and the unit interval of time is approximately 2000 CPU states or about 4 milliseconds.

The hand assembled M6800 code for the KLUGEHARP program is listed in Fig. 3. The mnemonics and notations have been taken from the Motorola *M6800*

Table II. WOLFGANG: Set the content of SCORE in memory to the codes in this table – given for the addresses of the M6800 program version – and KLUGEHARP will play four bars from the classical period.

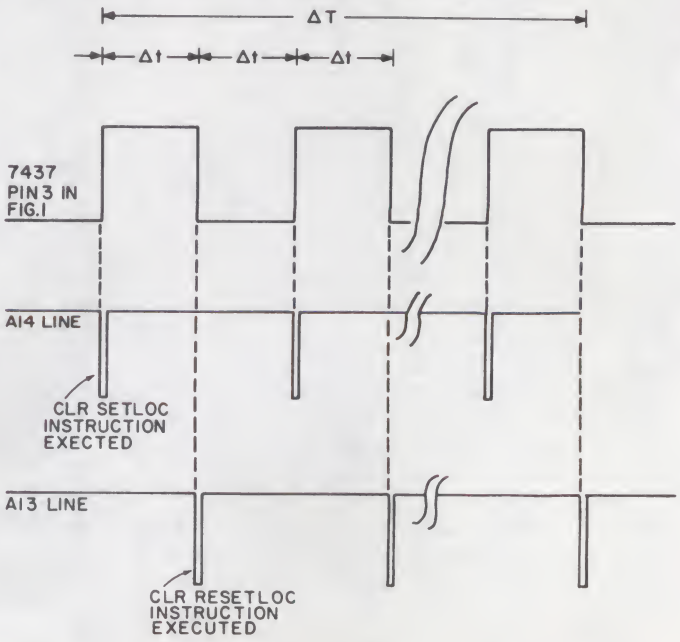
6800 Address	Value		6800 Address	Value	
FC00	9A34		FC40	5B56	
FC02	9A34		FC42	5B56	
FC04	9A34		FC44	5B56	
FC06	9A34		FC46	5B56	
FC08	9A34	Note 1	FC48	5B56	Note 8
FC0A	9A34		FC4A	5B56	
FC0C	9A34		FC4C	5B56	
FC0E	9A34		FC4E	5B56	
FC10	7A41		FC50	664D	
FC12	7A41	Note 2	FC52	664D	Note 9
FC14	7A41		FC54	664D	
FC16	7A41		FC56	664D	
FC18	664D		FC58	4D64	
FC1A	664D	Note 3	FC5A	4D64	Note 10
FC1C	664D		FC5C	4D64	
FC1E	664D		FC5E	4D64	
FC20	A331		FC60	664D	
FC22	A331		FC62	664D	
FC24	A331	Note 4	FC64	664D	Note 11
FC26	A331		FC66	664D	
FC28	A331				
FC2A	A331		FC68	7343 –	Note 12
			FC6A	664D –	Note 13
FC2C	9A34 –	Note 5	FC6C	7343 –	Note 14
			FC6E	7A41 –	Note 15
FC2E	893A –	Note 6	FC70	7343 –	Note 16
FC30	9A34		FC72	7A41	
FC32	9A34		FC74	7A41	
FC34	9A34		FC76	7A41	
FC36	9A34	Note 7	FC78	7A41	Note 17
FC38	9A34		FC7A	7A41	
FC3A	9A34		FC7C	7A41	
FC3C	9A34		FC7E	7A41	
FC3E	9A34		FC80	(end pointer points here)	

NOTE: This program is simpleminded and not at all optimized. As a challenge to readers, figure out a way to make the notation more compact yet preserving the total length of each note.

Microprocessor Programming Manual available from the manufacturer.

While not the greatest musical instrument in the world, the Kluge Harp represents an interesting and challenging diversion. The program presented here is by no means the ultimate in music systems – and can serve as a basis for further experimentation and elaboration. Some challenges for readers: modify the program to change the frequency of the notes without changing the SCORE data; write another (longer) music program which only specifies the pitch code/length information once – and represents the score as a series of one-byte indices into the table of pitch code/length information.

Fig. 4. Timing of the Kluge Harp Output Waveform. Δt is the amount of time spent in the inner loop, and is set by choice of pitch codes. ΔT is the length of the note, measured as a count of half-cycles at its frequency. See Table I for a consistent set of length codes.



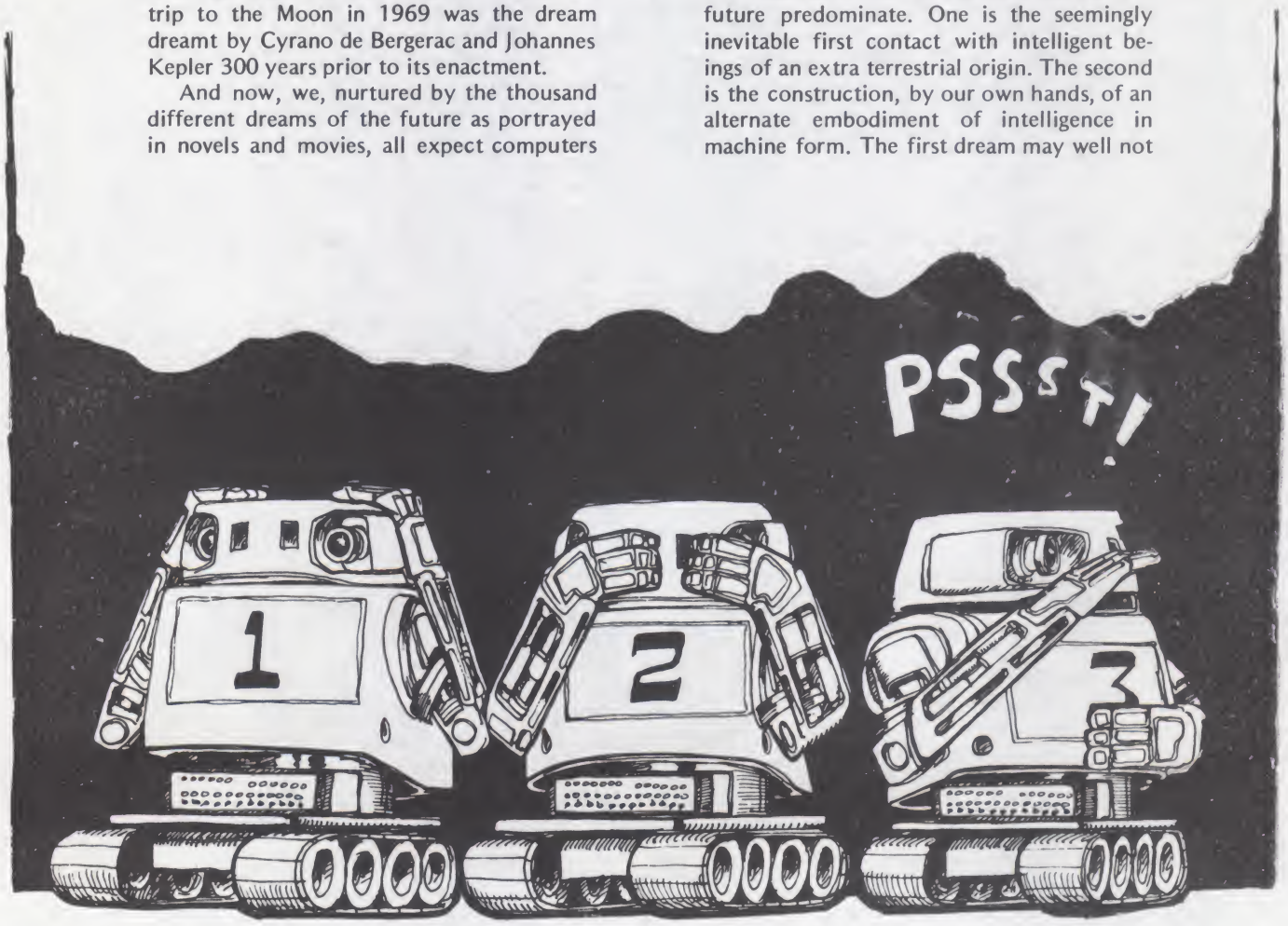
The Time Has Come to Talk

Wirt Atmar
Ai Cybernetic Systems
PO Box 4691
University Park NM 88003

The extent to which art and literature, particularly science fiction, affect the future course of civilization remains a persistent and perplexing question. Must a dream, by necessity, occur decades before its realization? Or does the presence of the dream itself generate its own reality? Mankind's trip to the Moon in 1969 was the dream dreamt by Cyrano de Bergerac and Johannes Kepler 300 years prior to its enactment.

And now, we, nurtured by the thousand different dreams of the future as portrayed in novels and movies, all expect computers to be able to talk in the near future. Whether we see the computer becoming the benign and obedient servant of man or wildly out of control, we all tend to see the computer becoming more anthropomorphic, more humanlike in behavior and form.

In science fiction two great dreams of the future predominate. One is the seemingly inevitable first contact with intelligent beings of an extra terrestrial origin. The second is the construction, by our own hands, of an alternate embodiment of intelligence in machine form. The first dream may well not



"The time has come," the Walrus said,
"To talk of many things:
Of shoes — and ships — and sealing wax —
Of cabbages — and kings —
And why the sea is boiling hot —
And whether pigs have wings."

— Lewis Carroll, 1871, in **Through the Looking-Glass.**

occur within the lifetime of our civilization; the second would seem to be almost guaranteed within the next 100 years.

The addition of speech to the computer's behavioral repertoire makes the computer no more intelligent nor aware than it was before. It remains a simple machine. But it undeniably takes on a human characteristic that it never possessed before. An observer finds it impossible not to personify the machine with an identity and a distinct personality. While the addition of speech is only a minor step toward achievement of a truly self-organizing, artifically intelligent machine, it is a psychologically important one. The computer, once it speaks, *seems* to be intelligent. But again, the dream of machine produced speech is much older than its reality. The ancient Greco-Roman civilization was fascinated with the idea of *deus ex machina.* Stone gods were often hollowed to allow a priest to speak from within, a practice that persisted well into the Christian era.

The first known practical realization of machine generated speech was accomplished in 1791 by a most ingenious engineer, Wolfgang von Kempelen, of the Hungarian government. Von Kempelen's machine was based on a surprisingly detailed understanding of the mechanisms of human speech production, but he was not taken seriously by his peers due to a previous well publicized deception in which he built a nearly unbeatable chess playing automaton. The "automaton" was unfortunately later discovered to actually conceal a legless Polish army ex-commander who was a master chess player.

By 1820, a machine was constructed which could carry on a normal conversation when operated by an exceptionally skilled person. Built by Joseph Faber, a Viennese professor, the machine was demonstrated in London where it sang "God Save the Queen." Both the Von Kempelen and Faber machines were mechanical analogs of the human vocal tract. A bellows was provided to simulate the action of lungs; reeds were used to simulate the vocal cords, and variable resonant cavities served to simulate the mouth and nasal passages.

The basic method, modelling the human vocal tract, remains to this time the only practical method of actually synthesizing speech. In the 20th century, such modelling is done electronically. The approach was first put in electrical analog form by Bell Laboratories in the late 1930s. The Bell Telephone VODER (Voice Operation DEmonstratoR) was initially shown at the 1939 New York's World Fair where it drew large crowds and considerable attention. The VODER consisted of a buzz source (similar to human vocal cords or mechanical synthesizers), a hiss source to simulate the rush of aspirated air, and a series of frequency filters to imitate the three, four, five or six preferred frequencies (called formant frequencies) passed by the resonant cavities formed by the mouth, tongue and nose.

The original VODER was played by highly trained operators using a keyboard, wrist switches, and pedals at an organ-like console. Twenty four telephone operators were trained six hours a day over a 12 month period for the 1939 World's Fair. The VODER itself was a full rack in height.

With the advent of digital computers, however, the synthesis of speech has been made much easier. All the information necessary to repeatedly and reliably generate any one speech sound (a "phoneme") can now be programmed into the machine. Through the proper connection of phonemes, a digital computer could be made to say words and sentences.

General American English, the dialect spoken in the midwest and southwestern parts of the United States, contains 38 distinct phonemes. These speech sounds can be divided into the following classes:

Pure vowels: produced by a constant excitation of the larynx and the mouth held in a steady position; eg: "ē".

Diphthongs: a transition from one

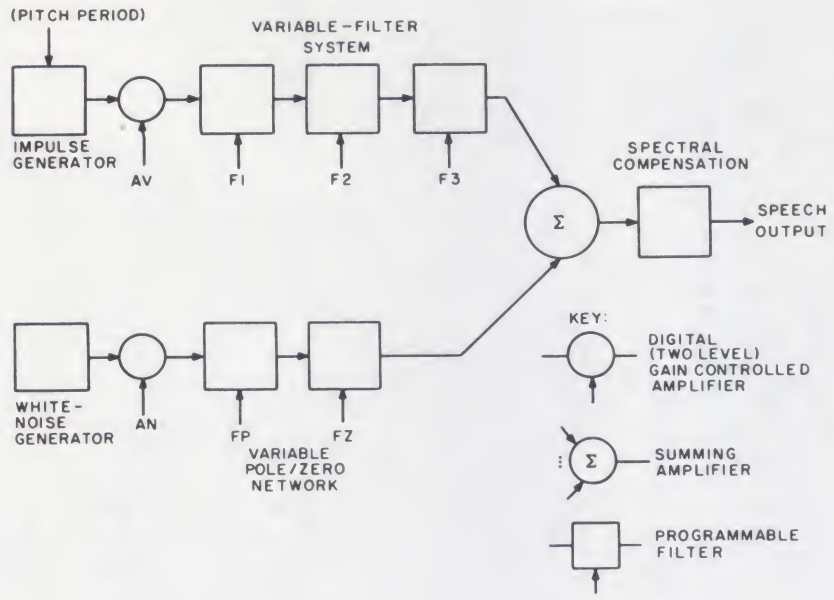


Figure 1: The serial analog speech synthesizer in block diagram form.

pure vowel to another, thus are not always considered as separate phonemes; "ī", "ū".

Fricatives: consonants produced by a rush of aspirated air through the vocal passages: "f", "s".

Plosives: explosive bursts of air: "p", "k", "t".

Semi-vowels: "w", "y".

Laterals: "l", "r".

Nasals: "n", "m".

To produce speech, a separate circuit, or combination of circuits, must be provided to generate each of the above classes of phonemes.

Among possible realizations of such a synthesizer, there are the serial analog and parallel analog forms. Figure 1 illustrates a block diagram of a serial analog design, and figure 2 shows the general organization of a parallel analog synthesizer.

The parallel analog method was the realization chosen by Ai Cybernetic Systems for its synthesizer module. The parallel realization was chosen because of the low digital information transfer rate and the smaller number of bits required to control the filters which simulate the resonant cavity of the vocal tract.

In the Ai Cybernetic Systems design, the rush of aspirated air is generated by the noise of a zener diode operated at its knee, amplified many times, as shown in figure 3. The action of the larynx is simulated by an integrated circuit function generator. One or both of these circuits is selected to produce the excitation necessary to generate any one class of phonemes. The actual phoneme perceived is determined by the duration of the excitation and the selected formant filters. Figure 4 shows the typical formant filter circuits which are digitally activated by analog switches.

The control of the several analog switches is provided by a read only memory which is addressed by the ASCII bit patterns identified in table 1.

No hard and fast rules exist in the design of the circuitry to generate a phoneme. In fact, small changes in component values can often make large differences in the phoneme which is actually heard. Because no set rules exist, a steady stream of listeners must parade before the machine while it is being designed in order to determine which phoneme the synthesizer is really saying. The phenomenon of "tired ears" rapidly sets in; and a person will begin, after a bit, hearing any one speech sound as a whole array of possible phonemes. Suggestion, on the other hand, is an ever obtuse enemy to the designer. Surprisingly, almost any speech sound can be suggested to sound like a great

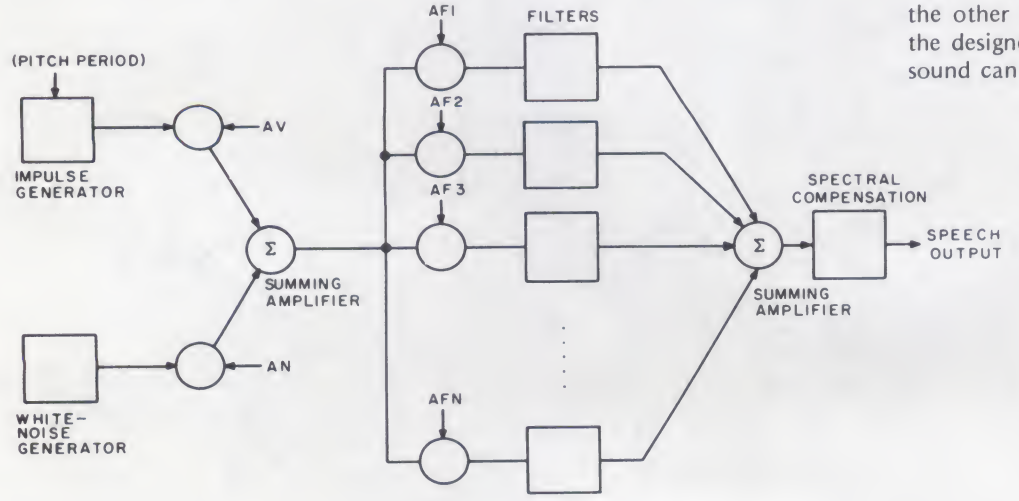


Figure 2: The parallel analog speech synthesizer in block diagram form.

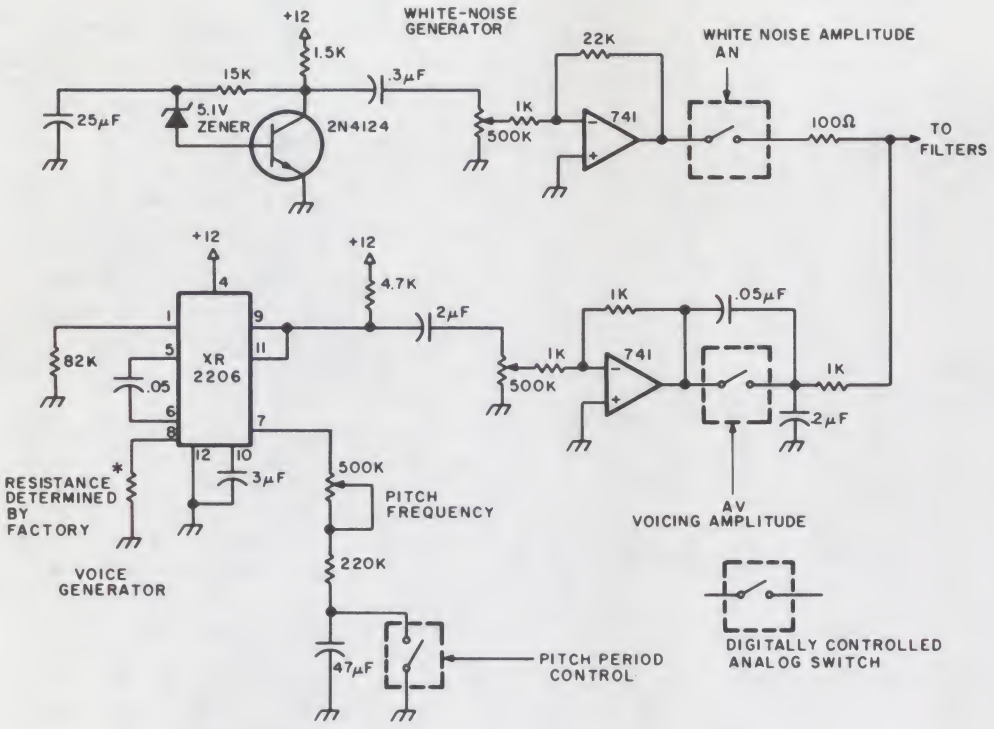


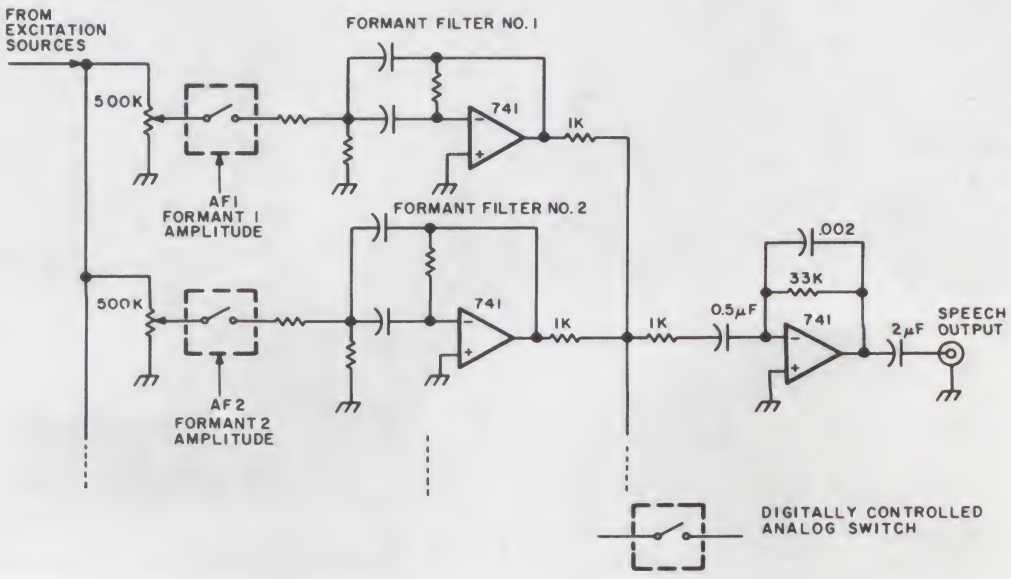
Figure 3: The excitation sources of the Ai Cybernetic Systems Model 1000 Speech Synthesizer. The rush of air through the vocal passages is simulated in the upper branch while the action of the larynx is simulated in the lower branch.

number of alternate phonemes, especially after 20 to 30 minutes of intense listening.

Once the design is experimentally determined, careful procedures must be followed to insure that when the circuit is duplicated, it produces each phoneme properly. This means precision components must be used, as small changes in values can make the difference between moderately distinct speech and a fairly mushy speech.

Analog simulation of the vocal tract is the only method of true speech synthesis known. A popular alternate method of speech production (actually, reproduction) is the storage of digitized speech in a ROM. When the stored information is clocked out of the ROM at the proper rate and smoothed by a low pass filter, the generated speech can be quite clear and distinct. But it is important to note that this is not synthesized speech. In effect, this method is no different than any other method of recording speech. Yet, the method does have the advantage of producing readily understood words by a

Figure 4: The parallel filter network of the Model 1000. The filter frequencies and quality factors chosen depend on the number of filters used to divide the voice frequency spectrum. Ideally, the center frequencies of the filters should lie somewhere near the commonly occurring formant frequencies.



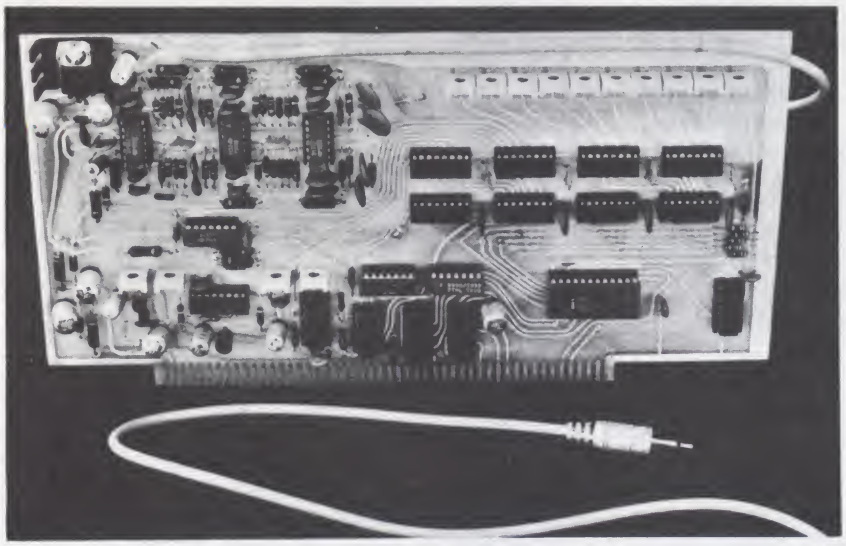


Photo 1: The Ai Cybernetic Systems Model 1000 Speech Synthesizer. The synthesizer is primarily an analog circuit controlled digitally. Ten active filters composed of 15 operational amplifiers are mounted in the upper left corner of the board. Directly beneath these resonant-cavity simulating filters are the vocal excitation circuits. The right half of the board is composed of the ASCII character decoding circuits and phoneme memories. Four 32 x 8 ROMs control the 16 analog switches to select the proper combination of circuits to generate any one phoneme. A device-busy flag is returned for the duration of the phoneme.

computer or calculator. However, the vocabulary is totally predefined and must remain small due to the high cost of storing this kind of generated speech. Moreover, the repertoire of this kind of speech is limited to the person who initially spoke the recorded words.

Synthetic speech, on the other hand, is generally not as clear and distinct. The proper transitions from phoneme to phoneme, the automatic emphasis given to leading or terminating consonants, and the intonation of a rhythm in speech which is associated with a word's importance or placement, are all facets of human speech which are difficult to properly recreate in machine produced speech. The determination of accurate rules to account for these factors has been the subject of active and intense research at centers here, and in Europe and Japan, including Bell Telephone Laboratories, the Haskins Laboratories of New York, the Royal Institute of Technology in Sweden, and the Musashino Electrical Communication Laboratory in Tokyo. On the whole, totally satisfactory rules have not yet been worked out although a great deal of progress has been made in the last 20 years. Machines which do incorporate the known rules quickly become elaborate and expensive (in the tens of thousands of dollars).

Simplified speech rules can be incorporated in a much smaller machine, but the burden of intelligibility now falls upon the listener. The produced speech is not natural speech. It sounds for all the world like the speech produced by the robots of 1950s grade B science fiction movies. But it is intelligible and it is quickly learned. Because the machine pronounces every phoneme in the same fashion each time it occurs, a listener quickly gains a feeling for the speech. The process is not unlike learning to listen to a newly-arrived foreigner who possesses a strong accent. The fashion by which he mispronounces the English phonemes is quickly learned and intelligibility increases rapidly. The difference with synthetic speech is that the speech is truly an alien form of speech, not often heard before by many of us.

As to the naturalness of synthetic speech, M D McIlroy of Bell Telephone Labs wrote this in 1974 *[in "Synthetic English Speech by Rule," Computer Science Technical Report No. 14, Bell Telephone Laboratories]:*

> The Computer Science Center at this laboratory has experimented with an inexpensive speech synthesizer [presumed to be the Votrax] as a regular output device in a general purpose computing system. Our intention was not to do speech research or to create artificial speech as an end in itself. In the present state of the art, those goals require much more elaborate facilities than we have at our disposal.
>
> We wished to see what uses might evolve when speech became available more or less on a par with printed output. For this goal, "naturalness" was not a prerequisite, any more than it is for printed output. Most computers still print mainly in upper case, are incapable of printing mathematical notation, and normally produce cryptic codes or tabular stuff that require considerable indulgence to be understood. Since printed gobbledygook is so widely accepted from computers — and fed into them, witness any manufacturer's operating system manual — we suspected that spoken gobbledygook might be quite passable, too, except for one severe difficulty: Being ephemeral, sounds must be understood at first hearing. As it turns out, long speeches *are* hard to understand, as are extremely short utterances of very simple words out of context. But given a little familiarity

with the machine's "accent", one finds short sentences to be quite intelligible.

The phonemes generated by the Model 1000 synthesizer appear in table 1. Each phoneme has been assigned an ASCII character to represent its particular sound. The assignment was done in the most intuitive manner possible; the consonants are generally the consonants as they appear on the keyboard, but there are many more vowels than a, e, i, o and u. Non-alphanumeric characters were chosen to represent the remaining vowels and consonants in such a manner that they could be easily associated with their sound. As examples of this, the number symbol, "#" is used to signify the vowel *er* as in numb*er*, "&" for the vowel *ae* as in *a*nd *"(" for ah* and ")" for *aw*

Table 1: List of Phonemes.

	Phoneme	ASCII Symbol	Usage
Vowels:			
	a	A	pace, bay
	ae	&	and, Altair
	ah	(	father, all
	aw	)	bought, robot
	e	E	see, harmony
	eh	'	excessive, ten
	er	#	number, bird
	i	I	hit, six
	o	O	Mexico, over
	oo	U	too, sue
	uh	!	the, computer
	^	†	putt, up
Semi-Vowels:			
	w	W	water, wind
	y	Y	yaw, yacht
Plosives:			
	p	P	pop, deep
	k	K	computer, Atlantic
	t	T	top, pot
	b	B	boy, bird
	d	D	dog, died
	g	G	go, great
Fricatives:			
	f	F	puff, food
	h	H	how, had
	s	S	saw, miss
	v	V	David, vow
	sh	/	slash, shoot
	th	+	thaw, Earth
	z	Z	zero, is
Liquids:			
	1	L	low, all
	r	R	row, round
Nasals:			
	m	M	miss, am
	n	N	now, nine
Others:			
	Glottal Stop	.	The pause associated with aspiration
	Draw Bar	-	An extended vowel with decay
	Pause	(space)	Normal word spacing

representing the position of the tongue when these vowels are spoken, "!" for the sharp sound of *uh*, "+" for the fricative consonant *th* as in *th*aw, and "/" for the *sh* in sla*sh*.

The Model 1000 accepts a string of ASCII characters as if it were a normal printing device. Read only memories on the board convert the incoming ASCII symbol into specific control information which in turn determines the vocal source, duration and frequency content of the spoken phoneme. Less than 50 bytes of machine code or 8 lines of the typical BASIC are all that is required to generate a subroutine to accept a string of characters and output it character-by-character to the synthesizer.

For example, to write the phrase "I am a talking robot" on a printer or display peripheral, an ASCII character string is set up and sent to the output device. In BASIC, if C$ is the argument of the output subroutine, the setup would be:

C$ = "I AM A TALKING ROBOT."

To have the synthesizer say the same phrase, the setup for the phonetic output routine with argument P$ might be:

P$ = "&IE AM AE T). .KEN- RO.B). .T"

(The ASCII symbols are taken from table 1.) The long vowels I and A occur in this passage. As a rule, most of the long vowels are not really vowels at all but rather diphthongs composed of a sequence of pure vowels. Pronounce out loud each of the phonemes in the phrase above, referring to table 1 as necessary. Remember that each phoneme has only one specific sound. Playing the part of a synthesizer yourself, you will find that you can say any English word with the phonemes of table 1.

Programming the Model 1000 synthesizer is easy once you actually begin to listen to what you say and learn to rely less on how a word is written. English is a hodge podge of languages and carries with it all the alternate symbolisms of the pronunciations of its root languages. Purely phonetic languages such as the Polynesian languages of Samoa or Tonga could be made to be spoken almost as they are written. This is unfortunately not true of English; homonyms such as "won" and "one" and "two", "too" and "to" abound.

Generally, only one phonetic spelling exists for any one word regardless of the number of alternate written spellings. It becomes important to identify the sounds that you actually are saying when a word is pronounced. The word "one" is phoneticized using the phonemes of table 1 as W!N in similarity to the word "won"; "two" is programmed as TOU- more as if it were the

written word "too". For most Americans, there is no difference in the way these words are pronounced.

Proceeding in the same fashion, the remaining numbers up to ten are typed in as:

T+#E- FO#- F&IE..V SI..KZ
S'-VIN AE..T N&IEN T'N

Again, pronounce these phonetic spellings to yourself. As you will discover, phonetic spellings are quickly deduced and learned.

In a very short period of time, it becomes possible to make the machine say anything. At that point, conversational computing takes on a whole new meaning. Interactive computing will never again be the same once your computer has actually spoken to you.■

BIBLIOGRAPHY

1. *Speech Synthesis, Benchmark Papers In Acoustics,* 1973. J L Flanagan and L R Rabiner, eds. Dowden, Hutchinson and Ross, Stroudsburg PA. A collection of the best papers on speech synthesis over the past 35 years.
2. "Synthetic Voices for Computers," 1970. J L Flanagan, C H Coker, L R Rabiner, R W Schater, N Umeda in IEEE *Spectrum* 7:22-45. An authoritative overview of the speech synthesis procedure.
3. "The Synthesis of Speech," 1972. J L Flanagan, *Scientific American* 226:48-58. A simplified rework of the IEEE *Spectrum* article above.
4. *IEEE 1974 Speech Recognition,* Proceedings, 1974. L Erman, ed. IEEE, NY. A bit too technical for a first introduction but a good measure of where things are going.

COMMERCIAL PRODUCTS

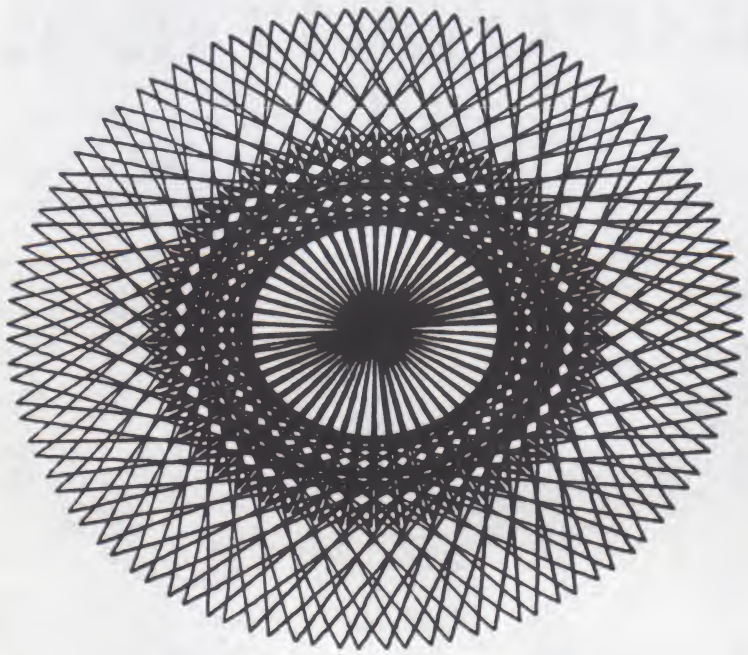
At the present time, two speech synthesizers are both commercially available and affordable by the hobbyist. One is the Votrax produced by:

Vocal Interface Division
Federal Screw Works
500 Stephenson Dr
Troy MI 48084
Price, approximately $2,000
Interfacing: Parallel or Serial (RS-232)

The second is the Model 1000 manufactured by:

Ai Cybernetic Systems
PO Box 4691
University Park NM 88003
Price, $425
Interfacing: Electrically and mechanically compatible with Altair/IMSAI/ Poly-88 bus structure.

Either company will be pleased to provide literature free of charge. A demonstration tape is available from Ai Cybernetic Systems for $5 and a complete programming guide, theory of operation manual and phonetic glossary is available for $2.50.



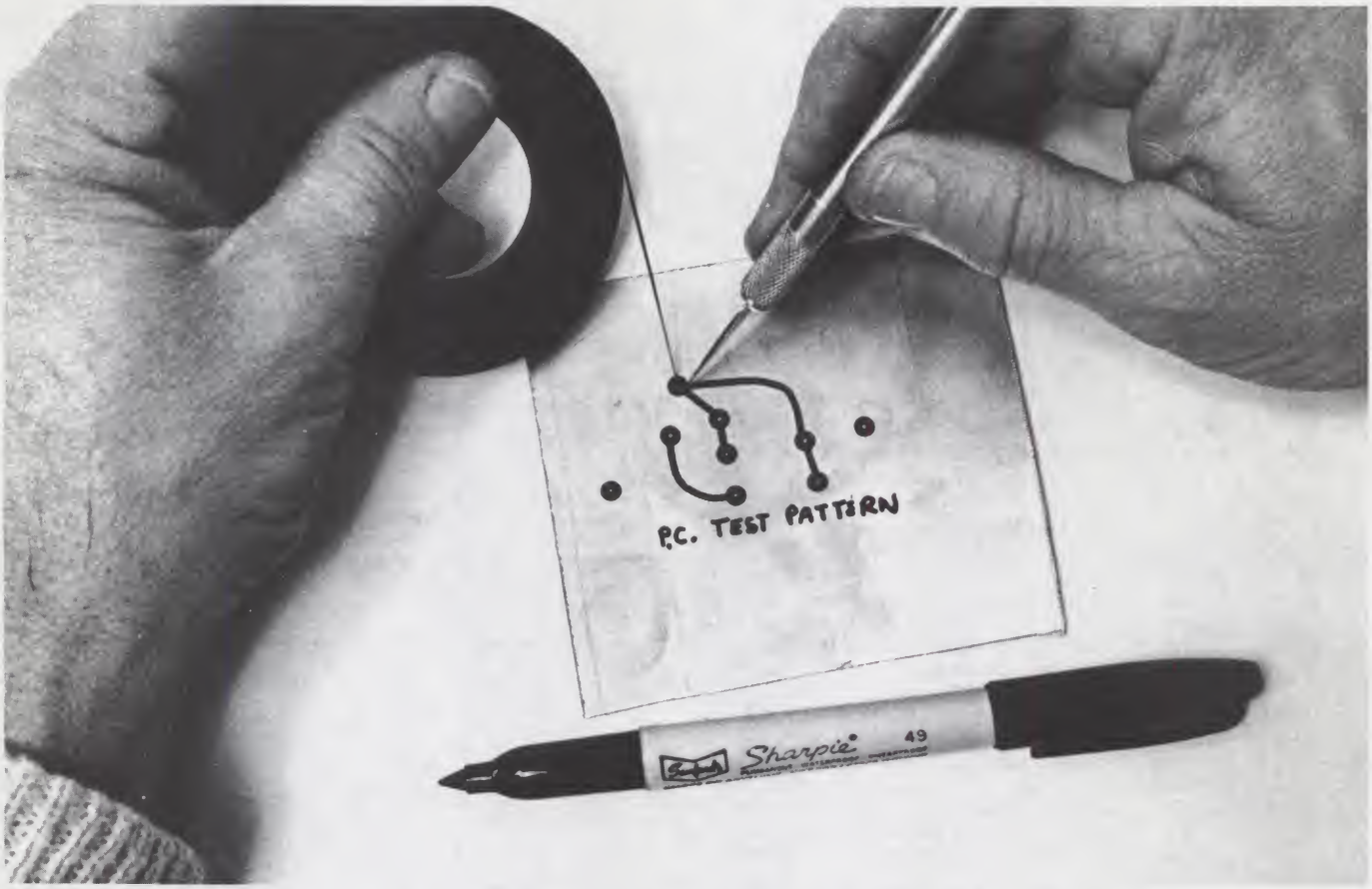


Photo 1: The Direct Etch Method. In this method, a one of a kind printed circuit is made by putting the pattern directly onto the copper. A Sanford's "Sharpie" pen (available in most stationery stores) can be used to draw patterns directly, and tape resist can be used for more uniform runs. If tape resist is used, care should be taken to avoid gaps in the adhesion of the tape to the copper.

Make Your Own Printed Circuits

James Hogenson
Box 295
Halsted MN 56548

Photos accompanying this article are by Ed Crabtree, using materials supplied by the author.

The widespread commercial use of printed circuits in electronic equipment began a few decades back when engineers started looking for more efficient wiring techniques to replace laborious hand-wiring methods. One of the first methods tried was to deposit (in other words, to print) a conductive ink pattern on a base of insulating material. The original method, printing, gave its name to all subsequent methods. Today, the term printed circuit refers to any electrical circuit in which individual wire lead connections have been replaced by a two dimensional conductive pattern bonded to an insulating base material.

Contemporary printed circuits consist of etched copper foil wiring patterns bonded to any of several insulating substrate materials sturdy enough to serve as a mounting base for the actual electrical components which make up the circuit. Although originally developed for mass production applications, printed circuit fabrication techniques have been refined until they can now be used by almost anyone with average mechanical skills.

Choosing your base material, the board, is a matter of price and purpose. The best is the epoxy glass board while phenolic (bakelite) is the cheapest. Phenolic base material is perfectly adequate for many applications, but since small boards are relatively inexpensive, epoxy glass is usually the optimum choice. The base material often comes

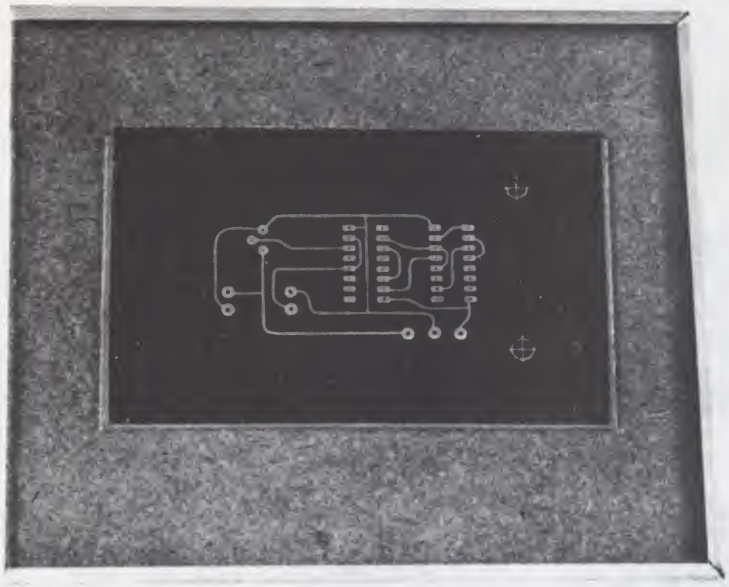


Photo 2: Printing the Circuit. Once a negative of the artwork has been created, the next step is to print the circuit. The negative is placed over a sensitized PC board and held firmly in place by a glass cover plate in the printing frame. The glass guarantees smooth and even contact for accurate transfer of the image. The board is then exposed to a photoflood lamp for one to three minutes.

laminated with copper foil on one or both sides.

The toughest part of making your first printed circuit board is getting started. In other words, the process may not be as difficult as you had thought.

A pattern of etch resist is applied by one of several methods to the copper foil. The board is then immersed in a chemical solution (usually a ferric chloride solution) which etches away all exposed copper. Then the board is washed and the etch resist pattern removed. The copper foil that was covered by etch resist remains on the board to provide you with a printed circuit.

Plan the Layout

The first step toward making your own printed circuit board is planning the layout. Draw the circuit pattern on paper as it should appear on the printed circuit board. You will use this as a guide for laying out the actual etch resist pattern. Keep in mind that you are looking at your board from the bottom when looking at the foil side. Be careful not to put the pattern on the printed circuit board upside down. (I've made that mistake more than once!)

Direct Etch

Direct resist is a method often used when a one of a kind board pattern is needed. Dry transfer etch resistant patterns are applied directly to the copper. The dry transfer patterns form integrated circuit pads, transistor pads, edge connectors, round donut pads, etc. Narrow etch resistant tape is applied to complete the circuit path between

A printed circuit is any electrical circuit in which individual wire leads have been replaced by a two dimensional conductive pattern bonded to an insulating base material.

The toughest part about making your own printed circuits is getting started.

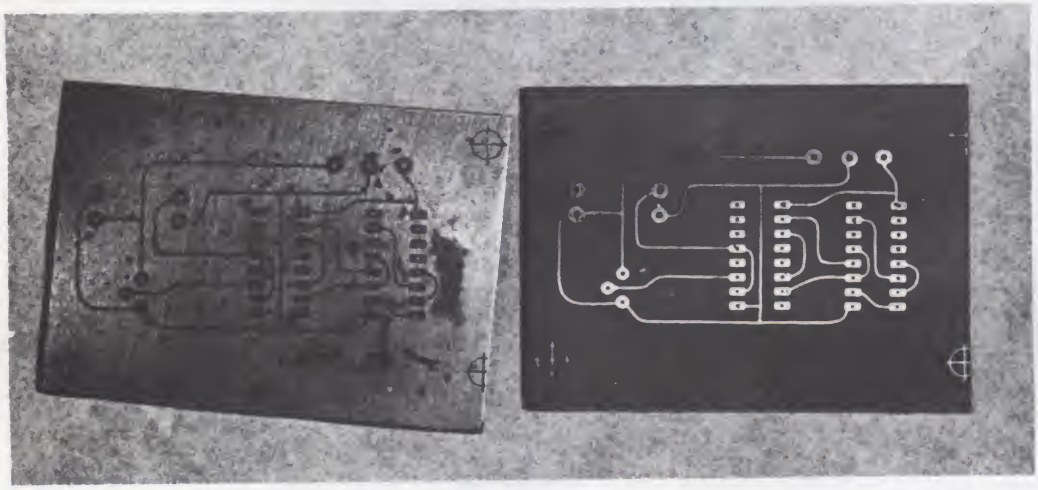


Photo 3: Results of Exposure. After being exposed, the photosensitive layer is developed, using an appropriate solution. An etch resist pattern will then remain on the board as in the example at left. (The dark blotches are oxidation on the copper.) The board is then etched with the usual ferric chloride solution. The finished product (hopefully free of imperfections) is a printed circuit board such as the one at right.

Photo 4: The "Cut-N-Peel" Method. A sheet of red mylar film on clear acetate backing is placed over the pattern to be copied. The negative is made by carefully tracing the pattern with a razor or sharp knife, then removing the red film wherever component pads and connections are to be made. (A trade name for the film used in this method is "Rubylith.") The negative is then transferred to sensitized copper and etched.

Photo 5: The Bishop Graphics " 'B' Neg" Method. In this method, a negative is made directly, using self adhesive black patterns on a mylar backing. The connections between patterns are made by cutting away the black layer with a sharp knife as in the "Cut-N-Peel" method.

component pads. Etch resistant ink pens and resist paint are also available for direct etching, as illustrated in photo 1.

After etching, copper will remain on the board only where dry transfer patterns or resist ink protected the copper foil from the etching solution. It should be noted that the etch resistant tape must be applied firmly, especially at overlaps, to keep the etching solution from getting under the tape and breaking the conductive copper path.

The direct resist method does not require extra steps for developing, as does the photo etch method. If only one printed circuit board is going to be made from a pattern, the direct etch method may be a time saver. If more than one board is to be made from one pattern, the direct etch method will quickly turn the element of time against you, since the pattern must be reconstructed on each board.

Photo Etching

The photo resist method is the most efficient method for making more than one printed circuit board of a kind. Photo resist etching is probably the most popular method, and is often preferred even for one of a kind printed circuit boards. The difference between photo resist and direct resist is the way the resist pattern is applied.

The copperclad board to be photo etched is first sprayed with a thin coat of a photo sensitive etch resist. This etch resist is sensitive to ultraviolet light. The sensitized board must be handled in a darkened room using a yellow light for illumination.

After the resist is dry, a negative of the printed circuit pattern is placed over the sensitized board in a print frame, as shown in photo 2. The board is exposed to the light of a photo flood lamp through the negative for one to three minutes. It is then immersed in a resist developer solution for about one minute. Only the etch resist which was exposed to the bright light will remain on the copper foil, as in photo 3. The resist is no longer light sensitive after developing, but should be allowed to dry for a short time. The board may then be etched. The copper which is protected by the remaining etch resist will not be removed. After the board has been etched, the resist is removed and the board may be cut, drilled, and assembled.

Making Negatives

It is plain to see that exposing a board through a reusable negative is much simpler than reconstructing the pattern by hand each time the pattern is used. The negative

may be obtained by a number of methods.

If a pattern is not too complex, the "Cut-N-Peel" method of photo 4 can be used. The pattern is simply cut into a red film on a clear acetate backing. The red film is peeled off, leaving a negative of the pattern.

If the pattern involves integrated circuits, the "Cut-N-Peel" method becomes rather difficult. The Bishop Graphics " 'B' Neg" ™ method would be more suitable. The ready made negative component patterns are laid out on a mylar sheet according to desired component placement. The areas between these self adhesive patterns are blacked out, using solid black acetate film. The only cutting necessary is for connections between component patterns. Photo 5 illustrates this method. The finished product is a negative of the entire printed circuit pattern.

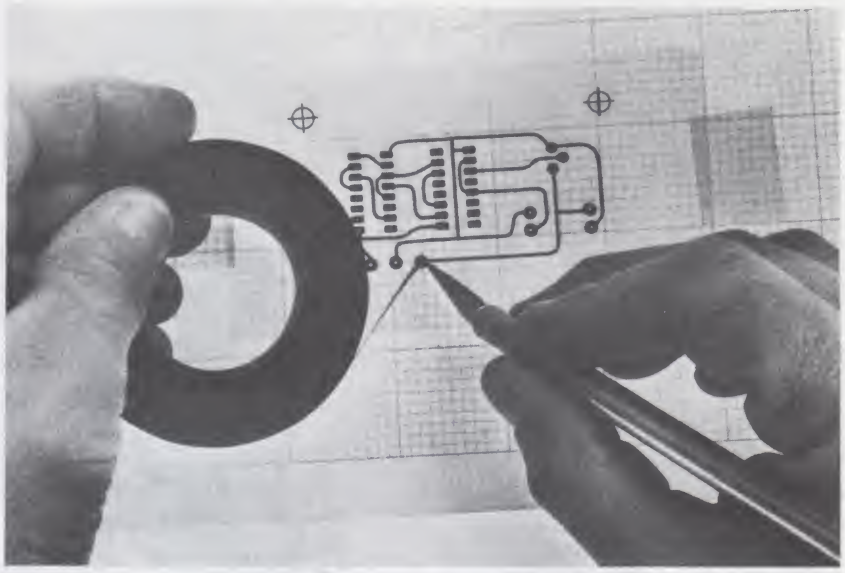


Photo 6: The Traditional Photo Negative Method. In this method, a positive artwork pattern is created, using preprinted self adhesive patterns and artwork tape. A sharp knife is used to cut the tape as it is being applied to the mylar film backing. A photographic process must be used to invert the image and create the negative form (see photo 5).

Photographic Negatives

Perhaps the easiest and certainly the most popular method of obtaining the necessary negative is to first make a positive pattern, then produce a negative by photographic methods.

Positive artwork is made on a sheet of clear mylar film with matte finish on one side. This film is dimensionally stable and similar to plastic drafting film used by draftsmen. Positive artwork patterns are widely available in a large number of sizes and shapes. Photo 6 shows an example of a circuit being laid out with these patterns. Unless the artwork is going to be photographically reduced, use 1:1 artwork patterns. The self adhesive positive artwork patterns are laid out on the mylar sheet according to your pencil layout. Narrow black tape is used to form conductive paths between components. Graph paper or a similar grid should be used as a guide for orderly and uniform positioning of patterns. Since components are normally configured for dimensions which are multiples of 0.1 inch a 0.1 inch grid should be used.

A negative reproduction of your positive pattern can be made by a photographer or (preferably) by you. If you enjoy experimenting with photography, you might try experimenting with lithographic and orthographic films.

Photography Without a Darkroom?

The most popular negative producing method does not require photographic darkroom facilities. The special reversing film used may be handled in subdued light or in a darkened room using a dim yellow light. The positive pattern is placed directly on top of the reversing film. The film is exposed through the positive artwork pattern to a photo flood lamp for one to three minutes and developed by rubbing gently with a cotton swab and a little film developing solution, as shown in photo 7. The opaque or colored emulsion on the film will rub off areas not exposed to light. The result is a clear pattern on a dark background.

A somewhat more involved but rather unique artwork developing system is made by Datak. With the Datak film and developing solutions, any of the following can be made: (1) negative from film positive or original artwork, (2) film positive from negative, (3) negative from negative, (4) film positive from film positive or original artwork, (5) film positive from black image on white paper, (6) film negative from black image on white paper. The last two methods allow you to copy a printed circuit pattern directly from a magazine page.

The Datak film is developed by methods similar to standard photographic procedures, so this method is more complicated and time consuming. Exposure and developing times are somewhat more critical. Datak film may, however, be handled in subdued tungsten light.

For a one shot printed circuit, simply draw the pattern onto copper with a resist pen and dump the board into ferric chloride until done.

Advantages of Photo Resist Techniques

One of several advantages in using the photo resist method will become apparent when a modification of an existing board is

Photo 7: Creating a Photo Negative for Etching. A negative is reproduced from the positive artwork pattern, using a reversing film. The film is exposed with a bright light, then developed by rubbing gently with a cotton swab and developing solution. The result is a negative version of the artwork with a 1:1 scaling.

For a unique approach to making jumpers on one layer boards, see Don Lancaster's "How to Build a Memory With One Layer Printed Circuits" in the April 1976 BYTE, page 28.

made. (Like when you need to make a board over because you forgot two or three connections. This **does** happen!) Rather than reconstructing an entire printed circuit pattern, make only the necessary changes or additions on the original artwork, then make a new negative and a new board. Making a new negative using reversing film requires only a few minutes of your time.

Double Sided Boards?

Sometimes a circuit will be too complex to fit on one side of a circuit board. Since a printed circuit is only two dimensional, conductor paths cannot cross. Jumper wires can be used to provide some crossovers, but if the circuit requires a large number of crossovers, a double sided circuit board might be considered. A double sided PC board is one which has a copper foil pattern on each side. The major consideration in making a double sided PC board is getting the pattern and terminals lined up. Both sides of the board are developed and etched at the same time.

Drilling

The step following the fabrication of a PC board is drilling out the holes. A small bench type drill press is ideal for this purpose. A standard hand held drill is unsatisfactory as the small drill bits break at low speeds. Commercially, small holes are drilled in boards at speeds as high as 70,000 RPM. A Dremel "Moto-tool" is a suitable compromise for work on printed circuit boards. This tool runs at 30,000 RPM. Such a tool will not only drill out extremely small holes, but cut and shape printed circuit boards, and lend itself to a host of other uses not related to making boards. A multipurpose tool like this is handy, especially for cutting out things like board edge connectors.

If repairs or small changes are needed on a printed circuit board, a piece of bare wire soldered over the foil is the cheapest and quickest modification. A conductive silver paint is available for printed circuit repairs, but the paint is quite expensive. GC Electronics, Techniques Inc, Kepro, and Datak each manufacture printed supplies for the

hobbyist in addition to their commercial products. Such supplies are distributed through a large number of mail order firms and retailers. The appendix lists the various products and who makes them. Cost of materials will vary depending upon a number of factors, but a figure of 20 cents per square inch of printed circuit board will provide a good rule of thumb to estimate the cost per board.

You will notice that Techniques and Kepro do not manufacture photo resist spray. Instead, they sell printed circuit board panels with the photo resist already applied. Presensitized panels (which come wrapped individually in dark paper) will assure you of a uniform and dustfree coating of photo resist. However, if you make a mistake developing the resist pattern, you will waste the extra cost of presensitized panels. It is a good idea to start with a spray resist, then graduate to presensitized panels once you have refined your circuit fabrication techniques. And the keys to refining your techniques are: Read instructions and familiarize yourself with what you're doing, follow the instructions, take your time, be careful, and practice first, using small sample boards. Follow those hints and you may surprise yourself with the fine boards you can turn out.■

APPENDIX: Sources of Supply

Direct etch materials

Ink resist is made by GC Electronics, Techniques, and Kepro.

Dry transfer resist patterns are made by Techniques, Datak, and Kepro.

An ordinary "Sanford's Sharpie" marking pen available for about 49 cents at any stationery store can be used as a resist pen.

Photo etch supplies

"Cut-N-Peel" and " 'B' Neg" supplies are distributed by GC electronics. (The " 'B' Neg" materials are manufactured by Bishop Graphics.)

Rubylith material, available at art supply houses, can also be used for cutting and peeling patterns.

Positive artwork patterns and supplies are made by Datak, Kepro, and Techniques. GC Electronics distributes artwork materials made by Bishop Graphics. Bishop Graphics materials are also distributed by independent distributors.

Photo etch supplies

Photo resist spray and developer are made by GC and Datak. Presensitized panels are distributed by Techniques and Kepro. Reversing film and developer are made by Techniques, Kepro, and Datak.

All of the above mentioned manufacturers make or distribute plain PC board panels (unsensitized) and etching solutions.

Photo flood lamps are available at photo supply houses. (Look for 375 Watt reflector flood lamp or No. 2 (EVB) Photoflood.)

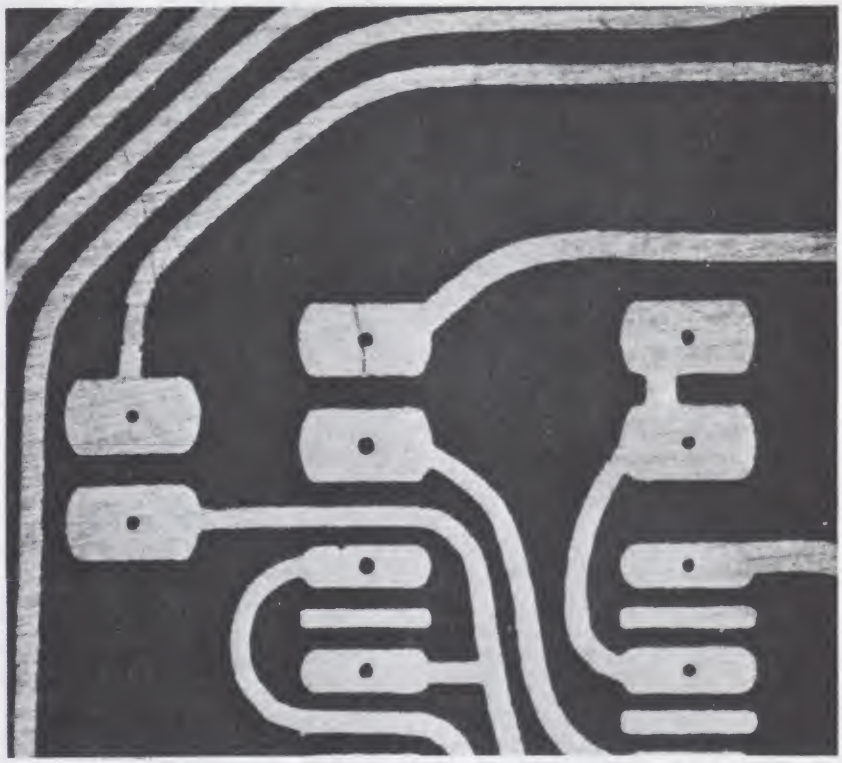


Photo 8: Close up, a successfully etched printed circuit will have even lines with no hairline cracks or other imperfections. This example shows such a result, prior to drilling out the holes for component leads.

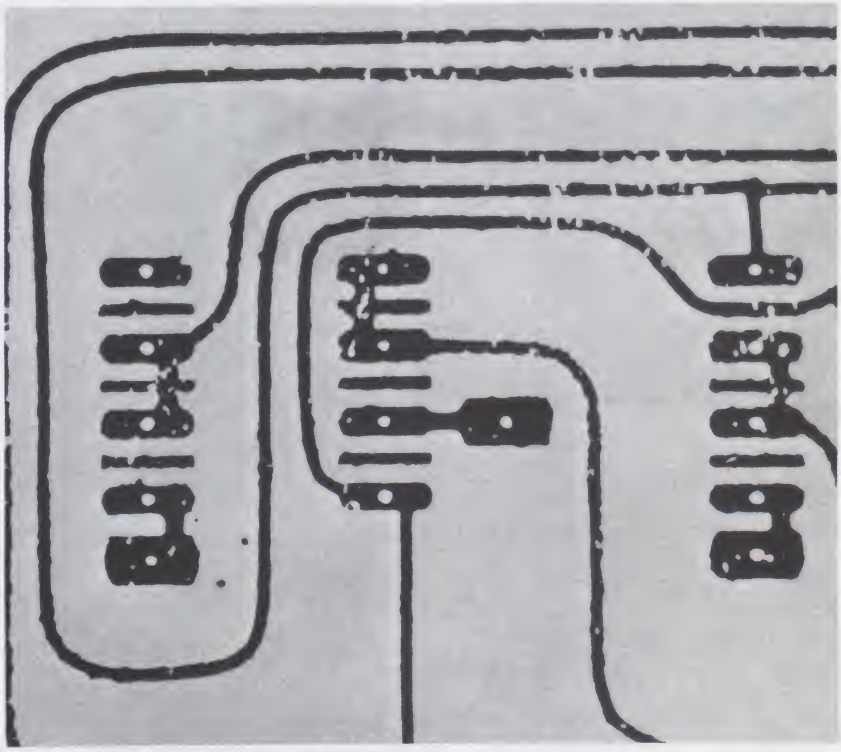


Photo 9: When various imperfections enter the picture, the result is not so clean. Here is a high contrast picture of an imperfect result. The resist layer has separated from the copper during the etch process at several points, resulting in holes in the copper and, in several instances, complete breaks in circuit runs.

Software

Write Your Own ASSEMBLER

To date I have not seen any detail descriptions of home brew self assembler systems for microcomputers such as the 8008, 8080, 6800 or PACE. Maybe Dan Fylstra's description of assemblers will start a few readers off in that direction. Dan describes in general terms what assemblers do, scanning techniques, symbol tables, hashing methods and some of the more advanced "bells and whistles" you might employ. Use Dan's article as a source of ideas on the organization and features for your own assembler software designs. . . . CARL

by
Dan Fylstra
HB
550 Memorial Drive
Cambridge
MA 02139

If you have done any work with microcomputers, you have doubtless seen programs written in assembly language. You probably know that assembly language programs must be translated into machine language before they can be executed on the computer. The translation is usually performed by another program, called an "assembler." Because assembly language lets you write mnemonic (easily remembered) names for instructions and data, rather than binary codes, programs may be written more quickly and with fewer errors. The assembler does the tedious job of putting together, or assembling, all of the right bit patterns to make up the program in machine language.

Most of the assemblers presently available for microcomputers are cross-assemblers: They run on big computers or time-sharing systems, and produce output which must be loaded in some way into the microcomputer system. Commercial time-sharing services are expensive, and the whole point of having a home computer is to be able to perform computing chores, such as program assembly, on your own system at ultra-low cost. Since a resident assembler — one which runs on your own micro system — may be unavailable or very costly, you might be interested in writing your own assembler. By doing it yourself you can learn a lot about programming and software design as well as the specs of your own microcomputer, save yourself the cost of program development, and produce a customized language suited to your own needs or fancies. And who knows? — you might even find that other hobbyists or microcomputer users might be willing to pay *you* for a copy of the assembler program that you had so much fun writing.

Now, it's only fair to warn you that writing an assembler is a big undertaking — you'll need a fair amount of time and perhaps some extra RAM chips to accommodate the finished product. But such obstacles have never stopped anyone with your boundless enthusiasm. So the only question is, how do you go about writing an assembler? That's the sort of question that BYTE magazine is designed to answer, and that's what this article is all about.

The assembler does the tedious job of putting together, or assembling, all of the right bit patterns to make up the program in machine language.

What Does an Assembler Do?

To answer this question, we have to take a look at some typical machine instructions and how they might be written in assembly language. A machine instruction usually consists of a binary code for some operation, such as addition, and one or more binary numbers denoting the "operands" of the operation. The binary number for an operand may have either of two interpretations: It may denote the binary value of, say, a number, or the ASCII code of a character, or it may denote the binary address of a memory location which holds the actual value of the operand. For example, on the Motorola 6800 the bit pattern

```
1000 1011   0011 0000
   ⌣           ⌣
 opcode      operand
```

means "add the number 48 (00110000 in binary) to the A accumulator." This might be represented in assembly language as

```
ADDA   #48
```

— note how much more convenient it is to write things this way! In contrast, the bit pattern:

```
1001 1011   0011 0000
   ⌣           ⌣
 opcode      operand
```

means "add the 8-bit number found in memory location 48 to the A accumulator." This might be written in assembly language as

```
ADDA   BETA
```

where it so happened that, just after the last instruction of a program which was 48 bytes long, the programmer had also written

```
BETA   RMB   1
```

meaning "reserve 1 memory byte at this point, and call it BETA."

These examples illustrate the basic functions of an assembler. In the first case, the instruction's operand was the actual number to be added. (This is often called an "immediate operand.") The assembler read the characters "ADDA" and substituted the proper binary opcode 10001011, and converted the decimal number 48 to its binary equivalent, 00110000. In the second case, the instruction's operand was the address of a memory location. The programmer called this memory location BETA and decided to put it after the instructions of a subroutine, which was to start at location 0. He indicated this by means of the mnemonic RMB, for "reserve memory byte." Since this assembly language statement doesn't actually represent an instruction, but instead tells the assembler what to do, it is often called a "pseudo-op." The assembler read the entire program, counting up the number of bytes that the subroutine would take, and determined that the address of the memory location called BETA was therefore 48, or 00110000 in binary. It assembled this address into the instruction.

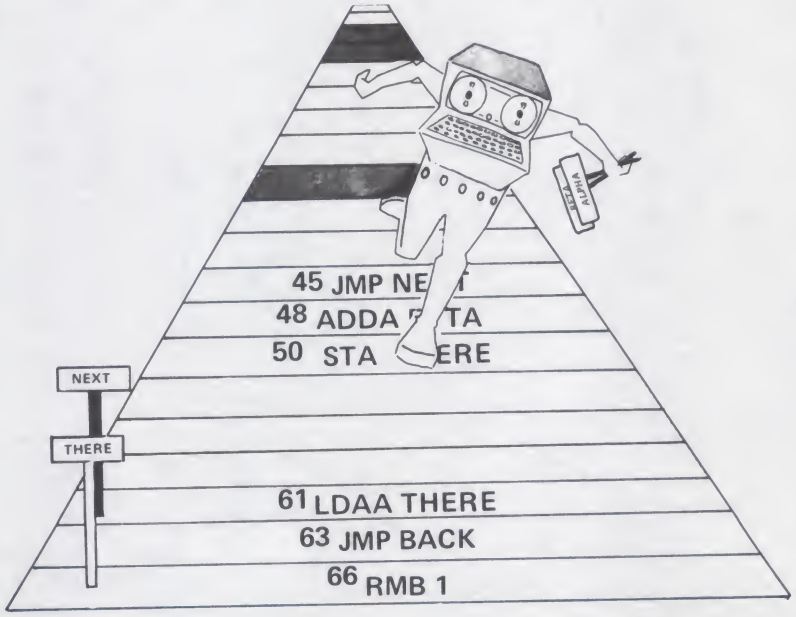
All well and good. But, being an alert reader, you ask, "Wait a minute! What if the ADDA instruction is in the middle of the subroutine? When the assembler reads the ADDA instruction, it doesn't know how many bytes the rest of the subroutine will take, and furthermore it doesn't know (yet) that the memory location BETA is supposed to be reserved just after the subroutine, since it hasn't seen the RMB pseudo-op."

Forward Reference

Fig. 1 illustrates a problem common to all assemblers and compilers, often called the "forward reference problem." There is no neat way out of it. In this case, the programmer could have reserved the memory location BETA *before the* subroutine rather than after it. But suppose that the subroutine had included a "jump" or "go to" statement:

```
        JMP    NEXT
        .
        .
        .
NEXT    ADDA   #7
```

Fig. 1. The Forward Reference Problem



A two-pass assembler solves the forward reference problem by reading the program twice.

It is rather impractical to try to write every program without any forward jumps!

There are basically two ways to cope with this problem. The first is to read the program once, but to keep sections of the program in memory until all forward references are resolved. Since RAM costs us money in a microcomputer system, we will reject this approach. The second alternative is to read the program twice; an assembler which adopts this strategy is called a two-pass assembler. This approach is slow, but it's also cheap, and that's what we want!

The first time that such an assembler reads the program (i.e., on the first pass), it simply looks at the instruction mnemonics, counts up the number of locations that each instruction will take, and builds a symbol table in memory which lists all of the programmer defined names for memory locations and their corresponding addresses. (We need RAM for this, but not so much as would be required for the first approach.) This process is (somewhat fancifully) illustrated in Fig. 2. Notice that the assembler picks up only the statement labels, ignoring (for the purposes of Pass 1) appearances of the same symbols in the operand fields of instructions.

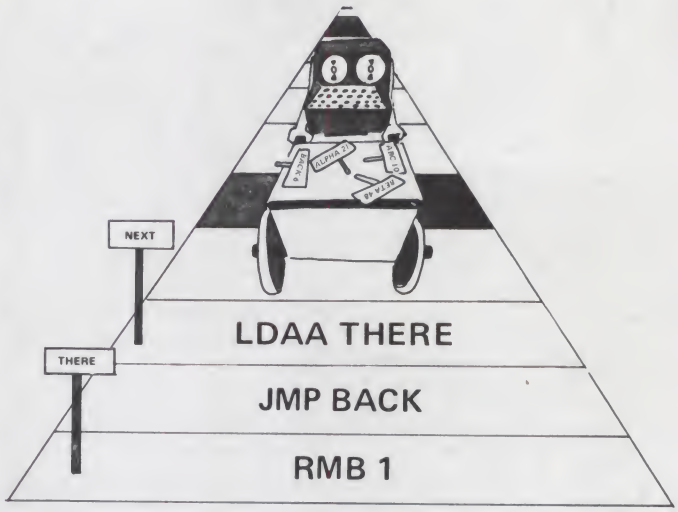
On the second pass, binary opcodes are substituted for the instruction mnemonics, constants are converted to their binary representation, and programmer defined names are replaced by their actual memory addresses, found in the symbol table. This is illustrated in Fig. 3. Any name appearing in the operand field of an instruction which is not already in the symbol table on Pass 2 is undefined in the program, and will cause an error message. One other note: looking up the binary opcode for an instruction mnemonic is essentially the same process as looking up the address for a programmer defined name, so the symbol table can be used for both purposes.

It should be pretty clear by now that an assembler spends most of its time 1) scanning characters, looking for names, numbers and punctuation symbols, and 2) building and searching the symbol table. If we can find simple and efficient ways of performing these operations, and avoid getting them hopelessly intertwined with the rest of the program logic, we should come out with a fairly decent assembler. So let's now take a look at programming techniques for scanning and searching symbol tables.

Scanning Techniques

Our assembler's first task is to scan the characters making up an assembly language program, and find things such as instruction mnemonics, constants and programmer defined names, while noticing but generally ignoring such things as blanks, punctuation symbols and comments. The amateur programmer's first impulse usually is to plunge in by writing a series of tests and branches to handle various

Fig. 2. PASS 1 picks up the labels.



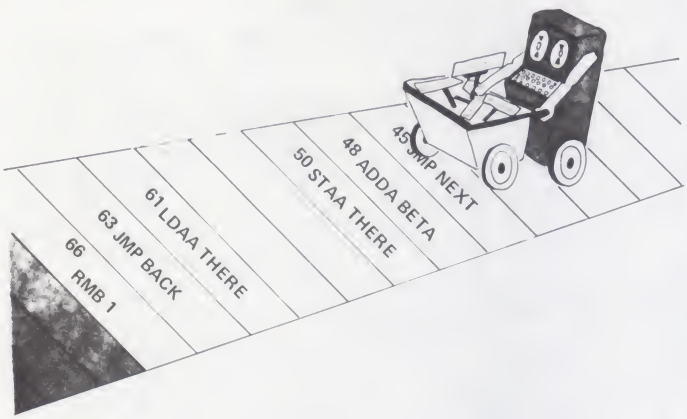
sequences of characters which may appear on a line. This approach frequently leads to the type of scanner known as a "kluge." The computer scientist, on the other hand, has nothing but contempt for this "ill-structured" approach, and prefers to work with regular expressions or right-linear grammars and finite automata. We will take a middle course, outlining some programming techniques that will help make a hand implemented scanner simpler, smaller and faster.

The first technique, if you are designing your own assembly language, is to make it simple to scan! An assembly language statement usually consists of an optional statement label (which then represents the address of the location into which the instruction is assembled), an instruction mnemonic, an operand field, and room for comments. Some assemblers require each element of an assembly language statement to begin in a fixed column or character position of a line, so that the problem of locating the elements for scanning is greatly simplified. However, this is a little rough on the assembly language user, and you will probably save yourself time in the long run by implementing a slightly more complex scanner. To permit a more flexible format, one may take either the "IBM approach," in which a statement label must begin in column 1, an instruction mnemonic must be preceded by at least one blank, and comments are separated from operands by a blank; or the "DEC approach," in which statement labels are followed by a colon (or other punctuation symbol), and comments are preceded by a semicolon. The "DEC approach" is somewhat more

A typical example would be:

EVAL	LDAA	BETA	BEGIN FUNCTION EVALUATION
statement label	instruction mnemonic	operand field	comments

Fig. 3. PASS 2 generates code referencing labels.



convenient and less error-prone for the user, but is slightly harder to analyze. For instance, one must be willing to scan a string of alphameric characters followed by blanks, waiting for a colon or an alphabetic character in order to decide whether the string was a statement label or an instruction mnemonic.

Sometimes a decision as to what to do next must be made on the basis of the type of the next (non-blank) character. If several alternatives are possible, one would like to use a "jump table," or an array of branch addresses indexed by the character code, instead of a sequence of character comparisons. But the ASCII character set allows for 128 different character codes, of which only about 45 are used in assembly language statements. Hence, a common technique for complex scanning problems is to first translate from ASCII to a more convenient set of character codes, using a 128 byte character translation table. The new character codes can be chosen so as to facilitate the use of jump tables at other points.

The elements of an assembly language statement (names, mnemonics and constants) generally consist of variable length character strings, separated by a variable number of blanks. Present-day computers, however, are more adept at handling fixed size objects such as bytes or words. So the most important technique you can use to keep your scanner coherent is to write a "next token" routine, which scans off an alphameric string, a constant (e.g., a string of digits) or a punctuation symbol each time it is called. This routine should return a *code* for the type of item or token just scanned (say, 1 for alphameric strings, 2 for digit strings, 3 for a colon, 4 for a comma, and so on), and a fixed-size *descriptor* giving the address of the first character and the number of characters in the string.

Fig. 4 illustrates descriptors for the statement label, instruction mnemonic, and operand of a typical assembly language statement.

Descriptors for character strings are handy for a number of purposes. Character string move and comparison routines can be written which take two descriptors as arguments. Output lines can be constructed from a sequence of descriptors, and error messages can also be handled in this way. By storing the fixed-size descriptors in the symbol table and the character strings themselves in another area, you can avoid the arbitrary restriction on the length of names to six or eight characters found in many assemblers.

Even more important, the use of a "next token" routine separates the details of scanning individual characters from the problem of deciding how to process each element of a statement. The symbol table routines described below similarly separate the details of identifying particular names and mnemonics from the other problems of processing. These are examples of the use of modularity and hierarchical structure to organize the solution of a complex problem.

Enough in the way of generalizations and philosophy; let's get on with an example to see how all this works. Fig. 5 shows the flow of information from a character code translation routine, to a next token routine, to a routine which determines the type of statement from the instruction mnemonic using a symbol table lookup subroutine. Assembly language for the Intel 8080 has been used in this example. Lower case letters are translated to upper case, and the codes for digits (0-9) and letters (A=10, B=11, . . . ,

Fig. 4. Descriptors Identify Text Tokens in a Line of Characters.

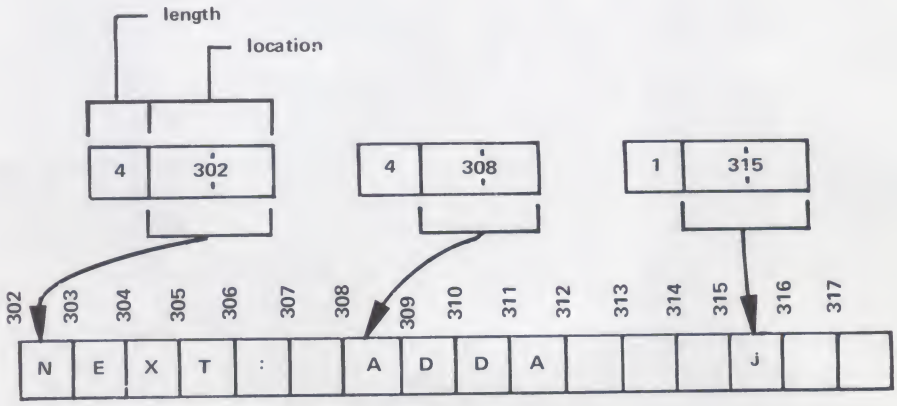
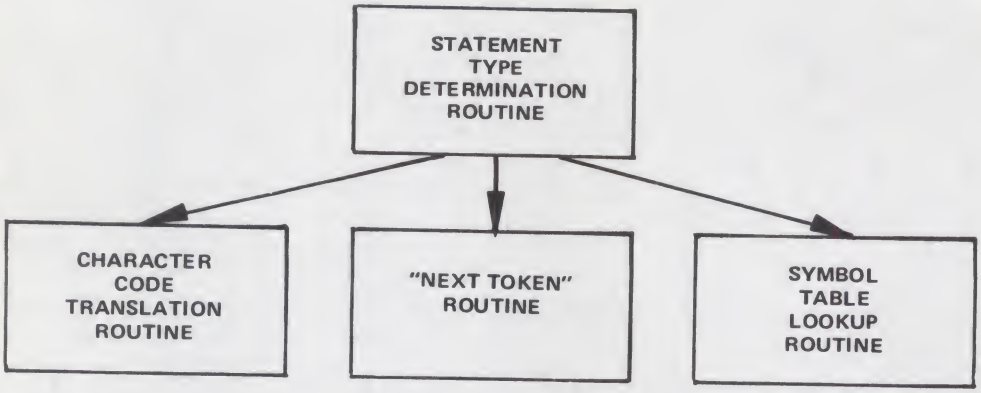


Fig. 5. Typical (8080) Character Translate and Next Token Routines



CHARACTER TRANSLATION ROUTINE

```
        MVI     H, TABLE    H → page holding table
        LXI     D, LINE     DE → begin of line
        MVI     C, 72       C = length of line
LOOP:   LDAX    D           get next char of line
        MOV     L, A        L = character code index
        MOV     A, M        A = table entry at index
        STAX    D           replace char in line
        INX     D           advance to next char
        DCR     C           reduce no. chars remaining
        JNZ     LOOP        loop for all 72 chars
```

"NEXT TOKEN" ROUTINE

```
        LXI     H, BRTAB    H → branch table base
        LDAX    D           get translated char from line
        RLC                 times 2 for branch table index
        MOV     C, A        set up 16-bit index
        MVI     B, Ø        in registers B and C
        DAD     B           add to branch table base
        PCHL                jump to appropriate routine
BRTAB:  JMP     LETTER
        JMP     DIGIT
        JMP     COLON
        JMP     COMMA
        ⋮
LETTER: XCHG                HL → begin of alpha string
        SHLD    DESCR + 1   put start addr in descriptor
        MVI     A, 36       max translated code for alphameric
        MVI     C, Ø        initialize count of chars in string
SCAN:   INX     H           advance to next character
        INR     C           increase character count
        CMP     M           code < max for Alphanumeric
        JP      SCAN        continue scan if so
        LXI     H, DESCR    HL → length part of descriptor
        MOV     M, C        put in no. chars in string
```

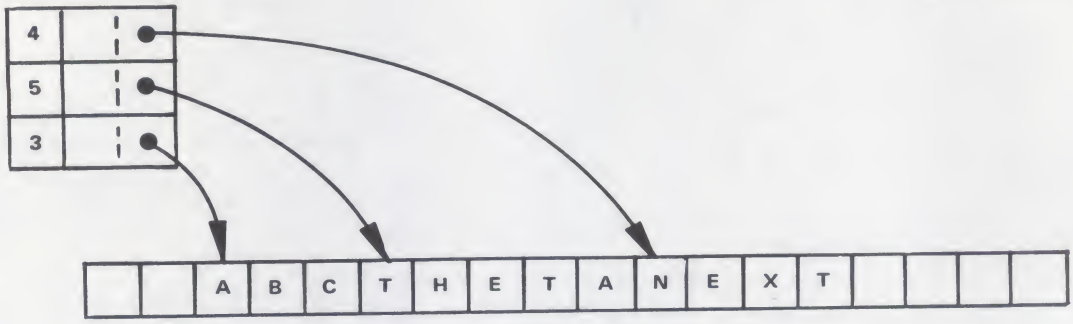
Z=35) are chosen so that alphameric and digit strings can be scanned off using a single comparison for each character. Note the use of a jump table "BRTAB" to select the appropriate handling routine for the next character in the next token routine. Descriptors are returned to the statement type determination routine, and are passed on to the symbol table lookup routine which uses them in character comparisons. The problem of distinguishing statement labels followed by a colon is handled easily at this level: The next token is obtained, and its descriptor is saved; the next token is obtained, and its code is tested; if a colon has been found, the saved descriptor is passed to the symbol table lookup routine, and two more tokens are obtained to balance things out before the instruction mnemonic is processed.

Symbol Tables

The greatest convenience that an assembler provides for the programmer is the ability to give names to memory locations and to refer to those names from other points in the program. The assembler determines the proper address of the memory location, and fills in the address wherever the name is referenced.

The assembler accomplishes this by building a symbol table on its first pass. Each entry of the symbol table contains a programmer defined name in character string form, and the binary address corresponding to it. In addition, the symbol table may contain other character string names, such as the instruction mnemonics or assembler pseudo-ops. The entry for an instruction mnemonic would contain the corresponding binary opcode, and the entry for a pseudo-op might contain the address of a processing routine in the

Fig. 6. An Array Symbol Table.



assembler itself. For a computer with several different instruction formats, the entry for an instruction mnemonic might also contain a type code indicating the proper format for this instruction, the number of operands expected, and the interpretation of the operands as addresses or values.

The simplest way of organizing the symbol table would be as an array of descriptors and address words, as illustrated in Fig. 6. Entries are added sequentially to the array during Pass 1. and a sequential search of the whole array is used to find the addresses of programmer defined names during Pass 2. (Each descriptor from the table is passed in turn to a character comparison routine, along with the descriptor for an operand. The comparison fails immediately if the string lengths in the descriptors were unequal.) This type of organization has the great virtue of simplicity, and is probably adequate for a first version of your own assembler. As the programs to be assembled get longer, however, the asembler will spend an increasing fraction of its time searching the symbol table. A faster way of searching the table is needed.

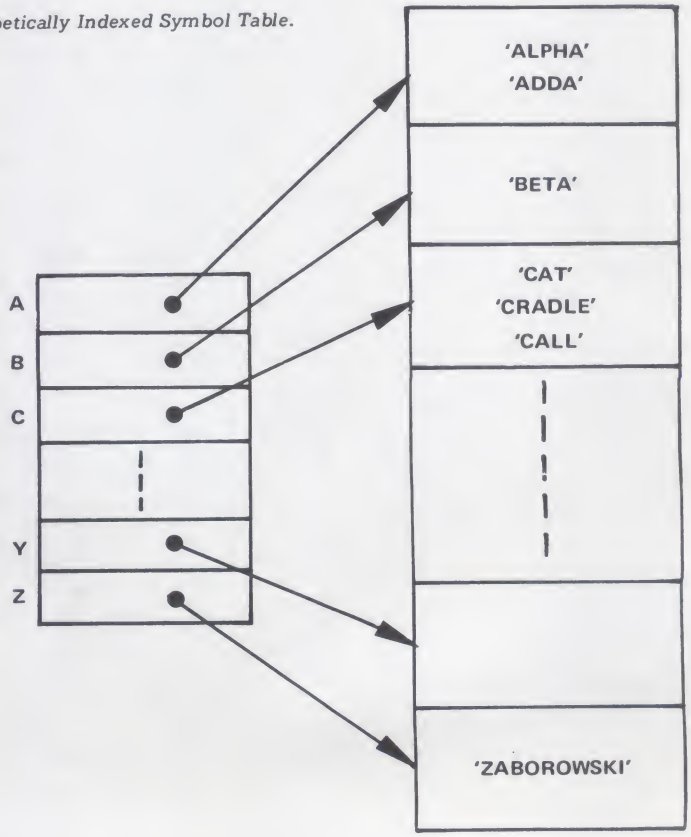
Think about how you would go about such a search, if you were the assembler. What do you do when you open a dictionary or a telephone book? Knowing the order of the alphabet and the thickness of the book, you look at the first character or two of the word, you make a guess at the approximate page, open the book to that page, and begin searching from that point.

Let's have the assembler do the same sort of thing. We will divide up the table into twenty-six sections, one for names beginning with each letter of the alphabet. We know the starting address of each section of the table (we can make a small array of the twenty-six starting addresses), so to look up a name, we look at its first character, go to the appropriate section and search just that section rather than the whole table.

This approach is depicted in Fig. 7.

This is a good first try, but there are some drawbacks. In a program called "assembler," say, you might have a lot of names beginning with A, while in a program called "editor," you might have many names starting with E. On the other hand, your friend Zaborowski might start *all* of his names with Z. Should all of the sections be of the same size? If not, how do you know (at the beginning of an assembly) which sections to make larger? If a section becomes filled, we can simply add the extra names to the next section of the table; now,

Fig. 7. An Alphabetically Indexed Symbol Table.



What happens if we use a random assortment of the names, placing them haphazardly into the various sections of the table?

however, if a name to be looked up on Pass 2 is not found in its original section, most of the following section will have to be searched before the name is found. This phenomenon is called "clustering." Your friend Zaborowski is especially likely to run into this problem, and even if you make the Z section large enough, searching the symbol table will take just as long using the new approach as it did with the old one.

Can we overcome these drawbacks of the new method? Here's where a little lateral thinking will help. We are making use of our knowledge of the ordering of the alphabet. Try the opposite approach: What happens if we use a random assortment of the names, placing them haphazardly into the various sections of the table? At first this sounds absurd, but on closer examination we realize that it *solves the problem!* The problem arose because people are fairly likely to choose a set of names which are related in the alphabetic ordering; by using a randomly chosen ordering, we can minimize the likelihood that a large number of symbols will be placed in a single section of the table. This technique, which is called "hashing" or "hash addressing" for obvious reasons, is used in most modern assemblers and compilers.

So, instead of using the first character of a name to select the proper section of the table, we will use a random assortment of bits, or an arbitrary function of the bit pattern of the entire name, to select a starting point in the table. A function of this sort is called a "hash function". So long as the function's possible values are evenly distributed over the range of addresses for table entries, the problem of the "clustering" or grouping of names will be minimized.

An example of a hash function which usually gives good results is to add together all of the bytes of a character string, ignoring overflow, or else to "exclusive or" the bytes together.

Similarly, in order to minimize the clustering of names which hash to the same starting address, we can "re-hash" the names so as to randomly distribute them around the table. Such a method is called a "random rehash." The following method is easy to implement, efficient, and works well when the table size is a power of 2, say 2**k (see Morris): Suppose a name initially hashes to table entry h, which is already occupied by another name. Initialize a variable R to 1. To rehash the name:

1. Set R = R*5 (shift left two bits, and add to the original number).

2. Mask out all but the low-order k+2 bits of R, and save this as the new R.

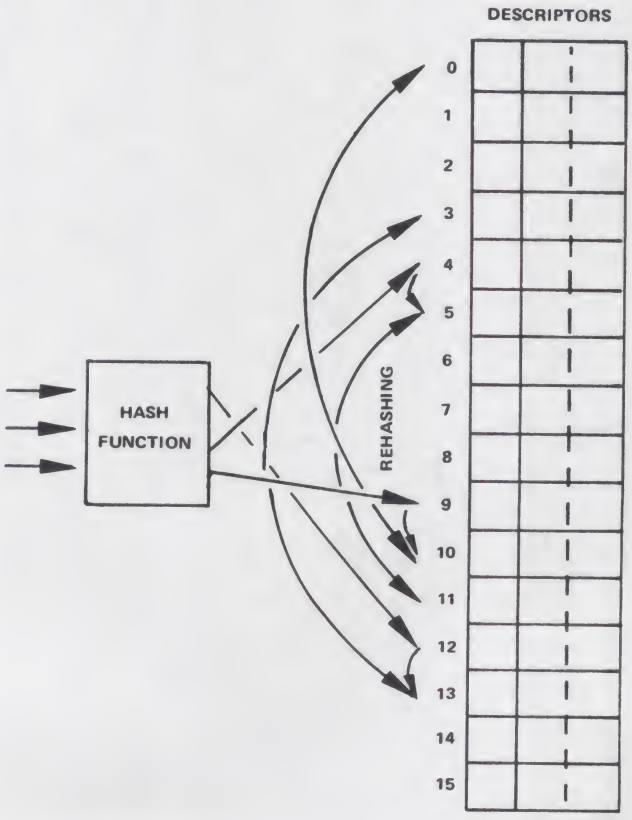
3. Shift R right 2 bits and add it to h to get the next table entry h. If this entry is occupied, rehash the name again.

To find a name in the table during Pass 2, we simply hash and rehash in exactly the same way, this time comparing each table entry h against the name to be found. The remarkable fact about this algorithm is that the number of comparisons needed to find an entry, on the average, depends only on *how full* the table is and *not on how large it is*. Even when the table is 90% full, only about 2.56 comparisons will be needed, on the average. In contrast, for a nearly full table of 512 entries, the sequential search method described earlier would take an average of 256 comparisons to find a name, or about 100 times as long!

Random rehashing is illustrated in Fig. 8. The first name to hash to table entry 12, for example, would be stored there, while the next name whose hash function value was 12 would be rehashed to table entry 13, and the next one would be rehashed twice and finally stored in table entry 3.

We have only described one method of hashing here; several other variations are possible. The most important of these is called "hashing with overflow chaining," in which all of the names which hash to the same starting address are chained together on a linked list. This method, which is often used on large computers with dynamic storage allocation, is less suitable for microcomputers because it requires an extra

Fig. 8. Hashing Symbol Table Descriptors.



address field for each symbol table entry. The references at the end of this article can be consulted for a more complete discussion of hashing.

Now that you have become acquainted with some of the basic programming techniques used for scanning and searching symbol tables, you're about ready to start writing your own assembler! You might want to actually try this, using the simplest techniques outlined in this article: Perhaps a fixed-column scanner and a sequentially searched symbol table for a first version. Very often, when it comes to actually getting a program up and running, the simple-minded approach turns out to be the one that works best. Once you've got a basic assembler working, you can consider adding some of the features that we'll discuss next.

More Assembler Features

Up to this point, we have been concerned with only the basic functions of an assembler: The conversion of mnemonics and programmer defined names to instruction opcodes and addresses. Many other features can be added to an assembly language to make it even more convenient for programming. Some of the more useful features of this kind will be considered here.

Very often, when it comes to actually getting a program up and running, the simple-minded approach turns out to be the one that works best.

Defining Constants

Most assembly languages have pseudo-ops which direct the assembler to reserve one or more locations containing constant values. For example, the Motorola 6800 assembly language has a pseudo-op FCB, for "form constant byte." An example of its use would be

```
FCB     23,$FA
```

which would reserve two bytes containing 00010111 (23 in decimal) and 11111010 (FA in hexadecimal or base sixteen).

Sometimes an instruction takes its operand in a memory location (rather than as an "immediate" operand), but the operand itself is actually a constant. Instead of writing

```
        ADDA   THREE
        .
        .
        .
THREE   FCB    3
```

we would like to be able to write

```
ADDA   =3
```

and have the assembler automatically reserve a memory location containing 3, and assemble its address into the instruction. Such an instruction operand is called a "literal." On machines where some instructions can address only a limited range of memory locations, this feature may be difficult to implement.

Equivalences

It is often convenient to be able to define a symbol with a constant value, or with the same value as another symbol. For example, a constant representing, say, the size of an array, may be used at several points in a program. By using a symbol in place of the constant throughout the program, and defining the symbol's constant value at the beginning of the program, we can make it easier to change the size of the array when producing a new version: Only the symbol need be redefined, and its new value will be substituted at the appropriate points by the normal process of assembly. (This is called "parameterizing" the program.) This feature is not too difficult to implement, and most assemblers have a pseudo-op such as

```
SIZE    EQU    25
```

which allows us to write

```
        LDAA   #SIZE
        .
        .
        .
ARRAY   RMB    SIZE
```

or, in general to use the symbol SIZE wherever the constant 25 could appear.

Expression Evaluation

Besides defining constants and constant-valued symbols in a program, it is frequently useful to be able to combine such elements into arithmetic expressions whose values can be computed at assembly time, and to use those values in place of other constants. For example, the same program with a parameter SIZE for the size of an array might include statements such as

```
        ADDA   #SIZE-1
        .
        .
        .
SPACE   EQU    3*SIZE+1
```

which would specify (for SIZE=25) that 24 should be added to the A accumulator, and that SPACE should have the constant value 76 wherever it appears in the program.

It is remarkably easy to evaluate expressions of this kind, taking account of parentheses and the normal precedence of arithmetic operations. An algorithm to perform the evaluation of such expressions will be the subject of an article in a later issue of BYTE; if you are impatient, you can consult Mealy or Gries (see the references).

Conditional Assembly

We saw how a program could be parameterized by the use of equivalenced symbols and arithmetic expressions. Sometimes a program can be parameterized in another way: Entire sections of the program can be included or omitted, depending on the values of certain parameters. For example, if the maximum value of a certain variable is less than 256, it can be stored in a single byte on most machines; but if the maximum value is 256 or more, two bytes or a word must be used. Thus we might wish to write something like

```
        .IF     MAXVAL LT 256
ARRAY   RMB     SIZE
        .END
        .IF     MAXVAL GE 256
ARRAY   RMB     2*SIZE
        .END
```

with the intent that, if an earlier EQU pseudo-op had defined MAXVAL as, say, 200, the first RMB statement would be assembled, while if MAXVAL had been defined as, say, 400, the second RMB would be assembled.

This feature is not too difficult to implement, and it is extremely useful. The assembler must simply recognize the .IF and .END pseudo-ops, evaluate the relations, and skip the intervening text on both passes if the relation is false. It is easy to imagine (but somewhat more difficult to implement) extensions to this feature, such as the repetitive assembly of certain program segments.

Macros and Relocation

The most sophisticated assemblers are comparable to compilers in complexity, size and versatility. Some assemblers implement a *macro* facility, which enables the programmer to define new instruction mnemonics which are replaced by parameterized sequences of assembly language statements wherever they appear in the program. When combined with features for conditional assembly, a macro facility provides a powerful tool for extending an assembly language to suit it for a particular application.

We have discussed only absolute assemblers: We began by assuming that the program was to be assembled starting at location 0 (or some other fixed location). When the program is going to be loaded into memory along with other, previously assembled programs, however, we don't know how big the other programs are or in which order they will be loaded. In this case it is necessary to put out *relocation* information along with the assembled program, which says, in effect, "If you load this program at location *m*, you should add the number *m* to the following bytes or words in order to make the addresses come out right." This relocation information is processed by a loader, which is responsible for loading all of the related programs into memory.

While both of these topics are interesting and very important, many pages would be required to do them justice and this article is pretty long already! So we'll content ourselves with the topics already discussed. By this time, you probably have either decided that writing an assembler is too much work, and have stopped reading this article, or else you have found the whole idea very intriguing and are looking forward for the last word. So here it is: Now that you know how to write an assembler, why not get out and give it a try? You have nothing to lose but your innocence about the complexities of system software, and perhaps a little of your time.

Good luck!

Now that you know how to write an assembler, why not get out and give it a try?

References

Barron, D. W. Assemblers and Loaders. *American Elsevier (Computer Monograph Series, No. 6), New York, 1969.*

The most complete, readily available text on the design of assemblers and loaders; also describes one-pass assemblers and meta-assemblers.

Gries, David. Compiler Construction for Digital Computers. *Wiley, New York, 1971.*

A highly recommended text on compiler design: Covers both the theoretical and practical aspects of the problem. Includes a good discussion of hashing and more sophisticated methods of scanning.

Mealy, George. "A Generalized Assembly System (Excerpts)," in Saul Rosen (ed.), Programming Systems and Languages. *McGraw-Hill, New York, 1967.*

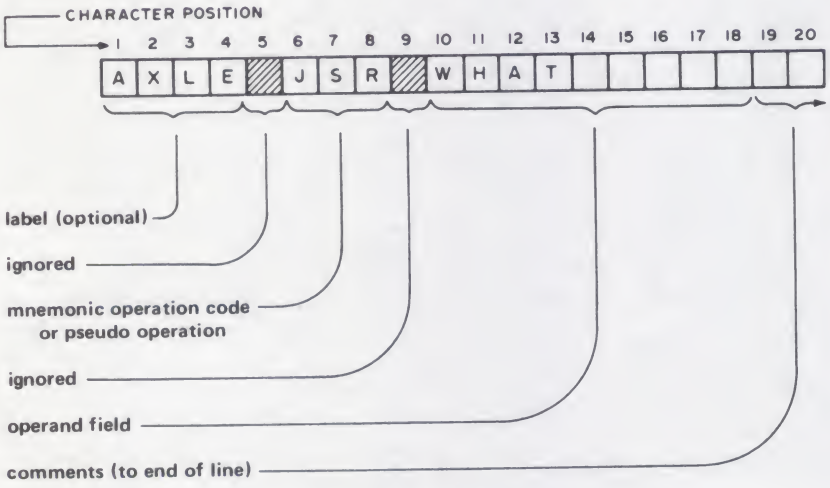
A classic paper by one of the pioneers of language translators and operating systems. Presents the idea of descriptors for character strings as well as many other innovations.

Morris, R. "Scatter Storage Techniques," in Communications of the ACM 11:1 *(January 1968), pp. 38-44.*

One of the best general surveys of hashing techniques; includes a good, brief description of hashing with overflow chaining.

Simplify Your Homemade Assembler

Gregory C Jewell
11855 Southeast 188th
Renton WA 98055



Notes: 1. A semicolon (;) in line position 1 indicates the whole line is a comment and will be ignored by the assembler.
2. If the .AS or .AZ pseudo operations are used, the operand field can be as long as required.

Figure 1: Summary of simplified assembler source format. This figure illustrates the fixed field format. The label field is used to define symbols, the operation code field is used to specify a mnemonic operation code or a pseudo operation, and the operand field is used to contain information according to the format of figure 2. Comments may be written by starting a line with a semicolon in position one, or following the operand field with the desired comments.

Our primary goal in the design of a simple assembler is to eliminate the need to parse a line in order to determine what information is contained in that line. Rather than asking "What are you trying to give me?", our assembler will demand, "I know where I am, so give me what I want."

The assembler described here is a three-pass assembler. The first pass compiles a symbol table; the second pass outputs the generated machine code, and the third pass produces a hexadecimal listing of the generated machine code with its associated addresses and source statements.

Labels

The first step on our path to simplicity is a major one even though its impact on our program writing will be slight. We will specify that all labels should have a fixed length of four characters with a restriction that the first character should be alphabetic. Although not the main objective, requiring fixed length labels adds the feature of allowing embedded blanks in the labels. Figure 1 illustrates the fixed field format of the simplified assembler. In the example, a line of assembler input is shown in the boxes, with shaded boxes indicating blanks. The label AXLE is shown on a statement containing the JSR operation code with operand WHAT.

Six or eight characters is a popular maximum for assembly language labels; however, our four character labels will save memory space and speed up the task of searching for a label in the symbol table.

The simplified assembler will demand "I know where I am, so give me what I want."

A label is defined when it appears for the first time in a statement of the program which is being assembled. A label is not required for every statement. However, if the first character position of the statement is found to have an alphabetic character, then the first four columns define a new label for the symbol table. If the first position is a blank, then the assembler should ignore the remaining positions of the label field. This is an example of what is called a fixed field syntax because we always expect a label or no label at all in these positions. Programming of the assembler is simplified by use of this limitation. The need for parsing has been nearly eliminated by this single requirement of fixed-length labels. But let's take a few more steps.

Operation Codes

As in commercially available assemblers, the next field on each line of the program being assembled is an operation code field. This field is separated by a blank character position from the label field, and thus begins in the sixth character position of our fixed field input format. In the operation code field, the assembler can find two types of information: an assembler pseudo operation or a mnemonic operation code for machine instructions.

Pseudo Operations

A mnemonic operation code is a symbol which the assembler in most instances will translate into a machine instruction. A pseudo operation code is a similar symbol which looks very much like a mnemonic operation code. However, the pseudo operation does not normally generate machine instructions and is used instead to control how the assembler will generate code. All assemblers have pseudo operations. Ours is no exception. When choosing pseudo operations, the goal of simplicity should be kept in mind. Most likely we will be able to get by without many of the fancy or powerful pseudo operations that add bulk and complexity to the assembler program.

I have defined nine basic pseudo operations for my assembler. All begin with a period so that the assembler program need only examine the first character to determine if the mnemonic is a pseudo operation. This speeds address calculation during the first pass since all other PACE instructions generate a single word of code. It also aids

Mnemonic	Description	Action
.SA	starting address	Defines the address for the next instruction. The assembler must know where to start assigning code whether by default or instruction. Similar to the ORG pseudo-op of other languages.
.RS	reserve storage	Saves space for specified number of words.
.XW	hexadecimal word	Loads specified hexadecimal value into location.
.AS	ASCII string	Breaks down a character string into its ASCII equivalent.
.AZ	ASCII string with zero	Same as .AS except that the ASCII code is terminated by a zero byte.
.DF	define address	Loads address of specified label into location.
.IL	inhibit listing	Inhibits listing during third pass.
.EL	enable listing	Enables listing during third pass (default condition).
.ND	end	End of source program.

Table 1: Pseudo operations.

human recognition. The nine pseudo operations are briefly described in table 1.

We now have all the information required to complete the first (address allocation) pass. It is possible to identify a label and calculate its address, since PACE has fixed length instructions. The label and its associated address are stored in the symbol table sequentially. A symbol definition requires three words, since we must store two words for the name and one word for the address. If desired, at the end of the first pass the labels may be sorted by the first character (it's surprising how close this

Table 2: An example of the output of an assembler implemented according to this definition. This assembly shows a memory test program written for the author's system. Bearing in mind all the limitations placed upon the source format to simplify writing the assembler, note that the listing looks like a "typical" output of an assembler. Note the frequent use of comment lines (starting with a semicolon) to explain various aspects of the program. The program uses the author's 3 character mnemonics instead of the PACE mnemonics, and the pseudo operations are shown in table 1.

```
END FIRST PASS,       0 ERRORS DETECTED

SYMBOL TABLE

 DATA     000E
 DSPL     0024
 ERR      0020
 LIM      0025
 NEXT     001D
 RITE     0001
 READ     0006
 REED     0015

END OF SYMBOL TABLE,     8 LABELS
```

```
          1             ; TWO-PART MEMORY TEST
          2             ; (1) ADDRESS-DATA CHECK
          3             ;     WRITE A UNIQUE NUMBER IN ALL LOCATIONS
          4             ;     IF A USED ADDRESS LINE IS BAD, THEN AT LEAST
          5             ;       ONE ERROR WILL OCCUR
          6             ;
          7                   .SA 0
          8 0000 5226         LIM R2,26     LOAD STARTING ADDRESS OF TEST
          9 0001 DA00 RITE    STA R2,(R2)   WRITE ADDRESS INTO LOCATION
         10 0002 F922         SNE R2,LIM    MEMORY LIMIT REACHED?
         11 0003 1902         JMP READ      YES
         12 0004 7A01         AIS R2,1      NO, INC INDEX
         13 0005 19FB         JMP RITE
         14             ;
         15             ; READ BACK UNIQUE NUMBERS
         16             ;
         17 0006 5226 READ    LIM R2,26     RELOAD STARTING ADDRESS
         18 0007 FA00         SNE R2,(R2)   COMPARE, SKIP IF ERROR
         19 0008 1901         JMP  +2
         20 0009 1916         JMP ERR
         21 000A F91A         SNE R2,LIM    MEMORY LIMIT REACHED?
         22 000B 1902         JMP DATA      YES, GO TO NEXT PART OF TEST
         23 000C 7A01         AIS R2,1      NO, INC INDEX
         24 000D 19F9         JMP READ+1
         25             ;
         26             ; (2) SHIFT-ONE DATA CHECK
         27             ;     TEST WORD HAS A SINGLE BIT SET
         28             ;     WRITE TEST WORD IN ALL LOCATIONS
         29             ;     TEST ALL BIT POSITIONS
         30             ;
         31 000E 5001 DATA    LIM R0,1      INITIALIZE TEST WORD
         32 000F 5226         LIM R2,26     LOAD STARTING ADDRESS
         33 0010 D200         STA R0,(R2)   WRITE TEST WORD
         34 0011 F913         SNE R2,LIM    MEMORY LIMIT REACHED?
         35 0012 1902         JMP REED      YES
         36 0013 7A01         AIS R2,1      NO, INC INDEX
         37 0014 19FB         JMP DATA+2
         38             ;                  ;
         39             ; READ BACK TEST WORD
         40             ;
         41 0015 5226 REED    LIM R2,26    'RELOAD STARTING ADDRESS
         42 0016 F200         SNE R0,(R2)   COMPARE, SKIP IF ERROR
         43 0017 1901         JMP  +2
         44 0018 1907         JMP ERR
         45 0019 F90B         SNE R2,LIM    MEMORY LIMIT REACHED?
         46 001A 1902         JMP NEXT      YES
         47 001B 7A01         AIS R2,1      NO, INC INDEX
         48 001C 19F9         JMP REED+1
         49 001D 2802 NEXT    SHL R0,1      SHIFT TEST WORD
         50 001E 45F0         BOC 5,DATA+1 WRITE NEW TEST WORD IF NONZERO
         51 001F 5E00         CPY R2,R0     TEST COMPLETE, DISPLAY 0 ERRORS
         52             ;
         53             ; ERROR ROUTINE: DISPLAY BAD LOCATION
         54             ;
         55 0020 5C80 ERR     CPY R0,R2
         56 0021 B102         STI DSPL      LOAD DISPLAY REGISTER
         57 0022 D903         STA R2,LIM +1  SAVE TEST DATA FOR REFERENCE
         58 0023 0000         HLT
         59 0024 8009 DSPL    XW 8009       ADDRESS OF DISPLAY REGISTER
         60 0025 03FF LIM     XW 3FF        MEMORY LIMIT = 1K
         61                   .ND

      END THIRD PASS,       0 ERRORS DETECTED
```

comes to actually alphabetizing the labels) and listed with their addresses. The sample assembly of table 2 shows the result of such a sort.

Mnemonic Operation Codes

The next step toward simplification is to specify that all mnemonic operation codes should also have a fixed length. National Semiconductor Corporation, PACE's manufacturer, suggests mnemonics containing from two to five characters. Even if we use the manufacturer's suggested mnemonics and specify a fixed length of five characters, the indirect notation @ would probably throw a wrench into the works since the @ usually directly precedes the label rather than immediately following the mnemonic.

I chose to define a set of 3 character mnemonics. This saves memory space and speeds up the search for mnemonics in the table of operation codes. The three characters of the mnemonic operation code can be stored in one and a half words (3 bytes) and the binary opcode may be kept in the remaining byte. There is nothing magic about mnemonics; they are simply aids to remembering the instructions. It's your computer, so you might as well use your own mnemonics – unless you plan to make your assembler commercially available. Table 3 shows the correlation between the

An effective address is a combination of an addressing mode and a displacement.

Table 3: Correlation between manufacturer's suggested mnemonics and the author's 3 character mnemonics.

	Manfacturer's Suggested Mnemonics	Description	Author's Mnemonics
1.	JMP	jump	JMP
2.	JMP@	jump indirect	JMI
3.	JSR	jump to subroutine	JSR
4.	JSR@	jump to subroutine indirect	JSI
5.	SKG	skip if greater	SGT
6.	SKAZ	skip if AND is zero	SAZ
7.	ISZ	increment and skip if zero	ISZ
8.	DSZ	decrement and skip if zero	DSZ
9.	LD@	load indirect	LDI
10.	ST@	store indirect	STI
11.	LSEX	load with sign extended	LSX
12.	AND	logical AND	AND
13.	OR	logical OR	IOR
14.	SUBB	subtract with borrow	SBB
15.	DECA	decimal add	DCA
16.	AISZ	add immediate, skip if zero	AIS
17.	LI	load immediate	LIM
18.	XCHRS	exchange register and stack	XRS
19.	CFR	copy flags into register	CFR
20.	CRF	copy register into flags	CRF
21.	PUSH	push register onto stack	PSH
22.	PULL	pull register from stack	PUL
23.	CAI	complement and add immediate	CAI
24.	SKNE	skip if not equal	SNE
25.	LD	load	LDA
26.	ST	store	STA
27.	ADD	add	ADD
28.	RXCH	register exchange	RGX
29.	RCPY	register copy	CPY
30.	RADD	register add	RAD
31.	RADC	register add with carry	RAC
32.	RAND	register logical AND	RND
33.	RXOR	register exclusive-OR	XOR
34.	BOC	branch on condition	BOC
35.	RTS	return from subroutine	RTS
36.	RTI	return from interrupt	RTI
37.	PUSHF	push flags onto stack	PSF
38.	PULLF	pull stack into flags	PLF
39.	HALT	halt	HLT
40.	SFLG	set flag	SET
41.	PFLG	pulse flag	PLS
42.	SHL	shift left	SHL
43.	SHR	shift right	SHR
44.	ROL	rotate left	ROL
45.	ROR	rotate right	ROR

All assemblers have pseudo operations. This one is no exception.

manufacturer's suggested mnemonics and the 3 character mnemonics which I selected to simplify my assembler.

Instruction Groups

So far we have defined a 4 character label field and a 3 character mnemonic field. To make the program readable, we'll allow a single character (blank) after each field and a semicolon in the first character position (column one) to signal a comment line. Our assembler now expects either a blank, a semicolon, or an alphabetic character in the first position. As noted earlier, if the first position of a line contains an alphabetic character, then a label exists in the first four positions. The fifth position is ignored. The sixth through eighth positions contain the operation code or pseudo operation mnemonic and the ninth position is ignored. What does the assembler expect in the tenth position? To answer this question, we must collect instructions with similar binary and source formats into instruction groups. The only variation within an instruction group is the mnemonic operation code. Figure 2 lists the ten PACE instruction groups.

After the instruction group is determined, our assembler will know exactly what to look for and where to find it. For example, if the instruction is in group three, the tenth character position is ignored (allowing you to specify RO, AO, XO, or whatever pleases you at the time), a digit less than four is expected in the eleventh position; the twelfth position is ignored, and the destination (DEST) field begins in the thirteenth position. If the instruction is in group four, then the assembler expects to find a digit less than four in the eleventh and fourteenth positions. If the instruction is in group seven, then the assembler's worries are over, since such instructions have no operands.

Destination Field

The destination field (DEST) is required to determine the effective address. An effective address is the combination of an addressing mode and a displacement. The four PACE addressing modes are program counter relative, relative to register R2 used as an index, relative to register R3 used as an index, and base page. All addressing modes of the destination field entries (destination modes) listed in table 4 are program counter relative except the last two: (R) is index mode and *K is base page mode. The index and base page modes are limited primarily by my own biases and could be chosen differently in your own version of such an assembler. As with all other fields of a personal assembler, the DEST field should be tailored to your own preferences. The modes of table 4 are sufficient while maintaining the goal of simplicity.

Figure 2: PACE Instruction groups.

Group	Instructions	Binary Format (15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00)	Operand Format
0	JMP,JMI,JSR,JSI,SGT,SAZ,ISZ,DSZ, LDI,STI,LSX,AND,IOR,SBB,DCA	OP | XR | DISP	Position 10 DEST*
1	AIS,LIM,CAI	OP | R | IMMEDIATE	R,K
2	XRS,CFR,CRF,PSH,PUL	OP | R | NOT USED	R
3	SNE,LDA,STA,ADD	OP | R | XR | DISP	R, DEST*
4	RGX,CPY,RAD,RAC,RND,XOR	OP | DR | SR | NOT USED	R,R
5	BOC	OP | CC | DISP	M,DEST*
6	RTS,RTI	OP | DISP	K
7	PSF,PLF,HLT	OP | NOT USED	none
8	SET,PLS	OP | FC | P | NOT USED	M
9	SHL,SHR,ROL,ROR	OP | R | N | L	R,K or R,K,L

R = R0, R1, R2 or R3
0 ≤ M ≤ F

0 ≤ K ≤ FF
L = "L" (letter L)

*** See Modes of the destination field, table 4.**

Again, by examining only the first character of the field, the assembler can determine if the DEST field has a label, a specified displacement, an index register, or a base page value. The + or – extension after the label will always be in the same relative position since we have declared that all labels contain four characters. If the first character of the DEST field is an alphabetic character, then the first four characters of the field form the label; and, if there is an extension, the + or – will always be the fifth character of the field.

Example

Table 2 shows the output of the first and third passes of a memory test program. It looks general even though strict rules were applied. The execution time is approximately 1.5 seconds for each 1 K of memory tested. Notice the destination LIM +1 in statement line 57. LIM+1 would have produced an UNDEFINED LABEL error. The trailing blank is part of the label.

If you desire simplicity and can live with LIM +1 rather than LIM+1 then you might implement the rules I have presented (or your own variation) in your homemade assembler.

Conclusion

The simplified homemade assembler's source language is now completely defined in a way which is simple and easy to implement, yet probably adequate for all our programming needs. Except for the .AS and .AZ pseudo operations, we have eliminated the need for parsing, mainly by specifying a fixed label length (with embedded blanks) and a fixed mnemonic length. Other simplifications were achieved by selecting only basic pseudo operations and destination modes. By using these techniques, you should have your homemade assembler running by tomorrow.■

Table 4: Modes of the destination field (DEST).

DEST		Description
LABEL		symbolic
LABEL+K		symbol relative
LABEL–K		symbol relative
.+K	(here plus K)	program counter relative
.–K	(here minus K)	program counter relative
(R)		index register
*K		base page

0 < = K < = FF
R = R2 or R3

REFERENCE

PACE Technical Description
National Semiconductor Corp
Santa Clara CA 95051
Publication number 4200078A
June 1975

GLOSSARY

ASCII: American Standard Code for Information Interchange. A 7 bit code used by many machines.

Assembler: An assembler is a program which accepts a symbolic representation of some computer program and transforms it into one which can be executed by a computer. The symbolic representation is called a source program; the executable representation is called an object program.

Character Position: Each line of the source program which is read by the assembler is a character string. In a fixed field syntax, the character positions are numbered (in this case, from 1 to the end of the line). Each field of the format is a group of characters specified by number, such as the label field which is positions 1 to 4 of a line in the example of this article.

Mnemonic: A technique to assist human memory. A mnemonic term is an abbreviation or acronym used instead of numeric codes in order to facilitate easy recognition. Example: BOC for Branch On Condition rather than 4.

Parsing: The breaking down of a general character string into its structural forms. This requires syntax rules for the computer language analogous to the grammar rules for English that define "subject," "predicate," "object," and so forth. In this assembler, we simplify syntax rules by requiring fixed positions for each piece of information on a line which eliminates the need for parsing.

Pass: An assembler typically must look at the entire data of a program several times. Each pass of an assembler is one complete scan through the program data. In the simplest home brew assemblers using audio cassette mass storage, each pass will require manual intervention to rewind and restart the appropriate tape cassette drive.

Pseudo operation: A group of characters having the same general form as a computer instruction, but never executed by the computer as an actual instruction. Pseudo operations are instructions to the assembler.

Source Program: A program coded as a human readable character string in some programming language, which must be translated into machine language.

Symbol Table: A dictionary relating one set of symbols to another set of symbols or numbers. The assembler builds a table of labels used in the assembly language program and assigns memory locations (addresses) to those labels.

Interact with an ELM

G H Gable
419 Jackson St
Grand Ledge MI 48837

The fundamental interface between the user and the hardware of a computer system is the system software. It runs the gamut from a dozen or so bytes of a bootstrap loader on a microcomputer to the multi-million word operating system of a large general purpose computer system. In fact, the microcomputer system can be made to do much of what the general purpose computer does with appropriate versions of systems software. One of the most significant differences between the big computer and the microcomputer is that the large computers typically operate on multiple bytes of information and often provide extended arithmetic and logical operations. Minicomputers and microcomputers can emulate these extended operations with software; the main difference is speed. The typical large computer might execute its built in instructions 1000 times faster than a microcomputer's software emulation. However, all the features of a large computer system can be implemented in the software of a microcomputer system. This includes assemblers, compilers, text editors, time-sharing and multiprogramming, disk operating systems, virtual memory, utilities, and of course applications programs. In addition, the powerful hardware of a big computer can be emulated with software. The principal hardware requirements, other than a general purpose instruction set, are access to the program counter, an interrupt structure and possibly direct memory access by the peripheral equipment. Program counter access and interrupt processing is available in most microprocessors; direct memory access is often implemented by peripheral device controllers using the system bus.

For microcomputers, the system software can be divided into two major segments: the operating system or monitor and a utility library of functions which extend the instruction set. The utility library is a set of subroutines written to redefine and expand the operations the computer can perform. It can range from a simple set of number conversion and formatting routines up to the complexity of a complete floating point mathematical package.

Monitors

The monitor program, sometimes called the executive program or operating system, is the program which the computer executes when it is not running some other program. The monitor's primary purpose is to decide what the system is to do next. Sophisticated monitors typically implement disk operating systems, time sharing and multi-programming. They call loaders, assemblers and compilers, handle input and output, and process user requests. In short, the monitor program is "the brains" of the system. In some very large systems, such as the Control Data Corporation's CDC-6500, the monitor program even has its own processor, separate from the central processors. The central processors are merely slaves to the monitor processor in such a multiprocessor system.

For a beginning, let's examine a very simple monitor program. If you have a microcomputer which needs system soft-

ware, this might be just the ticket to get you on the system. This monitor design will let you load and execute programs and edit the contents of memory. From such a basic monitor, more sophisticated software can be developed to upgrade the system to whatever level you desire.

ELM

Every routine should have a name, especially a system routine. Thus I call this the Eloquent Little Monitor, or ELM. ELM is designed to have a Teletype or a cathode ray tube (CRT) terminal such as a TV typewriter as its control console. A CRT running at 1200 baud makes a wonderful control console due to the brisk speed at which messages are transmitted. ELM implements four commands in its simplest version: LOAD which will load a program into memory beginning at a specified location; LIST which lists the content of selected memory locations; MODIFY which will modify the contents of selected memory locations, and GO which starts execution of a program at a specified location. My version of ELM features decimal addresses and allows input line editing.

Many processors begin execution at a fixed location at power-on or system reset. Some processors begin execution at a hardware programmed address which might be set by switches. Wherever the processor begins its execution, the implementation of ELM assumes that ELM will be the program which receives control as a matter of course. For the purposes of this article, we'll assume that ELM is located at the low end of memory address space. Following ELM comes the first available address (FAA) of user programmable memory, then the last available address (LAA). This memory organization for a typical monitor residing at the low end of address space is shown in figure 1. Other allocation schemes are of course possible. It is also desirable to have the monitor in a read only memory so that, when the computer is first switched on, the CPU will immediately begin execution of the monitor. With such a firmware monitor, your programs will not be able to destroy the monitor program itself. In addition to the address space for the monitor, the allocation shown in figure 1 includes 80 bytes of programmable RAM for use as data storage.

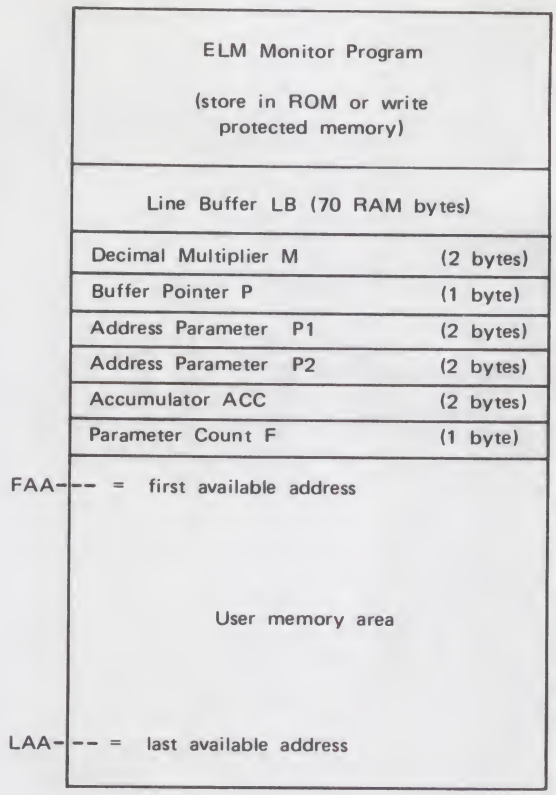
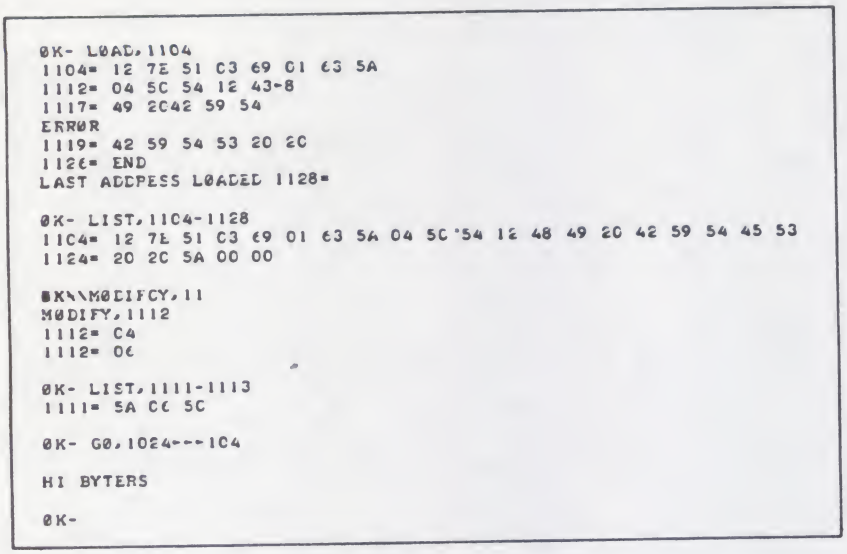


Figure 1: Memory Allocations for a Typical Monitor Program. This map assumes that the ELM monitor program resides at the low end of memory address space, and that programmable random access memory begins at the address of the line buffer.

Using ELM

First, let's look at the monitor from the user's point of view at the terminal. When the system is switched on, the Teletype or display will print "OK—". Any time the monitor is waiting for a command it will print the same message.

If you want to enter a program starting at location 123, type "LOAD,123" then a carriage return to end the line. The ELM program will respond with the message "123=" on the next line. ELM now expects you to type a string of hexadecimal characters grouped two per byte, with a single space between each group. See figure 2 for examples of this format. You can enter up to 22 double character hexadecimal codes on a line. The line is terminated with a carriage return. After the carriage return, these codes are entered into memory beginning at the address 123 in this example. Then the address waiting to be loaded will be printed at the beginning of the next line so that more hexadecimal codes can be entered. This process is repeated until you type the word "END" at the beginning of a line. After ending the load routine, the last loaded address is printed followed by the "OK—" message which indicates that ELM is



```
0K- LOAD,1104
1104= 12 7E 51 C3 69 01 63 5A
1112= 04 5C 54 1E 43-8
1117= 49 2C42 59 54
ERROR
1119= 42 59 54 53 20 2C
112C= END
LAST ADDRESS LOADED 1128=

0K- LIST,1104-1128
1104= 12 7E 51 C3 69 01 63 5A 04 5C 54 1E 48 49 2C 42 59 54 45 53
1124= 20 2C 5A 00 00

0K\\MODIFY,11
MODIFY,1112
1112= C4
1112= 0C

0K- LIST,1111-1113
1111= 5A CC 5C

0K- G0,1024---1C4

HI BYTERS

0K-
```

Figure 2: Sample Printout of an ELM Interactive Sequence. This listing shows ELM at work. Note the use of the Teletype back arrow (underscore character) to delete mistakes and one instance of a cancelled line. This listing illustrates use of ELM to load and execute a simple program which types out "HI BYTERS" and returns to ELM.

back in the command mode again. If the starting address is omitted or is less than the first available address (FAA) then FAA is assumed.

If you want to list the contents of memory locations 123 to 456, the command "LIST,123-456" will start the listing, printing 20 hexadecimal codes per line. If the address range is omitted, listing begins at the first available address (FAA) and continues until the last available address (LAA) or an end of program mark. Figure 2 illustrates the output format of a listing.

If you want to modify memory contents at locations 123 to 130, the command "MODIFY, 123–130" will first list the old contents of these locations, then it will enter the load routine to print "123=" as if you were loading these locations. Modified codes may then be entered, to be stored beginning at 123.

Finally, if you want to start executing the program at location 123, the command "GO,123" puts 123 into the program counter and begins execution of your program. Again, if the address is omitted, execution starts at the first available address, FAA.

It is certainly easy to make typing errors, especially for me. Thus I implemented ELM with a line buffer and two special line editing characters. The underscore (ASCII back arrow, hexadecimal 5F) effectively removes the preceding character typed, two underscores remove the preceding two characters, etc. The control X character (ASCII cancel code, hexadecimal 18) cancels the whole line. Several reverse slashes (ASCII, hexadecimal 5C) are printed on the cancelled line and a line feed is generated as shown in figure 2.

Architecture

Now that the monitor design is set, let's look at the architecture of the program needed to implement ELM. Figure 3 shows the logic for the whole monitor. After the power on restart, "OK–" is printed as the ELM input request message, then the system idles while waiting for input. Figure 4 shows the logic of the subroutine INPUT, which reads each character and puts it into the line buffer. If the terminal is running in the full duplex mode, the character should be echoed back to the printer. The buffer pointer, P, shows where to put the next character in the buffer. The editing characters are implemented as shown. An ASCII carriage return code (hexadecimal 0D) ends the input sequence. The test for carriage return is done after storing the input character since the load routine expects a carriage return as an end of line character.

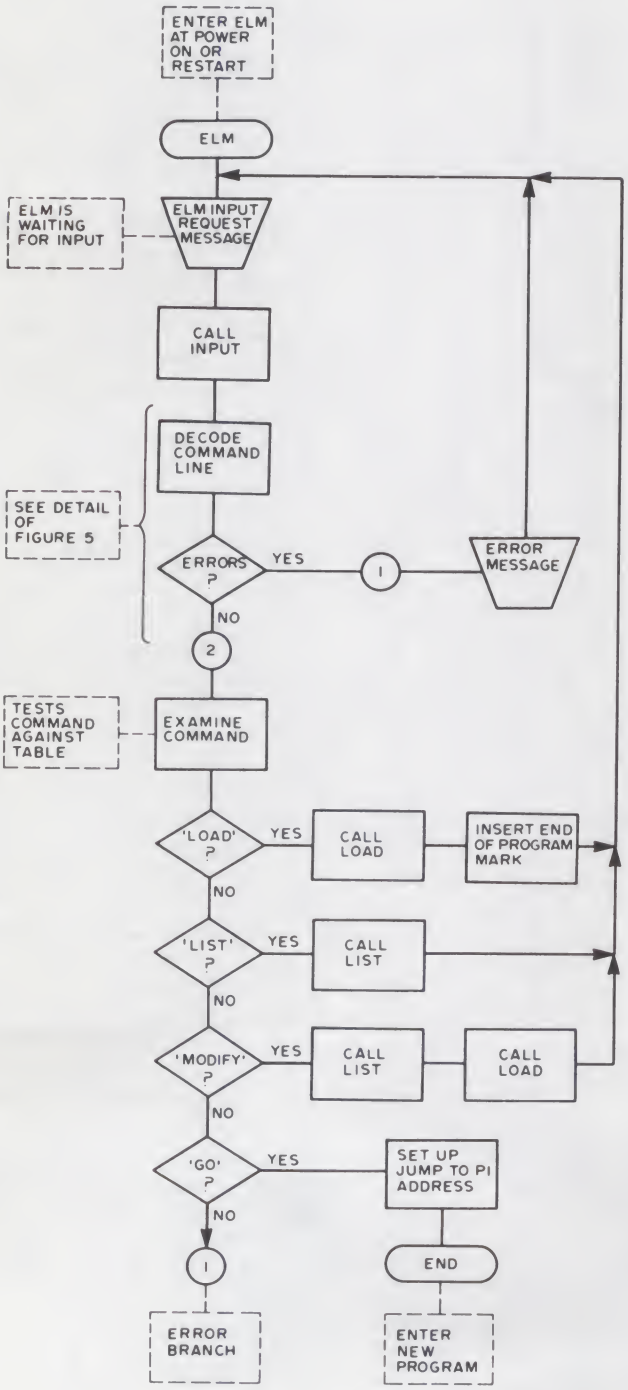
In figure 3, the parameter decoding and error checking logic is shown as a box and an error test with a note attached. This logic is expanded in more detail in figure 5. The parameter decoding logic has a structure that enforces a non ambiguous syntax on the command line. The command is examined by means of a command list. This list is a table which is sequentially searched, matching the command in the buffer with each possible command in the table. The result is used to determine the proper branch. An error message is printed if the command is not found in the table.

The LOAD subroutine is shown in figure 6. The logic consists of an outer loop for each line of input, and an inner loop which scans the line, loading memory from left to right in ascending address order. The LOAD routine checks the syntax for double character hexadecimal codes separated by blanks. If a syntax error is found, loading stops, an error message is printed, and the next address to be loaded is printed on the next line. A variable number of hexadecimal codes from 1 to 22 may be entered on each line. The initial address (P1) is incremented during the loading routine.

Note that after loading is completed and control returns to the main routine, an end of program mark is inserted into memory. In my version of ELM, the code for a jump to

address zero is loaded into the next three bytes as an end of program mark. This convention allows normal termination of a user program by running off the end and branching to the starting address of the monitor at location 0.

The LIST routine is shown in figure 7. This routine simply prints out the hexadecimal codes found at locations specified by the input parameters. This listing is done 20 bytes per line. Note that LIST stores the



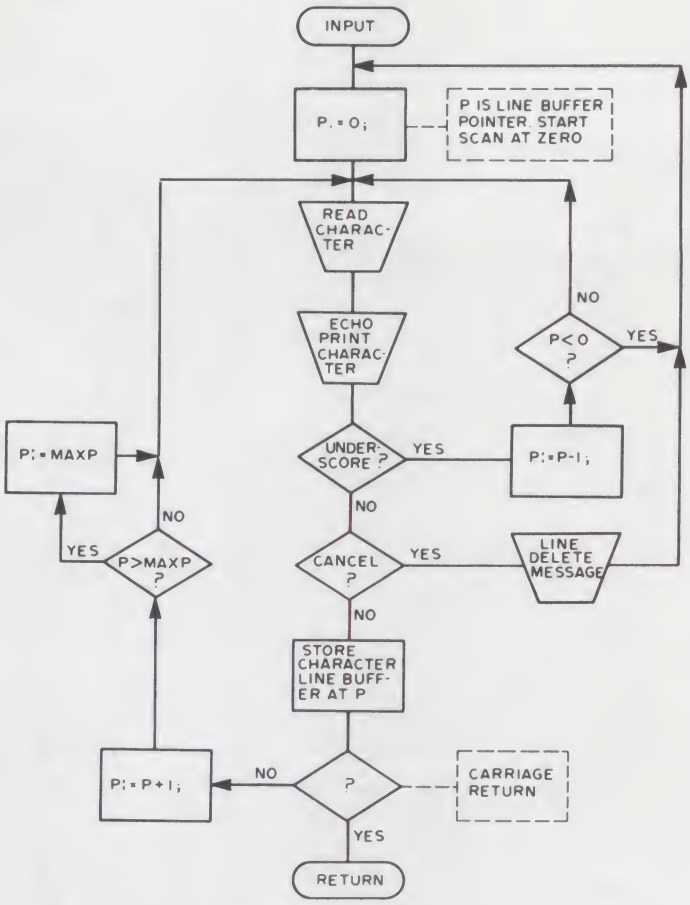


Figure 4: The Input Subroutine Specified as a Flow Chart. The main purpose of INPUT is to read one line of input, terminated by a carriage return. INPUT implements the line editing functions of character delete and line delete. When the carriage return code is detected, the line buffer LB is filled from position 0 to position P.

Figure 3: The ELM Program Specified as a Flow Chart. The main logic of the Eloquent Little Monitor is shown in this diagram. Flow begins at the top left and proceeds down the diagram. Normal operation of ELM involves a closed loop, returning to the ELM input request message printed near the top of the diagram. If the GO command is carried out, execution leaves ELM and proceeds to the selected address.

initial value of parameter P1 in the accumulator ACC during its operation. Then P1 is restored after the listing is completed. This allows LOAD to be called after LIST during a MODIFY sequence, so that both LOAD and LIST reference the same starting address.

In my version of ELM, addresses are handled as decimal numbers. This is reflected in the input numeric conversion logic (see figure 5) and in the creation of an output conversion subroutine: Both LOAD and LIST call a subroutine DECIMAL which prints the decimal addresses at the beginning of lines in messages. DECIMAL simply converts the first address parameter, P1, into five ASCII numeric characters, and prints them followed by an ASCII "=" character and a blank. I put decimal address conversion into ELM out of personal preference. The decimal conversions may be omitted and hexadecimal or octal address parameters could also be used. There is already a binary to ASCII hexadecimal routine implicit in the

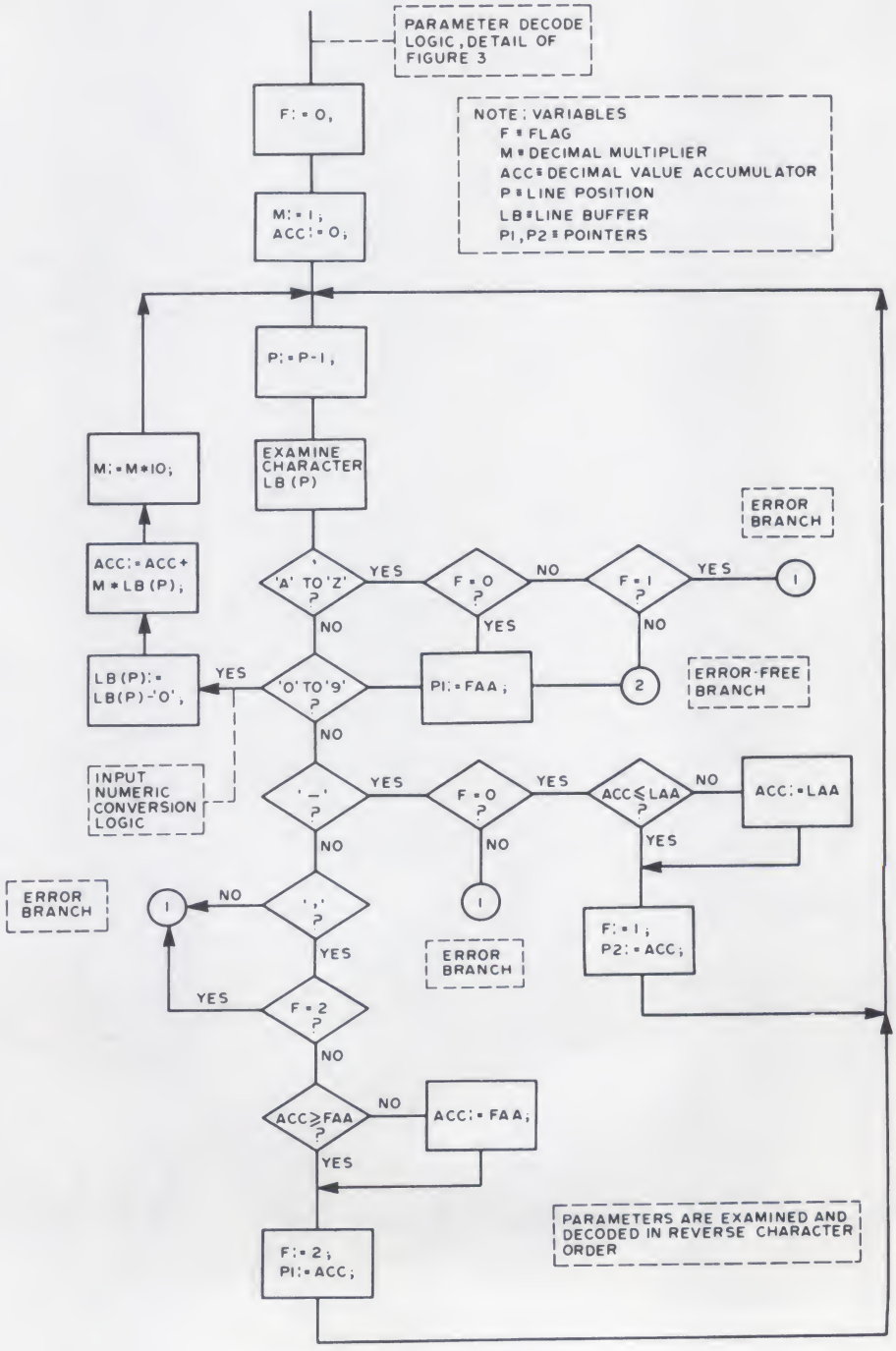


Figure 5: Parameter Decoding Logic Details. Figure 3 contains a box labelled Decode Command Line and a conditional test labelled Errors, with a note referencing figure 5. This figure contains the details of the logic needed to decode a command line into two parameters and a command. There are two possible exits from this logic. An error exit to terminal (1) occurs if an error is detected; an error free exit to terminal (2) occurs if no errors are detected.

LIST function. For input, the parameter decoding routine can be simplified somewhat by using hexadecimal parameters.

Expansions

There are several obvious expansions to ELM which should be easy to implement. You may even want to incorporate them into your own version of ELM right from the start. If you have an ASR Teletype (with paper tape reader and punch), you may want to add the following commands: LOADPT and PUNCH. Your Teletype should be able to receive the rubout character (ASCII delete, hexadecimal FF) but not transmit as is the normal configuration. LOADPT would operate the same way as LOAD except that there is no printing needed. The format of the tape would be lines of hexadecimal codes with a carriage return and two or more delete characters at the end of each line. You can skip the blanks between bytes to save tape if you like. When the processor sees the carriage return, it begins loading memory from the line buffer. The two delete characters give the computer time to load the line, so that by the time the next real character comes along the computer is ready for it. Instead of the word "END" at the end of the input, you might want to use

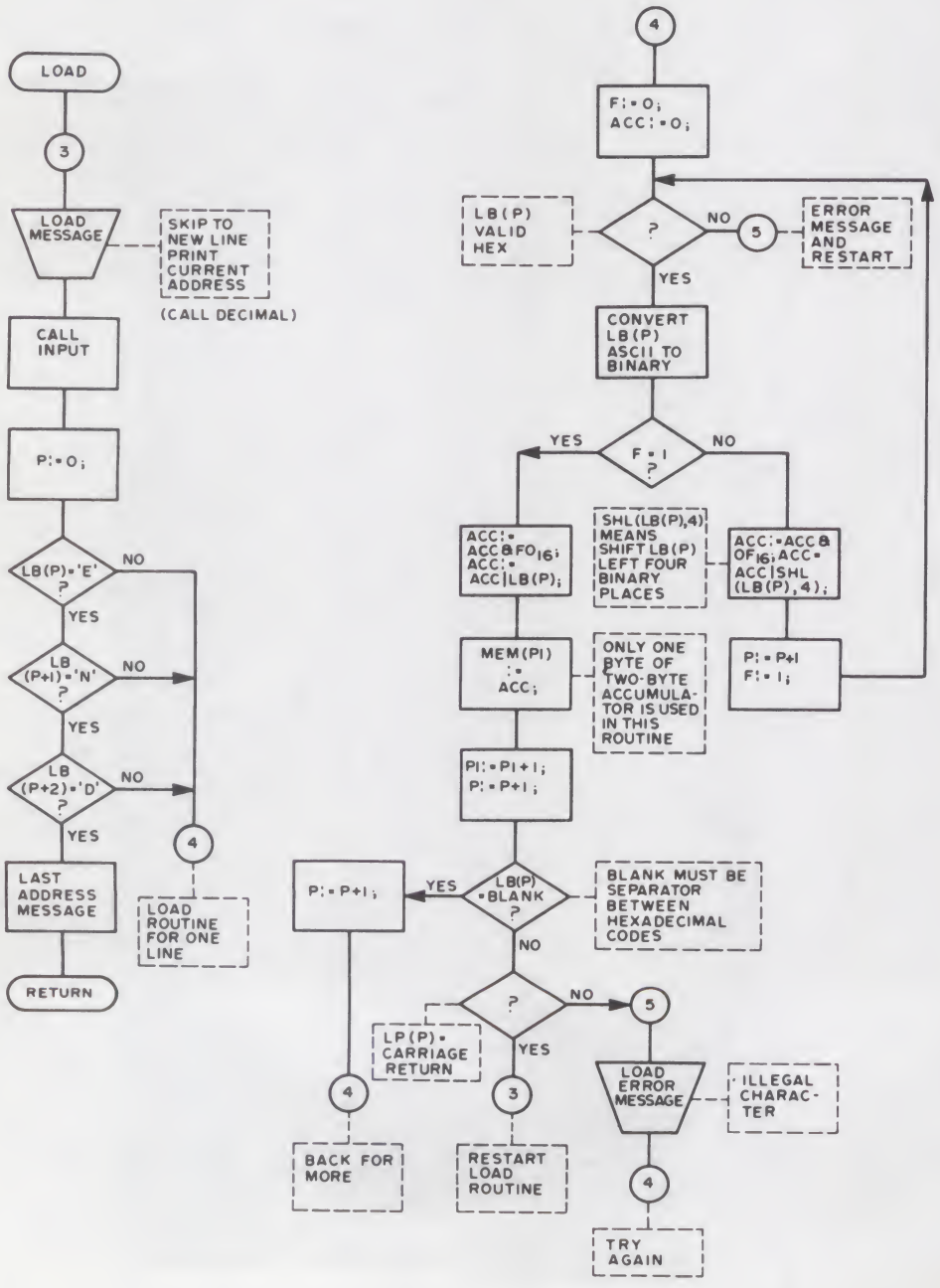


Figure 6: The LOAD Subroutine Specified as a Flow Chart. The purpose of LOAD is to set the contents of user programmable memory beginning at a location specified by the user. The routine continues indefinitely until the characters "END" begin a line of input.

the ASCII end of tape character (hexadecimal 04). The PUNCH routine would operate like LIST, without the addresses. It should punch the tape in exactly the same format read by LOADPT. If you are not using blanks between bytes in the tape format, you can get 34 hexadecimal codes on a line followed by a carriage return and the two delete characters. The last character punched might be the end of tape code or the END convention, depending upon your own preferences.

If you have a serial tape drive at a different IO port, you may want LOADMT and SAVEMT commands. These could be exactly like LOADPT and PUNCH except for the IO port address. Most tape interfaces are set up to use the null code (hexadecimal 00) instead of the delete code to give blank spacing. You may also want to implement absolute binary versions of SAVEMT and LOADMT to allow higher speed and eliminate conversions.

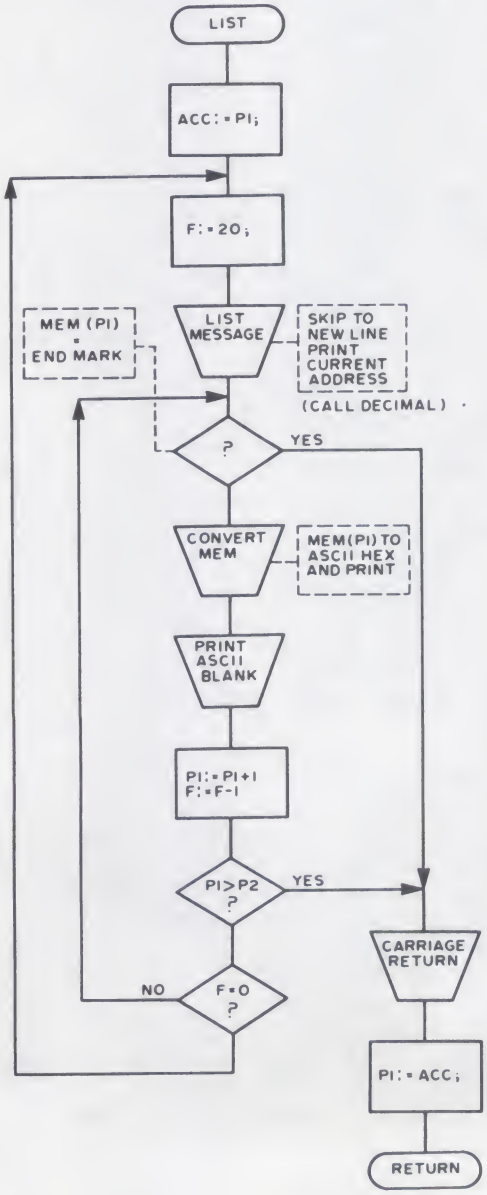


Figure 7: The LIST Subroutine Specified as a Flow Chart. The purpose of LIST is to dump the contents of memory, formatted as ASCII encoded hexadecimal digits. The dump routine types the address first on each line, then follows with 20 groups of two hexadecimal digits.

Philosophy

With this article, I've given you enough information on the design of a monitor to enable you to write the code for your own machine. After a few days of coding and debugging, you should be ready to go to the local computer store and have your ROMs zapped with a mighty ELM. The whole monitor could be put in and initially debugged via front panel switches; however, this is a tedious process at best. Once you have ELM installed, you can use this tool to help build software and programs on your own machine to your heart's content.

Even though ELM is a fairly simple monitor as monitors go, it can be further simplified and condensed. As mentioned before, the decimal conversions can be omitted. The syntax checking can be reduced, the printing of addresses at the beginning of lines might be omitted, and the commands could be reduced to single letter codes. None of these simplifications will reduce the basic functions of the monitor; however, these features add a sharp dimension of utility and a touch of class to your monitor.

In many years of designing systems and studying human interaction with computers, I've found that people (ie: users, be they systems engineers or airline ticket clerks) think most efficiently in words and decimal numbers. Addresses are a sequential stream of numbers and we have all been taught since childhood to think of streams of numbers in decimal base. Only computer nuts, putting on airs, pretend to be able to think in octal or hex. Likewise, we communicate with each other in words. The computer is capable of communicating with us in our own language, so let it. An instruction such as LOAD STARTING AT 489 is much easier to learn and more efficiently used than L,01E9. The latter, however, is easier to implement in the computer. ELM compromises with LOAD,489; retaining the keyword and the decimal address. My basic philosophy is: *Let the machine do the things it is good at.* It is good at base conversions and word recognition. It can convert binary to decimal and back again in the twinkling of an eye; we can't. Remember, you will probably want to use your monitor for a long time; the extra effort in its construction will be well worth the frustration.■

Design an On Line

Robert R Wier
PO Box 9209
College Station TX 77840

James Brown
2518 Finley St No 636
Irving TX 75062

Machine or assembly language will most likely be used by many computer experimenters. While many professional programmers will swear by the use of assembly language, others, perhaps equal in number, will swear at it, preferring the use of high level languages. To those new to the field, these terms may seem confusing. It's really quite straightforward when one remembers that the language a machine uses differs considerably from the one used by the people. As one surveys a continuum from machine to human languages, the language most easily understood by the machine is a binary language; next on the continuum is assembly language with additional features that make it considerably easier to use, thus avoiding all night debug sessions, frazzled nerves, and 2 AM programming logic which hardly ever works, etc. For a good discussion on assemblers, see the October 1975 issue of BYTE. Easier yet for the programmer are languages such as BASIC, FORTRAN, PL/I, and ALGOL. These languages allow the problem to be stated and solved in terms better adapted to human understanding. Unfortunately, there are rather serious difficulties encountered when these high level languages are to be used on small systems. They require a compiler or interpreter to transform the problem from the high level language to machine language and more memory than is found in most small hobby systems. Therefore you'll probably be using assembly and machine language. After the program is written and loaded into the machine, experience has shown an astronomical probability against the program working correctly if it is more than two instructions large. Considerable time will probably be spent at the front control panel surveying the address and data lights, mumbling "I dontunnerstand" and "(expletive deleted) machine!". This can lead to terrific pains in the back and neck from bending over to look at the panel square in the face and operate the switches. This is commonly named "minicomputer neck."

How much nicer would it be to sit in a chair and do approximately the same thing using a Teletype or CRT display (CRT is an abbreviation for Cathode Ray Tube, essentially a TV picture tube. A television typewriter is a unit often used in this application).

There are several ways to use the control panel:

1: Executing a few instructions, then examining memory to see what the blinking machine is *really* doing, or
2: Inserting or changing data in memory, or
3: Displaying the contents of specific memory locations, or
4: Searching through memory for a specific bit string or number, if you prefer, or
5: Displaying and possibly changing the values in the CPU registers.

The authors had occasion to be working with a 16 bit/word minicomputer which mainly was used as a remote job entry terminal into a large computer. It could, however, function as a stand alone computer. Since an assembler was available, a number of assembly programs were written and debugged. When the machine was first delivered, a temporary control panel was provided. Since this was to be removed at some future date, the following technique was used to implement a DEBUG program using a CRT terminal to replace the control panel.

The basic idea is to develop a program that will take care of the functions outlined above and interface to the console terminal

Debugger

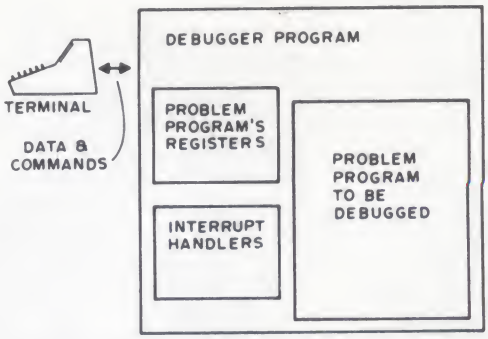


Figure 1: Logical arrangement of debugger.

and hopefully will protect itself from wild extremes of a program being debugged. This might be thought of as running a program within a program (figure 1). Hereafter, the program being debugged will be referred to as the problem program.

The debugger program must have provisions for a number of things. It has to handle the IO for the hardware and to converse with the human programmer. It has to keep track of the various status conditions of the program being debugged (the problem program). It must understand the input commands directing it to perform certain actions of the problem program. It must be transparent to the problem program so that when the final version is finished, the problem program may be loaded without the debugger, and still work.

In addition, the debugger should be small in size, and easy to implement to avoid the herculean task of debugging the debugger. (Although that's not strictly true. Once the IO and display portions were working, we used these to debug the rest of our debugger.)

The following commands are the results of our efforts to provide effective yet concise operations. In this list *adr* means a specific memory address, *val* a value, and *reg* a register.

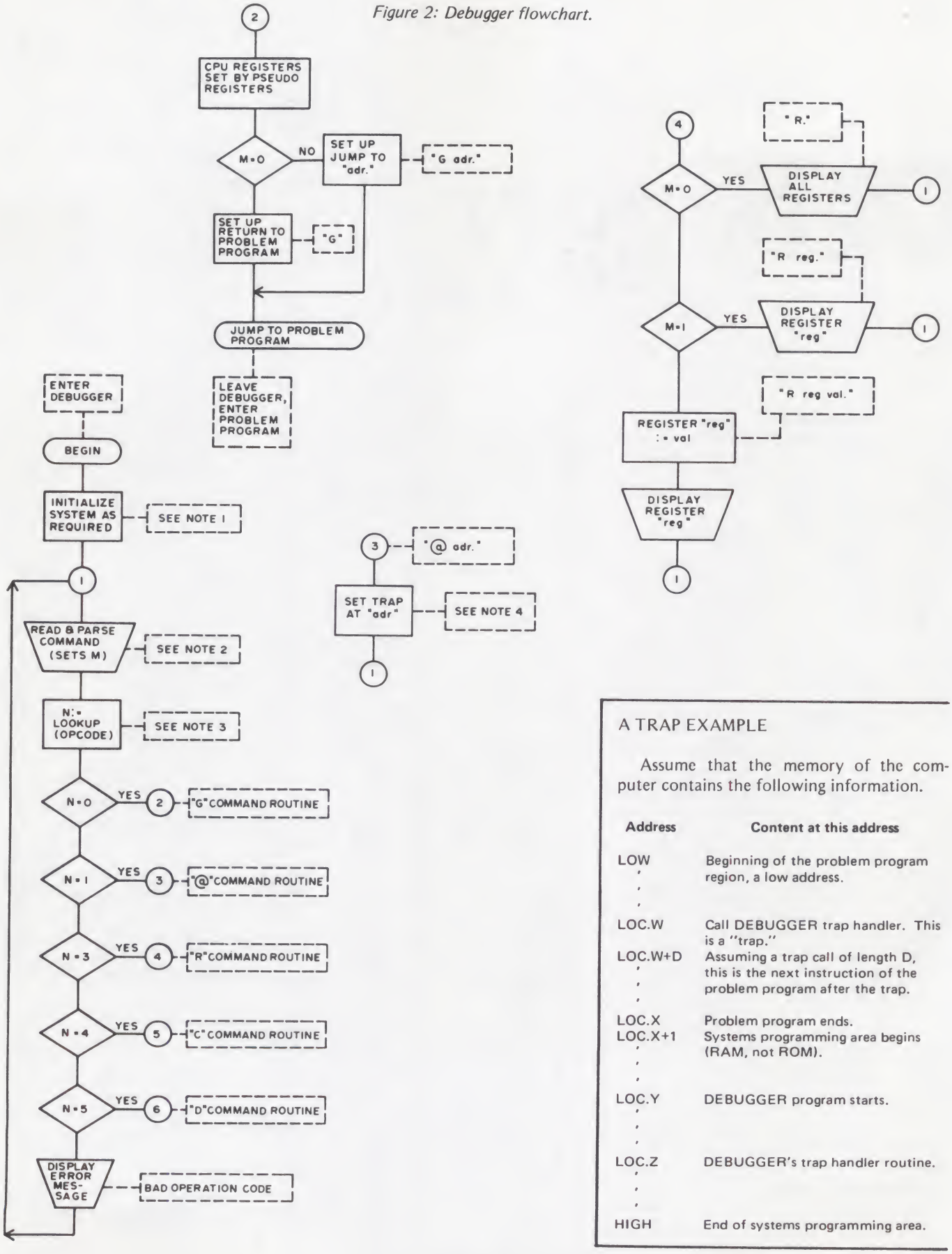
The final implementation including all the IO and interrupt handlers required 560 bytes, or about 256 instructions on the Lockheed SUE 1110 minicomputer. Figure 2 is an overview of the debugger logic flow. It is reasonably straightforward, except for the execute (G) instruction. Consider the debugger waiting for a programmer to enter a command. It just sits there wasting expensive electricity. As soon as you enter a command, the debugger checks it for vali-

text continued on page 60

Table 1: DEBUGGER program commands. Each command consists of an operation code character, followed by from one to three operands (numbers) separated by blanks. The command line is completed by a period. In implementing the program, the computer should respond with a carriage return and line feed after finding the period.

C adr val.	changes memory at adr to val
C adrl adr2 val.	changes memory from adrl through adr2 to val
D adr.	displays memory contents at adr
D adrl adr2.	displays memory contents from adrl through adr2 .
D adrl adr2 val.	searches memory from adrl through adr2 for val
R.	displays the contents of all registers
R reg.	displays the contents of register reg
R reg val.	changes the contents of register reg to val
@ adr.	sets return to debugger at adr in problem program
G.	go, i.e., continue or start execution of problem program using contents of the problem program's program counter register
G adr.	start execution of problem program at adr

Figure 2: Debugger flowchart.



A TRAP EXAMPLE

Assume that the memory of the computer contains the following information.

Address	Content at this address
LOW	Beginning of the problem program region, a low address.
LOC.W	Call DEBUGGER trap handler. This is a "trap."
LOC.W+D	Assuming a trap call of length D, this is the next instruction of the problem program after the trap.
LOC.X	Problem program ends.
LOC.X+1	Systems programming area begins (RAM, not ROM).
LOC.Y	DEBUGGER program starts.
LOC.Z	DEBUGGER's trap handler routine.
HIGH	End of systems programming area.

Note 1: The DEBUGGER program acts as a system monitor for your computer. Whenever the computer is restarted, the DEBUGGER is entered and will execute a power-on initialization sequence.

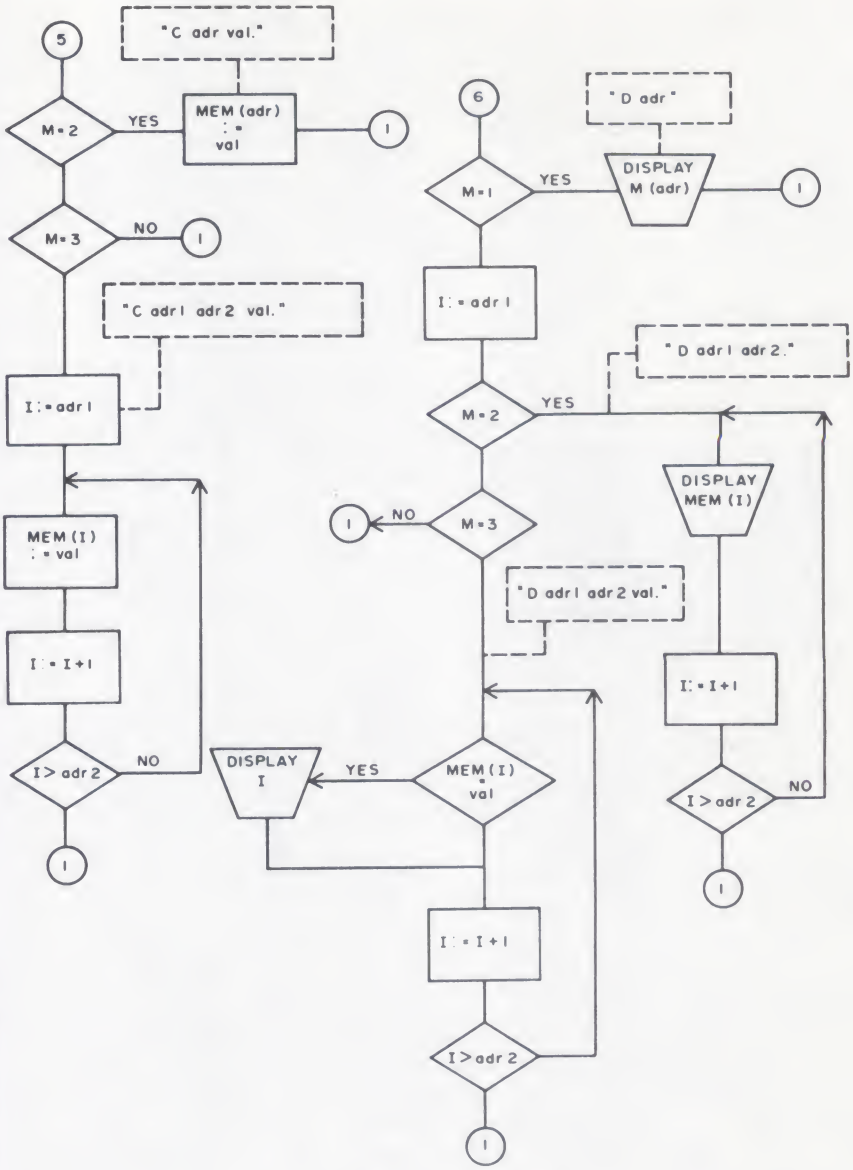
Note 2: The format of the command line and a list of all the variations on each command are found in table 1. The input routine should parse the command line by identifying the operation code and operands, stripping blanks, and counting the number of operands (M).

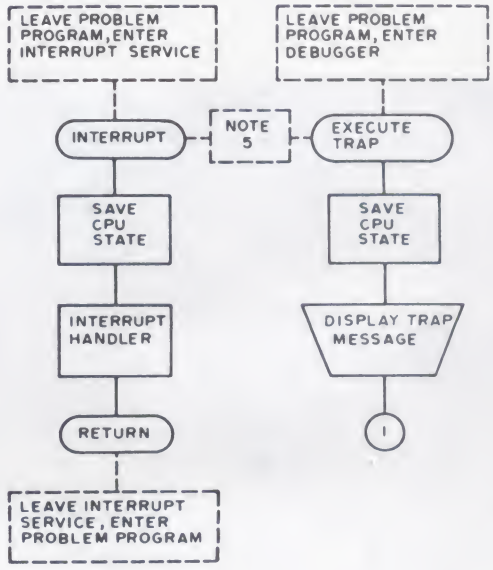
Note 3: The function LOOKUP is used to translate an input ASCII command character into a corresponding integer number. In the authors' system, this was accomplished by manipulating the bits of the ASCII character code; other schemes are possible.

Note 4: A trap is set by replacing the instruction at the trap address with a temporary alternate which causes a branch to the trap routine. The instruction used for this purpose in the authors' system was a "jump to subroutine." Depending upon the particular computer architecture, other instructions might be used, such as software interrupt, supervisor call, etc.

Note 5: Both trap instructions and interrupts require similar processing. One way to view the DEBUGGER program is as a large interrupt handler which is entered upon system restart, execution of a trap, or end of a problem program's execution.

Note 6: Command formats from table 1 are shown in quotes within comment boxes in this flow chart.





Assuming a stack oriented machine in which the state information is stored in the stack, the following sequence occurs in a typical case.

1. The user enters a program. After entering it, he decides to place a trap at location LOC.W in memory with the "@" command.
2. The problem program begins execution after a "G LOW." command. It reaches the trap at LOC.X and executes the subroutine call.
3. The subroutine call saves the address of the next instruction (at a minimum) and branches to the trap handler at LOC.Z. The trap handler continues the state saving process so that the machine's stack contains complete CPU state information.
4. The trap handler displays a trap message containing information on the address and register content of the machine at the time of the trap.
5. The trap handler passes control back to the DEBUGGER's command line interpreter.

text continued from page 57

dity, and if it is a legitimate command the various parameters are read and stored in memory to be accessed when necessary. Now the debugger looks at the part of the command line which tells it what to do (known as the opcode). Assuming that you are using ASCII, here is a sneaky way of determining which routine to go to.

1: Add 9 to the ASCII character,
2: Logically AND the opcode character with a 7,
3: Assuming the given opcodes (C,D,R,@,G), you now have a numerical index which you may use to test or use in a jump table to go to the proper code which accomplishes the desired function.

EXAMPLE: Suppose you have an ASCII 'R'; in binary this is:

```
0101 0010 - 'R'
0000 1001 - add 9
0101 1011 - AND result
0000 0111 - with 7
0000 0011 - final result is '3'
```

using this method then: G=0, @=1, R=3, C=4, D=5.

Now we offer a few comments on the various procedures shown in figure 2.

Change: This is perhaps the simplest of all the commands to implement. Using the last parameter supplied, step through memory from the first address zapping each location with the desired value until the ending address is reached (note: for a single address, adr1=adr2). Though not necessary, it is highly recommended to check the addresses for validity to avoid clobbering the debugger.

Display: Simply step through memory from the starting address to the ending address displaying memory contents as you go. We displayed in hexadecimal notation. You might alternately wish to use octal or (God forbid) binary. Since our CRT was capable of an 80 character line, we put 8 groups of 4 hex characters on each line:

```
LLLL: XXXX XXXX XXXX XXXX
      XXXX XXXX XXXX XXXX
```

The first number is the memory location of the lowest address displayed on the line (leftmost). Using this, it is easy to glance at the display and see patterns in memory.

For the search option of the display operation, you need only to print out the *addresses* where a compare was successful. You should be able to remember what you are looking for. When the search option is used, a flag is set which somewhat modifies the display such as:

```
: LLLL LLLL LLLL LLLL
  LLLL LLLL LLLL LLLL
```

where the L's are the memory addresses containing the argument.

Some commercial variants of the search operation allow you to look for certain bit patterns within words by masking out don't care bits; however, this is no small task to program for a feature of somewhat limited usefulness.

Register: Here you have three alternatives determined, once again, by the number of operands (i.e., how many parameters you specify with a particular opcode). No operands are used to indicate the display of all register contents. If one operand is present, then the content of that register only is to be displayed. Two operands indicate the contents of the specified register are to be changed to the given value.

Please note that these registers are really fixed memory locations, set aside inside the debugger (i.e., pseudo registers). These values are typically loaded into the CPU registers by the G command. Most CPUs have one or more general registers plus a program counter (i.e., the address of the next executable instruction), and a collection of indicators commonly referred to as status flags or sometimes as status registers. For our implementation we had seven general registers numbered (cleverly) one through seven. Register number zero was the program counter and register number eight was the status register (note: All registers were 16 bits large). Thus we only had to enter a single digit, zero through eight, to reference any register. On most micro or minicomputers, alphabetic type designators are used to reference registers, but with much luck a similar trick used to simplify opcode determination may be used.

GO and SET TRAP: This section is the most machine dependent implementation which requires very careful planning. The object here is to put the problem program into execution, and eventually have control returned *gracefully* to the debugger. The point where execution is to end and control to return to the debugger is called a breakpoint or trap.

Constructing a trap is not too difficult. The simplest method is to insert in the problem program an unconditional branch back to the debugger. A serious drawback of

Figure 3: How to set traps in the problem program (see text).

MEMORY MAP

Address	Contents
LO	Problem program starts
.	
.	
.	
W	Call debugger trap handler at address Z
W+1	Problem program continues
.	
.	
.	
X	Problem program ends
X+1	Stack starts
.	
.	
.	
X+n	Stack ends
Y	Debugger program starts
.	
.	
Z	Trap handler of debugger program
.	
.	
HI	Debugger program ends

ALGORITHM

The stack of n elements is located at address X, the debugger program at address Y, and the trap handler at Z. The following steps are executed:

1: The problem program executes a trap at location W, i.e., a subroutine call to the trap handler.
2: The subroutine call saves the address W+1. return address W+1 in the stack, e.g., in X+3
3: The trap handler at address Z is executed.
4: The trap handler fetches the return address W+1 from the stack (in this example X+3), reduces the stack by one element, and displays the address W+1.

this is that the location from which the branch occurred will be unknown. The solution is to use an unconditional subroutine call to the debugger. A call instruction places a return address somewhere, depending on the machine, and then branches to the location specified in the instruction. With this it is a simple matter to retrieve this return address as the program counter for the 'G.' option of the GO statement (figure 3). Our computer had fixed locations in which routine addresses could be placed, such that if certain types of interrupts occurred the return address was saved and a branch taken using the address at that location (vectored interrupts). One such interrupt was a "bad" instruction interrupt, hence the setting of program traps consisted of moving an illegal instruction to the location a trap was to occur.

The GO command should set the pseudo program counter if an operand is present, then load all general registers. The last two registers loaded are the status flags and the program counter (which would be identical to a branch). Typically a branch using the contents of the pseudo program counter would be used (note: Branches usually do not set or reset status flags).

In conjunction with the preceding, there should be a phantom routine which is the target for all traps. Its job is to save all registers and status before the debugger main routine uses them into the pseudo register area. It is suggested to display the program counter and the fact that a trap occurred, such as:

@ interrupt address

There is a dandy reason for this. If multiple traps exist, it is handy to know which trap was encountered. Additionally, since the trap itself may clobber one or more memory locations in the problem program, to remove a trap one must change these trap instructions back to the original contents (typically from the original assembly listings). In an earlier version of the debugger we allowed only one trap per execution and saved the good code from the trap location. When the trap occurred, we then restored the good code at that location. However, a serious drawback, of course, was that it isn't always known what branches will be taken between the G and @ instructions, and it was highly probable that the trap would be bypassed entirely. Thus in our present debugger we allow multiple traps but do not restore the previous code when a trap occurs.

Execute Instruction Considerations: If you happen to get tied up in an endless loop, you'll have to manually force a return to the debugger. This could be accomplished in several ways. You could physically reset the machine from the control panel (assuming you have one), and enter the debugger starting address. Or you could have previously set up an interrupt structure which would respond to some outside stimulus (such as an escape from the keyboard, or a special control panel switch) which would accomplish a branch to DEBUG. Some thought was given to simply kicking the power supply, initiating a power fail interrupt, but this was later discarded.

If you make extensive use of interrupts in the debugger (which is not really necessary) then you'll have to debug your problem program's interrupts separately. Otherwise the problem program's interrupts and the debugger's interrupts will be working at cross purposes.

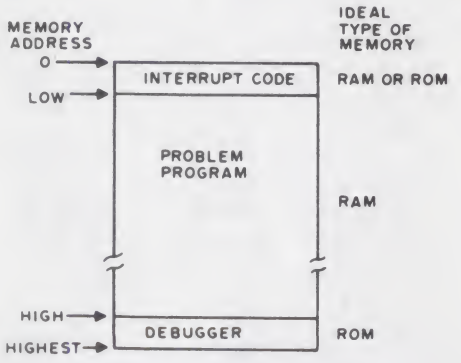
Should you place the breakpoint address in a branch of a conditional statement that doesn't happen to be executed, then the program will just skip around your break-

point. Or worse, placing the trap instruction as the operand of a multiword instruction could be distressing. The obvious solution to the first problem is the placing of multiple traps, so the problem program could not escape from the debugger regardless of the flow of control. The latter had no fool proof solution except exercising a little caution as to trap locations.

Some commercially available debuggers are really monitors that check the program counter every time a step is executed (interpreters). With a little thought it is apparent that this would involve considerably more programming than we've discussed here. Our debugger just allows you to set up the initial conditions and then "let fly," while the alternative is to have the debugger arrange every instruction which has the advantage of a more fool proof operation. But, it suffers from program complexity and a tendency toward slow execution which is critical in some IO operations.

The debugger ideally should be immune to anything which the problem program might try to do to it. This suggests the use of ROM (Read Only Memory). After you have the debugger working to your satisfaction, just place the debugger somewhere in your memory address space where you'll probably never need to move it. Usually this is in high memory. Since the debugger needs a small amount of RAM (Random Access Memory) in order to save the problem program's registers between G instructions, it cannot be made completely invulnerable. If the problem program happens to move garbage into the interrupt vectors, there is no telling what will happen on the next interrupt. But this is better than having the debugger completely in RAM. As a practical note, however, we found that there were not too many occasions when the problem program zapped the debugger if it was in RAM (figure 4).

Figure 4: Physical arrangement of debugger in memory.



If you want to get really fancy, you could include in the debugger an option to perform loading functions, such as retrieving a program off cassette tape. Assuming the debugger is in ROM you would never have to toggle in a bootstrap loader again, which is undoubtedly one of the worst aspects of small systems. Of course if you do not wish to get that fancy, you may still enter the loader via the debugger, which is certainly easier than using the front switches.

All in all, we've found that a good online debugger program is worth its weight in ROM. It will remove some of the worst aggravations of using small systems, and you'll learn a lot about logical flow of control, hardware software interfacing, and modularity of programming.

So let's get in there and STAMP OUT THOSE BUGS!■

Source listings of the debugger are available for the SUE 1110. Send one dollar to cover duplication and postage to Robert R Wier.

A version utilizing Intel's 8080 CPU chip is in the works, and when available a note will appear in BYTE.

You don't need high-powered compiler theory to process your own algebraic expressions – all you need are a few variations on one basic idea, developed in West Germany . . .

Processing Algebraic Expressions

W Douglas Maurer
University Library Room 634
George Washington University
Washington DC 20052

To the amateur programmer, algebraic expression processing may seem a formidable obstacle. How do you write a program which takes a character string like 2+3*(4–(14/7–1)) as input, and produces the right answer – in this case 11 – as output? The programmer seeking answers to such questions is usually led to a collection of sources on compiler theory, and to arcane-sounding terms like "recursive descent," "top-down and bottom-up parsing," and the like. These were developed for use by the compiler writer, although even compiler writers find much of compiler theory interesting for theoretical purposes only. The net result has been, in all too many instances, to scare the ordinary programmer away from algebraic expressions entirely – a decidedly unfortunate state of affairs.

Most people who do work with algebraic expressions in a small system setting have made use of what is called "Polish notation." Although we shall describe Polish notation next month, the warning must be given that Polish notation can be misused as easily as it can be used. The much more direct method which we shall describe was developed by F.L. Bauer and K. Samelson at the Technische Hochschule in Munich, Germany. We refer to it as the "Bauer-Samelson algorithm."

Before describing the Bauer-Samelson algorithm, let us first take up a number of elementary points about the processing of algebraic expressions. The input to any algebraic expression processor will, of course, be a string of characters. These are given in some sort of character-code format, and there are as many such formats as there are computers. Even the number of bits per character varies from one system to another. Some systems use five bits per character, some six, some seven, but most use eight – the standard IBM 360 (and 370) "byte."

Since there are 26 letters in the alphabet, at least 26 different codes must be used. To find out how many bits are needed to represent that many codes, we take the next higher power of 2, in this case 32, or 2^5. There are 32 different possible codes in 5 bits (from 00000 to 11111). Therefore 5 bits are enough to represent the 26 letters of the alphabet; whereas 4 bits would not be, because there are only 16 possible codes in 4 bits (from 0000 to 1111). If we wish to represent digits as well, we need 26 + 10 = 36 codes. Now five bits are not enough, and we must take the power of 2 next higher than 36, that is 64 = 2^6. There are 64 possible codes in six bits, and six bits are what is used on many big computers – the UNIVAC 1106, the CDC 6400, and the obsolete IBM 7094. (The PDP-10, DEC's biggie, has two character code schemes; one uses six bits, the other uses seven.) Once we have 64 codes, of course, we can represent characters other than letters and digits, such as +, –, *, /, =, parentheses, period and comma, and so on – known as *special characters*. Where five bit codes are used, the special characters include *shift characters*, analogous to the shift key on a typewriter,

enabling us to pass from one group of 32 codes (including the shift characters themselves) to another such group and back.

Once we know how many bits are in a character, the choice of the actual character codes is still quite variable from one computer to another. There is a "standard" code called ASCII, or American Standard Code for Information Interchange. But this, as its name suggests, is a standard code for information *interchange* (between different computers) only, and many individual computer makers continue to use their own code schemes.

All of the codes in common use, however, share certain characteristics. One of the most important of these is that the codes for the digits are all consecutive. That is, whatever the code for zero is (and this is quite variable), the code for 7, say, is 7 more than the code for zero. This is quite helpful to us in finding the binary equivalents of integers. Another common characteristic of character codes is that the codes for letters of the alphabet are given in numerical order (although not always consecutively). Thus the code for T, for example, will be greater than the code for R, because T follows R in alphabetical order; but it will not always be true that the code for T is 2 more than the code for R.

A sequence of characters is given in a character array. On a byte machine, character arrays are easy to index. As soon as we have loaded the first character in our array into a register, we add 1 to our index register (or indirect address location) and we are immediately set up to load the next character. If we are working on a machine which holds more than one character per word — such as a 16 bit or 18 bit machine — our best course, when processing character strings (of limited size) is usually to unpack them into a word array in which one character is contained in each word. This is illustrated, for a 16 bit machine, in figure 1. After unpacking, the characters may be processed in the same way as given above.

We will thus have an index in our program which is initialized to point to the first character in our array, and which is incremented, after we are through processing that character, to point to succeeding characters in the array. Let us now turn to the question of how these characters should be processed.

Suppose, first of all, that we load a character into a register and discover that it is a *digit*. Our first job should be to find out whether any of the characters immediately following this one are also digits. Since numbers are stored internally in binary form in almost all computers, a string of digits

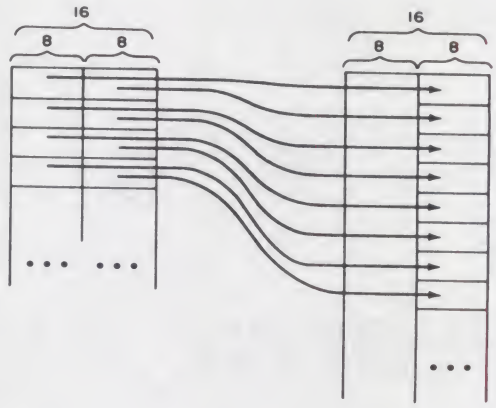


Figure 1: Unpacking characters on a 16 bit machine.

representing an integer will have to be converted to this form for further processing. Let us assume that we have a cell called NUMBER which is initialized to zero. Then our algorithm for finding the binary value of a string of digits is as follows:

1. Check the current character. If it is not a digit, stop.

2. Multiply NUMBER by 10; add the current character; and subtract the character code for zero.

3. Advance to the next character in the string and go back to step 1.

Thus for the character string 327, we perform 0 X 10 = 0, 0 + 3 = 3; then 3 X 10 = 30, 30 + 2 = 32; and finally 32 X 10 = 320, 320 + 7 = 327 — all in internal binary form. Each time we add the value of the next digit, which is equivalent to adding the character code for the digit itself and then subtracting the character code for zero.

Now suppose that, instead of a digit, we find the character code for a *letter* of the alphabet. The normal rule here is to look for letters and digits following this letter and to keep them in a string. Once we have found the first character that is neither a letter nor a digit, the letters and the digits that we have gathered so far constitute an *identifier*, which we may process further in a number of ways, depending on the way in which we are processing algebraic expressions.

As an example, consider the expression ALPHA*BETA+GAMMA*DELTA. We load the first character, namely A, into a register. Since this is a letter, we keep looking for

letters and digits, and we find L, P, H, and A. All of these characters are kept in a string. When we get to the *, the characters we have kept in our string so far constitute the identifier ALPHA. How we process this identifier depends on what it is supposed to mean. Is it a constant with a defined value? In that case the value is presumably in a table, and we can look it up. Or perhaps the purpose of processing this character string is to give this identifier some constant value. For example, suppose the string were ALPHA=3 and suppose we were processing this in such a way as to put the the value 3 in a table, corresponding to the identifier ALPHA. In this case, when we reach the character =, we can put ALPHA into our table, and then later put in the value.

Much algebraic expression processing involves *special identifiers*, or identifiers which are to be processed differently from the others — such as SIN, COS, and SQRT, or, perhaps, IF, STOP, and GOTO. All special identifiers should be collected into a table, and every time we recognize an identifier — that is, at the point in our program at which we have encountered a character that is not a letter or a digit, so that we know that the characters we have accumulated so far constitute an identifier — this table should be searched, to see whether any of its entries are equal to the current identifier. For each special identifier, we will then have a subroutine, or a section of our program, which handles it.

Let us now see what happens when, in the processing of our character string, we encounter an *operator* — a character such as +, —, *, /, or parentheses. This is where we use the Bauer-Samelson algorithm.

The Bauer-Samelson algorithm uses two *stacks* — one for operators and one for operands. Many programmers, although they understand the basic idea of a stack, have never actually written a stack-oriented program. The easiest way to do this is by using an array — call it S — together with a *current length* for the array, which we may call LS. At the start of our program, LS is set to zero. To put the quantity X on top of the stack, or, as we call it, to *push down* X on the stack, we perform

```
IF LS = MAX
        THEN GOTO OVERFLOW
LS: = LS + 1
S(LS): = X
```

where MAX is the dimension of the array S, and we transfer to OVERFLOW if we have *stack overflow*, that is, if the stack has grown too large. Pushing down X on the stack, of course, puts X on top of the stack while preserving all quantities on the stack that were previously put there. To take the quantity X off the top of the stack, or, as we say, to *pop up* X from the stack, we perform

```
IF LS = 0 THEN GOTO EMPTY
X: = S(LS)
LS: = LS — 1
```

where we transfer to EMPTY if the stack was empty when we tried to pop it. (This is usually not an error condition, but normally means that our stack process has terminated.) Of course, in the Bauer-Samelson algorithm, since we have two stacks, we may call them S1 and S2, with corresponding current lengths LS1 and LS2, both of which are initially set to zero.

In order to follow the operation of the Bauer-Samelson algorithm, we shall have to understand the basic idea of *precedence of operators*. Taking the five operators +, —, *, /, and ** (the last of these denoting exponentiation), we shall assign to each one a number called its *precedence*, as follows:

+	1
—	1
*	2
/	2
**	3

The easiest way to understand precedence is to consider what would happen if we didn't have it. For example, let us look at the expression 2*5+3*4. Suppose we tried to find the value of this expression in the following way: 2 times 5 is 10, plus 3 is 13, times 4 is 52. Clearly, this would be the wrong answer. What we want to do is to perform the multiplications first, namely 2 times 5 is 10 and 3 times 4 is 12, and then add together the resulting values, 10 and 12. Since we perform the multiplications *before* the addition, we say that multiplication (as an operator) has *higher precedence* than addition. The numbers which we have assigned to the operators reflect this fact; that is, 2 (the precedence of multiplication) is higher than 1 (the precedence of addition).

We shall now describe the basic operations of the Bauer-Samelson algorithm. The algorithm proceeds by scanning over the characters of the given string from left to right. Every time we encounter an *operand* — a constant or a variable — while we are doing this, we place it on the operand stack S1. Every time we encounter an *operator*, we try to place it on the operator stack S2. This is not done unless the *precedence test* is satisfied; that is, unless the precedence of the given operator is higher than that of the

operator at the top of the stack (or unless the operator stack is empty). If the precedence test is not satisfied, we remove an operator from the top of the stack S2, remove its operand or operands from the top of S1, calculate a result, and put this result back on S1. This is called *unstacking* an operator. We continue to unstack all operators from S2 until the precedence test is satisfied. When we reach the end of the entire original string, we unstack all operators from S2.

An example should make this clear. Suppose we have the string 2*5+3*4 mentioned earlier, and we are trying to calculate its value, which is in this case not 52 but 22. In the following analysis, we shall denote the contents of a stack by several quantities in parentheses, with the *last* of these quantities denoting the top of the stack. Thus (10,3,4) as the contents of the stack S1 means that the number 4 is at the top of this stack. The Bauer-Samelson algorithm proceeds in this case as follows:

1. The 2 goes on the stack S1.
2. The first * goes on the stack S2. (The precedence test is satisfied, since the stack S2 was empty.)
3. The 5 goes on the stack S1, which now contains (2,5).
4. We cannot put the + on the stack S2, because the precedence of + is lower than that of *. Therefore we unstack the * from the operator stack S2. This means: we take * off the stack S2, leaving S2 empty; we take its operands off the *top* of S1 (that is, we take 5 and 2 off S1); we calculate the result, namely 2*5 = 10 (the *second* operand of any operator is always removed from the stack first); and we put the 10 back on the operand stack S1, which now contains 10 and nothing else.
5. We are supposed to keep unstacking operators until the precedence test is satisfied. At this point, however, the precedence test is in fact satisfied, since the operator stack is empty, and we may therefore place a + on the operator stack and continue.
6. The 3 goes on the stack S1, which now contains (10,3).
7. The * goes on the stack S2, which now contains (+, *). The precedence test succeeded in this case, since the precedence of * is higher than that of +.
8. The 4 goes on the stack S1, which now contains (10,3,4). We are now at the end of the original string, and it is time to unstack all the operators from the stack S2.
9. The operator at the top of S2, namely *, is taken off this stack. Two operands are taken off the top of S1, namely 4 and 3; this leaves 10 on S1. The result, namely 3*4 = 12, is calculated and placed back on S1, so that S1 now contains (10,12).
10. The operator at the top of S2, namely +, is taken off this stack. Two operands are taken off the top of S1, namely 12 and 10; this leaves S1 empty. The result, namely 10+12 = 22, is calculated and placed back on S1.
11. The operator stack S2 is now empty; the Bauer-Samelson algorithm has finished; and the answer, namely 22, is on the operand stack S1. (Unless there has been an error, the Bauer-Samelson algorithm will always end with exactly one quantity on the operand stack, and this quantity will be the final result.)

This is the basic Bauer-Samelson algorithm. It may now be modified and extended in a number of ways.

Let us first consider parentheses. A *left* parenthesis is treated as an operator. It is always placed directly on the operator stack without making the precedence test; that is, it is treated as if it had the highest precedence. Once it is on the operator stack, however, it is treated as if it had the *lowest* precedence; that is, *any* other operator is placed directly above it on the stack, or, to put it another way, the precedence test always succeeds if there is a left parenthesis at the top of the operator stack.

A *right* parenthesis is treated somewhat like the end of the expression. We unstack all operators on the operator stack until we come to a left parenthesis, which we remove from the operator stack and continue to scan the given string. If there is no left parenthesis on the operator stack, there were too many right parentheses in the original expression. Conversely, if we come to the end of our string and start unstacking operators, and one of these is a left parenthesis, then there were too many left parentheses in the original expression.

As an example of the use of parentheses, we consider the expression 2+4*(5−(6−3))/8, the value of which is 3. We shall again "walk through" the Bauer-Samelson algorithm as it scans this string. This time, however, we shall use an abbreviated notation. In the second column below, marked action, we use one of the following codes:

O (Operand) — An operand is placed on the operand stack.

S (Succeed) — The precedence test succeeds, and therefore an operator is placed on the operator stack.

U (Unstack) — The precedence test fails (or else we are at the end of the expression, or at a right parenthesis), and thus an operator is unstacked.

Table 1: Calculation of the value of 2+4(5–(6–3))/8.*

Current Character	Action	Operand Stack	Operator Stack
2	O	(2)	Empty
+	S	(2)	(+)
4	O	(2,4)	(+)
*	S	(2,4)	(+,*)
(	L	(2,4)	(+,*,L)
5	O	(2,4,5)	(+,*,L)
–	S	(2,4,5)	(+,*,L,–)
(	L	(2,4,5)	(+,*,L,–,L)
6	O	(2,4,5,6)	(+,*,L,–,L)
–	S	(2,4,5,6)	(+,*,L,–,L,–)
3	O	(2,4,5,6,3)	(+,*,L,–,L,–)
)	U	(2,4,5,3)	(+,*,L,–,L)
	R	(2,4,5,3)	(+,*,L,–)
)	U	(2,4,2)	(+,*,L)
	R	(2,4,2)	(+,*)
/	U	(2,8)	(+)
	S	(2,8)	(+,/)
8	O	(2,8,8)	(+,/)
End	U	(2,1)	(+)
	U	(3)	Empty

L (Left parenthesis) – A left parenthesis is placed on the operator stack. (This is denoted by L in table 1.

R (Remove left parenthesis) – A left parenthesis is removed from the operator stack (this happens after unstacking, when the current character is a right parenthesis).

The operation of the Bauer-Samelson algorithm in this case can now be expressed by means of table 1.

Of course, the "current character" column in table 1 takes advantage of the fact that every operator and every operand in our example program consists of a single character. In a more general case, this column would be headed "current operator or operand."

Let us now consider unary operators. Superficially, there is no difference between a unary and a binary operator from our point of view, except that when we unstack a unary operator we must remove only *one* operand, rather than two, from the operand stack. However, when we put the unary minus sign on the operand stack, we must be careful to identify it as a unary, rather than a binary minus sign, so that we know how many operands to take off the stack S1. This in turn means that we are going to have to be able to tell the difference between a unary and a binary minus sign as we are scanning our string.

Table 2: Calculation of the value of –5–(–3–4).

Current Label	Current Character	Action	Operand Stack	Operator Stack
L1	–	S	Empty	(U)
L1	5	O	(5)	(U)
L2	–	U	(–5)	Empty
		S	(–5)	(–)
L1	(	L	(–5)	(–,L)
L1	–	S	(–5)	(–,L,U)
L1	3	O	(–5,3)	(–,L,U)
L2	–	U	(–5,–3)	(–,L)
		S	(–5,–3)	(–,L,–)
L1	4	O	(–5,–3,4)	(–,L,–)
L2	)	U	(–5,–7)	(–,L)
		R	(–5,–7)	(–)
L2	End	U	(2)	Empty

The simplest way to do this is to think of our Bauer-Samelson algorithm as having two basic labels, which we shall call L1 and L2. We start off at L1 (after all necessary initializations). When we are at L1, we are expecting to find an *operand*. If we find one, we put it on the stack S1, and go to L2. When we are at L2, we are expecting to find an *operator*. If we find one, it will be a binary operator; we put it on the stack S2 (after any necessary unstacking) and go back to L1. But if we are at L1 and we find an operator, it must be a unary operator; we put it on S2, after unstacking if necessary, and then return to L1.

Suppose now that we find a right parenthesis. Then we must be at L2 (if we are at L1, we have an error in the string we are scanning). We perform all necessary unstacking, remove a left parenthesis from the operator stack as described above – and then return to L2, since we are now expecting a binary operator. If we find an operand at L2, this is also an error condition. If we find a left parenthesis (of the type that we have so far described), we should be at L1; we put it on the operator stack and then return to L1, since we are again expecting to find an operand.

This interplay between L1 and L2 may be illustrated by the following example, containing two unary and two binary minus signs, in addition to parentheses. For the moment, we shall consider a unary minus sign to have precedence equal to 2. A binary minus on the stack S2 will be denoted by '–', a unary minus by U, and a left parenthesis by L. The codes in the "action" column are as in the preceding example. The string to be scanned is –5–(–3–4); its value, which is 2, is calculated by the Bauer-Samelson algorithm as in table 2.

It is, incidentally, a matter of controversy as to what the precedence of the unary minus should be. It should clearly be lower than that of exponentiation (thus –X**N is clearly –(X**N), and not (–X)**N) and higher than that of addition (thus –X+Y is clearly (–X)+Y, and not –(X+Y)). What about –X*Y, however? The two expressions (–X)*Y and –(X*Y) are equal, and the same is true of (–X)/Y and –(X/Y). It is not clear which choice leads to the greatest efficiency of calculation.■

Processing Algebraic

W Douglas Maurer
University Library Room 634
George Washington University
Washington DC 20052

In an article which appeared last month, we showed how the small system user can process algebraic expressions by using the Bauer-Samelson algorithm, developed by F L Bauer and K Samelson at the Technische Hochschule in Munich, Germany. The Bauer-Samelson algorithm has many variations, depending on the type of algebraic expression processing we wish to do. The most interesting of these have to do with the process of compiling.

Anyone who has ever thought about writing a compiler has probably already guessed that there are certain aspects to compiler writing that are not hard at all. Consider, for example, GO TO statements. If a compiler is reading, as input, a source program, and it comes to the words GO TO, it proceeds in a very simple manner. It reads the next few characters – let us say they are 305, so that the statement is GO TO 305 – and it writes, as output, whatever the machine code is for a transfer to some point in the object program corresponding to the label 305. The only part of this that is difficult at all is keeping a table of labels (such as 305) and their corresponding addresses. That can be somewhat complex, especially when GO TO 305 is a so-called *forward reference* – that is, when it *precedes* the label 305 in the source program.

A Single Register Machine

But all that pales into insignificance beside the problem of compiling code for algebraic expressions. Let us consider the simplest possible case. We have a machine with one register ("the accumulator") and six instructions as follows:

LDA	Load Accumulator
ADD	Add to Accumulator
SUB	Subtract from Accumulator
MPS	Multiply Single Register
DVS	Divide Single Register
STO	Store Accumulator

(We are assuming for the moment that our multiply instruction produces a single register result in the accumulator, and that our divide instruction divides a single register quantity in the accumulator by a quantity in memory. This restriction will be relaxed later on.)

Suppose now that we have an assignment statement such as

$$K = (I*J - I + J)/N$$

If this is read by the compiler as part of the source program, then the compiler must write out the equivalent machine language or assembly language code, which in this case would be

```
LDA  I
MPS  J
SUB  I
ADD  J
DVS  N
STO  K
```

or the corresponding machine language (absolute binary or octal or hexadecimal) code. How is this code to be produced?

Expressions Part 2

Once you know how to do basic processing on algebraic expressions, you can begin to learn how to write compilers.

We will now describe a modification of the Bauer-Samelson algorithm that produces such code. The main areas of modification are as follows:

(1) The "result" of a computation, which is calculated by the unstacking process, is no longer a number, but rather a *place* where the result of the computation is stored. For all of the instructions above (except STO), this will be the accumulator. Thus a special code to signify the accumulator will be placed on the operand stack.

(2) Every time unstacking takes place, output code is generated in addition to the calculation of the result.

Let us go through the above assignment statement as an example. (Refer to BYTE No. 6 for a general discussion of the Bauer-Samelson algorithm.)

(1) The left parenthesis goes on the operator stack.

(2) The I goes on the operand stack. (In this version of the Bauer-Samelson algorithm, we put *variables* — or pointers to them — on the operand stack, and not their values.)

(3) The * goes on the operator stack.

(4) The J goes on the operand stack.

(5) Now we cannot put the (binary) minus sign on the operator stack, because it has lower precedence than the * operator. So we must unstack the *. We take it off the operator stack, and its operands, I and J, off the operand stack; and now we must calculate a result.

Since I and J are the operands and * is the operator, we must generate code to multiply I by J. The code that does this is

```
LDA     I
MPS     J
```

or its machine language equivalent as above; and the result, I*J, is left, by these two instructions, in the accumulator. Let us denote the accumulator by $AC (the $ is there so that we cannot possibly confuse this with the name of a variable in the program, such as AC); then $AC goes on the operand stack. Now we can put the minus sign on the operator stack, directly above the left parenthesis.

(6) The second I goes on the operand stack.

(7) We cannot put + on the operator stack, because its precedence is equal to that of the minus sign, which we must now unstack. We take it off the operator stack, and we take its operands, I and $AC, off the operand stack. Remember that the *second* operand is taken off first; so the operands are actually $AC and I. What instruction performs the subtraction $AC — I? Clearly

```
SUB     I
```

is the one we want. (If the subtraction were I — $AC, this would have to be followed by another instruction which complements the value in the accumulator.) So the above instruction is generated; and, since it leaves its result in the accumulator, $AC is put back on the operand stack. Now we can put

An *interpreter* analyzes algebraic expressions every time a calculation is made; by carrying the process one step further we get a *compiler*, which analyzes the expression once while creating a specialized machine language program to do the calculations.

The Bauer-Samelson algorithm will always generate code from left to right. The result will not necessarily be as fast as optimal code by a human programmer.

+ on the operator stack, directly above the left parenthesis.

(8) The second J goes on the operand stack.

(9) Now we come to the right parenthesis. This means that we must unstack the + on the operator stack. Its operands are $AC and J (after reversing the order, as above); so, just as in step 7, we want to generate the instruction

```
ADD     J
```

and put its result register, namely $AC, back on the operand stack. Now the operator at the top of the operator stack is a left parenthesis; this is removed, leaving the operator stack empty.

(10) The / goes on the operator stack.

(11) The N goes on the operand stack.

(12) We are now at the end of the expression, and we must unstack the / and generate the instruction

```
DVS     N
```

This is done in the same way as in steps 7 and 9, leaving the operator stack empty and $AC on the operand stack.

There is often the problem of data types: If all data is in the form of n bit integers, this is not a problem; but when multiple types of data are allowed, mechanisms for conversion are required.

(13) Finally — and this is not, strictly speaking, part of the Bauer-Samelson algorithm — we look at the left side of the = for the first time, namely K, and generate the instruction

```
STO     K
```

to complete the generation of code in this case.

We have purposely picked a rather easy example, involving no temporary variables, no quotient register, and so on. This is by no means all there is to this version of the Bauer-Samelson algorithm, but the further refinements are not hard to visualize.

First of all, we must make sure that we can generate code for all possible cases. For a reason which will become apparent, the special symbol $AC will never be on the operand stack in two different places. So the operands of any given operator will always be in one of the following three forms: $AC and Y, Y and $AC, or X and Y. All of these cases have been treated, or at least mentioned, above. The first two cases are, of course, equivalent if the operator is + or * (since $AC + Y = Y + $AC and $AC * Y = Y * $AC).

The second case above, in which we may have to use more than one instruction (subtract followed by complement, for example, as discussed above) corresponds to the case in which a human being might generate different code from that generated by the algorithm. Suppose, for example, that our expression is A*D − B*C. A human being would generate the code to multiply B and C *first*, and later subtract it from A*D. The Bauer-Samelson algorithm, however, will always generate code from left to right. Ultimately, it will calculate B*C − A*D and then complement this, producing A*D − B*C. The resulting code will not, of course, be as fast as the code that a human being would generate. However, the difference in speed is minimal, and the so-called "optimization techniques" which allow computers to produce better code are probably too bulky to fit into your small system.

The above example expression, A*D − B*C, illustrates two further problems with compiling of expressions. The first is that of data types. If we use the FORTRAN conventions, A, B, C, and D are all real numbers, and we have to use floating point addition, subtraction, multiplication, and division. In many small systems, there are no real numbers, but the problem of data types may still remain. There may be 16 bit and 32 bit integers, signed and unsigned integers, and so on, each of which has its own instruction set. If mixed mode expressions are not allowed, we may determine the type of an expression as soon as we see the first variable in it, and make sure that we use only addition, subtraction, multiplication, and division instructions of that type.

(If we *do* allow mixed mode expressions, then it will be necessary to put a code — $REAL, $INT16, $INT32, or the like — on the operand stack along with *each* quantity placed there to record the type of that quantity. When we unstack, we must now generate code to add, subtract, multiply, or divide two quantities of the types given, and calculate not only the result — which we put back on the operand stack — but also its type. Thus all quantities on the operand stack, in that case, are pairs, each of which consists of a variable, register, etc., together with its data type.)

The second problem illustrated by A*D − B*C is the use of temporary variables. In our one register machine, we will have to store the value of A*D (or B*C) in a temporary location during the calculation. This store instruction is generated when we are trying to load a register — in this case the accumulator — whose contents cannot be destroyed, as evidenced by the fact that the symbol for this register is currently on the operand stack. To illustrate this process, we shall go through the above example in the same way as we did before. The code we will generate

for the evaluation of the expression A*D − B*C is

```
LDA     A
MPS     D
STO     TEMP1
LDA     B
MPS     C
SUB     TEMP1
COM
```

(where COM stands for "complement the value in the accumulator"), and this is generated as follows:

(1) A goes on the operand stack.
(2) * goes on the operator stack.
(3) D goes on the operand stack.
(4) We cannot put − on the operator stack, since it has lower precedence than *, which we must therefore unstack. We take * off the operator stack and A and D off the operand stack, and generate code to multiply A by D, just as before, that is,

```
LDA     A
MPS     D
```

Since the answer is left in the accumulator, we put $AC on the operand stack. At the same time we keep a pointer to this stack position in a special cell which we shall call ACSP (for $AC Stack Position). In this case, the pointer value is 1, since $AC is the *first* quantity on the operand stack (counting from the bottom). ACSP is initialized to zero, and whenever it is zero, it is assumed that $AC is *not* currently on the operand stack.

Now we can proceed to put − on the operator stack, which was left empty by the previous unstacking.

(5) B goes on the operand stack.
(6) * goes on the operator stack (since its precedence is higher than that of the operator at the top of that stack, namely −).
(7) C goes on the operand stack. We are now at the end of the expression and must unstack all the operators on the operator stack.
(8) First we unstack the *. Its operands are B and C, and it would seem that the code we should generate is

```
LDA     B
MPS     C
```

However, there is a problem. If we generate the first of these two instructions, we will be loading the accumulator and destroying its current contents, which we need. We can tell that we need the current contents of the accumulator because ACSP is not zero. In fact, the quantity currently in the accumulator is the value of A*D.

Therefore the rule is as follows: Whenever we are about to generate a *load* instruction, we first check to see if ACSP is zero. If it is, we may proceed. If it is not, however, we must generate another instruction to store the accumulator into a temporary cell − in this case, TEMP1. The name TEMP1 is now put on the operand stack *in place of* $AC. It is *not* (necessarily) put on the top of that stack. Instead, we look at the pointer to see where to place it. In this case, the pointer value is 1, so that TEMP1 becomes the *first* element on the operand stack (counting from the bottom), which is where $AC was before. At the same time, ACSP must be set to zero, denoting the fact that $AC is no longer on the operand stack.

(In some algebraic expression evaluations, we will need more than one temporary cell. Let us call these TEMP1, TEMP2, TEMP3, etc. We have a temporary cell counter which is initialized to zero. Every time we need a new temporary cell, as above, we may increase this counter by 1. Alternatively, we may check the operand stack to see what temporary cells are currently on it, and pick out a new one in this way. If a *new* temporary cell is needed, it cannot be currently on the operand stack; however, its choice is otherwise unrestricted.)

Evaluation of expressions on a single register machine often requires use of temporary operands in memory.

Let us assume, therefore, that we have generated

```
STO     TEMP1
```

followed by the two instructions as above. The result of these two instructions is left in the accumulator, so $AC goes back on the operand stack. Note that the contents of the operand stack were $AC, B, and C, with C on top; then $AC was changed to TEMP1 and B and C were taken off. Now with $AC put back on, the contents of this stack are TEMP1 and $AC.

(9) Now we unstack the −. Its operands, as given above, are TEMP1 and $AC. If the − were a + we could simply generate

```
ADD     TEMP1
```

and we would be done. However, as it stands, we have a problem, because simply generating

```
SUB     TEMP1
```

would perform the operation $AC − TEMP1, rather than TEMP1 − $AC. We may

Generation of code for expressions evaluated on multi-register machines can use the extra registers as temporary storage; however, this introduces the need to keep track of register usage and special cases.

notice that $AC − TEMP1 = −(TEMP1 − $AC), and, therefore, the instructions

```
SUB     TEMP1
COM
```

will perform the subtraction we want.

Multi-register Machines

The use of the special cell ACSP as above is a special case of the use of a *table of register contents*. In general, a computer will have more than one register. For *each* such register (that participates in the instructions to be generated), we will need a location like ACSP. All these locations are initialized to zero at the start of evaluation; each time one of them is used, it will appear on the operand stack, and a pointer to its position there will be kept in the location corresponding to that particular register. In some cases we may simplify matters and keep only Boolean values (zero or one) in these locations; we can always search the operand stack, if we have to, to find where a register is on that stack, if the corresponding Boolean value is 1.

As an example, suppose that we have a more typical kind of multiplication instruction which leaves a double word answer in the accumulator and a *quotient register*, which we shall denote by $Q on the operand stack. If the quantities being multiplied are integers and the quotient register is the least significant part of the double register result, we can assume that the result goes in the quotient register and put $Q on the operand stack immediately after generating a multiply instruction. This in turn means that we have to be prepared to accept $Q as an operand. If X and $Q, for example, are the operands of + (which is being unstacked) and there is no instruction to add X to the quotient register directly, there may be an instruction, which we can generate, to move the contents of the quotient register to the accumulator (or to exchange these two registers), after which we can generate an instruction to add X in the normal way. Of course, in this case, we have to check the location corresponding to $Q before we generate a multiply, to see whether the quotient register has to be stored in a temporary location.

In a multi-register machine, we will not usually need any temporary cells. (By a multi-register machine, we mean here, specifically, a machine with more than one *arithmetic* register, in which addition, subtraction, multiplication, and division can take place.) All we need is to make sure, whenever we load a new register, that this register is not already being used for some other purpose. Let us illustrate this by considering again the expression A*D − B*C. The code generated for this, on a multi-register machine, would be roughly as follows:

(1) Load some register U with A.
(2) Multiply D by the contents of register U.
(3) Load some other register V with B.
(4) Multiply C by the contents of register V.
(5) Subtract one register from the other.

Here the registers U and V will have corresponding stack position variables, which for the moment we shall call USP and VSP. At the beginning, these are set to zero. When it comes time to generate the first two instructions ("load A" and "multiply by D"), we search for a register in which this computation can be performed. We find that it can be performed in U (because USP = 0), and so we generate instructions which make use of the register U. At the same time, we set USP to indicate that register U is now in use (assuming that the multiply instruction leaves its answer in U). Now, when it comes time to generate the next two instructions ("load B" and "multiply by C"), we again look for a register that we can use. This time we determine that we cannot use U (because USP ¬= 0); so we have to keep on looking. If V is the next register that we look at, we see that we can use it, since VSP = 0, and so we generate the third and fourth instructions above in such a way that they make use of the register V.

Stack Machines and Lukasiewicz Notation

Besides conventional single register and multi-register machines, there are *stack machines*. A load instruction on a stack machine puts the quantity to be loaded on the top of a stack of registers; a store instruction removes (pops up) the quantity to be stored from the top of this stack. An add instruction removes two registers from the top of the stack, adds their contents, and puts the result back on top of the stack. Thus a stack machine effectively performs about half the Bauer-Samelson algorithm in

hardware, and the generation of code for such a machine is considerably simplified. We shall now describe how this is done.

The code for calculation of an algebraic expression on a stack machine is directly related to the form of that algebraic expression expressed in *Polish notation*. (The proper name for this is Lukasiewicz notation, but it is popularly called Polish notation because very few people can pronounce Lukasiewicz — an English approximation, however poor, is WOO-kah-SHEV-itch. Other names for Polish notation are "suffix notation" and "reverse Polish." There is also "prefix notation" or "forward Polish," but this is never used in this context in computing.)

The Polish notation equivalent of a one operator expression, such as A+B or C–D, is formed by taking out that operator and putting it at the end: A B + or C D –. The Polish notation equivalent of a more complex expression is formed by breaking it down into parts, normally two parts with an operator between them; this operator is placed at the end, and the two parts are themselves expressed in Polish notation. Thus for (A+B)*(C–D), the two parts are A+B and C–D, or, in Polish notation, A B + and C D –, and the operator is *, so the entire expression is A B + C D – *. Similarly, A*D – B*C is expressed in Polish notation as A D * B C * –.

Conversion of an algebraic expression in ordinary, non-Polish (or, as it is often called, *infix*) notation into Polish notation may be performed by using a drastically simplified version of the Bauer-Samelson algorithm. There is only one stack, namely the operator stack. Operands, instead of being put on a stack, are put directly on the end of the string in Polish notation which is being constructed. As an example, let us go through the string A*D – B*C once again, according to this version of the algorithm:

(1) A goes on the end of the string.

(2) * goes on the stack.

(3) D goes on the end of the string, which is now A D.

(4) * is unstacked, that is, placed on the end of the string, which is now A D *.

(5) – goes on the stack.

(6) B goes on the end of the string, which is now A D * B.

(7) * goes on the stack (as before, since its precedence is greater than that of –).

(8) C goes on the end of the string, which is now A D * B C.

(9) * is unstacked, so the string is now A D * B C *.

(10) – is unstacked, so the string is finally A D * B C * –.

Once a string in Polish notation (or "Polish string") has been formed, code to calculate the value of the corresponding algebraic expression may be generated directly. Each operand corresponds to a load instruction, and each operator corresponds to an instruction which implements it. Thus in the above case the instructions would be: Load A; load D; multiply; load B; load C; multiply; subtract. The first two of these instructions load A and D onto the register stack in our stack machine. The next instruction takes A and D off the stack and puts A*D back on. The next two instructions put B and C on the stack of registers, which now contains A*D, B, and C. The next multiply instruction takes B and C off the stack and puts B*C back on; the final subtract instruction takes A*D and B*C off the stack and puts A*D – B*C back on. After this code, we can have an instruction to store the result, and this instruction leaves the register stack the way it was before expression evaluation started.

A stack machine effectively performs half the Bauer-Samelson algorithm in hardware, so code generation is considerably simplified.

Polish notation is also often used in *interpreters*. An interpreter is like a compiler, except that no code is generated; instead, the interpreter actually performs the indicated instructions as it goes. Typically, an interpreter will go through an initial phase (often called, confusingly, a "compiler phase") in which the program to be interpreted is read, and all expressions converted to Polish notation (among other things) and stored internally in this way. The actual interpretation now follows, with the interpreter moving from one statement of the interpreted program to the next, doing what each statement says and proceeding to the interpretation of whichever statement comes next in logical order. (Thus if it is interpreting IF K=0 THEN GO TO ALPHA, and K is in fact zero, then the statement labeled ALPHA will be interpreted next.) The advantage of keeping expressions in an internal form corresponding to Polish notation, rather than ordinary infix notation, is that Polish notation may be evaluated much more efficiently than infix notation. All we have to do to evaluate a Polish string is to simulate the effect of a stack machine, as outlined in the preceding paragraph. That is, we go through the Polish string from left to right; whenever we come to an operand, we place it on a stack, and whenever we come to an operator, we act as if we were unstacking it. This is the "other half" of the Bauer-Samelson algorithm; like the algorithm given above to convert a string into Polish notation, it uses only one stack, but this is an operand stack rather than an operator stack.■

The "My Dear Aunt

Robert Grappel
MIT Lincoln Laboratory
Lexington MA 02173

The number of mathematical operations a computer can perform without the aid of programming is quite small. The bare machine can add and subtract, and perhaps it can also multiply and divide. It cannot comprehend a series of operations, nor can it evaluate a mathematical expression as a human would typically write it. It cannot group operations as required by the rules of mathematics. All these require software; programs which convert mathematical statements to sequences of machine instructions. This article describes a set of programs which can read a mathematical statement in its normal form and evaluate its result. The discussion is kept on a general level: no specific machine or language structure is assumed. These programs should find their way into many "do-it-yourself" assemblers, compilers and interpreters .... wherever it is useful to write mathematical expressions for input to a computer.

My Dear Aunt Sally

Shortly after students learn to do the basic operations of addition, subtraction, multiplication, and division (the same operations that bare computers can perform), they are faced with problems of the following kind: 3−5*2=?. A student with imagination finds such a problem paradoxical; there are several apparently "correct" answers. Performing the operations in the order of appearance, he gets −4 (scanning left-to-right). Performing the multiplication first, he gets −7. If he chooses to scan from right to left, then the results +4 and +7 are possible. Which is correct?

My teacher gave us a rule to remember how to proceed with these problems — multiply, divide, add, subtract — or, "My Dear Aunt Sally" as we came to remember this order. Mathematical statements are read from left to right, for each operation. The evaluation starts with all the multiplications, left to right. It then proceeds in the order of "my-dear-aunt-sally," evaluating all the divisions, then all the additions, then all the subtractions. Mathematicians call this ordering the "precedence of functions," and all mathematical operations can be ranked in the order in which they are to be performed. Hence, the example has only one correct result, and this is −7.

Unfortunately, precedence is not enough to force a single answer from every problem. Suppose one wishes to perform one operation upon the result of a group of several other operations, some of which are of higher precedence. One needs some mechanism to group certain parts of a mathematical statement so that they can be considered as a single unit to be treated by some other operation. For this purpose, mathematics uses parentheses. If one wrote the example from above as: (3−5)*2=?, it is clear that the subtraction should be performed first, in spite of the precedence of the two operators. With the two tools of precedence and parentheses, one can force the desired ordering of operations upon a mathematical statement and ensure that there will be only one correct result.

What Is a Parser?

Now, how can a computer deal with complex mathematical statements like the example? The computer can perform each operation individually, either by a single

Sally" Algorithm

elementary operation or by calling a subroutine. The problem is the same one that we faced as students; how does one break down a complex mathematical statement to perform the individual operations in the right order? The name for this operation of breaking down a complex statement into its component parts, determining the structure of the statement, and evaluating it as required, is "parsing." This article describes a set of procedures which together form a parser.

"My Dear Aunt Sally" is a precedence ordering rule.

Tokens

At the outset, the computer sees a mathematical statement as a string of characters. All that is known about the string is its starting address and its length. The statement

X−2/(YAXIS+Z)

is a string of 13 characters at some address. One of the first things that is necessary is a procedure to subdivide this string into its elements: variables, constants, operators, and parentheses. The example contains three variables: X, YAXIS, and Z. It contains one constant: 2. There are three operators: −, /, and +. There are also two parentheses. Each of these elements is a character string. These strings may be of differing lengths. There may be blanks embedded in the input string, but these are not desired in the element strings. The procedure which subdivides the input string and eliminates blanks is called NEXTOKEN. Each element of the input string is called a token. The first problem of constructing a parser is to find a way to inform a computer about the tokens contained in the character string representation of a mathematical statement.

A parser is a programming scheme to analyze statements.

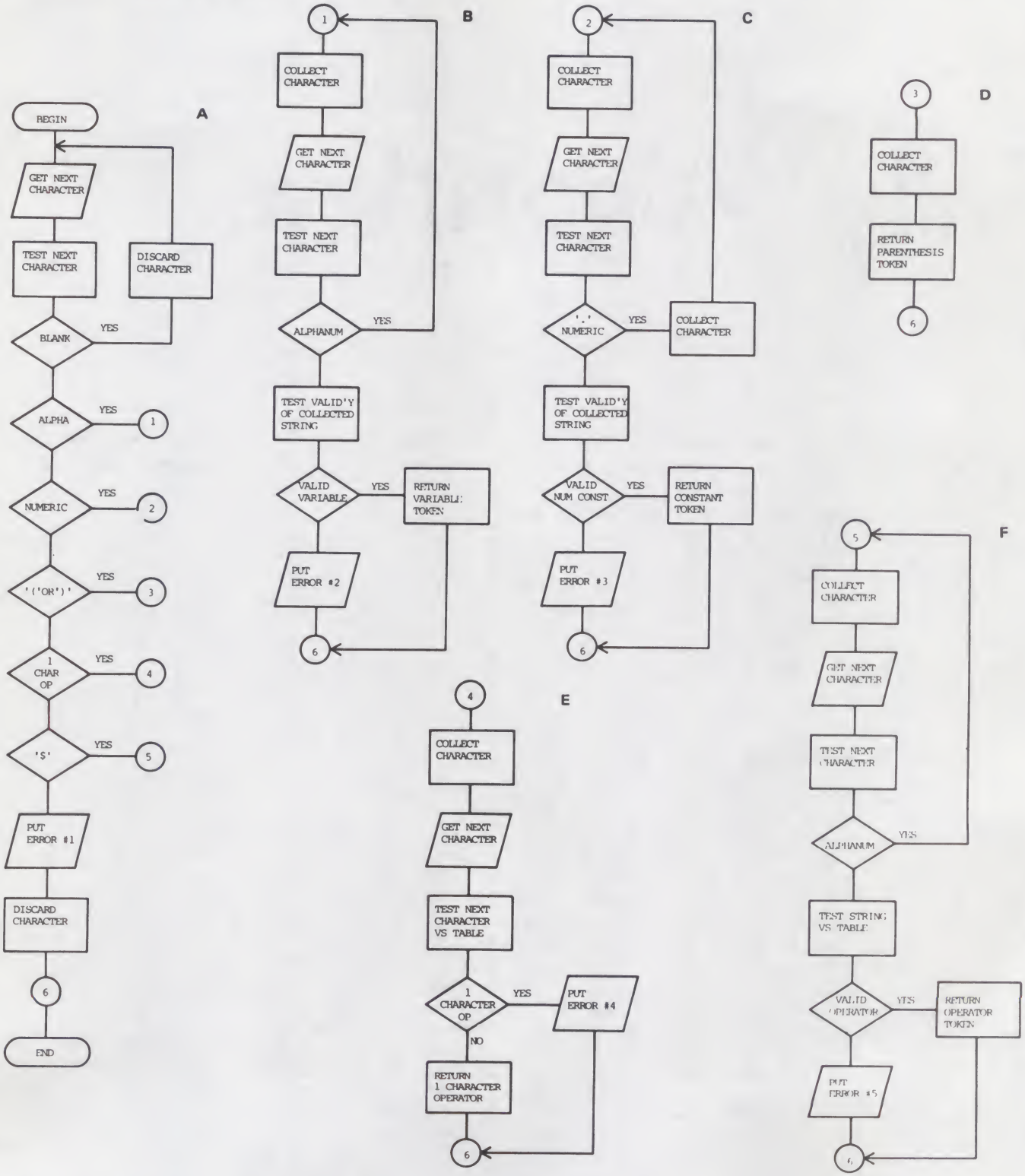
Blanks as Separators

There are several ways to approach this problem. Perhaps the easiest (in the sense that the coding of NEXTOKEN is simplest) is to require that the writer of a mathematical statement put at least one blank between every element or token in the statement. In this way, the human programmer breaks the input string into tokens before the computer gets it. We would require that the above statement be written as

X − 2 / (YAXIS + Z)

where all the extra spaces are required. With this method of token separation NEXTOKEN would work like this: Starting at the last character processed (the leftmost one at the start of the string), NEXTOKEN scans the input string from left to right until a blank occurs. This substring (from starting point to blank) is the next token. The next step is to determine what type of token has been scanned. The rule that will be followed in this article is that the first character of a token determines its type. If the first character is alphabetic, then the token is a variable. If the first character is a digit or decimal point, then the token is a constant. If the first character is neither of the above, then it is checked against a table of legal operators. If it is not an operator, then it is checked to see if it is a parenthesis. Variables might be checked for invalid characters in their names or too many characters in the name; constants might be checked for non-numeric

Figure 1: Flow chart of the NEXTOKEN algorithm used in this design. (a) The main routine. (b) The variable name collection algorithm. (c) The numeric constant collection algorithm. (d) Parenthesis handler. (e) Single character function name handler. (f) Generalized function name handler.



A character is fetched by the "get next character" operation. This character may be collected into an output string by NEXTOKEN, or discarded if it is to be ignored. If it is neither collected or discarded, the character will be reused the next time the "get next character" operation is performed. Flow is from left to right or from top to bottom unless an arrowhead indicates a different flow direction.

characters or more than one decimal point. A token which failed to match the model of any of the four token types would be flagged as an error, as would any of the error conditions described above. A check for string length of an operator might be used: The string '+=VAR' is not an operator, despite its first character.

A Smarter Token Separator

Requiring blanks around every token of a mathematical statement not only takes up valuable memory space but also makes the parser very susceptible to programmer errors. It is far too easy to forget one of those critical blanks. Fortunately, with a slightly more complicated mechanism for NEXTOKEN, one can parse a randomly written statement with any spacing. This algorithm is flowcharted in figure 1.

The routine starts by always scanning and ignoring any leading blanks. Eventually it finds a non blank character, the first character of some token. Remember that a rule was established for this parser: The first character of a token determines its type. Once the token type is known, the NEXTOKEN routine checks that subsequent characters are valid for that token type. As soon as a character is found which is invalid for that token type, the token is completed. For example, consider the expression X+5. The first character (leftmost) is alphabetic; this means that the first token is a variable. The second character is not alphanumeric, so it is not part of the variable type token. Hence, the first token is X which is a variable. The next token starts with a + which is found in the operator table. Hence it is an operator. The next character is the digit 5. This is the start of a constant type token. There are no further characters, so the token is complete.

Is It an Operator or a Variable?

One useful extension to the algorithm should be considered here. Many operator names that we would like to use are not single symbols, but are several characters in length. One might for example want to call the cosine operator by the name COS. Unfortunately, the simple minded NEXTOKEN procedure would confuse this with a variable named COS. There are several ways around this problem. One is to define new symbols for each operation added to the system. These are added to the character tests for operators and the parser will work fine. This makes for clumsy notation, however; and there may not be enough distinct characters available in keyboards, Teletypes, etc. A second approach is to require a special character, such as the dollar sign ($), as the first character of the string desired as an operator name. The cosine function might therefore be named $COS. The dollar sign would disqualify the name as a variable and identify it as a candidate for operator status. NEXTOKEN would then check the remaining characters in the name (using the same rules as for variables) against a table of operator names. If a match is found, then the string is an operator. The third approach is to forego the identifying first character in operator names and to treat operator names as a special kind of variable in NEXTOKEN: When a character string is typed as a variable, it is then checked against the list of operator names. If a match is found, then the token is changed in type from variable to operator after further statement analysis. Since this is complicated, NEXTOKEN assumes the dollar sign as identifying character for extended operator names. We see that NEXTOKEN starts with a character string, a starting pointer within that character string, and the position of the end of the string. NEXTOKEN returns a character string which is the new token, an indication of the token type, and the token length. It leaves the input string starting pointer with a new value after collecting or discarding each character needed to build the current token.

Some Small Procedures

There are several small procedures which are necessary to the parser and which are briefly described now. Two of these are required to convert the character strings, which are the tokens, into their values. One of these, called CONST, works on constants. The other, called VARIABLE, works on variables. Routines like these are usually available in a large computer's operating system. For minicomputers, the algorithms can be extracted from programming texts and programmed for software deficient home brew systems. The mechanism of VARIABLE depends on the structure chosen for the symbol table used to store variables. There should be some form of check that a variable has a value before it is used. If it has no value, an error message should be generated. Another routine needed is a form of branch table to convert the character string name of an operator into a call to the proper subroutine to perform the operation.

A mechanism for generating the precedence of operators is also needed, as was demonstrated in the introduction to this article. This amounts to a table of precedence values indexed by the operator name. Every legal operator is assigned a prece-

Computing the value of a statement is often easier if the statement is first rewritten in a form better suited to computers. Polish notation is such a form.

Operator	Precedence
Trigonometric functions, Logarithmic functions, Roots, etc.	7 or more
↑ (Exponentiation)	6
* (Multiplication)	5
/ (Division)	4
+ (Addition)	3
− (Subtraction)	2
parenthesis	1
empty stack	0

Table 1: Precedence of operators. This is a typical mathematical precedence of operations.

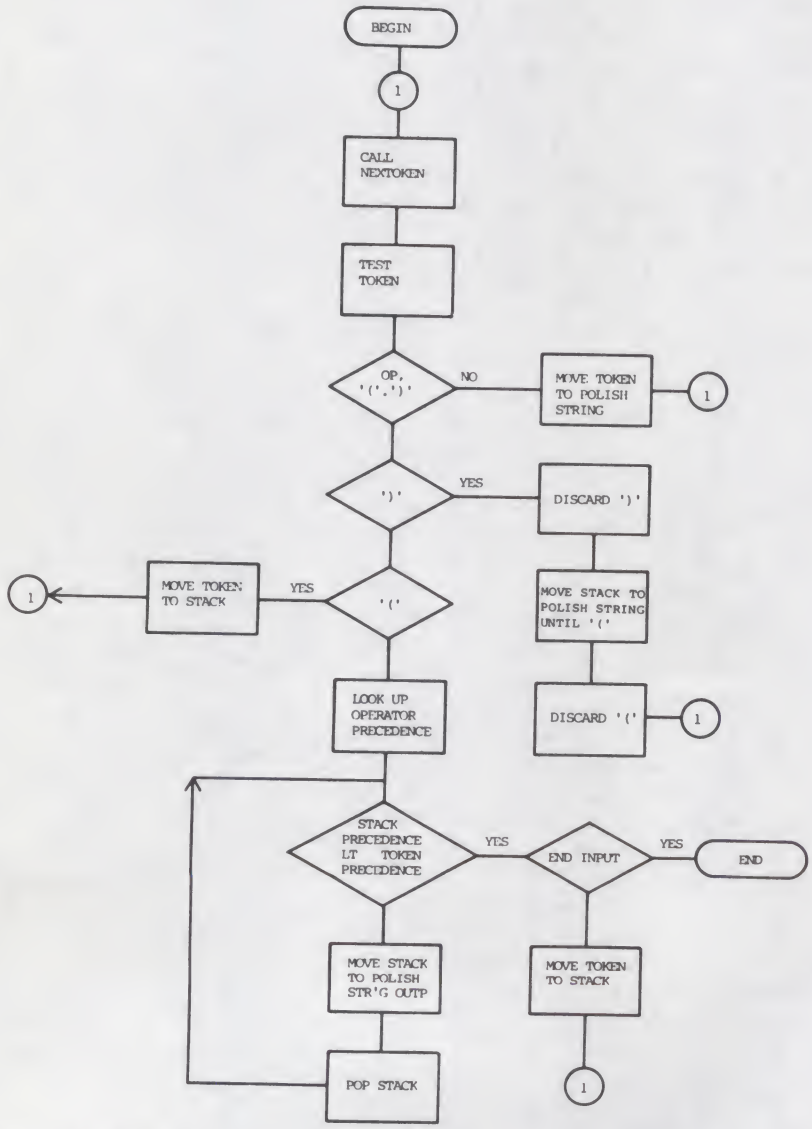


Figure 2: The POLISH routine specified as a flow chart. This routine uses NEXTOKEN to obtain individual tokens of the syntax, then uses its logic to rearrange these tokens in Polish notation. The output of this routine is a Polish string.

dence, as are parentheses and the right and left terminations of the input string. This article assumes that the higher the precedence value, the earlier the operation should be done. In other words, the operator with highest precedence goes first, the second highest goes next, etc. Table 1 is a typical precedence table. This precedence lookup procedure is called PRECEDE.

Intermediate Results

As shown in the introduction of this article, it is often necessary to evaluate a mathematical statement in a different order than that in which the statement was written. "My Dear Aunt Sally" does all the multiplications first, then the divisions, then the additions and subtractions. This parser operates by rewriting the statement into a more desirable form. In the new form, the operators are performed in the order that they appear. There are no parentheses required in the new form. The operands (the variables and constants) and operators are located in a simple and consistent relationship to one another in the new form. This makes the evaluation algorithm easier to write and more efficient. This "nice" form for writing mathematical statements was devised by the Polish logician J. Lukasiewicz. The next item to consider in this parsing system is a procedure to rewrite the input character string into Polish notation. This procedure, called POLISH, uses NEXTOKEN to disassemble the input string when required by its algorithm. POLISH also uses PRECEDE to compare the precedences of operators in different parts of the string. Figure 2 shows a flowchart of POLISH. This procedure forms the heart of the parser and is described in some detail. Several examples are also worked through. Once a mathematical statement is in Polish notation, it is rather easy to evaluate it.

A few examples of Polish notation and its evaluation are in order here. Consider the statement A*B+C*D. We know by precedence that the multiplications are to be performed first. In Polish notation we choose to write the operands first, followed by the operator. Hence, A*B becomes AB*. Similarly, C*D becomes CD*. Then the addition of these quantities is written AB*CD*+. This is the Polish form of the example statement. This string is evaluated as follows: Starting at the left, operands (variables and constants) are passed directly into a software pushdown stack in memory, which is a temporary holding area organized so that the first item in the stack will be the last item out of the stack. (This stack should not be confused with the subroutine call and return stacks of many microcomputers.) The

stack will hold the *values* of the operands, as returned by the procedures VARIABLE and CONST. When an operator is encountered, it is applied to the top two values in the stack. The result of the operation is returned to the stack in place of the operands. Figure 3 shows a flowchart of the evaluation procedure, called EVAL.

Let us proceed to evaluate the example AB*CD*+. First, we place A, which is a value, on the stack. Then we place B on the stack. Next, we encounter the operator *. The top two values on the stack are A and B, so we compute A times B and return that value to the stack. Next, we put C on the stack. Next, D goes on the stack. Then the operator * is encountered again. The top two entries on the stack (the last ones entered) are C and D. We compute C times D and return that value to the stack. The stack now holds two values which are the products of A and B, C and D, respectively. Finally, we encounter the operator +. We perform the addition and then we are done. In a similar manner, the first example of this article, 3–5*2, would be written as 352*– in Polish.

POLISH starts by calling NEXTOKEN for the first token. If it is not an operator or parenthesis, the token is output to the Polish string and NEXTOKEN is called for the next token in the input string. If the token was a left parenthesis, the parenthesis is placed in the stack and NEXTOKEN is called again. If the token was a right parenthesis, the contents of the stack are moved to the output Polish string until a left parenthesis is encountered or the stack is empty. Both the left and right parentheses are deleted and NEXTOKEN is called. Parentheses must occur in left-right pairs – if there is no left parenthesis in the stack after a right parenthesis is found, there is an error and the string cannot be parsed. If the token was an operator, then its precedence is checked against the precedence of the top of the stack. If the new operator is of lower precedence than the top of the stack, the top of the stack is output to the Polish string and the check is performed with the new top of stack. Eventually the new operator will have higher precedence than the top of the stack (an empty stack has zero precedence). If the new token is the end of the input string, then it is treated as an operator of lowest precedence. Some languages use a special character for the input string terminator, but this is not necessary. In any case, if the new token is the end of the input string, then POLISH is finished when the stack is empty. If the new token is not the end of the input string, then the token is placed on the stack and NEXTOKEN is

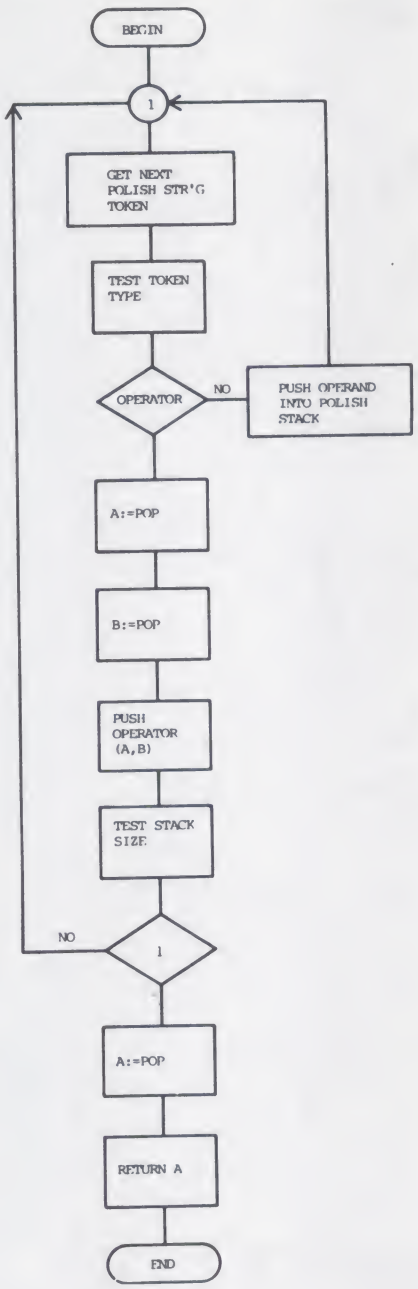


Figure 3: The EVAL routine specified as a flow chart. This routine is an example of an interpreter. It takes the Polish string created by POLISH, and decodes it and evaluates the mathematical value to be computed. Several functions are employed by the EVAL routine, as follows: PUSH means place the value in question into the operand stack, increasing the stack size by one value. POP means recover the top operand from the operand stack, decreasing the stack size by one value. OPERATOR(A,B) means evaluate the combination of the value A and the value B according to the definition of the current operator in the POLISH string. The data concepts employed during evaluation are as follows: Temporary data storage is found in A and B. The Polish string is a series of separated tokens created by POLISH as its output. The operand stack is a first-in-first-out stack of values defined by operand tokens (variables and constants) in the Polish string, or by the results of previous operations which are left in the stack.

Input	Stack	Output
A		A
+	+	
B	+	B
↑	+↑	
(	+↑(	
C	+↑(	C
*	+↑(*	
D	+↑(*	D
/	+↑(/	*
(	+↑(/(	
D	+↑(/(	D
+	+↑(/(+	
F	+↑(/(+	F
)	+↑(/	+
*	+↑(*	/
G	+↑(*	G
)	+↑	*
end of string		↑
		+

Table 2: An example of the POLISH routine in operation. The output string of a practical implementation of POLISH must have a convention to separate tokens. Storing output as a reconstructed character string with blanks to separate operands allows EVAL and UNPOL to use the same NEXTOKEN routine which POLISH calls. Other storage techniques, which do not require the use of blanks as separators, are possible.

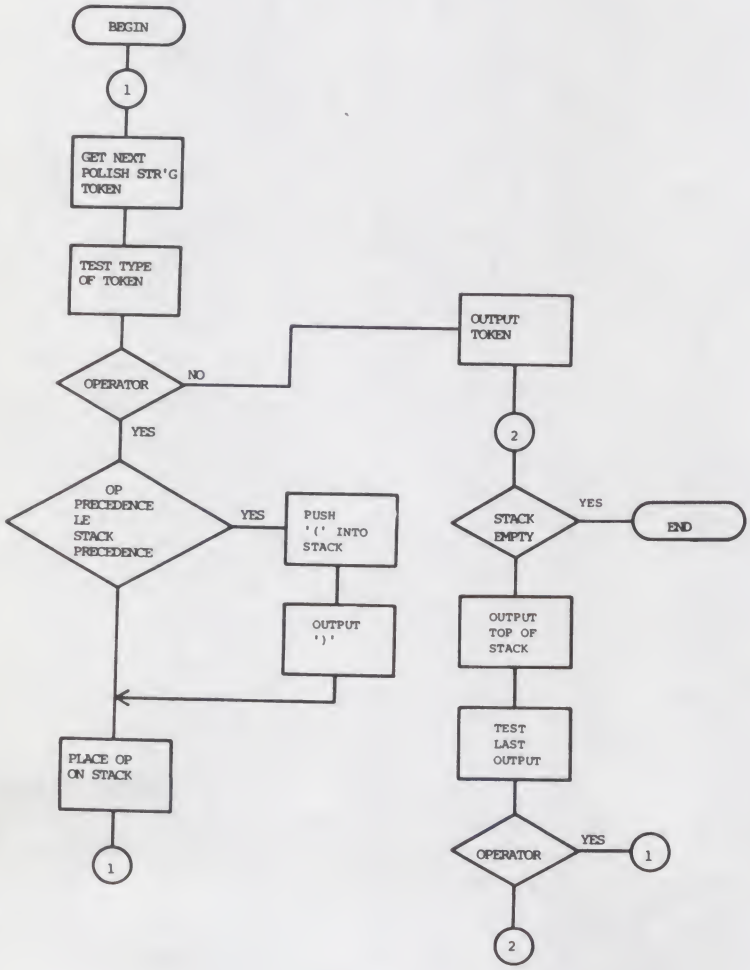


Figure 4: The UNPOL routine specified as a flow chart. This routine takes the Polish string created by POLISH and inverts the transformation to obtain a normal arithmetic expression again.

called again. Table 2 shows the input string, stack contents, and output string as POLISH works through the string A+B↑(C*D/(D+F)*G) where the upward arrow symbol represents the exponentiation. Exponentiation has a higher precedence than the other operations in this example. Working through the example shown in table 2 should convince the beginning programmer that this algorithm actually does translate to Polish notation. EVAL can then evaluate the Polish expression to obtain the final result.

Undoing What's Just Been Done

Polish notation is a convenient way to store a mathematical expression in computer memory. It tends to contain fewer characters, since parentheses are not needed. Also, it can be readily evaluated without the need to first perform a complicated conversion of the sort we just saw described. However, if one wants to edit an expression or change its structure, then one would really like to see the original form of the expression. Figure 4 shows the flowchart of a procedure called UNPOL which reverses the process of POLISH and converts a statement of Polish notation back to normal form. It scans from right to left (the reverse of POLISH) and outputs the normal string in reverse order. UNPOL can use the same NEXTOKEN and PRECEDE that POLISH uses (see the note in table 2). The only change is the sequence in which the tokens are used. Table 3 shows the input, stack contents, and output of UNPOL as it reverses the processing of the example A+B↑(C*D/(D+F)*G). Note that unnecessary parentheses are dropped when a mathematical expression goes through POLISH and then through UNPOL. For

Input	Stack	Output
+	+	
↑	+↑	
*	+↑(*	)
G	+↑(	G*
/	+↑(/	
+	+↑(/(+	)
F	+↑(/(	F+
D	+↑(	D(/
*	+↑(*	
D	+↑(	D*
C	+	C(↑
B		B+
A		A

NOTE: Processing starts at the top of this table. The Polish string is scanned in reverse order starting with its rightmost character. Proceeding down the table, the output is generated in reverse order also, starting with the rightmost character.

Table 3: An example of UNPOL in operation. The Polish string input to UNPOL is scanned in reverse order (right to left) and generates the output string starting at the left.

example, (A*B)+(C*D) becomes A*B+C*D. The parentheses were unnecessary because operator precedence ensured that the multiplications would be done first. UNPOL will not drop any necessary parentheses.

Trying a few examples through the parsing algorithms presented here should convince even a beginning programmer that Polish notation provides a straightforward way to make a computer evaluate complex mathematical expressions. Using these algorithms, it will be possible for readers to incorporate evaluation of mathematical statements into their programming systems.■

My Dear Aunt Sally's Glossary

Alphabetic Character: Any of the letters A through Z.

Assembler: A program which translates symbolic assembly language intput into machine language output. Assemblers frequently require arithmetic statement parsers in order to compute addresses and data values based upon symbolic assembly language statements.

Compiler: A program which translates symbolic statements of a high level language input into a machine language output. Compilers require some form of arithmetic statement parsing, although the output is generally converted one step further into actual machine code.

Constant: A constant is a way of specifying data which is fixed. In the My Dear Aunt Sally parser, constants are defined by input character strings which begin with a numeric character, and contain only numeric characters or at most one decimal point.

Interpreter: A program which translates symbolic statements of a high level language input into an immediate action. An interpreter could use the My Dear Aunt Sally parser to evaluate arithmetic expressions when required.

Mathematical expression: An input character string which obeys the syntactical rules of the My Dear Aunt Sally parser and can potentially be evaluated as a single resulting arithmetic value.

Numeric character: Any of the numbers 0 through 9.

Operator: An operator is a token specifying an action to be taken when the expression being parsed is evaluated. My Dear Aunt Sally recognizes two kinds of operators: Single character operators are used to denote the conventional arithmetic operations; multiple character operators are recognized by a dollar sign (as in $SIN) and are used for mathematical functions.

Parenthesis: Left and right parentheses are used to group operations in mathematical expressions. The only requirement for consistent evaluation of expressions is that left and right parentheses must balance.

Parser: A computer program mechanism for performing the parsing function.

Parsing: Given a set of syntax rules (a grammar) and an input string, parsing is the process of breaking that input string into a series of tokens according to the syntax rules.

Pop: Remove an element from a stack storage mechanism, in a last in, first out order.

Precedence: In evaluating an arithmetic expression, precedence is used to resolve ambiguities in the order of execution of several operations: The operations with higher precedence are performed first.

Push: Add an element to a stack storage mechanism.

Scan a string: The process of sequentially looking at each character or token of a string in a well defined order from left to right, or right to left.

Software pushdown stack: A stack storage mechanism can be implemented exclusively in hardware, or by using mechanisms which are part hardware and part software, or entirely in software. For the purposes of the My Dear Aunt Sally algorithm, all stacks are implemented in software. This means that each stack reserves a random access memory region and is supported by subroutines to perform the push and pop functions. The POLISH routine uses an *operator stack* to temporarily store and reorder the operator tokens when creating a Polish string; the EVAL routine uses an *operand stack* to temporarily hold values as it interprets the Polish string.

Statement: A statement is the programming language equivalent of a sentence in a natural language such as English.

String: A string is a series of values with definite starting and ending points. The parser of this article requires an input *character string* containing the human readable codes of an arithmetic expression, and produces a *Polish string* output of parsed tokens which can be evaluated by the Polish string interpreter.

Subroutine: A subroutine is a section of a program which is *called* to perform its function. When completed, it *returns control* to the routine which calls it. Subroutines are created for two purposes when programming: To modularize a program according to function, and to share common functions and save memory space.

Symbol table: A central collection of the variable names used in a program, along with related information. For the My Dear Aunt Sally parser, a symbol table would be composed of the variable token (a character string) and current numeric value for each variable found while parsing a statement. Note that the My Dear Aunt Sally algorithm by itself does not provide a means for setting the value of variables; an extension of the software into a full interpretive high level language with an assignment statement would provide such a means.

Token: A token is a basic unit of the syntax of an expression. In the My Dear Aunt Sally parser, tokens are character strings collected and returned by NEXTOKEN along with an indication of syntactical type.

Variable: A variable is a symbolically named data location. The parser of this article detects variables as character string names which begin with an alphabetic character.

Can YOUR Computer Tell Time?

by
James Hogenson
Box 295
Halstad MN 56548

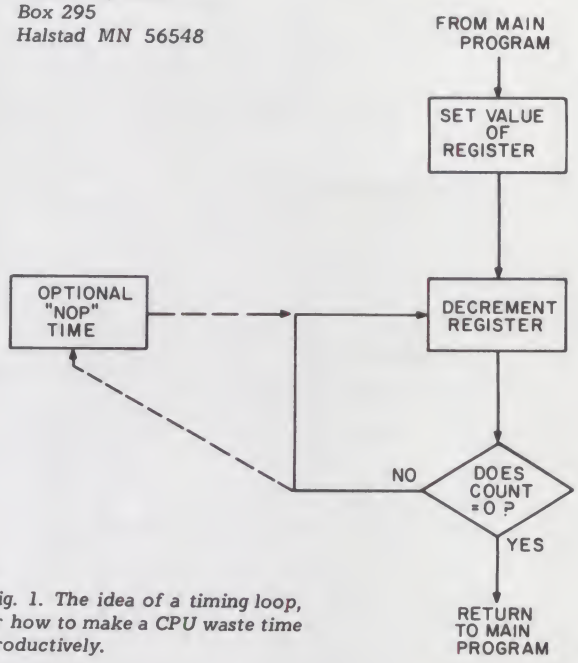


Fig. 1. The idea of a timing loop, or how to make a CPU waste time productively.

Loops are the basic time delay elements. Then there are loops within loops, loops within loops within loops and so on ad infinitum.

Can your computer tell time? O.K. Now take away the LSI clock chip, pocket watch, grandfather clock, or whatever else you managed to interface together. Can your computer still tell time? You bet it can!

It is a readily accepted fact that almost any type of hardware logic device can be imitated or simulated by computer software. That can also include timing devices if you wish.

We will examine a few methods and considerations for software timing, then apply what we've learned in making a novel "software only" clock which will keep time as well as any conventional clock.

The most efficient method (efficient referring to memory space used) to produce a time delay is the use of a loop. This loop is basically very simple, as shown by Fig. 1. By including NOPs or other non-functional time wasters in the loop, the loop can be significantly stretched.

An 8008 is being used in the examples in this article, but the principles hold for any computer. Only the numerical values will change.

The loop represented by Fig. 1 for an 8008 would be a simple three instructions (six bytes) long.

```
LBI "X"   ]  LOAD DELAY
DCB       ]  DECREMENT "X"
JFZ
L         ]  JUMP BACK
H         ]  UNLESS X = 0
```

The value of "x" loaded into the B register will be the main factor in varying the time delay provided by this loop. Calculating the exact time period is done by tabulation of instruction execution times. These examples will be based on the 8008 instruction execution times with the clock running at exactly 500 kHz.

To calculate the time for this loop, assume the value of "x" to be 1 so no part of the loop is repeated. Add up the number of microseconds required by each instruction.

```
LBI  =  32 US
DCB  =  20 US
JFZ  =  36 US
        88 US
```

Now go back to determine how many microseconds each repetition of the loop will produce. The LBI instruction is not repeated. Do not count any unrepeated instructions in this second tabulation.

```
DCB  =  20 US
JFZ  =  44 US
        64 US
```

Note the different execution times for the JFZ instruction. For the 8008, the execution time of conditional instructions depends upon the condition. If the condition results in a true branch, the instruction takes the longer of the two execution times. The false branch is the shorter time.

The time formula for this loop is

$$64X + 24 = N$$

"x" being the value loaded into B and "n" being the total execution time in microseconds. The unreduced formula is

$$(X - 1)64 + 88 = N$$

Since 64 us are added for each repetition, we must multiply 64 by one less than the value of "x."

255 is the largest possible value of "x" since we are limited to an 8-bit word. Therefore, the maximum time delay that can be provided by this loop is 16344 us. This loop can be stretched by placing a NOP instruction (op code 300) before the DCB, and re-routing the jump.

```
LBI "X"   SET VALUE OF "X"
NOP       ABSORB EXTRA 20 US
DCB       DECREMENT "X"
JFZ       JUMP BACK TO NOP
L         UNLESS X = 0
H
```

If desired, more than one NOP may be inserted. Each NOP will add another 20x microsedonds. The maximum time with one NOP is 21444 us, the NOP adding 5100 us.

If a timing loop is to be used a number of times at

various points in a program, it may be desirable to rewrite the loop as a called subroutine. The basic flowchart remains unchanged; only the method of implementing it changes.

```
    (MAIN
    PROGRAM)
    CAL   CALL
    L     TIME
    H     LOOP

(TIME
 LOOP)
   LBI      SET VALUE OF "X"
   "X"
   DCB      DECREMENT "X"
   RTZ      RETURN IF "X" = 0
   JMP
   L        JUMP BACK TO DCB
   H
```

Tabulation will show that the basic loop is good for 116 us with each repetition adding 76 us. The reduced formula is

$$76X + 40 = N$$

This loop is a little more complex. Although the CAL instruction which calls the loop is not a part of the loop itself, the execution time of the CAL instruction is a part of the time period produced. We, therefore, must add 44 us for the CAL.

As done before, we assume the value of "x" to be 1 for the first tabulation. The RTZ will be a true branch, so we stop adding there. An RTZ true branch will take 20 us, while an RTZ false branch will take 12 us.

Each repetition will add 12 us for the RTZ, 44 us for the JMP, and 20 us for the DCB instruction. The unreduced formula is

$$(X - 1)76 + 116 = N$$

NOPs placed before the DCB instruction will have the same effect as in the first loop, an additional 20x us per NOP.

The maximum time period produced by this second loop with one NOP is 24520 us. The minimum time period without any NOPs is 116 us. Anything under 116 us can be more efficiently implemented with straight NOPs than with a loop, should such a need arise.

If a time period much longer than 24000 us is needed, modify the time loop to make a double loop as shown in Fig. 2. Make an identical loop, but rather than using a NOP for more time, insert an entire loop.

```
    CAL   (TIME LOOP)
    L
    H

(TIME LOOP)
     LCI     SET VALUE OF "Y"
     "Y"
A    NOP
     LBI     SET VALUE OF "X"
     "X"
B    NOP
     DCB     DECREMENT "X"
     JFZ
     L       JUMP BACK TO DCB
     H       IF "X" ≠ 0
     DCC     DECREMENT "Y"
     RTZ     RETURN IF "Y" = 0
     JMP
     L       JUMP TO LBI
     H
```

Time calculations for multiple loops become somewhat more complex, but again the same principle is used.

The inside loop used here is the same loop first calculated at the beginning of this article. When calculating the main loop, the inside loop is treated as one combined unit of value. The tabulation will look like this:

```
MAIN LOOP:
   CAL   =   44 US
   LCI   =   32 US
   INSIDE
   LOOP  = (64X + 24) US
   DCC   =   20 US
   RTZ   =   20 US
           (64X + 140) US

EACH REPETITION OR
TRUE BRANCH WILL ADD:

RTZ   =   12 US
JMP   =   44 US
INSIDE
LOOP  = (64X + 24) US
DCC   =   20 US
        (64X + 100) US
```

The formula, unreduced, would be

$$64X + 140 + (Y-1)(64X + 100) = N$$

Reducing the formula gives us

$$64XY + 100X + 40 = N$$

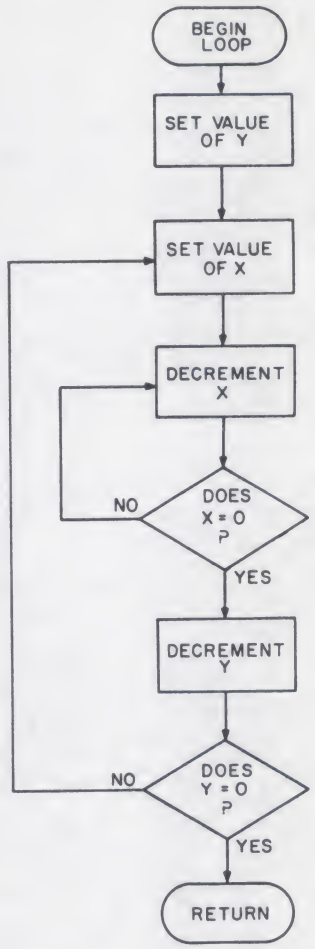


Fig. 2. Getting fancy. By nesting one timing loop within an outer loop, much longer delays can be obtained. Two parameters "x" and "y" are required to completely specify this loop. In a 16-bit machine, of course, the same result (here intended for an 8-bit 8008) can be obtained without nested loops since the 16-bitter can count much higher.

The maximum time delay provided by this loop would be 4187140 us. A NOP inserted at location "a" will add 20y us. A NOP at "b" will add 20xy us. The use of both NOPs will boost our maximum time to 5492740 us, or over 5 seconds.

The purpose of developing formulas is to determine the values of the registers needed to obtain a specified time period. For purposes of illustrating an example, let us assume we want exactly 5000 us to pass between point A and point B of a program. We would place a CAL instruction between point A and point B which would call the time loop. The shorter loop will be sufficient for this application, so the equation will now be

$$76X + 40 = 5000$$

Working the equation will give a value of 65.23615 . . . for "x." A fractional value will not fit in any single register of the CPU. To find out what to do now, multiply 65 by 76, add 40, and subtract the result from 5000. We find the difference

is 20 us. This is very simple to take care of. Insert a NOP instruction at any point in the routine where it will not be repeated. Before the LBI instruction would do fine. Now, with 65 (decimal notation) loaded into the B register, exactly 5000 us will pass between points A and B of our main program.

Finding an exact time period using the longer loop involves a certain amount of trial and error. To find an approximate value of "x" (using no NOPs) use this formula:

$$x = \left(\frac{N-40}{64Y}\right) - 1.5$$

Assign an arbitrary value to "y," replace "n" with the required time period.

Now, assume a time period of exactly 505904 us is needed. (This time period will be used later.) There is one stipulation in this case which will be explained in greater detail later. The value of "x" must be 255. Solving the formula equation for "y"

$$(64)(255)Y + 100Y + 40 = 505904$$

gives "y" a value of 30.8078. 30 must be used for "y." The total time of the loop is then 492640 us, 13264 us short of the required time. In most cases, you would re-assign values and try again, but in this case, the value of "x" cannot be changed. The alternative is to use the shorter loop to clean up the leftovers. After calling one loop, call the other loop. Then go on with the main program. Solving the short loop equation comes out at a nice even 174.

$$76X + 40 = 13264$$

What looked like a real oddball turned out to be perfect!

The formulas and all such may seem like a lot of monkey business just to waste time. Speed *is* the purpose of computers, but there are times when they must be slowed down.

The primary application of time loops is in I/O interface. If a computer is to monitor a data input which is to be read once every 10 ms, there are two alternatives for timing. The hardware of the device being monitored may include a timing device and a flag to indicate when the device is ready. The computer enters a loop which monitors the flag until the device is ready, then reads the data. The other alternative is to use the software time loop, and omit the extra hardware.

An interesting application along this line is a completely software "fabricated" keyboard debounce system. This method will not work in an interrupt type of input system, but for many small scale systems, this method is ideal.

Rather than connecting the *keypressed* line of the keyboard to some debounce, timer and latch circuitry, connect it to the eighth bit of the parallel data input on the computer. The loop used will test the eighth bit for the *keypressed* state. When a *keypressed* is sensed, a time loop of 16344 us is executed, then the data input is accepted. The loop then branches back to the main program to take care of the new data. When the program comes back to the input loop, the *keypressed* line is first tested to be sure no keys are being pressed. After all keys have been released, the loop will wait for the next *keypressed* state. This procedure will prevent more than one data entry from each keystroke.

When I first tried this keyboard debounce method over five months ago, I was so pleased with it that I'm still using the method for all data entry to my microcomputer. Not once has it missed some data, or given me false or duplicated data. And it was so easy to implement!

Time loops may also be used in output applications. I have an SWTPC TV typewriter, but I am not using the special computer interface board. I found that a simple time loop does the job well enough and fast enough.

Since we've done our homework, now we can play. An interesting and novel application of time loops is a completely software "fabricated" clock. The clock program presented here will have three major functions (see Fig. 4).

The clock will display hours, minutes and seconds. The "increment time count" segment of the program is responsible for computing the next time reading in sequence. It must consist of more than a straight counting sequence since time is not expressed in straight decimal format.

The "update display" segment is responsible for producing the newly computed time at an output device.

After the first two segments have been executed,

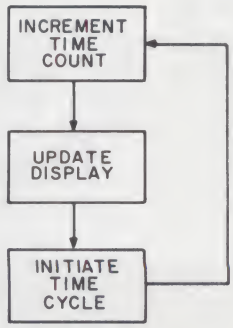
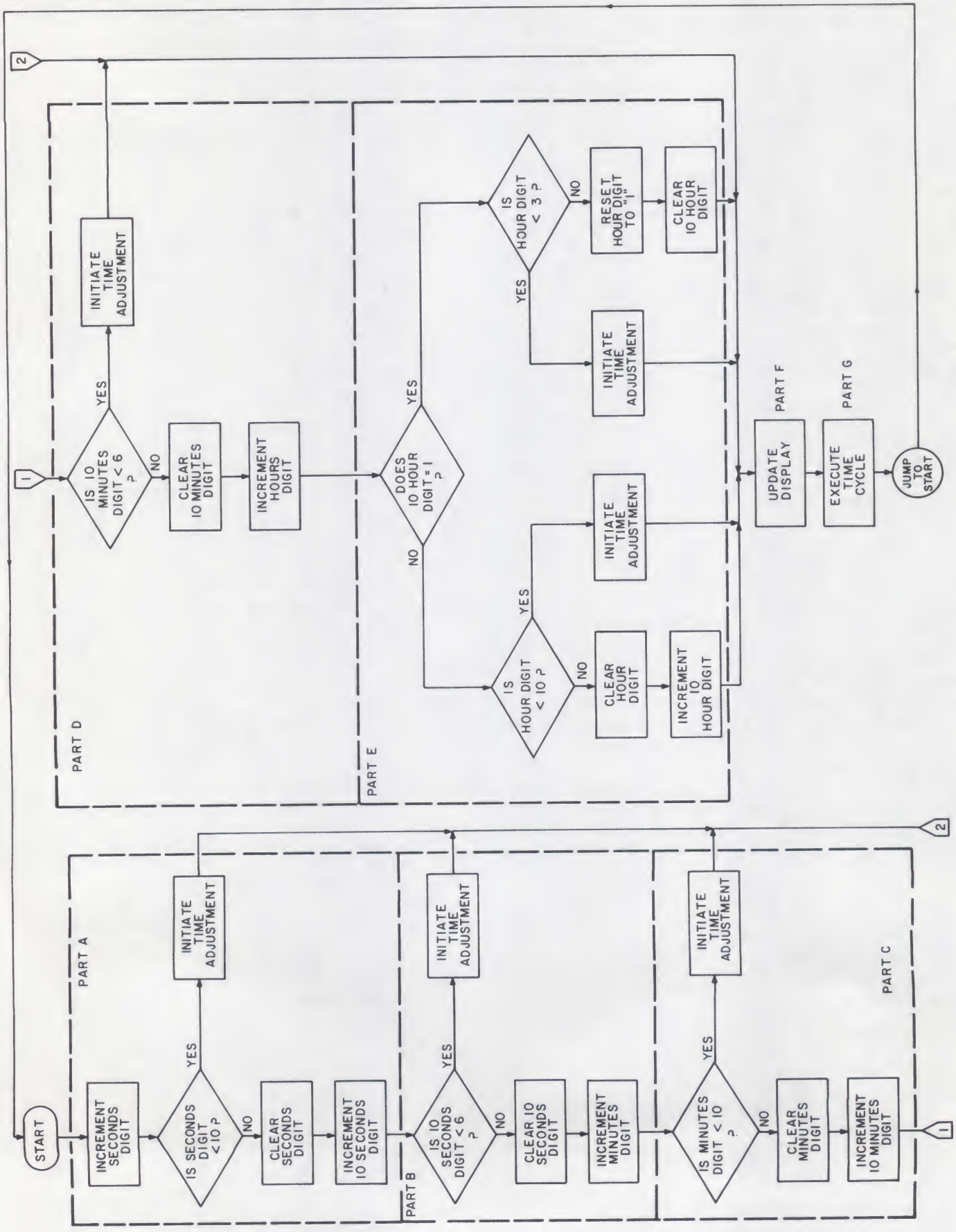


Fig. 4. The digital clock program looks simple at this level: Increment the time count, update the display, then initiate a time cycle such that the entire loop takes exactly one second!

Fig. 3. Software can be used to debounce a keyboard — simply loop around for a long enough time to ensure that keys have stabilized. The loop is started as soon as the "any key pressed" (keypressed) line indicates any non-null bit pattern.

```
07/115 = 101   IN
       116 = 002   RLC    TEST INPUT FOR KEYPRESSED,
       117 = 100   JFC    WAIT UNTIL CONDITION IS
       120 = 115   L      SATISFIED
       121 = 007   H
       122 = 101   IN
       123 = 002   RLC    TEST INPUT FOR KEYPRESSED,
       124 = 140   JTC    WAIT UNTIL CONDITION IS
       125 = 122   L      SATISFIED
       126 = 007   H
       127 = 026   LCI
       130 = 377   "255"  EXECUTE
       131 = 021   DCC    TIME
       132 = 110   JFZ    DELAY
       133 = 131   L
       134 = 007   H
       135 = 101   IN     ACCEPT INPUT OF DATA
       136 = 104   JMP
       137 = 1     L      JUMP TO MAIN PROGRAM
       140 = H     H      (MAY BE REPLACED WITH A
                           RETURN INSTRUCTION.)
```

Fig. 5. Ah, but the simplicity of Fig. 4 – as this figure reveals – hides a lot of low level detail. Here is the flow chart of the clock's operations.



8008 Timing Quick Reference Guide

US.	INSTRUCTION
20	INCREMENT INDEX REGISTER
20	DECREMENT INDEX REGISTER
20	ROTATE ACCUMULATOR
12/20	CONDITIONAL RETURN*
36/44	CONDITIONAL JUMP*
36/44	CONDITIONAL CALL*
20	UNCONDITIONAL RETURN
44	UNCONDITIONAL JUMP
44	UNCONDITIONAL CALL
20	RESTART
32	LOAD DATA IMMEDIATE (INTO INDEX REGISTER)
36	LOAD DATA IMMEDIATE (INTO MEMORY REGISTER)
32	ALU IMMEDIATE
20	ALU REGISTER
32	ALU MEMORY REGISTER
24	OUTPUT
32	INPUT
20	LOAD DATA - REGISTER(OP CODE 3--)
32	LOAD DATA - MEM. & REG. (OP CODE 3-7 OR 37-)

Here is a quick reference table for execution times of all instructions in the 8008 repertoire. Such a reference table can be easily made for any CPU. Simply multiply the number of machine states required for the execution of each type of instruction by the time required per machine state. AT 500 kHz, the 8008 takes four us per machine state. An unconditional jump instruction requires 11 states in the 8008 , therefore 44 us. Do not confuse machine states with machine cycles. The same jump instruction requires three machine cycles.

*Conditional instructions: Execution time depends upon condition. If condition causes true branch, the execution time is longer. If the condition causes a false branch (if condition is not satisfied), the execution time is shorter.

the "time cycle" segment makes up the difference so that the entire program takes exactly one second per pass. Writing a clock program isn't hard, but making it take exactly one second per pass definitely adds to the challenge. The major consideration is that branches from conditional instructions must be balanced in such a way that the program will take exactly the same execution time regardless of the combination of conditions and branches. That's where all the time loops come in, and that's where lots of fun comes in!

The program can best be described in the form of a flowchart, Fig. 5. The program listing in Fig. 6 is divided according to the flowchart divisions shown by dashed lines. The op codes are for 8008 systems. The mnemonics and op codes can be easily translated into 8080 format. However, all timing considerations must be recalculated for use with anything other than an 8008 running at exactly 500 kHz.

When time balancing a segment of a program, it is best to work from the bottom and go up. The time adjustment in part A of the flowchart must compensate for parts B, C, D and E, so before that time period can be calculated, the execution time of the other parts must be calculated.

Some of the time adjustments in part E do not use a time loop. The short time adjustments there (in part E) are more conveniently implemented with a combination of other time consuming instructions which will not change the function of the program.

To determine the time adjustment needed in one branch, tabulate the total execution time of the longer branch. Add or subtract 8 us (depending upon which branch is the true branch) to compensate for the difference in conditional jump instructions.

The same time loop will be used several times, yet the time periods will vary. This can be accommodated when using the short loop by placing the LBI instruction and loading the value of "x" *before* the loop is called. The location of the LBI instruction will have no effect on the overall time period produced.

Occasionally a time loop will not come out evenly. For example, another 12 us may be needed. This will not be accommodated in the loop, so the only alternative is to use a NOP instruction. But the only instruction which will absorb just 12 us is an unsatisfied conditional return instruction. Using such an instruction could result in trouble if used alone. However, if an AND instruction can be used without affecting the program functions, the AND instruction will insure that the conditional return (RTC) will not be satisfied. To keep the program in balance when balancing the time, insert a NOP in the opposite branch to offset the AND instruction, and the net difference will be 12 us.

Flowchart parts F and G need not be included in the time balancing considerations of A, B, C, D and E. The program returns to a common point before executing parts F and G, so those parts are not offsetting anything.

The output loop as given in the listing will provide an ASCII output for a TV typewriter. A sufficient time loop is provided between each individual output operation. The output loop may be easily modified for use with other devices. For use with Teletype, a line feed command must be added to the output characters. (Only a carriage return is used with a TVT.) For use with an LED display, deleting the ORI instruction at location 04/257 will leave a straight binary (also BCD equivalent, since vales do not exceed 9) output. Keep in mind, however, that modifying any part of the program will also require modifying the timing elements involved.

The execution time of the complete "increment time count" segment plus the "update display" segment totals 494096 us. Subtract that time from one second to find the time required of the timing cycle. The required time is 505904 us. The values for this loop have already been worked out in a previous example.

The reason the value of "x" cannot be conveniently changed in the long loop in this case is that the loop is called and used from two locations in the program. The value of "x" cannot be changed for one application without affecting the other. If the loop were modified to load B from another register which remained constant, both values would become variables which could be easily assigned values from any point in the program. This would also include recalculating the time formula of the loop.

Your clock should now be ready to run. (Oh, by the way, there is one little drawback: Your computer can't be used for anything else while it's keeping time, unless, of course, you really want to go to extremes with the calculating! This program is strictly a novelty!) When you are ready to start your clock, load the correct time plus a couple of minutes into memory locations 04/000 through 04/005. When the loaded time comes, start the computer. Jump into the program at 04/006.

The time kept by the computer will only be as accurate as the frequency of the clock driving the CPU. The oscillator must be set at exactly 500 kHz. Although this is difficult to do, any percentage of error in frequency will be directly reflected by the time kept. ■

Fig. 6. And finally, the lowest level of detail of all: A "pseudo assembly" listing of the program for the digital clock as implemented for an 8008 computer. Of course, those readers who have an 8080, a 6501, a 6800 or PACE will have to do a little bit of thinking to adapt Fig. 4 and Fig. 5 to the alternative microcomputer CPU designs.

```
THOUR:004/000 = XXX          10 HOUR DIGIT REGISTER
 HOUR:004/001 = XXX          HOUR DIGIT REGISTER
 TMIN:004/002 = XXX          10 MINUTE REGISTER
  MIN:004/003 = XXX          MINUTE REGISTER
 TSEC:004/004 = XXX          10 SECOND REGISTER
  SEC:004/005 = XXX          SECOND REGISTER
- - - - - - -
START:004/006 = 056  LMI     LOAD L/H WITH
      004/007 = 004  H(SEC)  ADDRESS OF SECONDS
(A)   004/010 = 066  LLI     DIGIT REGISTER
      004/011 = 005  L(SEC)
      004/012 = 307  LAM     INCREMENT SECONDS
      004/013 = 004  ADI     DIGIT
      004/014 = 001  "1"
      004/015 = 074  CPI     DECISION: JUMP IF
      004/016 = 012  "10"    SECONDS DIGIT IS NOT
      004/017 = 100  JFC     LESS THAN 10
      004/020 = 036  L ]---->GTENS
      004/021 = 004  H ]
      004/022 = 370  LMA     RETURN SECONDS DIGIT TO ITS REGISTER
      004/023 = 370  LMA     REPEAT INSTRUCTION FOR MORE TIME
      004/024 = 016  LBI
      004/025 = 017  "15"    SET VALUE OF "X"
      004/026 = 106  CAL     CALL TIME LOOP TO COMPENSATE
      004/027 = 247  L ]
      004/030 = 004  H ]---->TLOOP
      004/031 = 300  NOP     NEED A LITTLE MORE TIME
      004/032 = 300  NOP
      004/033 = 104  JMP     FINISHED THIS CYCLE
      004/034 = 275  L ]
      004/035 = 004  H ]---->DISPL
GTENS:004/036 = 006  LAI
      004/037 = 000  "0"     CLEAR SECONDS DIGIT REGISTER
      004/040 = 370  LMA
      004/041 = 061  DCL
      004/042 = 307  LAM     INCREMENT
      004/043 = 004  ADI     10 SECONDS DIGIT
      004/044 = 001  "1"
- - - - - -
(B)   004/045 = 074  CPI     DECISION: JUMP IF
      004/046 = 006  "6"     10 SECONDS DIGIT IS NOT
      004/047 = 100  JFC     LESS THAN 6
      004/050 = 063  L ]---->GOMIN
      004/051 = 004  H ]
      004/052 = 370  LMA     RETURN 10 SEC. DIGIT TO REGISTER
      004/053 = 016  LBI
      004/054 = 015  "13"    SET VALUE OF "X"
      004/055 = 106  CAL     CALL TIME LOOP
      004/056 = 247  L ]
      004/057 = 004  H ]---->TLOOP
      004/060 = 104  JMP     FINISHED THIS CYCLE
      004/061 = 275  L ]
      004/062 = 004  H ]---->DISPL
GOMIN:004/063 = 076  LMI     CLEAR 10 SEC. DIGIT REGISTER
      004/064 = 000  "0"     (LAI, LMA ARE USED INSTEAD OF LMI
      004/065 = 061  DCL     WHERE TIMING WORKS OUT BETTER THAT WAY.)
      004/066 = 307  LAM     INCREMENT
      004/067 = 004  ADI     MINUTES DIGIT
      004/070 = 001  "1"
- - - - - -
      004/071 = 074  CPI     DECISION: JUMP IF
(C)   004/072 = 012  "10"    MINUTES DIGIT IS NOT
      004/073 = 100  JFC     LESS THAN 10
      004/074 = 110  L ]---->GTENM
      004/075 = 004  H ]
      004/076 = 370  LMA     RETURN MINUTES DIGIT TO REGISTER
      004/077 = 016  LBI
      004/100 = 012  "10"    SET VALUE OF "X"
      004/101 = 106  CAL     CALL TIME LOOP
      004/102 = 247  L ]
      004/103 = 004  H ]---->TLOOP
      004/104 = 317  LBM     NEED AN EXTRA 12 US.
      004/105 = 104  JMP     (LBM - 32 US, OTHER 20 US  BALANCED BY NOP)
      004/106 = 275  L ]
      004/107 = 004  H ]---->DISPL
GTENM:004/110 = 300  NOP     20 US BALANCE
      004/111 = 006  LAI
      004/112 = 000  "0"     CLEAR MINUTES REGISTER
      004/113 = 370  LMA
      004/114 = 061  DCL
      004/115 = 307  LAM     INCREMENT
      004/116 = 004  ADI     10 MINUTES DIGIT
      004/117 = 001  "1"
- - - - - - -
      004/120 = 074  CPI     DECISION: JUMP IF
(D)   004/121 = 006  "6"     10 MINUTES DIGIT IS NOT
      004/122 = 100  JFC     LESS THAN 6
      004/123 = 137  L ]---->GOHOUR
      004/124 = 004  H ]
      004/125 = 370  LMA     RETURN 10 MIN. DIGIT TO REGISTER
      004/126 = 016  LBI
      004/127 = 007  "7"     SET VALUE OF "X"
      004/130 = 106  CAL     CALL TIME LOOP
      004/131 = 247  L ]
      004/132 = 004  H ]---->TLOOP
      004/133 = 300  NOP     KEEPING THE TIME IN BALANCE
      004/134 = 104  JMP     FINISHED THIS CYCLE
      004/135 = 275  L ]
      004/136 = 004  H ]---->DISPL
GOHOUR:004/137 = 006  LAI
      004/140 = 000  "0"     CLEAR 10 MINUTES REGISTER
      004/141 = 370  LMA
      004/142 = 061  DCL
      004/143 = 307  LAM     INCREMENT
      004/144 = 004  ADI     HOURS DIGIT
      004/145 = 001  "1"
      004/146 = 370  LMA     PUT THE HOURS DIGIT BACK FOR THE
- - -   004/147 = 061  DCL     TIME BEING.
(E)   004/150 = 307  LAM
      004/151 = 074  CPI     DECISION: JUMP IF
      004/152 = 001  "1"     10 HOUR DIGIT = 1
      004/153 = 150  JTZ
      004/154 = 213  L ]---->NOON
      004/155 = 004  H ]
      004/156 = 060  INL
      004/157 = 307  LAM
      004/160 = 074  CPI     DECISION: JUMP IF
      004/161 = 012  "10"    HOUR DIGIT IS NOT
      004/162 = 100  JFC     LESS THAN 10
      004/163 = 177  L ]---->GTENH
      004/164 = 004  H ]
      004/165 = 307  LAM     NEED      ***A NOP IS ADDED TO BOTH BRANCHES
      004/166 = 307  LAM     TO        TO BALANCE  THE TRUE  BRANCH FROM
      004/167 = 307  LAM     WASTE     04/153. A BETTER PLACE FOR THE NOP
      004/170 = 307  LAM     200 US    WOULD HAVE BEEN  04/156,  BUT WHO
      004/171 = 307  LAM     TO        WANTS TO REWRITE HALF A PROGRAM TO
      004/172 = 300  NOP     BALANCE   SAVE ONE MEMORY LOCATION?
      004/173 = 300  NOP     BRANCH
      004/174 = 104  JMP     NOW LET'S GET OUT'A HERE
      004/175 = 275  L ]
      004/176 = 004  H ]---->DISPL
GTENH:004/177 = 006  LAI
      004/200 = 000  "0"     CLEAR HOUR DIGIT REGISTER
      004/201 = 370  LMA
      004/202 = 061  DCL
      004/203 = 307  LAM     INCREMENT
      004/204 = 004  ADI     10 HOUR DIGIT
      004/205 = 001  "1"
      004/206 = 370  LMA     RETURN 10 HR. DIGIT TO REGISTER
      004/207 = 300  NOP     ***
      004/210 = 104  JMP     CYCLE FINISHED
      004/211 = 275  L ]
      004/212 = 004  H ]---->DISPL
NOON: 004/213 = 060  INL
      004/214 = 307  LAM
      004/215 = 074  CPI     DECISION: JUMP IF
      004/216 = 003  "3"     HOUR DIGIT IS NOT
      004/217 = 100  JFC     LESS THAN 3
      004/220 = 232  L ]
      004/221 = 004  H ]---->RESHOUR
      004/222 = 016  LBI
      004/223 = 002  "x"     SET VALUE OF "x"
      004/224 = 106  CAL     CALL TIME LOOP
      004/225 = 247  L ]
      004/226 = 004  H ]---->TLOOP
      004/227 = 104  JMP     JUMP TO 004/275/036
      004/230 = 275  L ]
      004/231 = 004  H ]---->DISPL
RESHOUR:004/232 = 006  LAI
      004/233 = 001  "1"     RESET HOUR DIGIT TO "1"
      004/234 = 370  LMA
      004/235 = 061  DCL
      004/236 = 006  LAI
      004/237 = 000  "0"     CLEAR 10 HOUR DIGIT REGISTER
      004/240 = 370  LMA
      004/241 = 241  NDB     (YES, 241 = 241; THAT'S NOT AN ERROR)
      004/242 = 043  RTC     FOR TIME BALANCING. THE NET DIFFERENCE
      004/243 = 043  RTC     BETWEEN BRANCHES FROM 04/153 WAS 24 US.
      004/244 = 104  JMP     2 X RTC = 24 US. THE NDB IS BALANCED
      004/245 = 275  L ]                BY THE NOP'S MENTIONED
- - -   004/246 = 004  H ]---->DISPL      IN THE NOTE ***
TLOOP:004/247 = 011  DCB     SHORT TIMING LOOP
      004/250 = 053  RTZ
      004/251 = 104  JMP
      004/252 = 247  L ]
      004/253 = 004  H ]---->TLOOP

OUTL: 004/256 = 307  LAM     OUTPUT SUBROUTINE STARTS
      004/257 = 064  ORI     GENERATE ASCII CHARACTER
      004/260 = 060  "48"
      004/261 = 121  OUT     PRINT CHARACTER
      004/262 = 026  LCI
      004/263 = 005  "5"     SET VALUE OF "Y"
      004/264 = 106  CAL     CALL LONG TIME LOOP
      004/265 = 324  L ]
      004/266 = 004  H ]---->LTIME
      004/267 = 031  DCD
      004/270 = 053  RTZ     ARE WE DONE PRINTING?
      004/271 = 060  INL     CONTINUE IF NOT
      004/272 = 104  JMP
      004/273 = 256  L ]
      004/274 = 004  H ]---->OUTL
- - - - -
DISPL:004/275 = 036  LDI     SET UP COUNT - DISPLAY ROUTINE
      004/276 = 006  "6"
(F)   004/277 = 066  LLI     SET UP ADDRESS
      004/300 = 000  "0"
      004/301 = 106  CAL     OUTPUT ROUTINE
      004/302 = 256  L ]
      004/303 = 004  H ]---->OUTL
      004/304 = 006  LAI
      004/305 = 015  "13"    OUTPUT CARRIAGE RETURN COMMAND
      004/306 = 121  OUT
- - - - -
      004/307 = 026  LCI
(G)   004/310 = 036  "30"    SET VALUE OF "Y"
      004/311 = 106  CAL     CALL LONG TIME LOOP
      004/312 = 324  L ]
      004/313 = 004  H ]---->LTIME
      004/314 = 016  LBI
      004/315 = 255  "175"   SET VALUE OF "X"
      004/316 = 106  CAL     CALL SHORT TIME LOOP
      004/317 = 247  L ]
      004/320 = 004  H ]---->TLOOP
      004/321 = 104  JMP     JUMP BACK TO THE BEGINNING AND RECYCLE
      004/322 = 006  L ]
      004/323 = 004  H ]---->START
- - - - - -
LTIME:004/324 = 016  LBI     SET VALUE OF "X"
      004/325 = 377  "255"
LTIMI:004/326 = 011  DCB     DECREMENT "X"
      004/327 = 110  JFZ     JUMP BACK TO DECREMENT AGAIN
      004/330 = 326  L ]
      004/331 = 004  H ]---->LTIMI
      004/332 = 021  DCC     IF "X" DOESN'T EQUAL "0"
      004/333 = 053  RTZ     DECREMENT "Y"
      004/334 = 104  JMP     GO BACK TO PROGRAM IF Y = 0
      004/335 = 324  L ]     REPEAT LOOP
      004/336 = 004  H ]---->LTIME
      004/END = SET THE TIME A MINUTE OR TWO IN ADVANCE AT LOCATIONS 04/005,
                WAIT UNTIL THE RIGHT TIME, AND START THE PROGRAM BY JUMPING
                IN  AT 04/006.  THE PROGRAM  IS STOPPED BY  LOADING A  HALT
                INSTRUCTION INTERRUPT FROM THE FRONT PANEL.
```

A Plot Is Incomplete Without Characters

Richard J Lerseth
8245 Mediterranean Way
Sacramento CA 95826

Who would want to miss the opportunity of creating customized graphics for special applications?

As computer hobbyists, a number of us will sooner or later play around with graphics using vector CRTs or XY pen plotters; but very few of us will be willing to pay the high price of a number of copyrighted plotting packages available today through computer graphics houses. Besides, most of us will not want to miss the opportunity of creating our own packages.

So, in the process of interfacing your graphic media to your computer, you will normally have built the software needed to control simple vector moves on your media, as well as be able to window your plottings (that is, confine your moves within a specified area).

But, you will find that one of your major efforts will be building the character generation module. As you will soon realize, computer graphics take large chunks of memory space for the graphic routines and plotting tables describing plotting sequences. Particularly, you will find that character generation will take a large portion of that memory space.

The design of a plotting data format can be likened to designing a special purpose computer instruction set; this instruction set is emulated by the plotting software in real time.

In this article, I will describe some of the basic concepts of character generation, and describe techniques of saving memory space through efficient programming and by maximizing the packing of information in the plotting tables.

I assume at this point that you have within your basic plotting software: (1) the capability of shifting the relative origin within the plotting frame and (2) the capability of chain plotting. That is, plotting a vector from the ending point of the last vector move to the new position on the plotting field without explicitly defining the beginning point every time you make a vector move. We make full use of these two capabilities in plotting the character strings.

Plotting Frame

The easiest way of plotting a character is to define a plotting frame or grid upon which a sequence of vector moves are made from grid point to grid point. To minimize the complications involved with signed vectors, it is best to set the origin in the lower left hand corner of the field on which the character is to be plotted. With this convention, the vector moves are positive upward and to the right in the grid. In this way, we can define the ending point of a vector move with positive integer coordinates.

Limiting Frame and Plotting Resolution

Next, we have to define the resolution in plotting the characters. That is, we have to decide how many grid points we desire within a character frame. This depends on many factors: How fine you want your plotting; how many different characters you are to plot; how you are to pack the moves into memory; what special effects or options you desire. These considerations are all interrelated and must be considered as a whole.

I will propose an optimum character grid field within a limiting frame which will have a resolution as fine or finer than any used today by the graphic houses in their charac-

The choice of a character grid should reflect the realities of the common machine designs. For most microcomputers (and minicomputers), a character frame optimized for 8 or 16 bit words is desirable.

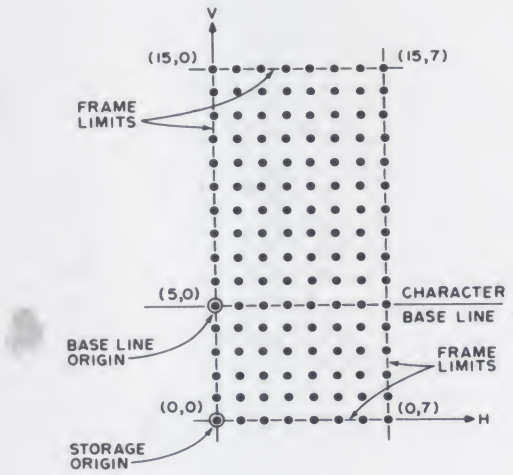


Figure 1: 8 by 16 Character Frame. Characters are plotted (or drawn on a vector graphics display) with reference to this local coordinate system. A series of 8 bit codes identifies the successive locations of the pen (or electron beam) and whether or not a line is to be drawn while moving to the location. The codes which reference the bottom row of this grid are treated as special operation codes for the plotting software: subchain reference, half shift right option, and floating subchain operation are defined in this article.

ter plotting packages. It will minimize the use of storage, and will also have some capability for special options. However, as a user of this software, you can make appropriate changes in your own system to reduce the resolution or to eliminate some of the special options.

Figure 1 shows the (8 x 16) grid I propose. The storage origin (0,0) is defined to be in the lower left hand corner of the grid. The character base origin (0,5) is at the lower third point of the left hand side of the limiting frame such that upper case alphabetic characters will be confined to the upper two-thirds of the grid frame. The lower third will be used for the tails of lower case alphabetic characters. The lower row of the grid will not be used for plotting; this row of 8 points will be reserved for flagging special options, which will be explained later.

Specifying Moves

Since most of us are using or will be using 8 or 16 bit machines, choosing this grid optimizes the packing of information for a vector move into an 8 bit byte of memory. A move to any point in this grid field (figure 1) could be defined with 3 bits for the horizontal (H) position, 4 bits for the vertical (V) position, and 1 bit for the Z function or the status (P) of the move (pen up or pen down for pen plotters, or intensity modulation in video graphics).

The 8 bits of H, V and P data for a move can be packed in the six different ways, such as HVP, VPH, PHV, VHP, HPV, or PVH. However, when packing such data into the byte, one must consider which is the fastest way to unpack the values. This greatly depends on the machine used. In most cases, it simply entails masking and shifting. I am going to use (VHP) as my standard. Why? No reason except that it can be implemented on most of the micros in use today without excessive effort. One procedure of unpacking the byte is given in appendix A.

To clarify further discussions on vector moves, the coordinates of the moves within the limiting frame will be written as (V,H). When the Z function is included, the move will be defined as (V,H,P) where

V is the vertical portion of the move
H is the horizontal portion of the move
P is the status of the Z function where,
0 is pen down or display tube electron gun on
1 is pen up or display tube electron gun off.

The lower portion of the grid (V = 0; H = 0 to 7; P = 0 or 1) will be reserved for special options which will be defined later.

Optimization of the Storage of Character Moves

Once a user starts playing around with developing the moves for each and every character, he soon realizes that there are a number of instances where a chain of moves is duplicated in the patterns of several characters. One can take advantage of this by building subchains and referencing them where it is appropriate to combine them in a large sequence. For example, the upper case

When implementing this software for a graphic display mechanism such as a CRT, pay attention to speed of execution. Flicker will result if your computer and software cannot keep up with your eye's timing characteristics.

alphabetic characters, (G, C, O, and Q) can all be combined together in one single chain. Also, the many lower case alphabetic characters have (c, ɔ, or o) as part of their chain. Taking advantage of such duplications can significantly lower the storage requirements of character plotting tables.

Special Options

With the character defined, the next task is to shift, twist, stretch or squeeze the characters as they are drawn.

When I defined the character limiting frame previously, I reserved the lowest horizontal line of grid points for special options. There are 8 grid points on this line. This gives 8 special options that can be used. If one considers the Z function, there are 16 options in all. Whenever (O,H,P) is encountered in a plotting chain of moves, then a special option is initiated. The special options can use the following bytes in the plotting sequence and, as such, can involve one, two, three, or more bytes.

The first special option we need is a subchain option. I shall define the code as (O,O,O). When this code is encountered, the next byte in sequence is the subchain number. As one can see, there can be 256 subchains. You will probably never need all 256 unless you build a large multi-language or multi-font character set.

The second option needed is a 1/2 shift right option. The code I used is (0,1,0). This option increases the resolution of the plotting in the horizontal direction and comes in handy when plotting upper case alphabetic M, T, V, W and a number of other characters to make them symmetric in the particular grid frame I propose. It is a one byte instruction to shift the horizontal portion of the next move byte one half grid space to the right. That is, if the sequence of bytes (5,0,0) (0,1,0) (5,3,0) (5,4,0) was encountered, then the next two moves would begin at (5,0), move to (5,3-1/2) and end at (5,4) with pen down (or gun on).

These last two options I consider to be the minimum you should have in your system if you are to have the resolution required to plot large character sets.

Another option that could be used is the floating subchain option, (0,2,0). (This option is not shown in figure 6.) It takes three bytes of code to complete the sequence of this option. For instance, a period is used extensively for a number of punctuation characters and lower case i's and j's. The subchain sequence (1,0,1) (1,1,0) (2,1,0) (2,0,0) (1,0,0) plots a period in the lower left hand corner of the grid. Now, by using the floating subchain option, this period can be floated anywhere on the grid. A three-byte sequence (0,2,0), (SV,SH,0), (subchain no.) will move the period to any location desired by using positive offset values (SV,SH). This would save at least two bytes of storage for every different position of the period in the grid field, if there are more than two positions to be plotted. But, it takes some extensive programming to include this option; the advantage is large in large character sets, but minimal in small sets. Also, since timing is important in using CRT graphic systems, one must consider whether the extra computing effort is worth the savings in memory. I will leave it up to you to dream up exotic plotting options of your own for the 13 additional options which remain undefined.

Pointer and Move Sequence Tables

The pointer and move sequence tables now have to be established. A general schematic of the tables is shown in figure 2, along with the relationship of the tables to one another.

The primary pointer table defines the starting point in the character vector move sequence table, and the number of moves for each particular sequence. The pointer table is two bytes per character and shown in figure 3. Five bits of the first byte gives a maximum number of 31 primary steps per character. This is more than enough for any character contemplated, even if it were script or gothic. It is conceivable that a sequence table can be as large as 8 pages or 2 K bytes long. The remaining 3 bits of the first byte could designate the page number.

APPENDIX A
UNPACKING A VECTOR MOVE FROM AN EIGHT BIT BYTE

Using V, H and P to denote bits, the move is VVVVHHHP in packed form. The unpacking procedure is as follows:

1. An arithmetic shift right will make the Z function of the move available in the carry flag. The user can make use of this information through appropriate compares and jumps. Note that masking all but Bit P will also make the Z function available, but the action of shifting also readies the horizontal position of the move.
2. Temporarily store present value of the accumulator in any other register.
3. Mask the accumulator with octal 7. The horizontal position is now available. Send it out to the graphic device or store it for later use in another register.
4. Bring back the stored value of the accumulator from Step 2, shift right three times and mask the result with octal 17. Now the vertical portion is available.

The 8008 microprocessor assembly code would look like:

```
032          RAR          Shift right.
310          LBA          Load results temporarily in Register B.
  }      User defined portion using the Z function code in the carry flag.
301          LAB          Load ACC with value in Register B.
044 007      NDI 007      Mask the ACC with 007(8).
  }      User defined portion using the unpacked horizontal portion of the
         move.
301          LAB          Load ACC with value in Register B and rotate right
012          RRC          three times.
012          RRC
012          RRC
044 017      NDI 017      Mask ACC with 017(8).
  }      User defined portion using the unpacked vertical portion of the move.
```

The second byte would designate the starting point within that page.

This two byte table will fit into one 256 byte page of memory if there are 128 or less characters in your set. So, the full ASCII character set would fit easily in one page. The 7 bit ASCII code, if it resides in the upper portion of the address byte (bits 7-1) with a zero in the LSB of the byte, can address the location table directly. The location table for the subchains will also use the same format.

In figure 4, I give my version of the full ASCII 128 character set. Tables 1-3 give the values needed to plot this set. The tables contain octal 2235 (decimal 1181) bytes of data. The tables are set up so that you can easily reduce the size of the tables to a minimum set containing only 63 upper case alphabetic, numeric, and punctuation char-

Table 1. **PRIMARY POINTER VALUES**

Octal Address	Octal ASCII Code	Decimal Starting Location	Decimal No. of Moves	Octal 2-Byte Packed Code (see figure 3)
000	000	587	11	132-113
002	001	598	17	212-126
004	002	615	12	142-147
006	003	627	8	102-163
010	004	635	10	122-173
012	005	645	12	142-205
014	006	657	14	162-221
016	007	671	9	112-237
020	010	680	10	122-250
022	011	690	8	102-262
024	012	698	8	102-272
026	013	706	6	062-302
030	014	712	7	072-310
032	015	719	11	132-317
034	016	730	2	022-332
036	017	732	4	042-334
040	020	736	11	132-340
042	021	747	7	072-353
044	022	754	9	112-362
046	023	768	8	103-000
050	024	776	6	063-010
052	025	782	10	123-016
054	026	792	14	163-030
056	027	806	14	163-046
060	030	820	6	063-064
062	031	826	8	103-072
064	032	834	6	063-102
066	033	840	10	123-110
070	034	850	4	043-122
072	035	854	6	063-126
074	036	860	9	113-134
076	037	869	6	063-145
100	040	875	1	013-153
102	041	245	11	130-365
104	042	262	12	141-006
106	043	274	8	101-022
110	044	278	14	161-026
112	045	292	9	111-044
114	046	301	12	141-055
116	047	256	6	061-000
120	050	323	6	061-103
122	051	329	6	061-111
124	052	313	6	061-071
126	053	317	6	061-075
130	054	243	7	070-363
132	055	317	2	021-075
134	056	245	5	050-365
136	057	335	2	021-117
140	060	147	13	150-223

Table 1 (continued):

Octal Address	Octal ASCII Code	Decimal Starting Location	Decimal No. of Moves	Octal 2-Byte Packed Code
142	061	160	5	050-240
144	062	165	9	110-245
146	063	174	11	130-256
150	064	185	4	040-271
152	065	189	9	110-275
154	066	206	11	130-316
156	067	217	5	050-331
160	070	198	17	210-306
162	071	222	11	130-336
164	072	233	10	120-351
166	073	238	12	140-356
170	074	344	3	031-130
172	075	337	4	041-121
174	076	341	3	031-125
176	077	347	12	141-133
200	100	359	19	231-147
202	101	0	8	100-000
204	102	8	12	140-010
206	103	23	8	100-027
210	104	34	7	070-042
212	105	41	6	060-051
214	106	41	5	050-051
216	107	20	11	130-024
220	110	47	6	060-057
222	111	53	8	100-065
224	112	61	6	060-075
226	113	67	6	060-103
230	114	73	3	030-111
232	115	76	6	060-114
234	116	82	4	040-122
236	117	23	9	110-027
240	120	86	7	070-126
242	121	23	11	130-027
244	122	86	9	110-126
246	123	95	12	140-137
250	124	107	6	060-153
252	125	113	6	060-161
254	126	119	4	040-167
256	127	123	6	060-173
260	130	129	4	040-201
262	131	133	8	100-205
264	132	141	6	060-215
266	133	378	4	041-172
270	134	382	2	021-176
272	135	384	4	041-200
274	136	388	3	031-204
276	137	391	2	021-207
300	140	393	2	021-211
302	141	395	10	121-213
304	142	407	10	121-227
306	143	397	8	101-215
310	144	418	4	041-242
312	145	397	10	121-215
314	146	422	8	101-246
316	147	430	7	071-256
320	150	437	7	071-265
322	151	444	10	121-274
324	152	446	10	121-276
326	153	456	6	061-310
330	154	462	5	051-316
332	155	467	12	141-323
334	156	479	4	041-337
336	157	409	9	111-231
340	160	483	4	041-343
342	161	487	4	041-347
344	162	491	6	061-353
346	163	497	12	141-361
350	164	512	6	062-000
352	165	518	7	072-006
354	166	525	4	042-015
356	167	529	5	052-021
360	170	534	4	042-026
362	171	538	7	072-032
364	172	545	6	062-041
366	173	551	7	072-047
370	174	565	2	022-065
372	175	558	7	072-056
374	176	567	4	042-067
376	177	571	16	202-073

acters. Appendix B explains how to reduce the size of the tables to the minimum set. But, I encourage you to go in the opposite direction and build up other subsets to add to this basic set. For example: Greek alphabet and mathematical sets, or centered symbol sets for line graphs.

Table 2. **SUBCHAIN POINTER VALUES**

Octal Address	Decimal Subchain Code	Decimal Starting Location	Decimal No. of Moves	Octal 2-Byte Packed Code
000	1	397	8	101-215
002	2	409	8	101-231
004	3	438	6	061-266
006	4	598	6	062-126
010	5	617	10	122-151
012	6	627	6	062-163
014	7	604	5	052-134
016	8	578	4	042-102
020	9	578	9	112-102
022	10	671	5	052-237
024	11	617	6	062-151
026	12	701	5	052-275
030	13	598	11	132-126
032	14	571	7	072-073
034	15	571	11	132-073
036	16	587	4	042-113
040	17	665	6	062-231
042	18	629	4	042-165
044	19	784	6	063-020
046	20	802	4	043-042
050	21	591	4	042-117
052	22	810	10	123-052
054	23	810	4	043-052
056	24	627	5	052-163
060	25	842	6	063-112

Position, Orientation, and Scale

Now that we have the ability to pull out the coordinates for a sequence of moves, we have just begun the job of plotting a character chain. We must translate each character into its appropriate position on the plotting media, then scale it up or down, rotate it into the proper position, and if desired, slant the character. What usually is done is to build conversion coefficients prior to plotting the desired character string. While going through the process of plotting, these coefficients transform the move coordinates residing in the move sequence table to the appropriate coordinates on the plotting media.

This requires that you have the capability of multiplying and dividing floating point numbers in your system. I assume you will either have a calculator chip interface or a floating point software package to draw on. Additionally, you will need the capability of obtaining sines and cosines if you want the ability to rotate the character string out of a horizontal position or to define the slant of a character with an angle.

Before we get into the procedure of shifting, twisting, stretching or squeezing the characters onto the plotting media, we must define a few parameters which are required prior to plotting the character string. In

Table 3.

MOVE SEQUENCE VALUES

000	040	100	140	200	240	300	340	000	040	100	140	200	240	300	340
121	167	132	124	376	325	176	224	307	234	347	226	365	242	346	262
320	076	176	132	121	370	256	260	350	174	002	206	370	142	326	000
364	121	376	176	376	130	312	320	370	152	146	147	130	355	330	003
372	360	121	216	361	125	300	364	366	142	371	144	124	134	271	263
336	372	360	252	136	134	360	372	346	365	346	124	241	000	266	022
136	336	377	244	361	321	374	336	350	320	264	126	304	001	126	000
221	176	200	300	002	364	273	276	305	360	224	146	236	353	026	002
236	132	245	320	226	372	314	172	346	376	146	273	101	372	062	000
121	120	136	364	376	336	354	124	366	120	130	310	116	364	363	001
360	241	361	372	002	276	372	311	364	137	367	306	323	124	122	275
372	250	120	336	227	254	364	330	344	132	350	264	214	127	143	034
354	377	136	361	002	202	342	326	346	176	272	224	275	122	274	263
314	360	121	376	126	120	302	306	311	136	232	206	134	243	207	264
272	120	360	002	213	136	264	310	352	255	150	210	155	250	134	124
260	136	002	367	304	363	221	171	372	130	126	232	132	000	371	205
273	121	126	002	361	376	264	210	370	124	125	313	124	001	366	272
236	360	376	126	376	314	272	206	350	160	372	212	142	275	126	234
176	241	136	361	120	272	236	166	352	220	301	174	242	054	125	143
132	256	121	160	136	266	176	170	221	326	316	216	264	032	130	124
120	377	360	124	125	273	132	067	236	346	201	336	272	024	121	132
251	136	136	132	142	236	124	130	277	364	216	372	254	042	260	154
256	365	376	176	200	176	160	131	260	362	161	364	214	363	241	174
136	372	121	376	300	132	220	150	367	340	256	320	202	122	262	212
337	002	360	361	342	124	324	146	124	320	320	160	363	243	264	204
372	367	372	002	364	160	372	126	131	136	337	124	122	264	246	222
364	002	336	126	372	133	321	130	372	143	240	132	143	272	126	242
320	126	276	376	354	372	360	002	355	354	176	373	124	254	247	264
160	125	232	361	316	160	376	167	344	343	341	366	132	134	270	272
124	132	220	122	216	176	210	370	322	154	364	126	154	125	272	254
132	221	231	002	154	161	130	366	262	241	372	132	254	130	254	000
176	160	136	246	132	124	277	002	244	256	336	365	272	331	134	000
336	124	161	134	124	132	232	166	252	002	254	132	264	350	123	000
037	077	137	177	237	277	337	377	037	077	137	177	237	277	337	377

UPPER CASE ALPHA | NUMERALS | PUNCTUATION | LOWER CASE ALPHA

An Aside:

The techniques used in this article can be directly applied to any repeatable set of plotting sequences for display on a vector graphics device. For example, the chess pieces and chess board of a chess game display are one possible data display; similarly, a Space War game's space ship symbol output to a graphic display device could use techniques of vector generation and rotation.

figure 5, we see that we need the standard height (S) and width (W) of each character, the gap (G) between each character, the starting coordinate position (X_O,Y_O) of the character string defined as the baseline origin (identical to the relative origin), and the angles (θ&β) defining the orientation and slant of the character string. These parameters must be made available prior to plotting the character string.

Now, let's list the formulae you will use in your plotting routine.

1. Scale Equations
   - SS = S/10.0 vertical scale (1)
   - SW = W/7.0 horizontal scale (2)
   - SG = G/SW width-gap ratio (3)
2. Rotation Equations
   a. Horizontal (H) portion of move
      - HX = Cos θ (4)
      - HY = Sin θ (5)

   b. Vertical (V) portion of move
      - VX = $-$Sin θ = $-$HY (6)
      - VY = Cos θ = HX (7)

   c. Vertical (V) portion of move corrected for the slant
      - VX = $-$HY + HX*Sin β (8)
      - VY = HX + HY*Sin β (9)
3. Final Coefficients for Rotation and Scale
   - DHX = HX*SW (10)
   - DHY = HY*SW (11)

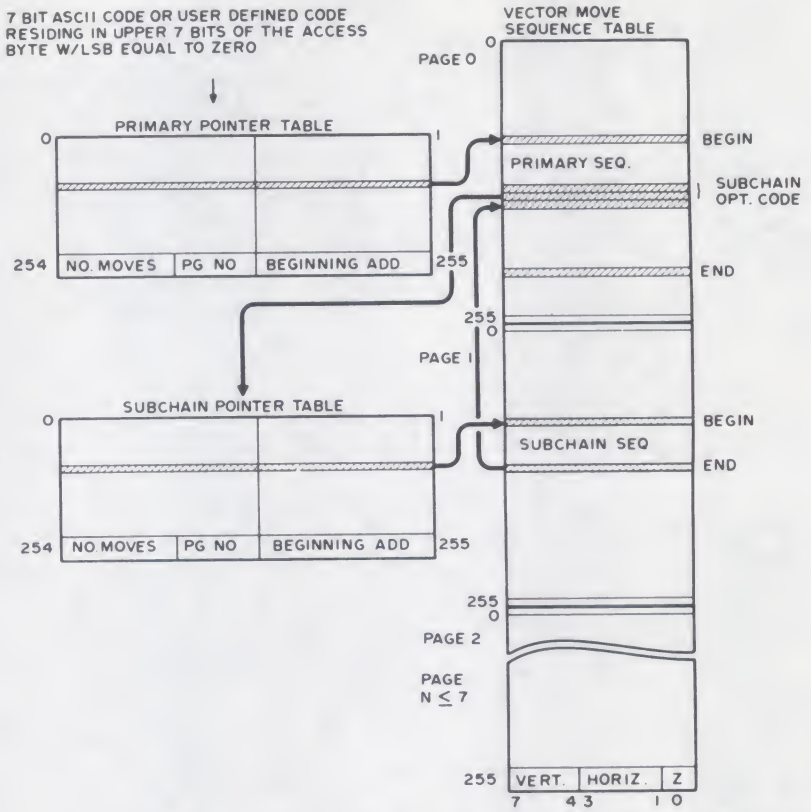


Figure 2: Relationship of the Character Generation Tables. The selected character code is rotated left by one bit to define a number from 0 to 254. This number accesses a 16 bit quantity in the primary pointer table. The primary pointer table in turn locates the beginning of a series of pen locations in the move sequence table which define the character's plot representation. Within that series, there might be a pointer to the subchain table, which in turn points to an often used fragment of the graphics representation located at a different place in the move sequence table. Note that to minimize retrieval effort on machines such as the 8008 and 8080, sequences of moves should be restricted to single pages of memory.

000	040	100	140	200	240	300	340	000	040	100	140
263	274	264	264	176	346	225	000	000	002	272	320
272	205	260	171	002	324	230	016	017	166	172	325
331	212	273	070	172	306	361	265	171	071	000	266
346	263	264	131	002	266	002	164	176	170	004	000
126	274	164	136	072	000	262	172	076	076	000	031
132	122	172	177	000	006	366	177	070	176	025	361
263	134	225	076	006	000	000	170	133	000	000	260
142	373	230	000	165	011	013	070	136	006	026	266
124	366	171	004	264	000	367	076	000	000	000	366
132	270	070	265	172	012	360	131	017	013	006	000
154	246	076	272	272	000	260	134	117	177	273	031
275	230	261	002	115	006	321	000	110	170	264	137
134	126	360	267	076	273	324	017	174	070	224	
263	132	266	002	070	264	000	153	074	076	232	
002	365	366	166	170	224	014	174	000	116	172	
126	370	265	171	176	232	367	074	020	134	164	
274	266	164	076	076	172	360	073	165	130	000	
261	250	172	071	261	164	260	076	264	135	027	
122	226	272	176	360	261	266	000	272	156	000	
206	130	171	321	366	360	165	017	172	176	030	
132	124	070	324	266	321	264	151	225	000	000	
274	371	076	367	321	326	272	172	232	022	031	
123	130	367	360	326	367	232	176	000	000	000	
274	221	360	260	000	266	224	136	021	023	022	
263	264	320	266	010	000	231	110	000	000	326	
134	232	326	000	171	013	172	070	004	024	324	
263	276	266	005	070	361	000	076	265	000	000	
124	261	260	000	111	260	015	000	224	006	031	
132	360	265	006	176	266	000	000	232	165	261	
025	364	164	000	133	273	004	000	272	264	360	
030	346	172	007	076	264	271	000	002	002	366	
132	306	272	171	367	164	170	000	227	226	326	
037	077	137	177	237	277	337	377	037	077	137	177

LOWER CASE ALPHA | ADDITIONAL PUNCTUATION | ASCII CONTROL CHARACTERS

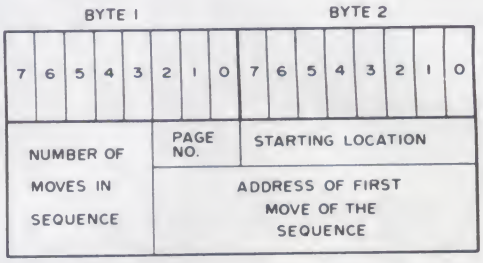


Figure 3: Primary and Subchain Pointer Formats. The pointer tables are composed of two byte elements which contain information on the number of moves required, and the address of the first move of the sequence.

As always, climb the highest mountain rather than be content with a mole hill.

$$DVX = VX*SS \quad (12)$$
$$DVY = VY*SS \quad (13)$$

4. Shift Coefficients Between Character Baseline Origins

$$DSX = DHX\ (7.0 + SG) \quad (14)$$
$$DSY = DHY\ (7.0 + SG) \quad (15)$$

5. Final Transformation Equations to be Applied to Each Move

$$X = XO + H * DVX + (V-5.0) * DHX \quad (16)$$
$$Y = YO + H * DVY + (V-5.0) * DHY \quad (17)$$

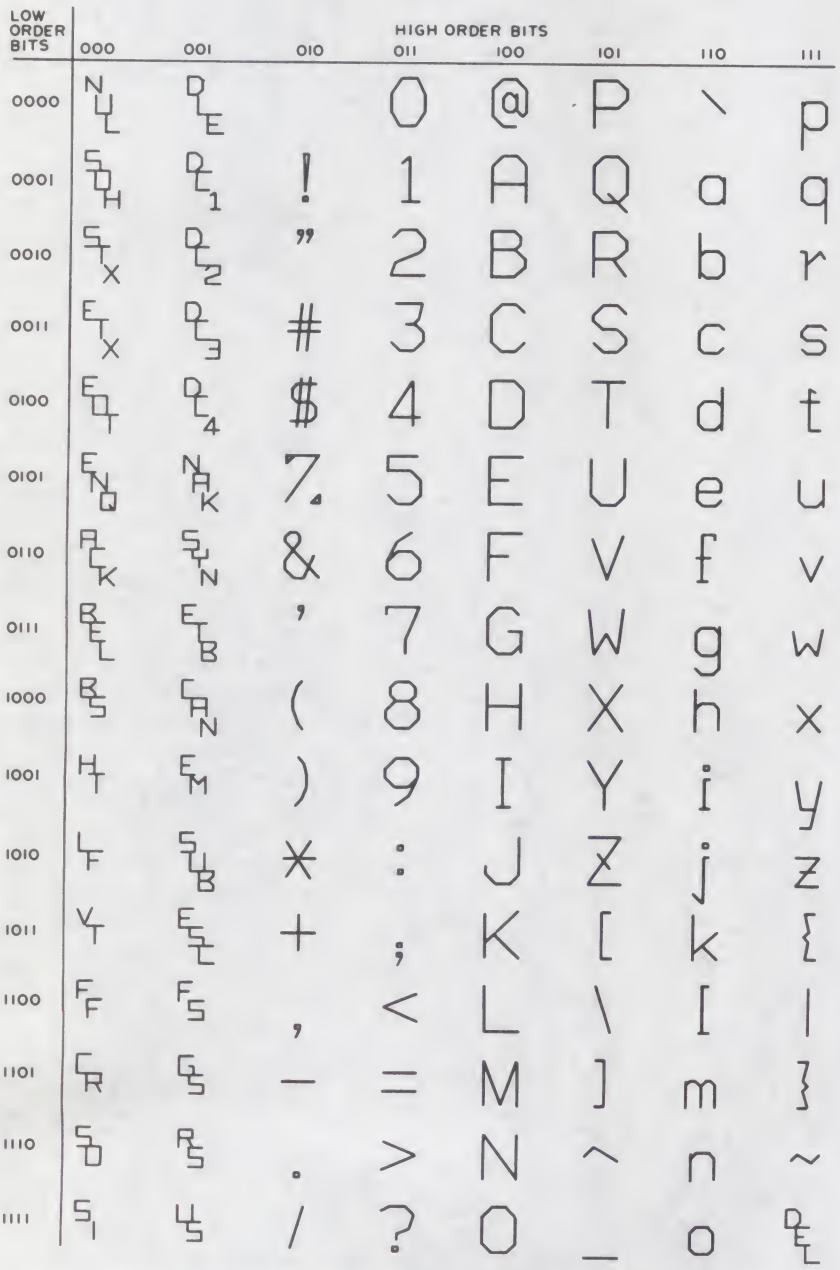


Figure 4: an ASCII Graphic Character Set. The plotting tables 1-3 are used to define this set of characters when displayed or drawn on an XY plotter. This figure was prepared by the author, using a commercial plotter as the output device.

6. Shift Relative Origin from Character

$$XO = XO + DSX \quad (18)$$
$$YO = YO + DSY \quad (19)$$

Formulas (1) through (15) are calculated prior to plotting the first character of a line. The coefficients thus derived will not change throughout the plotting of the character chain. Note that the equations simplify considerably when the angles θ and β are limited to special cases. Two common special cases are $\theta = 0°$, $\beta = 0°$ and $\theta = 90°$, $\beta = 0°$. Substituting the special values of sine and cosine for these angles produces the special cases. These values are:

$$SIN(0) = 0.0 \quad COS(0) = 1.0$$
$$SIN(90) = 1.0 \quad COS(90) = 0.0$$

Equations (16) and (17) are the transformation equations used during the plotting where only the values (H) and (V) change for each move. XO and YO are updated as we move to the next character in line to be plotted by using equations (18) and (19).

Plotting Routine

The plotting routine is outlined in figures 6 and 7 as a flow chart. If you have BASIC, you should not have any problems imple-

APPENDIX
ABRIDGING THE ASCII PLOTTING TABLES

NOTES:

1. To abridge the plotting tables, do the following:
   A. For upper case **alphabetic,** numerals, and punctuation, only use:
      Primary Pointer Table — bytes (octal) 100 to 301
      Subchain Pointer Table — none
      Move Sequence Table - bytes (octal) 0 to 613
   B. For all characters except ASCII control characters, use:
      Primary Pointer Table — bytes (octal) 100 to 375.
      Subchain Pointer Table — bytes (octal) 0 to 5
      Move Sequence Table — bytes (octal) 0 to 1070
2. If you want abridged Set A above, note that you do not need to include the traps for special subchain option in your program.
3. Note that the move sequence table is set up so that no sequence of moves crosses the boundary of a 256 byte page of memory. This eases the programming of micros such as the Intel 8008 or 8080.
4. Note that the blank or space character was included at the end of the move sequence table. If you abridge the table, move the code to the end of your abridged table and correct the location code in the primary pointer table. Better yet, include in your program a trap to catch any spacing, as there is no actual plotting for this character. Just shift the relative origin to the next character to be plotted.

menting this routine, as BASIC has the required floating point arithmetic and the transcendental functions, sine and cosine. If you plan to implement the routine in machine language, then I dare say you will have a little more work cut out for you. But, the advantage of going this route is that you will take full advantage of your microcomputer's design in order to minimize the use of memory and increase the speed of plotting. Speed is very important if you have a CRT graphics terminal, because of the refreshing problem.

Summary

In summary, I think you have here a start in creating your own vector character generation package on whatever graphic media you have or plan to use. You can implement the package as I have given it to you or abbreviate, expand, or abridge the package to suit your needs.

I encourage you, though, to expand the

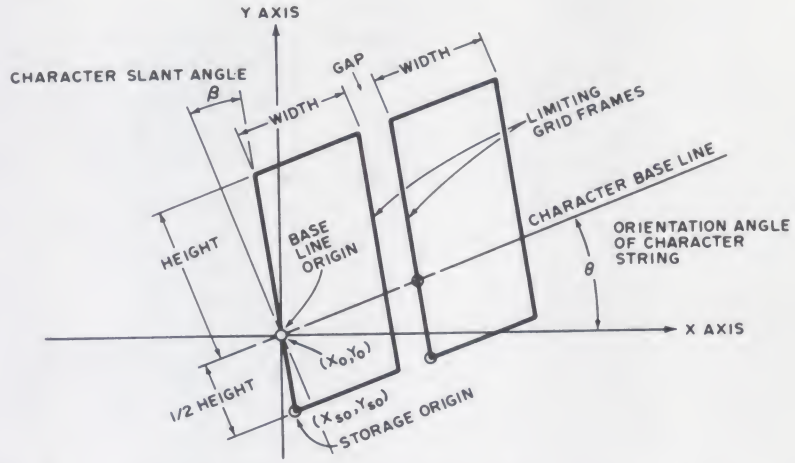


Figure 5: Character Orientation. To add an element of finesse to the plotting function, provision for general purpose rotation and slanting is a desirable feature. There are two angles to specify: angle θ is the orientation angle of the baseline for a character string; angle β is the frame slant relative to a perpendicular through the base line.

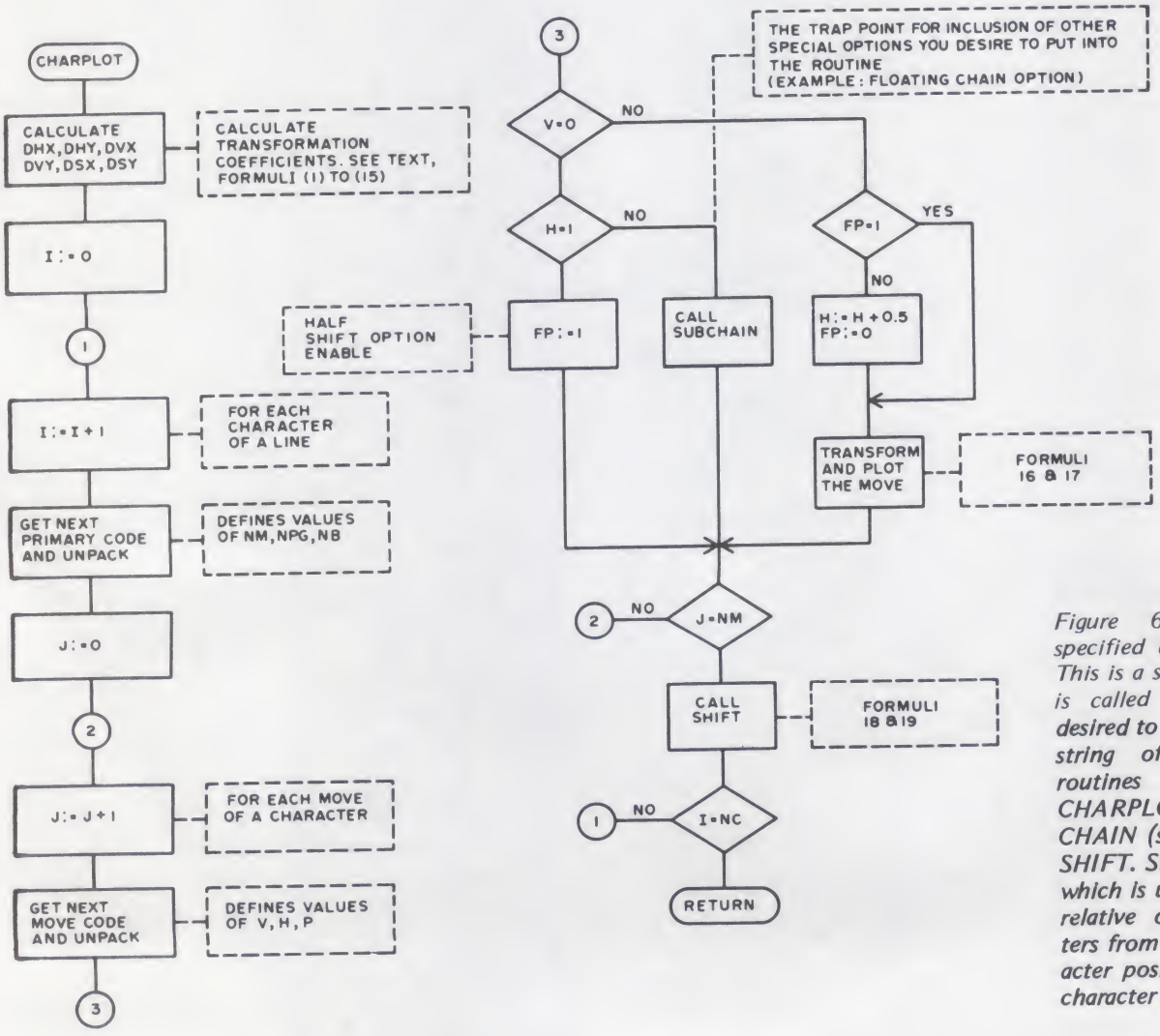


Figure 6: CHARPLOT specified as a flow chart. This is a subroutine which is called whenever it is desired to draw a character string of output. Subroutines referenced by CHARPLOT are: SUBCHAIN (see figure 7) and SHIFT. SHIFT is a routine which is used to move the relative origin of characters from the present character position to the next character position.

basic character set I have given you to include foreign language alphabets, a music symbol set, a mathematical symbol set, or a centered symbol set for line graphs. The horizon in character plotting is only limited to your own efforts or imagination. Climb the highest mountain, rather than be content with a mole hill.■

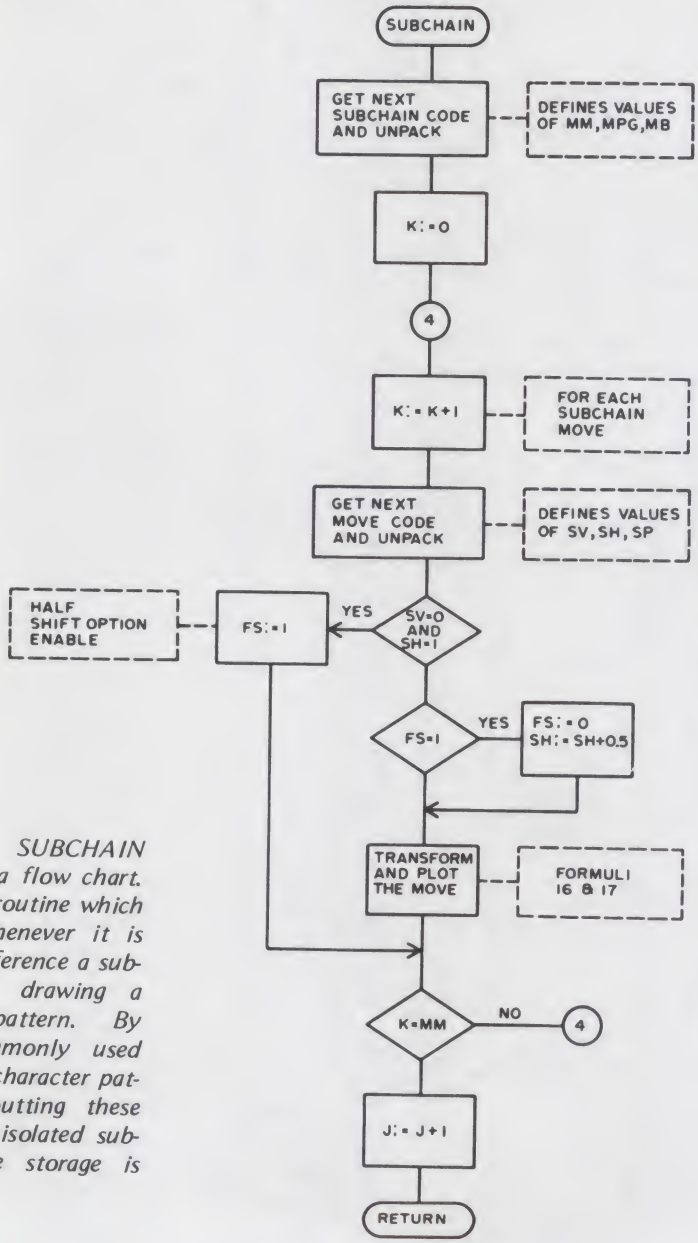


Figure 7: SUBCHAIN specified as a flow chart. This is a subroutine which is called whenever it is desired to reference a subchain when drawing a character pattern. By picking commonly used segments of character patterns and putting these segments in isolated subchains, table storage is conserved.

GLOSSARY

Absolute origin: In a typical plotter or display device, there is an absolute origin for all possible positions of the writing mechanism. A common location of this origin is the lower left hand corner of the plotting field, so that points to the right and above can be specified by positive integer displacements.

Byte: A cell in memory which can store 8 bits of information.

Chain: A set of vector moves to be performed sequentially.

Chain plotting: The technique of specifying a movement of the plotting or display mechanism by a series of small movements.

Character frame: A small region of the plotting medium in which motions will take place while plotting a single character. See figure 1.

Coordinates: A point in a two dimensional space can be specified by a pair of numbers. These numbers are the coordinates of the point with respect to a reference point called the origin.

Masking: The technique of selecting bits for inspection using the AND operation and a mask. The word which is being tested is combined with the mask using the AND operation. Every logical 1 bit in the mask will select a corresponding bit in the word being tested; every logical zero bit in the mask forces a zero in the result independent of the word being tested.

Medium: A plot or a display is usually performed on a two dimensional object which can be viewed by a human being. In the context of this article, the medium is the piece of paper or display tube on which you see the resulting characters.

Page: In many microcomputers it is convenient to divide memory into blocks of multiple bytes, called pages. In the context of this article, the Intel 8080 and 8008 definition is intended: a block of 256 bytes whose high order address byte is identical.

Plotting frame: The range of possible positions for the plotting or display mechanism. In most equipment, this is a grid of points specified by two integer coordinates for horizontal and vertical position.

Relative origin: A local origin which is used for convenience of programming. The relative origin is specified by a coordinate pair with respect to an absolute origin of the mechanism used; movements involved in plotting a character are specified with respect to the relative origin to simplify placement of character patterns.

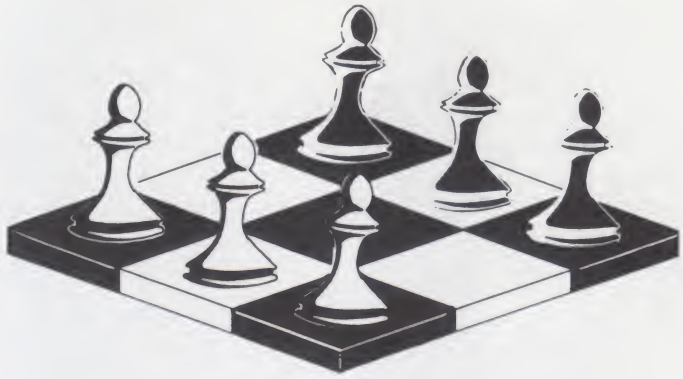
Resolution: A degree of detail involved in the plot. Ultimately this is limited by the resolution of hardware, which is specified as the number of points per linear inch (or centimeter) of display in each coordinate direction.

Subchain: In a chained plotting table, a subchain is like a subroutine of a computer program. It is a fragment of a plot which is often referenced, so use of the subchain economizes the memory requirements of the data tables.

Vector move: In the context of this article, a vector is a line segment which connects two points in the plotting frame. A vector move is the act of moving the plotting mechanism (pen or electron beam) from one of the points to the second point. In a chained approach, as used in this article, the starting point is implied by the last position of the mechanism and the ending point of the move is specified by the coordinates of the position.

HEXPAWN

A Beginning Project



in ARTIFICIAL Intelligence

What is intelligence? Pushing aside the philosophical and psychological questions for the moment, I can offer an operational definition of intelligence in programs: An "intelligent" program is one which was designed with a range of possible circumstances in mind, rules defining successful and unsuccessful responses to such circumstances, memory of the history of past responses and relevant circumstances, and an algorithm for using such past history information when similar circumstances occur again. Robert Wier has provided an example of a simple game application which illustrates this definition of intelligence in programs. Does it sound too deterministic for you? Hardly – the response is in some sense inherent in the program and its context. But, just as in natural life, the order and degree of the various inputs to the AI program cannot be predicted in advance with any great certainty. Just as each individual person is unique, each individual run of a good AI program will tend to differ – AI programs, like people, are good for lots of surprises.

by
Robert R. Wier
1208 Mistletoe Drive
Fort Worth TX 76110

Artificial intelligence. The very words themselves are at once frightening and fascinating. Hal lip reading; Colossus communicating with Guardian in a *real* "machine language"; M5 taking over the Enterprise. Yet these are still media creations, and we are cushioned by the comforting buffer of a movie or TV screen. To realize what artificial intelligence (or AI) is *really* like, you have to create it yourself (ever have an urge to play Frankenstein?). HEXPAWN originally appeared in *Scientific American* (Vol. 206, No. 3, p. 138) in Martin Gardner's "Mathematical Games" column. It is simplicity itself. The game board is identical to that of the standard two-dimensional tic-tac-toe, and two players control three pieces (or Xs or 0s or whatever) each. Each player's objective is to advance his pieces to the opposite side of the board, or eliminate or block the opposition's pieces. Moves of each piece are the same as the pawn in chess (i.e., move 1 forward to a vacant square, take diagonally).

HEXPAWN rules are very simple: To win, attempt to move one of your pieces to the opponent's side of the board, or block him from making any move. Moves are those of the pawn in chess. That is, you may move one square forward to an unoccupied square, or you may move one square diagonally in order to "take" an opponent's piece. Only these two moves are allowed. You may not move diagonally *without* a "take"; you may not move forward *with* a "take". Fig. 1 illustrates a typical game situation of occupied and unoccupied squares. In this "model" of the layout, the computer(X) can move in two ways which "take" the human pawn in the central square (number 4). The computer can move in one way which will not "take" a human pawn.

For a complete explanation, please refer to the original article (every library should carry *Scientific American*, and if yours doesn't, ask them why).

This version of HEXPAWN is a game that *learns*. You

may play it several times beating the computer (which keeps track of the board, as well as acting as one of the players) with ridiculously simple strategies. However, as play progresses, the computer notes its mistakes, and eventually, after 8 to 10 games, you may only tie or lose to the computer. The machine has "learned" how to play the game successfully.

The method described here to implement HEXPAWN is strictly brute force, and many techniques may be used to improve both the execution time and storage efficiencies. But in order to fully appreciate the internal workings of HEXPAWN, it is nice to keep it simple. Also, since this is a self-modifying program (a necessity in almost all AI), programmers will recognize that "simple is good," since after the code runs wild a few times and produces strange and wonderful results, it is fortunate to have code which is easy to debug.

HEXPAWN was implemented by the author on a 16-bit/word mini with an assembler. In this version it occupies 88E hex bytes, or 2190 decimal bytes, or 4218 octal bytes. It would be possible to reduce the memory requirement by using two or three bits instead of two bytes for the board representation of the playing pieces, but this would require a lot of bit diddling that is tedious unless you are really tight on memory. The minimum representation of the three states requires a two-bit binary number, using three of the four possible states of two bits. This requires only one word of memory. A less compact but easier to program bit level representation is to use three bits, one for each state. Only one bit would be "on" at any given time if the corresponding state is present. But on many computers it's considerably simpler to use two bytes so that pieces may be represented by "X", "0", and " " (space). The storage requirement will also vary considerably with the nature of the peripherals used, due to whatever interface programming is necessary. The original was implemented with a CRT where the cursor was ''locked'' in synchronization with a programmed counter notifying the program of the board location of the square being referenced.

Basically, the structure of the implementation is quite simple. In the *Scientific American* article, all possible board configurations are presented. Note that some are mirror images of others, but these are still required. These board configurations are hereafter referred to as ''models''. The program attempts to match the current board configuration with the models stored in memory. When a model is found, several courses of action may be available. In some cases, only one move will be possible, thus the computer is limited to that move. In other cases several moves are possible. The computer will select one (whichever is first on the list) and make the move. If a model is not found, this is an error situation; an illegal move has been made on *your* part, and an error message should be output. Fig. 2 is a "macro" flowchart of this process.

Following each model in memory is a string of move index bytes followed by a hex "FF". The "FF" is used as a terminator for that particular model. The bytes between the model and the "F" are index numbers for possible moves — the index references a jump table to produce a correct move by executing a jump.

A jump table is a very handy device when you need to reference several different

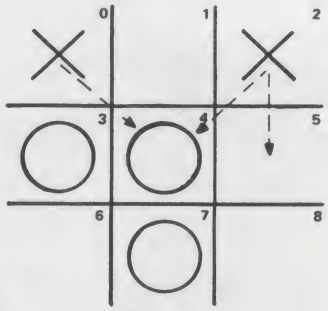
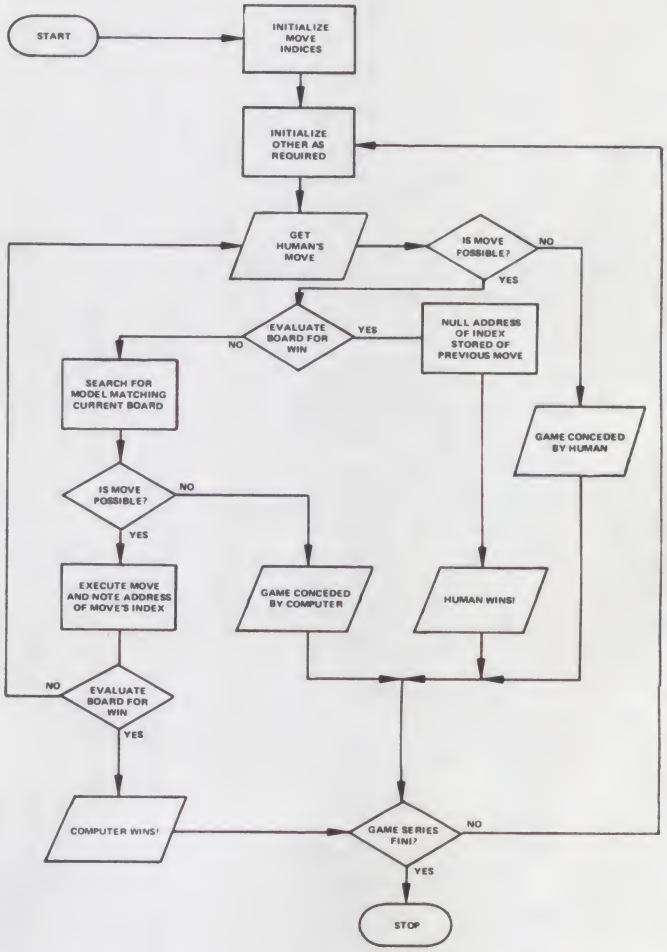


Fig. 1. The game layout for a typical move.



Note that the move indices are initialized only once for each series of games. Initialization for each game will defeat the learning process.

Fig. 2. Control flow logic for the HEXPAWN program.

Fig. 3. Table of All Possible Moves (Models).

	BOARD POSITION MODEL SQUARE									COMPUTER'S POSSIBLE MOVES (see Fig. 4)
	0	1	2	3	4	5	6	7	8	
	X	X	X	0				0	0	3, 4, 7
	X	X	X			0	0	0		1, 4, 5
	X	X	X		0		0		0	1, 2
	X		X	X	0				0	2, 6, 8
		X	X	0	X				0	3, 7, 11
EXAMPLE OF FIG. 2	X		X	0	0			0		2, 6, 7
	X	X		0		0			0	3, 4, 5
		X	X		X	0	0			5, 10, 11
		X	X	X	0	0	0			5, 6
	X		X	X		0		0		8, 9
	X	X		0	0	X			0	2, 3
		X	X	0		0	0			3, 4, 5
		X	X		0			0		6, 7
		X	X		0		0			6, 7
	X		X	0				0		7
			X	X	X	0				8, 11
	X			0	0	0				2
		X		X	0	0				8, 5
		X		0	0	X				3, 14
	X			X	X	0				8, 11
	X		X	0		0			0	15
	X			0	X			0		15
			X	0	X	X				11, 14
			X	X	0					6, 7, 8
		X		0	X					3, 11
		X		X	0					5, 11
	X		X	0						2, 8
			X		0	X				6, 14
	X			0	0					2
	X		X		0	0		0		1, 2, 6
		X			0					15
	X		X	0	X	0		0		15
			X	0	C	0				6

Key: **X = computer piece occupies square**
0 = human's piece occupies square
blank = square is empty

locations in your program using numerically sequential indices. The advantage is that after assembly, debugging is facilitated. If you desire to change all the jump addresses of a particular segment of code, you need only change the jump table, rather than each reference containing the desired jump address. It is also unnecessary to worry about having to make the code referenced in the jump table equal in length. All that is taken care of in the jump table itself in an easy and consistent manner. The jump table is particularly appealing in that you have multiple-level-indirect addressing capability.

HEXPAWN learns by removing the index which leads to a defeat for the computer. Thus, if a move to square 8 results in a loss, the index following the appropriate model is changed to a null character, which *eliminates the losing move.* It is easily seen that if a particular move always leads to a loss, it will be completely nulled, thus allowing the computer to "know" several moves ahead that it has lost the game. As each losing move's index is nulled, the learning process effectively progresses toward earlier moves.

As noted in the original article, this version only penalizes the *losing* move. Also of possible consideration is the rewarding of a winning move, but this would complicate our code considerably.

For convenience's sake let us number the squares of the playing board 0-8 starting in the upper left corner, working horizontally and down. Let us also establish the convention that the human player always moves first. This does not seem to compel a deterministic game. That is, either player may win regardless of who moves first. Now suppose that there has occurred a particular board configuration (Fig. 2). Note that the "X" pieces belong to the computer, while the "0" pieces are yours. You have just made the preceding move, and now the computer must decide what to do. The computer's possible moves are indicated by the dotted lines. But how does the computer know this? It searches through memory until the following bit pattern is found (in hex):

E740E7D6D64040D640020607FF

The first 9 bytes represent the board. Note that in EBCDIC, E7 is an "X", D6 is an "0", and 40 is an " ". Remember that these are EBCDIC codes (my peripherals use it), but it could just as easily have been ASCII. The next three bytes represent the indices for possible moves as they exist at the beginning of the game. That is, the possible moves are these:

02: "X" in square 0 moves to square 4 taking your "0"

"To realize what artificial intelligence is *really* like, you have to create it yourself . . . "

06: "X" in square 2 moves to square 4 taking your "0"

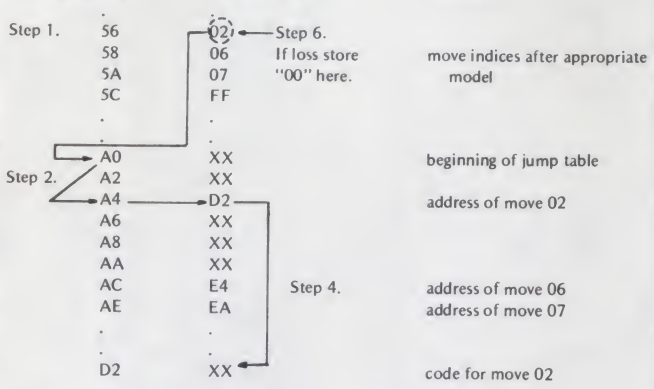
07: "X" in square 2 moves to square 5

Now, either move 02 or move 07 will result in a loss the next move that "0" makes (assuming that you are trying to win) and *that* index will be nulled so it cannot be selected again in the event of this same board configuration. Move 06 is correct since it removes your piece and also blocks you from obtaining "X's" side of the board. Since the computer simply selects the first move on its list, the first time this board configuration is encountered, the computer will lose (as a result of move 02). However, after this game the computer will select move 06, which is correct, since it is next on the list. The number of the index has no particular significance; it could be anything as long as it denotes the displacement needed in the jump table to direct the flow of control to the proper code for the move desired. The "F" is a terminator that signals the end of that particular model and move list.

We will not present the actual code to accomplish the HEXPAWN algorithm since there are so many machines of a differing nature in hobby use. However, copies of the author's LOCKHEED SUE Minicomputer version are available from him for $3 to cover the most of duplication and postage.

A few hints are in order to help you avoid some of the more obvious problems. The

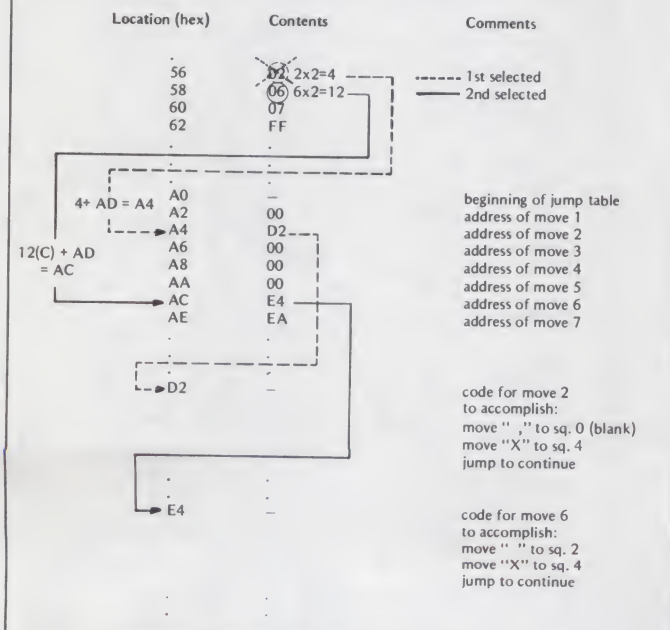
EXAMPLE 1: To illustrate, assume that the following is in memory at the start:



The "learning" sequence is composed of the following steps:

1. Search models until match is found.
2. Select first index of possible moves, add to location of beginning of jump table, giving location of address of that move's code – in this case, index 02 x 2 (to get even byte boundary) + A0 = A4. If no possible move (no non-null index) is available, concede game to human player.
3. Note the address of the index used – in this case "56."
4. Jump using indirect addressing to the move's code and execute – in this case, location D2.
5. Evaluate board for win or loss.
6. If loss has occurred, null the location of the last index used – in this case "56", thereby removing this move from the machine's repertoire of responses to this *particular* board configuration. If a tie or computer win has occurred, do nothing to the index.

EXAMPLE 2: Assume that the following is in memory *after* example 1 is completed:



Suppose move index 2 has been selected. The index "2" is multiplied by 2 (shifted left 1 bit) in order to produce an even word address, and added to the address of the beginning of the jump table – A0 – resulting in an address of – A4 –. At location A4 is the address – D2 – of the code to accomplish move # 02. At location D2, move 2 consists of blanking the computer's "X" in square 0, and inserting an "X" at square 4, taking your "0". Since this is a losing move, the index 02 will be made null (replacement by 00 is good for error checking), and move 06 will be accomplished in the same manner next time this board configuration occurs.

biggest hang-up with this program is to get it running correctly in regard to the jump table. If a wrong index is obtained, the program will run off into the boondocks and never be heard from again. Therefore it is nice to include checks on the validity of the index retrieved and to output an error message in the event of something strange happening. A reasonable board may be printed using dashes and exclamation marks. However, if you do this, you will have to "unpack" the board as represented in memory into a more suitable form for I/O. If you don't have a CRT with machine programmable cursor, you can use the numbers assigned to the squares to indicate your moves. Of course you'll want the machine to have a variety of responses for being blocked, losing and winning. For debugging it is good to output the number of the index which is nulled after a losing game. In this way you may keep track of the learning process as it advances. Also you should be aware that if the human player makes some illegal moves, no model will be found, and a message should be output indicating this fact.

But not *all* illegal moves will result in an error condition. In this case, should the human player win, the machine will null the last move's index *even if it is correct.* After this happens a few times, the machine will start making illegal moves, acting illogically, and generally approximating a nervous breakdown!

Programming HEXPAWN will painlessly (?) introduce you to a number of worthwhile aspects of the logical arts. You'll see that many segments of code (such as the board evaluation) are similar and are potential

---

In a written communication, Martin Gardner points out that the original Hexapawn article is reprinted as Chapter 8, "A Matchbox Game-Learning Machine" in his book The Unexpected Hanging and Other Mathematical Diversions *(Simon & Schuster, 1969). The book version includes updates of the drawings in the original* Scientific American *article, notes on reader reactions to Hexapawn, and reference to an article on the more general game "Extendapawn." Our thanks to Martin Gardner for his assistance in supplying a corrected version of Fig. 5 for use in BYTE.*

Fig. 4. Table of Computer's Moves ("X" Graphic).

COMPUTER'S (X's) MOVES

MOVE INDEX #	SQUARE TO SQUARE		COMMENTS
1	0	3	
2	0	4	
3	1	3	
4	1	4	
5	1	5	
6	2	4	
7	2	5	
8	3	6	computer wins!
9	3	7	"
10	4	6	"
11	4	7	"
12	4	8	"
13	5	7	"
14	5	8	"
15	–	–	computer blocked

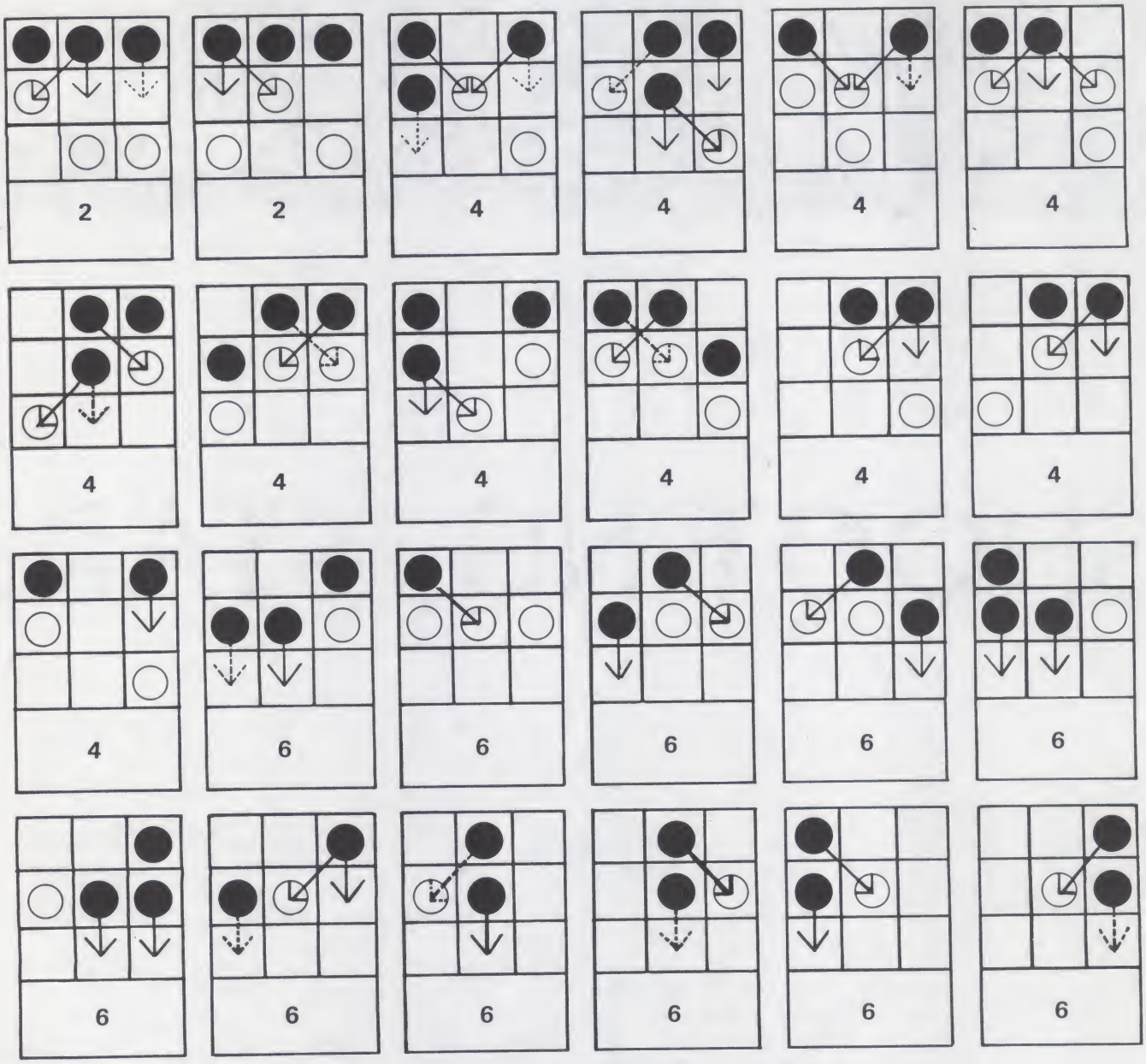


Fig. 5. The set of possible Hexapawn game situations faced by the HEXPAWN program after 2, 4 and 6 moves. (Reprinted from Chapter 8, "A Matchbox Game-Learning Machine," in The Unexpected Hanging and Other Mathematical Diversions *by Martin Gardner.)*

A BASIC Version of This Program:

For those with systems running the BASIC language, a BASIC version of this program called HEX is found on page 122 of the third printing of *101 Basic Computer Games*, available for $7.50 + 50¢ postage from Digital Equipment Corp., Software Distribution Center, Maynard MA 01754.

candidates for subroutines. You'll see that indirect addressing does indeed have some practical uses, if you can ever get the code debugged. You'll see that it is *very* important to try and anticipate possible sources of error in your code before you run the program, and at least to include a mechanism to warn you when problems occur. (I didn't anticipate any problems with the jump table and consequently spent several hours trying to figure out how the move indices were coming up with such strange values. If I had put in some code to check them first, this process would have been shortened considerably.) You'll see that some programs are complex to such a point that you simply cannot sit down and write them without thinking about the logical design first! You'll see why you should *never, ever* write programs that are self-modifying in nature (except AI, naturally).

Lastly, amaze (antagonize) your friends by sitting down at your computer and winning four or five games, then inviting them to try. When they can't, you can smile smugly and explain how your computer learns from its mistakes, and so should they!

Figure 1: Three special patterns of stars and black holes. The game begins with a single star representing the Big Bang theory (left), and is won when the pattern of only one central black hole is achieved (center). The pattern shown on the right represents a loss and terminates the game.

SHOOTING STARS

Willard I Nico
Delta t
11020 Old Katy Rd, Suite 204
Houston TX 77043

There are probably as many reasons to have a computer in the home as there are computers in homes. For whatever reason you have one though, it's only human nature to want to show it off to other people.

Say you have a super program called "Investment Portfolio Analysis and Statistical Summary" (IPASS) up and running on your Scelbi 8H or whatever. It took months to write and debug the program and it involved several unique concepts of which you are justifiably proud. You can picture the furious activity going on inside the little heart of the computer and would dearly love to show off your skill to Mr and Mrs Nexdor and bask in their admiration. So you invite them over for cocktails.

The program runs flawlessly and, as the results flash on the display screen, you step back slightly to receive your praise. Mr Nexdor looks at you with a blank expression and says, "But will it grind pepper?"

That actually happened to me. One way around this problem is to save IPASS for your own enjoyment and have a game program or two available to show off. Of course, for some people game programs are the primary interest in having a home computer. Whatever your games interest, I think you'll find SHOOTING STARS an interesting addition to your library.

I started my quest for a "show-off" game about a year ago, searching everywhere for one that was just right. I learned a very interesting fact quickly: My computer doesn't speak BASIC, and to date many games have been written and published in that language.

So I had to do it myself. The result is SHOOTING STARS, a game with enough challenge to intrigue, enough variables to make learning to win difficult (but not impossible), and a couple of goodies thrown in to involve the player with the computer.

A complete program listing for 8008 computer is included, as well as the various messages that allow the computer to interact with the player.

The Game

Nine dot or asterisk characters are arranged in a 3 by 3 matrix on the playing field which may be shown on a CRT screen. The matrix represents the universe; asterisks are stars and dots are black holes. The player shoots stars which die and turn into black holes. When a star dies, it affects other stars and black holes in its particular galaxy.

How To Play

Each position in the universe is assigned a number (see figure 2). The computer outputs the current composition of the universe and asks YOUR SHOT? The player responds by typing the position number of the star he decides to shoot. Then the new constellation is displayed for the next shot.

Effect Of Shooting A Star

When a star dies, it affects the stars and black holes of its particular galaxy. The *effect* is that fragments of the star move into black holes to become new stars and other fragments collide with other stars and knock them out of orbit producing black holes. Each star has its own galaxy as shown in figure 3.

The Program

The game proceeds in an orderly manner which is shown in the Flow Chart of figure 4. The heading, rules and interactive messages require approximately 1600 B of memory. I use a Delta t Digital Recorder for message storage and retrieval since it operates in the reverse as well as forward incremental modes. Each message is prefaced with a message number surrounded with STX and ETX characters. A search routine in the main program finds the first address, decides whether the desired message is ahead or behind the current tape position, and rewinds or spins forward as necessary.

Table 1 is a list of the interactive messages. For computers with limited memory the essential messages are in the first portion of the table; the fancy heading is next, and the rules of the game occupy the largest number of bytes at the end of the text.

When the program is entered at address 014000, the 8008's H and L pointers are set to the beginning of the heading. Then the message control routine is called. It outputs sequentially each character of the message until the EM delimiter is encountered which returns control to the main program.

The status of the universe is stored in the B and C registers. Universe positions 1 through 4 and 6 through 9 are represented by the eight bits in the B register. A one bit represents a star, and a zero a black hole. Bit 0 of the C register keeps track of position 5.

The universe is set up in the beginning by clearing the B register and setting C to 001

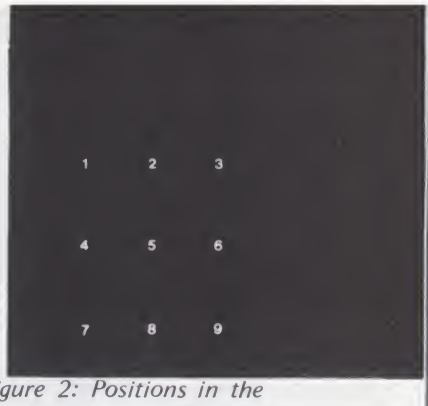
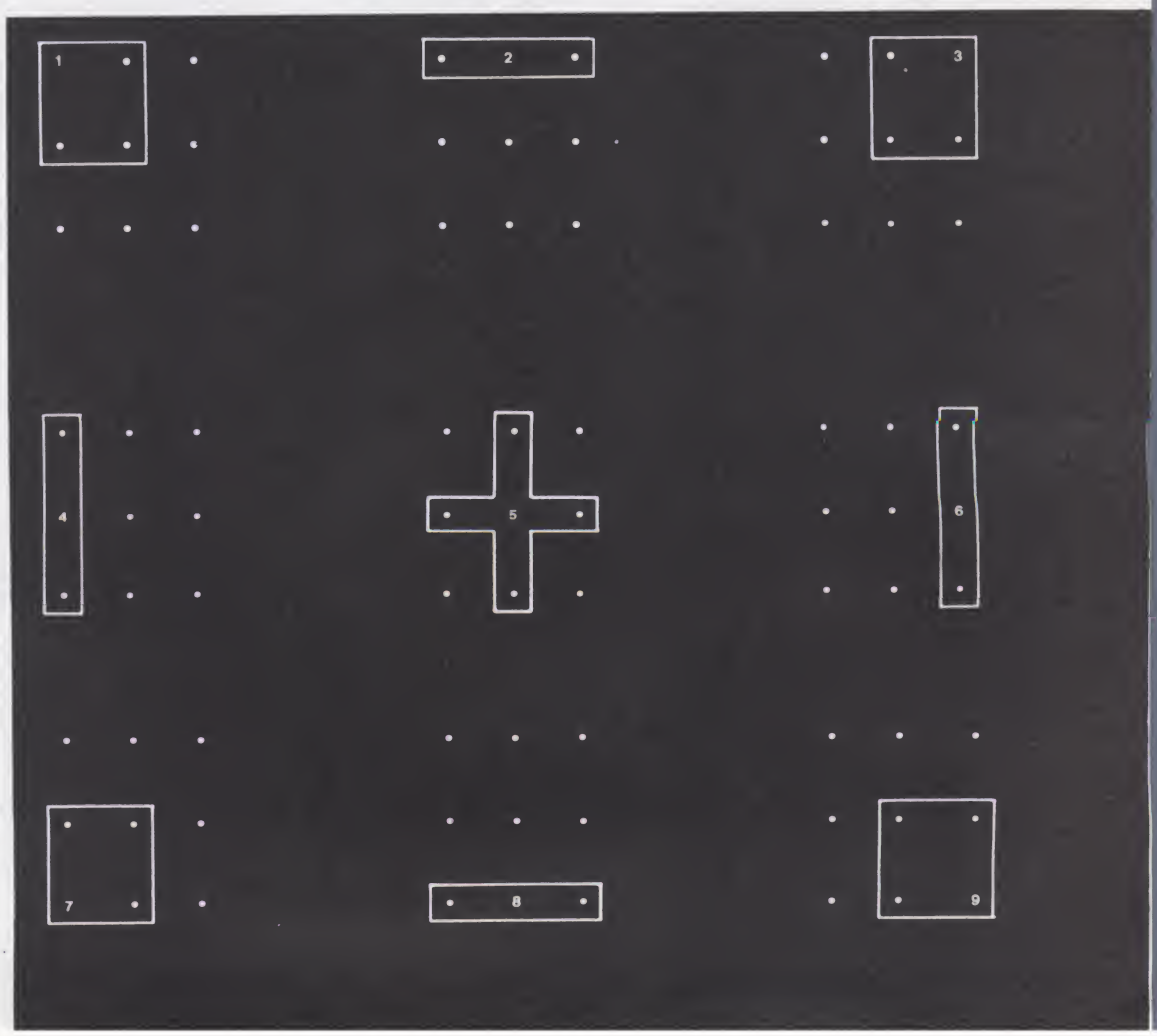


Figure 2: Positions in the universe are identified by numbers.

Figure 3: A complete set of galaxies which are associated with every star or black hole position. Stars or black holes within a galaxy are affected whenever the respective position has been chosen.



octal. The D register, which will tally the number of shots fired, is also cleared as part of the initialization process. Each time the print universe routine is entered after a valid shot, the D register is incremented to count the shot.

Displaying The Universe

First, the print universe routine is entered. This routine sets the E register to octal 012 and will decrement the register each time the print loop is executed. The E register tells the program when it needs to insert a couple of linefeeds for spacing, when it needs to branch to the position 5 special routine, and when it has finished printing the universe. These events occur at the following E register exception counts:

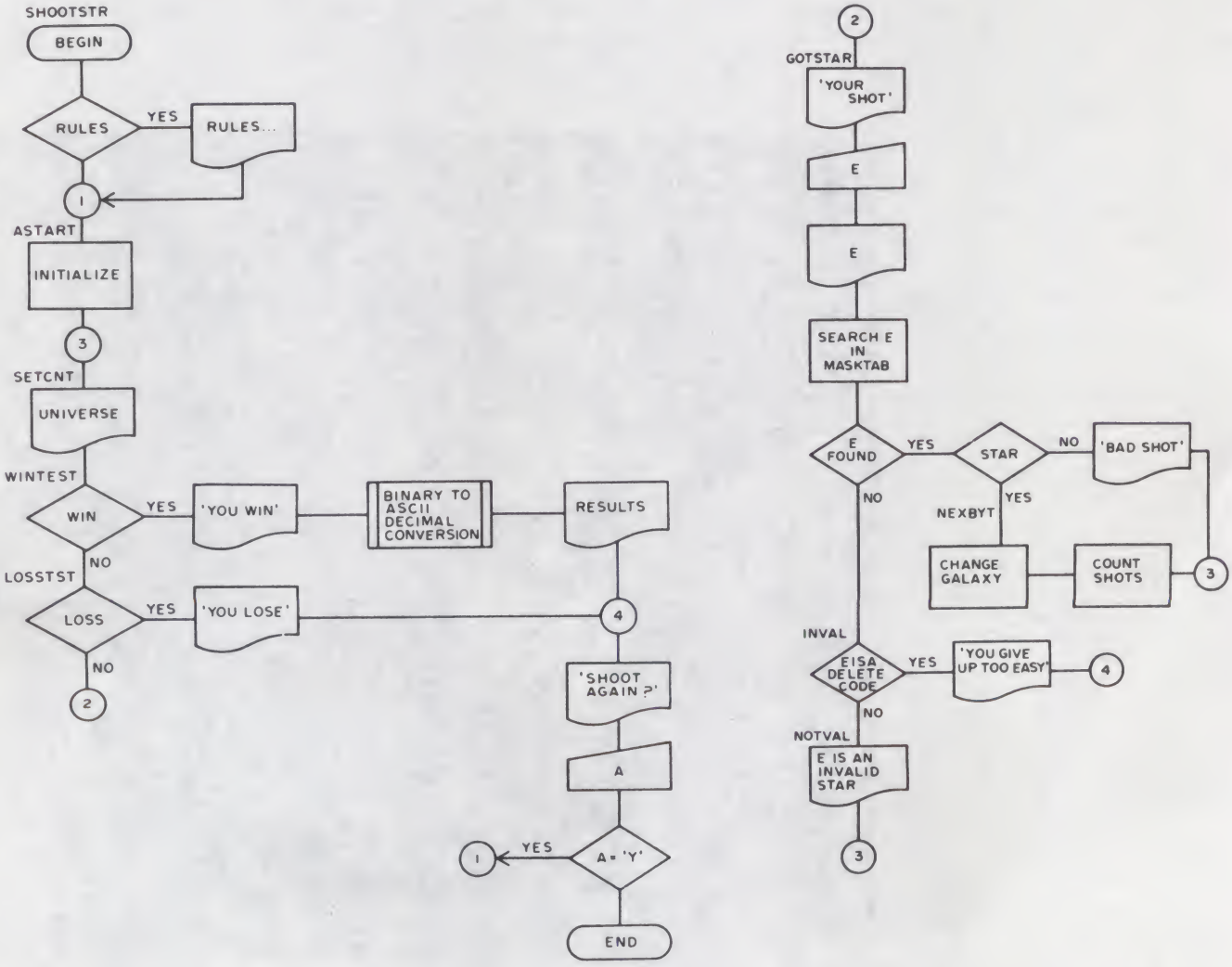
006 – Insert two linefeeds
005 – Go to position 5 subroutine
003 – Insert two linefeeds
000 – Done Print; exit

In normal processing, the positions represented by the bits in the B register are inspected one-by-one for star or black hole status, and the corresponding symbol is printed. It's done like this: The B register is loaded to A and rotated one place to the right. The rotated byte is loaded into B to be ready for the following position next time around in the loop. The carry flag is then tested for a one or zero. If the carry is zero, the program jumps to the dot output section. A one in the carry bit causes the asterisk output to be executed.

At the exception counts, further processing is required.

Thus when the E register count indicates that position 5 is the next one to be printed, the program loads the C register to A and

Figure 4: A flow chart of the SHOOTING STARS program acts as a guide to the listing. The labels indicated on this flow chart correspond to the labels found in table 3.



rotates the least significant bit to carry. The program then jumps back to the asterisk and dot output portion of the loop. Note that the rotated C register content is not loaded again to C, since we are only interested in the least significant bit.

Shoot A Star

When the universe has been displayed, the message YOUR SHOT? is printed and the computer waits for the player to type a number from 1 to 9 which indicates the star he wants to shoot. The ASCII code for the number the player types is compared to the first byte in each group of four contained in the MASKTAB table 2. The number of tries at the table is monitored by the E register, which starts at 011 and is decremented each time around the "test for match" loop. If the E register gets to 000 without finding a match, the input is tested for code 177 (delete), indicating that the player gives up and wants to start over. If a match still can't be found, the NOT A VALID STAR NUMBER message is printed, and the universe displayed again. If this happens, the print universe routine is entered just after the instruction that causes the shot to be counted, so the player won't be charged for his mistake.

When a find is made in the MASKTAB table, the program is ready to process the player's shot. First, it must make sure the player is following the rules and hasn't shot a black hole. The second byte of the four byte group is used as a "mask" to blank out all the positions of the universe except the one that has been shot. Figure 5 shows how the mask is used with the Boolean AND function to isolate the bit representing the shot position from among the eight bits of the B register. After masking out all but the selected position, the resultant byte is tested to see if it is zero. If it is, the shot position was a black hole and the message HEY! YOU CAN ONLY SHOOT STARS, NOT BLACK HOLES! is printed. If this happens, the universe is displayed again without counting the shot.

If the mask itself is zero, it indicates that position 5 was selected, and so the program

Table 1: Program Messages. This table lists all the messages used by SHOOTING STARS. Each message entry in the table starts with a symbolic name and an absolute address. The text should be stored at ascending memory address locations, and terminated with an end of message (EM) delimiter of octal 031, which is printed as ■. The symbolic names in this table are referenced by table 3.

```
MESS1:
    016000
    HEY! YOU CAN ONLY SHOOT STARS,
    NOT BLACK HOLES.
    TRY AGAIN!
    ■

MESS2:
    016077
    THAT WASN'T A VALID STAR NUMBER!
    TRY AGAIN!
    ■

MESS3:
    016156
    YOU LOST THE GAME!
    WANT TO SHOOT SOME MORE STARS?  ■

MESS4:
    016243
    YOU WIN! GOOD SHOOTING!
    YOU FIRED  ■

MESS5:
    016310
     SHOTS.
    BEST POSSIBLE SCORE IS 11 SHOTS.
    WANT TO SHOOT AGAIN, DEADEYE?  ■

MESS6:
    017022
    YOU GIVE UP TOO EASILY!
    WANT TO SHOOT SOME MORE STARS?.  ■

MESS7:
    017114
    YOUR SHOT?  ■

HMESS:
    017131
    S H O    SSS    TTT    AAA    RRR    SSS
             S        T    A A    R R    S
    O • T    SSS      T    AAA    RRR    SSS
               S      T    A A    RR       S
    I N G    SSS      T    A A    R R    SSS
    ••••••••••••SHOOTING STARS••••••••••••
                A BRAIN TEASER GAME
    WANT THE RULES?  ■

PAGE1:
    020147
    THERE ARE STARS:  •
    AND BLACK HOLES:  .
    IN THE UNIVERSE:    . . .
                        . . .
                        . . .

    YOU SHOOT A STAR
    (NOT A BLACK HOLE)
    BY TYPING ITS NUMBER     1 2 3
                             4 5 6
                             7 8 9
    THAT CHANGES THE STAR TO A BLACK HOLE!
    (TO SEE MORE RULES, TYPE ANY KEY.) ■

PAGE2:
    021277
    EACH STAR IS IN A GALAXY. WHEN YOU
    SHOOT A STAR, EVERYTHING IN ITS GALAXY
    CHANGES. ALL STARS BECOME BLACK HOLES
    AND ALL BLACK HOLES BECOME STARS.
    GALAXIES:
    1 • .   • 2 •   . • 3   • . .   . • .
    • • .   . . .   . • •   4 . .   • 5 •
    . . .   . . .   . . .   • . .   . • .
    . . •   . . .   . . .   . . .
    . . 6   • • .   . . .   . • •
    . . •   7 • .   • 8 •   . • 9
    (TYPE ANY KEY FOR LAST PAGE OF RULES.) ■

PAGE3:
    023137
    THE GAME STARTS          . . .
    WITH THE UNIVERSE        . • .
    LIKE THIS                . . .

    YOU WIN WHEN YOU         • • •
    CHANGE IT TO THIS        • . •
                             • • •

    YOU LOSE IF YOU          . . .
    GET THIS                 . . .
                             . . .
    READY TO PLAY. TYPE ANY KEY TO START
    THE GAME. GOOD LUCK! ■
```

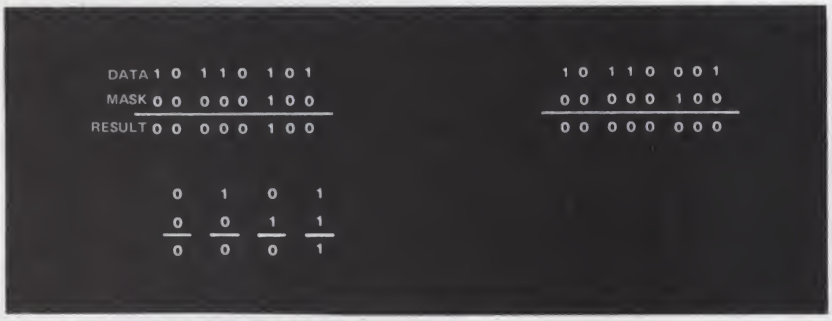


Figure 5: The AND function of Boolean logic is used to mask the current universe in order to select one position for testing each shot.

	LOCATION	SHOT	POSITION MASK	GALAXY MASK	CENTER MASK
MASKTAB	015070	061	001	013	001
	015074	062	002	007	000
	015100	063	004	026	001
	015104	064	010	051	000
	015110	065	000	132	001
	015114	066	020	224	000
	015120	067	040	150	001
	015124	070	100	340	000
	015130	071	200	320	001

Table 2: MASKTAB, a table of masks to test and alter galaxies. This table gives the data needed for memory locations 015/070 to 015/133 in the SHOOTING STARS program. This table is used to check the shot fired for a valid star number and to change the portion of the universe which is affected by the star's change.

Figure 6: The EXCLUSIVE OR function of Boolean logic is used to complement bits selected according to the galaxy information stored for the position just shot.

tests the C instead of the B register for a star.

Change A Galaxy

Once the program has determined that the shot was valid, it can use the next byte in the MASKTAB table to change the dots and stars in the galaxy of the "shot" star. Again, the table entry is a mask, but this time the Boolean EXCLUSIVE OR function is used. The result is that the selected positions are *complemented;* one bits are changed to zero bits and the zeros are changed to ones. Figure 6 shows how the mask does this neat trick. After the change is made, the new universe is stored in the B register.

Byte four of the MASKTAB table entry contains a mask that is used to EXCLUSIVE OR the C register to change position 5 if required. If star 5 is to be complemented, the mask will be octal 001; if not, it will be octal 000.

After the universe in the B and C registers is changed, the new universe is displayed and the cycle repeats until a win or a loss is detected, or until the player gives up.

Win Or Loss Test

Each time the universe is displayed, it is tested for a win or a loss. If both the B and C registers contain the octal number 000, the YOU LOST THE GAME message is printed, and the opportunity to play again is offered.

If the B register contains octal 377 and C is octal 000 a win is detected. After displaying the proper message, the binary content of the D register is converted to decimal numbers and the number of shots fired is printed. The calculation is performed by the binary to decimal conversion subroutine.

Binary To Decimal Conversion

The B, C and E registers are assigned the functions of summing the hundred, ten and unit digits of the score respectively. The process is one of repetitively adding a one to the three digit number while subtracting a one from the shots fired register (D). Looping continues until all shots fired have been counted in the 3 digit decimal form.

The somewhat unusual feature of the binary to decimal conversion is that it is done directly in ASCII numeric code. The three registers B, C and E are intially loaded with octal 060, which is the ASCII numeric character zero. After each increment, the least significant digit register (E) is tested to see if it contains octal 072. If it does, the register has counted 060, 061 ... 071, which is 0 through 9 in ASCII, and has just been incremented one more to 072. When

the register has 072, a carry condition exists. When this condition is detected, the register is reset to 060 and the next register in line (C) is incremented. After incrementing, the second register is tested for a carry in the same manner, and so on. When all the shots have been counted, the registers B, C and E will not only represent the decimal equivalent of the shots fired, but will contain the proper ASCII codes for the decimal digits of the count.

Print The Shots

To suppress leading zeros, the hundreds digit (B) is tested for octal 060. If it contains any other code, the contents of all three registers will be printed. If it contains octal 060, the tens register (C) is similarly tested and the output will be one digit if it is at zero (code 060) and two digits if it is not.

Figure 7 contains a flow chart of the binary to decimal conversion program. You may find use for it in some of your other programs.

Program Listing Conventions

Table 3 contains the complete program as it was implemented in my 8008 system using the SCELBI 8H computer. The listing is in symbolic assembly language with absolute octal address and memory contents.

The 8008 computer has 8 possible restart instructions which are one byte calls to locations in the first portion of memory address space. These are used to access utility subroutines needed by the SHOOTING STARS program. The required restarts are as follows:

RST0: User's input routine, starting at location 000/000 which is used to wait for one character input from the keyboard device.

RST1: Exit Routine, starting at location 000/010. This is a return address to the system monitor for the computer.

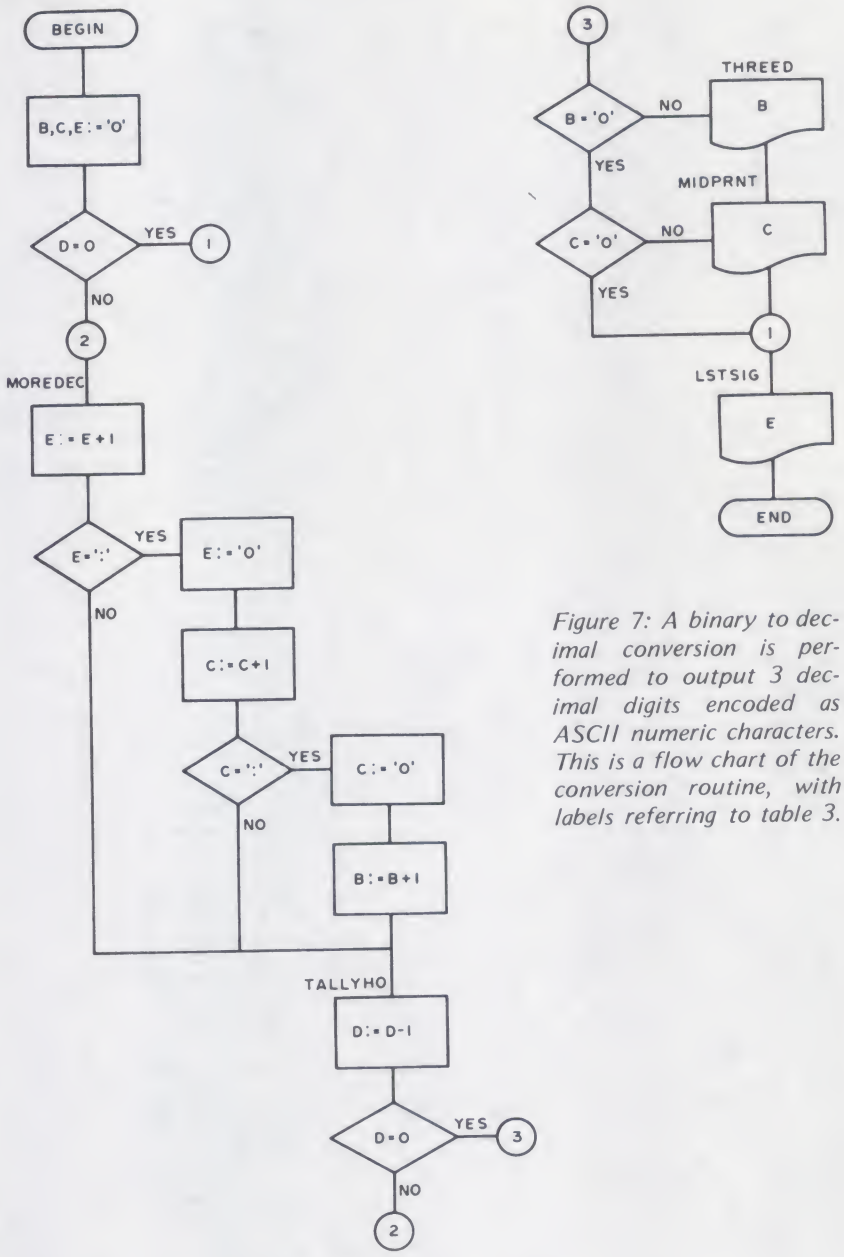


Figure 7: A binary to decimal conversion is performed to output 3 decimal digits encoded as ASCII numeric characters. This is a flow chart of the conversion routine, with labels referring to table 3.

octal address	octal code	label	op.	operand	commentary
014/000	006 012	SHOOTSTR	LAI	012	display a linefeed to
014/002	025		RST	2	initialize display;
014/003	066 131		LLI	L(HMESS)	set address pointers
014/005	056 017		LHI	H(HMESS)	to heading message;
014/007	106 134 015		CAL	OUTPUT	print message & return;
014/012	106 151 015		CAL	INPUT	call input looper;
014/015	074 116		CPI	'N'	is first letter 'N'?
014/017	150 052 014		JTZ	ASTART	if so then plunge into game;
014/022	066 147		LLI	L(PAGE1)	if not then point to first
014/024	056 020		LHI	H(PAGE1)	page of rules text;
014/026	106 134 015		CAL	OUTPUT	and go output rules message;
014/031	075		RST	7	wait for goahead;
014/032	066 277		LLI	L(PAGE2)	point to second page of
014/034	056 021		LHI	H(PAGE2)	rules text;
014/036	106 134 015		CAL	OUTPUT	display second page of rules;
014/041	075		RST	7	wait for goahead;
014/042	066 137		LLI	L(PAGE3)	point to third page of
014/044	056 023		LHI	H(PAGE3)	rules text;
014/046	106 134 015		CAL	OUTPUT	display third page of rules;
014/051	075		RST	7	wait for goahead;
014/052	006 012	ASTART	LAI	012	set up linefeed;
014/054	025		RST	2	display one linefeed,
014/055	025		RST	2	then a second linefeed,
014/056	025		RST	2	then a third;
014/057	016 000		LBI	0	initialize the universe
014/061	026 001		LCI	1	to starting pattern;

Table 3: The SHOOTING STARS program specified in symbolic assembly language with an absolute listing of addresses and codes for the author's system.

```
octal
address     octal code      label     op.     operand     commentary

014/063     331                       LDB                 then clear shot counter;
014/064     030             CNTSHOT   IND                 count a shot (anticipatory);
014/065     046 012         SETCNT    LEI     10D         loop count 10 iterations;
014/067     041             DISLOOP   DCE                 is the loop done?
014/070     150 321 014               JTZ     WINTEST     if so then go to win testing;
014/073     304                       LAE                 if not then continue display;
014/074     074 006                   CPI     6           is it fourth cycle?
014/076     150 142 014               JTZ     LINFEED     if so then new line needed;
014/101     074 003                   CPI     3           is it seventh cycle?
014/103     150 142 014               JTZ     LINFEED     if so then new line needed;
014/106     074 005                   CPI     5           is it star number 5?
014/110     150 151 014               JTZ     FIVTST      if so then go test star 5;
014/113     250             NEDOT     XRA                 clear the carry (and A too);
014/114     301                       LAB                 move universe to A;
014/115     012                       RRC                 rotate next place into carry;
014/116     310                       LBA                 save it in B for a while;
014/117     100 130 014     PSEUDOT   JFC     LOADOT      if dot then go output dot;
014/122     006 052                   LAI     '*'         otherwise load a star;
014/124     025                       RST     2           then print the star;
014/125     104 133 014               JMP     SPCNOW      branch around dot logic;
014/130     006 056         LOADOT    LAI     '.'         load a dot;
014/132     025                       RST     2           then print the dot;
014/133     006 040         SPCNOW    LAI     ' '         load a space;
014/135     025                       RST     2           print one space,
014/136     025                       RST     2               then print a second;
014/137     104 067 014               JMP     DISLOOP     waltz around loop once more;
014/142     006 012         LINFEED   LAI     012         load a line feed;
014/144     025                       RST     2           display a line feed,
014/145     025                       RST     2               then a second one;
014/146     104 113 014               JMP     NEDOT       back to print next dot or star;
014/151     250             FIVTST    XRA                 no operation intended -- leftover;
014/152     302                       LAC                 get position 5 status;
014/153     012                       RRC                 put status into carry;
014/154     104 117 014               JMP     PSEUDOT     rejoin main line after RRC;
014/157     006 012         GOTSTAR   LAI     012         load a line feed;
014/161     025                       RST     2           have finished universe print,
014/162     025                       RST     2               so print several
014/163     025                       RST     2               line feeds
014/164     025                       RST     2               to separate
014/165     025                       RST     2               successive rounds;
014/166     066 114                   LLI     L(MESS7)    point to the 'your shot'
014/170     056 017                   LHI     H(MESS7)        message;
014/172     106 134 015               CAL     OUTPUT      then go print it;
014/175     005                       RST     0           call input for character;
014/176     025                       RST     2           immediately echo the input,
014/177     340                       LEA                 save input temporarily in E;
014/200     006 012                   LAI     012         load a line feed;
014/202     025                       RST     2           print three line feeds to
014/203     025                       RST     2               space out the response
014/204     025                       RST     2               a bit more;
014/205     304                       LAE                 recover input for testing;
014/206     046 011                   LEI     9D          loop count for table search;
014/210     066 070                   LLI     L(MASKTAB)  set up pointer to the
014/212     056 015                   LHI     H(MASKTAB)      the mask table;
014/214     277             NEXGRUP   CPM                 is input equal table character?
014/215     150 233 014               JTZ     FOUND       if so then go alter structure of
014/220     041                       DCE                     the universe otherwise just
014/221     150 273 014               JTZ     INVAL           check end of loop;
014/224     060                       INL                 increment the L
014/225     060                       INL                     register pointer
014/226     060                       INL                     four times to get
014/227     060                       INL                     to next table entry;
014/230     104 214 014               JMP     NEXGRUP     then go test next entry;
014/233     060             FOUND     INL                 point to position mask
014/234     307                       LAM                     and load mask into A;
014/235     074 000                   CPI     0           is it zero?
014/237     110 253 014               JFZ     UNIV2A      if not then fringe position;
014/242     302                       LAC                 otherwise the center position;
014/243     074 001                   CPI     1           is a star in center?
014/245     110 165 015               JFZ     BADFELO     if not then have wrong move;
014/250     104 260 014               JMP     NEXBYT      if so then go process star;
014/253     301             UNIV2A    LAB                 rest of universe to A;
014/254     247                       NDM                 AND with mask to isolate star;
014/255     150 165 015               JTZ     BADFELO     if not star then wrong move;
014/260     060             NEXBYT    INL                 point to the galaxy mask;
014/261     301                       LAB                 fetch universe again;
014/262     257                       XRM                 and complement the universe
014/263     310                       LBA                     on a fine performance;
014/264     060                       INL                 point to center mask;
014/265     302                       LAC                 fetch center of universe;
014/266     257                       XRM                 complement center if required;
014/267     320                       LCA                 save center of universe;
014/270     104 064 014               JMP     CNTSHOT     go display a new universe;
014/273     074 177         INVAL     CPI     177         was invalid shot a 'delete'?
014/275     110 307 014               JFZ     NOTVAL      if not then recycle bad star;
014/300     066 022                   LLI     L(MESS6)    otherwise point to giving up
014/302     056 017                   LHI     H(MESS6)        message;
014/304     104 034 015               JMP     PRNTIT      display then test for restart;
014/307     066 077         NOTVAL    LLI     L(MESS2)    point to the invalid star
014/311     056 016                   LHI     L(MESS2)        number message
014/313     106 134 015     OUTMES    CAL     OUTPUT      output a message then
014/316     104 065 014               JMP     SETCNT          go display the universe again;
014/321     301             WINTEST   LAB                 move universe to A,
014/322     074 377                   CPI     11111111B   are all fringe stars present?
014/324     110 050 015               JFZ     LOSSTST     if not see if player has lost;
014/327     302                       LAC                 fetch center of universe;
014/330     074 000                   CPI     0           is center of universe empty?
014/332     110 157 014               JFZ     GOTSTAR     is full then not win;
014/335     066 243                   LLI     L(MESS4)    no star! got a win, folks
014/337     056 016                   LHI     H(MESS4)        so point to win message;
014/341     106 134 015               CAL     OUTPUT      then display win message;
014/344     046 060                   LEI     '0'         begin binary to decimal conversion
014/346     314                       LBE                     by setting all three working
014/347     324                       LCE                     register to (ASCII) zero;
014/350     031                       DCD                 get rid of last shot;
014/351     303                       LAD                 move shot count to A for test;
014/352     074 000                   CPI     0           test for zero (not needed in
014/354     150 026 015               JTZ     LSTSIG          SHOOTING STARS but generally
                                                              useful with conversions);
014/357     006 072                   LAI     ':'         need compare to ASCII '9' + 1;
014/361     040             MOREDEC   INE                 count up one in 1.s. digit;
014/362     274                       CPE                 is it equal to overflow code?
014/363     110 000 015               JFZ     TALLYHO     if not then tally and continue;
014/366     046 060                   LEI     '0'         else reset 1's digit to zero
014/370     020                       INC                     and carry into next digit;
```

RST2: User's output routine, starting at location 000/020. This routine prints or displays one character on the output device for the system. The character to be output is in the A register when RST2 is entered.

RST7: A "do Nothing" keyboard input acknowledgement routine, starting at location 000/070. Any character typed on the keyboard causes return from this subroutine.

For the optimum use of the program, the output device should be a cathode ray tube terminal with a scrolling feature.

Game Background

I first saw the SHOOTING STARS game in the September, 1974, issue of PCC† as a program called TEASER. If you are an analytical person, you can figure out all of the possible positions.

PCC Editor, Bob Albrecht, told me that the program was contributed to the Hewlett-Packard software library, and originally written in BASIC.■

---

†PCC is People's Computer Company which publishes a tabloid size computer hobbyist newspaper five or more times during the school year. It's filled with games written in BASIC, art, and computer news. If you are interested, write to People's Computer Company, PO Box 310, Menlo Park CA 94025.

Symbol table, in order of appearance

SHOOTSTR	014 000
ASTART	014 052
CNTSHOT	014 064
SETCNT	014 065
DISLOOP	014 066
NEDOT	014 113
PSEUDOT	014 117
LOADOT	014 130
SPCNOW	014/133
LINFEED	014 142
FIVTST	014 151
GOTSTAR	014 157
NEXGRUP	014/214
FOUND	014/233
UNIV2A	014/253
NEXTBYT	014 260
INVAL	014/273
NOTVAL	014 307
OUTMES	014/313
WINTEST	014/321
MOREDEC	014/361
TALLYHO	015 000
THREED	015/023
MIDPRNT	015/025
LSTSIG	015 026
RECYC	015/032
PRNTIT	015/034
LOSSTST	015/050
MASKTAB	015 070
OUTPUT	015/134
INPUT	015/151
GETNEXT	015/154
BADFELO	015/165
MESS1	016/000
MESS2	016/077
MESS3	016/156
MESS4	016/243
MESS5	016/310
MESS6	017 022
MESS7	017 144
HMESS	017/131
PAGE1	020/147
PAGE2	021/277
PAGE3	023 137

```
octal
address   octal code      label      op.    operand     commentary
014/371   272                         CPC                is it equal to overflow code too?
014/372   110 000 015               JFZ    TALLYHO     if not then tally and continue;
014/375   026 060                   LCI    '0'         else reset middle digit to zero
014/377   010                       INB                    and carry into m.s. digit;
015/000   031             TALLYHO   DCD                decrement score counter for tally.
015/001   110 361 014               JFZ    MOREDEC     if not zero then keep loopin;
015/004   301                       LAB                fetch leading digit to A;
015/005   074 060                   CPI    '0'         is it (ASCII) zero?
015/007   110 023 015               JFZ    THREED      if not go display three digits;
015/012   302                       LAC                fetch middle digit to A;
015/013   074 060                   CPI    '0'         is it (ASCII) zero too?
015/015   110 025 015               JFZ    MIDPRNT     if not go display two digits;
015/020   104 026 015               JMP    LSTSIG      if so display only one;
015/023   025             THREED    RST    2           display three digits, left first,
015/024   302                       LAC                fetch middle digit to A;
015/025   025             MIDPRNT   RST    2           display two digits, left first,
015/026   304             LSTSIG    LAE                fetch 1's  digit;
015/027   025                       RST    2           display remaining digit;
015/030   066 310                   LLI    L(MESS5)    point to first part of you win;
015/032   056 016           RECYC   LHI    H(MESS5)    second part of MESS5/MESS6 pointer;
015/034   106 134 015       PRNTIT  CAL    OUTPUT      display the message;
015/037   106 151 015               CAL    INPUT       fetch a character for continue
015/042   074 131                   CPI    'Y'            query, is it "yes"?
015/044   150 052 014               JTZ    ASTART      if so then continue game;
015/047   015                       RST    1           otherwise call EXIT.
015/050   074 000           LOSSTST CPI    0           is fringe universe all black holes?
015/052   110 157 014               JFZ    GOTSTAR     if not then continue game;
015/055   302                       LAC                if so then test center position,
015/056   074 000                   CPI    0           is center also black hole?
015/060   110 157 014               JFZ    GOTSTAR     if not then continue game;
015/063   066 156                   LLI    L(MESS3)    else point to loss message,
015/065   104 032 015               JMP    RECYC       and go print loss,

015/070   see Table II    MASKTAB   BLK    036D        36 bytes of mask table,

015/134   307             OUTPUT    LAM                fetch next message byte,
015/135   074 031                   CPI    031         is it a delimiter?
015/137   053                       RTZ                return when delimiter found,
015/140   025                       RST    2           otherwise display byte;
015/141   060                       INL                point to next byte,
015/142   110 134 015               JFZ    OUTPUT      is it page boundary?
015/145   050                       INH                if so increment page,
015/146   104 134 015               JMP    OUTPUT      and then recycle;

015/151   005             INPUT     RST    0           get next character,
015/152   340                       LEA                save it in E,
015/153   025                       RST    2           echo on display,
015/154   005             GETNEXT   RST    0           get next character,
015/155   025                       RST    2           echo on display,
015/156   074 012                   CPI    012         was it a line feed?
015/160   110 154 015               JFZ    GETNEXT     if not continue scan,
015/163   304                       LAE                if so, restore first input,
015/164   007                       RET                and then return to caller,

015/165   066 000         BADFELO   LLI    L(MESS1)    point to the error message
015/167   056 016                   LHI    H(MESS1)       admonishing bad 'star',
015/171   104 313 014               JMP    OUTMES      and go display error,
```

Notation:

L(HMESS) = low order 8 bits of address of HMESS;

H(HMESS) = high order 8 bits of address of HMESS;

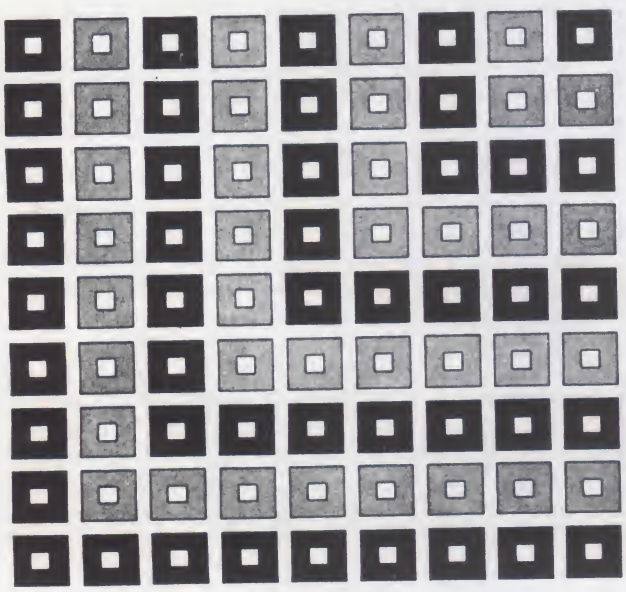
'N' = the ASCII character "N";

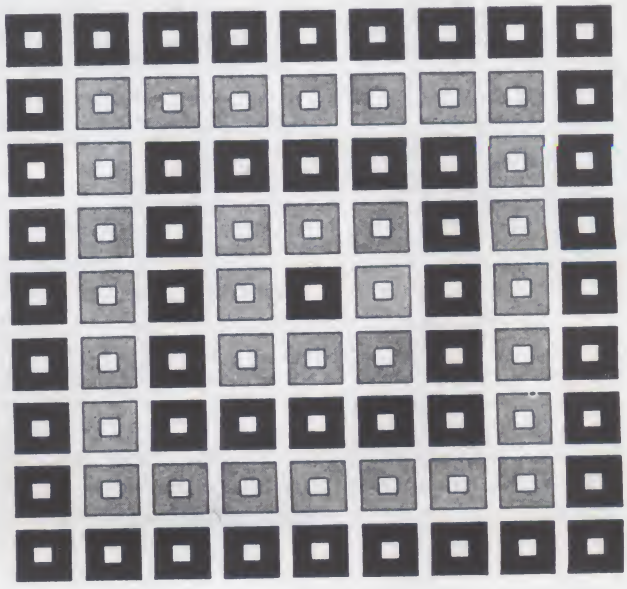
9D = the decimal number 9;

7 = the octal number 7 (with high order zeros as needed);

mnemonics are from original Intel 8008 documentation;

octal code is shown in ascending address order top to bottom, left to right;





Biorhythm for Computers

Joy and Richard Fox
1364 Campbell St
Orlando FL 32806

According to the biorhythm hypothesis, there is a reason for those doldrum days when even your computer refuses to communicate with you.

There is no doubt that all living things have biological rhythms. The study of three of these rhythms in humans has led to the development of a pseudo science, biorhythm, that, through the use of computers, is growing in popularity in the United States. This article describes a program, written in BASIC, which you can run in your own computer to plot biorhythm curves.

The purpose of the program is to use the biorhythm hypothesis to "predict" physical, emotional and intellectual patterns that indicate up, down and critical days for any period of time. These predictions are based on what purport to be scientific studies of human behavior. Biorhythm people claim to have learned through their studies that a physical cycle occurs every 23 days, an emotional cycle occurs every 28 days and an intellectual cycle occurs every 33 days. The plotting of these rhythms is printed out as a two-dimensional graph on a Teletype or similar output device, showing the three cycles as a function of time.

The biorhythm hypothesis is nothing new. It was first proposed in the late nineteenth century by a Viennese psychologist and a German physician, each working separately. In the 1920s, an Austrian teacher added the 33 day intellectual cycle after studying the performance of high school and college students.

According to the biorhythm hypothesis, there is a reason for those doldrum days when even your computer refuses to communicate with you. Each of the three cycles oscillates between ups and downs. When your cycles are up, you feel physically strong, emotionally high or intellectually brilliant. When your cycles are down, you feel physically weak, emotionally depressed or intellectually dull. But the days to really watch out for are the transition days when you are crossing from a low to a high or a high to a low. It is during these transition days that you are especially susceptible to accident and illness. A few times each year, two or even all three of your cycles will cross the transition simultaneously. According to biorhythm people, these critical days are best spent quietly.

The biorhythm hypothesis has gained acceptance in a growing number of industries. In Japan, 2,000 businesses use biorhythm calculations. One Japanese firm reports a 35% reduction in computer data errors by assigning workers to other tasks when they are going through critical days. Another Japanese firm using biorhythm predictions claims to have reduced its yearly vehicle accident loss by 45%. An American survey of 1,000 industrial accidents showed that 90% of them occurred on critical days.

Mike Bertalot, a supervisor for United Airlines, estimates that between 6,000 and 8,000 of United's 40,000 employees are using biorhythm predictions as a guide for safety awareness. United uses the printouts, which they distribute to interested employees, as "an excuse to warn employees about safety." The result has been that some departments have shown a 50% decrease in accidents. It is not clear whether this reduction is due to the extra warnings or to the predictive value of the hypothesis. Although the future of the biorhythm experiment at United is uncertain, the results are being sent to the United States Naval Laboratory, which is studying the hypothesis.

Biorhythms have also been used for profit. The September 15 1975 issue of *Newsweek* quotes Lester Cherubin, president

[NOTE: *The ideas presented in this article are a hypothesis about human mental states and are not necessarily a valid predictive theory. One danger of computer programming is the assumption that a logically correct program which executes without bombing out will necessarily produce meaningful results. Whatever the final conclusion with regard to the biorhythm hypothesis, the calculation makes an interesting example of a BASIC language application program. . . .* CH]

of Time Pattern Research, Inc, as having sold 100,000 biorhythm printouts for $10 to $20 each in the past three years. Other companies sell plastic biorhythm calculating devices for anywhere between $4 and $20. Some shopping center vendors sell for a mere 50 cents a computer printout of your rhythms for one day.

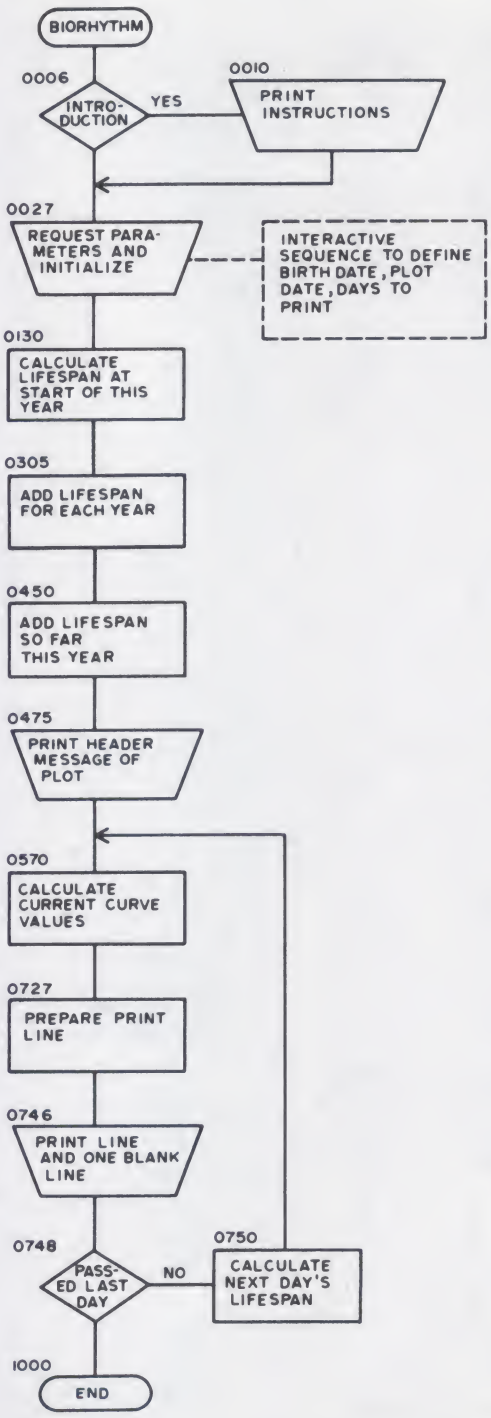
The calculation of biorhythm curves is not easy to do with a pencil and paper. First, the subject's age in days must be calculated. This problem, of course, is complicated by all the peculiarities of the modern calendar. Then you must calculate how many complete 23 day cycles the subject has lived through and how many days he is into the next cycle. (The biorhythm hypothesis makes a simplifying assumption that all cycles originate at birth with zero relative phase.) The same must be done for the 28 and 33 day cycles. The fraction of each cycle is multiplied by two pi radians and the sine of that number is taken to obtain the points of the biorhythm curve for that day. The calculation must be rerun for each succeeding day, and the results plotted on graph paper, in order to obtain the biorhythm curves.

The program to calculate biorhythm curves is shown in the form of a flow chart in figure 1; figure 2 shows the complete listing of this program in BASIC. The operation of the program is as follows:

Line 0001 dimensions the strings N and S and the array T. N will be filled with the character set for the days of the month and S will be filled with the image of each line of the graph, as it is prepared for printing. T will be filled from the data statement at line 0080 with the number of days in each month of the year. The input statement at line 0008 and the if statement at line 0009 together allow the user to skip over the explanatory printout at the beginning of the program and go directly to the calculation which starts at line 0027.

Line 0040 defines the numeric values for the month, day and year that the subject was born. Line 0050 defines the month, day and year for the start of the printout. The year can be supplied as a two digit number ('76) or as a four digit number (1976), but the same format must be used for both the birth date and the printout target date. Line 0065 defines the number of days to be plotted.

D3 in the program is the variable which will contain the age of the subject in days. At line 0130, D3 is initialized to 0. The program will now calculate the number of days between the subject's birth date and the requested plotting date. The calculation is performed in several steps, and at the end



of each step, the value calculated at that step is added to the total in D3.

Next, the program checks if the subject was born in January or February of a leap year. The test for a leap year, at line 0150, is made by dividing the birth year by four and checking for a remainder. Only leap years divide by four with a remainder of zero. If the subject was born in January or February of a leap year, one day is added to the running total, at line 0160. Otherwise, the running total is left at zero.

Watch out for evil omens on transition days.

While not intended to apply to machines, maybe biorhythms can be used to predict computer behavior. Enter the birth date of your computer and predict when your cybernetic monster plans its next bomb out!

Figure 1: Flow Chart of Biorhythm Calculator. This chart illustrates the general outline of the program found in figure 2. The numbers noted next to symbols in the flow chart refer to line numbers of the listing in figure 2.

```
0001    DIM N$(72), S$(72), T(12)
0002    N$="000102030405060708091011121314151617181920212223242526272829303l"

0004    REM BIORHYTHM CREATED BY JOY AND RICHARD FOX
0006    PRINT "DO YOU WISH AN INTRODUCTION TO BIORHYTHM? TYPE 1 FOR YES,".

0007    PRINT "OR 0 FOR NO."
0008    INPUT A
0009    IF A=0 THEN 27
0010    PRINT TAB(25), "BIORHYTHM"
0011    PRINT
0012    PRINT
0013    PRINT
0015    PRINT "THE PURPOSE OF BIORHYTHM IS TO PREDICT A PHYSICAL,"
0016    PRINT "EMOTIONAL AND INTELLECTUAL PATTERN THAT INDICATES YOUR"
0017    PRINT "UP AND DOWN DAYS FOR ANY PERIOD OF TIME. BIORHYTHM CAN"
0018    PRINT "SHOW WHICH DAYS WERE GOOD OR BAD FOR YOU BEGINNING WITH"
0019    PRINT "YOUR BIRTH. IT CAN ALSO SHOW YOU WHICH FUTURE"
0020    PRINT "DAYS WILL BE GOOD OR BAD FOR YOU."
0021    PRINT "THESE PREDICTIONS ARE BASED ON SCIENTIFIC"
0022    PRINT "STUDIES TO DETERMINE WHY ACCIDENTS OCCUR. IT WAS LEARNED"
0023    PRINT "THROUGH THESE STUDIES THAT A PHYSICAL CYCLE OCCURS EVERY"
0024    PRINT "23 DAYS, AN EMOTIONAL CYCLE OCCURS EVERY 28 DAYS, AND AN"
0025    PRINT "INTELLECTUAL CYCLE OCCURS EVERY 33 DAYS."
0026    PRINT
0027    PRINT "PLEASE TYPE YOUR BIRTH DATE USING THE FOLLOWING FORMAT."
0028    PRINT "MM,DD,YY. EXAMPLE: JANUARY 17, 1942 = 01,17,42"
0040    INPUT M, D, Y
0045    PRINT "AT WHAT DATE ARE YOU INTERESTED IN BEGINNING BIORHYTHM?"
0050    INPUT M1, D1, Y1
0060    PRINT "HOW MANY DAYS DO YOU WISH TO HAVE PLOTTED?"
0065    INPUT D2
0080    DATA 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
0110    REM M=MONTH, D=DAY, Y=YEAR
0120    REM D3=TOTAL NUMBER OF DAYS ELAPSED
0130    D3=0
0140    IF M<2 THEN 200
0150    IF INT(Y/4)=(Y/4) THEN 200
0160    D3=1
0200    FOR I=1 TO 12
0210    READ T(I)
0220    REM T=DAYS IN EACH MONTH
0230    NEXT I
0240    D3=T(M)+D+D3
0250    FOR I=M+1 TO 12
0260    D3=T(I)+D3
0270    NEXT I
0280    REM Y3=YEAR COUNTER FROM BIRTH TO DISPLAY
0290    Y3=Y
0299    Y3=Y3+1
0305    IF Y3<=Y1 THEN 400
0310    IF INT(Y3/4)=(Y3/4) THEN 320
0315    D3=D3+365
0316    GOTO 299
0320    D3=D3+366
0325    GOTO 299
0400    IF M1<=2 THEN 450
0405    IF INT(Y1/4)=(Y1/4) THEN 450
0410    D3=D3+1
0450    FOR I=1 TO M1-1
0455    D3=T(I)+D3
0460    NEXT I
0470    D3=D1+D3
0475    PRINT "PHYSICAL CYCLE = P"
0480    PRINT "EMOTIONAL CYCLE = E"
0490    PRINT "INTELLECTUAL CYCLE = I"
0491    PRINT
0492    PRINT
0493    PRINT
0500    PRINT "DATE";
0505    PRINT TAB(13), "DOWN",
0510    PRINT TAB(35), "CRITICAL",
0520    PRINT TAB(63), "UP"
0525    PRINT ".....  .....  .....  .....";
0526    PRINT "....."
0530    LET M4=M1
0540    LET D4=D1
0550    LET Y4=Y1
0560    REM M4,D4,Y4 = DATE PRINTED OUT IN PLOTTING CHART
0570    GOTO 580
0571    REM F=FRACTION INTO CYCLE
0580    F=(D3/23)-INT(D3/23)
0600    REM X=THE ARGUMENT FOR THE SINE FUNCTION
0610    X=F*2*3.1416
0640    REM P=THE PHYSICAL POSITION ON THE GRAPH
0650    P=((SIN(X)+1)*24)+15
0655    REM E=EMOTIONAL POSITION ON THE GRAPH
0660    F=(D3/28)-INT(D3/28)
0670    X=F*2*3.1416
0680    E=((SIN(X)+1)*24)+15
0690    F=(D3/33)-INT(D3/33)
0700    X=F*2*3.1416
0710    REM I=INTELLECTUAL POSITION ON THE GRAPH
0720    I=((SIN(X)+1)*24)+15
0727    FOR X=1 TO 32
0728    S$(2*X-1,2*X)="  "
0729    NEXT X
0731    S$(39,39)=" "
0732    S$(P,P)="P"
0733    S$(E,E)="E"
0734    S$(I,I)="I"
0735    S$(3,3)="/"
0736    S$(6,6)="/"
0741    S$(1,2)=N$(M4*2+1,M4*2+2)
0742    S$(4,5)=N$(D4*2+1,D4*2+2)
0743    IF Y4>99 THEN 950
0744    S$(7,7)=N$((INT(Y4/10)*2+2),(INT(Y4-10)*2+2))
0745    S$(8,8)=N$((Y4-INT(Y4/10)*10)*2+2,(Y4-INT(Y4/10)*10)*2+2)
0746    PRINT S$(1,63)
0747    PRINT
0748    IF D2=1 THEN 1000
0750    D2=D2-1
0800    D3=D3+1
0810    D4=D4+1
0815    IF M4=2 THEN 820
0816    IF D4=29 THEN 820
0817    IF INT(Y4/4)=(Y4/4) THEN 820
0818    GOTO 570
0820    IF D4=T(M4) THEN 570
0830    M4=M4+1
0835    D4=1
0840    IF M4=12 THEN 870
0850    GOTO 570
0870    M4=1
0880    Y4=Y4+1
0900    GOTO 570
0950    Y4=Y4-(INT(Y4/100)*100)
0951    GOTO 744
1000    END
```

Figure 2: BASIC Program of the Biorhythm Calculator. This is the complete listing of a BASIC program to perform calculations and plot the results on a hard copy printer.

Lines 0200 through 0230 fill the array T with the values in data statement 0080 so that the array contains the number of days in each month of the year. Line 0240 calculates the number of days from the subject's birth date to the end of his first calendar month, and adds that number to the running total in D3. Lines 0250 through 0270 calculate the number of days in each month during the remainder of the subject's birth year, and add that number to the running total.

The birth year, Y, is transferred to the year counter Y3, and the year counter is incremented at line 0299. If the year counter is greater than or equal to the year to be printed out, Y1, then the program jumps to line 0400. Otherwise, the program adds 365 or 366 to the running total for each year between birth and the target year. Each time that is done, the year counter is incremented. When it matches the printout target year, the program jumps to line 0400.

Next the program calculates the number of days between the start of the display year and the display day. If the display month is March or later, then the program checks if the display year is a leap year. If it is, one day is added to the running total at line 0410. Lines 0450 through 0460 add the number of days in each month between the start of the display year and the display month to the running total D3. Line 0470 adds the number of days into the display month to the running total. D3 now contains the age of the subject in days, as of the requested display date.

Lines 0475 through 0526 print the header of the graph. Lines 0530 through 0571 set up three new variables, M4, D4, and Y4, which will contain each consecutive date as it is printed out.

Now the program calculates the phase of each of the three biological cycles for the subject for the dates requested. The physical cycle has a period of 23 days. If you divide the age of the subject in days by 23, the remainder is a number between 0 and 22.9. That remainder is proportional to the phase of the subject's physical cycle at the requested date. The remainder is stored in variable F at statement 0580. F is then multiplied by two pi radians and the answer is stored in X. X is therefore a number between zero and two pi and is proportional to the phase of the subject's physical cycle. Line 0650 takes the sine of X. The result is a value between +1 and −1. This number is then normalized to a value between 15 and 63 and is stored in P. The values 15 and 63 represent the beginning and ending column numbers of the graph on the Teletype.

Extreme down days will plot in column 15. Extreme up days will plot in column 63. Critical days will plot in column 39, and other days will plot in between these points.

The same calculation is then repeated at lines 0660 through 0680, with a period of 28 days, for the emotional cycle; and at lines 0690 through 0720, with a period of 33 days, for the intellectual cycle. Lines 0727 through 0729 loop to fill up the string S with blank characters, to wipe out old data still in the string. Line 0731 places a dot character in element 39 of the string, so that the zero crossing will be clearly marked by a string of dots down the 39th column of the page. Line 0732 stores the character "P" into the column calculated by the equation for the physical cycle. Lines 0733 and 0734 do the same for the characters "E" and "I". Next the program places slashes in elements three and six of the string S, so that they will print out as slashes in the date at the left of the graph.

The month is placed in array elements one and two and the day is placed in elements four and five. If the operator typed the year as a four digit number, the program truncates the most significant two digits. Line 0744 places the ten's digit of the year into element seven of the string and line 0745 puts the unit's digit of the year into element eight.

The string S is now ready for printing. Line 0746 prints elements one through 63 across the output device page as a month, day, year, a dot at column 39 and the letters "P", "E" and "I" in appropriate positions. Line 0747 causes the typewriter to double space so the graph is easier to read.

If the number of days left to print, D2, has been reduced to one, then the program exits. Otherwise, D2 is decremented by one, and the age of the subject in days is incremented by one.

The date in the month, D4, is incremented and the program checks if the day to be plotted is February 29 of leap year. If it is, the next day's data is plotted. If it is not February 29 of a leap year, then the number of days in the month is checked against the maximum number of days in that month as defined in table T. If the day in the month, D4, is too large, it is reset to one and the month is incremented. If the month has been incremented to 13, it is reset to one and the year is incremented. The program prints the next day's data and keeps looping till all the requested data has been printed.

This program has an unusual application that you may not yet have considered: enter the birth date of your computer, and predict when your cybernetic monster plans its next bomb out!!!■

```
DO YOU WISH AN INTRODUCTION TO BIORHYTHM? TYPE 1 FOR YES,OR 0 FOR NO.
?1
                        BIORHYTHM

THE PURPOSE OF BIORHYTHM IS TO PREDICT A PHYSICAL,
EMOTIONAL AND INTELLECTUAL PATTERN THAT INDICATES YOUR
UP AND DOWN DAYS FOR ANY PERIOD OF TIME.  BIORHYTHM CAN
SHOW WHICH DAYS WERE GOOD OR BAD FOR YOU BEGINNING WITH
YOUR BIRTH. IT CAN ALSO SHOW YOU WHICH FUTURE
DAYS WILL BE GOOD OR BAD FOR YOU.
THESE PREDICTIONS ARE BASED ON SCIENTIFIC
STUDIES TO DETERMINE WHY ACCIDENTS OCCUR.  IT WAS LEARNED
THROUGH THESE STUDIES THAT A PHYSICAL CYCLE OCCURS EVERY
23 DAYS, AN EMOTIONAL CYCLE OCCURS EVERY 28 DAYS, AND AN
INTELLECTUAL CYCLE OCCURS EVERY 33 DAYS.

PLEASE TYPE YOUR BIRTH DATE USING THE FOLLOWING FORMAT:
MM,DD,YY. EXAMPLE: JANUARY 17, 1942 = 01,17,42
?01,17,42
AT WHAT DATE ARE YOU INTERESTED IN BEGINNING BIORHYTHM?
?11,25,75
HOW MANY DAYS DO YOU WISH TO HAVE PLOTTED?
?40
PHYSICAL CYCLE = P
EMOTIONAL CYCLE = E
INTELLECTUAL CYCLE = I

DATE          DOWN                  CRITICAL                        UP
------  ----------------------------------------------------------------
11/25/75       I      PE                .
11/26/75      I   PE                   .
11/27/75      IPE                      .
11/28/75      I                        .
11/29/75      E I                      .
11/30/75      E P I                    .
12/01/75        E  PI                  .
12/02/75          E    I               .
12/03/75             E    I P          .
12/04/75                  E    I       P
12/05/75                       E    I     P
12/06/75                            E  I       P
12/07/75                            .   E  I        P
12/08/75                            .        E I        P
12/09/75                            .             E I      P
12/10/75                            .                  EI   P
12/11/75                            .                     IP
12/12/75                            .                   P   I
12/13/75                            .                 P      IE
12/14/75                            .          P            I
12/15/75                            .  P                   EI
12/16/75                         P  .                  E I
12/17/75                    P       .              E   I
12/18/75                 P          .          E  I
12/19/75              P             .    E     I
12/20/75           P                E.     I
12/21/75        P                E     . I
12/22/75        P           E       I  .
12/23/75          P      E       I     .
12/24/75             EP      I         .
12/25/75          E      I P           .
12/26/75       E    I           P      .
12/27/75       E  I                    P
12/28/75       EI                      .    P
12/29/75       I E                     .         P
12/30/75       I    E                  .              P
12/31/75       I         E             .                   P
01/01/76          I           E        .                     P
01/02/76             I              E  .                      P
01/03/76                I              E                       P
```

Figure 3: Output of the Biorhythm Calculator. Here is a listing of the output of the program found in figure 2. In this case, the introductory text was printed prior to entering the parameter definition sequence.

LIFE *Line*

by
Carl Helmers
Editor, BYTE

Games played with computer equipment are applications of value above and beyond the momentary "hack" value of putting together an interesting program. The creation of a game is one of the best ways to learn about the art and technique of programming with real hardware and software systems. LIFE Line concerns a game — the Game of LIFE, originated by Charles Conway and first publicized by Martin Gardner in *Scientific American*. The Game of LIFE serves as the central theme of LIFE Line — a well defined application of the type of hardware and software which is within the reach of BYTE readers. The description of the LIFE application is the "down to earth" goal of LIFE Line. However, I have an ulterior motive as well — LIFE Line is a very convenient and practical vehicle for teaching ideas about program and system design which you can apply for your own use. Even if you never implement a graphics output device and interactive input keyboards, you can gain knowledge and improve your skills by reading and reflecting upon the points to be made in LIFE Line. The LIFE application also has the side benefit of illustrating some techniques of interactive visual graphics which can be used much more generally.

The Starting Point

In developing a system, it always helps to know *what you want to do!* The ability to pin down a goal for a programming effort — indeed, any effort you make — is one of the most important tools of thought you have available (or can develop) in your personal "bag of tricks." Goal setting does not necessarily mean a complete and detailed description of the result — the feedback from the process of reaching the goal can often modify the details. Goal setting means the setting of a standard in your mind — and on paper — of what you want to accomplish. This standard is used to evaluate and choose among alternatives in a methodical approach to a system which meets that standard.

How to Get From Here to There

The goal of LIFE Line is a hardware/software system which enables the home brew computer builder such as you or me (the "byter") to automate the game of LIFE using relatively inexpensive equipment. It's appropriate here to give a preliminary road map of the course LIFE Line will take, as an illustration of the first steps in the development of a complicated system . . .

1. The facts of LIFE. Defining the rules of the game and its logical requirements always helps — after all, I would not want to confuse it with chess, poker or space war!

2. What do I need to implement LIFE? Once I know the rules, my next problem is to sketch the hardware and software requirements for a reasonable implementation.

3. Programming. Given the necessary hardware, the biggest lump of effort is the process of programming the application. Some parts of this lump include . . .

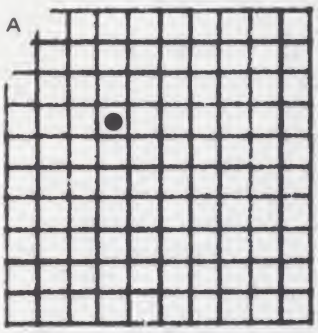
—*Control flow:* Outlining the major pieces of the program and their relationships.

—*Partitioning:* A well designed system is simple! But how can the desired simplicity be reconciled with "doing a lot." One way is to partition the system into pieces. Within each piece, a further partition provides a set of sub-pieces and so on. Each piece of the program is thus kept at a level of relative simplicity, yet the whole system adds up to a quite sophisticated set of functions.

—*Coding:* With the application design laid out in some detail, the program must be coded and debugged for a particular computer. The result could be a series of octal or hexadecimal numbers for your own computer, or a high level language program which can be translated by an appropriate compiler.

Fig. 1. Three views of LIFE: (a) on paper; (b) in memory; (c) on a display.

A



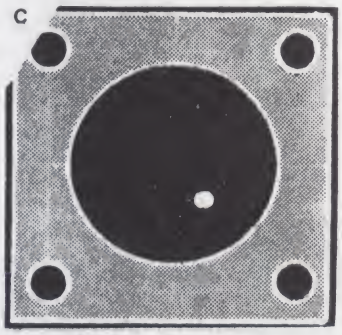
A live "cell" is a dot on paper.

B

```
0000000000000
0000000000000
0000000000000
0000000000000
0000000000010000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

A live "cell" is a "1" bit in memory.

C



A live "cell" is a point of light on a graphics display.

What Are The Facts of LIFE?

Ask a biologist the question "What are the facts of life?" and you will get one answer; ask a "byter" and you'll get the "real" answer — an evolution algorithm used to generate the placement and "cell" content of a square grid given the previous state of cells in the grid. The inspiration of the game is a combination of modern biology, the concept of "cellular automata" in computer science and the pure fun of mathematical abstractions. In making a computer version of the game, the simplest approach is to think of a group of individual "bits" in the computer memory — with your thoughts assigning one memory bit to each "square" of the grid. (The hand operated form of the game algorithm uses graph paper for the squares in question.) If I have a place in memory which can store one bit, it can have a value of logical "zero" or logical "one".

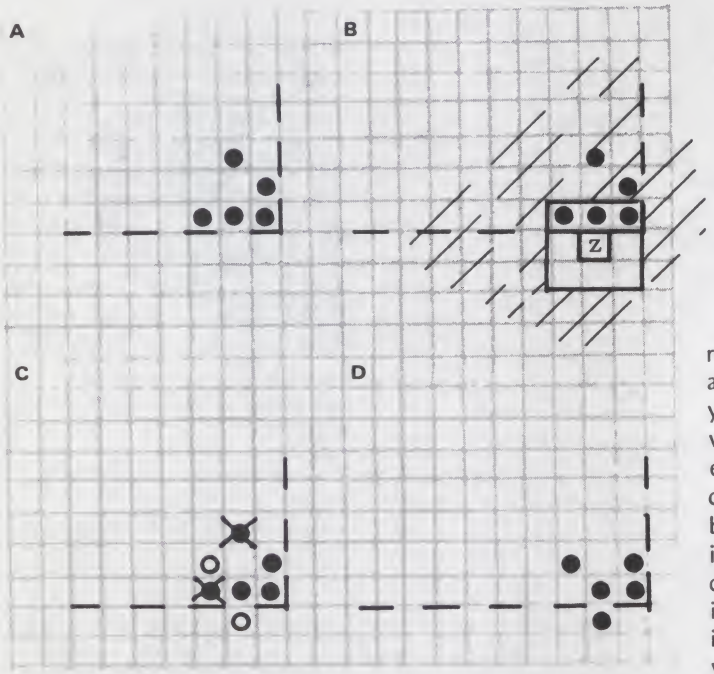
The LIFE game treats each location of the grid (its "squares") as a place where a "cell" might live. If the place is empty, a logical "0" value will be used in the computer memory; if the place is occupied, the "cell" will be indicated by a logical "1" value. The rules of the LIFE algorithm are defined in terms of this idea of a "cell" (logic 1) or "no cell" (logic 0) at every point in the universe of the grid. Fig. 1(a) illustrates a single live cell on a section of graph paper as I might record it when I work out the LIFE process by hand. Fig. 1(b) shows a similar section of the computer memory in which bits ("0" mostly, but "1" for the cell) stand for the content or lack of content of a square on the grid. Fig. 1(c) shows a third view — the output of a program which puts the computer memory bits of the grid onto a graphics display.

Look again at Fig. 1(a). The "cell" on the graph paper grid is a black dot placed in some location. Count the number of graph paper squares which directly surround the live "cell" location. There are 8 possible places which are "nearest neighbors" to the place held by the live cell. Similarly, if you pick an arbitrary square on the graph paper, you can count up its nearest neighbors and find 8 of them also. The rules of the LIFE algorithm concern how to determine whether to place a "cell" in a particular square of the grid for the "next generation", given the present content of that square and its 8 nearest neighbors.

What are the properties of a specific grid location of the game? I've already mentioned its binary valued nature (it has a "cell" or it doesn't) and its neighbors. One more property which is crucial to the game of LIFE is that of the "state" of its 8 nearest neighbor squares. For LIFE, the "state" of the neighbors of a grid location is defined as "the number of occupied neighbors." In the examples of Fig. 1, the "state" of the grid location with the live cell is thus "0" (no neighboring cells), and the state of any cell location which touches the single live cell's location is "1". If I were to fill the entire graph paper or its memory equivalent with live cells, the state of any grid location in the middle would be "8".

Stated in words, the rules of the LIFE algorithm determine the content of each grid location in the "next generation" in terms of its present content and the state of its nearest neighbor grid locations. The rules divide into two groups depending upon the present content of the grid location whose "next generation" value is to be calculated:

Fig. 2. (a) A "glider" generation #n. (b) Examining location "Z" and its nearest neighbors. (c) What has to change for generation #n+1. (d) The second phase of the glider (generation #n+1).



Rule 1. *LIVE CELL LOCATIONS. If* the location to be evolved has a live "cell" at present ("this generation") then,

1.1 *Starving for Affection. If* the location to be evolved has a state of 0 or 1, there will be no cell at the location in the next generation. Metaphorically, if the cell has only one or no nearest neighbors it will die out for lack of interaction with other members of its species.

1.2 *Status Quo. If* the location to be evolved has a state of 2 or 3, the present live cell will live into the tomorrow of the next generation.

1.3 *Overpopulation. If* the location to be evolved has a state of 4 thru 8, there will be no cell at the location in the next generation. Metaphorically, the cell has been crowded out by overpopulation on a local basis.

Rule 2. *EMPTY LOCATIONS. If* the location to be evolved has no live "cell" at present ("this generation") then,

2.1 *The Sex Life of Cells. If* the location to be evolved has a state of 3, a new cell will be "born" in the formerly empty location for the "next generation." Metaphorically, the three neighboring "parent" cells have decided it is time to have a child.

2.2 *Emptiness. If* the location to be evolved does not have three cells in neighboring locations, it will remain empty.

This is the simplest set of rules for the LIFE algorithm, a version which will allow you to begin experimenting with patterns and the evolution of patterns. More complicated extensions can be made to provide an actual interactive (two people) competitive game version; an interesting variation I once implemented is a LIFE game with "genetics." In the genetics variation, each grid location (graph paper square) is represented in the computer as a "character" — an 8 bit byte — of memory. The character in the square is the "gene" pattern of that cell. Then, when rule 2.1 is implemented, LIFE with genetics uses a set of genetic evolution rules to determine which character will be put in the newborn cell based upon the "genes" of the parents. (This genetic evolution program for LIFE was written for my associates at Intermetrics, Inc., as a test program to try out a new compiler's output.)

How Do You Use The Facts of Life?

To illustrate the facts of LIFE, a hand-worked example is a valuable tool of understanding. Consider a "typical" pattern of LIFE as shown in Fig. 2(a). Fig. 2(a) shows what LIFE addicts call a "glider" for reasons which will become clear a little bit later in this article. The glider pattern of Fig. 2(a) consists of the five cells indicated by black dots, and their positions relative to one another. I have also indicated a dotted line in all the illustrations of Figs. 2 and 3 as a fixed reference point in the grid.

The algorithm for evolving one generation to the next is illustrated for one grid location in Fig. 2(b). The LIFE program will examine each location in the grid one by one. This examination is used to figure out what the content of the cell will be in the next generation according to the facts of LIFE. Since these facts only require knowledge of the given grid location Z and its 8 nearest neighbor locations, Fig. 2(b) depicts a box of 9 squares including Z. The rest of the universe is shown shaded. To determine what grid-space location Z will be like in the next generation, the LIFE program first counts up the live cells in all the nearest-neighbor positions. The count is the "state" of Z. In this case there are 3 live cells on the top edge of the box containing Z. Then, the program chooses which rule to use depending upon whether or not location Z has a cell. In this case, Z is empty so the "empty location" set of rules (numbers 2.1 or 2.2) is used. Since the state of Z is 3, rule 2.1 applies and a cell will be born in location Z for the next generation.

Now if I had a true "cellular automaton" to implement the LIFE program, all grid locations would be evolved "simultaneously" — and very quickly — in the computation of the next generation. In point of fact, however, I have

a computer which can only handle 8 (or 16) bits at a time which are stored in words of memory. For small microcomputers, these bits for the LIFE grid will be stored as "packed" bit strings and will be accessed by a series of subroutines which will be described in LIFE Line when the time comes. I have to sequentially look at every bit of the internal LIFE grid of the program and examine its *old* nearest neighbors in order to calculate its new value. I emphasize *old* for the following reason: if I store the new value of the grid location just evolved back into that location with no provision to recall its old value, I'll end up with a mixture of old and new data when I look at the next grid location in the row. That mixture is not part of the rules and constitutes a "faulty" program for evolution. It turns out to be sufficient to remember all the data in one previous row before it was changed in order to calculate the next row after the change. Similar problems of keeping track of partially updated data often occur in computer programming, to be solved by the identical technique of temporarily remembering a copy of the un-updated data.

In Fig. 2(c), the result of examining all the grid locations in the vicinity of the glider of Fig. 2(a) is illustrated. The changes are indicated by three notations for cells:

○ – this indicates a new cell generated by rule 2.1

⊗ – this indicates an old cell which dies by rules 1.1 or 1.3

● – this indicates an old cell which is retained by rule 1.2

Generation "n+1" of the grid of LIFE is illustrated in Fig. 2(d), which was obtained by "executing" the changes noted in Fig. 2(c). When the LIFE program is run, all this is done automatically for each point in the grid – resulting in a new generation as soon as the computer can complete all the calculations. The patterns will be seen to "evolve" in real time as new generations are calculated and sent to the scope output. One "dot" on the scope display corresponds to each live cell of the grid pattern. Fig. 3, (a), (b) and (c), continue the pattern evolution illustrated in Fig. 2 for the "glider". In Fig. 3(a), changes to generation n+1 are indicated with the same notation as was used in Fig. 2(c). The resulting generation n+2 pattern is shown at the right. Fig. 3(b) shows the changes from generation n+2 to generation n+3, and 3(c) shows the change going to generation n+4.

One of the most interesting features of the LIFE game is the evolution of patterns which "move" across a graphics display device. With a fast enough processor, a glider such as the one used in this example will "glide" to the lower right of the screen at a breakneck speed, going off into limbo at the edge – or if the program is sufficiently "smart", reappearing elsewhere on the screen due to a "wrap-around". The reason that the glider gets its name is because of its motion attributes. Note now the fourth generation ("n+4") in the sequence repeats the original glider pattern, *but* has moved one unit along a diagonal of the LIFE grid toward the lower right. (The reference line shows this movement.) It took four generations for the glider pattern to regenerate its original form, which defines the "period" of this pattern. When you get your graphics interface up and running, you will find numerous other classes of patterns, some of which have periods which run into hundreds of generations. There are also other forms of moving patterns similar to the glider.

What Do I Need to Implement LIFE?

The fun part of LIFE is to experiment with patterns of cells and observe how the evolution from generation to generation changes with patterns and classes of patterns. In the lexicon of LIFE lovers, there are whole classes of "gliders", "space ships", "blocks", the "blinkers", "beehives", the "PI" and other patterns. You'll be able to set up initial configurations of these and other patterns, and observe the course of evolution using the hardware/software system

Fig. 3. (a) Third phase of the glider. (b) Fourth phase of the glider. (c) Back to the first phase, but displaced!

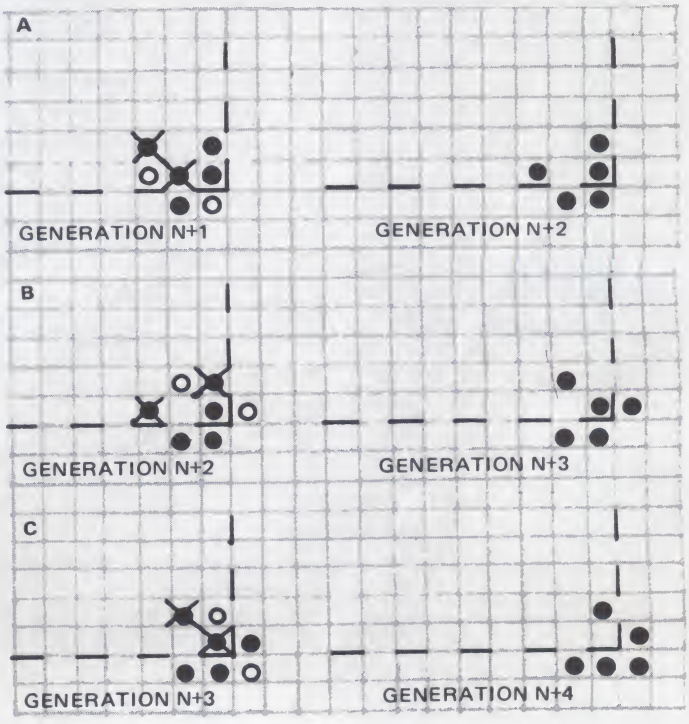
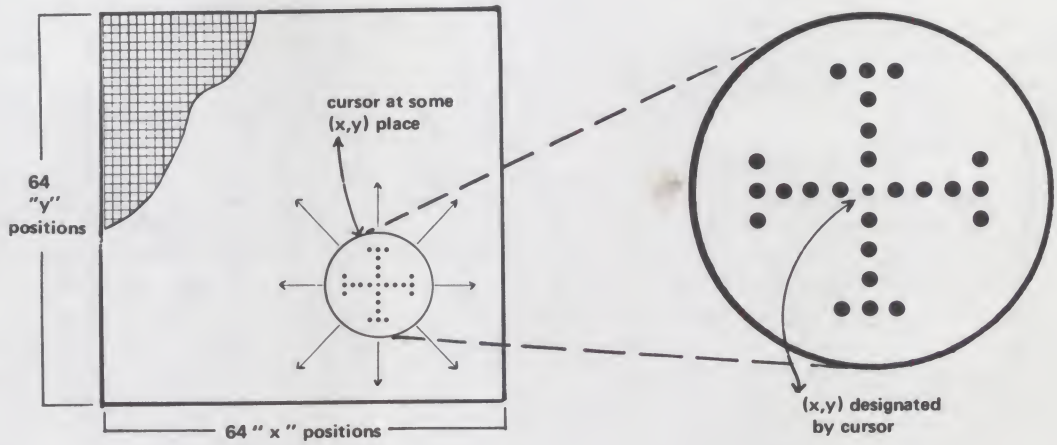


Fig. 4. The LIFE grid display with cursor detail (showing suggested pattern).



concepts of LIFE Line. The hardware requirements of this application's first simple form are three:

1. *An input method.* The best all around input you can get for your computer is an ASCII encoded typewriter keyboard. This hardware will be assumed, with 7-bit ASCII codes used in the examples of programs. If you feel like embellishing the program with special hardware, a "paddle" with several keys can be wired in parallel with your main keyboard to control the special functions of the LIFE program. The input keys used to control the display will require a keyboard which can detect two simultaneous (or three) keys being pressed. A normal ASCII encoded keyboard with an LSI encoding chip will not work "as is" in this application since pressing two keys (other than *control* or *shift* and one other) will be resolved into two characters. An alternate "paddle" type of arrangement is to use a single input port with one switch key switch for each bit of the port, debounced by software. A keyboard which is encoded by a diode matrix can be used since the diode matrix will give a new code (logical sum) based upon which keys were depressed.

2. *A processor.* The game can be implemented on any conventional computer. As a measure of capacity, however, the simple form will assume a 64x64 bit array for the playing field, and an available home brew processor such as an Intel 8080 (i.e.: Altair), Motorola 6800, or National PACE. The total programming capacity of your memory should be roughly 4000 8-bit words, or 2000 16-bit words; the playing field will require 512 8-bit words, or 256 16-bit words — and programming will include a set of subroutines to access individual bits.

3. *A display.* My first version of LIFE was implemented on a PDP-6 in FORTRAN at the University of Rochester when I was a student. That program used a direct link out to a DEC Scope controlled by a PDP-8 — with a teletype for input. I have since implemented life programs using character-oriented terminal output and line printers.

The display to be used for LIFE Line purposes I'll leave undefined in detail, but with the following characteristics: It should have an X-Y selection of coordinates for display elements (LIFE grid locations), which can be individually controlled. Its size will be assumed 64x64.

A Note Regarding Speed

The LIFE algorithm to be illustrated in LIFE Line is optimized fairly well for speed — a requirement which will become obvious in the context of your own system if you use a typical microprocessor. With a fairly large pattern of cells, it may take as much as a minute or more to compute the next generation. Trading off against speed is memory size — use of a packed bit structure is necessary if the matrix and programs are to fit in a micro computer which is inexpensive. But the packed bit structure requires time to access bits (eg: the shift/rotate instructions several times might be used in the access process). I predict that the program will be "dreadfully slow" if run on an 8008, and perhaps passably quick if you use a 6800 or 8080. ("Passably quick" means under 10 seconds per generation.) A used third-generation mini (high speed TTL) would be ideal.

User Features

No application is complete without taking into consideration the *user* of the system. The interface which controls the system is an important section of the design. There is a temptation on the part of individuals such as you or I to say words to the effect: "Since I am making it for me, who the heck cares about the user interface." But! Removing the system from the working product realm to the purely personal realm does not eliminate the need to design a

usable system. You have at least one user to think of – yourself! In point of fact, however, I doubt that any reader who builds a scope or TV graphics interface will be able to resist the temptation to show it off to his or her family and friends; so, even for "fun" systems, consideration of users is still a major input to the design.

The user interface for the LIFE program will provide the following functions to enable a pattern to be drawn on the screen and initiated:

1. *Cursor.* The display output should provide a "cursor" which is maintained all the time by a subroutine in the software at a given "X" and "Y" position of the matrix. Fig. 4 illustrates the point matrix of the screen (here assumed 64x64) and the cursor pattern. The cursor is a visual feedback through the display to the user of the LIFE program, illustrating where the program will place or erase information. Fig. 4 shows a blow-up of one possible cursor pattern.

Two additional features are required for a useful cursor output of the program for LIFE. These are:

– A blinking feature. Suppose you have filled the screen with a complicated pattern drawn with the cursor controls described below. A significant number of the screen points are now filled with dots – and there will be a strong tendency to confuse the cursor pattern of Fig. 4 with the actual data pattern you have entered. A "blink" feature can be built into the programs which create the cursor so that you will always be able to distinguish it by its flashes.

– A blanking feature. For the LIFE game, a necessary attribute of cursor control is the ability to blank out the cursor during the actual evolution of patterns. I consider this necessary due to observation of a demonstration LIFE program for one desk top programmable CRT terminal: its cursor is always present and mildly annoying when the LIFE game is in operation.

A basic way to make the cursor disappear from view at certain times is to require active control by cursor display routines when the program is in its input mode. If the LIFE program leaves the input mode to go evolve some patterns, the cursor will die a natural death until the active maintenance is resumed on return to the input mode.

2. *Cursor Control.* The whole purpose of the cursor is to provide a means of feeding back to you – the user – the current grid location the LIFE program is pondering. Movement of the cursor provides the opportunity for three types of data entry to the program:

– Positioning of the Cursor. By simply moving the cursor under control of the keyboard (see below) you can direct the LIFE program's attention to different parts of the screen.

– Sowing Seeds of LIFE. By moving the cursor while indicating a "birth" function, the cursor will leave a trail of "cells" indicated in the display by illuminated points. (One keyboard key is required for this function.)

– The Grim Reaper. By moving the cursor while indicating a "death" function, any cells in the path of the cursor will be eliminated, by turning off the corresponding display point. (One keyboard key is required for this function.)

Motion control is also used to enter data. By picking a data key and at the same time depressing one or two of the cursor direction keys, a "trail" will be left. A timing loop in the input program will be used to set a reasonable motion rate in the X (horizontal) and Y (vertical) directions, so that the data entry will be performed automatically as long as the keys are depressed. The motion control keys and useful combinations are illustrated in Fig. 5.

3. *Program Control Commands.* This is the section of the LIFE program design which is the software analog of the "backplane" data bus concept in a hardware system. LIFE Line concerns a modular LIFE program which will be subject to many variations and improvements.

Birth – the cursor leaves a path of "cells," illuminated points.

Death – cells in the cursor's path are eliminated.

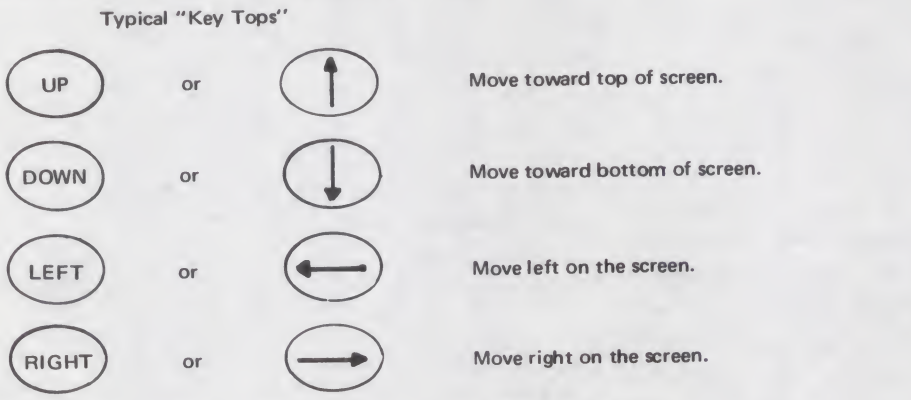
KILLING TWO BIRDS WITH ONE STONE, or "HOW I DESIGNED A GENERAL INTERACTIVE GRAPHICS SOFTWARE INITIALIZATION PACKAGE IN THE GUISE OF A SPECIFIC APPLICATION.

The ideas contained in this article are by no means limited to control of the graphics display type of device in the LIFE context used for this application. The only necessary connection between the LIFE program proper and the display "drawing" and updating functions is in the existence of several subroutines needed to turn on/turn off selected points, and the ability of the display input ("drawing") routines to call the LIFE program. One logical extension of the program control mechanisms to be included in LIFE Line is to allow the invocation (ie: activation, calling, etc.) of other programs and games which use the display.

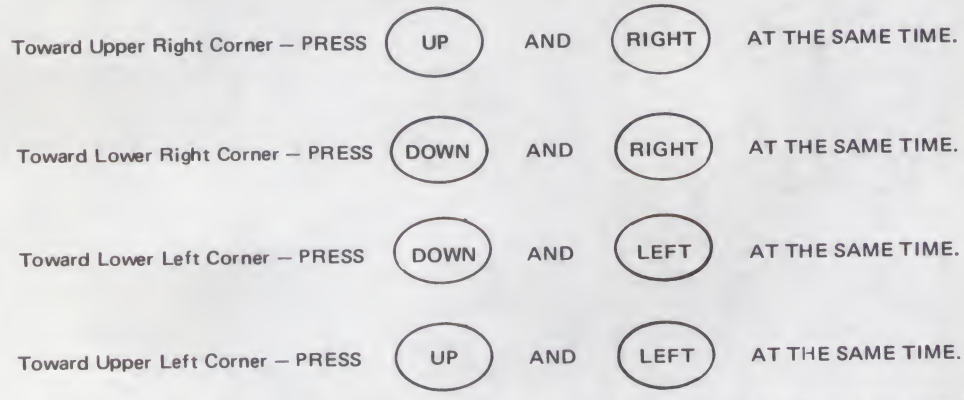
When the "drawing" routines are up and running, even before you hook up the LIFE algorithm proper, you'll be able to manipulate the contents of the scope under software control and draw pictures on the screen.

Fig. 5. Cursor motion control commands.

The following commands (one key on your keyboard for each) are used to simply move the cursor in one of the grid directions at a rate set by the cursor control software:



The following combinations can be used to achieve motion in diagonal directions:



Remember that all eight of these possibilities can be used to "sow the seeds" or erase data if the appropriate data key is pressed simultaneously.

The first demonstration of LIFE in these pages is just the bare bones of a LIFE program. When it is fully described you will see the input display routines, the evolution algorithm, the program control *mechanism* and little else. The program control *mechanism*, however, is quite general and will be used to integrate additional commands, variations on LIFE, etc. The means of achieving this modularity is a set of "hooks" which enable you to add commands beyond the bare minimum by coordinating new modules with the program. The following is a minimum set of program control commands for the first version:

RUN – a key assigned to this function will terminate the input ("drawing") mode, and begin the "run mode."

DRAW – a key assigned to this function will be tested during the "run" mode to cause a return to the "draw" mode.

CLEAR – a key assigned to this function will be used to clear the screen in the "drawing" mode, leaving only the cursor and a blank screen.

The above features are only a minimum set of user controls for LIFE. Additional program control commands which will prove invaluable when added include:

SAVE/RESTORE – commands to write and read LIFE patterns on cassette tape or other mass storage device in your home brew system.

INITIALIZATION – functional key entries for the generation of various "standard" LIFE patterns placed at the current cursor location.

Next month, LIFE Line will enter into the realm of software design to describe the LIFE program software in more detail.

LIFE Line Glossary.

Communication of meaning requires definition of terms. The following is a listing of selected terms used in LIFE Line with short explanations. The terms which are marked "L" are primarily significant only in the LIFE application – all others are fairly general terms.

"Active Control" – in the LIFE example, a desired requirement for the cursor is that it disappear automatically if not continually refreshed. This can be accomplished in software by instituting a "garbage sweeper" for the screen which clears the screen memory periodically and updates from the latest non-cursor sources of data. Normally, the cursor control/display subroutine would be called after the screen is updated – but if the cursor control routine is not called, the cursor will be absent after garbage sweeping. The cursor is thus said to require "active control" because it must be explicitly posted on the screen following the garbage sweeping operation if it is to appear at all. (L)

"Algorithm" – this term has a formal mathematical origin as the generalized methodology for arriving at some result. In the computer science area, it retains this definition: an algorithm is the most general processing required to achieve some result. "Algorithm" is a term which includes the term "program" in the following sense: a program is an algorithm (general) as written and coded for a specific system.

"Application" – an application is a specific system designed to accomplish some goal. In the computer systems area, applications are generally composed of hardware and software components which must "play together" to accomplish the desired functions. The LIFE Line's target – a working game of LIFE – is an example of an application.

"Backplane Bus" – the hardware concept of a set of wired connections between identical terminals of multiple sockets. In modular systems, the common wiring makes each socket identical to every other socket. Hardware modules can then be inserted without regard to position in the cabinet containing the equipment.

"Cellular Automata" – conventional computers employ a serial or sequential method of processing. One instruction, then the next, is executed in a time-ordered sequence. The "cellular automata" concept is one way of visualizing large and complicated parallel computing elements. Hypothetically, the LIFE game could be played by such a cellular computer, one which calculates each matrix element simultaneously. In the present state of computer technology, this is not possible, so you have to settle for a simulation of the parallel computation's result, using a serially executing program.(L)

"Coding" – the process of translating a functional specification of a program or routine into a set of machine readable elements for actual use in a computer. Coding can mean writing FORTRAN statements, writing PL/1 statements, writing assembly language statements, or . . . if you have no compiler, coding is the writing of machine codes directly onto a sheet of paper using tables of op codes, an eraser and patience.

"Cursor" – a mark on a display screen used to identify a particular place. This interpretation is an electronic adaptation of the standard definition in Webster.

"Evolution" – patterns in the game of LIFE change from generation to generation according to the rules. The sequence of such changes can loosely be called the evolution of the pattern.(L)

"Feedback" – in the context of system development, feedback is the use of observed system behavior to modify and improve the design of the system.

"Functional Specification" – a functional specification of a system is one which describes "what" the system must do, more or less independent of any technology which is required to make the "what" work. It is easy to come up with loose functional specifications – the hard part is to refine the specification and pin it down to something which is "do-able" in a given context of technology. I have a functional specification in my mind, for instance, of a useful interplanetary travel method – but whether or not I ever see such a system depends upon advances in physics, engineering and economic understanding. *BYTE* often concerns itself with functional specifications of much more "do-able" systems which readers can and will implement on home computers.

"Generation" – this term in the LIFE context means the present "state" of all the locations in the "universe of the grid" at some point in time.(L)

"Implement" – technical jargon verb for the creation of a system or element of a system. A hardware designer might implement a controller or a CPU; a software programmer implements a system of programs; a systems designer implements a hardware/software combination which achieves a desired functional end.

"Indexing" – the technique of referencing data in collection of similar items by means of numerical "indices." In the LIFE Line example, the collection is that of the 64x64 array of bits in the computer representation of "grid space." Indexing by row and by column is used to pick a particular bit within this array when the program requires the data.

"Interact" – when a system "interacts" with "something/person" it is operating under an algorithm which allows conditional behavior dependent upon data. The data is obtained from the "something/person" and may in fact be influenced by previous interactions as well as new inputs. In many computer contexts "interact" has the additional implication of "quick" response in "real time." Thus when you think of an "interactive" terminal or game, you think of a computer programmed so that it keeps up with the inputs from the human operator.

"Lexicon" – the list of buzzwords in any given field. This glossary is a subset of a lexicon coupled with explanations. In compiler and language design, "lexical analysis" is a derivative of this term concerned with language keywords and their relation to a grammar.

"n", "n+1", "n+2"... – when it is useful to specify a sequence of things, where no particular number is intended, a "relative" notation of the sequence is useful. "n" is some arbitrary number; "n+1" is one number greater than an arbitrary number, and so on. When I say "generation n+1" of LIFE, I mean the next generation after generation "n" where "n" is arbitrary.

A suitable LIFE display peripheral is an oscilloscope graphics interface such as the Digital Graphic Display Oscilloscope Interface designed by James Hogenson and printed in the May 1975 issue of ECS Magazine, *the predecessor to* BYTE. *The graphics interface article will be expanded and published in* BYTE *No. 2, October 1975. Until supplies are exhausted, back issues of May ECS (and earlier articles) can be ordered at $2 each. Orders and inquiries regarding ECS back issues should be sent to M. P. Publishing, Box 378, Belmont MA 02178.*

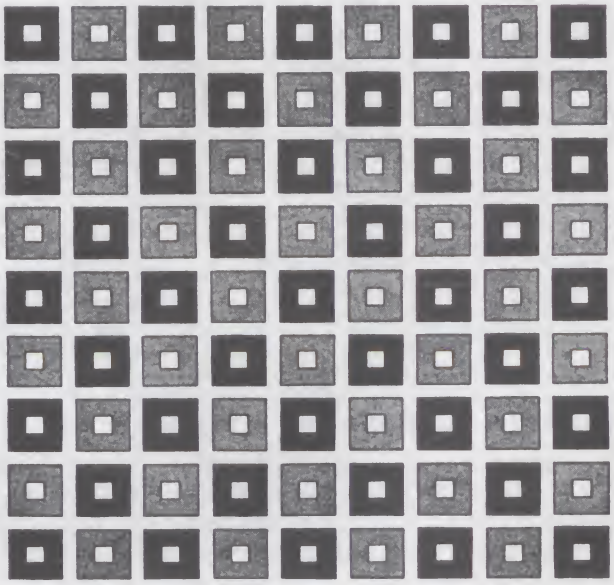
"Partitioning" – the technique of "divide and conquer." Rather than view a complicated system as a monolithic blob of "function," an extremely useful design method is to partition the system into little "bloblets" of function which are easy to understand. Hardware designers of CPUs thus think of MSI chips as sub-elements in partitioning; hardware systems designers think of CPUs and peripherals and memories as sub-elements of partitioning, and software designers consider divisions of complicated programs and program libraries as their sub-elements.

"State" – the present condition of some system, or elements of the system. This term applies to any system which has "memory" to distinguish one possible "state" from another. The term applies equally well to small sub-elements of a system such as the bits of a memory: in the LIFE Line context, the "state" of a single grid location is a number from 0 to 8 counting how many "neighbor cells" are present.

"System" – the most general of all general purpose terms. A system is a collection of component elements (technological, hardware, software, human-interface) selected to play together according to some design or purpose. A system is a human-invented way of doing things.

"Undefined in Detail" – I know what is needed, can specify its interface, but am not at present supplying the detail design. This is a useful attitude since it allows for "plug compatible" designs differing widely in their internal principles of operation. A similar expression would be to call the subsystem in question (the graphic display mentioned in this LIFE Line example) a "black box" and leave it at that. (Software always seems to reference hardware in this way, and hardware does the same for software.) A synonym for the attitude is the mathematician's way of saying "in principle there exists a solution!" without telling you what it is.

"Universe of the Grid" – this is the set of all possible places in which a LIFE cell could be placed. These places are called "grid locations".(L)



LIFE *Line* 2

by
Carl Helmers
Editor, BYTE

LIFE Line 1 (BYTE #1) presented the general picture of the LIFE program application of your computer. That picture includes the rules of the game, methods of interactively entering graphic data, major software components in verbal description and some of the hardware requirements of the game. In this installment, the discussion turns to some of the program design for the LIFE application. The discussion starts "at the top" (overall program flow) and works down to more detailed levels of design, concentrating upon the "evolution algorithm" which generates new patterns from old patterns.

As in the previous LIFE Line, the goal of the article series is as much to explain and instruct as it is to elaborate upon this one particular system. This article concentrates on the program design process as illustrated by a real LIFE example.

What Is This Process — Designing A Program?

For the readers who are only just now beginning to learn the programming of computers, an elementary acquaintance with some machine's language, a BASIC interpreter, or high level languages would tend to give the impression that programming is fundamentally simple. It is! To write a program which fills a single page of listing — whatever the language or machine architecture involved — is not a tremendously difficult task. When it comes to more complex projects — say 1000 or more words of hand or machine-generated code on your microcomputer — the problem is how to preserve the blissful innocence of simplicity in the face of the worldly forces of complexity.

When you begin to talk about programs more complex than a one page assembly or machine code listing of some specialized service routine or simple "gimmick" program (see the Kluge Harp article in this issue), the complexities and subtleties of scale begin to enter into the programming art. For an application such as the LIFE program, proceeding from the vague notion "I want this application" to a working program can be done in innumerable ways — many of which will work quite well. This is the first ambiguity of scale — where do you head as you start programming? Unless you have a unique parallel processing mind, you can't possibly concentrate on the whole problem of programming at once.

In order to make a big application program work, you have to select "bits and pieces" of the desired result, figure out what they do and how they fit into the big picture, then program them one by one. These little pieces of the program — its "modules" — are like the multiple layers of stone blocks in a pyramid. In fact, defining what to do is very much like the tip of some Egyptian tyrant's tomb in the spring flood . . . as the murky generalities recede, more and more of the structure of the program is defined and clarified. Fig. 1 illustrates the pyramid of abstractions at the start of a program design process. The top layer is clear — a LIFE program is the desired goal. The next layer down is for the most part visible through the obscuring water. But the details of the base of the pyramid — while you know they have to be there in some form — are not at all visible at the start. The design process moves the logical "water level" surrounding the pyramid lower and lower as you figure out more and more of the detail content of the program.

Start at the Top . . .

In LIFE Line 1, I mentioned two major functions which compose a practical LIFE program — data entry and manipulation is one, the LIFE evolution algorithm is the second. Together, these functions define the "program control" layer of the LIFE pyramid. Fig. 2 is a flow chart illustrating the program control algorithm which is the top level of the program structure. Although the diagram — and the algorithm — are extremely simple, they

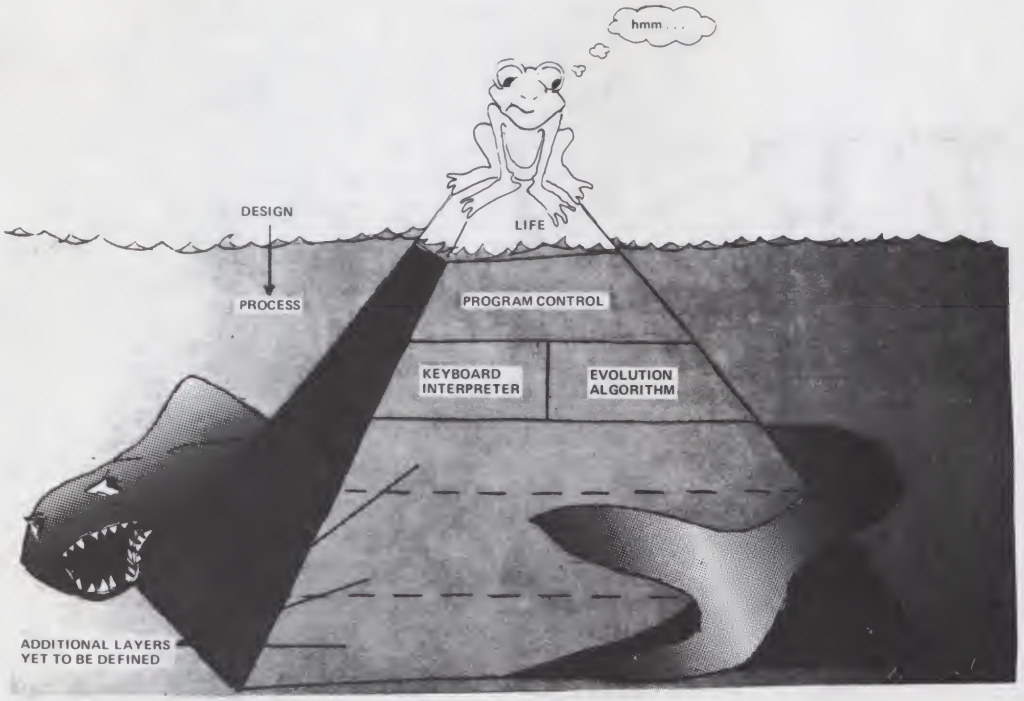


Fig. 1. Defining what to do is like the tip of some Egyptian tyrant's tomb in the spring flood . . . as the murky generalities recede more and more of the structure is defined and clarified . . .

serve a very useful purpose in the program design process: *This high level design has split most of the programming work into two moderately large segments, each of which is less complicated than the whole program.* This view of the problem now gives us two major components upon which to concentrate attention once the top level routine is completed. The program control algorithm of Fig. 2, elaborated in Fig. 3, is the "mortar" which cements together these two blocks of function.

The LIFE program is entered by one of a number of methods. Fig. 2 illustrates branch or jump possibilities from a systems program called a "monitor," "executive" or "operating system" – the preferred way once you get such a system generated. If your system runs "bare bones" with little system-resident software, you might select the starting point and activate the program through use of hardware restart mechanisms and a front panel console.

The first module of the LIFE application to be entered is the KEYBOARD_INTERPRETER, a set of routines which is used to define the initial content of the LIFE grid using interactive commands and the scope display output. The KEYBOARD_INTERPRETER eventually will receive a "GO" command or an "END" command from the user – whereupon it will return to the main routine with the parameters "DONE" and "N" defined. If "N" is greater than zero, control flows to the evolution process – and "N" generations of LIFE will be computed and displayed as they are completed. After the "N" generations have been completed, the scope display and the LIFE grid have the last completed results. If the program is not "DONE," control flows back to the KEYBOARD_INTERPRETER for modification of the data, clearing the screen and starting over, or other operations. If the program is "DONE" then the control flows back to the systems programs – or to a halt point.

This program control algorithm is elaborated in more explicit detail in Fig. 3.

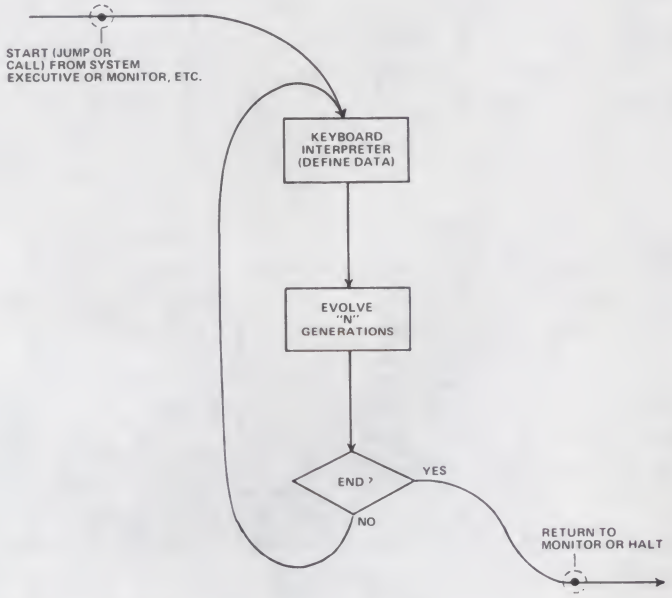


Fig. 2. LIFE program flow of control.

Fig. 3. The main control routine of LIFE specified in a procedure-oriented language . . .

```
1  LIFE:
2     PROGRAM;
3        DONE = FALSE;
4        DO UNTIL DONE = TRUE;
5           CALL KEYBOARD_INTERPRETER (N,DONE);
6           DO FOR I = 1 TO N;
7              CALL GENERATION;
8           END;
9        END;
10       RETURN; / * TO EXECUTIVE, MONITOR, OR JUST HALT */
11       CLOSE LIFE;
```

Subroutines Referenced by LIFE:

KEYBOARD_INTERPRETER . . . This is the routine which looks at the interactive keyboard and interprets user actions such as specifying initial patterns, modifying patterns, etc. N is defined by the GO command which causes return from this subroutine to LIFE.

GENERATION . . . This is the routine which is used to evolve one generation of the life matrix and display the result. Since the entire matrix is kept in software by *GENERATION* until after a new matrix has been evolved, there will never be any partially updated patterns on the scope.

Data (8 bit bytes) used by LIFE at this level:

FALSE – the value "0".

TRUE – the value "1".

DONE – variable set by *KEYBOARD_INTERPRETER* after a user command (GO) to start execution.

N – a variable set by user interaction in *KEYBOARD_INTERPRETER* giving the number of generations to evolve.

I – a temporary loop index variable.

. . . the problem is how to preserve the blissful innocence of simplicity in the face of the wordly forces of complexity.

Fig. 3 uses a "procedure-oriented language" (see the box accompanying this article) to specify the program in a more explicit and compact form than is possible with a flow chart. Each line of the program as specified in Fig. 3 could potentially be compiled by an appropriate compiler – but for the purposes of most home computer systems, generation of code from this model would be done by hand. The outer loop is performed by a "DO UNTIL" construct starting at line 4 and extending through line 9. The program elements found on lines 5 to 8 are executed over and over again until DONE is found to be equal to logical 1 or "TRUE" when a test is made at the END statement of line 9. A "DO FOR" loop is used to sequence "N" calls to a subroutine called GENERATION which does the actual work of computing the next generation content and then displaying it on the scope. The remainder of Fig. 3 summarizes the data and subroutines referenced by LIFE.

From this point, the LIFE Line can extend in two directions. In order to have a complete LIFE program, both areas have to be traversed – the KEYBOARD_INTERPRETER and the GENERATION routine . . .but the partitioning has nicely separated the two problems. The simpler and most self-contained of the two segments is the GENERATION algorithm, so I'll turn attention to it next.

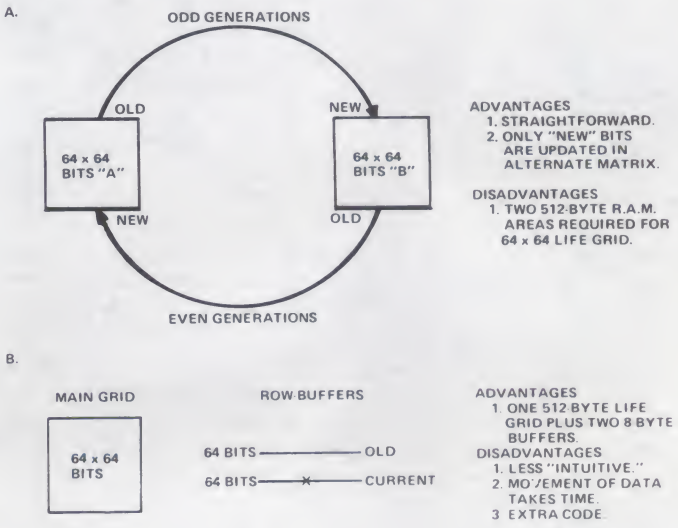
Grid Scanning Strategies

The GENERATION subroutines of the LIFE program has as its design goal the transformation of one complete LIFE grid pattern into the "next" complete pattern The rules of the Game of LIFE – the "facts of life" – must be applied to each location in the grid to compute the next value of that location. Fig. 4 illustrates two potential strategies for computing the next generation – methods of scanning the grid to compute one location at a time.

The first strategy, Fig. 4(a), is to employ alternate copies of a complete LIFE grid of 64 by 64 points. If generations are numbered consecutively, the generation algorithm would transform copy A into a "next" copy in B on odd generations, and complete the cycle by transforming copy B into a "next" copy in the A grid on even generations. Since each grid requires 4096 bits – which can be packed into 512 bytes – a total of 1024 bytes is required for data storage if this method is used. The primary advantages of this method are its "straightforward" nature and its separation of old and new data at all times.

A second strategy is illustrated in Fig. 4(b), the strategy of using alternate row-buffers with only one

Fig. 4. The LIFE evolution algorithm – matrix scanning techniques which preserve relevant old information while creating new information in overlapping storage areas.



main grid copy. Two 64 bit rows must be maintained – the last previous row and the current row – as 8 byte copies. These copies contain information *prior to updating* in the row by row scan down the matrix. The main advantage is a saving of data areas (partially offset by more complicated software). The main disadvantages are its less "intuitive" nature, the extra time required to make data copies, and a slightly larger program.

The choice between these two methods is primarily one of the amount of storage to be devoted to data. The tradeoff is in favor of the double matrix method when very small LIFE matrix sizes are considered. The extra 8 bytes required for a second copy of an 8 by 8 grid of bits hardly compares to the programming cost of the alternate row-buffer strategy. When large matrices are considered, however, the memory requirements of an extra copy of the data are considerable, but the programming involved is no more difficult. For example, consider the limit of an 8 bit indexing method – a 256 by 256 grid. This will require a total of 8192 bytes for *each copy* of the LIFE grid. Two copies of the LIFE grid would use up 16k bytes, or one fourth of the addressing space of a typical contemporary microcomputer, and all of the addressing space of an 8008 microcomputer! At the 64 x 64 bit level, the tradeoff is much closer to the break-even point, but I expect to find at least 100 bytes saved as a result of using the row-buffer method. An assumption which is also being made when the alternate row-buffer method is used is that the scope display or TV display you use for output will have its own refresh memory so that the "old" pattern can be held during computation of the new. If this is not the case, a less desirable output in which partially updated patterns are seen will be the result. (Counting the CRT refresh, the method of Fig. 4(a) thus requires three full copies of the matrix information, and the method of Fig. 4(b) requires two full copies.)

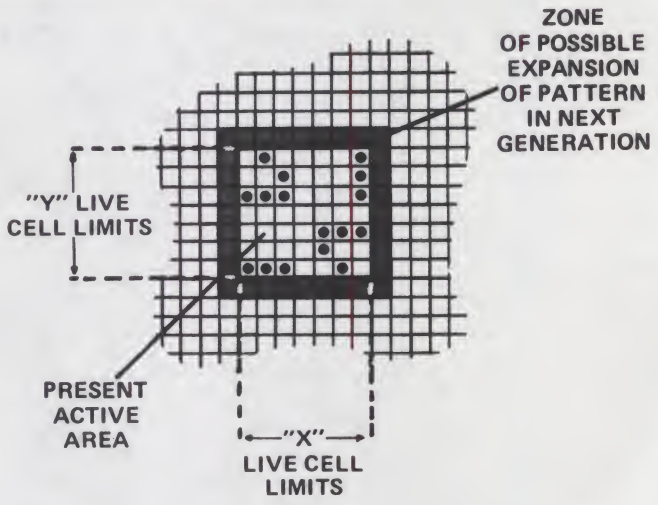
An objective: Split the processing into moderately large segments, each of which is less complicated than the program taken as a whole.

Active Area Optimization

With the choice of a matrix scanning strategy – the alternate row-buffer method – another consideration in designing the generation algorithm is a computation time optimization method. There is no real need to calculate a new value of every cell in a mostly empty LIFE grid. If I only have one glider with its corner at location (34, 27) of the grid, why should I compute any new generation information outside the area which could possibly be affected by the present pattern's evolution? Again, the savings in computation time using active area optimization depend upon the size of the grid. If most patterns occupy the full grid, then very little will be saved – for the small 8 x 8 grid "straw man" used in discussing scanning strategy, there would also be no point to active area optimization. But with a huge 256 by 256 grid, and an 8 by 8 active area, this optimization might mean the difference between a 15 minute computation and a 1 or 2 second computation of the next generation.

Fig. 5 illustrates the concept of active area optimization in a LIFE program. The current generation's active area is defined as the set of X and Y limits on the extent of live cells in the grid. In Fig. 5, the active area is the inner square of 7 x 7 = 49 grid locations. In computing the next generation, a box which is one location wider in each of the four cardinal directions is the "zone of possible expansion" for the pattern. In Fig. 5, this zone is the outer box of 9 by 9 locations. The computation of "next generation" values need only be carried out for the 81 grid locations bounded by the outer limits of the zone of possible expansion. Thus in the case of the 64 by 64 matrix of LIFE points, this optimization for the pattern of Fig. 5 will limit the program to calculation of 81 new points versus the 4096 points which would be calculated if at least one cell was found at each of the minimum (0) and maximum (63) values of the X and Y

Fig. 5. Active area optimization – never compute more than the absolute minimum if speed is at a premium.



The "facts of life" must be applied to each location in the grid to compute the next value – cell or no cell – of that location.

coordinates. This case yields a savings of 98% of the maximum generation to generation computing time.

The GENERATION Subroutine

Fig. 6 illustrates the code of the GENERATION routine, specified in a procedure-oriented language, along with notes on further subroutines and data requirements. The procedure starts by initializing the data used for the scan of the matrix, in lines 3 through 15. THIS and THAT are used to alternately reference the 0 and 1 copies of an 8 byte data item called a 2 by 8 byte data area called "TEMP". (Subscripts, like in XPL and PL/M are taken to run 0 through the dimension minus 1.) NRMIN, NRMAX, NCMIN, and NCMAX are used to keep track of the new active area limits after this generation is computed; NROWMIN, NROWMAX, NCOLMIN and NCOLMAX are originally initialized by the KEYBOARD_INTERPRETER and are updated by LIMITCHECK after each generation is calculated – using the new active area limits.

The actual scan of the grid of LIFE, stored in the data area called LIFEBITS, is

Fig. 6. The GENERATION routine specified in a procedure-oriented language . . .

```
 1 GENERATION:
 2    PROCEDURE;
 3    THIS = 0;  /* INITIALIZE POINTERS TO TEMPORARY ROW  */
 4    THAT = 1;  /*     COPY VARIABLE "TEMP"              */
 5    DO FOR I = 0 TO 7;
 6       IF NROWMIN = 0 THEN
 7          TEMP (THAT,I) = LIFEBITS(63,I);
 8       ELSE
 9          TEMP (THAT,I) = 0;
10       /* THIS ESTABLISHED WRAP-AROUND BOUNDARY CONDITION */
11    END;
12    NRMIN = 99;  /* THEN INITIALIZE ACTIVE AREA LIMITS  */
13    NRMAX = 0;
14    NCMIN = 99;
15    NCMAX = 0;
16 ROW_LOOP:
17    DO FOR IROW = NROWMIN TO NROWMAX; /* SCAN ACTIVE ROWS ONLY */
18       DO FOR I = 0 TO 7;  /* COPY THIS ROW TO TEMPORARY */
19          TEMP (THIS,I) = LIFEBITS (IROW,I);
20       END;
21       DO FOR ICOL = NCOLMIN TO NCOLMAX; /* SCAN ACTIVE COLUMNS ONLY */
22          CALL FACTS_OF_LIFE (IROW, ICOL);
23       END;
24       X = THIS;
25       THIS = THAT;
26       THAT = X; /* THIS SWITCHES BUFFERS */
27    END;
28    CALL LIMITCHECK;
29    CALL DISPLAY;
30    CLOSE GENERATION;
```

Subroutines Referenced by GENERATION:

EVOLVER . . . This is the routine used to calculate the next value of the ICOLth bit in the IROWth row of LIFEBITS using the current value of the next row, the saved value in TEMP of the previous row, and the saved value in TEMP of the current row before updating.

LIMITCHECK . . . This is the routine used to calculate the next values of NROWMIN, NROWMAX, NCOLMIN, NCOLMAX using the current values of NRMIN, NRMAX, NCMIN and NCMAX.

DISPLAY . . . This routine transfers the LIFEBITS data to the display, on whatever kind of device you have.

Data (8 bit bytes) used by GENERATION at this level:

X = TEMPORARY
I = temporary index (not the same as the I in Fig. 3)
ICOL = index for column scanning . . .
IROW = index for row scanning . . .
NCMAX = current maximum column index of live cells
NCMIN = current minimum column index of live cells
NRMAX = current maximum row index of live cells
NRMIN = current minimum row index of live cells

Data (8 bit bytes) used by GENERATION but shared with the whole program:

THIS = current line copy index into TEMP.
THAT = previous line copy index into TEMP.
TEMP = 2 by 8 array of bytes containing 2 64-bit rows.
NROWMIN = minimum row index of live cells.
NROWMAX = maximum row index of live cells.
NCOLMIN = minimum column index of live cells.
NCOLMAX = maximum column index of live cells.
LIFEBITS = 64 by 8 array of bytes containing 64 rows of 64 bits.

Assumptions:

LIFEBITS, NROWMIN, NROWMAX, NCOLMIN, NCOLMAX are initialized in *KEYBOARD_INTERPRETER* for the first time prior to entry – and retain old values across multiple executions of *GENERATION* thereafter.

Fig. 7. The LIMITCHECK routine specified in a procedure-oriented language . . .

```
 1  LIMITCHECK:
 2     PROCEDURE;
 3     /* CALCULATE NEXT ROW LIMITS */
 4     IF NRMIN-1 < NROWMIN THEN NROWMIN = NRMIN-1;
 5     IF NRMAX+1 > NROWMAX THEN NROWMAX = NRMAX+1;
 6     IF NROWMAX > 63 THEN NROWMAX = 63;
 7     IF NROWMIN < 0 THEN NROWMIN = 0;
 8     /* CALCULATE NEXT COLUMN LIMITS */
 9     IF NCMIN-1 < NCOLMIN THEN NCOLMIN = NCMIN-1;
10     IF NCMAX+1 > NCOLMAX THEN NCOLMAX = NCMAX+1;
11     IF NCOLMAX > 63 THEN NCOLMAX = 63;
12     IF NCOLMIN < 0 THEN NCOLMIN = 0;
13     CLOSE LIMITCHECK:
```

Subroutines Referenced by LIMITCHECK:

None.

Data (8 bit bytes) used by LIMITCHECK but shared with the whole program:

NCOLMAX, NCOLMIN, NROWMAX, NROWMIN
NRMAX, NRMIN, NCMAX, NCMIN *(see Fig. 6)*

Assumptions:

The arithmetic of the comparisons in this routine is done using signed two's complement arithmetic – thus a negative number results if 0 - 1 is calculated . . . this is consistent with code generation on most 8 bit micros.

performed by the set of DO groups beginning with ROW_LOOP at line 16. For each row of the matrix, ROW_LOOP first copies the row into TEMP as the THIS copy (the THAT copy is left over from initialization the first time at lines 5 to 11, or from the previous ROW_LOOP iteration thereafter). Following the copying operation, another DO FOR loop goes from NCOLMIN to NCOLMAX applying the FACTS_OF_LIFE to each grid position in the current (THIS) row as saved in TEMP. New data is stored back into LIFEBITS by FACTS_OF_LIFE. At the end of the row loop, prior to reiteration, the THIS and THAT copies of temp are switched by changing the indices. What was THIS row becomes THAT row with respect to the next row to be computed.

After all the rows have been computed, line 28 is reached. Line 28 calls subroutine LIMITCHECK to compute the next generation's active area computation limits using the results of this generation. Line 29 then calls a module named DISPLAY to copy the results of GENERATION into the output display device. The LIMITCHECK routine simply performs comparisons and updating – Fig. 7 illustrates the high level language description of its logic.

Computing The Facts of LIFE . . .

Fig. 8 contains the information on implementing the Facts of LIFE in a programmed set of instructions. The computation is divided into two major parts. The first part is to determine the STATE of the bit being updated, where "STATE" is a number from 0 to 8 as described in LIFE Line 1 last month. The second major step is to evolve the grid location using its current value and the STATE.

FACTS_OF_LIFE begins by performing left and bottom boundary wrap-around checks by adjusting indices. Lines 8 to 18 determine the current STATE by referencing all 8 grid locations surrounding the location being computed at (IROW, ICOL). In determining the state, the subroutines TGET and LGET

Two copies of a 256 by 256 grid would require more memory than (for example) an 8008 can address if you want to have programs along with your data.

Why should I compute any new generation information outside the area which could possibly be affected by the present pattern's evolution?

Fig. 8. The FACTS_OF_LIFE routine specified in a procedure-oriented language. FACTS_OF_LIFE does the actual calculation of the next value for the LIFEBITS location at the IROWth row and ICOLth column based upon the previous value of the 8 neighboring locations. (The state defined in LIFE Line 1, last month.) This routine implements the rules described in BYTE #1, page 73.

```
 1  FACTS_OF_LIFE:
 2     PROCEDURE (IROW,ICOL);
 3     M = IROW + 1;
 4     IF M >63 THEN M = 0; /* BOTTOM BOUNDARY WRAP CONDITION */
 5     N = ICOL - 1;
 6     IF N < 0 THEN N = 63; /* LEFT BOUNDARY WRAP CONDITION */
 7  DETERMINE_STATE:
 8     STATE = TGET (THAT,N);
 9     STATE = STATE + TGET (THIS,N);
10     STATE = STATE + LGET (M,N);
11     N = ICOL;
12     STATE = STATE + TGET (THAT,N);
13     STATE = STATE + LGET (M,N);
14     N = ICOL + 1;
15     IF N > 63 THEN N = 0; /* RIGHT BOUNDARY WRAP CONDITION */
16     STATE = STATE + TGET (THAT,N);
17     STATE = STATE + TGET (THIS,N);
18     STATE = STATE + LGET (M,N);
19  EVOLVEIT:
20     NEWCELL = 0; /* DEFAULT EMPTY LOCATION UNLESS OTHERWISE */
21     OLDCELL = TGET (THIS, ICOL);
22     IF OLDCELL = 1 THEN DO;
23        IF STATE = 2 OR STATE = 3 THEN NEWCELL = 1;
24     END;
25     ELSE DO;
26        IF STATE = 3 THEN NEWCELL = 1;
27     END;
28     CALL LPUT (IROW, ICOL, NEWCELL);
29     IF NEWCELL = 1 THEN CALL SETLIMIT (IROW, ICOL);
30     CLOSE   FACTS_OF_LIFE;
```

Subroutines Referenced by FACTS_OF_LIFE:

TGET . . . This is a "function" subroutine which returns an 8 bit value (for example in an accumulator when you generate code) of 00000001 or 00000000 depending upon whether or not a referenced column in one of the two temporary line copies in TEMP is 1 or 0 respectively. The first argument tells which line of the two, and the second argument tells which column (0 to 63) is to be retrieved.

LGET . . . This is a "function" subroutine which returns an 8 bit value similar to TGET, but taken instead from the bit value at a specified row/column location of LIFEBITS.

LPUT . . . This subroutine is used to set a new value into the specified row/column location of LIFEBITS.

NOTE: The routines LGET and LPUT will be referenced from the KEYBOARD_INTERPRETER *routine in the course of manipulating data when setting up a life pattern.*

SETLIMIT . . . This subroutine is used to check the current active region limits when the result of the facts of life indicate a live cell.

Data (8 bit bytes) used by FACTS_OF_LIFE at this level:

IROW = Parameter passed from *GENERATION.*
ICOL = Parameter passed from *GENERATION.*
M = temporary, row index.
N = temporary, column index.
STATE = count of "on" bits in neighborhood of *IROW, ICOL.*
OLDCELL = temporary copy of old cell at *IROW, ICOL.*
NEWCELL = new value for location *IROW, ICOL.*

Data (8 bit bytes) used by FACTS_OF_LIFE but shared with the whole program:

THAT, THIS (see Fig. 6)

What was THIS row becomes THAT row with respect to the next row to be computed. (What's in a name? A pointer of course!)

are used to reference bits in TEMP and LIFEBITS respectively, using appropriate bit location indices. The values returned by these two "function subroutines" are either 0 or 1 in all cases — thus counting the number of "on" cells consists of adding up all the TGET or LGET references required to examine neighboring grid locations.

Once the STATE of the grid location is determined, the Facts of LIFE are implemented by examining the *positive* cases of an "on" (live cell) value for the grid location. A cell will be in the grid location for the next generation in only two cases: If the old content of the location was a live cell and the STATE is 2 or 3; or if the old content of the location is 0 (no cell) and the STATE is 3. A default of NEWCELL = 0 covers all the other cases if these two do not hold. Line 28 stashes the new value away in LIFEBITS with subroutine LPUT, and if the new value of the grid location

Fig. 9. The SETLIMIT routine specified in a procedure-oriented language.

```
1 SETLIMIT:
2    PROCEDURE (IROW,ICOL);
3    IF IROW < NRMIN THEN NRMIN = IROW;
4    IF IROW > NRMAX THEN NRMAX = IROW;
5    IF ICOL < NCMIN THEN NCMIN = ICOL;
6    IF ICOL > NCMAX THEN NCMAX = ICOL;
7    CLOSE SETLIMIT;
```

Subroutines Referenced by SETLIMIT:
None.

Data (8 bit bytes) used by SETLIMIT at this level:

IROW = parameter passed from *FACTS_OF_LIFE.*
ICOL = parameter passed from *FACTS_OF_LIFE.*

Data (8 bit bytes) used by SETLIMIT but shared with the whole program:

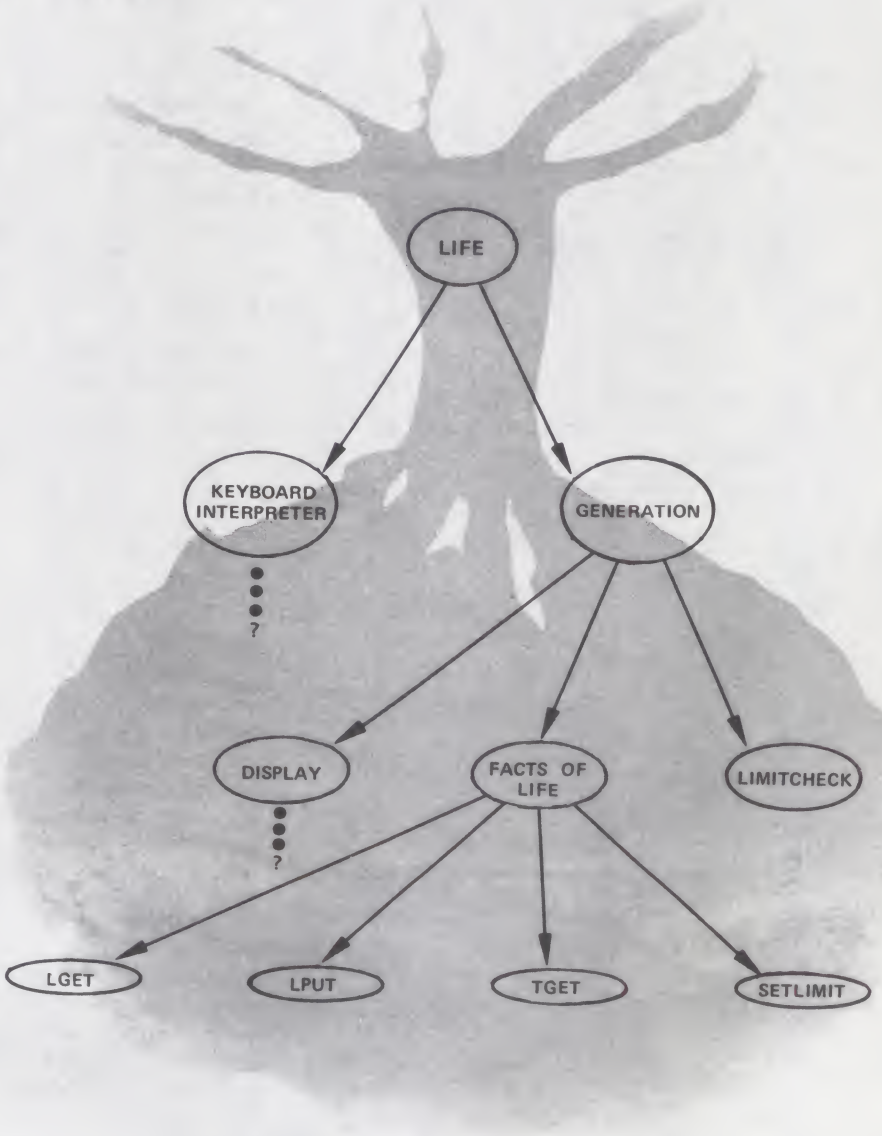
NRMIN, NRMAX, NCMIN, NCMAX *(see Fig. 6)*

is a live cell, SETLIMIT is called (see Fig. 9) in order to update the active area pointers NRMAX, NRMIN, NCMAX and NCMIN.

Where Does the LIFE Application Stand?

An alternative to the pyramid structure way of viewing programming program designs introduced at the beginning of this article is a "tree" notation showing the heirarchy of modules in the application. The "Tree of LIFE" is shown in Fig. 10 as it exists in materials printed to date. The next installment of LIFE Line will explore the left hand branch of the tree diagram by a similar presentation of a KEYBOARD_INTERPRETER algorithm.

Fig. 10. The Tree of LIFE.



LIFE Line 2 Addendum

Procedure-Oriented Computer Languages

The examples of programs accompanying two articles in this issue have been constructed in a procedure-oriented language. This method of program representation is compact and complete. In principle, one could write a compiler to automatically translate the programs written this way into machine codes for some computer. By writing the programs in this manner, more detail is provided than in a flow chart, and the program is retained in a machine independent form.

The particular representation used here resembles several languages in the "PL/1" family of computer languages, but is not intended for compilation by any existing compiler. For readers familiar with such languages, you will find a strong PL/1 influence and a moderate XPL influence. In a future issue BYTE will be running articles on a language specifically designed for microcomputer systems, PL/M, which is an adaptation of the XPL language for 8-bit machines. For the time being, this representation is used with some notes to aid your understanding.

Programs and Procedures

A *program* is a group of lines which extends from a PROGRAM statement to a matching CLOSE statement. It is intended as the "main routine" of an application. A *procedure* is a similar group of lines which extends from a PROCEDURE statement to its CLOSE statement. A procedure may have *parameters* indicated in the PROCEDURE statement, and may be *called* as a "subroutine" from a program or another procedure. A procedure may be called in a "function" sense as well, in which case a RETURN statement would be required to set a value.

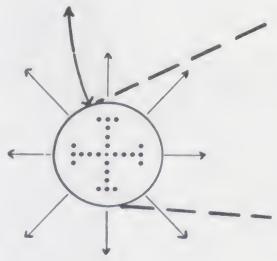
Data

For the purposes of these examples, no "data declarations" are put into the programs to complicate the picture. Instead, each example has a section following it which verbally describes each data name used. Only one "data type" is considered at this point – integers – and these are generally assumed to be 8 bits.

Arrays of integers are used in several examples. An array is a group of bytes, starting at the location of its *address* and extending through ascending memory addresses from the starting point. The purpose of an array is to reference "elements" within the array by "subscripts". For these examples, the elements are referenced by the numbers 0 through "n-1" for an array dimension of length "n". If LIFEBITS is an array of 64 by 8 bytes, then LIFEBITS(63,7) is the last element of the last row of the array, and LIFEBITS(I,J) is the byte at row I, column J provided I and J are within the proper ranges.

Statements

A program or procedure consists of statements which specify what the computer should do. In a machine language, these would correspond to the selected operation codes of the computer which is being programmed. For a high level language, one statement typically represents several machine instructions. In these the high level language statement has a "semantic intent" – a definition of its operation – which can be translated into the lower level machine language. In these examples several types of statements are employed . . .



"IF . . . THEN . . . ELSE . . ." constructs are used for notation of decisions. The first set of ellipses indicate a condition which is to be tested. The second set of ellipses in the model is used to stand for the "true part" – a statement (or DO group) which is to be executed if and only if the condition is true; the third set of ellipses is the "false part" – a statement which is only executed if the condition is false. The word ELSE and the whole false part are often omitted if not needed.

"CALL X" is a statement used to call a subroutine, in its simplest form. A more complicated form is to say CALL X(Z) where Z is a set of "arguments" to be passed to the routine. Another form of subroutine call is the "function reference" in an assignment statement, where the name of the subroutine is used as a term in an arithmetic expression.

"assignment" – a statement of the form "X = Y;" is called the *assignment statement.* Y is "evaluated" and the result is moved into X when the statement is executed. If X or Y have subscripts as in "TEMP(THIS,I) = LIFEBITS(IROW,I);" then the subscripts (such as "THIS,I" and "IROW,I" in the example) are used to reference the name as an array and pick particular bytes.

"DO groups" – a grouping of several statements beginning with a "DO" *statement and* running through a corresponding "END" is used to collect statements for a logical purpose. In "DO FOR I = 0 to 7;" this purpose is to execute the next few statements through the corresponding "END;" 8 times with I ranging from 0 to 7. "DO UNTIL DONE=TRUE;" is an example of a group which is repeated indefinitely until a condition is met at the END. "DO FOREVER" is a handy way of noting a group to be repeated over and over with no end test, a practice often frowned upon.

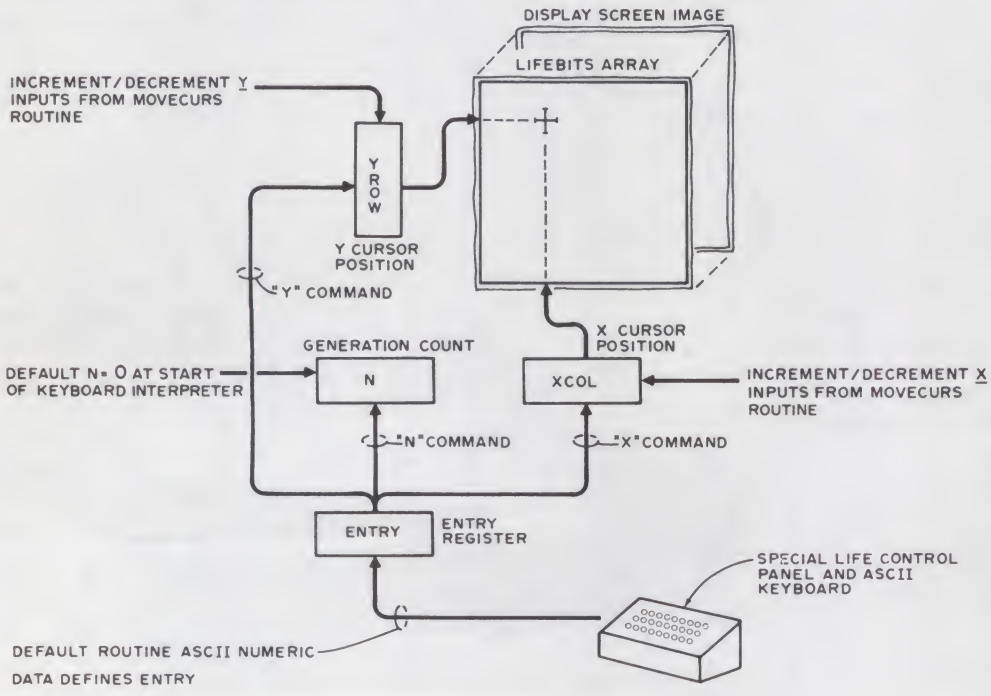
LIFE Line 3

by
Carl Helmers
Editor, BYTE

Program design is a process which can be approached in a haphazard manner – or by a systematic exploration of what is needed to achieve the desired end. LIFE Line 2 in BYTE #2 began the systematic exploration of the Tree of LIFE by presenting information on the overall program design of LIFE, as well as the details of the GENERATION algorithm used to carry one generation of LIFE into the next.

LIFE Line 3 continues the development of LIFE by a discussion of the KEYBOARD_INTERPRETER procedure. This procedure monitors the "user inputs" of a keyboard, and uses the command keystrokes detected to dictate what LIFE will do. As in the exploration of the GENERATION algorithm, the presentation starts at the top and works downward.

Fig. 1. Data concepts for LIFE *program and graphics control. The variables XCOL, YROW, N and ENTRY are 8-bit "software registers" maintained as variables in the* LIFE *program.*

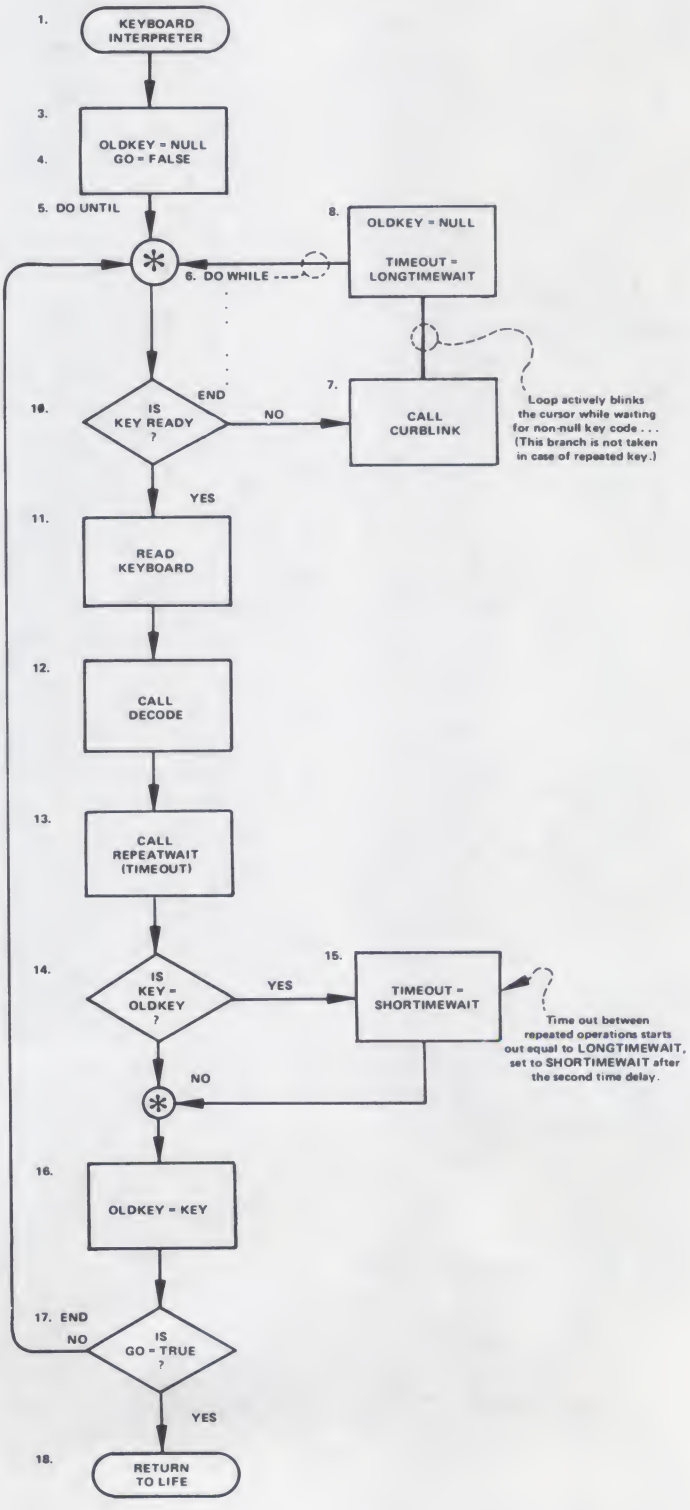


Much of the challenge and fun of the LIFE application is the fact that it is best implemented with some form of interactive graphics. In the partition of the application presented in LIFE Line 2, one of the major pieces of the program is the KEYBOARD_INTERPRETER with its interactive graphics concepts. A good place to start the discussion of the KEYBOARD_INTERPRETER is the *software* block diagram of the interactive graphics system of LIFE.

A Software Block Diagram?

Yes! Strange as it may sound to hardware types, the ebb and flow of data in a program can be depicted in block diagrams. While Fig. 1 looks very much like an ordinary hardware block diagram of some system, it is descriptive of the *plan* of data flow in a program rather than actual wires. Fig. 1 is the programming equivalent in every respect of the hardware block diagram of some dedicated interactive graphics system. By retaining the system in software, LIFE is inherently more flexible than any hard-wired system could be. This block diagram illustrates the potential flow of data in LIFE as controlled through the KEYBOARD_INTERPRETER and its

Fig. 2. An overall view of the KEYBOARD_INTERPRETER. *This is a flow chart of the control algorithm for the* LIFE *application's* KEYBOARD_INTERPRETER *routine. Fig. 3 shows the same information in the form of a procedure-oriented language.*



subroutines. Data flows and changes in response to the several input commands defined for the program.

As was pointed out in LIFE Line 1, the fundamental tool of an interactive graphics application is a *cursor* which illustrates where the program thinks attention should be placed. This cursor is flashed on and off on the screen, and can be moved through appropriate commands of the user sent via a keyboard. The cursor concept is implemented in the LIFE program application by means of two "global" variables called XCOL and YROW. These are both 8-bit bytes of data. But since the maximum dimension value in either the X or Y directions of the display is 63 (i.e., 6 bits) only the low order 6 bits have significance for cursor control. At any point in time during the execution of LIFE, the variables YROW and XCOL retain the location of the cursor for KEYBOARD_INTERPRETER's use.

Fig. 1 also shows arrows directed from XCOL and YROW to intersecting dotted lines in the LIFEBITS array. These two numbers together have 12 bits of significance. This is sufficient to uniquely specify one of the 4096 bits in the array using the utility subroutines LGET and LPUT to reference and change LIFEBITS, respectively. These routines are left to a later LIFE Line for their details.

A "ghost copy" of LIFEBITS is also shown in back of the main copy in the drawing to emphasize the following point: Each bit of the internal LIFEBITS array maps directly into a corresponding bit in the refresh memory of the CRT display subsystem. This is an example of a common theme throughout the use and abuse of computer systems: Software systems map into corresponding hardware — and vice versa. This mapping is of course one to one, and is carried out by the DISPLAY subroutine whenever the internal data is changed. As with LGET and LPUT, DISPLAY is left to a future LIFE Line for its details.

What Does it Take to Move the Cursor?

Since the cursor position is maintained by the values of XCOL and YROW, the movement of the cursor is simplicity at its essence: To move the cursor, all you have to do is change the value of XCOL, YROW or both! The interactive graphics portion of KEYBOARD_INTERPRETER has as its primary concern the various ways of

The simultaneous advantage and disadvantage of the multiple conditional test method of decoding: It is a plodding (but straightforward) approach which squanders memory resources.

changing the values of these two crucial variables – while possibly leaving a trail of changed data points in LIFEBITS. Fig. 1 illustrates several of these changes –

– To move the cursor up, YROW is incremented.

– To move the cursor down, YROW is decremented.

– To move the cursor left, XCOL is decremented.

– To move the cursor right, XCOL is incremented.

– To completely redefine the column of the cursor, the ENTRY register is transferred to XCOL.

– To completely redefine the row of the cursor, the ENTRY register is transferred to YROW.

KEYBOARD_INTERPRETER performs these actions at the whim of the user via commands entered at the keyboards of the system – with the flashing cursor mark on the screen showing the results.

The ENTRY Register

In order to provide a means of entering 8-bit integers into the program for control purposes, the software of KEYBOARD_INTERPRETER maintains a numeric input area called ENTRY. Whenever an ASCII character is sent to the program which cannot be decoded by DECODE's COMMAND table, the last resort is to call DEFAULT. In DEFAULT, the recover assumption is to interpret the unknown command as a numeric digit (0 to 9) and push it into ENTRY. A routine in DEFAULT performs a BCD to binary conversion of the ASCII character after it has been trimmed to the range 0 to 9. Later, when the user wants to define XCOL, YROW or N, the commands X, Y and N respectively are used to transfer ENTRY to one of the other registers, after which ENTRY is set to 0 in preparation for re-use. It is important to emphasize that ENTRY is a *binary number.* When decimal digits are entered by the user, the input routines convert the digits into the appropriate binary number and decimally shift the significance of the previous value.

The N Register

In LIFE Line 2, the LIFE program given in Fig. 3 references a variable called N. This N is used to control the number of times GENERATION is called. N, like XCOL and YROW, is a "software register" in the LIFE program which may be set by a user command. The "N" command is what is used to transfer the ENTRY value to N for use in controlling the

Fig. 3. The KEYBOARD_INTERPRETER *routine's overall flow, expressed in a procedure-oriented language. Note that the interpretation of the* "DO WHILE" *differs from a* "DO UNTIL" *– the former has its test prior to execution of the loop statements, and the latter has its test at the end of the loop. Nesting of the* DO *groups is indicated by the indentation of lines.*

```
1   KEYBOARD_INTERPRETER:
2     PROCEDURE;
3     OLDKEY = NULL;
4     GO = FALSE;
5     DO UNTIL GO = TRUE; /* LOOP UNTIL DONE WITH INPUTS */
6       DO WHILE NOTREADY(KEYBOARD) = TRUE;
7         CALL CURBLINK;              /* THIS LITTLE LOOP WAITS */
8         OLDKEY = NULL;              /* FOR A KEYSTROKE AND BLINKS */
9         TIMEOUT = LONGWAIT;         /* THE CURSOR ALL THE WHILE */
10      END;
11      KEY = INPUT(KEYBOARD);        /* WHEN READY, READ KEYBOARD */
12      CALL DECODE;                  /* EXECUTE COMMAND */
13      CALL REPEATWAIT(TIMEOUT);     /* DON'T LOOK TOO SOON */
14      IF KEY = OLDKEY THEN          /* SHORT DELAY AFTER FIRST */
15        TIMEOUT=SHORTIMEWAIT;       /* TWO OPERATIONS DONE */
16      OLDKEY = KEY;
17    END;
18  CLOSE KEYBOARD_INTERPRETER;
```

Subroutines Referenced by *KEYBOARD_INTERPRETER:*

NOTREADY = a function subroutine (also referenced by *INPUT*) which is used to control an idle loop. It returns *FALSE* as its value if the selected device (in this case, *KEYBOARD*) is ready for input, and it returns *TRUE* as its value otherwise.

CURBLINK = a subroutine which "blinks" the cursor on for a fixed period of time, followed by a fixed period of "off" time. Since it must be called each time a single blink is required, this implements the "active control" feature mentioned in LIFE Line 1.

INPUT = a function subroutine which returns the current input data value for the selected device (in this case, *KEYBOARD*). *INPUT* has its own wait loop referencing *NOTREADY* – which for KEYBOARD_INTERPRETER is redundant, but is not redundant in general.

DECODE = the major subroutine of *KEYBOARD_INTERPRETER.* This routine analyzes *KEY* based upon tables and the previous inputs to the program from the operator. Using this analysis it will select the appropriate subroutine to execute. These "command subroutines" will in turn affect *LIFE* program data and the course of the *LIFE* program's execution.

REPEATWAIT = a subroutine designed to call *CURBLINK* a number of times specified by *TIMEOUT.* This implements a delay between multiple responses to the same key held down continuously.

Data (8-bit bytes) used locally by *KEYBOARD_INTERPRETER:*

OLDKEY = 8-bit value of the last previous keystroke.

NULL = 8-bit value of a null key pattern as read from the keyboard.

TIMEOUT = 8-bit value of the current repeat key time delay.

SHORTIMEWAIT = the timeout parameter used after the first delay in a multiple input of the same key. This specifies the rate of rapid motion of the cursor under manual control.

LONGTIMEWAIT = the value of the timeout parameter used for the first delay following a key entry. A longer wait is required at first to avoid false duplication of keystrokes for heavy-handed players of the game.

Data (8-bit bytes) used by *KEYBOARD_INTERPRETER* and shared with the whole program. See Table II for explanations.

GO, DONE, TRUE, FALSE, KEYBOARD, COMMAND, KEY

extent of the next run. Since this application uses 8-bit data, the limit is 255 generations of LIFE at present.

Figuring Out What the User Said

The KEYBOARD_INTERPRETER routine serves the function of controlling the input of information to these software register and to the LIFEBITS grid. The routine itself is a loop which executes over and over until the user is ready to run the GENERATION algorithm for one or more generations. The KEYBOARD_INTERPRETER terminates for one cycle of LIFE when the user inputs a "G" control command which is interpreted semantically as "GO generate N generations". The flow chart of the KEYBOARD_INTERPRETER logic is illustrated in Fig. 2, with the equivalent procedure-oriented language version shown in Fig. 3 as a detailed reference. In Fig. 2, line numbers are provided for comparison to Fig. 3.

Execution of the KEYBOARD_INTERPRETER begins with some initialization statements. The values of GO and OLDKEY are set at the start of execution (lines 3 and 4). These values will be changed during execution of the KEYBOARD_INTERPRETER based upon input data. OLDKEY is used to detect duplications of keyboard input which occur when a key is held down for continuous operations. After a given KEY is held down continuously for two operations, the repetition goes into a high speed mode with SHORTIMEWAIT controlling the delay between operations. GO is the control variable which is used to govern whether or not the loop is to continue — it is initialized to FALSE and will be changed to TRUE when the "G" user command is decoded.

Programs Are the Willing Servants of the Noble User?

Interaction of programmed computers with human beings is often a waiting game. This waiting game is aptly illustrated in the loop which checks for user input keystrokes at lines 6 to 10 of the KEYBOARD_INTERPRETER routine. The function NOTREADY (KEYBOARD) is a notational convention used to indicate a test for the keyboard ready condition. Like a ready and willing servant, the computer program keeps marking time in this loop until the user — you or I — has given it a character to digest. Two statements are included in this loop for the purpose of coordinating multiple keystroke conditions: Setting OLDKEY = NULL is used to re-establish a null history if the program ever has to wait (it never waits when keys are

Table I. ASCII Command encoding for the LIFE *application. This is an initial specification of the command codes used to control the* KEYBOARD_INTERPRETER *routine's effect. The command table locations go up by three as in Fig. 4. No addresses for the command subroutines are given yet — these will be filled in when the program is compiled for your computer. Command table locations and command characters are given as hexadecimal numbers.*

Command Table Location	Command Character	ASCII Key	Command Subroutine	Meaning of the Command (its "semantics")
00	xx	???	*DEFAULT*	The first table position is the "default" routine position, which is called when no other matching key is found in the table search.
03	47	G	*RUN*	The "run" command which sets a flag called GO in order to end the KEYBOARD_INTERPRETER and have LIFE call the GENERATION routine.
06	49	I	*INITIALIZE*	The "initialize" command to set up the screen with predetermined patterns selected by additional keystrokes.
09	53	S	*SAVELIFE*	The "save" command to dump the current screen content onto a waiting audio cassette or other mass storage device.
0C	52	R	*RESTORELIFE*	The "restore" command to recover a screen pattern previously saved by "S".
0F	58	X	*SETXLOC*	The "set X" command to explicitly set the horizontal cursor location, XCOL.
12	59	Y	*SETYLOC*	The "set Y" command to explicitly set the vertical cursor location, YROW.
15	4E	N	*SETNGEN*	The "set N" command to explicitly set the generation count for subsequent execution with the "G" command.
18	43	C	*CLEARS*	The "clear screen" command to wipe out all data and place the cursor at the center. CLEARS requires confirmation with a second S key stroke to avoid accidental clears.
1B	45	E	*LIFEDONE*	The "done" command is an E followed by an L (for End Life.) The second character confirmation is checked by LIFEDONE.

Note that the ASCII characters 0 to 9 are used to define the "current input" maintained by software in *ENTRY*. *ENTRY* may then be transferred to *N*, *XCOL*, or *YROW* by the N, X and Y commands respectively.

held down continuously). TIMEOUT = LONGTIMEWAIT re-establishes a longish debounce period between key interpretations following a series of continuous inputs. The program of course thinks that if a key is not ready upon restarting the main loop at line 6 it could not possibly be a repeat. While idling and waiting for your interactive whims, the computer program is not completely devoid of useful work. It calls CURBLINK once each time around the wait loop in order to flash the cursor actively on the screen.

Finally, after some time of unspecified duration, you make up your mind to input a key. This has one major effect upon the program: The next time around the loop at the test of the WHILE condition, a result of FALSE ends the loop. Execution then flows from the DO WHILE (line 6) to line 11 where the KEY is read from the waiting keyboard device by a subroutine called INPUT.

With KEY defined, DECODE is the next item on the agenda. DECODE is one of the major subroutines of KEYBOARD_INTERPRETER, a routine which takes KEY and compares it to a COMMAND table. The result of the COMMAND table search is execution of a "command subroutine" if a match is made, or execution of a DEFAULT routine if no match to KEY is found. Upon return to KEYBOARD_INTERPRETER (all subroutines by nature return to the caller except in very rare cases), the flow of control reaches the REPEATWAIT call using the current value of TIMEOUT.

During normal execution of single isolated commands, the TIMEOUT value is LONGTIMEWAIT — which might be chosen to be from 0.1 to 0.5 seconds. This TIMEOUT sets the minimum time between the first 3 keystrokes of a repeated sequence. But, after two long delays have been executed, the match of OLDKEY = KEY is detected at line 14 and TIMEOUT is changed to SHORTIMEWAIT allowing a speedy repeated motion case. SHORTIMEWAIT might be chosen in the 0.05 to 0.1 second range for rapid motion. The values of these two motion control constants are left unchosen for now, and can be figured out as binary integers to be used in REPEATWAIT when details of the CPU and REPEATWAIT routine are filled in. Note that if fast operation is desired immediately after the second operation of a repeated sequence, then line 13 of Fig. 3 should be moved to a location between lines 15 and 16.

In order to control the repeat logic, the statement OLDKEY=KEY is executed at line 16 so that the last input will be retained for comparison purposes the next time around.

The KEYBOARD_INTERPRETER routine finishes up with the CLOSE statement of LIFE line 18, which stands for the end of the routine and return to its caller. There is one and only one caller of this routine, the LIFE program itself, illustrated in Fig. 3 of LIFE Line #2.

It's All in DECODE of the LIFE Program

When giving the details of the KEYBOARD_INTERPRETER logic, the principle

---

While idling and waiting for your interactive whims, the computer is not completely devoid of useful work. It calls CURBLINK once each time around the wait loop in order to seductively flash its cursor on the screen.

Fig. 4. Decoding by multiple conditional tests. This method of decoding keystrokes and activating routines in software is most efficient when a small number of possible commands is involved.

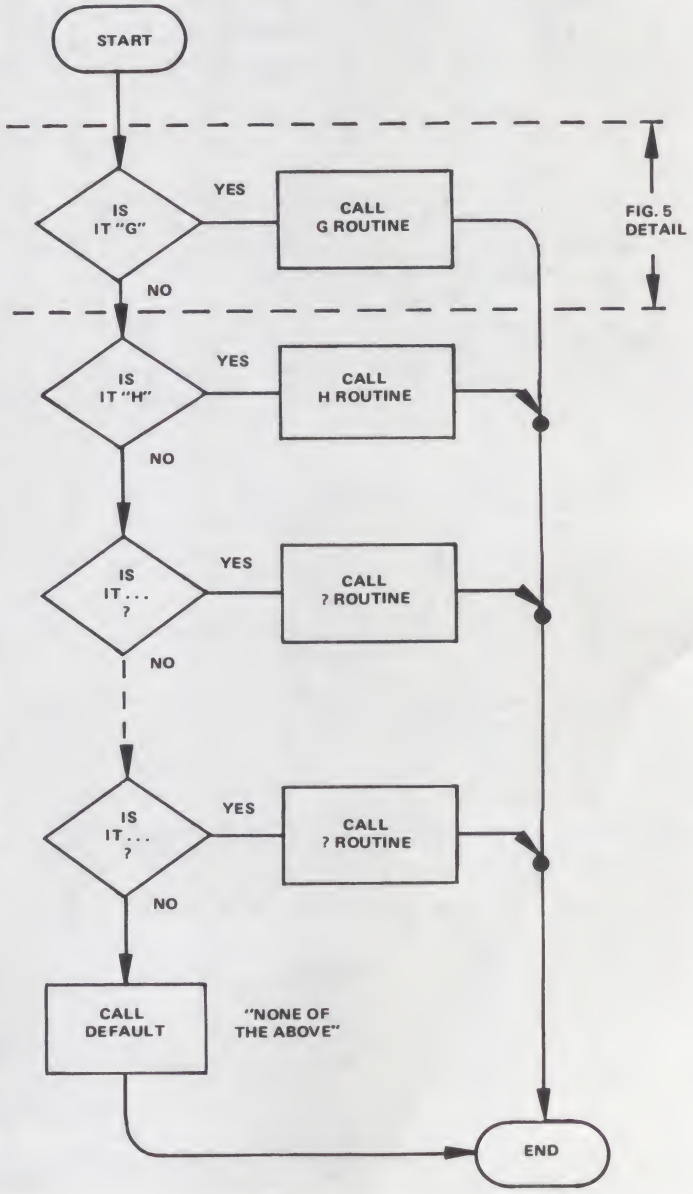


Fig. 5 Typical code for a single conditional test in the scheme of Fig. 4. The example here is using Motorola 6800 system mnemonics. This example assumes accumulator A is set up with the character being decoded.

Bytes	Mnemonic	Comment
2	CMPA # 'G	Compare A to literal
2	BNE * + 4	Branch around JSR and RTS
3	JSR GROUTINE	Call the G subroutine
1	RTS	Return from decoder rather than continue the testing

8	= Total number of bytes per test.	

This is the "generated code" of the following statements in the procedure-oriented language used for LIFE Line examples:

```
IF KEY = 'G' THEN
   DO; /* HAVE MADE A MATCH */
      CALL GROUTINE;
      RETURN; /* FROM DECODE COMPLETELY */
   END;
```

of keeping the program design locally simple results in a CALL DECODE at line 12. Whenever some subroutine is left unspecified except for its inputs (KEY for DECODE) and its outputs (a command subroutine's execution), sooner or later the details must be filled in. In designing a DECODE algorithm to fill in this set of details, there are numerous alternatives. For high order language aficionados, something called a "computed GO TO" (FORTRAN) or "DO CASE" (PL/1 family languages such as XPL or PL/M) would suffice following a table search. However, for this particular application, a slightly lower level approach is justified to conserve memory.

Two major alternatives come to mind as possible ways to map an input KEY value into the execution of a selected subroutine. The simplest (least elaborate) "straightforward" approach is the method of multiple conditional tests. This is illustrated schematically in Fig. 4's flow chart, and in a concrete form in Fig. 5's example of a segment of the typical conditional test. In this approach, each possible command code is tested in turn by the routine. Eventually, all the explicit possibilities will have been exhausted if no match is found. Then, if "none of the above" match the KEY input, a DEFAULT routine is called. The main advantage of this approach is also its disadvantage: It is a plodding and straightforward approach which squanders memory. While the code's intent is obvious, it requires — in the example of Fig. 5 — a total of 8 bytes per test.

There should be a better way — comparisons and branches are repeated in this method. The segment of generated code and its corresponding procedure-oriented language version in Fig. 5 shows four instructions which are repeated over and over but with varying data (the character being compared and the address of the subroutine). Why not put the instructions in only once and tabulate the variable data? There might be a saving of memory if this table driven approach is used instead.

Fig. 6 illustrates the concept of an alternative structure, the "command table," which will result in a lower memory requirement once the number of commands to be tested exceeds some break even point. In this concept, the *changing data* for each test is stored in the table, and the program to go along with it uses a looping technique to scan that table. The changing data for tests comprises:

— The command character. This is the keyboard code which is matched against the actual KEY input.

— The command subroutine. This is the address of the subroutine which will be called if KEY matches the corresponding command character.

The table is organized in 3-byte groups consisting of a command character followed by its subroutine address. Note that on first inspection, this form of DECODE requires only 3 bytes of storage per test versus the 8 bytes in the example of Fig.

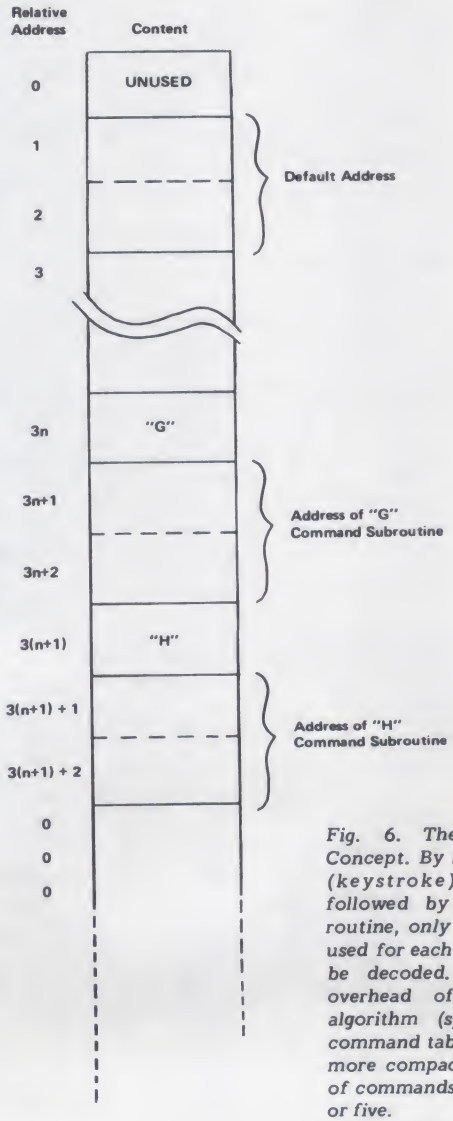


Fig. 6. The Command Table Concept. By storing the character (keystroke) being decoded, followed by the address of its routine, only three bytes need be used for each routine which could be decoded. Allowing for the overhead of a longer decode algorithm (specified once), the command table method will prove more compact when the number of commands get larger than four or five.

Fig. 7. The Command Table DECODE *routine specified in a procedure oriented language.*

```
1     DECODE:
2        PROCEDURE; /* TO FIGURE OUT WHAT USER SAID */
3        /* COME HERE WITH THE KEY TO THE COMMAND */
4        DO FOR I = 3 TO LENGTH(COMMAND) BY 3; /* SCAN TABLE */
5           IF KEY = COMMAND(I) THEN
6              DO; /* WOW!! I GOT A MATCH I GOT A MATCH! */
7                 I = I + 1; /* POINT TO ADDRESS ENTRY */
8                 CALL CALLX(COMMAND(I));
9                 /* NOTATION FOR CALL OF SUBROUTINE, INDEXED */
10                RETURN;
11                /* THIS FORCES EXIT FROM DECODE */
12             END;
13          /* ONLY GET HERE IF NO MATCH IN TABLE */
14          CALL CALLX(COMMAND(1)); /* CALL DEFAULT FROM TABLE */
15       END;
16    CLOSE DECODE;
```

Data (8-bit bytes) used locally by *DECODE:*

I = temporary used for loop control and indexing.

Data (8-bit bytes) used by *DECODE* but shared with the whole program. For details see Table II.

COMMAND, KEY

Subroutines referenced by *DECODE:*

DECODE does not use any "real" subroutines, but does use the following two notational conventions which look like subroutines:

LENGTH(COMMAND) stands for the length (in bytes) of the *COMMAND* table. When you know what it is, you put in the value.

CALLX(X) is used to denote using the two bytes starting at the address X as the address of a subroutine to be called. This is an *indexed* subroutine call effectively. For a Motorola 6800 CPU, this would be performed by an LDX instruction indexed off the *COMMAND* table position, followed by a JSR instruction with the indexed addressing mode.

5. For a 10 command table, this would be a 50 byte saving at first inspection. However, the 50-byte figure does not take into account the longer looping routine required to scan the table and indirectly jump when a match is found. But for 10 commands (the number found in Table I) this 50 byte saving potential goes a long way. I expect the actual DECODE routine of the table driven variety to be considerably less than 50 bytes in length when it is generated for the 6800 system instruction set used as the straw-man in Fig. 5. I'll leave the final conclusion on that to a later LIFE Line.

There is an additional advantage to be obtained from the table driven method. This is an advantage which concerns some of the finer points of programming: The table driven method results in "pure code" in which potentially variable data is completely segregated off in the table. This achieves an often desirable end of separating data from instructions. In the multiple conditional test version, the data of the DECODE is embedded right in the instruction stream, both as the literal value of the character being tested and as the address of the routine being selected. If I want to modify the multiple conditional test version, I must certainly recompile or reassemble the whole routine (a pain in small systems work). In contrast, to modify the table driven version, I only have to alter the table itself, and the variable which specifies the table's length. But this is a minor point in addition to the major memory conservation argument for the table driven approach.

The actual algorithm for DECODE is shown in a procedure-oriented language in Fig. 7. The scan of the table is a DO FOR loop with

Notes on Notation:

Concerning Indentation: The listings of procedures for the LIFE program make use of an indentation convention to help show the structure of the routines. The significance of the indentation is that it shows the opening and closing of various local software constructions and in so doing helps convey the meaning of the program to human readers. Note how the statements from line 7 to line 11 of DECODE in Fig. 7 are indented one level compared to the DO (line 6) and END (line 12) statements. This indentation shows that lines 7 to 11 are part of the DO . . . END construction which is executed if the test on line 5 gives a true result.

The notation "/" followed by arbitary remarks and then a "*/" is the "comments" convention used in these examples. This convention is stolen from the PL/1 family of languages.*

Concerning names of variables: With each procedure specified in LIFE Line, data is separated into two categories: Local data *is used only within the procedure question. Local data may have a name which duplicates names used in other procedures, but is always qualified by its local nature. Thus "I" in* GENERATION *(Fig. 6, LIFE Line 2) is a different data location in memory than the "I" in* DECODE *(Fig. 7, LIFE Line 3).* Data shared with the rest of the program, *which is often called* global data *in programming terminology, is in contrast defined universally for* LIFE. *Global data is summarized for* LIFE *in Table II. Thus whenever* KEY *is referenced (as in* KEYBOARD_INTERPRETER *or in MOVECURS) the same data is intended, since these have been classified as shared or global in the notes accompanying the program listings.*

the index, I, running from 3 (the first entry is reserved for the default) to the length of the table by 3. When a match is found, the 16-bit address in the table is used for an indirect subroutine call (lines 7 and 8). For a Motorola 6800 system, this would be accomplished by an indexed JSR instruction after loading the index register from the table. When the selected command subroutine returns to DECODE (as would any well structured subroutine in the same circumstance), the RETURN statement is executed causing an exit from DECODE and resumption of the KEYBOARD_INTERPRETER at the calling point. If no match is found, the loop eventually runs out and line 14 of Fig. 7 is reached, where the DEFAULT routine is called. This is shown notationally in a general purpose form with reference to the command table, but in generating the code for the statement of line 14, a simple call to DEFAULT might be substituted. (If the generality of the DECODE routine is to be preserved for possible use with other command tables, this optimization would not be possible.)

What about data for the COMMAND table? Table I provides a preliminary answer to this question by giving a list of command table entries including relative location, the corresponding character code, the ASCII key which invokes the command, the name of the subroutine and a verbal description of the subroutine. This table will be used as the basis for creating a detailed data table when the actual programs of LIFE are generated for a particular computer in a future LIFE Line. For now, Table I serves to list the areas which remain to be covered in the discussion of the KEYBOARD_INTERPRETER and all its subroutines.

LIFE Line 4 will continue the presentation of the KEYBOARD_INTERPRETER portion of the LIFE program. To fill out the remaining portion of the Tree of LIFE, the next installment includes the integration of graphics control commands into the KEYBOARD_INTERPRETER and the first hardware details of LIFE – a simple circuit which combines an ASCII keyboard input with the special purpose controls for an interactive cursor. ■

Does Anyone Know What Happened to Robert T. Wainwright?

This series of articles inadvertantly duplicated the name of Robert T. Wainwright's LIFELINE newsletter, published through 1973. Thanks to Bob Albrecht of People's Computer Co. for sending us his copy of LIFELINE's last issue. Does anyone know where Mr. Wainwright is now (he's no longer at the address given by Charles A. Dunning Jr. in the Letters column), and is LIFELINE still being published?

Table II. Global Data. Data which is shared by an entire program or application is often called "global". The word global is used to indicate the widespread effects of such data in the program's execution. Many procedures will alter and change such data. This table summarizes the global data variables of the LIFE *application as used in procedures given in LIFE Lines #2 and #3.*

COMMAND = the table of commands interpreted by *DECODE*, containing the ASCII codes of command keys and the addresses of the appropriate command subroutine. The format of this table is illustrated in Fig. 6. The information content, in preliminary form, is found in Table I.

DONE = the variable used to control continued execution of the main *LIFE* routine (see LIFE Line #2, Fig. 3).

ENTRY = the entry register used to receive numeric ASCII digits, after weighting the previous value in a BCD fashion. While the entry to *ENTRY* of new digits is done in a BCD manner (multiplying by 10 then adding the digit's value) the content of *ENTRY* is a binary number of 8-bit precision with values 0 to 255 and is thus not itself BCD. (BCD = "binary coded decimal.")

FALSE = the value "0" (00 hex, 000 octal, 00000000 binary). This name is used to indicate the software equivalent of a hardware gate input wired to logical zero.

GO = the flag (value is *TRUE* or *FALSE*) which controls continued execution of *KEYBOARD_INTERPRETER*.

KEY = the 8-bit data area which receives keyboard inputs.

KEYBOARD = the logical unit number of the keyboard I/O device. This is a bit pattern which specifies the device one is talking to.

LIFEBITS = the object of the whole exercise – an array of 64 by 64 *bits* stored as 64 by 8 *bytes*.

N = the variable used to control the number of generations to be evolved by LIFE before returning to *KEYBOARD_INTERPRETER* graphics control.

NCMAX = current maximum column index of live cells.

NCMIN = current minimum column index of live cells.

NCOLMAX = maximum column index of live cells for active area optimization.

NCOLMIN = minimum column index of live cells for active area optimization.

NROWMAX = maximum row index of live cells for active area optimization.

NROWMIN = minimum row index of live cells for active area optimization.

NRMAX = current maximum row index of live cells.

NRMIN = current minimum row index of live cells.

TEMP = 2 by 8 array of bytes containing two 64-bit rows of cells.

THAT = previous line copy index to *TEMP* used in *GENERATION* (see LIFE Line #2, Fig. 6). THAT should always have a value of 1 or 0, opposite of THIS.

THIS = current line copy index to *TEMP* used in *GENERATION* (see LIFE Line #2, Fig. 6). THIS should always have a value of 0 or 1.

TRUE = the value "255" (FF hex, 377 octal, 11111111 binary). This name is used to indicate the software equivalent of a hardware gate input wired to logical one.

XCOL = the current cursor position in the horizontal (column) direction.

YROW = the current cursor position in the vertical (row) direction.

LIFE Line 4

Integrating graphics control commands

Carl Helmers

In LIFE Line 3, the design of the DECODE routine of the LIFE program was presented. DECODE is designed as a table driven mechanism for selecting one of several subroutines which carry out the functions of the LIFE program's KEYBOARD_INTERPRETER. However if you examine table 1 of LIFE Line 3 (see p. 51 of BYTE #4), you will note one conspicuous and intentional lack: There are no routines which process the interactive graphics commands required to set up LIFE patterns on the scope display. Yet in LIFE Line 1, several special purpose keys were introduced as manual inputs for cursor motion control and data definition purposes. Where is the missing part of the program which interfaces these keys? What are the hardware implications of requiring a special keyboard? Answers to these questions are the major concern of LIFE Line 4. Integrating the graphics control commands is a combined hardware and software topic. The software is that of the DEFAULT routine that interprets several keyboard inputs not handled by DECODE; the hardware consists of the design of a special keyboard interface to automatically switch between an ASCII keyboard's 7-bit parallel output code and the LIFE graphics control keypad.

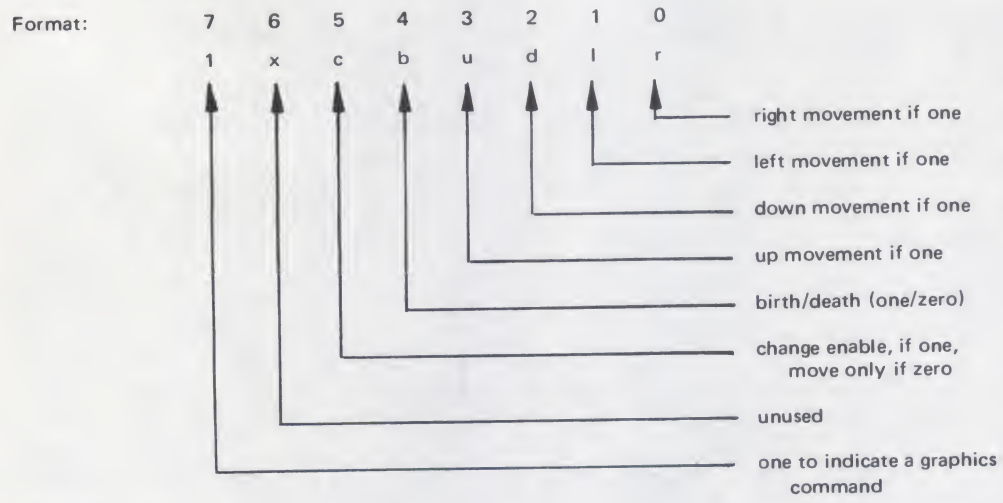
The main requirement for LIFE cursor motion and data control is that one, two or three of the input keys can be depressed at the same time. This capability is needed in order to specify all the possible combinations of motion control and optional cell birth or death data inputs. The individual motion control possibilities (one key at a time) are the movements in four principal directions: up, down, left or right. When two motion control keys for perpendicular directions are selected at the same time, diagonal motion is the desired result. With either form of motion control, entry of data can optionally be performed by depressing either the birth key or the death key at the same time. Thus as many as three keys may be sensibly pressed simultaneously when entering data.

The large number of combinations possible for the six bits which will be needed for six switches strongly argues against making the software use a table driven algorithm such as DECODE. This is the reason why no cursor motion and data entry commands are found in table 1 of LIFE Line 3. Since each bit of the parallel information from the motion control switches can have an independent meaning, a specially programmed determination of motion control actions uses less memory than the huge table which would be required for all the combinations. Thus handling of motion control is left to the DEFAULT routine which is called by DECODE when it fails to decode one of the commands in table 1 of LIFE Line 3. (DEFAULT also handles ASCII numeric inputs, as you'll see a bit later in LIFE Line 4.)

Graphics Control Hardware Considerations

For the hardware of LIFE, how can the need of this special set of input codes be reconciled with the need to input ASCII via the same eight-bit input port? One answer lies in the choice of an eight-bit format in which the most significant bit determines what lies in the low order seven bits. With this format, one state of the most significant bit indicates when an ASCII code is present in the low order; the other state of the most significant bit indicates when graphic control keyboard information is in the low order. This choice of format is supported in hardware by the addition of a simple interface module which uses seven integrated circuits to switch between data sources and debounce the motion control keyboard.

Graphics Control Commands



ASCII Keyboard Input

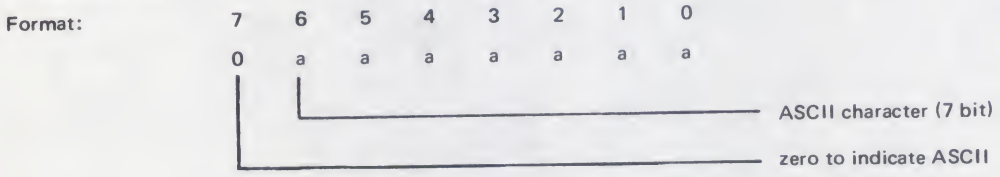


Figure 1: Data formats for graphic control commands and ASCII keyboard input.

The combined ASCII and control data format is illustrated in figure 1. When the value of bit seven of the interface is read as one, the programming of the DEFAULT routine will always be entered and the low order bits will be analyzed as graphic control information as shown by the upper diagram in figure 1. The low order bits zero through three represent the individual key states of the motion control switches and the next two bits, four and five, are encoded with information on data entry from the birth and death switches. If the value of bit seven is read as a logical zero, the program will interpret the ASCII value of the low order bits through the DECODE routine of LIFE Line 3, or through the DEFAULT routine if the command is not in the table which drives DECODE.

The hardware needed to implement this special interface is shown in figure 2. The interface consists of a two way data selector (IC6 and IC7) which determines whether the eight bit pattern presented to the system bus interface comes from the cursor motion control keyboard or from the ASCII keyboard. The ASCII data is routed straight to the data selectors from a jack (J1) which receives a cable which connects to the keyboard unit. (The LIFE Line system prototype is currently using one of the surplus Sanders 720 keyboards described in BYTE #1.) The graphic control information is derived through jack J2 from the special keyboard via the 7474 flip flops IC1, IC2 and IC3. These D flip flops are being used as set reset flip flops by grounding the clock line and employing the preset (PRES) and clear (CLR) inputs for data and keyboard acknowledge functions respectively. The flip flop outputs for bits zero to three go directly to the data selector to define cursor motion inputs. The flip flop output for bit four (birth switch) also is directly connected to the selector. However, bit five of the selector's cursor motion inputs is taken from NAND gate IC4D which encodes a CHANGE ENABLE signal when either birth or death data input is indicated. (Note that the user of the LIFE cursor motion control keyboard is on his honor not to push both birth and death keys simultaneously — with this encoding logic, birth always locks out death.) One item derived from the cursor control keyboard is a key pressed signal produced by 7430 NAND gate IC5. This signal is inverted by IC4C and used to control the data selector: If any key on the cursor motion control keyboard is pressed, the ASCII keyboard will be locked out;

A flip flop with preset and clear inputs can be used in place of a hand-wired set reset flip flop.

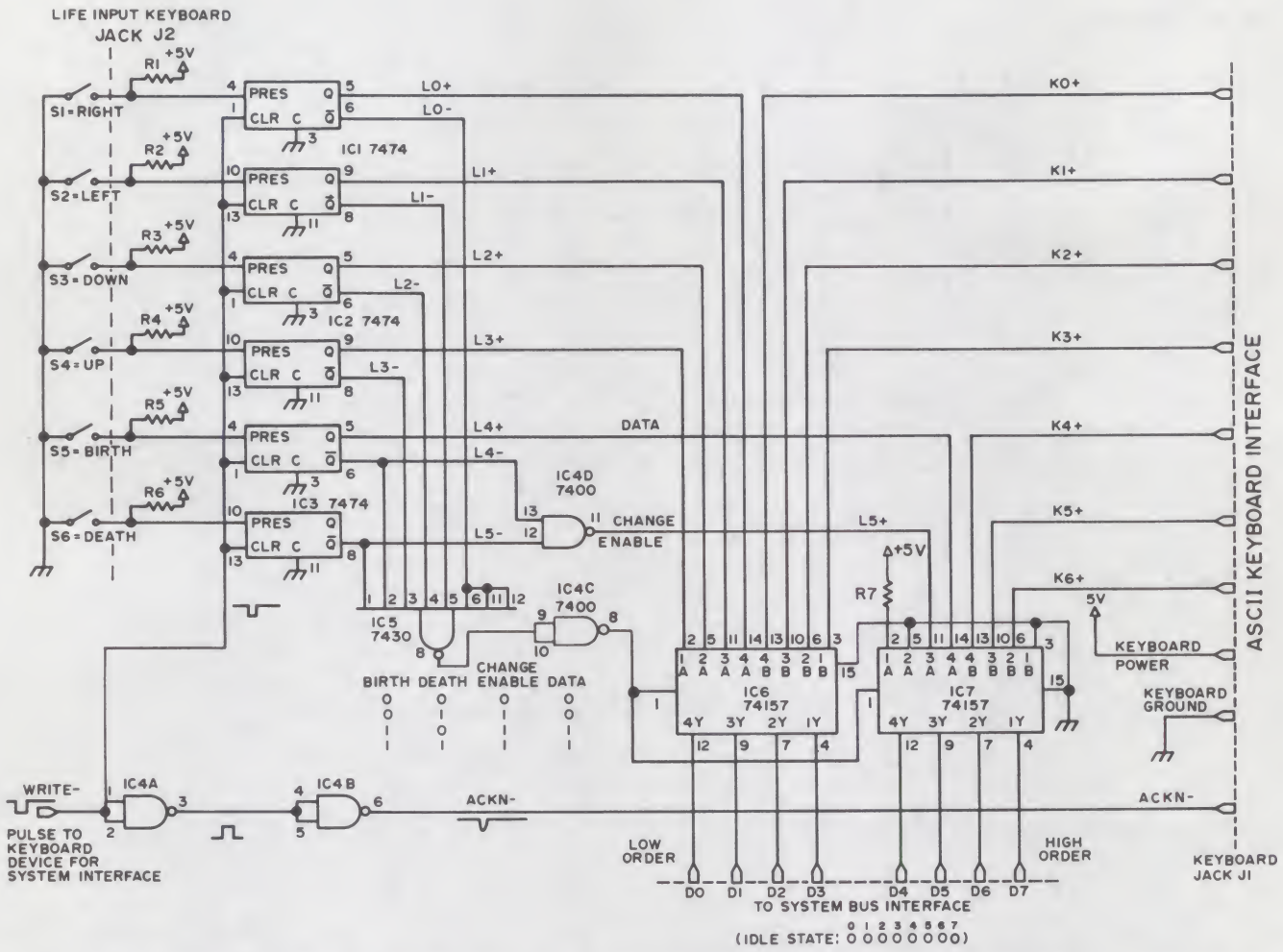


Figure 2: The logic diagram of a keyboard interface which implements the formats of figure 1. Resistors R1 to R7 are TTL pull up resistors. The value are not critical, and may range from 1 K0 to 10 K0, ¼ W.

otherwise the ASCII keyboard is connected and the cursor motion control keyboard is ignored. Note that the cursor motion input *has priority* over the ASCII keyboard since it controls the data selector.

Finally, to complete the interface logic sections A and B of IC4 are used to buffer the computer-generated keyboard WRITE- signal which occurs when the computer writes data into the keyboard location. This signal is used to reset the graphics control flip flops. The buffered version of the signal (pin 6 of IC4) is used to drive the acknowledge line of the ASCII keyboard unit. A separate buffer is recommended due to the unknown loading of the ASCII keyboard device. In LIFE Line's design of a program, the logic of the KEYBOARD_INTERPRETER procedure is used to manipulate the interface.

What is not shown in figure 2 is the actual system bus interface. The design of such an interface must be done consistent with a given computer's data bus. In the prototype system for LIFE Line, a Motorola 6800 computer's data bus, buffered by National DM8833 Tri State bus transceivers is used. The interface thus consists of two DM8833's used to drive the bus, plus the address selection logic needed to detect the address of the keyboard and produce the bus enable signal as well as the WRITE- signal. For a computer based upon a kit, the input port logic will be in a standard form designed by the kit manufacturer. What is needed is a parallel input port, which might already exist if your computer kit comes with a keyboard and parallel interface.

Notes on Assembly

The prototype version of the graphic control keyboard is illustrated in photo 1. The keys were made from conventional magnetic reed switches obtained from keyboard units found at a computer auction. Any single pole single throw keyswitch can be used; options on mounting are left to the ingenuity of the builder. The arrangement of keys shown in photo 1 is designed so that the cursor motion controls are at the top in

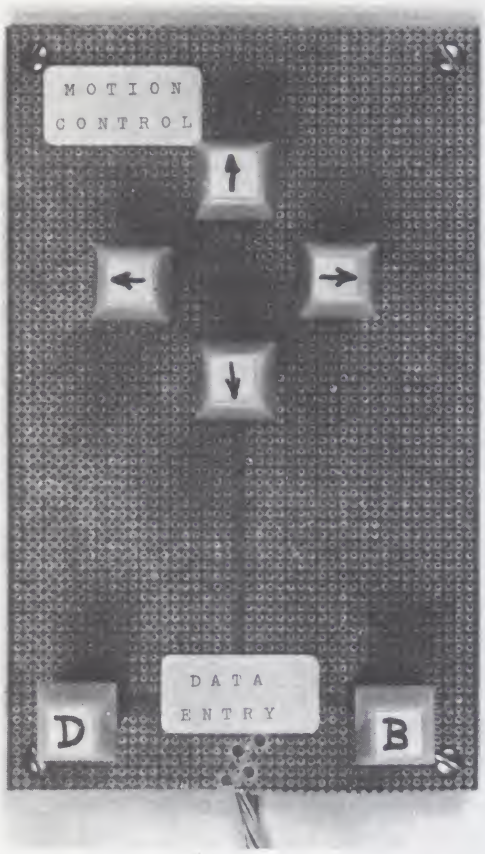


Photo 1: The graphics control keyboard of the LIFE Line prototype system. The group of four switches with arrows are cursor motion control keys. The two switches with captions "B" and "D" are the birth and death data keys, respectively.

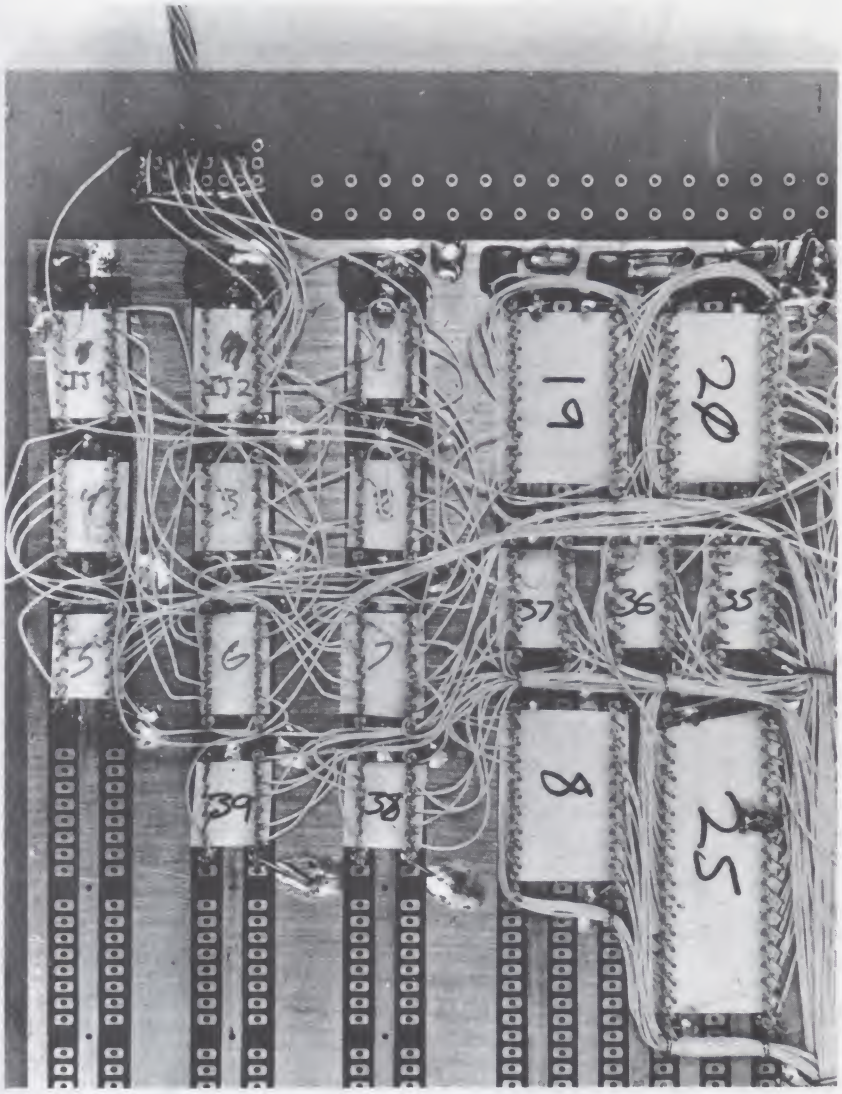


Photo 2: Detail illustrating wire wrapped assembly of figure 2 using a general purpose prototyping board.

a group of four. The arrows were applied using small pieces of self-sticking address labels of the type often used by computer centers. The two isolated switches at the bottom of this arrangement are the birth (B) and death (D) keys. The wiring of the keyswitches to the computer is accomplished through a multi-conductor bundle of wires trailing away at the bottom. This cable terminates in a dual-inline header plug which fits into a socket on the wire wrap board containing the computer and interface. Photo 2 illustrates the wire wrap wiring of the interface logic in the LIFE Line prototype system.

Using The Control Information

The purpose of the interface hardware is to combine two keyboards into a single input port with software distinguishing "who called" on the basis of the format shown in figure 1. How does the LIFE software handle this data format? Recalling the presentation in LIFE Line 3, the DEFAULT routine is called by the DECODE routine whenever DECODE cannot match an input from this port to an entry in the COMMAND table. Decoding of the graphics control format and ASCII numeric characters is left to the DEFAULT routine because of the systematic nature of these inputs.

How is this decoding done? One answer of course lies in the design of the DEFAULT routine. DEFAULT is specified in a procedure-oriented language in figure 3. Basically the DEFAULT processing follows one of two paths of execution according to the high order format identifier bit, bit seven of the formats illustrated in figure 1. The input data from the interface is passed to DEFAULT in the variable KEY which is set at line 11 of KEYBOARD_INTERPRETER (see LIFE Line 3, figure 3). The high order bit of KEY is tested by the AND operation of line 3. The masking bit string 10000000B

```
1   DEFAULT:
2     PROCEDURE;
3     IF (KEY AND 10000000B) NOT = 0 THEN
4       DO;  /* GRAPHICS CONTROL INPUT CASE  */
5         CALL MOVECURS;  /*  MOVE CURSOR PER INPUT  */
6         IF (KEY AND 00100000B) NOT = 0 THEN
7           DO;  /* CHANGE IS INDICATED  */
8             IF (KEY AND 00010000B) = 0 THEN
9               CALL LPUT(XCOL, YROW,0); /* TURN OFF POINT  */
10            ELSE
11              CALL LPUT(XCOL,YROW,1);/* TURN ON POINT  */
12            CALL DISPLAY; /* SEND UPDATED LIFEBITS OUT  */
13          END;
14      END;
15    ELSE
16      DO;  /* ASCII NUMERIC DEFAULT CASE  */
17        NUM = KEY - 30H;
18        IF NUM >9 THEN NUM = 9;
19        ZUM = 0;
20        DO FOR I = 1 TO 10;  /* MULTIPLY = REPEATED ADD */
21          ZUM = ZUM + ENTRY;
22        END;
23        ENTRY = ZUM + NUM;
24        /* ENTRY NOW HAS NEXT DECIMAL DIGIT ADDED IN WITH  */
25        /*     A BCD SHIFT BY ONE PLACE
26      END;
27    CLOSE DEFAULT;
```

selects only the high order bit of KEY so that the result of the masked test will be zero if bit seven is zero, non zero if bit seven is one. If the result of the AND is not equal to zero, the graphics control case will be executed: the DO . . . END group extending from line 4 to line 14. If the result of the AND is zero, ASCII input is present so the character is forced into a numeric entry interpretation. The ELSE DO . . . END clause of lines 15 to 26 handles this alternative.

Graphics Control Processing

The processing of the graphics control format is not at all complicated. A procedure, called MOVECURS is executed first to decode the four low order bits of the graphics control format and adjust the cursor position.

MOVECURS is specified in a procedure-oriented language in figure 4. This routine contains four IF statements which test the four motion control bits. Motion is achieved for each logical one bit by simply adding or subtracting one from the corresponding cursor position variable XCOL or YROW. Note that this software takes care of an invalid combination of up and down (or left and right) in a unique way: nothing happens. If contradictory commands are input, the

Data (8-bit bytes) used by *DEFAULT* at this level:

NUM = temporary data byte used to hold a BCD digit for conversion to binary.

ZUM = temporary data byte used to form the product when ENTRY is shifted left 1 BCD digit by multiplication with 10, lines 20 to 22.

Data (8-bit bytes) used by *DEFAULT* but shared with the whole program.

KEY, XCOL, YROW, ENTRY

Subroutines Referenced by DEFAULT:

LPUT . . . Routine (used also by FACTS_OF_LIFE) which places the bit value of the third argument at a location specified by the first two arguments. Thus lines 9 and 11 define a new value for the bit at XCOL and YROW in the LIFEBITS matrix.

MOVECURS . . . The routine (see Fig. 11) which moves the cursor up, down, left or right depending upon the motion control switches which are read into the low order bits of KEY.

DISPLAY . . . The routine which copies LIFEBITS to the graphics output device for viewing.

Figure 3: The DEFAULT routine specified in a procedure-oriented language.

```
1    MOVECURS:
2       PROCEDURE;
3       /* MOVE THE CURSOR BASED UPON THE FOUR LOW ORDER BITS */
4       /* OF THE GRAPHICS CONTROL CHARACTER INPUT            */
5       IF (KEY AND 1000B) NOT = 0 THEN
6          YROW = YROW + 1;
7       IF (KEY AND 0100B) NOT = 0 THEN
8          YROW = YROW – 1;
9       IF (KEY AND 0010B) NOT = 0 THEN
10         XCOL = XCOL – 1;
11      IF (KEY AND 0001B) NOT = 0 THEN
12         XCOL = XCOL + 1;
13      /* NOW, IF THE SELECTED KEY WAS ON, THE APPROPRIATE */
14      /*  CURSOR POSITION REGISTER HAS BEEN CHANGED      */
15      CLOSE MOVECURS;
```

Data (8-bit bytes) used by DEFAULT but shared with the whole program. See Table 2 of LIFE Line 3, BYTE # 4, pg. 55, for details.

KEY, XCOL, YROW

Figure 4: The MOVECURS routine specified in a procedure-oriented language.

Decoding "who called" is done in software by the DEFAULT routine.

cursor position variable in question is both incremented and decremented with a net result of no change. Remember also that time delays are built into KEYBOARD_INTERPRETER to govern the speed of changes when keys are held down continuously.

Upon return from MOVECURS with the newly updated position, the remaining portion of the graphics control processing consists of program logic to test for data entry. If the change enable bit (bit five) has a value of one, a change is indicated. Then if the data bit (bit four) is zero, the current position in LIFEBITS is turned off, indicating a death; if the data bit is one, the current position in LIFEBITS is turned on, indicating a birth. Graphics control change processing is completed at line 12 when DISPLAY is called to put the new data out on the display screen.

Numeric Default Processing

In the alternative DEFAULT processing case of an ASCII character which is not recognized by the DECODE routine, the program will *assume* numeric entry. In effect what this means is that any unrecognized non-numeric ASCII character will cause invalid data to be placed in the ENTRY register of the software since this little routine uses brute force to extract a numeric meaning. At line 17, a value of hexadecimal 30, denoted 30H, is subtracted from the key code. Since valid numeric ASCII characters run from hexadecimal 30 to 39, this will result in data running from 0 to 9 for valid numeric codes. The test of line 18 excludes invalid codes by forcing a 9 value. (Unsigned arithmetic is assumed here so that all 8-bit integer values not in the range 0 to 9 will be larger than 9.) Then the previous ENTRY value is multiplied by 10 using a repeated addition loop at lines 20 to 22. The new entry digit value is then added in to the low order at line 23. Note that ENTRY is a binary number, but that the digit being defaulted is entered with a decimal weighting. (For multiplication, an alternative to repeated addition in this case would be to observe that 10 x = 8 x + 2 x. Thus using three arithmetic left shifts both twice and eight times the original ENTRY could be obtained and summed producing the 10 x product.)

After execution of one or the other of the two paths determined by the format bit of the data in KEY, DEFAULT reaches its CLOSE statement and returns to DECODE.

Where Does LIFE Stand?

In the course of LIFE Line through this installment, the structure of the LIFE program has been the major topic. LIFE Line 4

has introduced the first hardware considerations — the special keyboard — as a requirement in the specification of graphics control processing. LIFE Line 5 will continue the software theme by completing the initial specification of the LIFE program design exclusive of the RESTORELIFE, SAVELIFE and INITIALIZATION procedures which together form a major software subject in their own right. LIFE Line 5 will cover the DISPLAY, RUN, SETXLOC, SETYLOC, LIFEDONE, and SETNGEN procedures as its main theme. Then the series will turn to the hardware of the LIFE system prototype in more detail, to provide a basis for the generation of actual executable programs which will run on the prototype system. The first major phase of the LIFE Line project will be completed when it is possible to draw a LIFE pattern on an oscilloscope output device using the cursor motion control keyboard, then initiate the pattern evolution according to the facts of LIFE as presented in LIFE Line 1.

The second major phase of the project will be the addition of the data management hardware and softwave facilities of the SAVELIFE, RESTORELIFE and INITIALIZE procedures. These facilities will enable the construction of initial patterns from "standard parts" saved on a mass storage device. As always, the aim of the entire series of LIFE Line articles is to show how the bits and pieces of hardware and software design fit together to produce a working application system. ■

A bibliography of Scientific American *information on LIFE (all references are to Martin Gardner's "Mathematical Games" column).*

October 1970: page 120. This is the original LIFE article, including the definition of the Facts of LIFE, and illustration of numerous fundamental patterns.

November 1970: page 118. Answers to several questions posed in the first article on the subject, including definition of the several varieties of "spaceships."

January 1971: pages 105, 106 and 108. Continued progress on the LIFE front including answers to several unsolved questions and results of a flurry of computer LIFE activity.

February 1971: Special "Mathematical Games" article on "cellular automata theory."

March 1971: pages 108 and 109. Short note about progress made by John Conway and R. William Gospers, plus illustration of a large scale flip flop pattern which is delicately balanced and easily destroyed by minor disturbances such as impact of a glider.

April 1971: pages 116 and 117. Examples of fuses, the five cell cross series, and announcement of Robert T. Wainright's LIFELINE newsletter.

November 1971: page 120. Short note on continued progress at the MIT AI Laboratory.

January 1972: page 107. The discovery of the "eater" by Bill Gospers at MIT.

This is an essential list of readily available information on the LIFE game which interested readers can research in any complete public or university library.

An Aside Regarding the Ultimate LIFE

LIFE on a 64 x 64 grid is an achievable project for the home brew computer enthusiast. But it is far from the ultimate. My thanks to Bob Clements of Lexington, Massachusetts for arranging a demonstration by R. William Gospers, Jr., at the MIT Artificial Intelligence Laboratory one recent Saturday evening. When LIFE was first widely publicized by Martin Gardner in his October 1970 Mathematical Games column in *Scientific American*, it helped set up a flurry of research work on the subject.

Bill Gospers and his associates at the MIT AI Lab took the definition of John Conway's game and began constructing a highly efficient LIFE system running on a Digital Equipment Corporation PDP-6 computer with a high resolution 1024 x 1024 position oscilloscope display. This research tool was used by the MIT people to generate numerous mathematically interesting LIFE patterns. These include such fundamental discoveries as glider guns, space ship factories, a binary transcendental number calculator, and a Turing machine pattern. The ultimate climax of the evening's demonstration was Bill's demonstration of a disproof — by example — of John Conway's conjecture that no LIFE pattern could grow without limit. The particular example he used is a colossal moving glider gun factory — a pattern which leaves a trail of active glider guns behind it as it travels slowly to the right on the display screen. This pattern fills the plane of the LIFE matrix with cells, and the number of active cells increases in proportion to the square of the number of generations the pattern has lived. After an arbitrary length of time, an arbitrary region of the plane will be filled with glider patterns emanating from the residue of glider guns produced by this LIFE machine.

The programs which form the MIT LIFE system are run on equipment far beyond the range of price a home brewer could consider — but with the advances in technology it is now possible to make a LIFE system which demonstrates many principles without breaking budgets.

Applications

Total Kitchen Information System

Ted M Lau
7740 P Chalmette Dr
Hazelwood MO 63042

I have become a hateful person just because my grocery list is unsorted.

I want to outline a plan for a total kitchen information system (TKIS) suitable for implementing on a home computer. This outline is the first step in the development of TKISs of arbitrary complexity from the simplest inventory modules to artificial intelligence modules (such as those suggested by Richard Gardner in the October 1975 issue of BYTE). The functional approach used here should allow the reader to plan a complex system using small and manageable, "byte-sized" pieces, or to interface independently developed modules.

This project began as a gripe list my wife and I compiled after many frustrating experiences in the kitchen; throwing out spoiled food we'd forgotten in the refrigerator, abandoning a recipe for lack of a key ingredient, reeling with confusion after reading pages of grocery specials, neither being able to remember an appealing recipe nor to find it among all our cookbooks, and so on.

Hierarchy Chart

Figure 1 shows the functions to be performed by a TKIS, structured in hierarchic fashion — meaning that every function is made up of several subfunctions, each function box performs one general task which can be divided into several specific tasks, and so on. This chart differs from a flowchart in that the function boxes are not necessarily performed in left to right order, nor are the conditions for execution given. The hierarchy (H) chart attempts to outline *what* a system must do, but not *how*, *when*, or *if*.

Each rectangle in the chart represents a transformation of some inputs into some outputs. For example, box 1.0 takes grocery prices from several markets and spits out a list of bargains to be scheduled into meals. Box 2.0 accepts a list of on-hand perishables, in addition to the output from 1.0, and yields a schedule of meals. Box 3.0 transforms the meal schedule into the food needed. Box 4.0 transforms raw, separate foodstuffs into cooked fare. Box 5.0 turns a meal into leftovers and garbage, and 6.0 turns garbage into cleanliness.

Notice that I've ignored inputs that appear unchanged as outputs, such as the recipes consulted to plan the meal (2.0): They are brought in at the beginning and returned unchanged at the end of the task. These unchanged or rarely changed inputs are the tables and files referenced by the function boxes. These tables and files appear to be internal to the boxes, and therefore can be ignored for the time being, thus allowing me to concentrate on TKIS functions. Though file design itself can be put off, provision must be made for the creation and maintenance of this data (7.0). Examples are the writing of recipes onto blank recipe cards, or the (presumed) structuring of a previously unstructured human brain nerve net to respond to a low price in hamburger.

Notice that action boxes (3.5, 4.5, 5.0) are mixed in with thought boxes. The H chart attempts to completely describe all the functions involved in operating a kitchen, whether primarily physical or primarily informational. While no one can seriously attempt to computerize these physical tasks at the present time, we must remember that all physical processes have informational

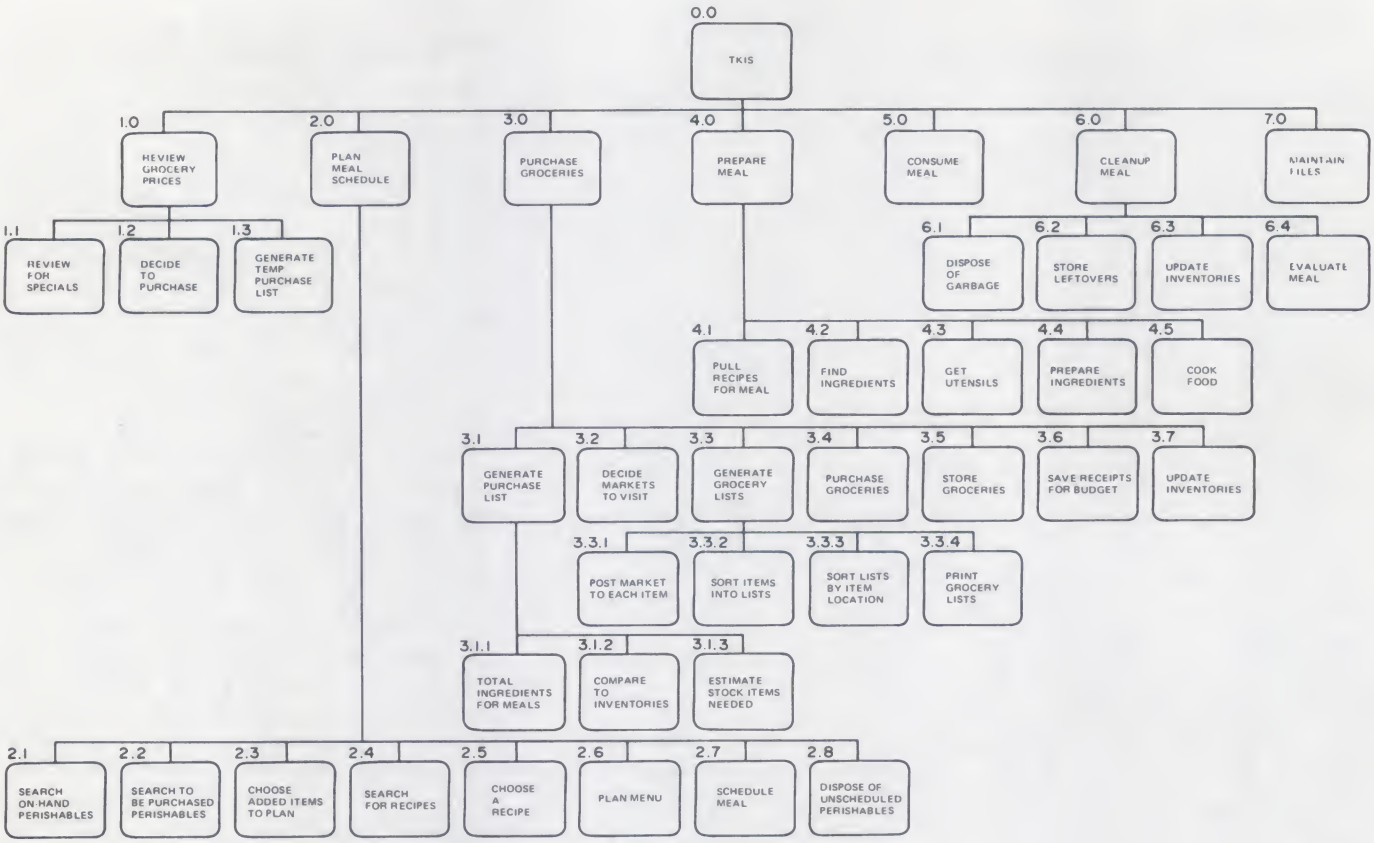


Figure 1: Functions of Total Kitchen Information System.

components (and are thus fair game for computer enhancement), and that any distinction between physical and informational is strictly provisional (and is subject to erosion as computers expand their capabilities to manipulate objects, as in robotics and automation). So the H chart incorporates into its comprehensive structure both modules that are subject to present data processing solutions and modules that must wait for future technology. (Readers will identify box 4.0 as the voice responsive vending machine in the rec room of the Star Ship Enterprise.)

Notice that the H chart says nothing about computers. It describes my conception of a very rigorous manual system that could be performed with paper and pencil. It purposely steers clear of computer concepts to allow you to be flexible in making software and hardware design decisions. To paraphrase: "Hardware and software may pass away, but functions endure." The tasks to be performed by the TKIS remain unchanged from one system configuration to another.

The H chart functionally describes my view of what must be done to get meals on the table. It is triggered by specials and perishables in that it tries to cut costs by planning meals using bargains, and to reduce wastage by scheduling perishables in timely fashion. Specifically, TKIS plans to review a large number of grocery item prices and to call attention to those that meet a *specials* criterion specified by the developer (1.1). It plans to call attention to items in inventory whose perishable date falls within the next meal scheduling period (2.1). It plans to retrieve recipes based on key ingredients and other characteristics such as casserole, quick-meal, Chinese, price-per-serving, nutritional values, etc., (2.4), and to reveal the recipe ingredients not on hand, or to reveal only those recipes whose ingredients are all on hand. It plans to help the kitchen operator decide which markets to visit by simulating the expenses of buying at various markets, including labor time and gasoline costs (3.2). It plans to calculate the quantities of ingredients needed for recipes with adjusted servings (3.1.1, 4.1). It plans to collect menu and recipe evaluations (yum, good or echch) (6.4), along with keeping past meal schedules and market receipts, for future analysis in planning menus, purchasing foods, budgeting, and dietetics. It even plans to sort the items on each grocery list into store location order, so that by walking through the store in a prescribed way the items will be encountered in order (3.3.3). This is big on my gripe list: I hate carrying a pencil to mark the groceries I buy, and I also hate chasing all over the store to find the last

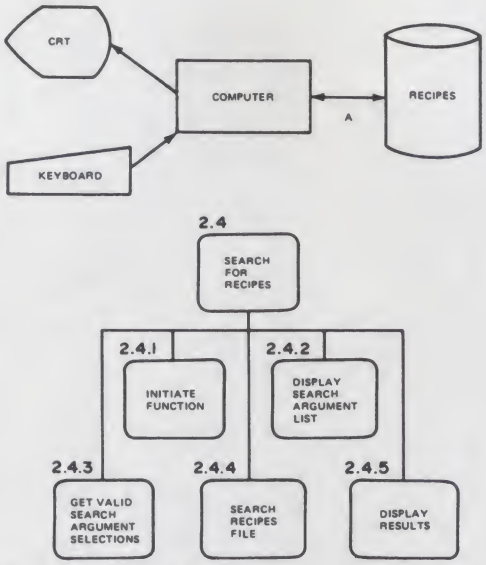


Figure 2: Recipe Subsystem.

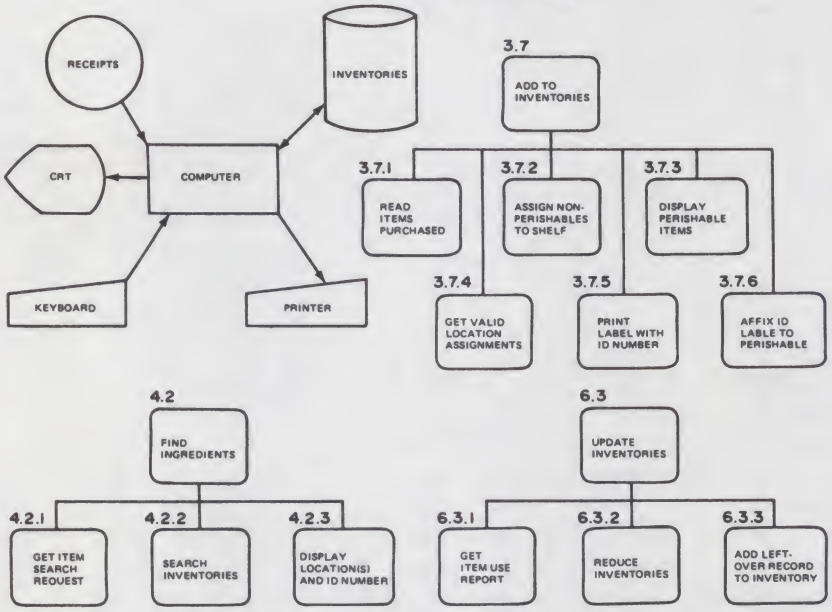


Figure 3: Inventory Subsystem.

few unmarked items. I have become a hateful person just because my grocery list is unsorted.

Beyond The Hierarchy Chart

The H chart tells us *what* to do but not *how*, so where do we go from here? I would hope that some of the readers will come forward with data base and file designs, hardware specifications, and program descriptions. This is a massive project and certainly in need of special talents and diverse opinions. There are many well known techniques for designing computer systems, and I think it is sufficient for me to mention some of the potential problems that may be encountered.

1. Is the proposed TKIS technically and economically feasible in a home? If not now, will it become so in a time frame approximately equal to the development time? To answer these questions, someone must expand the effort to prepare detailed hardware, software, and manpower estimates.

For example, a recipe retrieval subsystem might use a CRT with keyboard to initiate a search of the recipe file, and to display the results on the cathode ray tube (CRT) (see figure 2 for a schematic of the hardware and an H chart of the software functions). Assuming a file contains 2000 recipes averaging 500 B each, what are the cost and performance tradeoffs between a tape system versus disk system? In order to make this estimate we must know, first, what response time is acceptable to the kitchen operator. If the program reads records sequentially, what tape speed is required? What bandwidth is needed for data path A in figure 2 and how fast must programs execute? And so on . . .

2. How can the human labor required for data entry be kept below that required for the manual system? Data entry is the process by which humans, through the sweat of their brows, convert data into machine readable form so that the computer can do marvelous things with it and look like a genius. Data entry is probably one of the most costly items in the operating budget of the TKIS, and certainly one of the most boring.

Table 1 lists the tables and files needed to implement a basic version of the TKIS. Of the tables, *recipes* and *prices* represent large data entry tasks at initial system startup and at periodic intervals. It would be very nice if the kitchen operator could acquire data in machine readable form (on cassettes or via the phone line). The book and magazine publishers could supply a periodic update of recipes and the markets an update of prices. Standard formats would have to be developed for these interfaces, and a customer base must be developed to provide an economic incentive.

The files, on the other hand, originate within TKIS and change continually with use, making it difficult to solve the data entry problem in the same way. For example, figure 3 describes a test design for an inventory module. The functions of this subsystem are to add a record (or a count) of each purchased item to the inventory file corresponding to the storage location, to allow retrievals by item, and to decrement the inventories as items are used. The major data entry requirements are to tell the computer what was used and what was bought.

An efficient way to do the former is to *signify* what was used, instead of *specifying* in detail what was used. By entering the recipe name (say, recipe B), the operator

Table 1: Functional Storage Requirements.

File Name	Contents	Possible Source
A. Tables		
1. Prices	grocery item prices by brand for each market	Grocers
2. Recipes	ingredients, instructions, recipe characteristics, nutritional data, number of servings	Book and Magazine Publishers
3. Menus	groups of recipes, menu characteristics	Book and Magazine Publishers
4. Calendar	dates, meal times, number of guests, other requirements	TKIS User
5. Markets	market name, address, distance	TKIS User
B. Files		
1. Inventories	number and quantity of ingredients by location	TKIS User
freezer, refrigerator }	also perishable items by ID No.	
shelf stock items –	number and quantity of items, rate of use (salt, soy sauce)	
2. Meal Schedule	menu or recipes for each meal	TKIS User
3. History	past schedules and evaluations, market receipts, etc.	TKIS User
4. Working Storage	purchase list, grocery lists, etc.	TKIS User

says in effect that "the ingredients for recipe B were used." This requires that the computer have a recipe file for translating "recipe B = ingredients D, E, F." If the computer lacks this file, the operator must enter the specific ingredients used. Thus a stand alone inventory subsystem is less data entry efficient than one integrated into a full TKIS (a truism about systems in general).

On the other hand, telling the computer what was bought can be handled rather neatly, by adhering to the rule that once the data is in machine readable form it should not be degraded out of same. Instead of a paper receipt, the bag person at the market will plop a cassette in your bag containing all the items you purchased and their prices. This cassette will have been produced by the market's point-of-sale terminal which so graciously performed the data entry chore for you by optically scanning your groceries. (In fact – or rather in fantasy – the market won't even have to provide the cassette: you will bring the purchase list created by TKIS on cassette to the store, insert it into the computer at the front door which sorts and prints your grocery list in location order (3.3.3 and 3.3.4), and carry the cassette to the checkout counter for recording of your receipt.)

3. What does the kitchen operator do when the system goes down because a disk *crashes*, or the bus turns *flaky*, or a program *blows up*? (This picturesque lingo seems to less accurately describe the condition of the computer than it does our emotional state after the unthinkable has happened.) Backup manual procedures or alternate computer services must be provided to allow the kitchen operator who has become dependent on the TKIS to function while the system is down. Adequate system recovery and restart procedures must be designed, and a technique developed for catching the computer up on what transpired while it was unconscious. The importance of these considerations will depend upon the complexity and reliability of the hardware and software, but must be conceived and designed as an integral part of the total system.

4. Finally, assuming a TKIS was developed, would a kitchen operator use it? Besides being more efficient, less costly, and all the other good reasons for which we developed it, the TKIS must be flexible enough to allow for human inefficiency and taste preferences. What if the TKIS user doesn't want to prepare the scheduled meal for the evening? TKIS must be able to take account of human inconsistency.

Summary

I have briefly outlined the functions I think a kitchen information system should perform, and mentioned some considerations affecting its design. I hope this article will help catalyze development efforts in what appears to be a fruitful home computer applications area. ■

A Small Business Accounting System

Or, How Your Microcomputer Can Take the Worry Out of Tax Time

John A Lehman
716 Hutchins #2
Ann Arbor MI 48103

The least sophisticated form of bookkeeping is single entry accounting; it is not, however, generally suitable for preparing financial statements for banks, investing brothers-in-law, and so forth.

Double entry bookkeeping has the advantage of incorporating redundancy and error checking techniques. It is the most common form of business accounting.

Here's an outline of an accounting system suitable for small business use on a microcomputer. It is designed for a small, inexpensive system having a central processor, Teletype IO, one or preferably two cassette tapes for storage, and a high level language facility such as BASIC. It could probably be written in assembly language, but at a price of inconvenience. The system is designed to be used by an individual proprietorship (one man business) or a small partnership. While perhaps suitable as a bookkeeping system for a small corporation, it is not intended to produce the sort of reports which various regulatory agencies may require of one. It is designed to keep books, produce tax returns (either Form 1040 schedule C for proprietorships or Form 1065 for partnerships), produce balance sheets which may be required either for management information or for the information of banks and other outside investors, and to be adaptable for check reconciliation, cash budgets, pro forma balance sheets and the like. Its use requires about the same amount of time and effort as keeping a journal would normally, with the added advantage that the entries are pretty much self checking. All other reports are produced by the programs which would be used. I'll try to describe the system in enough detail so that anyone who is skilled in BASIC and knows a little about accounting could write a program to do all of the above.

First, however, it might be a good idea to take a quick look at accounting systems and what they're used for.

Of the various systems available, the simplest is the single entry system. A check book is a good example; each time money goes in or out, a notation is made of the date, the amount, and any comments on sources, uses, etc. This sort of system is obviously very simple to keep, and has the additional advantage of being accepted by the IRS for preparing tax returns. However, it has a number of disadvantages. The first is that it is not self checking, as anyone who has ever tried to balance a checkbook can testify. Also, while capable of producing an "income statement" (the generic term for what a tax return amounts to), it is not suitable for the preparation of other financial statements that may be required by banks, investing brothers-in-law and so forth. These disadvantages make a single entry accounting system unsuitable for the system under discussion here.

Double Entry Accounting

The other major accounting system is the double entry system. It was invented about 600 years ago, and came into widespread use because it was self checking. It is also quite a bit more complicated than a single entry system. The basic idea behind the double entry system is that each transaction has two parts: where money comes from and where it goes. So each transaction is entered twice, each time in a different account. The mechanism behind this is the idea of debits (DR) and credits (CR). Very briefly, a debit represents an addition to something which you have (an asset) or to an expense. A credit represents a subtraction from one of these. On the other hand, a debit represents a subtraction from something which you owe or from a revenue, while a credit represents an addition to one of these. All of which can be very confusing.

As a quick example, suppose you pay

$100 on your BankAmericard and receive $150 for some service which you performed. You would debit accounts payable (subtracting from what you owe) for $100, and credit cash (subtracting from something you have) for $100. Then you would debit cash (adding to something you have) for $150, and credit income or revenue (adding to revenue) for $150. The self checking feature is provided by the fact that debits must always equal credits. It would probably be a good idea to look through a beginning accounting book to get more examples to help explain accounting techniques. I've listed some at the end of the article.

Besides being self checking, a double entry system has the advantage of being able to churn out all sorts of reports on what is going on in the business in question. The IRS approves of it; and in fact, large companies have no choice — they *have* to use it. Now that we've described the major accounting systems, let's get on to what they do and how they can be used in a computerized system.

The purpose of any accounting system is to provide information (another purpose is to provide employment for accountants, of course). This information is of use to various people. The owner of a business uses it to see how well he's doing, and more important, where he's not doing so well. Another important user is your friendly local IRS agent; anyone in business is required to produce accounting reports to the Internal Revenue Service's specifications. Banks and other investors also are likely to be quite interested in this sort of information, especially when their services are requested for loans rather than for deposits. Corporations are also required to provide statements to various government agencies, but we're not going to be concerned with that here.

The basic statements and reports were mentioned earlier. The first is the balance sheet such as the one shown in figure 1. This represents the financial state of the company at a particular time. The left hand side (in the US at least) represents assets, or what the firm has. The right hand side represents liabilities and equities. (Liabilities and Equities is accounting terminology for where the stuff on the left came from.) Liabilities are amounts owed; equities are amounts contributed or earned by the owner(s). The second statement is the income statement. As was mentioned above, a tax return is a species of income statement. This shows what happened over a period of time. Other statements, such as the cash budget and the pro forma balance sheet, show what may happen in the future. These are the statements which our system is going to be able to churn out. Now, having got an overview of what we're trying to do, let's take a look at our data base requirements.

The first thing we are going to need here is a set of names for our accounts. This is a "chart of accounts" to use the jargon of the accounting trade. A small system such as ours will need about 35 of these, selected for the most part to make our output match

ASSETS:		**LIABILITIES:**	
Cash	1000	Payables	2000
Receivables	2000	Notes from bank	1000
Equipment	4000		
		EQUITY:	
Total	7000	Proprietor	4000
		Total	7000

Figure 1: The Balance Sheet. This document shows the current financial state of a business operation. It is used by businesses large and small, and is one of the end products of the automated accounting system.

EXPENSES (Debit to add, Credit to subtract)	
Return & Allowances	RTN
Depreciation*	DEP
Business Taxes*	TAX
Rent	RNT
Repairs*	RPR
Salaries & Wages	SAL
Insurance	INS
Professional fees	PRF
Commissions	COM
Amortization*	AMT
Pension/Profit sharing	PEN
Interest	INT
Bad Debts	BDB
Depletion	DPL
Other (specify if common, eg:	MIS
Fuel	FUL
Electricity	PWR
Telephone	FON
Cost of Goods Sold which includes	CGS
Purchases	PUR
Materials/supplies	MAT
Other costs	OTR
Labor (used for or directly related to Production — does not include money paid to you)	LAB

ASSETS (Debit to add, Credit to subtract)	
Cash	CSH
Receivables	RBL
Inventory	INV
Prepaid expenses	PPD
Supplies	SUP
Equipment	EQT
Investments	IVS
Misc.	ETC
LIABILITIES & EQUITY (Credit to add, debit to subtract)	
Payables	PBL
Notes	NOT
Long Term Payables	LTP
Proprietor	PRP
Drawing	DRW
REVENUES (Credit to add, Debit to subtract)	
Gross Receipts	RCP
Other Revenue	REV

***Items for which the IRS requires supplementary schedules or statements**

Figure 2: Account Files Example. When the double entry accounting system is designed, one of the first steps is to create a list of accounts and their corresponding mnemonic codes. The mnemonic codes are used internally by the computer in order to save memory space. If you are lavish with memory, texts of the long names could be looked up in a table when you generate reports.

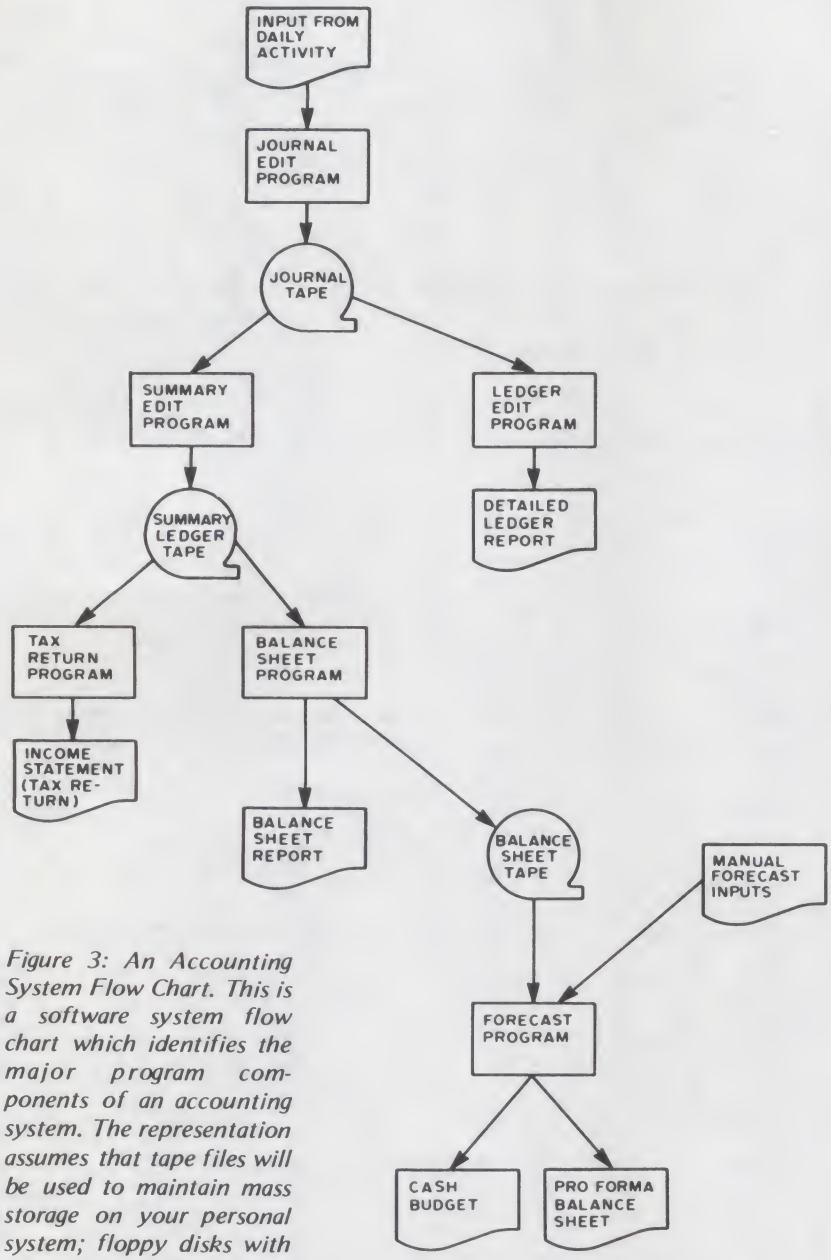


Figure 3: An Accounting System Flow Chart. This is a software system flow chart which identifies the major program components of an accounting system. The representation assumes that tape files will be used to maintain mass storage on your personal system; floppy disks with sequential access file organizations could be used as well.

what the IRS requires. In order to save memory space in the programs, each of these accounts is also given a three letter mnemonic code. Two letters would be possible, but some ease of use would be sacrificed. Figure 2 gives a sample list of accounts and mnemonics, broken down by classification. A brief explanation of some of the accounts might be in order. Returns and Allowances is for goods which are returned for one reason or another. Its purpose is to reduce the amount in gross receipts while keeping the amount of returns separate. The category SAL includes only those wages paid which are not included in cost of goods sold (CGS). This would involve such things as clerical help. INT is interest paid, not received. BDB (bad debts) is used if we want to use the specific charge off method of accounting for such unfortunate happenings. The IRS also allows use of another method, called the reserve method, but it is more complicated. DPL (depletion) is used for things like oil wells and mines. DEP (depreciation) is used for equipment, machines and the like, while AMT (amortization) is used to charge part of the cost of such things as organization expense, capitalized research and development and so forth. Some of these things can be listed as assets when the money is first spent, and the cost spread over several periods. For details see the IRS books listed at the end of the article. Cost of goods sold (CGS) is the total of the costs incurred to get something ready for sale; the breakdown is listed below it. Cash (CSH) is mostly checking account balances. Receivables (RBL) are what customers owe you on account. Payables (PBL) are what you owe on account. Proprietor (PRP) is what you put into the business and what it has earned so far. Drawing (DRW) is the account you use to take money out of the business for personal uses. Notes (NOT) is money borrowed from banks and other lenders. The rest should be pretty much self explanatory. These 35 or so accounts are the data files which we're going to be working from; all of the information we put into the system goes into them and all of the output uses them as building blocks. Now, having taken care of all of the groundwork, we are ready to start running information through the system.

Reference to the system flow chart of figure 3 shows that the journal is the first thing we produce. It's shown being produced on tape, since that way we can use it to produce all of the other reports without having to type in any more material, at least until we come to the forecasts. Also, by writing our journal entries onto tape as soon as they're checked by the editing program, we save much memory space, since we need keep only a little bit of data in memory at any given time. So, in this automated system, the journal is the only file we really have to manipulate on a day by day basis. To use it, first we enter the date. Then we enter each transaction through the checking program which makes sure we have two entries for each amount and that the numbers we give the machine match. A sample of a possible format is given as figure 4. We debit the power expense account (remember, we debit an expense when we want to add to it) for $58, and enter the comment that this is for the month of March. Then we credit cash (to decrease it), but reverse the

numbers. The program sees that the debits do not equal the credits, and fires off an error message, prompting us to enter a correction. Note here that we include the check number; this is very important when it comes time to reconcile our records with what the bank statement says. Also, the editing program should provide the ability to debit and credit unequal numbers of accounts so long as the totals are equal. If this would be too much of a demand on memory, amounts can be split up before entry. Going on, the OK indicates that the entries check, and at this point they should be written onto the tape. Entries for the journal can come from cash register tapes, bills, etc. Up through this point our system is about as much work as a manual system, but from here on in things get much easier.

The next item on the system flow chart is the ledger. This is a set of files which puts all of the journal entries for each account together. In our system, there are two types: summary and detailed. In a more advanced system, all of the ledgers would be detailed, but this would require much more memory than most small systems would have available. Basically, what we do at this point is have the program read the journal entries one by one and keep a running count of the amount for each of the different accounts in use. Beginning balances may be read in either via the Teletype or via a separate ledger tape. The ending balances should be printed on the Teletype if the user wishes to see what they are, but they should also be saved on tape for use in preparing the rest of the statements. Detailed ledgers will require a separate run for each one desired; they might be run on a weekly or monthly basis. The most important one is the cash ledger, since this will provide a record of every check written and every deposit made to the checking account by date and number. This should make balancing one's checkbook a fairly simple task. The one thing to be careful of in this program is to be sure that the rules for addition and subtraction of debits and credits are carefully written into the program. Otherwise all that will come out is garbage.

Once we have the ledger, it's fairly easy to see how the balance sheet is generated. A look back at figure 1 will show that there are only about a dozen of the ledger accounts which have to be put together. All of the asset accounts are added together, and the sum is listed at the bottom of the column as total. Subtracting the sum of the liabilities from the sum of the assets leaves what is left for the owner. If the amount in the drawing account is set beforehand, that leaves only the Proprietor (PRP) account to be "plugged," which is to say, given whatever value is necessary to make the two columns come out equal. So, if the assets total $7000, the liabilities total $3000 and there are $500 in the drawing account, that leaves 7000–3000–500=3500 for PRP. The only other detail is that the program should either write the date at the top, or it should be filled in by hand. A balance sheet may be prepared at any time; it will often be required for getting a loan from a bank. Besides being run on paper, it should be run onto tape for use in preparing forecasts.

The balance sheet is a snapshot of the current status of the business.

Probably the most important report which our system will prepare is the income statement. This is a report which shows what has happened over a period; usually a year, but often prepared on a quarterly or a monthly basis. Its importance arises not so much from the fact that people like to see how much money they've made as from the fact that the government is quite interested in this information – so they can take their cut, of course. The system being illustrated produces an income statement patterned

A mass storage file comes in handy for business accounting, since much of the work involved is accomplished by reviewing the same data with different criteria to produce reports.

Figure 4: An Example of the Interactive Dialog with the Journal Edit Program. The purpose of this program is to filter your own manual inputs looking for certain known discrepancies which can be detected by the double entry bookkeeping method. In this example, upper case letters are the computer output to a Teletype (or video terminal) and the lower case letters indicate manual keyboard inputs taken from daily activity records such as receipts, checks written, etc.

Interactive program for journal entries might read:

```
ENTER NAME OF ACCOUNT DEBITED,AMOUNT,AND COMMENTS SEPARATED BY COMMAS:
pwr,58,march
ENTER ACCOUNT CREDITED,AMOUNT,AND COMMENTS SEPARATED BY COMMAS:
csh,85,check 346
DEBITS DO NOT EQUAL CREDITS–ENTER IF DR OR CR TO BE CHANGED:
cr
ENTER ACCOUNT CREDITED,AMOUNT,AND COMMENTS SEPARATED BY COMMAS:
csh,58,check 346
OK
ENTER NAME OF ACCOUNT DEBITED,AMOUNT,AND COMMENTS SEPARATED BY COMMAS:
iam done
OK GOODBYE
```

after Form 1040 Schedule C (figure 5), but could produce Form 1065 for partnerships with minor changes. As is fairly obvious to those who can wade their way through the governmentese, what we have to do here is state all income and then subtract expenses. The accounts which we have been working with will do this on what is called an accrual basis, which is to say future expenses and revenues are included if they are certain and we know how much money is involved. For example, if we have charge customers, we include what they are scheduled to pay us in revenues. For a small business it is often better to file a tax return on the cash basis in which only cash in is considered revenue and cash out is considered expense. This system can prepare cash basis returns too; one must eliminate receivables, payables, prepaid expenses and materials and supplies not yet part of cost of goods sold. The effect of all of these should be taken out of the revenue and expense accounts too.

That's the basic system. Using this system alone would be a pretty respectable accounting setup for a small business. But as long as we're using a personal microcomputer, we might think of adding a few bells and whistles. These would pretty much depend on individual wants. We could have the computer automatically calculate FICA deductions when payroll expense is debited. We might also have the machine figure our depreciation and amortization schedules for us. For this we would need (for each item or class of items) initial value, estimated life and age. For tax purposes we would want to get our annual depreciation by taking two divided by the life of the object and multiplying the total times the remaining value. In more symbolic form:

(2/total life)*(initial – depreciation).

This would give us the depreciation to date and the amount for this year, both of which are needed for the flip side of the tax form. We could also do forecasting with the system. For this we would want an interactive program which would ask for estimated expenses and receipts in all the different categories for x number of months. Then we would prepare a (pro forma) balance sheet for the end of the period if our predictions were correct, so that we could see where things would stand if the predictions came true. It could also prepare a month by month schedule to show whether the firm would have enough on hand to meet projected outflows. This is called a cash budget, and is quite a handy thing to have since it enables you to forecast cash shortages far enough in advance to do something about them, and also to compare the results of different courses of action.

And there's the system. While not very fancy from either an accountant's or a system designer's point of view, it ought to be enough to handle much of the record-keeping for those firms on the other end of the spectrum from GM, IBM and ITT. It might be too that the availability of a few business oriented systems like this will help increase the sales of microcomputers and bring the prices down even more through mass production.■

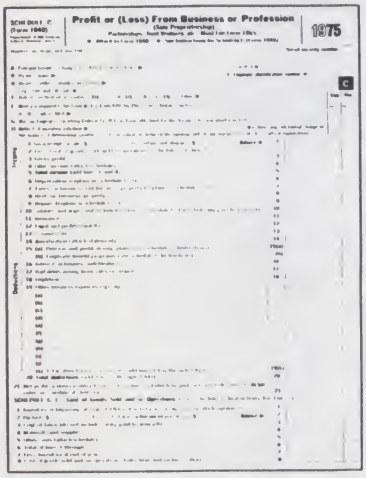


Figure 5: The object of much of this program activity is filling out IRS Schedule C for your small business.

GLOSSARY

Accrual: Including payments and receipts in the future.

Check reconciliation: Accounting buzzword for balancing a checkbook.

Credit (CR): An addition to the righthand side of the balance sheet or to income.

Debit (DR): An addition to the lefthand side of the balance sheet or to an expense.

Journal: The accounting equivalent of a check register.

Ledger: Book or file which contains the totals from the journal broken down by categories.

Payables: Amounts which will have to be paid in the future.

Pro forma: Buzzword used to describe reports which show how things might be or might have been rather than what they are.

Proprietorship: A one man business; one owner.

Receivables: Amounts which are not yet on hand in cash but which will definitely be coming in in the near future.

REFERENCES

1. *Accounting Essentials*, Margolis, Wiley and Sons 1972.

2. *Elementary Accounting*, College Outline Series #39.

3. *Management Accounting*, Anthony and Reece, Irwin, Inc, 1975 (Note: this is a college accounting textbook – heavy reading).

4. *Recordkeeping for a Small Business*, IRS #583, 1976.

5. *Tax Guide for Small Business*, IRS #334, 1976 (complete handbook).

6. *Tax Information on Accounting Periods and Methods*, IRS #538, 1975.

The last three are available free from any IRS office.

Chips Found Floating Down Silicon Slough

Roy H Trumbull
833 Balra Dr
El Cerrito CA 94530

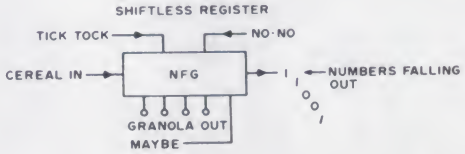
The state of the Art is changing rapidly. In fact I ran into him in New Mexico last month. Seems he had just gotten back from China where he had seen their latest computer. It was really fantastic, but they still have a problem with noise from the beads. I asked Art to clue me in on the latest devices coming out of research and these are the ones he told me about:

The Don't Gate

You don't get no output no matter what's at the inputs. It is believed that the don't gate was the breakthrough that made the LSI write only memory possible.

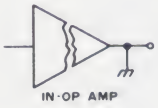
Noise Emitting Diode (NED)

When connected across a 1000 volt supply it makes a loud noise (once). The NED was discovered by Igor Pravaganda whom you'll recall worked many years trying to filter AC with electrolytics. He'll always be remembered as the father of the confetti generator.



Shiftless Register

Must be used with 3 speed forward clutch gate. Shifts at 15, 25, and 35 bits per second. Double clutching with logic 2s is not suggested.

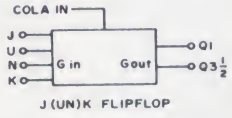


Inoperational Amplifier (IN-OP AMP)

Linear cousin of the DON'T gate. Provides no output for any input at a slew rate of 0 volts per microsecond. Mil Spec. version available at 100 times the cost of OEM version.

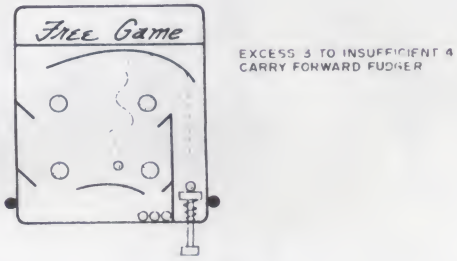
J(UN)K Flip Flop

Doesn't change state when clocked regardless of input states. Changes state only when cola machine down the hall makes change.



Excess 3 To Insufficient 4 Carry Forward Fudger

Used to enter Murphy factor and get the programmer off the hook.



Moss

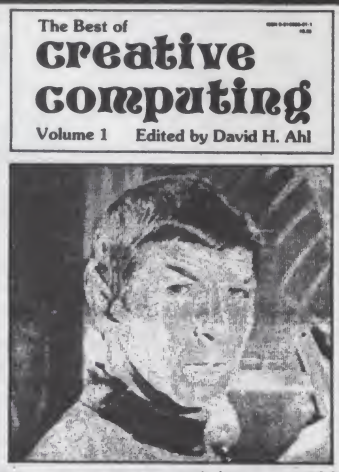
Highly experimental material. Very rare at present since only source is from under grizzly bear toenails. Turns green when facing north while on wood substrate.

Fuzz Locked Loop

Great if you want to avoid radar speed traps. ■

Resources

The diversity in *The Best of Creative Computing — Volume 1* can only be described as staggering. The book contains 328 pages of articles and fiction about computers, games that you can play with computers and calculators, hilarious cartoons, vivid graphics and comprehensive book reviews.



Authors range from Isaac Asimov to Sen. John Tunney of California; from Marian Goldeen, an eighth-grader in Palo Alto to Erik McWilliams of the National Science Foundation; and from Dr. Sema Marks of CUNY to Peter Payack, a small press poet. In all, over 170 authors are represented in over 200 individual articles, learning activities, games, reviews and stories.

This 328-page book has 108 pages of articles on computers in education, CAI, programming, and the computer impact on society; 10 pages of fiction and poetry including a fascinating story by Isaac Asimov about all the computers on earth linking up after a nuclear war to support the few remaining survivors; 15 pages of "Foolishness" including a cute cartoon piece - called"Why We're Losing Our War Against Computers"; 26 pages on "People, Places, and Things" including the popular feature "The Compleat Computer Catalogue" which gives capsule reviews and lists sources for all kinds of computer-related goodies; 79 pages of learning activities, problems and puzzles; 29 pages continuing 18 computer games including a fantastic extended version of the single most popular computer game — Super Star Trek; and 32 pages of in-depth book and game reviews including Steve Gray's definitive review of 34 books on the Basic language.

The Best of Creative Computing - Volume 1 is available by mail for $8.95 plus 75¢ postage from Creative Computing Press, Attn: Becky P.O. Box 789-M, Morristown, N.J. 07960.

THE BEST OF BYTE — VOL. 1

The Best of Byte - Volume 1 is a 384-page blockbuster of a book which contains the majority of material from the first 12 issues of *Byte* magazine. 146 pages are devoted to "Hardware" and are cram full of how-to articles on everything from TV displays to joysticks to cassette interfaces. The section on computer kits describes building 7 major kits. But hardware without software might as well be a boat anchor, so there are 125 pages of "Software and Applications" ranging from on-line debuggers to games to a complete small business accounting system. A section on "Theory" examines the how and why behind the circuits and programs, and a final section "Opinion" looks at where this explosive new hobby is heading.

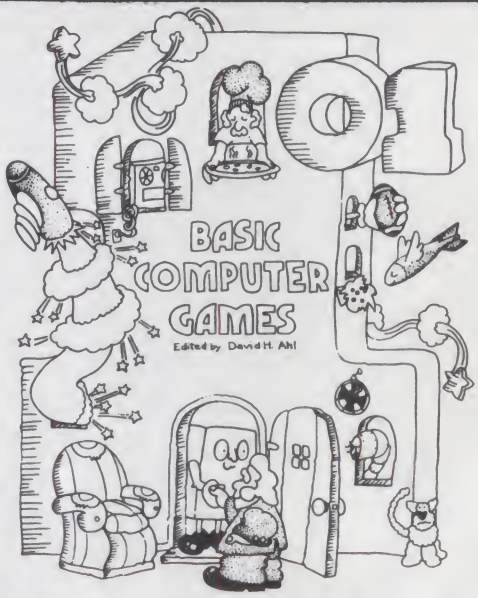
The Best of Byte - Volume 1 is edited by Carl Helmers and David Ahl and published by Creative Computing Press. Price in the US is $11.95 plus $1.00 shipping and handling ($12.95 total); foreign orders add $1.00 ($13.95 total). Orders from individuals must be prepaid. Creative Computing Press, Attn: Becky , P.O. Box 789-M, Morristown, NJ 07960. Allow 8 weeks for delivery.

ARTIST AND COMPUTER is a unique new art book that covers a multitude of computer uses and the very latest techniques. In its pages, 35 artists who work with computers explain how the computer can be programmed either to actualize the artist's concept (such as the visualization of fabric before it is woven) or to produce finished pieces. Illustrated with more than 160 examples of computer art, 9 of them in full color. ARTIST AND COMPUTER will fascinate and inspire anyone who is interested in art or computer technology. Size 8½" x 11".

ARTIST AND COMPUTER

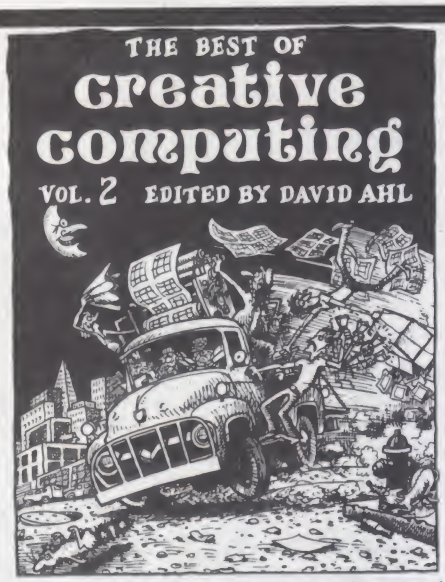
Edited by RUTH LEAVITT

Paper $4.95, cloth $10; now at selected bookstores, or send payment plus 75¢ handling to Creative Computing, P.O. Box 789-M, Morristown, N.J. 07960. N.J. residents add 5% sales tax.



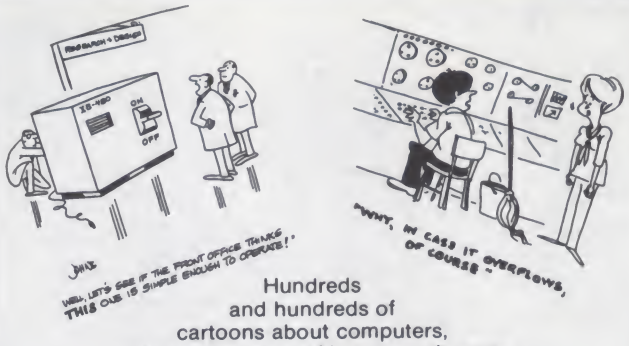
101 BASIC Computer Games is the most popular book of computer games in the world. Every program in the book has been thoroughly tested and appears with a complete listing, sample run, and descriptive write-up. All you need add is a BASIC-speaking computer and you're set to go.

101 BASIC Computer Games. Edited by David H. Ahl. 248 pages. 8½x11 paperbound. $7.50 plus 75¢ postage and handling ($8.25 total) from Creative Computing, P.O. Box 789-M, Morristown, NJ 07960.



This fascinating 336-page book contains the best of the articles, fiction, foolishness, puzzles, programs, games, and reviews from Volume 2 issues of *Creative Computing* magazine. The contents are enormously diverse with something for everyone. Fifteen new computer games are described with complete listings and sample runs for each; 67 pages are devoted to puzzles, problems, programs, and things to actually do. Frederik Pohl drops in for a visit along with 10 other super storytellers. And much more! The staggering diversity of the book can really only be grasped by examining the contents, or better yet, the book itself.

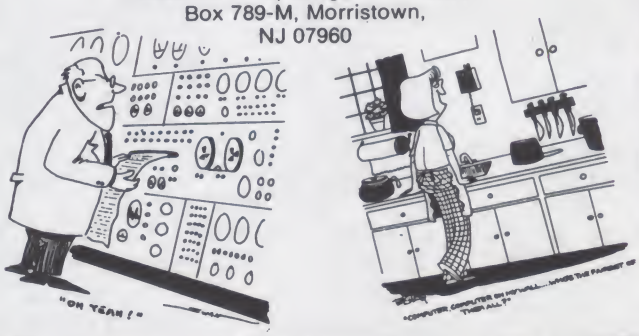
Price is $8.95 plus $0.75 shipping and handling in the USA ($9.70 total); outside USA, add $1.00 ($10.70 total). Individual orders must be prepaid. Creative Computing Press, Attn: Becky P.O. Box 789-M, Morristown, NJ 07960.



Hundreds
and hundreds of
cartoons about computers,
robots, calculators, AI, and much more.

THE COLOSSAL COMPUTER CARTOON BOOK

128 big pages! Paperbound. Only $4.95
plus 75¢ postage ($5.70 total).
Creative Computing, Attn: Becky
Box 789-M, Morristown,
NJ 07960



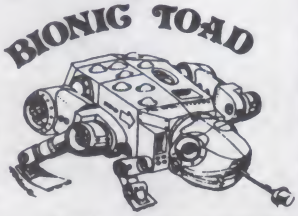
Outrageous T Shirts!

Einstein in black, white shirt, scarlet sleeve and collar trim.

Scarlet design, orange shirt.

Black design, light green shirt.

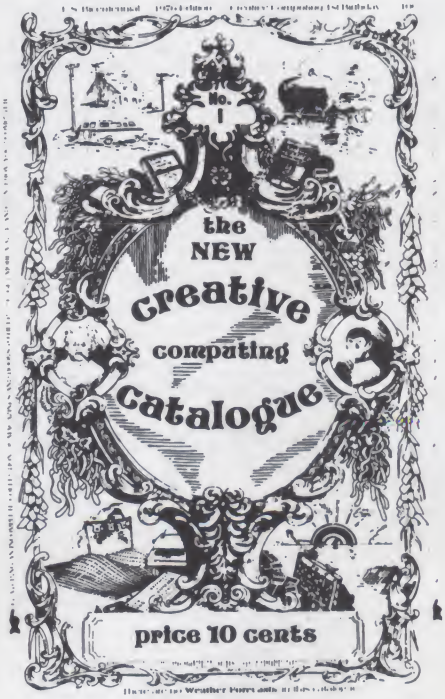
Hot pink design, yellow shirt.



Purple design, powder blue shirt.

T-Shirts available in adult sizes S, M, L, XL. All cotton, made in USA. $4.50 each postpaid in USA, $5.50 to foreign addresses. Send order (specifying design and size) with payments to Creative Computing, P.O. Box 789-M, Morristown, NJ 07960. Allow 8 weeks for delivery.

Enterprise in sliver, dark blue shirt.



The New Creative Computing Catalogue is cram full of goodies you'll want to know about or order. Described are over 60 books, art prints, posters, T-Shirts, and magazines. Double Wow!! Send for one today — FREE!

Creative Computing Press, Attn: Becky P.O. Box 789-M, Morristown, NJ 07960.

101 Basic Computer Games

David H. Ahl. An anthology of games and simulations—from Acey-Deucey to Yahtzee, all in the BASIC language. Contains a complete listing, sample run, plus a descriptive write-up of each game. Our most popular book! Large format, 248 pp. $7.50 [6C]

What to Do After You Hit Return

Another collection of games and simulations—all in BASIC—including number guessing games, word games, hide-and-seek games, pattern games, board games, business and social science simulations and science fiction games. Large format. 158 pp. $6.95 [8A]

Fun & Games with the Computer

Ted Sage. "This book is designed as a text for a one-semester course in computer programming using the BASIC language. The programs used as illustrations and exercises are games rather than mathematical algorithms, in order to make the book appealing and accessible to more students. The text is well written, with many excellent sample programs. Highly recommended." —*The Mathematics Teacher* 351 pp. $6.95 [8B]

Game Playing With the Computer, 2nd Ed.

Donald Spencer. Over 70 games, puzzles, and mathematical recreations for the computer. Over 25 games in BASIC and FORTRAN are included complete with descriptions, flowcharts, and output. Also includes a fascinating account of the history of game-playing machines, right up to today's computer war games. Lots of "how-to" information for applying mathematical concepts to writing your own games. 320 pp. 1976 $14.95 [8S]

BYTE Magazine

If you are considering a personal computing system now or later, BYTE provides a wealth of information on how to get started at an affordable price. Covers theory of computers, practical applications, and of course, lots of how-to build it. Monthly. 1-Year sub'n $12.00 [2A], 3-Years $30.00 [2B]

Games & Puzzles Magazine

The only magazine in the world devoted to games and puzzles of every kind — mathematical, problematical, crosswords, chess, gomoko, checkers, backgammon, wargames, card games, board games, reviews, competitions, and more. Monthly. 1-Year sub'n $12.00 [3A]

Games With The Pocket Calculator

Sivasailam Thiagarajan and Harold Stolovitch. A big step beyond tricks and puzzles with the hand calculator, the two dozen games of chance and strategy in this clever new book involve two or more players in conflict and competition. A single inexpensive four-banger is all you need to play. Large format. 50 pp. $2.00 [8H]

Games, Tricks and Puzzles For A Hand Calculator

Wally Judd. This book is a necessity for anyone who owns or intends to buy a hand calculator, from the most sophisticated (the HP65, for example) to the basic "four banger." 110 pp. $2.95 [8D]

So you've got a personal computer.

Now what?

Creative Computing Magazine

So you've got your own computer. Now what? *Creative Computing* is chock full of answers — new computer games with complete listings every issue, TV color graphics, simulations, educational programs, how to catalog your LPs on computer, etc. Also computer stories by Asimov, Pohl, and others; loads of challenging problems and puzzles; in-depth equipment reports on kits, terminals, and calculators; reviews of programming and hobbyist books; outrageous cartoons and much more. *Creative Computing* is *the* software and applications magazine of personal and educational computing. Bi-monthly.
1-year sub'n $8.00 [1A], 3- years $21.00 [1B], sample copy $1.50 [1C]

The Best of Creative Computing — Vol. 1

David Ahl, ed. Staggering diversity of articles and fiction (Isaac Asimov, etc.), computer games (18 new ones with complete listings), vivid graphics, 15 pages of "foolishness," and comprehensive reviews of over 100 books. The book consists of material which originally appeared in the first 6 issues of *Creative Computing* (1975), all of which are now out of print. 324 pp. $8.95 [6A]

Computer Lib/ Dream Machine

Ted Nelson. This book is devoted to the premise that everybody should understand computers. In a blithe manner the author covers interactive systems, terminals, computer languages, data structures, binary patterns, computer architecture, mini-computers, big computers, microprocessors, simulation, military uses of computers, computer companies, and much, much more. Whole earth catalog style and size. A doozy! 127 pp. $7.00 [8P]

Computer Power and Human Reason

Joseph Weizenbaum. In this major new book, a distinguished computer scientist sounds the warning against the dangerous tendency to view computers and humans as merely two different kinds of "thinking machines." Weizenbaum explains exactly how the computer works and how it is being wrongly substituted for human choices. 300 pp. $9.95 [8R]

Artist and Computer

Ruth Leavitt, ed. Presents personal statements of 35 internationally-known computer artists coupled with over 160 plates in full color and black & white. Covers video art, optical phenomena, mathematical structures, sculpture, weaving, and more. 132 pp. $4.95 [6D] Cloth cover $10.95 [6E]

Computer Science: A First Course (2nd Ed.)

Forsythe, Keenan, Organick, and Stenberg. A new, improved edition of this comprehensive survey of the basic components of computer science. There has been an updating of important areas such as Programming, Structured Programming, Problem Solving, and other Computer Science Concepts. The quantity of exercises and problems has been increased. 876 pp. $16.95 [7D]

Mr. Spock Poster

Dramatic, large (17" x 23") computer image of Mr. Spock on heavy poster stock. Uses two levels of overprinting. Comes in strong mailing tube. $1.50 [5B]

Problems For Computer Solution

Gruenberger & Jaffray. A collection of 92 problems in engineering, business, social science and mathematics. The problems are presented in depth and cover a wide range of difficulty. Oriented to Fortran but good for any language. A classic. 401 pp. $8.95 [7A]

A Guided Tour of Computer Programming In Basic

Tom Dwyer and Michael Kaufman. "This is a fine book, mainly for young people, but of value for everyone, full of detail, many examples (including programs for hotel and airline reservations systems, and payroll), with much thought having been given to the use of graphics in teaching. This is the best of the introductory texts on BASIC."—*Creative Computing* Large format. 156 pp. $4.40 [8L]

BASIC Programming 2nd Ed

Kemeny and Kurtz. "A simple gradual introduction to computer programming and time-sharing systems. The best text on BASIC on almost all counts. Rating: A+"—*Creative Computing.* 150 pp. $8.50 [7E]

Build Your Own Working Robot

David Heiserman. Complete plans, schematics and logic circuits for building a robot. Not a project for novices, this robot is a sophisticated experiment in cybernetics. You build him in phases and watch his capabilities increase and his personality develop. Phase I is leash led, Phase II has a basic brain, while Phase III responds and makes decisions. 238 pp. 1976 $5.95 [9M]

Computers and Society

R. Hamming. Provides a framework for thinking about and drawing conclusions about how machines should be used in our society. Presents, in a non-technical way, the principles of computer operations, programming and use. 288 pp. 1972 $7.95 [8T]

Problem Solving: The Computer Approach

LaFave, Milbrandt, and Garth. Describes the process of thinking through the steps needed to solve a problem, flowcharting the steps, coding in a computer language, development of appropriate test data, and manual checking. 176 pp. 1973 $10.40 [8U]

Problem Solving With The Computer

Ted Sage. This text is designed to be used in a one-semester course in computer programming. It teaches BASIC in the context of the traditional high school mathematics curriculum. There are 40 carefully graded problems dealing with many of the more familiar topics of algebra and geometry. Probably the most widely adopted computer text. 244 pp. $6.95 [8J]

A Simplified Guide to Fortran Programming

Daniel McCracken. A thorough first text in Fortran. Covers all basic statements and quickly gets into case studies ranging from simple (printing columns) to challenging (craps games simulation). 278 pp. $8.75 [7F]

Understanding Solid State Electronics

An excellent tutorial introduction to transistor and diode circuitry. Used at the TI Learning Center, this book was written for the person who needs to understand electronics but can't devote years to the study. 242 pp. $2.95 [9A]

Microprocessors

A collection of articles from *Electronics* magazine. The book is in three parts: device technology; designing with microprocessors; and applications. 160 pp. 1975 $13.50 [9J]

Microprocessors: Technology, Architecture and Applications

Daniel R. McGlynn. This introduction to the microprocessor defines and describes the related computer structures and electronic semi-conductor processes. Treats both hardware and software, giving an overview of commercially available microprocessors, and helps the user to determine the best one for him her. 240 pp. $12.00 [7C]

The Art of Computer Programming

Donald Knuth. The purpose of this series is to provide a unified, readable, and theoretically sound summary of the present knowledge concerning computer programming techniques, together with their historical development. For the sake of clarity, many carefully checked computer procedures are expressed both in formal and informal language. A classic series. Vol. 1: Fundamental Algorithms, 634 pp. $20.95 [7R]. Vol. 2: Seminumerical Algorithms, 624 pp. $20.95 [7S]. Vol. 3: Sorting and Searching, 722 pp. $20.95 [7T].

ALGOL by Problems

B. Meek. A set of programming exercises, both abstract and concrete, to give the reader a thorough working knowledge of ALGOL. Good companion to vendor's language manual. 168 pp. 1972 $8.95 [8V]

Computer Algorithms and Flowcharting

G. Silver and J. Silver. A straightforward approach to analyzing problems and structuring solutions suitable for the computer. Branching, counters, loops, and other important concepts are presented in easily-grasped modular units in the text. 176 pp. 1975 $6.95 [8W]

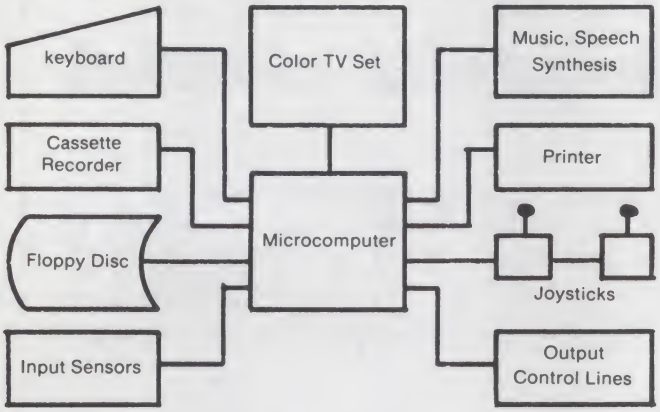
Creative Computing Catalogue

Zany 12-page tabloid newspaper/ catalog lists books, magazines, art prints, and T-Shirts. A conversation piece even if you don't order anything. Free. [5A]

Use order form or send amount plus $1.00 postage and handling to Creative Computing, Box 789-M, Morristown, NJ 07960.

When you get your home or office computer, will you know what to do with it?

The typical home or small business computer system starts with a microcomputer, keyboard, cassette recorder, and TV set. From there you can add the peripherals, sensors, controllers, and other devices you need for your own special applications.



Creative Computing Magazine is dedicated to describing applications for home, school, and small business computers completely and pragmatically in non-technical language. You won't need a Ph.D in Computer Science, or a technical reference library, or a computer technician beside you to get these applications up and running. We give you complete hardware and software details. Typically, applications utilize commercially available systems. However, if an application needs a piece of home-brew hardware, we tell you how to build it. Or if it requires a combination of high-level and machine language code, we give you the entire listings along with the flowcharts and algorithms.

We also run no-nonsense reviews of computers (assembled and kits), peripherals, terminals, software, and books. We're frank and honest, even if it costs us an advertiser, which it occasionally has.

Here are just some of the applications you'll see fully described in future issues of *Creative Computing*.

Building Management and Control
1. Alarm monitoring/police notification
2. Environmental control (heating, air conditioning, humidification, dehumidification, air purity, etc.)
3. Fire and smoke detection
4. Appliance control (microwave oven, gas oven, refrigerator)
5. Perimeter system control (sprinklers, outdoor lights, gates)
6. Solar and/or auxiliary energy source control
7. Watering system control based on soil moisture
8. Fuel economizing systems
9. Maintenance alert system for household devices (key component sensing and periodic preventative maintenance)

Household Management
1. Address/telephone file
2. Investment analysis
3. Loan/annuity/interest calculations and analysis
4. Checkbook maintenance
5. Periodic comparisons of expenditures vs. budget
6. Monitor time and cost of telephone calls
7. Record incoming telephone calls and select appropriate response to caller
8. Recipe file
9. Diet/nutrition analysis
10. Menu planning
11. Pantry inventory/shopping list

Health Care
1. Medical/dental record keeping
2. Insurance claim processing
3. Health maintenance instrumentation control (EKG, blood chemical analysis, diet analysis, self-diagnosis)

Education and Training
1. Mathematics drill and practice
2. Problem solving techniques
3. Tutorial instruction in a given field
4. Simulation and gaming
5. Music instruction and training
6. Music composition and synthesis
7. Learning to program
8. Software development
9. Perception/response/manipulation skills improvement

Recreation and Leisure
1. Games, games, games
2. Puzzle solving
3. Animation/kinetic art
4. Sports simulations
5. Needlepoint/stitchery/weaving pattern generation
6. Computer art
7. Library cataloging (books, records, etc.)
8. Collection catalog/inventory/value (coins, stamps, shells, antique auto parts, comics, etc.)
9. Model railroad control
10. Amateur radio station control
11. Astronomy; star, planet, satellite tracking
12. Robotics
13. Speech recognition and synthesis

Business Functions
1. Small business accounting
2. Word processing/text editing
3. Customer files
4. Software development
5. Operations research
6. Scientific research
7. Computer conferencing
8. Telephone monitoring
9. Engineering calculations
10. Statistical analysis
11. Survey tabulation
12. Inventory control
13. Mailing lists

SUBSCRIPTION ORDER FORM

Type	Term	USA	Foreign
Individual	1-Year	□ $ 8	□ $ 10
	3-Year	□ 21	□ 27
	Lifetime	□ 300	□ 400
Institutional	1-Year	□ 15	□ 15
	3-Year	□ 40	□ 40

□ New □ Renewal

□ Cash, check, or M.O. enclosed

□ BankAmericard Card No. _____

□ Master Charge Expiration date _____

□ Please bill me ($1.00 billing fee will be added)

Name _____

Address _____

City _____ State _____ Zip _____

Send to Creative Computing, Attn: Becky
P.O. Box 789-M, Morristown, NJ 07960

Subscriptions to CREATIVE COMPUTING

creative computing

ORDER FORM

MAGAZINE SUBSCRIPTIONS

Term	USA	Foreign Surface	Foreign Air
12 issues	☐ $ 15	☐ $ 23	☐ $ 39
24 issues	☐ 28	☐ 44	☐ 76
36 issues	☐ 40	☐ 64	☐ 112
Lifetime	☐ 300	☐ 400	☐ 600

BOOKS AND MERCHANDISE

Quan. *Cat.* *Descriptions* *Price*

_____

_____

_____

Books shipping charge $1.00 USA, $2.00 Foreign _____

NJ Residents add 5% sales tax _____

TOTAL (magazines and books) _____

☐ Cash, check, or M.O. enclosed

☐ BankAmericard } Card No. _____

☐ Master Charge } Expiration date _____

☐ Please bill me ($1.00 billing fee will be added)
Book orders from individuals must be prepaid.

Name _____

Address _____

City _____ State _____ Zip _____

Books and Merchandise

creative computing

ORDER FORM

MAGAZINE SUBSCRIPTIONS

Term	USA	Foreign Surface	Foreign Air
12 issues	☐ $ 15	☐ $ 23	☐ $ 39
24 issues	☐ 28	☐ 44	☐ 76
36 issues	☐ 40	☐ 64	☐ 112
Lifetime	☐ 300	☐ 400	☐ 600

BOOKS AND MERCHANDISE

Quan. *Cat.* *Descriptions* *Price*

_____

_____

_____

Books shipping charge $1.00 USA, $2.00 Foreign _____

NJ Residents add 5% sales tax _____

TOTAL (magazines and books) _____

☐ Cash, check, or M.O. enclosed

☐ BankAmericard } Card No. _____

☐ Master Charge } Expiration date _____

☐ Please bill me ($1.00 billing fee will be added)
Book orders from individuals must be prepaid.

Name _____

Address _____

City _____ State _____ Zip _____

Subscriptions to BYTE

BYTE SUBSCRIPTIONS

For a subscription to BYTE, please complete this form.

Name _____

(Title, optional) _____

(Company, optional) _____

Address _____

City _____

State _____ **Zip** _____

Please check:

[ ] 1 year $12

[ ] 2 years $22

[ ] 3 years $30

[ ] Lifetime $150

[ ] Check enclosed

[ ] Bill me

[ ] Bill BAC # _____

[ ] Bill MC # _____

Please allow eight weeks for processing.

Thank You

BYTE

PLACE
STAMP
HERE

BYTE SUBSCRIPTIONS
Publications, Inc.
70 Main Street
Peterborough, N. H. 03458

Dept. 50Z

Place
Stamp
Here

creative computing
P.O. Box 789-M
Morristown, New Jersey 07960

Attn: Becky

Place
Stamp
Here

creative computing
P.O. Box 789-M
Morristown, New Jersey 07960

Attn: Becky

BYTE

the small systems journal