Don Lancaster's

MACHINE LANGUAGE PROGRAMMING COOKBOOK

Part One

Machine Language Programming Cookbook I

by Don Lancaster

An eBook reprint of chapters 6 and 7
of Micro Cookbook Volume II

SYNERGETICS SP PRESS
3860 West First Street, Thatcher, AZ 85552 USA
(928) 428-4073 http://www.tinaja.com

THIRD EDITION
FIRST PRINTING—2010

International Standard Book Number: 1-882193-14-1

Created in the United States of America.

ABOUT THE AUTHOR

Don Lancaster heads *Synergetics,* a new-age software, prototyping, and consulting firm involved in micro applications and electronic design. Don is the well-known author of the classic *CMOS* and *TTL Cookbooks.* He is one of the microcomputer pioneers, having introduced the first hobbyist integrated circuit projects, the first sanely priced digital electronics modules, the first low cost TVT-1 video display terminal, the first hobbyist keyboards, and lots more. Don's numerous books and articles on personal computing and electronics applications have set new standards for understandable, useful, and exciting technical writing. Don's other interests include ecological studies, firefighting, cave exploration, tinaja questing, and bicycling.

Other Howard W. Sams books by Don Lancaster include *Active Filter Cookbook, CMOS Cookbook, TTL Cookbook, RTL Cookbook* (out of print), *TVT Cookbook, Cheap Video Cookbook, Son of Cheap Video, The Hexadecimal Chronicles, The Incredible Secret Money Machine, Don Lancaster's Micro Cookbook,* Volume 1, and the continuing *Enhancing Your Apple II* series.

Preface

Machine Language Programming is the second of three volumes on the fundamentals of microprocessors and microcomputers. In this volume, we (that's you, me, and that gorilla) look into the details of the micro's own language.

Volume 1 covered the fundamentals of microprocessors needed for us to start understanding machine language programming. Volume 3 is a reference volume containing detailed descriptions of hundreds of popular and micro-related integrated circuits.

Why machine language? Because, as it turns out, virtually *all* winning and top performing microcomputer programs run *only* in machine language. The marketplace has spoken. It has not only spoken but is shouting: BASIC and PASCAL need not apply.

Volume 2 will show you the fundamentals of machine language programming through a series of *discovery modules* that you can apply to the microprocessor family and the microcomputer of your choice. Once you get past these modules and gain a deep understanding of what machine language is all about, then you can step up to the wonders of *assembly language,* which is really nothing but automated machine language programming that is made much faster, lots more convenient, and bunches more fun.

Volume 2 picks up at Chapter 6 in this continuing series. Here we look at address space and addressing concepts, as well as working registers and how they are used. Next is a study of system architecture, seeing what goes where in a typical microcomputer, with heavy emphasis on understanding system buses and how they work. From there we go into memory maps and on to addressing modes, those all-important methods a microcomputer's CPU has of accessing memory and its own working registers. We look at seven fundamental addressing modes that apply to most micros one way or another, either by themselves or in combination.

Address modes are then summarized in a group of quick-reference charts. Next come some stock forms useful for hex dumps, machine language programming, and assembly language programming. This chapter ends up with a toolkit that you can put together for machine language work.

Chapter 7 is the real heavy of this volume. Here we actually do lots of machine language programming. We use the "those #$!#$ cards" method, in which you work one-on-one with each individual op code as the need arises, again on the microprocessor of your choice. There is a series of nine *discovery modules* here. These are elementary programming problems that start with the simplest of

op codes and programming concepts and work their way up into some fairly fancy results, using practically all the available microprocessor op codes on the way. As we go through the modules, we also pick up details on flowcharting and using programming forms; measuring time and frequency; calculating branch values; using a stack; testing individual bits; creating text messages; using files, subroutines, interrupts, breakpoints, arithmetic, and much more.

We do not dwell on micro arithmetic because math uses of micros are not all that important when it comes to real programs doing real things for real people. Math on micros simply does not deserve the overblown treatment some texts give it.

While many examples are given that involve the 6502, you can easily do the discovery modules on any micro of your choice—4-bit, 8-bit, 16-bit, or whatever. All program problems and examples have purposely been done on a mythical and nonexistent trainer, so that you are forced to think things out on your own, solving your own problems in your own way on your own machine.

In Chapter 8, we take a detailed look at I/O, or input/output. We find there are four levels of I/O and then explore the two lowest levels in detail. At the device level, we check into parallel and serial ports, look at the different port types, and examine specific chips. Then we find out how to interface such things as keyboards and displays, using a minimum number of port lines.

Next, at the circuit level, we examine the simple circuitry needed to "amplify," "isolate," or "convert" micro port lines into signals powerful enough to go out into the real world with a vengeance. Here we include such things as transistor drivers, triacs, optocouplers, input conditioners, analog-to-digital converters, digital-to-analog converters, and things like that.

Chapter 9 both wraps up this volume and completes the "how" part of the trilogy. First and foremost, we check into the *micro applications attack,* a real-world problem-solving method that I use. It has been thoroughly tested and, above all, it works. Emphasis is placed on everything that has to be done *away* from the micro, using the "stickiest box" method to zero in on the real problem hidden inside what you are trying to do.

The micro applications attack is followed by some real-world problems that you can solve using this method. Project "F" is particularly challenging. Then we consider where you have to go from here. Finally, for those of you still wondering "What good is all this stuff?," we end the book with a list of sixty-three microcomputer ideas that you can immediately put to challenging, unique, and profitable uses.

DON LANCASTER

Contents

CHAPTER 6

Addresses and Address Spaces 9
Address Spaces—Working Registers—Architecture—Address Space Decoding—The Memory Map—The Programmer's Model—The Package to Albuquerque—Which Address Mode?—The Resource Sheet—The Micro Toolkit

CHAPTER 7

The Discovery Modules 113
What Is a Program?—Von Neumann Architecture—Machine Language Programs—Those #$!$# Cards—MYTH-1 Discovery Trainer—Flowcharting—NOP and JMP—Discovery Modules—Loading and Storing—Time, Frequency, and Clock Cycles—Flags—The IF Instructions—Calculating Relative Branches—Block Counting Method—Loop Use Rules—The Stack—Subroutine Uses—Absolute Short Addressing—.Y-Time Delay—User-Friendly Code—Passing Variables to a Subroutine—Bit Twiddling—Files—Interrupts—Breaks and Breakpoints—What? No Math?—Add and Subtract

CHAPTER 8

Interface and I/O 311
Micro Level Interface—"Less Than a Port" Outputs—Real Microcomputer Ports—Simple Parallel Ports—The 8212—The 6522—The Simplified I/O Diagram—Minimizing Port Lines—Serial I/O Ports—"More Than a Port" I/O—Open Collector Outputs—Circuit Level Interface—Output Circuit Interface—Output Conversion—Input Circuit Level Interface

CHAPTER 9

The Micro Applications Attack 407
Write a Brief Description of the Problem—Write a Detailed Description of the Problem—Partition Hardware and Soft-

ware—Assign Port Codes—Draw Timing Diagrams and Decision Trees—Make a Block Diagram and Flow Chart—Attack the Stickiest Box—Build Software and Hardware Modules—Prepare an Improved Flow Chart and Schematic—Write, Test, and Debug Your Code—Have a Knowing Outsider Test It—Annotate and Document Everything—Sit on It—Evaluate and Improve—Using the Applications Attack—Now What?—Sixty-Three Ideas

Appendix: Simplified I/O diagram 443

Index 445

This book is dedicated to microcomputer pioneers everywhere. You can tell them by all the arrows in their backs.

— six —

Addresses and Address Spaces

Have you ever been behind the scenes in a post office? There are lots of similiarities between what goes on there and what happens inside a typical microcomputer.

Our postmaster acts the same way a micro's CPU does when it decides what mail goes where. Large banks of boxes are available where users can go to pick up their mail. Any particular box might be for a family, for a business, for a club, or for a church, just as any particular location in a micro's address space can have various uses. These locations can be used for temporary or permanent storage of data and programs, or they can let you input or output to the real world.

Some post office boxes may be empty or unrented. Others may be seldom used. Yet others will be very busy and may even overflow if they aren't continuously emptied. In the same way, some locations in a micro's address space will be extremely busy, while others will not be used at all or may rarely see any action.

The rules at the post office say you have to use the postmaster to get something from one box to another. You aren't allowed to stuff something into someone else's box on your own. Most older microprocessors work the same way. Almost everything you do with a micro has to go through the CPU's "hands." Some of the newest micros do have very powerful "memory-to-memory" transfer features built into their architectures, but this is not yet common.

We see that the postmaster also has several sorting bins that simplify handling mail. Most pieces of mail have to go through one or more of these temporary stashes to allow sorting, routing, or forwarding. Some of the stashes are simple bins that can be used any old way the postmaster wants. Others have one special use, such as the safe for registered mail.

The CPU in a microprocessor also has its sorting bins. These are called the *working registers* of the micro. Working registers are involved in practically all micro actions. Some of these working registers are very general stashes that can be used any way you like. Others have one special use. Some microprocessors have lots of working registers. Others may have fewer working registers but will have very fancy ways of getting things between the registers and the address space. These fancy ways are called *address modes,* and we will see lots more on them shortly.

Buzzwords . . .

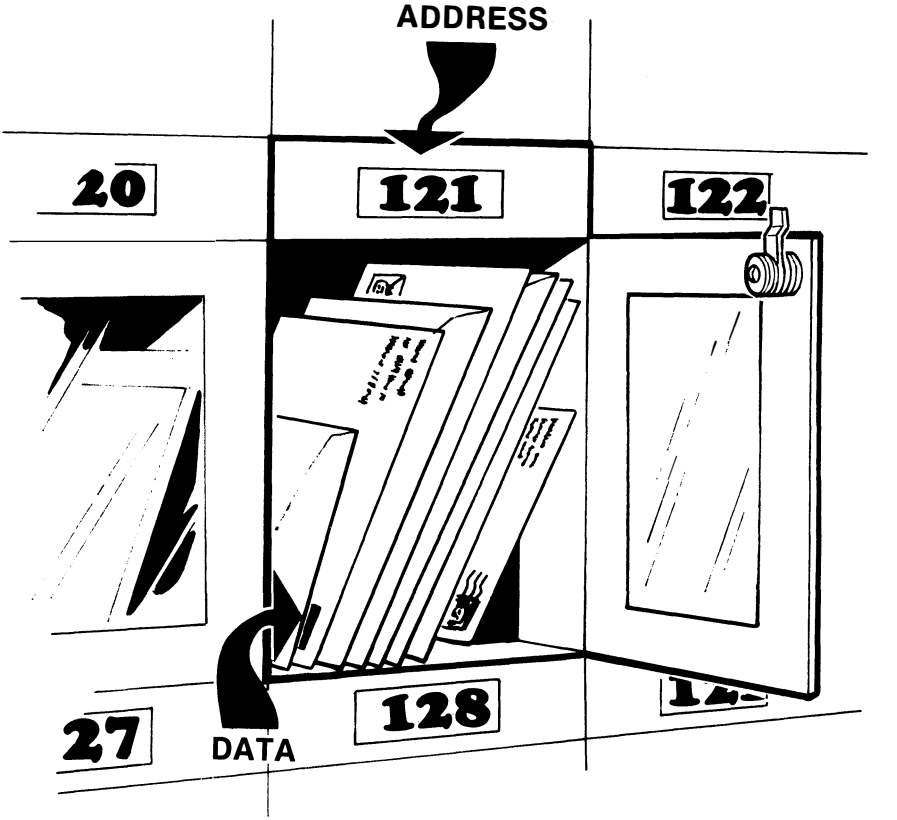
ADDRESS SPACE—The "reach" of a microprocessor's CPU. The total number of available locations the CPU can communicate with.

WORKING REGISTERS—Temporary stashes available inside the micro's CPU that involve themselves with practically everything the CPU does.

ADDRESS MODES—Ways for the CPU to get something into or out of a working register or an address space location.

In most micros, the address space is *outside* the microprocessor chip and the CPU while the working registers are *inside* the microprocessor chip. This is similar to the user boxes, which are available to anyone from the lobby, compared to the sorting bins, which are available only to the postal employees. Some single-chip micros do include some or all of their address space internally, but in general, the address space area is separate and different from the working register area.

Let's take a closer look at one of our post office boxes. We'll assume it's in a small western town where everybody goes to the post office to get their mail. A typical box looks like this . . .



Once again, an address is a location, and data is what goes in that location. Each address in a microcomputer must be unique. No mix-ups can be allowed. The addresses in the address space are often identified by a hex number. The working registers are usually identified by name or by a single letter.

The user box in a post office is like a single location in a micro's address space. How much mail this box can hold depends on the micro. In a 4-bit micro, we can put only four letters in a single address location. In the more popular 8-bit microcomputers, and in many locations of the 8/16 hybrid micros, we can put eight different letters in a box.

We can call a bill a zero and a check a one, and limit ourselves to letters that are only bills or checks. As we've seen, the eight bits of an 8-bit word have 256 different states, just as there are 256 possible combinations of eight letters that can be bills or checks. We can put any meanings on these states we want. An address location may hold a computer command, a piece of data, an ASCII character, a door in an adventure file, or almost anything we like.

So . . .

In a typical 8-bit micro, each location in the address space can hold one 8-bit word.

Unlike the post office, we never really remove the mail unless we are replacing it with something else. The process of *reading* an address takes a look at what is in the address and makes a *copy* of it to carry somewhere else. The process of *writing* an address destroys whatever data was in that location and replaces it with something new . . .

READING—Checking an address location to see what it contains. A copy of the contents is taken somewhere else.

Reading DOES NOT alter the contents of the location.

WRITING—Changing the contents of an address location by taking new data and storing it there. The old data is destroyed and gone forever.

Writing DOES alter the contents of a location.

Two obvious points here. First, there is no way to tell what is stashed in a location unless you previously put something there.

Locations are not "empty" till you fill them. Instead, previously unused locations will contain useless garbage. You should never read any location that you haven't previously filled with something useful. If you really want an area of memory to be all zeros or, say, contain the hex $20 code for all ASCII blanks, then you have to take time out early in your program to store the zeros or the chosen ASCII code values where you want them.

The second point is that you can write only to an address location that has some writeable hardware in it. You cannot write to Read Only Memory or to an empty or unused location.

Thus . . .

SOME ADDRESS SPACE RULES

( ) All address space locations ALWAYS have something in them.
( ) You must fill an address space location with useful contents before you try to use it.
( ) You can only write to an address space location that contains writeable hardware.

We now have an address space made up of lots of boxes. Each box can hold exactly one word of eight bits each. We already know what meanings we can put on the 8-bit words we put in any address location—anything we want to. The whole truth and beauty of micros is based on this extreme flexibility of making the data in a location be anything we want and fill any need we choose.

What physically goes into the address space? The obvious answer is hardware of some sort. It turns out that there are only four main types of hardware that you are likely to see in any particular address space location. These four hardware types are . . .

ADDRESS SPACE HARDWARE

( ) RAM
( ) ROM
( ) I/O
( ) nothing

RAM, of course, is memory that we can change fast and easy at system speeds. We use RAM for anything that is going to change, such as a working program, a data block, a message, the text file of a word processor, and so on. We can both write to RAM and read from it. Most RAM is volatile, so it will contain garbage on power-up.

ROM is memory that is more or less permanent. We use ROM anywhere we want to keep its contents for a long time. PROM, EPROM, and EEPROM are variants of ROM that we can occasionally change but will hold their contents for us during power-down times. The monitor and operating system are often kept in ROM so that they are always there.

You CANNOT write to ROM locations at system speeds. You can tell the micro to do a write, and it may go through all the motions for you, but it won't work. Write to a ROM location that contains an $A5, and you'll still have an $A5 when you are done.

I/O stands for Input and Output. We can pass information to and from the real world through suitable I/O *ports.* If the port is an *input only* port, we can only read this location with the micro. If the port is an *output only* port, we can only write to this location with the micro. If the port is *bidirectional,* we can read from those port lines set up as inputs and write to those port lines set up as outputs. Other locations can be used to control a bidirectional port, keeping track of which lines go where. These other locations are often found nearby in the address space. More details on this in Chapter 8.

The I/O ports that are located in the address space just like RAM and ROM are called memory mapped I/O . . .

MEMORY MAPPED I/O—Input and output ports that are located in the address space just like RAM and ROM.

The big advantage of memory mapped I/O is that anything you can do to a RAM or ROM location, you can also do to a suitable I/O port, since the CPU does not know what is out there in the address space. Any and all microprocessor systems support memory mapped I/O.

There is another type of I/O called *direct* I/O that is provided on some earlier 8080 school chips. While direct I/O can be faster and more easily decoded, it is limited to one micro family and has pro-

gramming restrictions. A direct I/O micro does not prevent you from using memory mapped I/O on it, and this is exactly what most people end up doing.

The final kind of hardware that we can put in a micro's address space is nothing at all. If this seems dumb at first, think about it for a while. Post offices usually have some unrented boxes. If they don't, they have to add boxes for new customers.

In a micro, there is often no reason to fill all the locations in the address space. In a simple application, 1K of ROM and a few dozen words of RAM may be all you will need. The leftover space can be saved for later expansion. Just remember not to write to or read from these unused locations.

Sometimes, the unused locations can be used to simplify decoding. For instance, if you have extra address space to burn, you could give a single I/O port 256 consecutive locations, any of which could be used to reach the port. This can greatly simplify the decoding hardware but can become a dangerous trap on a later expansion.

ADDRESS SPACE

Let's take a closer look at the address space of a typical micro. We've seen that the address space is the "reach" of a microprocessor, made up of all the possible locations into which we can put RAM, ROM, I/O, or nothing at all. We also know that the micro has working registers that are usually outside the address space but inside the microprocessor chip itself.

Each location in the address space of an 8-bit micro can store one 8-bit word for us. We are free to put any coding and any meaning on what we put into any location.

There are several popular sizes of address space. By far the most common and most important size is made from the 65536 address space locations in a typical 8-bit micro.

The number 65536 is the sixteenth power of two, so we can reach any point in this address space with sixteen binary address lines. As we will shortly see, these address lines are most often broken down into a pair of 8-bit words to simplify memory space access.

While a 65536 location address space is the most common, we can have larger or smaller sizes of microcomputer address spaces. Some single-chip microprocessors have an address space of only 4096 locations. This needs only twelve address lines and is popular for smaller systems or other dedicated applications.

We can also go the other way. One simple route is to have several *banks* of 65536 locations, just as there are several bays of user boxes at the post office. Banks are selected by a process called *bank switching*. Bank switching is a simple and effective way to double or quadruple the available address space without going to fancy hardware.

The new 16-bit micros have gone totally overboard on address space. Some of these have an address space of 16,777,216 locations. This is usually broken down into 256 *segments* of 65536 locations each.

Here are some typical uses of different sized address spaces . . .

ADDRESS BUS SIZES

SIZE	LINES	USES
4096	12	dedicated micros
65536	16	personal computers
262144	16*	business systems
16772216	24	heavy applications

*bank switched

We'll stick with a 65536 location address space for now, since it is the most popular as well as the baseline for everything else.

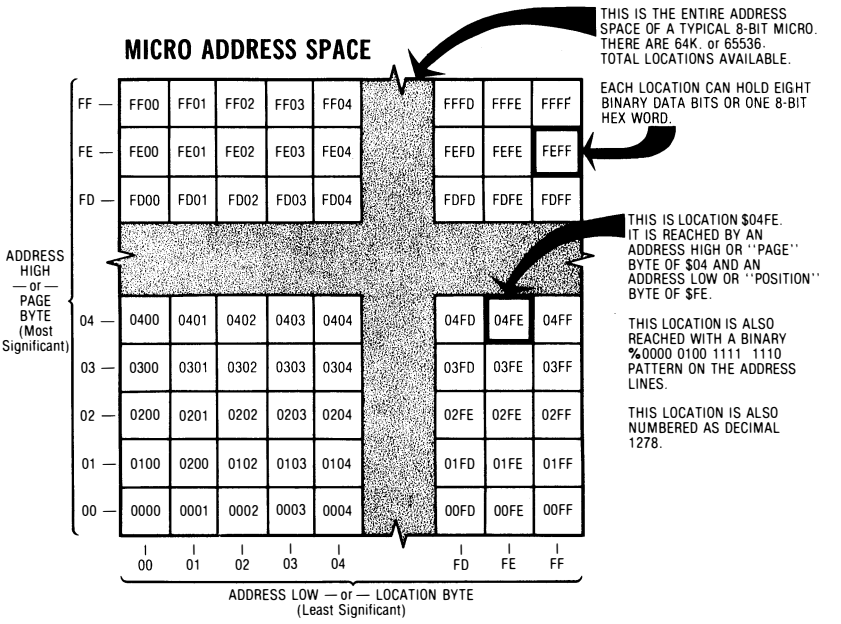
Each location in a micro's address space must be reachable by a unique address. Good old straight binary seems to be the standard and best way to reach a single location in the address space. For convenience, this binary address is normally called out in the form of a hexadecimal word.

So, we can number our boxes from $0000 through $FFFF, for the 65536 locations that specify the positions from 0 through 65535. To do this, we need a 16-bit binary number to call out the address. We will shortly see that this addressing number goes out some lines on an address bus to select a single location in the address space.

We could string all our addresses out in one long row. But even the postal service isn't this dumb. Note how the user boxes are in bays. Besides reaching the box by a particular number, we can also reach the box by going to a particular bay and then selecting a suitable row and column on that bay. The place where row and column cross on the selected bay is the box we are after. Mathematicians would call a two- or three-dimensional grouping of boxes a *matrix*.

The address space of a typical micro can be thought of as a humongous post office box bay that is 256 boxes wide and 256 boxes high.

Something like this . . .



We see that we have 65536 boxes and that these boxes are numbered in order from hex $0000 through $FFFF. Besides finding a box by its number, we can also locate any box by finding its row and column.

The two rightmost hex digits tell us which column the box is in. These two digits can have a value from hex $00 to FF, representing 256 possible positions. One 8-bit word is needed to call out 256 positions.

The column-selecting word is called the *low address* byte or the *position* byte.

The two leftmost hex digits tell us which row the box is in. These two digits have values of 0, 256, 512, . . . on through 65280, stepping along in exact multiples of 256. One 8-bit word is also needed here to call out the 256 different multiples.

The row-selecting word is called the *high address* byte or the *page* byte.

To recap . . .

( ) The 65536 locations of a typical micro's address space can be located with an address word of sixteen binary bits.
( ) The address is usually broken down into two address bytes of eight bits each.
( ) One of these 8-bit address bytes is called the address low byte or the position byte.
( ) The other 8-bit address byte is called the address high byte or the page byte.

Thus, we can say that some address location is in some position on some page. We can also say that the same address location has some low byte address and some high byte address.

For instance, the fourth location up and the fifth one to the right will have an address of $0304. The threes and fours result because we start with zeros rather than ones. We can say this address has a high byte of $03 and a low byte of $04. We can also say that this address is position four on page three.

A position here means one of 256 possible vertical locations on a page. A page means one of 256 possible horizontal rows, each of which holds 256 possible positions.

When you get around to actual machine language programs, you will find that the address usually appears in the listing *backwards,* with the low byte or location byte first and the high byte or page byte second. This sounds strange, but it has speed and program advantages.

To repeat . . .

On most microprocessor families, machine language instructions will use addresses "backwards" with the low address byte first and the high address byte last.

All of the documentation, assembler listings, and so on will show the address in its expected way. Only the actual machine language op-code listings will be backwards. For instance, on the 6502, one possible command to load the accumulator from address location $1234 will be "AD 34 12." The *AD* here is the op code for "load the accumulator from somewhere in the entire address space." The *34* is the position on a page or low address byte, and the *12* is the page or high address byte.

Both the 6502 and 8080 school micros use this seemingly backwards convention. A few oddball VCIWs, including Motorola's 6800, do the opposite from everybody else and put the addresses in the page-position form.

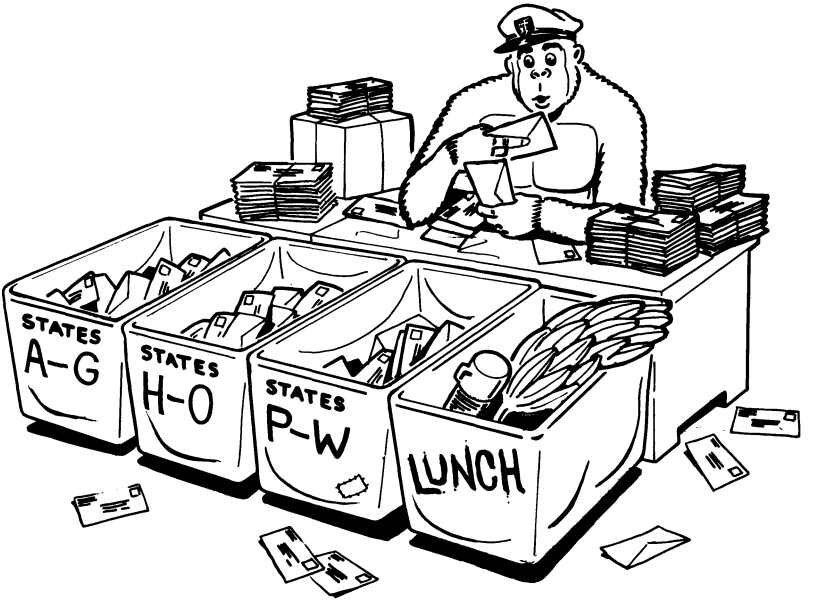
To review, the address space is the reach of a microprocessor's CPU that calls out the maximum number of places into which you can put RAM, ROM, I/O, or nothing at all. On an 8-bit micro, each of these locations can hold one 8-bit data word. This data word can be anything you like and used anyway you want.

Again, on a typical 8-bit micro, the total number of available locations in the address space is 65536. Other less popular address space sizes include the 4096 words of a one-chip dedicated micro, multiples of 65536 locations through bank switching, and the 256 segments of 65536 locations each used in some newer 16-bit micros.

Each location in the address space must have a unique address. These addresses are numbered in straight binary from 0 through 65535 and are usually shown more conveniently as four hex digits ranging from $0000 through $FFFF. Addresses turn out to be *much* easier to visualize in hex than in decimal, so practice hex until you have it down cold.

Any address can be reached through the sixteen lines of an address bus. Addresses are usually broken down into two separate 8-bit words. One of these words picks one of 256 pages of locations and is called the high address byte or the page byte. The remaining word picks one of 256 positions on a single page and is called the low address byte or the position byte. In most program codes on most microprocessors, these address bytes will appear in seemingly reverse order, with the low or location byte first and the high or page byte second.

Shortly, we'll find out how we decide what goes where in our address space. For now, let's go back to the post office and those sorting boxes.



The sorting bins in the post office are used to simplify handling of the mail. We have similar sorting bins in a microprocessor's CPU. These are a few words of special RAM that help the micro do useful stuff in an orderly and logical manner. These RAM bins inside the microprocessor chip are called . . .

WORKING REGISTERS

Working registers serve as a workspace or scratchpad for the CPU. They give a temporary place to put stuff being worked on. They give ways of keeping track of where you are in a program and where to go next. Other working registers keep tabs on what is happening and what kind of results you are getting. Still others provide orderly ways to count through a loop or to pick one of many entries out of a file. Still other working registers can show us where to go to get new material or where to put old results.

Since the working registers are inside the microprocessor chip, the CPU can quickly and easily get to them. It is usually faster and simpler for the CPU to reach a working register than for it to reach a location strung out somewhere in the address space.

Working registers typically are called out by a letter, such as A,X,Y or A,B,C,D, or possibly by a letter and number, as R0 through R7. Each manufacturer tacks its own name on things. The number of

available registers and the details of their exact use will vary from manufacturer to manufacturer. Often a micro that has only a few working registers will have very powerful ways of running around the address space, while micros that have lots of working registers tend to have weaker and slower ways of reaching the main address space. The newest micro chips give us the best of both worlds. They have the equivalent of lots of working registers and powerful addressing modes.

You can easily reach some of the working registers and put anything you like into them or take anything out of them. Others are harder to get at but are automatically taken care of by the instructions you give the CPU.

The newest microprocessor chips simply give you lots of on-chip RAM and let you use much of it any way you like. But most of the mainstream devices today have special and more-or-less committed working registers, each of which has to obey certain use rules.

One way to classify working registers is by how flexible they are. There are three main types of working registers . . .

TYPES OF WORKING REGISTERS

GENERAL USE—These can be used for anything you like and are usually involved in most of the micro's instructions.

An accumulator or A register is a typical example.

INTENDED USE—These have one thing they do particularly well but can also be used for other purposes.

An index register or an address register are examples.

DEDICATED USE—These have only one purpose and cannot be used for anything else.

Examples include a program counter or a flag register.

The three main kinds of working registers include completely general ones that can involve themselves with just about anything. You can use these any way you like. Then there are the intended-use registers that have one big purpose in life. You can use them to do their thing, or else you can use them for other stuff if you don't need their specialty.

Finally, there are dedicated-use working registers that are forever restricted to do one task for the micro. You may not be able to get at these directly or else the job these registers do is so important that your program will bomb if you mess with them.

Regardless of type, all the working registers are simply RAM or read-write memory. The difference between the various types lies in how the working register interacts with the microprocessor's CPU, with you as programmer, and with the address space.

Let's look at some typical working registers and see what they can do. Later we will pick up more specific details on how certain registers in certain families work. For now, we'll stick with the big picture.

The most obvious general-purpose register in a CPU is usually called an *accumulator* . . .

ACCUMULATOR—A general-purpose working register that usually holds actions and results of CPU activities.

The accumulator is often used to receive data from the address space, to hold intermediate results of calculations, and to be the source of data to be stored in the address space. Most of the commands that involve arithmetic, logic, or testing will end up with the result in the accumulator.

The name dates back to the dino days when computers worked on a single serial bit at a time instead of dealing with whole words. One very expensive register was built to accumulate the results painfully, bit by bit.

In traditional computer architecture, the single accumulator was a narrow funnel through which everything had to go. Modern micros often have other places to put things besides in the accumulator and this roadblock to quick and simple programs is fast being removed.

Most micro families have at least one "main" accumulator as well as other handy places to put things. In the 6502, there is an A register that handles most of the work, but there are two other 8-bit registers called X and Y. These also can read from or write to the address space. With some limits, the X and Y registers can also make comparisons and do some logic operations.

In the 8080 family, there is a main accumulator, along with companion B, C, D, and E registers, while the 6800 has a pair of accumu-

lators called A and B. The 8048 has an accumulator as well as sixteen R registers that can easily swap roles as needed.

The traditional single accumulator computer is horribly out of date, and use of accumulators is waning. The newest micros have direct register-to-register actions that are far more flexible and greatly simplify doing several things nearly at once.

As an example of how an accumulator works, suppose you want to add two numbers. You reach into the address space and get one number and load it into the accumulator. You then reach elsewhere into the address space and get a second number and add this to what is already in the accumulator, replacing the first number with the sum of the two. This final result in the accumulator then can be stored back somewhere in the address space.

The accumulator usually has fancier capabilities than any other single register. Besides addition and subtraction, you can often shift bits right and left, rotate them in either direction, compare values, complement a result, increment, decrement, do logic, clear, and so on. The accumulator may be the only general-use register that has access to a special memory area called a stack. It is used for subroutines, interrupts, and temporary storage. Because of this, the accumulator gets involved more often than any other working register in the microprocessor system.

There are usually instructions called *transfer commands* or *moves* that let you swap things between the accumulator and any other register that happens to be handy. These transfer commands can be faster and shorter than those needed to reach any location out in the address space.

You may also find other general-use registers in your micro. These may be secondary accumulators or simply places to store things. They may or may not have all the power an accumulator does. It depends on the microprocessor and what you want to do with it.

An example of an intended-use register is the *index register* . . .

> **INDEX REGISTER**—An intended-use register that usually is used to count the number of trips through a loop or point to the contents of a certain location in a file of data.

A machine language program often needs some way to do the same thing over and over again for some number of times. A programming concept called a *loop* is usually involved. To use a loop,

you put a number somewhere and then count that number down each time you go through the loop. The program that uses the loop may simply be stalling for time or may have to do things a certain number of times or continue until some special result occurs.

Since our accumulator will most likely be busy doing other things for us during loop times, we need some other place to put the number that we are going to count down for the loop. One possible other place is the *index register.*

To use an index register in a loop, you put some number in it and then do your loop once. You then decrement the number, test for zero, and do the loop again. You keep this up till you really get to zero, and then the zero test gets you out of the loop and on to something else.

You could also use an index register to count up to some number, but there are several good reasons that counting down is far more popular. One reason is that it is far easier to test for zero than any other value. Another is that the program is easier to modify if you decide to change the number of trips through the loop. A final reason is that when you count down, the index register will always hold the number equal to the remaining number of trips needed.

Thus . . .

When an index register is used to count the trips through a loop, it most often counts down to zero rather than up to some number.

Another use of an index register is as a way to get something out of a file. Say you have a data list stashed somewhere in RAM. Rather than specifying the exact address every time you need something out of the list, it's easier to say, "Go to the start of the list plus an index value." If you want the third entry in a list, you put an 02 in the index register and then tell the micro to look at the starting address plus 02, and so on.

Why 02? Because the first address is START+00, the second is START+01, and the third is START+02.

We'll look at more details on this when we get into address modes. What indexing does is greatly simplify how we reach into a file and pick out data.

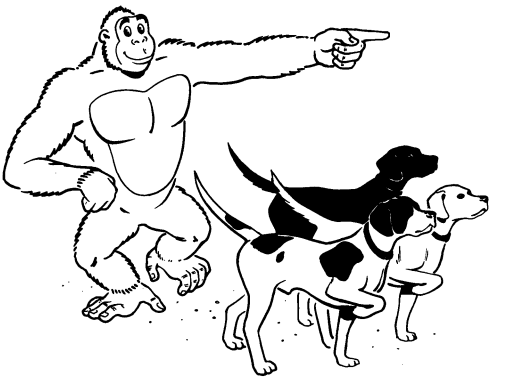
There are two popular *widths* of index registers. An 8-bit index register can only count down from 255 or reach 256 locations in a file from a given starting address. The X and Y registers of the 6502 family are typical. You can also have 16-bit wide index registers,

such as the X register in the 6800 family. A 16-bit index register can reach any point in the 65536 location address space but may be harder to load or be otherwise limited in its abilities.

Some microcomputer families, such as the 8080 gang, do not have index registers as such. You can still do loops and pick things out of files with these micros, but you have to do it with something else. Something else is often called an *address register*.

The reason an index register is an intended-use register is that you are free to use it for anything you like if you don't happen to need it for a loop or an indexed file pickoff. Index registers typically can do some but not all of the things the accumulator can. Eight bit wide index registers can be loaded from and stored to the address space and often can support comparisons and other logic operations. It is usually easy to transfer things between the accumulator and 8-bit index register and vice versa.

There's a whole class of working registers that can serve either intended or dedicated uses, depending on how they are connected. These working registers are called *pointers* . . .



So close. Very close.

Actually, the dogs are used to point to somewhere else. We are not so interested in the dog itself as in where the dog is pointing. Let's try it one more time . . .

POINTER—A memory location that holds an address, rather than data.

Pointers are used to show a location where data is to come from or go to.

A pointer holds an address for us rather than data. It is used whenever you want temporarily to remember where to go to get something or where to go to put something.

The most common intended-use pointer is called an address register . . .

> **ADDRESS REGISTER**—An intended-use register that holds an address for us. Data will be fetched from or put into the address pointed to.

The 8080 family has a pair of 8-bit registers called the H register and the L register, standing for *High* address and *Low* address. If you want to use these as an address register, you put in the address where you want to get data or where you want to put data, and then later instructions will tell the accumulator, "Put a copy of what you have in the location pointed to by the HL register." The sixteen bits of the HL register pair can point anywhere in a 65536 location address space.

Using an address register has two advantages. The big one is that you can calculate or change where you want to store things as you go along, rather than being stuck with absolute values locked into the program. The second advantage is that a faster and shorter instruction can be used to store something at an address pointed to by an address register, compared with spelling out exactly where in the address space you are to go.

The 8048 has four address registers, called R0, R1, R0', and R1'. These are intended-use registers since they are the only ones that are allowed to point to an address. If you do not want to put an address in any of these, you are free to use them any way you like. The same holds true of the H and L registers in the 8080 family. Although you can use H and L as general-purpose registers, this register pair is intended to be used as an address pointer.

The 6502 and 6800 families do not have address pointers. Instead, they use their powerful index registers. The 6502 also has a very flexible way to let a pair of ordinary RAM locations down on page zero serve as an address pointer.

This "index-vs-address-register" capability is typical of the many differences you will find among major micro families. They all can do almost anything in some way or another, but some will offer powerful ways to do one thing very well and others will have strong advantages in other areas.

Your particular micro may have additional general-use registers available. The newest micros bypass the register problem completely by giving you lots of RAM locations on chip that you can use any way you like in a totally general way. These RAM locations are also inside the address space. The idea is to make things as fast and as flexible as possible without tying you down to doing things exactly as the chip designers ordained.

Let's now look at some dedicated working registers. These are pretty much locked into doing one job and are not available or usable for anything else.

One of the most important dedicated working registers is a pointer called the program counter . . .

PROGRAM COUNTER—A dedicated-use working register that points to the starting address of the next instruction.

Just as you can't tell the players without a program, a micro always has to know what instruction it is working on and where to go to begin doing the next instruction.

A dedicated working register called the *program counter* does the job for us. After each instruction is complete, the program counter figures out where to go for the next instruction. This is not as simple as it sounds, since instructions may take one, two, three, or even more sequential locations in the address space, and since the program counter always has to know when the present instruction ends and the next one begins.

Sometimes the program counter will be given a new address far away from where it happens to be. This happens when we jump somewhere else, or when we temporarily jump to a series of *subroutine* instructions somewhere else, or when we stop what we are doing to service an outside world *interrupt*.

There is usually no immediate way to write to or read from the program counter. The CPU will do what you tell it to and, as the result of these instructions, will end up telling the program counter where to remember to go for the next instruction address. Thus, doing anything to the program counter ends up as a "Mother, may I?" game between you and the CPU.

Jumping to a new location is one sure way to set the program counter to a new value. This is the usual way for a monitor to start running your program for you.

The program counter has to be big enough to point to every possible location in the address space. Thus, on a typical 8-bit micro with a 65536-location address space, you need a 16-bit program counter. A dedicated micro with a 4096 location address space will need a 12-bit program counter.

There is another dedicated-use working register used to point to a set aside group of special locations somewhere in your system RAM. This one is called a . . .

STACK POINTER—A dedicated-use working register that points to the next available location in a special RAM memory area called the stack.

We'll learn lots more about the stack later. For now, a stack is a temporary stash somewhere in RAM that you can quickly shove things onto. The stack is *not* random access. Instead, the last thing onto the stack is the first thing you get out, sort of like storing dishes on a shelf.

Stacks are used to remember return addresses for subroutines and to remember both the return address and the exact condition of the CPU for interrupts. They are also a handy place to shove something temporarily that you will soon want back.

The length of the stack depends on the micro, and can be as short as eight words for a dedicated micro, on up through the entire 65536 words of the whole address space. The 6502 family has a stack family that uses up to 256 locations on page one of memory.

The stack pointer has to remember where the next available location in the stack is. This pointer has to be as wide as needed to point to all possible stack locations. In a dedicated micro, part of a word will do the job. In a 6502, a full 8-bit word is needed, to which a "hard-wired" 01 is added in front to always reach page one. This guarantees that the stack always stays on page one. A runaway program can destroy the stack, but not the entire memory space.

In the 6800 family, the stack pointer is a full sixteen bits wide. With this wide a pointer, a runaway stack will take everything else with it.

The stack pointer's address moves around as you use it. The CPU will automatically fill and empty the stack as it services subroutines and interrupts. There is usually some way to initialize the stack pointer to some location and some way to read exactly where the stack pointer is pointing.

The key ideas here are that there is a dedicated register available called the stack pointer that points to the next available location in the stack; and that we have ways of setting this pointer, reading the pointer, and automatically moving the pointer around as the stack gets used.

One important detail . . .

The stack pointer is nowhere near the stack. The stack pointer is a dedicated working register in the CPU.

The stack itself is a bunch of RAM locations out in the address space somewhere.

Thus, the stack is a collection of RAM locations. How this RAM gets used is decided by the stack pointer working hand in hand with the CPU.

That just about covers the pointer type of registers that hold some address for us. Every micro has one final special dedicated-use register. It's called the flag register . . .

FLAG REGISTER—A dedicated-use working register that holds all of the present conditions of the micro for us.

Flag registers are also called PROCESSOR STATUS registers.

You'll find out later that flags are like the idiot lights on your car. They tell you that some condition exists. If you want to, you can test a flag and make a decision based on it.

Each individual flag is usually a single bit wide. For instance, most micros will have a *zero* flag that tells you if the last operation, whatever it happened to be, ended up with a zero result. Most micros will also have a *negative* flag and a *carry* flag to aid in arithmetic, logic, and other tests.

There can be lots of other flags that vary from micro to micro. One may have a *decimal* flag that automatically keeps track of decimal versus binary arithmetic. Others may use a *half carry* flag to let you repair a binary result into its decimal equivalent. Some micros have flags you can use any way you like, and others have flags to keep track of overflow problems on signed binary arithmetic.

The most useful thing about flags is that you can test them and you can use them to control what the micro is to do. Each flag is a single bit and each behaves differently according to some rule or rules. Some flags you can set or clear yourself. Others are controlled only by the CPU. It depends on the individual flag. We will be seeing much more on flags later.

Just as you can group the idiot lights on a car's dash, you can group all your flags into a single word, simply by putting them side by side. When all the flags are in a single word, we call it a *processor status* word. The processor status word tells us the exact condition of the micro as of right now.

One important use of the flag register or the processor status register is to remember where we are if we are interrupted. Should an outside world interrupt arrive, we shove the program counter and then the flag register onto the stack. These two things together will let us pick up where we left off.

There are always ways to read your flag register and ways to get a flag register onto or off the stack.

To sum up, working registers are special RAM locations inside the microprocessor chip. These registers serve as temporary stashes where the CPU can work on problems and keep track of where you are and where you are going. The number and types of working registers will change with the micro family. Enough registers and enough ways of running around the address space are provided so that almost any microcomputer can do almost any task.

There are three main types of working registers. These are general-use, intended-use, and dedicated-use registers. The accumulator is the most common, the most often involved, and the most powerful general-use register. Other general-use registers may be provided to take some of the load off the accumulator.

Examples of intended-use working registers include: index registers that can be used to count the number of trips through a loop or pick a value out of a file; and address registers that point to some location in the address space where data is to come from or go to.

A working register that contains an address rather than data is called a pointer. The program counter is a dedicated-use working register that keeps track of where the next instruction is to come from. The program counter is wide enough to reach any point in the address space. The stack pointer is another working register that holds the address of the next available stack location. A stack is some area set aside in RAM that serves as a temporary stash on a last-in-first-out basis and is used for temporary storage and to keep track of subroutines and interrupts.

One final dedicated-use register is the flag register or process status register. It keeps track of the exact condition of the microprocessor at any given time.

More on all this later. Right now, we are just seeing what working registers are and what needs they can fill.

We now have some address space and some working registers. How are they related? To answer this, we have to take a close look at . . .

ARCHITECTURE

It would be very nice if a microprocessor had no personality at all. Ideally, the micro should behave like a new canvas just placed on an easel. A canvas by itself has little personality but with your personal value added, it becomes a custom work of art. In much the same way, a microprocessor should be able to do anything you want in any way you want. The micro should do this without any special use rules or other restrictions.

Unfortunately, most microprocessors aren't nearly so flexible, although the newest ones are coming close. Almost all microprocessors have distinct personalities. They may do one thing very well but do other tasks only with difficulties or hassles.

Some micros have powerful address modes combined with few working registers. Others feature the exact opposite. Some have lots of bit manipulation commands that are handy for industrial control uses. Still other micros are very strong in doing fast arithmetic, and others easily handle the strings, files, and decimal arithmetic needed for business applications.

We call the arrangement of resources within a micro the architecture . . .

ARCHITECTURE—The arrangement of resources within a micro.

You'll find two different architectures. One of these is the micro*processor* architecture that decides what is available inside the microprocessor chip itself. The second is the micro*computer* architecture that tells you what the whole system has in the way of memory, I/O, user access, and other available resources.

For instance, the architecture of the microprocessor tells you how many working registers are available, what the total available address space is, what width buses are provided, and similar chip-level features. The architecture of the microcomputer tells you how much RAM, ROM, and I/O are in the address space, the system meanings put on certain locations, and how you can reach specific address slots.

Something like this . . .

MICROPROCESSOR ARCHITECTURE—The arrangement of things inside the CPU, including the number of working registers, bus structures, and processing details.

MICROCOMPUTER ARCHITECTURE—The arrangement of things inside the entire system, including the amount of RAM, ROM, and I/O, the access rules, and overall system organization.

In picking a certain microprocessor chip, you are pretty much stuck with the architecture locked into it. If you are designing your own microcomputer system, you are free to arrange the overall system microcomputer architecture in almost any way you like. You must, of course, keep what you are doing compatible with the use rules and architectural limits of the CPU.

Two simple architectural resources are the *programmer's model* and the *memory map*. The programmer's model gives you a quick picture of the microprocessor architecture. The memory map shows you a simplified layout of the overall microcomputer system architecture . . .

PROGRAMMER'S MODEL—A simplified picture that gives you a quick overall look at the microprocessor architecture.

Programmer's models will show you the number of working registers and how they are used.

MEMORY MAP—A simplified picture that gives you a quick overall look at the microcomputer architecture.

Memory maps show you how much RAM, ROM, I/O, and unused expansion space you have available and where these are located.

Later in this chapter, I'll show you how to build up a *micro toolkit* that will give you many of the weapons you need to understand and work with the micro of your choice. The programmer's model and the memory map will be two of your first-line tools. More on these shortly, but first, let's find out what system architecture is all about.

What does an architect think about in designing a new home? Maybe this . . .

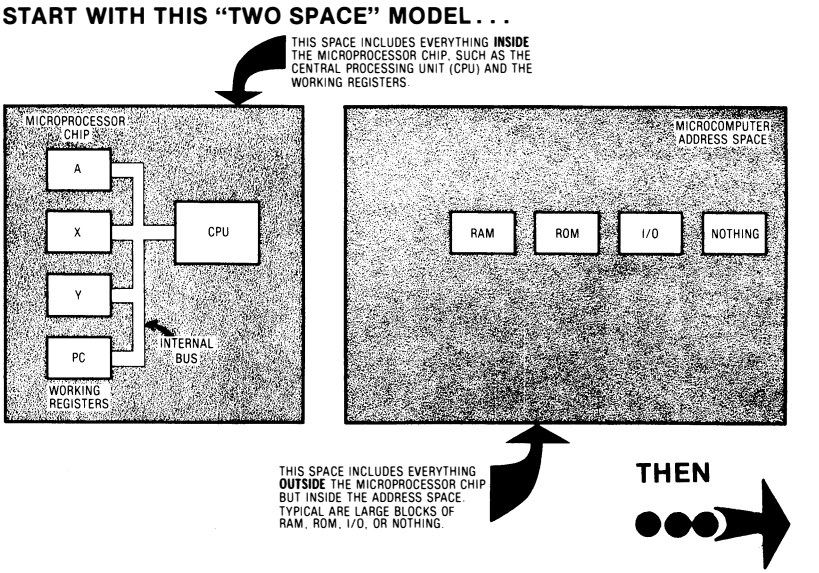
A house architect will take into account the personality of the owners, the available budget, the location, the climate, the types of materials, zoning rules, and lots of other obvious things. The architect does this to end up with a home that suits both the owners and their budget.

In much the same way, a microprocessor architect starts with a certain amount of silicon and a set of processing rules. Within those limits, the architect tries to come up with a microprocessor architecture that will do lots of good things for as big a market as possible.

Just as most homes are different from each other, most microprocessors also have their own architecture. Differences will be greatest between the three micro schools, but each and every micro has its own own unique structure.

Let's paint a big picture of a general micro architecture that is typical of what you can expect. It turns out that many of the general features of micro architecture are pretty similar, regardless of the device. Let's home in on these first.

We will assume that our microcomputer uses only one microprocessor. Some newer systems add a second or even a third slave micro chip to offload tasks such as video display, animation, sound, speech, keyboard service, or printer spooling. But let's keep it simple for now . . .



We see that we have a block called a microprocessor. This microprocessor contains a CPU, short for Central Processing Unit. The CPU acts as a postmaster to control totally what goes on where in the system. The CPU has immediate control over a group of working registers. How many of what kind depends on the micro you have chosen and is shown by the upcoming programmer's model.

You may have few or many of these working registers. We've seen how these are classified as general-use registers (such as accumulators), intended-use registers (such as index registers), and dedicated-use registers (such as program counters and stack pointers).

The working registers are usually inside the microprocessor's CPU but outside the address space. Working registers are usually identified by a letter or other callout rather than by an address in the address space. Although a few newer micros overlap address space so that RAM in the address space can be used as working registers, most micros don't. For now, let's assume that the address space is separate from the working register area. So, typically . . .

Working registers are usually inside the CPU but outside the address space.

One limit of this two-area workspace is that you can't directly reach a working register without the CPU's help. On the other

hand, the CPU can easily get to these handy stashes without having to hunt all over the address space for them.

Our address space contains RAM, ROM, I/O, and unused empty areas. We know by now that RAM is memory that you can change quickly and often but that is usually not permanent. ROM is memory that is more or less permanent. I/O is input and output that give our micro ports for real-world access. While the newest RAM and ROM are getting more and more like each other, it still is safer to assume that RAM and ROM are physically different devices. You will normally use RAM for things that change (such as a program or data file) and ROM for stuff that has to be permanent (such as a monitor or operating system).

The RAM, ROM, and I/O are usually grouped into blocks. Micro systems often use much more RAM than ROM. These blocks also depend on the size of available chips. For instance, a 16K block of RAM is a very common module size for older personal computers. For more RAM, you usually add increments of 16K to bring the total to 32K, 48K, or more. ROM tends to be in blocks of 2K or 4K and expands in multiples of 2K. The reason that the ROM blocks seem smaller is that most ROM is byte wide, containing a full 8-bit word at each internal address location. Most RAM is a single bit wide, so that a 16K block of RAM may take eight different chips, one for each bit in the 8-bit word. If you wanted a full 16K of ROM, you would also need eight chips, only this time it would take eight chips of 2048 bytes each.

Newer microcomputers will use 64K or larger RAM or ROM chips.

The area in the address space reserved for I/O is usually very much smaller than that reserved for RAM and ROM. Few micro systems need more than a handful of I/O ports. With a 4K space set aside for I/O, you could have 4096 different ports of eight bits each. This is vastly more than you normally would ever use.

Much of our address space could remain empty. If you are building a simple system such as a solar panel controller, you will still use the same microprocessor everyone else uses, but you use only small amounts of RAM, ROM, and I/O. The unused area in the address space is ignored. You can use this for later expansion, or you can use empty spaces to simplify the decoding process needed to access your existing RAM and ROM.

For instance, you can use only the bottom eighth of your address space and ignore the top three address lines in your decodings. Other stunts like this are possible, but they may cause trouble on later expansion.

What goes where in the memory map is decided by the microprocessor chip itself. In the 6502 school, it's easy to get to memory page zero (addresses $0000–$00FF) and the handy stack storage area

is always on page one ($0100–$01FF). This tells us that it's smart to put RAM in the bottom of the address space. Like most micros, the 6502 also needs vectors to decide where to go on a reset or one of two possible interrupts. These 6502 vectors always go at the very top of the address space at locations $FFFA through $FFFF. These locations are best kept in ROM if you want to keep control of your system.

So, the 6502 school wants you to put RAM in low addresses, starting at the bottom, and ROM in the high addresses, working down from the top. The I/O, by default, goes in the middle. Chips in the 6800 family have similar needs with RAM low and ROM high.

The 8080 school wants you to do the exact opposite. Low addresses are saved for interrupt and reset vectors that normally must go in ROM. The tradition here is to put ROM on the bottom, RAM on top, and, once again, I/O in the middle. The 8048 family usually has a small address space of 4K, split into a low 2K of ROM and a high 2K of RAM. If you are using less, you build ROM from the bottom up and RAM from the top down.

Don't worry too much about these special rules just yet. The point here is that the microprocessor chip you have chosen sets definite limits on what goes where in the address space.

Here are some address space rules . . .

( ) The address space contains ROM, RAM, I/O, or nothing.

( ) The RAM, ROM, and I/O are usually in large blocks set by the chip sizes. 16K is a typical older RAM block while 2K or 4K are typical ROM blocks.

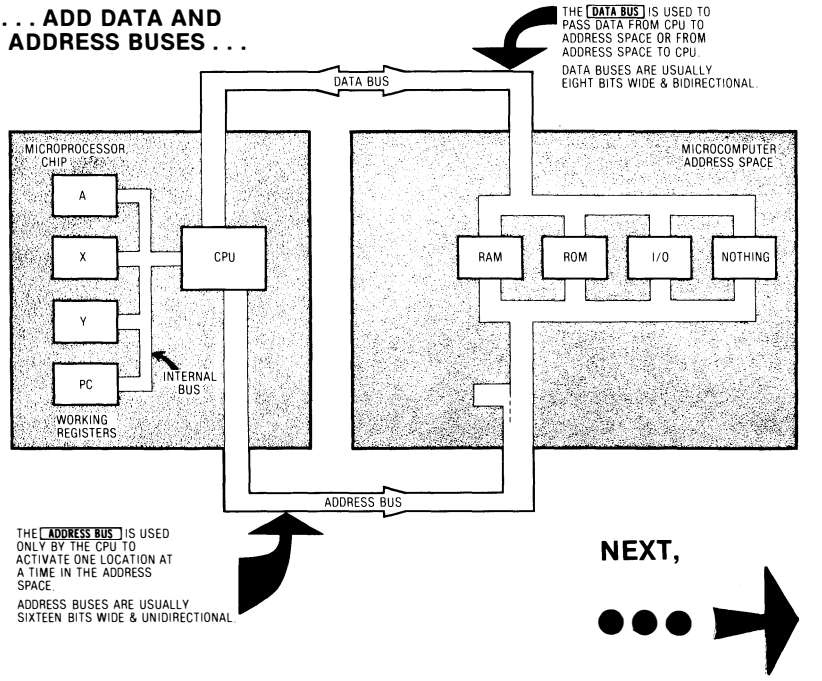
( ) The arrangement of the RAM and ROM blocks depends on the micro family. The 6502 and the 6800 families put RAM on the bottom and ROM on the top. The 8080 and 8048 families do the opposite.

( ) The address space need not be completely filled. Dedicated controllers will leave many empty locations. Other uses may save space for later expansion. A technique called bank switching can be used to overfill the address space, working with as many larger modules as needed.

We'll pick up more details on what goes where when we look at the memory maps. For now, the important points are that there are blocks of RAM, ROM, and I/O in the address space and that their arrangement depends on your specific microprocessor.

So far we have two separate boxes. We have the microprocessor box with its CPU and working registers. We have the address space with its blocks of RAM, ROM, I/O, or empty space. To do anything useful, we have to be able to communicate between the two.

We already know that we use bus structures to talk back and forth in a microcomputer system. Let's add two buses to our architecture. One of these is a data bus, used to pass information back and forth, and a second is an address bus, used to activate only one selected location in the address space at any time . . .



Let's look at the data bus first. A data bus has to be able to pass the contents of something in the microprocessor to the address space, or else handle the opposite task of getting something from somewhere in the address space and putting it into one of the CPU's working registers.

The width of the data bus is usually equal to the word size of the microprocessor. Thus, a typical 8-bit microcomputer has a data bus that is eight bits wide. A 16-bit microcomputer has a data bus that is sixteen bits wide.

Note that there are times when data must go from the CPU to address space and other times when information must go from address space to the CPU. The data bus must be able to work in both directions. Thus we need a bidirectional data bus . . .

The DATA BUS is used to pass information between the CPU and the address space.

The DATA BUS is eight bits wide on an 8-bit microcomputer.

The DATA BUS is bidirectional since it must work both ways.

The data bus is used to pass information between locations in the address space and working registers in the microprocessor. The data bus is normally controlled by the microprocessor's CPU, but it has to work both ways and be able either to get stuff from the address space routed to the CPU or vice versa.

How do we know where in the address space to go to get something? We already know that each location there has a distinct address, numbered in straight binary and called out in hexadecimal. In a microcomputer with a 64K address space, the 65536 addresses go from $0000 to $FFFF. These addresses are often located with two 8-bit words. One word is called the high address byte or the page byte. The other is called the low address byte or the position byte.

To find a specific location in the address space, we have to provide an address. This address goes out on some lines that are called, of all things, an address bus.

The number of lines needed on an address bus depends on the size of the address space and has nothing directly to do with the data word size. For instance, the 4K address space of a dedicated controller can be reached with twelve address lines for addresses $0000 through $0FFF. The most common address space is 64K, reached with sixteen address lines to grab addresses $0000 through $FFFF. Newer micros may have an extended address space as large as sixteen megabytes, reached with sixteen address lines and up to

eight *segment* lines, over an address range of $00 0000 through $FF FFFF.

Since the CPU must always be in charge, the CPU is usually the only thing allowed to activate the address lines. Thus, an address bus is normally unidirectional, going only from CPU to address space.

Summing up . . .

The ADDRESS BUS is used to select an address in the address space for later access by the microprocessor's CPU.

The ADDRESS BUS width is decided by the size of the address space. A 64K address space needs sixteen address lines in the bus.

The ADDRESS BUS is always under control of the CPU and thus is unidirectional.

Where you want to go is decided by the address bus. What you want to get or put in a certain location then goes in or out via the data bus.

Address buses and data buses are normally separate and on separate pins. A few micros use the same pins to route data and addresses on a time-shared basis. This is called a *multiplexed bus* . . .

MULTIPLEXED BUS—A bus that can have data and addresses on it at different times.

System timing must be able to sort out addresses and data as needed.

Multiplexed buses are used by some VCIWs and by many 16-bit microprocessors.

On a multiplexed bus, you take turns doing things. The big advantage is that you save lots of package pins, particularly with 16-bit micros. The big disadvantage of a multiplexed bus is that you need special timing circuits on each end to separate and route the

address and data commands. Another disadvantage of bus multiplexing is that you may have to use special and nonstandard chips made by one manufacturer rather than using industry standard parts. Interface to ordinary everyday RAMs and ROMs may take extra parts and extra hassle.

Still another disadvantage of multiplexed address and data buses is that there is always a speed penalty for their use, since time has to be taken to sort things out at both ends of the bus. A microprocessor that uses multiplexed buses is inherently slower than one that does not.

Another version of a multiplex bus gives you either eight bits of data or the eight low bits of an address. Separate address lines are provided for higher addresses. This technique is used in some smaller microcomputers.

Multiplexed buses are becoming popular on newer micro chips, particularly 16-bit devices. Multiplexed buses are an obvious system-level complication.

Another address bus complication may rear its ugly head on certain micros. The address bus has to contain valid addresses only while the CPU is actively trying to put something into or get something out of an address slot. There may be garbage or other signals on the address bus at other times.

The 6502 school has far and away the cleanest address lines of all popular micros. Addresses are always there and always output. The 8080 school doesn't address during the first cycle of some instructions. Instead, a special system status word is output to identify what the machine is about to do. Thus, if you look at address lines on an 8080, there will be great holes chopped in them at the beginning of each instruction.

The 6800 family disconnects the address bus for half of each cycle. This means you have valid addresses half the time and garbage the other half. This same feature can be added to the 6502 family by adding some tri-state drivers to the address lines. Using the address bus for only half of each cycle can have a very powerful advantage. Other microprocessors or other hardware can access and share the address space during the time the main micro doesn't need access. This process is called Direct Memory Access, or DMA. Two examples of things you can do when you chop up the address bus signals this way are to transparently drive a video display or to share operations between a pair of micros.

These details are rather technical and very system specific. What you need to know at this point is that address lines, except on the 6502, may not be as nice and neat as you'd expect when you look at them on a scope.

A reminder . . .

Address lines may not be totally "clean" in some micro families.

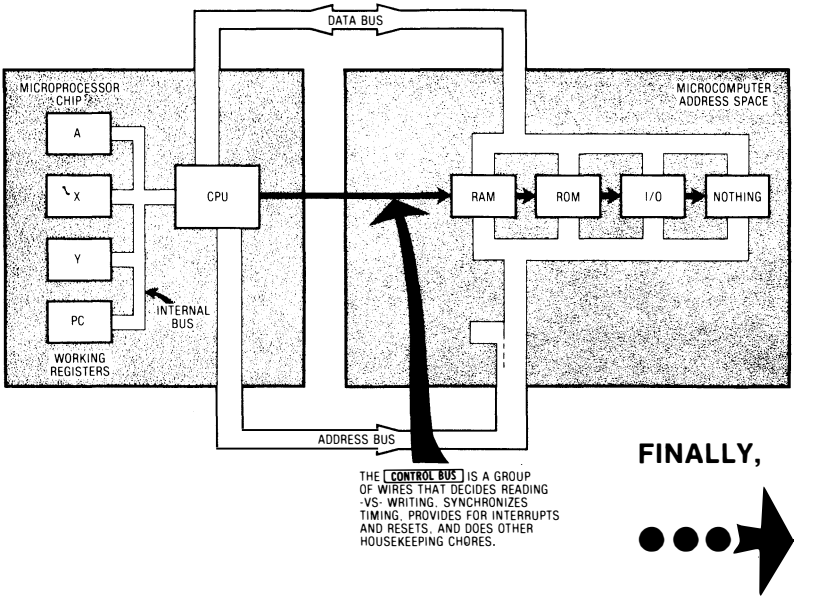
During the times that an address is not actually needed, some other signals may go over the address bus.

These other signals may tell you the status of the CPU or may be used for direct memory access, shared processing, or video display.

For now, we'll ignore all these added hassles. We'll assume that we have a separate address bus that always has nice clean addresses.

Let's add one more bus to our system architecture. This one is called a control bus . . .

. . . AND ADD SOME CONTROL LINES . . .



The details of the control bus very much depend on which microprocessor you are going to use. The control bus contains a group of lines that keep track of the essential timing and control information that the micro needs.

One control line on the bus tells whether we are going to *read* or *write* to RAM memory. This one is sometimes called a R/$\overline{W}$ line. It is obviously important to be able to tell whether we are reading from or writing to a location. The R/$\overline{W}$ line is used to control hardware inside each RAM that sets things up for transfer in the direction you want to go.

Another control line is the system *clock*. This is a high frequency signal that is the master crank for the microprocessor. We'll look at more details on this when we get to system timing. The system clock lets you lock everything together. This gets important when you are using fancy peripheral chips or are address-pin multiplexing dynamic RAM. Sometimes several different *phases* of the system clock will be available and may be labeled $\phi 1$ and $\phi 2$ or something similar. Different clock phases are used for different timing needs. Timing on clock phases can be very critical. In particular, phase overlap must be zero.

The *reset* line is another member of the control bus. The reset line gets everything restarted properly on power up. When the microprocessor's CPU is reset, it goes into a known state of a known program and picks up from there. Some fancy I/O chips also need to be reset to get started on the right foot. For instance, if you had a port that output random data on startup, you might simultaneously be giving "forward" and "reverse" commands to an entire steel rolling mill or bring about other unpleasantries. The reset bus is also your panic button to stop a wayward micro system when it is up to no good.

If the address lines don't contain addresses all the time, some control line signal must be available to tell us when the addresses are legal. This may be called a VMA line, short for *valid memory address*. Another control line tells us when an instruction is to begin. This is often called a *sync* line.

Other systems may have special lines for other uses. Some micros can be stopped, either briefly or for a longer time. This is done by using a *halt* or *wait* line. One use of a brief delay is to allow for access to a memory that may be slower than the rest of the system. If a multiplexed bus is used for both data and addresses, another line on the control bus has to tell us what arrives when.

There can also be one or more *interrupt* lines as part of our control bus. These interrupt lines can let an outside world event change what the CPU is doing. There are various types of interrupts, some of which can be turned off and on and others which demand immediate attention. Some micros have many interrupt lines, and others have only one or two that are daisy-chained as needed. More on this in the next chapter.

A definition . . .

The CONTROL BUS is used to give all additional information needed to run a microcomputer.

CONTROL BUS details vary with the microcomputer in use.

Typical CONTROL BUS lines include sync, reset, read/write, halt, interrupts, valid memory address, system status, halt, clock, timing signals, and so on.

The control bus isn't a single-purpose sort of thing like the data bus or the address bus. Instead, the control bus is a group of wires used to control the rest of the system as needed. The object of the game is to have as simple a control bus as possible and to use as few different control signals as possible, but most micros still end up with a handful of lines.

Most of the lines on the control bus go from CPU to address space and other peripherals. A few lines, such as the reset line and interrupt lines, go the other way, bringing outside world commands into the CPU. Most control bus lines are unidirectional and go only one way.

Here's a rundown of typical control bus lines . . .

TYPICAL CONTROL BUS LINES

CLOCK—A master frequency used to lock all timing together.

HALT—A line used to stop the CPU temporarily.

INTERRUPT—One or more lines that let an outside-world event gain control of the micro.

READ/WRITE—A line that tells RAM or I/O whether it is being written to or read from.

RESET—A line that gets the CPU started on power-up or restarted after some system error.

STROBE—A line or lines that tell when data or address signals are valid.

SYNC—A line that tells the beginning of each CPU instruction cycle.

Once again, the object of a control bus is to use as few lines as possible to control the response of the microprocessor chip, the address space, and any interaction with the outside world. While most control lines go from CPU to address space, a few, such as the interrupt lines, may go the other way.

Details of use and names will vary with the microprocessor family you pick. For instance, you'll find a single R/$\overline{W}$ control line in the 6500 and 6800 families, while the 8080, Z80, and 8048 families use two separate read and write lines. Other details will also change from family to family.

ADDRESS SPACE DECODING

Our address space is the total reach of the microprocessor's CPU. Into this address space we put blocks of RAM, ROM, I/O, or nothing. How do we know which block we are going to access at any time?

We must use an address that calls out a unique slot in the address space. The trick is to make each address correspond to something we want.

Back in Volume 1, we saw that we could decode any 16-bit address with a 16-bit AND gate and a handful of inverters. This may be the way to go if you need only a single slot decoded for a special use, but brute force decoding gets unbearably complicated when you need lots of locations uniquely decoded.

We saw that one way to simplify decoding was to split up the problem. We might take a 65536 location address space and break the space up into 256 pages of 256 locations per page. We might take the inner circuitry of a 16K RAM and use seven column addresses and seven row addresses in a matrix, since 2↑7 * 2↑7 = 2↑14 = 16384. Anything that breaks the decoding down into two or three things working together is bound to help.

For instance, suppose we have a microcomputer system that has three blocks of 16K RAM and one block of 16K ROM in the address space. Each block will need fourteen address lines to select one unique location. What we do is connect all fourteen low address lines to *all* address inputs of *all* blocks at once.

That leaves us with two high address lines. We take those two lines and route them to a one-of-four decoder. The output of the decoder then drives the chip-selects or otherwise activates only one block at a time.

Thus, in a micro system, we always address everything everywhere, but we are careful to activate only one block of something at once . . .

Address space decoding is done in at least two steps.

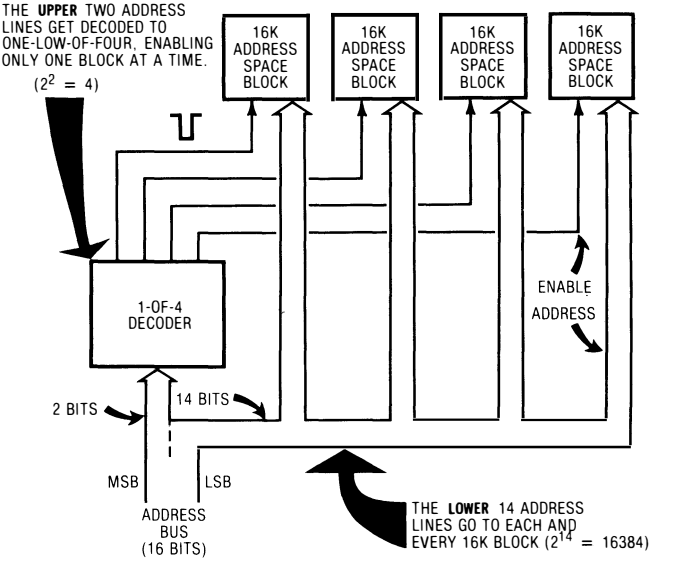
ALL of the low address lines go to ALL address inputs of ALL blocks at ALL times.

The remaining high address lines are decoded to activate or enable only one block at a time.

Usually, you end up doing two different things with your address lines. You put all the low address lines directly into each block of RAM, ROM, I/O, or empty space. The remaining high address lines then go to a fast decoder that activates only one selected block at a time.

How many address lines go where is decided by the size of each block. If you are using 16K blocks, then the lower fourteen address lines go to each and every block, since 2↑14 = 16384. The high two lines get one-of-four decoded and activate one block at a time. Like this . . .

DECODING BLOCKS OF ADDRESSES:



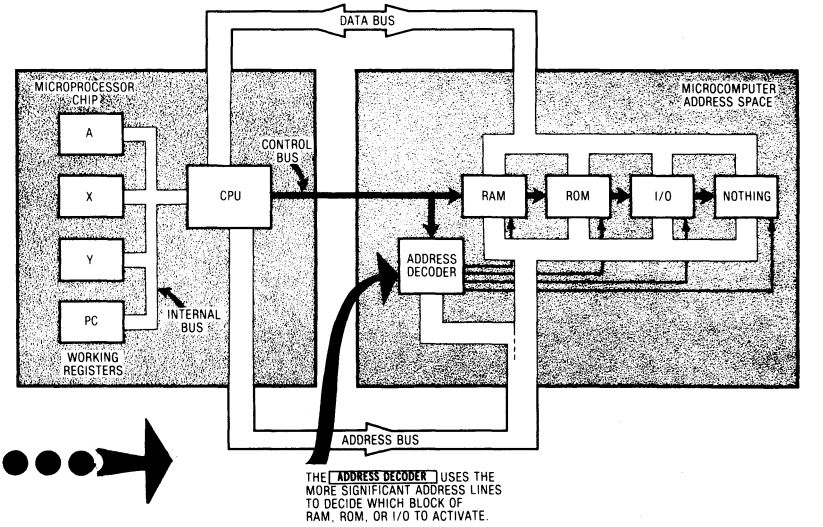
There are several important points about decoding the address space. You use as many decoding steps as you need to reduce the use of brute force hardware. Each and every block of the address space will always get all of the low address bits all of the time. But only one block will be activated at a time from a decoder that works with the high address bits. If a block is going to do lots of smaller things, such as serving a bunch of I/O ports or whatever, extra decoding internal to that block can be added to sort things out further.

Any and all decoders you use have to be much faster than everything else in your microprocessor system, since we can't waste time waiting for a decoder to make up its mind about which output to activate. As a general rule, decoders should be at least twenty times faster than the rest of the system. This means you use LSTTL decoders with their 25-nanosecond speed in the normal NMOS or CMOS microprocessors. If you are building a faster bipolar microprocessor, then your decoders have to be super fast, using ECL or some other horrible logic family to pick up the blinding speed you will need.

From the CPU point of view, all the CPU is doing is sending out an address. From the single address location viewpoint, that address location is activated only whenever the CPU sends out an address. It is only at the system level that we see all this fancy two-step or three-step decoding going on. An address is an address, both at the CPU and inside the address space.

Let's put all this together into a final architectural picture . . .

. . . TO GET THIS ARCHITECTURAL MODEL OF A TYPICAL MICRO:



Reviewing, we see that the typical microcomputer has an internal microprocessor area and an external address space area, and that the two are normally separate. The microprocessor includes the CPU, which acts as postmaster or system traffic cop, along with some working registers that get involved as temporary stashes with just about everything the microcomputer does.

The address space consists of blocks. Each block can be RAM, ROM, I/O, or nothing at all. These blocks are arranged to suit the needs of both the microprocessor and the system designer.

The microprocessor communicates to the address space with three buses, called the data bus, the address bus, and the control bus. The data bus is used to pass information to and from the address space. On a typical 64K micro, an 8-bit bidirectional data bus is most often used.

The address bus is used only to send out addresses from the CPU to memory and has to be wide enough to address each and every location in the address space. On a 64K micro, there is a 16-bit unidirectional data bus, usually arranged as two 8-bit groups to select one of 256 pages and one of 256 positions on each page. A few micros will multiplex their address and data lines over the same lines to save package pins. If this is done, addresses and data have to be separated at each end. System strobe pulses have to keep track of what is going where.

The address lines are normally split into at least two groups. The low address lines go to the address inputs on all the chips in every block. The high address lines go to a special, fast decoder that picks one of the available blocks. The decoder will often use the chip select pins to activate only the ICs in that one enabled block. Sometimes a second decoder will be found inside a block to further break down the decoding process. Multiple-step decoding is done to greatly simplify the hardware needed to recognize a certain address. However the decoding is done, the CPU puts out an address and only one slot at a time recognizes and responds to its own unique address.

The control bus is really a group of lines used to keep track of what is happening inside the microcomputer. Control bus lines include clocks, resets, read and write lines, strobes, sync signals, halt lines, interrupts, and whatever else is needed. Most of the control lines originate within the CPU, but others, such as interrupts and resets, can come from the outside world.

Each system has its own unique architecture. What we have looked at is a general architecture of a general microprocessor. Whatever system you pick, you will always find an address space, a CPU, working registers, one or more buses to get addresses and data back and forth, and a few control lines that let either you or the CPU control the action.

THE MEMORY MAP

There are two very useful tools that will show you what goes where in the memory space and inside the CPU. The address space tool is called a *memory map,* and the microprocessor or CPU tool is called a *programmer's model.* Let's look at the memory map first.

The memory map is simply a picture of what is located where in the address space. The choice of what goes where is made by the system designer but is restricted by the microprocessor being used.

There are two types of memory maps. The *simplified memory map* paints the big picture. In particular, the simplified map should show you how big the address space is, indicate where the user RAM is located, point out where the monitor is, and identify the general area of I/O locations. The purpose of a simplified memory map is to get you started understanding and using a micro system.

There is also a *detailed memory map.* The detailed memory map tells you the exact use of each and every location in the entire machine. Detailed memory maps can turn out to be gory messes. Save these for later on when you understand exactly what you are doing.

Detailed memory maps will also change with whatever happens to be in use. For instance, in the Apple II, the use of HIRES graphics, the DOS operating system, machine language, the mini-assembler, the Sweet-16, the floating point package, Applesoft, and Integer Basic may all want to use certain locations for different things. Obviously, any given location can be used for only one thing at any one time.

Thus . . .

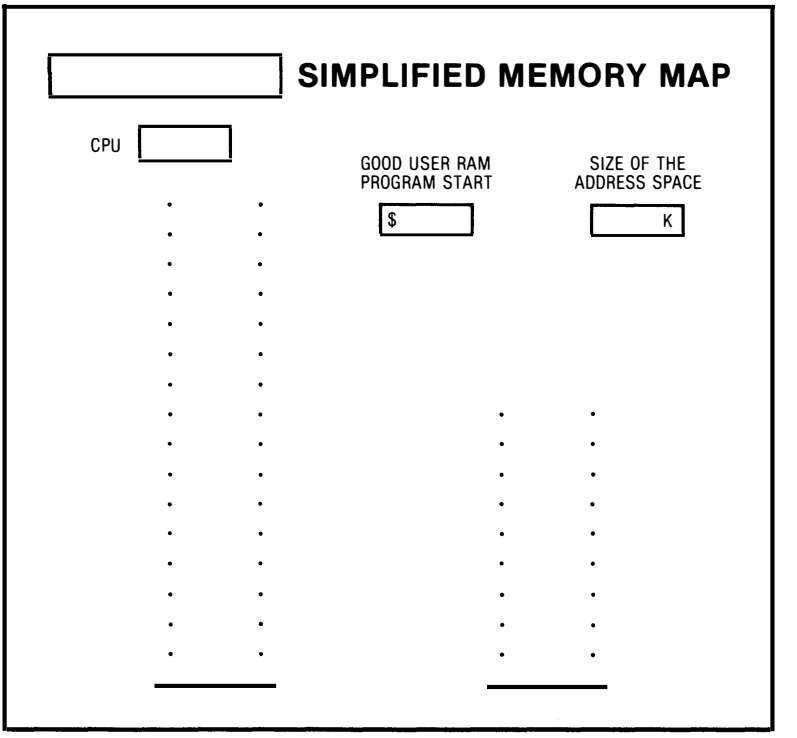
SIMPLIFIED MEMORY MAP—Gives the big picture of address space usage. Shows you user RAM, I/O, monitor and unused locations.

DETAILED MEMORY MAP—Spells out the exact uses of each and every address space location. Gives all the detail an experienced programmer needs for total system control.

For instance, a simplified memory map for the older Apple II would show that there is RAM in the three bottom 16K blocks and that the very bottom 2K of RAM is reserved for "system" uses. What system uses? We don't care for now. All you need to know is that

this area is temporarily off limits to you as a programmer until you know more about what you are doing.

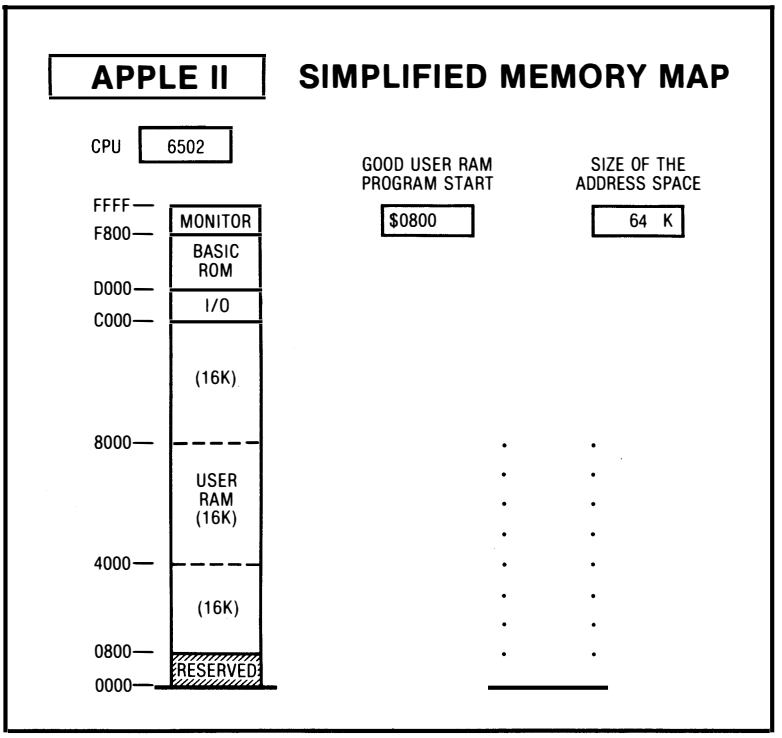
A form for a simplified memory map looks like this . . .



The important things the simplified memory map should show you are the name of the system, the microprocessor you are using, the size of the address space, the intended uses of address space blocks, the location and beginning of "safe" user RAM, and the location of the monitor and I/O areas.

The simplified memory map is often shown like a thermometer, with one "degree" of the thermometer for each page of the address space, and with the low addresses on the bottom and the high addresses on top. If slightly more detail is needed in some area, this area can be expanded. But it is important not to get into too much detail.

Here is the simplified memory map for the original Apple II . . .



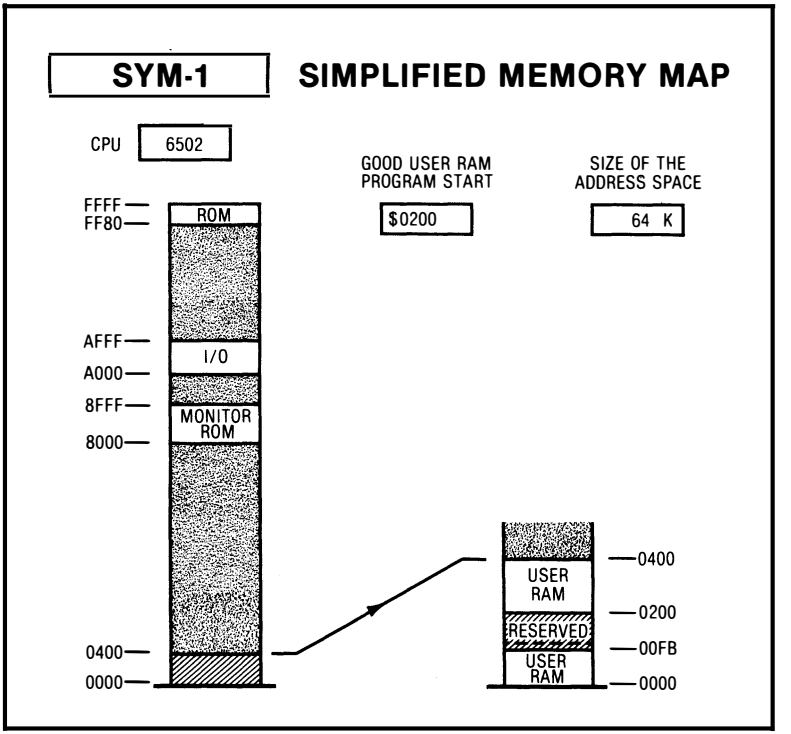
We see that the bottom 16K is RAM. The next 16K was originally intended for expansion RAM, as was the next 16K. Today, most older Apples start with a full 48K of RAM and then stuff up to a megabyte of extra bank switched RAM into the I/O slots. The next 4K of the usual 64K address space is reserved for I/O, and the top 12K of the address space is ROM. The uppermost 2K of ROM is used for the system monitor. The other ROM areas normally contain firmware for a higher level language, such as Applesoft or Integer Basic.

The bottom 8K of RAM is shown as "reserved." Actually, some great things are happening in this "reserved" space, but we don't need to see the details just yet.

Later on, we might find out that the $0400 to $0800 space in this RAM is a video display page. For more detail, we would find out which location corresponds to which place on the screen. We would even find out that there is some scratchpad RAM stuffed in here for I/O, and so on. But it is very important always to start with a simplified memory map. That way you aren't tripping over a bunch of things that you don't yet need or understand.

Simplified memory maps with exactly the right amount of detail are very rare, so you will almost always have to make your own, starting with a blank form.

Let's look at some more examples of simplified memory maps. Here is the SYM-1, a trainer from the 6502 school . . .



The first thing we notice is that much of the address space is unused. This is typical of most trainers. You can learn bunches with nothing but short programs that take up only a few hundred words of RAM. Further, the trainer's monitor and other firmware are also usually simple and take up little space. If you like, you can add your own RAM and ROM to this space later on, but chances are you will use some fancier micro instead. So . . .

Memory maps for most trainers will have lots of empty space.

It takes only a few hundred RAM and ROM locations to do simple yet useful micro tasks.

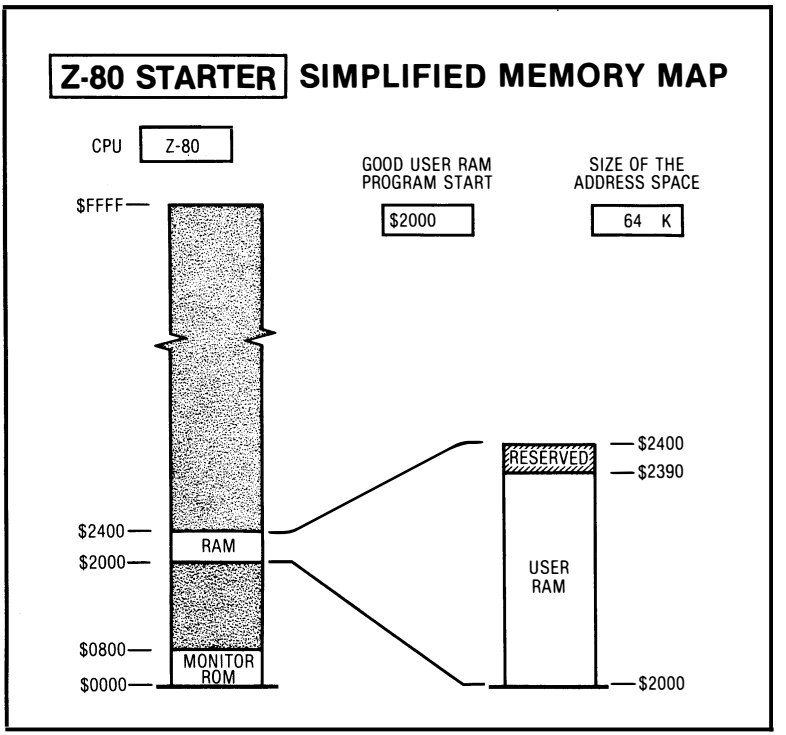
On the SYM-1, we see we have a thousand words of RAM down on pages zero through three. As with most systems in the 6502 family, we put the RAM on the bottom and the ROM on the top.

RAM area $00FB through $01FF is shown reserved. The eight locations on page zero are used by the monitor to save registers for you. Page one is set aside for the stack. Most often, the stack is active only at the upper end of page one, and you may be free to use most of the lower space on page one. But this is risky for someone new to micros since you can either plow the stack or let your stack run down into your program or data. For now, leave page one alone.

We see there is a monitor ROM at $8000 through $8FFF, and an additional ROM area at the very top of memory. This additional ROM area holds our reset and interrupt vectors so we know where to go on a system reset or an outside world interrupt.

We also see that we use a 64K address space and that location $0200 is a good place to start a user program.

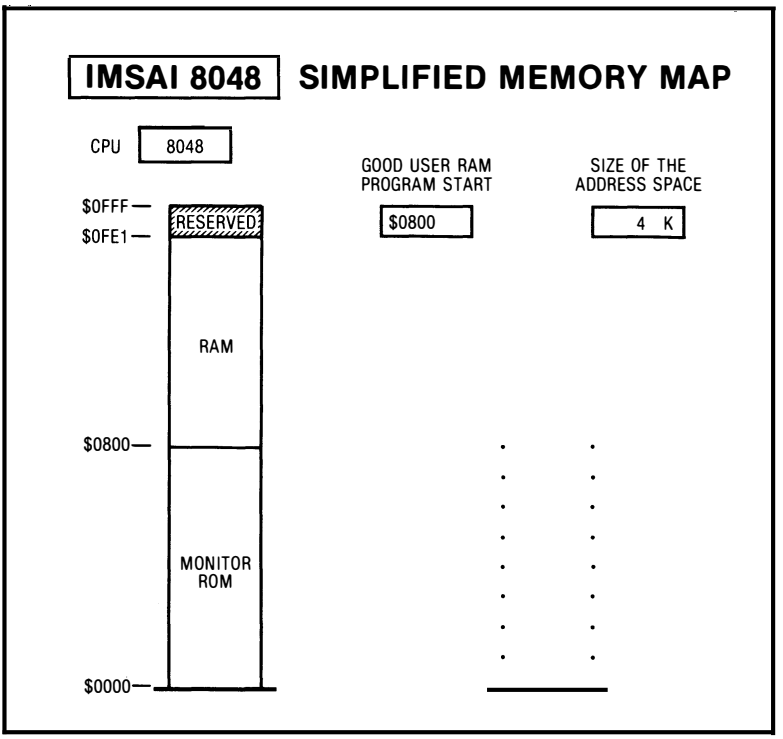
Here's an example of a simplified memory map for the Z-80 starter . . .



Since the Z-80 starter is from the 8080 school, we expect to find ROM on the bottom and RAM further up. In this case, our monitor takes up the first 2048 locations in the address space. This time, the reset and interrupt vectors go at the very bottom.

Our user RAM goes in the middle, providing 1024 words starting at hex $2000. A portion of user RAM above $2390 is set aside for the monitor's private use. Location $2000 is often a good starting place for your programs on this trainer. As with the SYM-1, we have available a full 16-bit, 65536 location address space.

Here is a horribly out-of-date example of a trainer with a much smaller address space, the Imsai 8048 . . .



This trainer is extinct, but it does shows us the typical layout for most current 8048 microcomputers. The 8048 is a smaller microprocessor intended for dedicated control uses. Its normal memory map is only 4K, or one-sixteenth the size of the two we just looked at. This memory space is split into two *banks* of 2K each. The ROM

bank goes on the bottom and the RAM bank goes on the top. Your best starting place for a user program is at location $0800 at the bottom of the RAM space. When you examine the 8048 in detail, you'll find that this limited address space is more than made up for with very powerful ways of handling input and output. The 8048 is good for appliance and automotive uses but normally cannot support a higher level language or any use needing much memory space.

The important thing with a simplified memory map is to keep it simple. Later on, you can pick up specific details as you need them. But adding detail also adds confusion, particularly for beginners.

Be sure to create your own simplified memory map for each and every micro you run across. All the info you will need should be buried in the micro's documentation somewhere. Use the same shape and form for each of your simplified memory maps.

A detailed memory map is often shown as a table or a book of tables rather than as a simple thermometer style map. For instance, here is a fragment of the Apple II's detailed memory map . . .

$0020	WNDLEFT	Left side of scrolling window (normal range $00 to $27)
$0021	WNDWDTH	Width of scrolling window (normal range $00 to $28) (WNDLEFT + WNDWDTH <$28)
$0022	WNDTOP	Top of scrolling window (normal range $00 to $18)
$0023	WNDBTM	Bottom of scrolling window (normal range WNDTOP to $18)
$0024	CH	Horizontal cursor position (normal range $00 to $27)
$0025	CV	Vertical cursor position (normal range $00 to $17)

As you can see, these six locations tell a lot about what you need to know to use the scrolling window in the Apple II. Since there are another 65530 locations left to cover, a detailed memory map can be a very involved listing. Things get even worse when one location can have more than one use.

The detailed memory map is often far too detailed for you early in the game. You tend to get lost out in left field rather than zeroing in on what you are trying to do. Here are two good use rules . . .

DO— Have a specific goal if you must use a detailed memory map.

DO NOT— Use a detailed memory map unless you really need it.

Wait to use the detailed memory map on a system until you really need it. Detailed memory maps also are often hard to find. You may have to try a user group or some independent publisher to pin down this information.

Let's review. The memory map is a tool that tells you what goes where in a micro's address space. There is a simplified memory map that shows you only the essentials and a detailed memory map that gives you all the specifics on each and every location. The important things the simplified memory map should show are the CPU used, the size of the address space, the locations of the ROM, RAM, and I/O blocks, the area of user RAM available without restrictions, and a good starting place for simple programs. The detailed memory map should show the use of each and every location for all possible uses. But don't use such a map until you have a specific need and understand what you are doing.

DOING IT:

( ) **Create simplified memory maps for a trainer, a personal computer, and an industrial controller of your choice.**

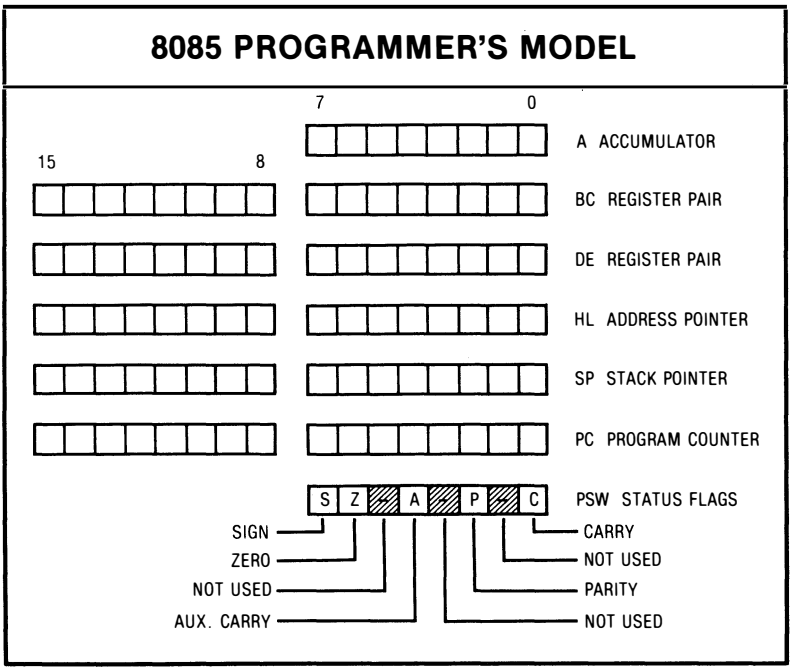
As you tear into the manuals that go with most micros, you'll be horrified at how atrocious most of them are. A simplified memory map properly done in a standard form is a joy to behold. Make sure that any simplified memory maps you do show enough, and just enough, information to be useful. A simplified memory map will be a very important part of the micro toolkit that we will be putting together shortly.

THE PROGRAMMER'S MODEL

Our memory map tells us the uses we put on locations in the address space. The programmer's model is another tool that does the same thing for the microprocessor's *internal* space . . .

PROGRAMMER'S MODEL—A reference that shows the working registers and other resources available inside a microprocessor.

Let's look at a programmer's model and see what it will do for us. Here is the programmer's model for the 8085, an upgraded version of the original 8080 . . .



Each block corresponds to a single bit location. Each row of blocks corresponds to one working register. We see at the top that we have an accumulator that is eight bits wide. The accumulator usually ends up holding the result of a logic or arithmetic instruction.

We then see that we have six register pairs labeled B, C, D, E, H, and L. The B, C, D, and E are general-use registers that can be used as handy stashes. You can use these as separate 8-bit registers, or you can pair B and C or D and E into a single 16-bit wide register.

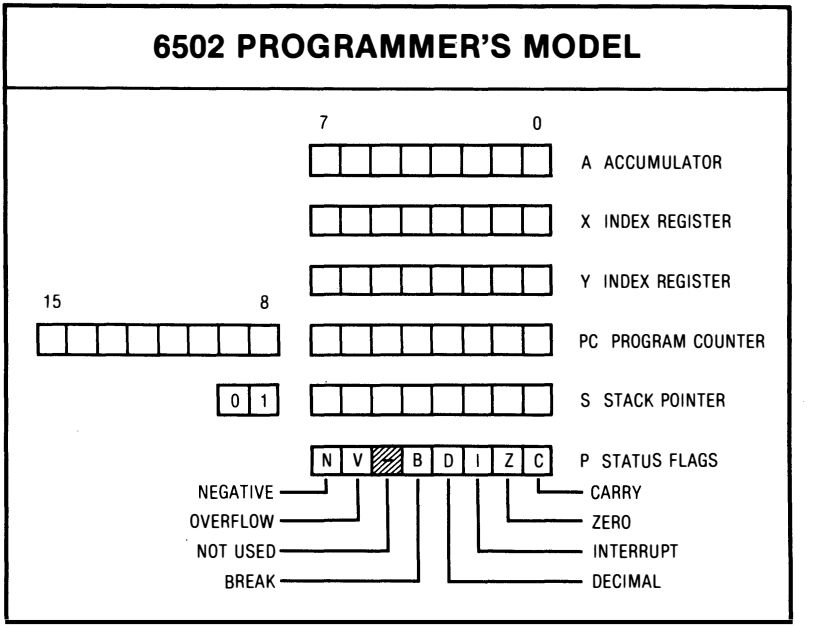
H and L are an intended-use register pair. They will often contain an address, rather than data, and will point to the address space. The H and L pointer will show the source or destination of a move command that is to go out into the address space. While you can use H and L as plain old registers, it is better not to.

Continuing down the model, we see there is a 16-bit wide stack pointer that can point anywhere in the address space. There is also a 16-bit wide program counter that can point to any location in the 64K address space. We will find out much more on stack pointers and program counters later. The stack pointer will usually point to the next available location on the stack, which is a convenient stash for temporary storage and for handling subroutines and interrupts, and the the program counter will tell the starting address of the present instruction we are working on.

Finally, we have a bunch of flags grouped together in a flag register. The carry, negative, and zero flags are common to most micros. The 8080 school calls its negative flag an S flag, short for *sign*. The same thing in the 6502 school is called an N flag. The name is different but the purpose is the same.

We use flags to show that something has happened in the micro. Flags become very powerful when you test them and then make a decision based on the test result. We'll learn more on flags later. The point here is that the programmer's model shows you how many of what kind of flags you have and how they are arranged.

The programmer's model for the 6502 looks like this . . .

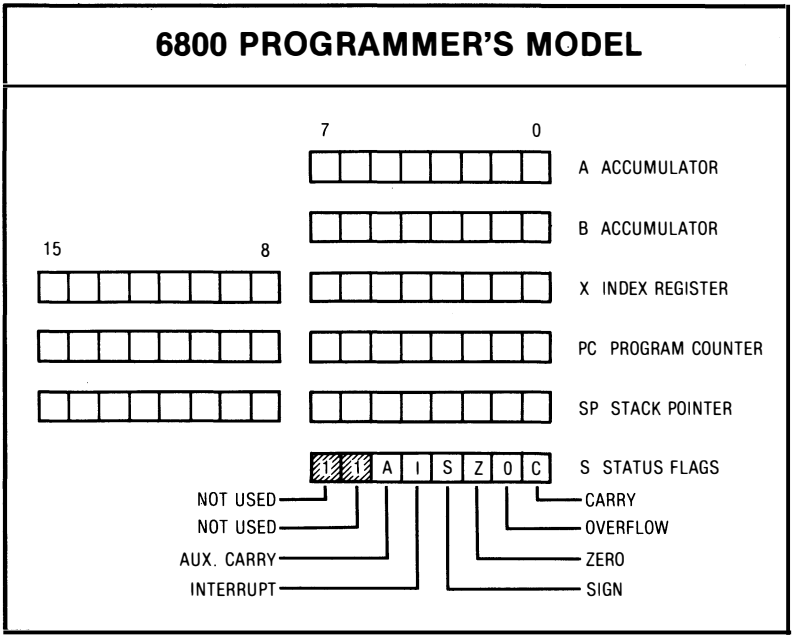


We see at once that the 6502 school has fewer and shorter working registers than the 8085. There is an 8-bit general-use accumulator and a pair of intended-use 8-bit X and Y registers. There is a 16-bit wide program counter that can point anywhere in the address

space and an 8-bit wide stack pointer that always points to a location on page one. The flag register holds seven flags for us, including the negative, carry, and zero flags common to most micros.

The programmer's model also hints that 16-bit wide arithmetic and memory moves will be harder than with a device from the 8080 school. But what the model doesn't show is that the 6502 has all 256 locations of page zero in the address space available to us for use as working registers and that there are powerful address modes that make running around the entire address space very easy and convenient.

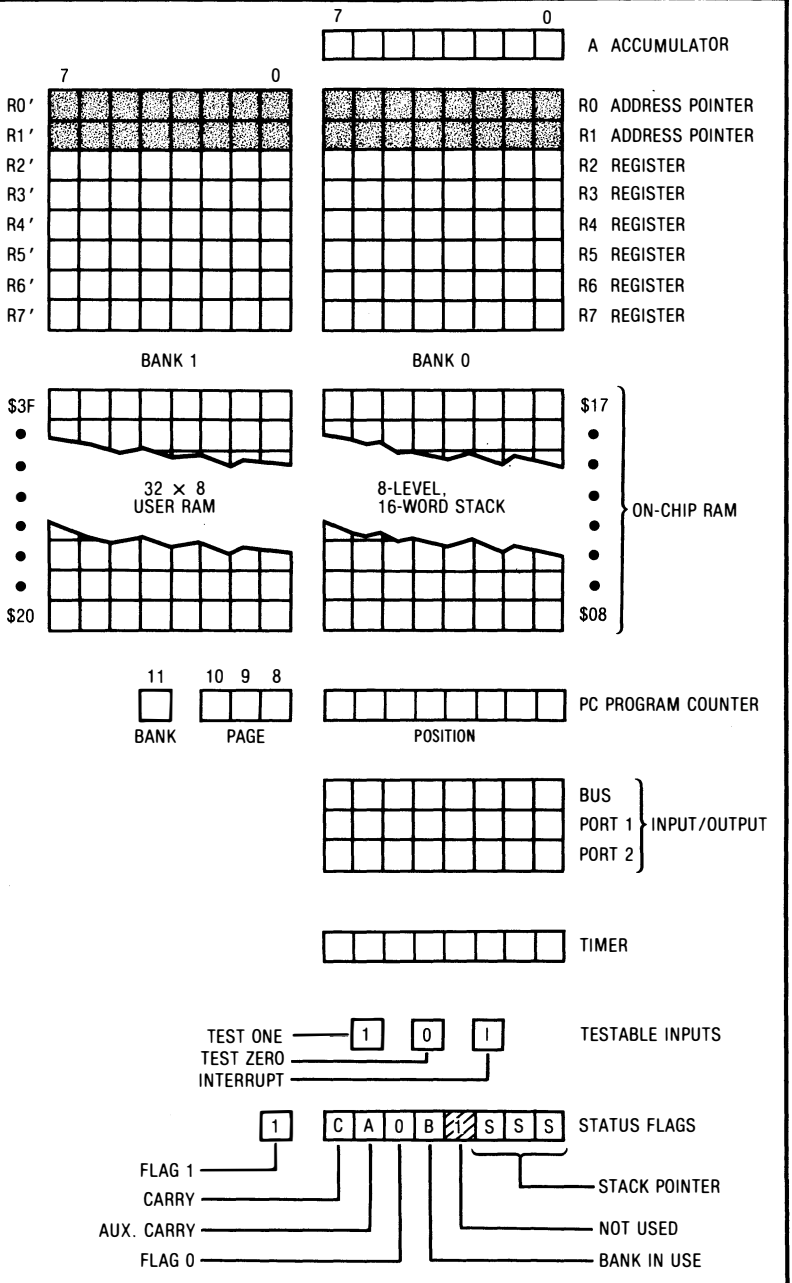
The 6800's programmer's model looks something like the 6502's . . .



This time we have two 8-bit accumulators and three 16-bit wide pointer registers—namely, the index register, the program counter, and the stack pointer. There are six flags available, including the usual carry, zero, and negative flags. The 6800 turns out to have much weaker address modes than does the 6502 and is inherently slower because it lacks a feature called *pipelining*. It is interesting to compare this VCIW with the 6502 to see the "alike but different somehow" similarities between micro families.

Remember how our 8048 microprocessor had a much smaller address space than the others? Well, it more than makes up for this with its programmer's model . . .

8048 PROGRAMMER'S MODEL



We see there is the usual 8-bit accumulator. In addition, there are *sixteen* working registers, numbered R0 through R7 and R0′ through R7′. R0, R1, R0′, and R1′ are intended-use registers since they can contain an address for us. The others are general-use registers.

If sixteen registers aren't enough, there are also another thirty-two words of user RAM inside the CPU, along with a sixteen-word stack.

Our program counter is shorter than usual since it only has to cover 4K of address space. The program counter is further broken down into three parts. The most significant bit selects the 2K *bank* in use, picking between the low 2K ROM and the high 2K RAM. The next three bits pick one of eight *pages* of 256 bytes out of the bank selected. The final low 8-bit word gives us the exact *location* on the page and bank we selected.

We also see bunches of new and fancy stuff further down in the model. There is a *timer* register that can be used to provide time delays or count events. There are three input lines that can be tested. There are two I/O ports of eight bits each on board, along with an expansion port called the BUS I/O. Only five flags are available, and the stack pointer, which is allowed to point to only eight different locations, is included in the flag register.

Don't worry about how all this stuff works just yet. Our interest here is in the programmer's model itself and how it shows the available resources inside the CPU. We've gone into extra detail so you can see the variations you will meet from micro to micro. Most micros can do almost any task, one way or another, but the details can and will vary.

DOING IT:

( ) **Create a programmer's model for three different new microprocessors.**

We have used programmer's models of the older devices as examples here, since most of the newer chips add to or enhance these "oldies but goodies."

The newer micros may have very involved programmer's models because of all the resources they include. Often you can find the programmer's model on the pocket card for the microprocessor. If not, it's bound to be somewhere nearby, such as in a programming manual or on a detailed data sheet.

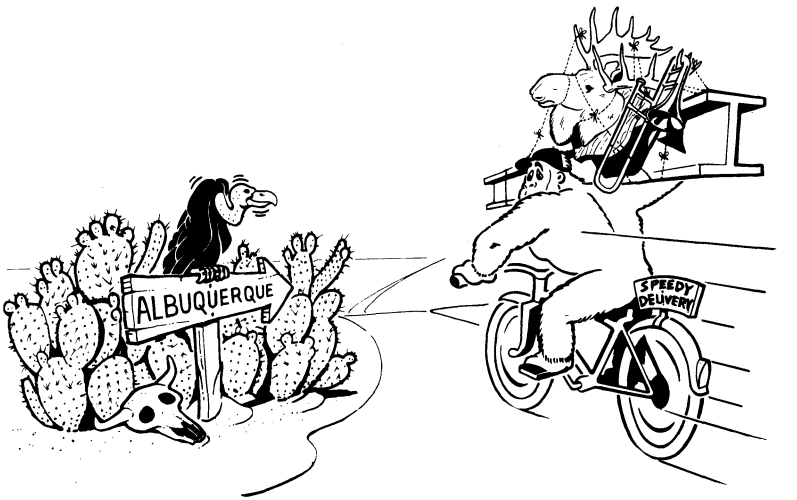
Even if you have a nice, neat programmer's model available, be sure to make yourself a personal copy BY HAND. This forces you to think about what is available and how it is used as you make the copy. Hand copying a model also raises the "what if?" question. If you don't see enough details on how to use some resource, think about it for a while, ask "what if," and then try it out on your own to see what it will do.

As a general rule, anything you ignore on a micro will turn out to be super powerful and super useful, once you nail down the use details. The longer you put off learning about it, the more it will do for you, and the sillier you will feel when you ultimately understand what is going on.

Wow. We've gone through a lot of details on both the memory map and the programmer's models. Both of these will be essential parts of your micro toolkit. But before we look at all the other tools you will need, we have to check into another matter that involves *address modes*. I like to call these . . .

THE PACKAGE TO ALBUQUERQUE

How would you go about getting a package to Albuquerque? . . .



There are lots of ways to get a package to Albuquerque. Speed, cost, reliability, protection, convenience, weight—all these things enter into your choice of how to ship a package. For instance . . .

DOING IT:

How would you get each of the following to Albuquerque?

( ) A jar of peanut butter
( ) 50 pounds of peanut butter for a small bakery
( ) 50 tons of peanut butter for a candy factory
( ) A gasket for routine farm machinery maintenance
( ) A similar gasket for a hospital's only kidney dialysis machine that failed suddenly
( ) A tin of corned beef hash
( ) A kilo of Lebanese hash
( ) A live giraffe
( ) 1000 marbles
( ) 1000 Krugerrands
( ) 1000 cubic feet of helium
( ) An idea

What is good for one package is clearly ungood for another. Where you are shipping something can also lead to big differences . . .

DOING IT:

How would you ship a barge of coal from . . .

( ) Pittsburgh to Cincinnati?
( ) Amarillo to Albuquerque?

From Pittsburgh, we simply cut the rope and float the whole works down the Ohio River. Simple, convenient, and cheap.

From Amarillo—let's see now. First we borrow every skateboard in Texas and neatly lay them all out on I-40. Then we find some . . . But what about that long hill down Sandia Pass? As long as they aren't too particular about exactly *where* in Albuquerque we deliver the coal, it just might work. But clearly there are better ways.

Microcomputers also offer lots of ways to get a package to Albuquerque. There are many different methods you can use to get to a

certain location in the address space or to access a certain working register. We call these methods *address modes* . . .

> **ADDRESS MODE**—Any method a microprocessor uses to access a working register or reach a certain location in the address space.

Much of the richness and variety of the different micro families comes from the different address modes available. Some micros have only a few address modes; others have over a dozen. Some modes are everyday plain vanilla things; others are extra powerful.

Which address mode do you use? The answer is the same as how you get your package to Albuquerque.

It depends.

The main differences between available address modes are in their length, their speed, and what they can reach . . .

> **Address modes differ in how many bytes they need, how fast they work, what they can get at, and how convenient they are to use.**
>
> **Not all address modes are available on all micros. But almost any task can be done on a micro using some combination of the available address modes.**

There are lots of tradeoffs here. Sometimes, you can pick any one of a handful of different address modes. Other times, only one will do, or else one will do the job far better than any other. In general, you try to make your program as *short* as possible and try to get it to run as *fast* as possible. These two goals are usually opposed to each other, for anything you do to shorten your code may lengthen how long it takes to do the job . . .

> **Usually, you want to make your programs as short and as fast as possible.**
>
> **Operating speed and program length tend to fight each other, so different address modes are available for horse trading.**

One classic example is a loop. Say something needs to be done ten times. By looping the same code ten times over, you can shorten the program almost to a tenth of its "brute force" form. But each trip through the loop means unavoidable loop *overhead* that takes time. As we said, short code often equals long execution times and vice versa.

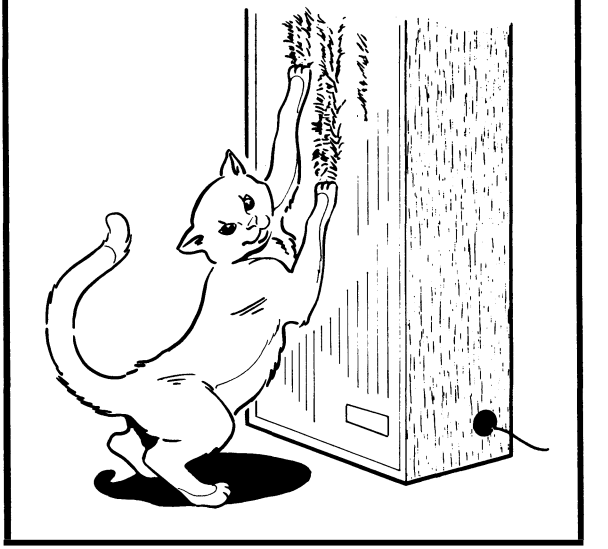
Each manufacturer has its own name for its address modes, and there's lots of PR fluff to make the modes sound better than they really are. Sometimes one mode may be split up several ways to make the machine sound more powerful.

When you remove the flack, most microprocessors use combinations or variants of only seven basic address modes. It is up to you to sort out what these modes are, how they are used, and what's really behind the name used for each mode.

To repeat, there are different address modes because each mode does one particular job better than the others; and there are lots of tradeoffs involved in writing a program that is both short and fast.

Let's pretend we have a general and universal microprocessor that doesn't have strange names for its address modes. Let's also try to relate each mode to something in the real world and see where we get to. Then we'll sum everything up in a handy address mode reference chart.

IMPLIED ADDRESSING



You just got home, and there, sharpening its claws on your best hi-fi speaker grille, was the cat.

Now, exactly *what* you say probably isn't printable, but it will be short and to the point, leaving no doubt whatsoever *which* cat and *which* speaker grille you are referring to.

The simplest microcomputer addressing mode does the same thing. It is short, obvious, and leaves no doubt what is to be done where. This is called *implied* addressing . . .

IMPLIED ADDRESSING—An address mode where it is completely obvious what will happen where, without any further information needed.

An implied instruction usually needs only one byte of op code.

For instance, we now know we have carry flags in most micros. A command to "clear the carry flag," often shortened to the mnemonic CLC, will clear the flag for us.

On a CLC command, the carry flag gets cleared. There is no question about which flag or what we are going to do to the flag.

Register moves and transfers are other examples of implied addressing. The command MOVBC moves a copy of working register B into working register C and destroys anything old left in C. The command TXY transfers a copy of register X into Y. These actions are pretty much the same. Only the name changes from micro family to family.

The big advantage of implied addressing is that it is short and sweet. One single byte is needed for a command and that is all there is to it. Any task that can be done without any further information lends itself to implied addressing.

The big disadvantage of implied addressing is that it is usually limited to manipulation of working registers or housekeeping actions. There is no way to nail down a specific location in the address space with implied addressing. Nor is there a way to answer "with what," "from where," "to where," or "how much."

Address modes will differ through their op codes. A different command will be used for each and every different address mode that a micro can handle, even if the commands may end up doing nearly the same thing.

The symbol for an implied addressing mode is just the mnemonic, and nothing more . . .

An implied addressing mode is shown by the mnemonic only. Nothing else is needed . . .

CLC

Let's look at some more examples of implied addressing . . .

TYPICAL IMPLIED COMMANDS

BRK—Break, or do a software interrupt for debugging.
CLA—Clear the accumulator, or reset it to zero.
CLC—Clear the carry flag.
NOP—No operation. Go on to the next step.
MOVBC—Move a copy of the contents of B to C.
PHA—Push a copy of the accumulator onto the stack.
PLA—Pull off what is on the top of the stack and put it into the accumulator.
ROLA—Take what is in A and rotate the bits all one to the left.
RORA—Take what is in A and rotate the bits all one to the right.
RTI—Return from an interrupt to wherever you left off before.
RTS—Return from a subroutine to wherever you were before.
SEC—Set the carry flag.
TAY—Transfer a copy of the contents of A to Y.

This assortment of op codes is from several micro families, but you get the idea. Any action that can be done without any further help uses the implied addressing mode.

Implied addressing commands are usually the fastest available for a given micro. Since no further information is needed, the micro can right now go and do whatever has to be done. Thus, implied addressing is usually both the fastest and the shortest way to get a job done.

However, there are a very few implied addressing commands that may take a long time to execute, perhaps even longer than any other command in any other mode. In these special cases, a single command starts the micro off on a long song and dance routine. Returning from an interrupt is one example. The micro has to look

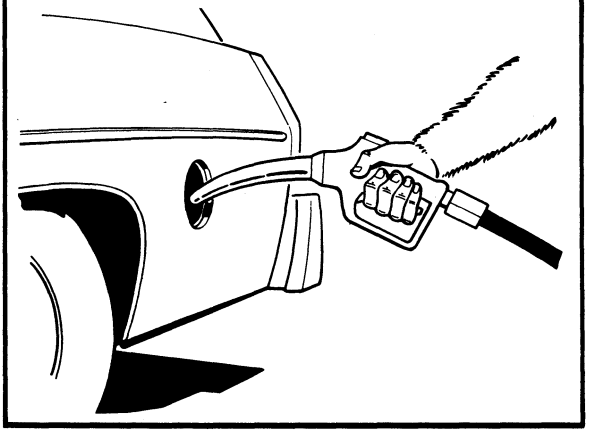
around in the stack to find out where to return to. It then has to get some flags off the stack and finally has to pick up where it left off. All these details take time, even though one simple command is all it takes to start the routine.

Returning from a subroutine and generating a software-controlled break are other examples of implied commands that take a long time. But remember that these are exceptions. Most implied commands are very fast and very short.

Generally, you'll find bunches of implied op codes in any micro family. These get used any time and any place they will work. Practically all of your service and housekeeping commands will use the implied addressing mode.

Suppose we want to put some value into a register. Obviously we can't do this with an implied address mode, because we also have to know "how much?" This leads us to another addressing mode. This one is called . . .

IMMEDIATE ADRESSING



When you pull up to a gas pump, you imply that you want some gasoline. But that is not enough. Besides implying that you want some gasoline, you also have to tell how much of what kind of gas you want. Usually you will do this with some sort of quantity value, such as "ten dollars worth," "ten gallons, please," or "whatever the tank will hold." Even this last value ends up as something definite when the tank is filled.

To buy gasoline, you need two things. First you say you want gas and then you say how much.

The immediate addressing mode does the same thing for the microcomputer. The immediate mode usually puts some value somewhere for you. After an op-code word that says to do something, there is a second word that tells "how much." The first word says "jump," while the second says "how high" . . .

IMMEDIATE ADDRESSING—An address mode that puts a value into a location.

An immediate address instruction often needs two bytes. The first byte tells us a location and the second tells us the value to put in that location.

The name *immediate* sounds rather urgent, but here it simply means that we know exactly what we want to fill something with. We put a value immediately into a register, rather than going to some other location, getting some variable value from that location, and moving it.

The usual symbol for an immediate command is the mnemonic followed by a # number symbol and then a numeric value.

For instance, the command LDA #$56 tells us that we are to immediately put the hexadecimal value $56 into the accumulator. Note several important things here. First, the immediate address mode takes both an instruction and a value. Second, the # symbol is absolutely essential, so that you or an assembly program can tell an immediate address mode from other upcoming modes. Finally, you must show whether the value is in decimal or hex. Hex is normally shown by a dollar sign in front and decimal is shown with nothing in front . . .

An immediate addressing mode is shown by a mnemonic, followed by a number symbol, followed by a value . . .

LDA #$56

Here are some assorted examples of immediate addressing commands . . .

LDA #$56—Puts a value of hex 56 in accumulator.

LDA #56—Puts a value of decimal 56 in accumulator

EOR #$FF—Does a bit-by-bit exclusive-OR of the accumulator and the mask of value #FF (in this example, we complement A.)

CPY #$04—Compare the Y index register against the value hex 04.

SBC #$01—Subtracts one from accumulator.

We use the immediate addressing mode whenever we want to fill a register with some value, or whenever we want to do a logic operation against a fixed mask or a fixed value.

By the way, we are showing you these examples in what is called *assembler format*. Assembler format is easy to read and will be most helpful later. When you actually load these values into a machine language program, though, you will simply be punching hex numbers into the machine. In the immediate addressing mode, the first number will be a command to do something and the second number will be a value or a logic mask.

Two fine points. The LDA #56 decimal example will work only if you or an assembly program calculates the "real" value of hex $38. And the SBC #$01 example will work properly only on a set carry flag in most micro families.

An immediate addressing mode is usually slower than an implied addressing mode. The reason for this reduced speed is that there are usually two bytes in the instruction. After the micro knows what it is to do, it has to look into the second byte to find out "how much?" or "with what?" Time is needed to process and then use this second piece of information.

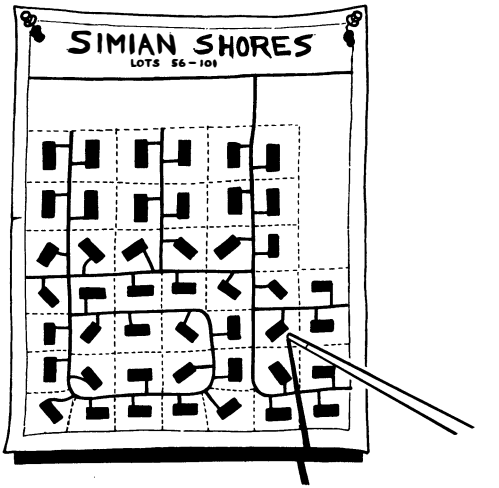
The advantage of immediate mode addressing is that it lets you shove a fixed value into or logically operate a fixed mask against a register or other location.

The disadvantage of immediate mode addressing is that it is *only* immediate. There is no ordinary way it can handle variables or changing values. Although great for putting numbers into registers, immediate mode usually can't get the numbers back out, since a store command has to tell where to put something. Immediate mode can ask "how much" or "with what "—it cannot ask "where."

Another disadvantage shared by the immediate mode addressing with the implied mode is that it usually works only with working registers and cannot reach a point out in the address space. If we

want to get anywhere in the address space, there is one address mode that always works for us. This one is called . . .

ABSOLUTE LONG ADDRESSING



A city zoning map shows you the location of every street and every address in the entire town. In much the same way, an absolute long address lets us reach each and every location in a micro's address space . . .

ABSOLUTE LONG ADDRESSING—An address mode that can reach any and all locations in the entire address space.

An absolute long address instruction often takes three bytes. The first byte tells us an action and the second and third bytes tell us the exact location in the address space where that action is to take place.

Absolute long addressing always works. It can hit any spot anywhere in the entire address space. Absolute long addressing is available on all micros and is one main way we have of reaching into a

RAM, ROM, or I/O location out in the address space, either loading from or storing to that address.

The usual symbol for an immediate address is the mnemonic followed by enough hex digits to specify an exact address. Most often, you will use four hex digits to nail down one location in an address space of 65536 locations . . .

An absolute long addressing mode is shown by a mnemonic, followed by an address value of four hex digits . . .

LDA $FA62

For instance, the command LDA $FA62 will reach into the address space, get the value stored in location hex $FA62, and put that value into the accumulator or A register. Usually three bytes of op code are needed. The first byte tells us that an absolute load of the accumulator is needed. The second and third bytes tell the exact address from which loading is to take place.

Now, on most microcomputer families, including the 6502 and 8080 schools, the *third* byte tells us the *page* of memory and the *second* byte tells the *position* on that page. Thus, when reading actual machine language coding, addresses are picked up in "backwards" order . . .

Most micro families use the third byte of the command to show the page or high address and the second byte of the command to show the position on that page or the low address.

This is true of the 6502 and 8080 schools but is NOT true of the 6800, which is backward from, and thus slower than, everybody else.

WATCH THIS DETAIL

The reason we do things backwards is that there is a speed advantage called *pipelining* that lets the micro set up what it must do ahead of actually doing it. Another advantage to the backwards address entry is that op codes are much more consistent from address mode to address mode. This way, a position on a page always follows the op code, both for absolute long and absolute short addressing.

Here are some absolute long addressing examples . . .

LDA $FEFD—Loads the accumulator with a copy of what is in location $FEFD.
STA $CAFE—Puts a copy of what is in the accumulator into location $CAFE.
INC $0145—Adds one to the value stored in location $0145 if there is RAM there.
DEC $EF03—Subtracts one from the value stored in location $EF03 if there is RAM there.
ASL $1234—Shifts the contents of location $1234 one to the left if there is RAM there.
ORA $6666—Does a bit-by-bit logical OR of $6666 and the accumulator; puts the result in the accumulator.
ADC $4545—Adds the contents of $4545 to the accumulator and then puts the result in the accumulator.
JNZ $8888—Jumps to location $8888 if the last thing done didn't give a zero result.
JSR $9999—Temporarily jumps to a subroutine at $9999. Will most likely return later.

We will pick up details on how to use many of these instructions later. Right now, we just want to recognize that the absolute long addressing mode is the most obvious and most certain way of reaching any location in the entire address space.

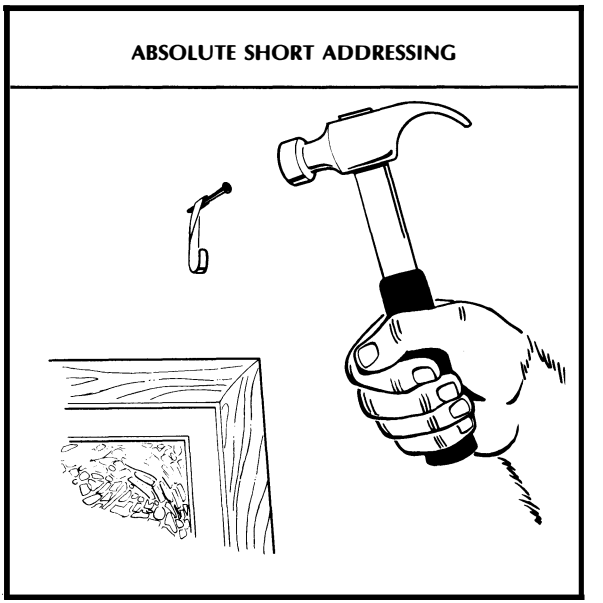
Absolute long addressing is both long and slow. It usually takes three op-code bytes for a single absolute long instruction. The CPU also has to do a lot of work. After it gets its op code telling it to do an absolute long action, it has to go to the next location and pick up part of the address. After this, it has to go to a third location and pick up the rest of the address. Finally, after all three bytes are read, the absolute action can be completed.

The advantage of absolute long addressing is that it will always work and can reach any location in the address space.

There are lots of disadvantages to absolute long addressing. This mode is slow in executing and long in listing. It becomes involved and tedious if you are going to do things to a bunch of adjacent or nearby address locations. For this sort of thing, there are usually more powerful modes available.

Some of the newest 16-bit micros will have an absolute super-long addressing mode. In this, you specify the usual 16-bit address as well as picking one of 128 sectors needed to access the entire sixteen megaword address space.

One way to shorten and speed up an absolute addressing command is to make some assumption about where you are going to end up in the address space. This leads us to . . .



If someone asks you to hang a picture on the east wall of the living room, you usually assume that the picture is to hang in the house you happen to be in at the time. You don't have to go back to the zoning map to find out which house on which street you are talking about. The nail hits an absolute location, but it is limited to one known house.

Similarly, if we make some assumption about where in the address space we are, we can shorten our absolute addressing mode to one I call *absolute short* . . .

ABSOLUTE SHORT ADDRESSING—An address mode that reaches an exact space in a known or assumed small part of the total address space.

An absolute short address often needs two bytes. The first byte tells us the action needed and the second byte tells us the exact location in the assumed part of the address space.

Compared with absolute long addressing, absolute short addressing runs faster and takes up less room in the program, but it is limited to one particular area in the address space.

The details on absolute short addressing vary from family to family. On the 6502, there is a *page zero* addressing mode. Anything you do in this mode will go to or come from a location on page zero, or locations $0000 through $00FF. This makes page zero prime real estate since you can both speed things up and shorten programs by staying on this page as much as you can. This also strongly suggests putting RAM in the bottom of your address space, since these locations are easy to get at.

The 6800 has an addressing mode called, in a triumph of PR doublespeak, "direct" addressing. Direct addressing is the same as page zero addressing. The address specifies one location on page zero.

The 8048 goes about its absolute short addressing in a different way. The normal address space of an 8048 is 4096 locations, usually split into a lower 2K ROM bank and an upper 2K RAM bank. Each of these two banks in turn is split up into eight pages of 256 bytes each. Instructions are available that assume you are either working on the *same* page or else are jumping to a *known* other page. What they have really done is given you eight different jump commands, one for each page. This takes eight different op codes but lets you quickly reach any area you like.

There are other variations of absolute short addressing modes on other micros. The object is to avoid specifying the entire address space, limiting yourself to 256 locations in a known area.

Notation for an absolute short address is usually the op code, followed by the location on the assumed page . . .

An absolute short addressing mode is shown by a mnemonic, followed by a value of two hex digits . . .

LDA $62

Note the important difference between immediate addressing and absolute short addressing. LDA #$34 puts the hexadecimal *value* $34 in the accumulator. In a micro with page zero absolute short addressing, LDA $34 goes into *location* $0034 and gets what-

ever happens to be in that location and puts that into the accumulator.

One more time:

The # symbol after the mnemonic is crucial to your telling the immediate mode from the absolute short addressing mode. LDA #$45 puts the hex value $45 into the accumulator. On a micro with page zero absolute short addressing, LDA $45 reaches down into location $0045, picks up whatever value happens to be in location $0045, and puts it in the accumulator. One mode gives you a fixed *value*. The other goes to a fixed *location* and picks up whatever value happens to be there.

The micro tells the difference between address modes with different op codes. You tell the difference with different symbols. Later on, when you get into assemblers and assembly language, these symbols will be automatically converted to the right mode for you. For now, you have to learn the right notation and usage for each mode and the symbols involved. Getting the notation right is extremely important.

Beginning students mix up the immediate and the absolute short modes more often than any others. Remember that immediate addressing puts a value into a register or operates a mask against it. Absolute short addressing goes into some location in the memory space, gets some unknown value, and either puts it in or removes it from the addressed location.

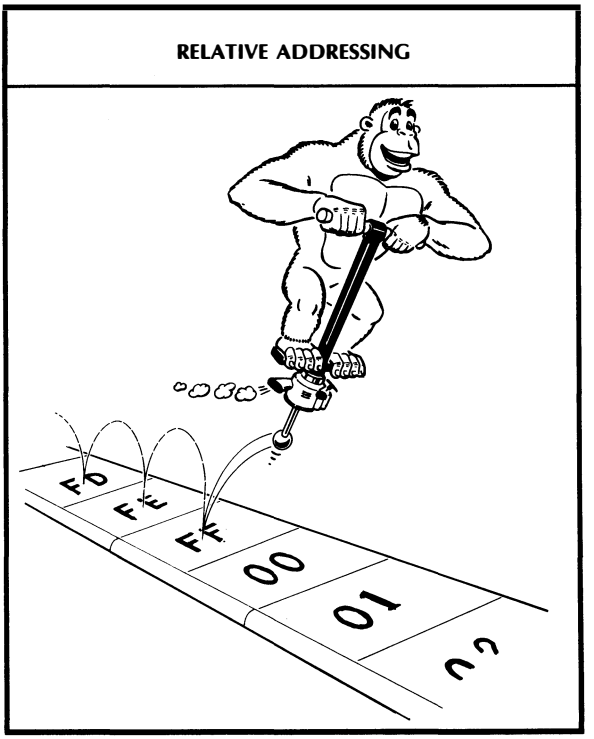
For now, we will skip examples of absolute short addressing, since these vary from family to family. Most often, the first op-code byte will tell us that an absolute short action is needed, and the second byte will tell us the exact location in the assumed area of the address space where that action is to take place. Many of the more important absolute long instructions will also be available as absolute short ones in a typical micro.

The advantage of absolute short addressing is that it is faster and shorter to use than absolute long addressing. The disadvantage is that you are limited to a certain known area in the address space. Fewer commands are usually available in this mode than in absolute long. Another limitation shared with any absolute addressing mode is that things get long and tedious when you try to work with a nearby or sequential group of address space locations.

Yet another limitation of absolute short addressing is that you get into turf fights when different applications all need access to a few locations. Now 256 locations sounds like a bunch, but if you get into a personal computer where a monitor, two languages, a DOS operating system, some graphics, and all sorts of applications software all demand absolute short locations, things get hectic fast. It is

important to define and reserve your absolute short locations carefully whenever a conflict is likely.

Oh oh. Look out. Here he comes . . .



Our friend here is having so much fun he doesn't even know where he is. But, with his diesel pogo stick, he can easily go as many squares forward or backward as he wants. He needn't even read the labels on the sidewalk. Just say "bounce back three" and he will do it for you.

And, regardless of where he starts, the commands "bounce back three" or "go forward seven" will get him somewhere else. You may have done the same thing when lost in a big city. Asking directions, you get an answer of "right for three blocks and then left for five." You still don't know where you are, but you now can reach your goal by following these *relative* instructions.

Micros often like to get someplace else in the address space by going so many steps backward or forward from where you happen to be. This is called *relative addressing* . . .

RELATIVE ADDRESSING—An address mode where you will go so many spaces forward or backward from where you are now.

A relative instruction usually needs two op-code bytes. The first tells you the action and the second tells you how far to go in which direction from where you are.

Thus, relative addressing has nothing to do with where your Uncle Louie lives. Relative addressing in a micro means simply to go forward or backward so many address space locations relative to where you happen to be.

Relative addressing modes always involve a command that is trying to get you somewhere else. If you are always to go somewhere else, this is called an *unconditional jump,* or an *unconditional branch,* depending on the micro family.

More often, a relative addressing mode will be involved with some sort of a *test.* "If so and so is true, back up nine squares. If it is false, just keep going."

The test will often involve a flag. For instance, we could "branch if carry set." This means that we test the carry flag. If the flag is set, we take the branch, going off so many squares in the address space. If the flag is cleared, we ignore this test and go on to the next instruction, as if nothing had happened.

We will find out much more about these relative branches when we look at loops and testing in the discovery modules of the next chapter. As we found out back in Volume 1, much of a micro's intelligence stems from its ability to test something and then change its course of action based on the results of that test.

How far do we go and how do we know the direction? We use 2's complement signed binary to tell us how far. Often, a relative branch will use only a single 8-bit byte to tell us both distance and direction. When this distance and direction byte is shown as 2's complement signed binary, this gives us the option of going forward from $00 through $7F squares, equal to +0 through +127 locations. The same byte also can let us back up from $FF through $80 squares, the equivalent −1 through −128 steps.

The notation for relative addressing can be confusing. It consists of the mnemonic and the absolute location you wish to branch to . . .

A relative addressing mode is shown by the mnemonic and the absolute address the branch is to go to . . .

BEQ $18A4

BUT . . . the op code will consist of a command followed by a 2's complement number telling us how far backwards or forwards to go . . .

F0 06

How do you tell relative addressing from absolute long addressing? By the mnemonic. Many mnemonics for many relative branches start with a "B." You get to recognize these after a while.

It also turns out that most branch commands that use relative addressing are available in *pairs*. If the right one don't get you, the left one will. Thus, you can branch on carry set with one relative command, and branch on carry clear on another. This is an example of *complementary* instructions.

Here are more relative addressing f'rinstances . . .

TYPICAL RELATIVE COMMANDS

BCC $1244—If carry is clear, branch to address hex $1244.
BCS $1244—If carry is set, branch to address hex $1244.
BEQ $1244—If last result equals zero, branch to address hex $1244.
BNE $1244—If last result was NOT zero, branch to address hex $1244.
BPL $1244—If the last result was zero or positive, branch to address hex $1244.
BMI $1244—If the last result was negative, branch to address hex $1244.
BRA $1244—Always branch to address hex $1244.
BNV $1244—Never branch to address hex $1244.

Now that last one looks kinda dumb. The relative command tells us never to go somewhere. But there are at least two uses for something like this. The first is that it preserves the symmetrical and complementary pairs of the other branches, and the second is that it is useful for debugging or working parts of a program that may have different uses in different places. The branch never is put into the

program ahead of time and changed as needed for later options or debugging.

One big advantage of relative addressing is that the code is position independent . . .

POSITION INDEPENDENT CODE—Computer instructions that can be relocated anywhere in memory without needing any modifications.

Thus, if you move a part of a program somewhere else, all the relative branches will stay the same, since six steps backward are still six steps backward, regardless of where you start. If you try moving a program with a bunch of absolute addresses in it, many of these addresses will have to be changed to suit the new locations of everything.

Relative addressing is usually faster and shorter than absolute addressing. Only two bytes of op code are normally involved, compared with the three needed for a long absolute address.

Relative addressing has several disadvantages. This mode is pretty much limited to "test and go" branches and jumps. You also are normally restricted to short hops of plus or minus 127 counts from where you are. But most useful "test and go" hops turn out much shorter than this, and you can always throw in an absolute jump if you want to go out of range.

Another limitation to relative addressing is that it thoroughly confuses beginners, since they have to calculate a 2's complement number to find the offset. We'll check into a simple "count the squares" method that will straighten this out for you later. When you get into using an assembler, this branch calculating hassle is taken care of for you.

To add to a beginner's confusion, if you move a relative branch, the op code stays the same but the operand changes! This is because six counts are six counts, but after a move, the "go to" location shown after the mnemonic is now pointing somewhere else in the absolute address space.

A few of the newer 16-bit micros let you do long relative branches of plus or minus 32767 counts that let you end up anywhere in a 65536-slot address space. You can partially fake this "long branch" with the short 8-bit branches used on older micros by branching to a location that holds a jump to an absolute address.

So far, we seem always to know exactly where we want to go. Our previous addresses have been known or fixed. Sometimes, it is nice to be able either to calculate an address or to go to an

unknown address, or to be able to go to any of a bunch of different possible addresses. To do this takes a powerful new addressing mode called . . .

You're vacationing in a strange part of the country and you want to find an old friend you know is around somewhere. Back East, you'd probably try the phone book, the city directory, the police, or even a local bar. But out here in the real West, it's no contest. You go straight to the little old lady in the post office.

Works every time.

"Well, go out past where the bowling alley used to be, through Nat Clemson's place, and then a fur piece down the old stage road. Sit with Crazy Andy for a spell so's he doan up an' shoot you, and then . . ."

What you are doing is going to one location to get the *address* you really want. That is, you go to a first address to find a second address. Micros use a similar scheme to let you go to a calculated or changing address. This is called *indirect addressing* . . .

INDIRECT ADDRESSING—An address mode that looks into one address to get the ADDRESS of where the action is to take place. It then goes to the address it just found to finish the action.

The number of bytes of op code and address locations needed varies with the micro family chosen.

Now, indirect addressing sounds like a runaround, and it is. But it's useful runaround. You can calculate an address or change an address as you go along. For instance, say you have some program that is menu driven. Say further there are twenty-six different options, A through Z. Pick an option. Then, with the value of this option, find an address that contains the starting address of where this action is to happen. Then go and do it.

Indirect addressing is very powerful. Its exact use changes with the micro family. Two of the more powerful types of indirect addressing are called *register indirect* and *absolute indirect* . . .

REGISTER INDIRECT—An addressing mode where a working register holds the final address for you.

ABSOLUTE INDIRECT—An addressing mode where some location in the address space holds the final address for you.

The 8080 school is heavily into register indirect action, while the 6502 uses absolute indirect.

For instance, there is a register pair in the 8080 and 8085 called the H and L registers. The H stands for High and the L stands for Low. You put an address into this register and then, if you use a register indirect op code, you load to, store from, or otherwise interact with the address pointed to by the H and L registers. Register pairs B, C and D, E can also sometimes be used to hold indirect addresses.

The 8048 offers register indirect addressing through its R0, R1, R0', and R1' working registers. The 8048 also has one variation on indirect addressing called *accumulator indirect*. Here, you put your absolute short address in the accumulator and the CPU then replaces the address with the contents of that address location. Sounds very powerful.

Sneaky, too.

One advantage of register indirect addressing over absolute addressing is that it is faster, since the address is already sitting there in the H and L registers. The CPU doesn't have to take time out to look into extra bytes as it would with absolute long addressing. It is also a simple matter to increment or decrement the H and L registers, so you can pick off sequential locations in a table of values or some other file in memory.

The 6502 school uses absolute indirect instead. One example is the jump indirect op code. When this is encountered, the CPU goes to the address shown by the second (low) and third (high) byte of the op code, gets the address of where it is to jump to, and then goes to this new address.

There are five bytes involved in an absolute indirect jump on the 6502. The op code takes three bytes. The first of these is the command, followed by the low address address, followed by the high address address. Then, out in the memory space, we have to put the address we are to go *to* in two more locations. Thus, there are three bytes of op code and two address locations involved.

Several even more powerful modes are provided on the 6502 that combine the upcoming indexed addressing with absolute short (page zero) and indirect. Together, these can form a powerful way to hit any location in the entire address space. These exotic indirect address modes are what give the real power to the 6502 school, despite its limited number of working registers.

One way to show an indirect addressing mode is by using parentheses . . .

An indirect addressing mode can be shown by a mnemonic, followed by an address in parentheses . . .

JMP ($27AF)

The location in the parentheses is the place we go to get the address we are trying to reach.

Notation on indirect addressing varies from family to family. Sometimes an @ or an X will be involved with the indirect addressing mnemonic. On register indirect, sometimes the register name will be tacked onto the mnemonic. For instance, mnemonic LDAB might mean "load the accumulator with the address pointed to by the B and C register pair." The programming manual or detailed data sheet for the micro you are interested in should show you details on this.

Anyway, an indirect addressing mode goes somewhere to find an address and then goes to that found address for the action. Indirect

addressing's main power lies in working with groups of addresses, unknown addresses, or changing addresses. We can reach any point in the address space and do so with a vengeance.

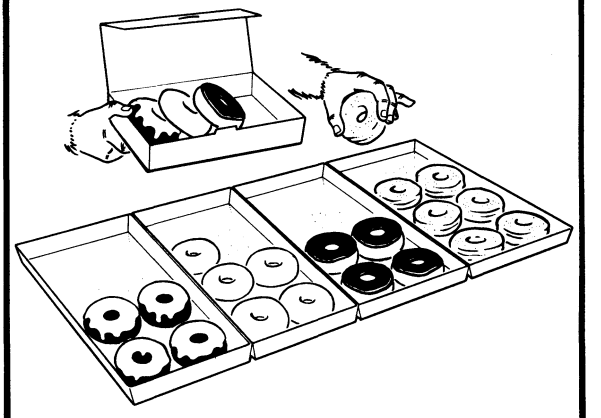
One disadvantage of indirect addressing is that it is complicated. You have to go to a lot of trouble ahead of time to put the address you need where the CPU can find it, and the stop along the way to pick up the address can add to the execution time.

Another place where indirect is handy is for reset or system start-up. On reset, we always want to start off in some known direction, going to an address pointed to in ROM. We can use an indirect jump to get from this address to where we really want to start. We can also play some double indirect games that let us jump to some RAM location once we are sure the CPU is up and off on the right track.

You'll want to save indirect addressing for later, but anytime you want to go to a calculated, a changing, or an unknown address, one of the indirect modes will do the job. A monitor routine that uses calculated addresses can reside in ROM. Only the indirect locations need be in RAM. Indirect addressing solves the dilemma of code that has to be both always there and changeable on the fly.

There is one more address mode we should look at. This one is very handy for moving things around in memory since it lets us work sequentially through a memory area. This mode is called . . .

INDEXED ADDRESSING



Have you ever gone into the 146 flavors donut store and ordered one of each for a party? If the counter person thinks about what

they are doing, they will go to the upper lefthand corner and start a tray at a time, picking off the donuts in sequential order. The same simple "one of each" command starts off a series of donut pickups, all of which are the same but each of which gets a different donut in sequential order.

Now, this sounds obvious, but you could order each donut separately. Micros face the same dilemma. With absolute addressing, you have to go to each location, one at a time, in a long and drawn out process. It would be much nicer if we could pick off bunches of nearby locations, either in sequential or random order, quickly and conveniently. This is done with an *indexed* addressing mode . . .

INDEXED ADDRESSING—An address mode used to reach groups of nearby addresses. This is done by adding an index value to a base address and then going to the sum of the two.

In our donut shop, the base value will be the upper left tray. The index will be the number of trays we have used so far. Note that the base value does not change. Only the index changes, so we can use the same indexed address instruction each and every time we grab a donut.

More terms . . .

BASE ADDRESS—The starting point, or lowest address in a file or other group of locations.

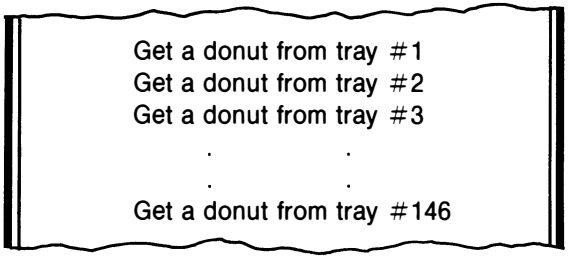
INDEX VALUE—The offset, or amount that is added to the base address to reach a particular location.

Let's look at an example. Say you have a table of some sort, stashed starting at location $0800 in a 6502. The 6502 has an addressing mode called absolute, indexed by X.

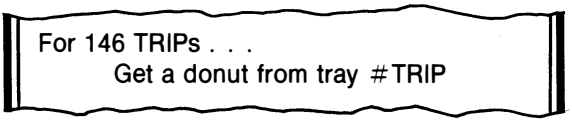
If we tell the CPU to LDA $0800,X, the CPU will find the index value in the X register, add it to the base address of $0800, and then go to the address that is the sum of the base and the index. It then gets whatever is in that location and loads it into the accumulator.

If there were an $07 in the X register, location $0807 would be loaded into the accumulator. The only time you would actually load from the base address of $0800 is when the X register, or index, equaled $00.

This can be confusing the first time you see it, but it is super powerful. Think back to the donuts. We could program our counter person to . . .



This would certainly work, but it sure needs lots of instructions. Here is a much shorter way . . .



The code is much shorter this way. This is the power of the indexed addressing mode. It lets us use one instruction to access bunches of different locations in memory, simply by messing with an index value.

The notation for an indexed addressing mode looks like this . . .

An indexed addressing mode is shown by a mnemonic, the base address, a comma, and finally the name of the index register used . . .

LDA $1800,X

The big advantage of indexed addressing is the ease with which you can reach adjacent locations in the address space.

There are several disadvantages to indexed addressing. First, it may not be available on some older micros. In this case, you can often play games with the register indirect addressing mode to fake the same thing. Second, the index range may be limited. You may only be able to index 256 locations at a time. Third, indexed may be slower than absolute addressing since it takes time to add the index and takes additional time in the program to change the index and test the loop or whatever you are using.

Usually, indexed addressing will involve lots of options. There may be a choice of index registers, and there may be a choice of an absolute short or absolute long base address. To really get powerful, some families, including the 6502, will combine the indexed

addressing with page zero and indirect addressing. The use of a page zero or absolute short address is shorter and faster than an absolute long one would be.

This mind-boggling combination can be used to go to any of a range of different addresses, or it can be used to pick any value out of any file anywhere in the machine. More on this later.

WHICH ADDRESS MODE?

Let's see where we are. We decided that there are lots of different ways to get a package to Albuquerque. These ways differ in speed, expense, convenience, reliability, and so on. Different people with different needs will pick different ways to get their individual packages to Albuquerque.

In much the same way, a microprocessor CPU has lots of different ways to reach working registers and address space locations. These different methods are called address modes. These address modes vary from micro family to family, but most address modes boil down to combinations chosen from seven basic types . . .

We need a way to do obvious things . . .
So we have an IMPLIED address mode.
We need a way to put fixed values into registers . . .
So we have an IMMEDIATE address mode.
We need a way to reach any address space location . . .
So we have an ABSOLUTE LONG address mode.
We need a way to reach a few address locations quickly . . .
So we have an ABSOLUTE SHORT address mode.
We need a way to step forward or backward . . .
So we have a RELATIVE address mode.
We need a way to handle variable addresses . . .
So we have an INDIRECT address mode.
We need a way to handle a file or a block of data . . .
So we have an INDEXED address mode.

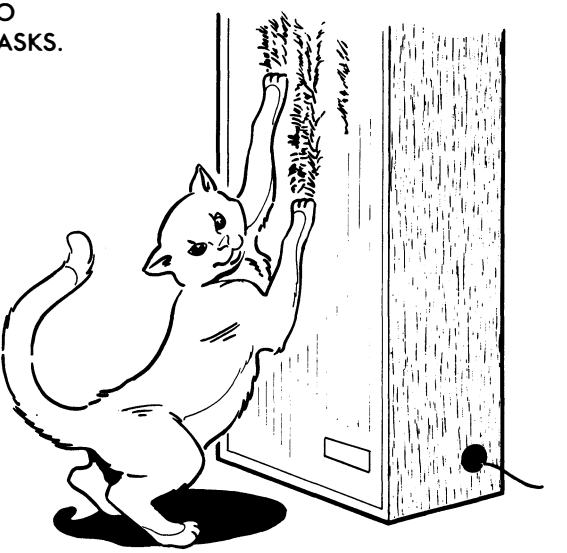
When you are writing a microcomputer program, you normally want to make the program as compact as possible, using as few instructions as you can. You also will want to make the program run as fast as you can. Because these two goals usually fight each other, you will have to use different address modes in different places and at different times.

Let's put all these address modes into a handy form for later reference . . .

1

IMPLIED ADDRESSING

USED TO DO
OBVIOUS TASKS.



LENGTH — One byte for command only.

SYMBOL — Just the mnemonic . . .

CLC

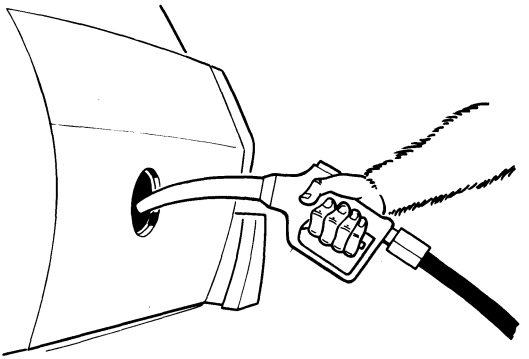
MAIN USES — Housekeeping and control.

ADVANTAGES — Short and fast.

LIMITS — Only works when no further info needed.
Cannot reach the address space.

IMMEDIATE ADDRESSING

FILLS A REGISTER WITH A FIXED VALUE.



LENGTH — Two bytes. One gives command. Second says "how much?" or "against what?"

SYMBOL — Mnemonic, followed by # symbol, followed by fixed value . . .

LDA #$17

MAIN USES — To put a FIXED value into a working register or to logically use a fixed mask against a working register.

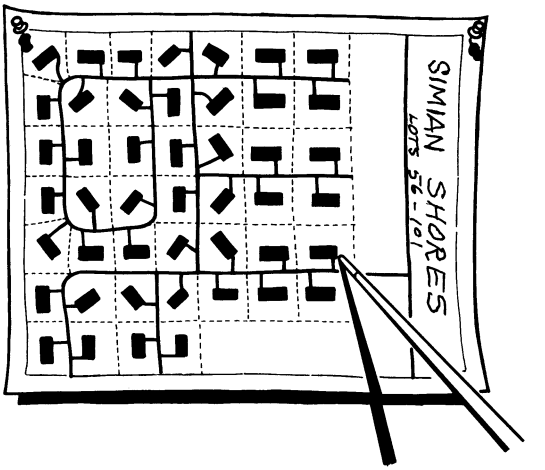
ADVANTAGES — Puts constants and fixed values into your program.

LIMITS — Only useful for fixed values. Cannot reach the address space.

3.

ABSOLUTE LONG ADDRESSING

REACHES ANY LOCATION IN
THE ENTIRE ADDRESS SPACE.



LENGTH — Three bytes. One describes action. The next two give the absolute location.

SYMBOL — Mnemonic followed by a full address . . .

LDA $FA62

MAIN USES — To definitely reach any KNOWN location in the entire address space.

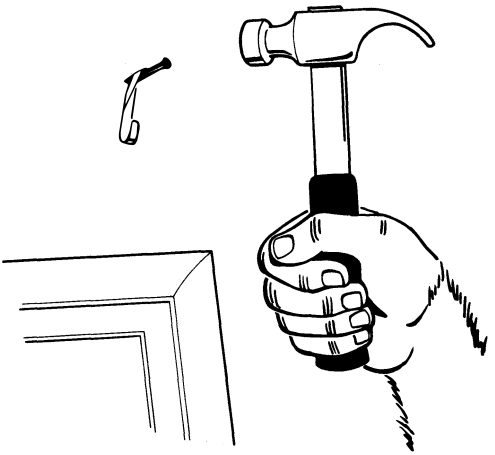
ADVANTAGES — Always works.

LIMITS — Slow and tedious. Exact address must be known and fixed.

4.

ABSOLUTE SHORT ADDRESSING

REACHES ANY LOCATION IN A KNOWN OR ASSUMED SMALL PART OF THE ADDRESS SPACE.



LENGTH — Two bytes. The first describes action. The second gives the location in the assumed small portion of the address space.

SYMBOL — Mnemonic followed by a short address . . .

LDA $62

MAIN USES — To definitely reach any KNOWN location in a small portion of the address space.

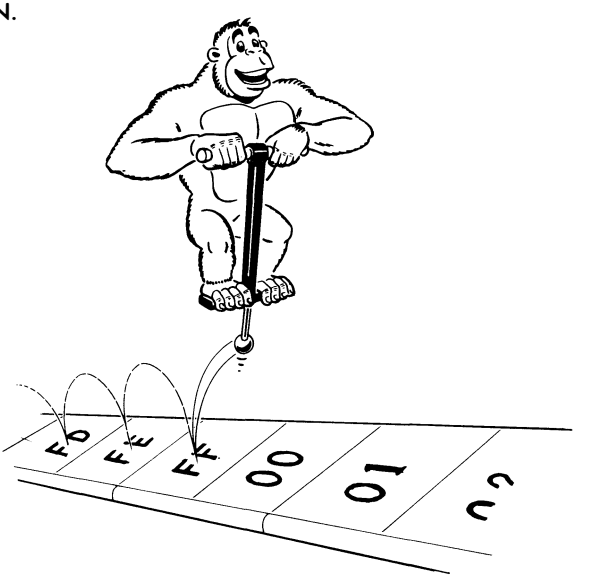
ADVANTAGES — Faster and shorter than absolute long.

LIMITS — May not reach entire address space. Turf fights over valuable locations.

5.

RELATIVE ADDRESSING

GOES SO MANY STEPS FORWARD OR BACKWARD FROM PRESENT LOCATION.



LENGTH — Two bytes. The first describes action. The second gives the LENGTH of the jump in 2's complement signed binary.

SYMBOL — Mnemonic followed by a long address . . .

BEQ $18A4

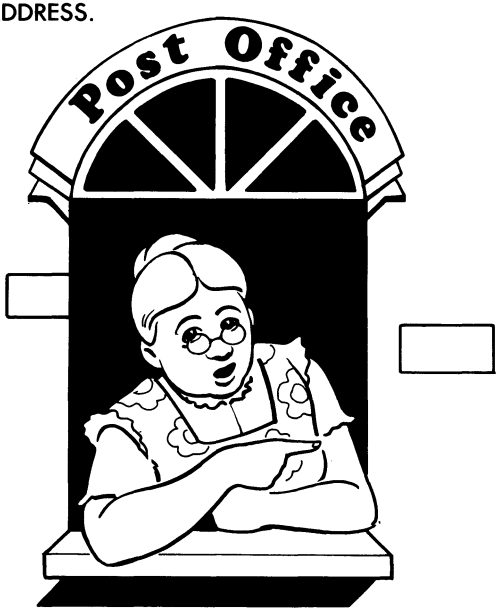
MAIN USES — Testing and branching to other parts of a program.

ADVANTAGES — Code is relocatable. Shorter and faster than absolute long.

LIMITS — Usually limited to conditional branches. Range may be restricted.

INDIRECT ADDRESSING

GOES TO ONE ADDRESS TO GET A SECOND ADDRESS, THEN USES THAT SECOND ADDRESS.



LENGTH — Three bytes. The first describes the action. The next two give an absolute address pair at which the final ADDRESS will be found.

SYMBOL — Varies with family. Often a mnemonic followed by a long address in (parentheses) . . .

JMP ($27AF)

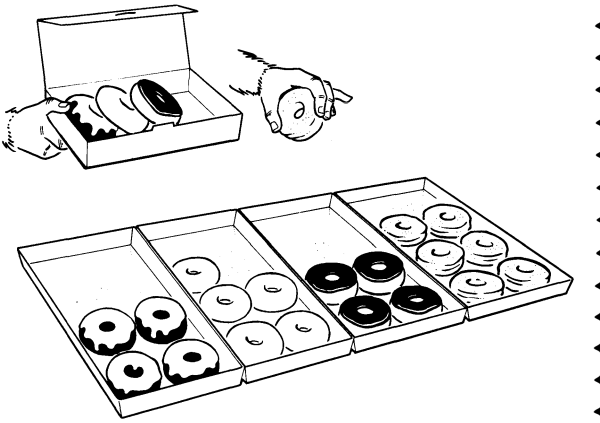
MAIN USES — Getting to a calculated address.

ADVANTAGES — Handles unknown, variable, or changing addresses. Very powerful.

LIMITS — Slow, long, and involved. Requires setup ahead of use.

INDEXED ADDRESSING

GOES TO AN ADDRESS THAT IS THE SUM OF AN ABSOLUTE BASE ADDRESS AND AN INDEX REGISTER VALUE.



LENGTH — Three bytes. One for action, the next two for the base address to which the index value will be added.

SYMBOL — Mnemonic followed by a long address, followed by a comma and index name . . .

LDA $1800,X

MAIN USES — Sequential or random access of nearby or related addresses.

ADVANTAGES — Very efficient at handling files and moving blocks of data.

LIMITS — Needs careful program design. Requires setup ahead of use.

Unfortunately, manufacturers have their own different names for each of their own address modes and the assembler symbols change from family to family. Worse yet, "nonofficial" assembly programs may substitute their own symbols and ways of handling address modes, even for the same micro chip. Address modes are sometimes combined with each other into more powerful but specialized ones. Also, address modes may be subdivided so it sounds as if you have lots more available than you really do.

To add to these hassles, not every mode is available on every machine. If a mode is not available, there will often be some other mode that fills in for it. You can do almost any task on almost any micro, one way or another. We have seen how micros with absolute indirect addressing may lack register indirect addressing, and so on.

Some examples. The name that the 6800 gives its page zero or absolute short addressing is "direct" addressing. Their idea of absolute long addressing is called "extended." Early 8080 literature speaks of an "implied" addressing mode that we know today is really a register indirect addressing mode. The 6502 offers four different indexed address modes, involving X and Y index registers and absolute short (zero page) and absolute long (absolute) base addresses. They also have an "accumulator" address mode that is nothing but an implied address mode involving only the accumulator. There are also some super-powerful address modes that combine indexed addressing with indirect and page zero addressing. These are especially strong when handling files and moving data beyond the 256-byte range of single 8-bit commands.

I won't try to show you all the different names for all the different address modes of all the major micro families. As you get into any one family, you will find that most of the available modes break down into mix-and-match combinations of the basic seven we just looked at. Instead, you do it . . .

DOING IT:

- ( ) Look into the addressing modes available for two micros of your choice.
- ( ) What names have they put on each mode? Ignoring the name, what does the mode really do?
- ( ) What are the assembly language symbols for each address mode?
- ( ) How is each mode related to the seven basic address modes we have looked at?

Now, if you look into the newest microprocessors, you will find lots of different address modes. But almost all of these fancy modes are really only mix-and-match combinations of our basic seven modes.

We might find a super-long variation on absolute long addressing that lets us hit any slot in a sixteen-megaword address space. We can expect improved relative addressing that applies to more instructions and reaches farther. A 16-bit relative address is sometimes called a *long branch*.

We can expect all sorts of combinations of indexed and indirect modes that aggressively cover the full address space. And the newest machines will give us lots of choices as to what a word is. The addressing modes may apply to individual bits, to 4-bit bytes, to 8-bit words, to 16-bit words, and even to paired 16-bit words.

You will also find some simple commands that can do very involved and fancy things, again based on the seven basic address modes. For instance, *relocatable* addressing lets you move a program anywhere you want without having to recalculate or change everything. This usually uses combinations of indexed and indirect instructions, along with a relocation file or table of some sort. The big advantage of relocatable code is that it fits anywhere you want it to and is more or less machine independent.

A fancy variation on relocatable addressing is traditionally called *virtual* addressing. In virtual addressing, address space locations placed on a floppy disk's tracks or elsewhere is moved from and to the micro, giving the illusion of immediately having an enormous address space. In reality, you use only a small part of the address space at once, so you pick only what you need to use at any one time.

Another example of a fancy addressing mode is called *block move* addressing. In block move addressing, one command will move the contents of an entire block of many data locations from one area in the address space to another. But all this really takes is repeated use of our basic indexed addressing mode. The Apple II system monitor has a powerful block move command that is faked by repeated use of simpler address modes.

You might also find some very oddball VCIW addressing modes that won't fit any nice mold, but these are very rare. One example is called *associative* addressing, in which the content of a memory location is partly specified by that location's address. Manipulation of humongous files, such as those needed at a regional air traffic control center, might use this mode.

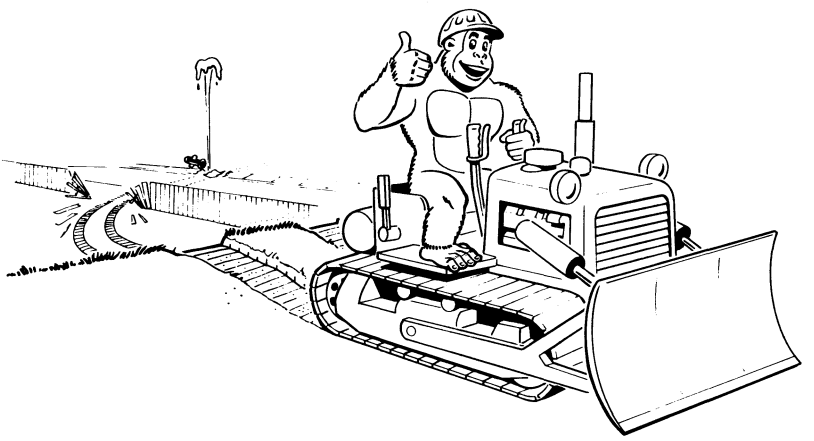
Some 16-bit micros have a *trace* addressing mode that keys on certain coded addresses. This is a powerful debugging technique that also goes by the name of *signature analysis*. Other new micros split their address modes up depending on whether you are in a "user" or a "supervisor" mode, or they may limit access to certain

address space areas. Still other new micros work directly in "higher level" languages that prevent you from ever reaching or controlling the actual address modes being used.

But take away all the fancy stuff, find out what is really happening, and you are back to the seven basic address modes. Get these down so you can understand and use them, and you will be able to handle almost any mode on any machine. As with everything else we've looked at, it takes practice and use to master address modes. Just reading about them simply won't hack it.

THE RESOURCE SHEET

Do you know anything about running a bulldozer? . . .



Dozer operation should be simple enough. Just jump on, hit the starter button, and away you go. Right?

Wrong.

It turns out that many bulldozers do not have an electric starter. Instead, there is a gasoline *pony engine* on the side of the main diesel engine. To start things up, you electrically start the pony engine and then use the pony engine to start the main engine. There is an elaborate and exact procedure you have to go through.

The same is true of most microcomputer systems. Although some simply let you "punch and go," there is usually one right way to bring up a microcomputer to the point where someone can use it. This is just the same as there being one correct way to start a bulldozer.

A micro resource sheet helps you understand what a microcomputer consists of and how to get it working . . .

RESOURCE SHEET—A card that tells you all the component parts of a microcomputer, what goes with it, how it is powered, and how to start it properly.

This may sound obvious and trivial, but if you have several micro systems, and several people using them, the resource sheet can eliminate all sorts of confusion and hassles.

You can easily make up your own resource sheets. They are absolutely essential in schools, labs, clubs, and other multi-user places where people you may not totally trust can get their hands on your micro.

Here is a typical resource sheet, this one for a Z-80 Starter . . .

Z-80 STARTER

EAC #24331
Large Green PC Board

A. PARTS:	1 - Z-80 Starter Microcomputer 1 - Z-80 Pocket Card 1 - Z-80 Programming Manual
B. POWER SUPPLY:	Needs external +5-volt, 2-ampere supply. WARNING—Voltages over +5 will destroy this unit.
C. BRINGING IT UP:	1. Connect red to +5 and black to − supply lead. Jumper supply − to supply case ⏚ terminal. 2. Disconnect red supply lead. Turn supply on, and set voltage to +5 and current limit to maximum. 3. Turn power off, reconnect red lead, and turn power on. 4. Press BLACK reset button at top of trainer. A "−" should appear on leftmost digit of display.

Z-80
(CPU)

You can decide what is important for your own resource sheets. This one gives the name of the trainer, what it looks like, and who owns it at the top. After that, we list everything that is supposed to go with the trainer. Then, we show the type of power supply

needed. This is followed by detailed instructions for bringing up the micro. Finally, we show the microprocessor used in the system.

DOING IT:

( ) Create resource sheets for a micro trainer and a personal computer of your choice.

It is particularly important to have detailed start-up instructions when an external power supply is needed. Beginning users always get the − and case leads on a supply confused and often do not understand how a current-limited power supply works. One Z-80 trainer I use got several chips fried when a student had the current limit set too low and turned up the voltage anyway. When the reset button was hit, the current dropped momentarily, raising the voltage well above +5 volts and roasting a bunch of chips.

Anything that can be destroyed easily should have its procedures spelled out on the resource card. The keyboard connectors on the AIM-65 trainer are one example. These ordinary DIP cable plugs simply aren't rugged enough for beginning students to use.

You'll also want to put a permanent copy of the resource sheet wherever the trainer lives when it is not in use. This prevents mix-ups and helps insure that everything you need stays together.

Start-up procedures change from trainer to trainer. Some, like the old KIM-1, have you punch lots of keys to get them started off on the right track. Others, like the Apple, need warnings NEVER to make or break any connection ANYWHERE on the machine while the line cord is connected. Each and every machine is different.

Don't omit anything from your resource sheet just because it would be completely obvious even to an idiot. If it can be done wrong it will. There are some students and some industry types who can break an anvil just by walking within a few feet of it.

Spell everything out. Completely.

THE MICRO TOOLKIT

Let's take a quick quiz. What is the secret of success in any of the following?

( ) Mountain climbing
( ) Watch repair
( ) Heavy equipment maintenance
( ) Spelunking
( ) Sign painting
( ) Weaving

The obvious answer is "the right tools to get the job done." For just about anything you set out to do, there are correct tools that, properly used, make just about any task simple and easily done.

The same is true of microcomputers. There are bunches of basic tools you need to deal with micros expertly and effectively. Many of these tools are simple and cheap, often nothing but the right form on the right sheet of paper. Others take time and effort to pin down.

Here's what I consider to be the essential tools you absolutely must have as a beginner to understand microprocessors and microcomputers . . .

THE MICRO TOOLKIT

(0) The right attitude

(1) A microprocessor trainer
(2) Resource sheet for trainer
(3) Pocket card for trainer
(4) Pocket card for CPU

(5) Manual for trainer
(6) Programming manual for CPU
(7) Simplified memory map
(8) Programmer's model
(9) Simplified I/O circuit

(10) Machine language forms
(11) Assembly language forms
(12) Hex dump forms
(13) Quad graph paper pads
(14) Logic template

(15) Highlighters—all colors
(16) Pencils
(17) Lots of large erasers
(18) 3 × 5 cards—many colors
(19) Oscilloscope (optional)

(20) Glomper clips, 16/24 pin
(21) Grabber test leads
(22) Usual electronic hand tools
(23) Two quiet workspaces

This sounds like quite a pile of junk. But, properly used, this pile of junk will introduce you to micros and then let you understand them. Much of the paper stuff will later on get done by the computer, but it is absolutely and utterly inexcusable for someone new to micros to start using things like assemblers and program development aids without fully and thoroughly understanding *exactly* how micros work and exactly how to use them.

These tools should let you deal with the micro of your choice on your own terms. We'll find out just how in the next chapter. For now, let's take a more detailed look at some of the tools in the micro toolkit.

the right attitude

This one goes at the top of the list. Forget all those stupid lies that people have been yapping at you since year one about doing things right the first time and never making mistakes.

Let's repeat some key points from Volume 1 . . .

( ) Your first attempt at anything you do with micros WILL be wrong.

( ) Mistakes are ABSOLUTELY ESSENTIAL to learning and using micros.

( ) You are never anywhere near where you think you are in any micro problem.

( ) The simplest possible models and smallest possible steps must be taken at all times.

( ) Complete and continuous documentation of everything you do is ABSOLUTELY a MUST!

( ) The unsolved and unworked part of any micro anything will ALWAYS be much more of a hassle than you expect.

Or to quote Murphy, anything that can go wrong, will.

If you are not willing to accept or believe this, then go away. Right now. Better yet, go get a job designing dinos or old line minicomputers. This will do a great service to humanity by killing two birds with one stone.

To win with micros, you must have the right attitude and the right frame of mind. Nothing past tool number zero will help you if you do not.

a trainer

The best possible way to learn a microprocessor is with a trainer, such as the ones we looked at back in Volume 1.

What you want is something simple that lets you directly write machine language programs, single stepping and debugging as you go along. That something must have a simple and useful system monitor and must have parallel ports on board and ready to go.

A personal computer will do if (1) you can take it apart, (2) it easily and conveniently speaks machine language, (3) it has single step, trace, and debug features in a powerful system monitor, (4) you can do an absolute reset to machine language, (5) parallel ports are available, and (6) you don't feel bad about running around barefoot inside an expensive machine.

One very poor way to learn micros is with a microprocessor development system or demo board. As we've seen, these are far more expensive and generally less useful than trainers.

The worst possible way to learn micros is to get yourself a CPU, some RAM, and some EPROM and build yourself a system from the ground up. Working with chip sets is bad because you have no system monitor until you write one. This creates the worst sort of chicken-and-egg problem you ever saw. A second reason to steer clear of chip sets is that somehow you have to be able to breadboard them. And this can lead to rat's nest wiring, wirewrap horrors, super-expensive PC layouts, and other problems. The saddest thing about all this is that your focus ends up on hassles peculiar to your own system, rather than on understanding and using microprocessors and microcomputers in general.

Now, if you want to design some dedicated micro controller using your own chips, that's probably okay. Dumb, but okay. Starting with chips almost certainly will end up costing more than a trainer and will give you fewer features, but if that is your trip, fine. Just don't expect a chip set to teach you microprocessors and machine language, because it won't.

No way.

Trainer or whatever, try to pick a machine that uses the CPU family you think you eventually will be working with. For the 8080 school, the Z-80 starter or the HP 8800 is a good choice. For the 6500 school, there's the AIM-65, the SYM-1, or the good old KIM-1. Avoid using oddball trainers such as 6800 or COSMAC 1800–based units unless you are actually going to try to do something useful with these VCIW families. Stay in the mainstream unless you have

an overwhelming reason not to. A trainer that is $50 cheaper is no reason. It's not even an excuse.

pocket cards

We've already seen that micros have pocket cards available to give you all the key info you need at a glance. You will need two pocket cards, one for the trainer or microcomputer system and a second for the CPU itself. Be sure to label these so you can tell them apart at a glance. Then get yourself the most vicious junkyard dropout guard dog you can find and keep these cards under his collar.

manuals

Most trainer manuals are utterly and totally atrocious. The only thing worse you are likely to find is a manual for some dino product. The same goes for most programming books.

Unfortunately, you are pretty much stuck with this trash and have to live with it, for this is the way things have been done in the past. One thing you can do that will help bunches is to go through the manuals and color-code all the important details with page highlighters.

Another thing you can do is ask around for the best available software and hardware books and manuals for any micro. Stuff that looks really bad the first time through may not be that awful after you know what you are doing and have some experience under your belt. More material may be available through a user group or special interest section of a local club. But don't expect miracles.

If you are a beginner, avoid totally anything that uses assembly language. This is great stuff later, but for now, you must concentrate on the fundamentals of machine language.

maps, models, circuits

We've already looked at the simplified memory map and the programmer's model. There are two good reasons for doing your own version of these. The first is that usually there is far too much detail in what you will find with the system documentation. The second, of course, is that redoing something yourself forces you to focus on what is happening and why. This is a key learning process that is hard to pick up any other way.

There's also something called the Simplified I/O Circuit that we will look at in Chapter 8. This one shows you where all the I/O ports are and how you use and connect them.

forms

There are three forms you should have photocopied by the hundreds. The first is called a machine language programming form and looks like this . . .

MACHINE LANGUAGE PROGRAM FORM

DONE BY ☐ PROGRAM ☐
DATE ☐ VERSION ☐
RUNS ON ☐
CPU ☐

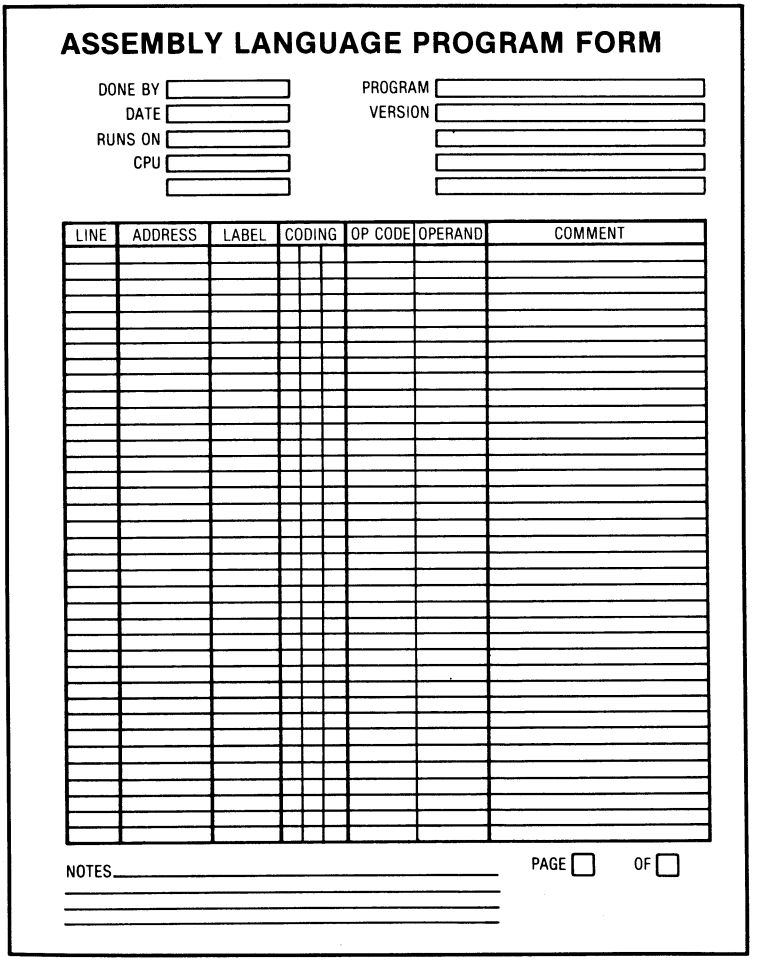
ADDRESS	OP CODE	BYTE #2	BYTE #3	MNEMONIC	HOW?	NOTES

NOTES_____ PAGE ☐ OF ☐

The machine language programming form is used to write and understand your simplest microcomputer programs. You will learn much more about this in the next chapter.

By the way, if you want larger copies of any of the forms I am showing you, find a local Xerox machine that enlarges. Make one big copy, and then copy the copy as often as you like.

This is an assembly language programming form . . .



The assembler form is used to write longer and more involved programs AFTER you are an experienced programmer. It differs from a machine language form in that it has line numbers for editing and provisions for labels that identify certain points in your program.

Once again . . .

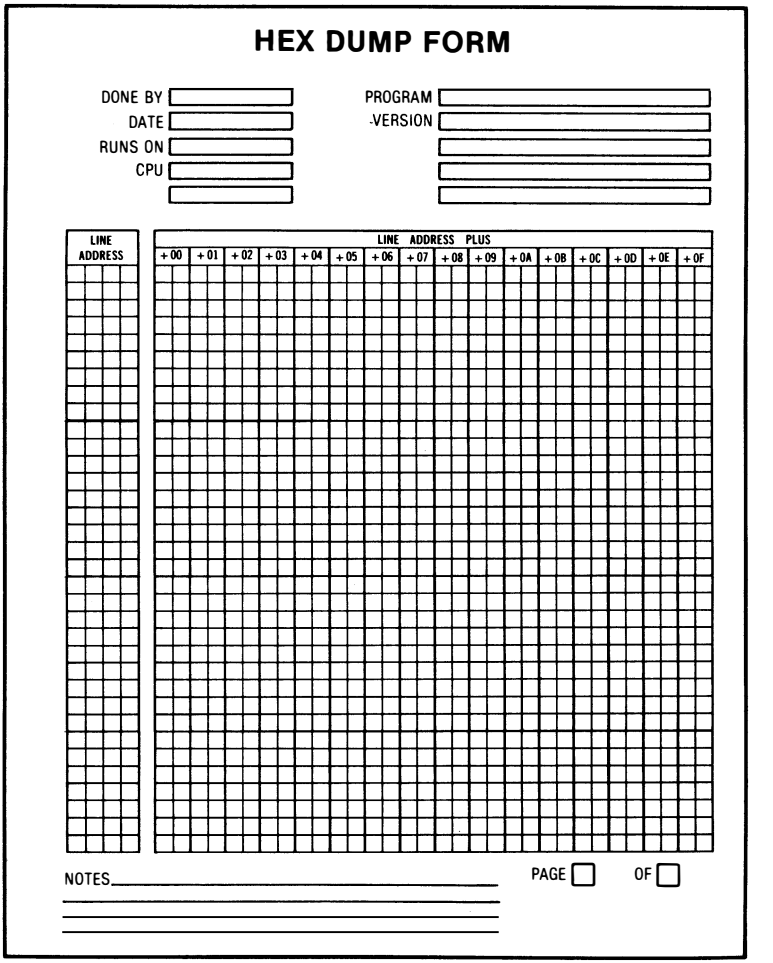
A NO–NO!

DO NOT USE ASSEMBLERS OR ASSEMBLY LANGUAGE FORMS UNTIL WELL AFTER YOU ARE AN EXPERIENCED MACHINE LANGUAGE PROGRAMMER!

Actually, you will use this assembler form very little, because once you get into assembly language programming, you will probably have a printer and an Editor/Assembler program to do all the work for you. Editor/Assemblers eliminate practically all the dogwork of machine language programming. The problem is that assembler programs throw out the baby with the washwater. When the dogwork goes, so does insight and understanding.

And, more importantly, so does the discipline needed to attack any micro task at any level.

Here's a form called a hex dump . . .



The hex dump is used to save an exact image of what is in a micro's address space. These are useful for files and for complete programs, when you want to have the program in its most compact form.

The advantage of the hex dump is that it is the simplest and most compact way of storing data or entering programs. The disadvantage is that hex dumps totally lack documentation and give you no clue to what the code is or what it does.

from the office supply

You will need several pads of quality quadralle graph paper. These get used for timing diagrams, flowcharts, and anything else when you want to combine words with sketches. I like the ten-to-the-inch type myself, although these are harder to get. You may prefer using a journal that has graph paper pages. If you do, you will still need pads for first tries, mistakes, corrections, and so on.

Get yourself a logic template. Notice I said logic template and not a programmer's flowchart template. On a micro, all you will ever need in the way of flowchart symbols are a rectangular box, a diamond symbol, a title oval, and straight lines with arrows to connect everything. These needed symbols are easier and cheaper to pick up with the logic template, and you can also use the logic symbols for interface diagrams and so on.

You will want to get yourself bunches of page highlighters. Be sure to get lots of different colors. Most assortments seem to give you six shades of yellow, so look around. Get several of every color you can find. Thin ones and fat ones. Light, see-through colors.

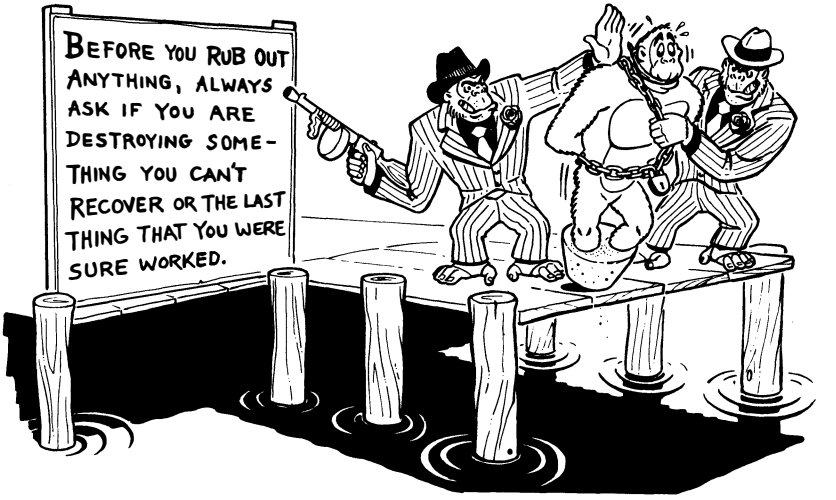
And use them. Anytime you find anything interesting or important anywhere, highlight it with some color code meaningful to you. The first place to start is the title page of the programming books. Highlight each mnemonic as it crops up. The highlighters are most useful when you are analyzing a longer machine language program. Color coding lets you break up the program flow into understandable small pieces.

I hate pencils. I am a pen person. So do as I say, not as I do . . .

Pens have no place whatsoever in a micro toolkit.

ALWAYS USE PENCILS!

Lots of big erasers are obviously needed for learning micros. But only use erasers to correct small and obvious defects and to make small and obvious changes . . .



Beginners often end up erasing their last copy of good working code and then replacing it with something dumb that doesn't work. *Always think before erasing.*

A final need from the office supply is several packs of 3 × 5 cards in different colors. These cards will be your key to completely learning the microprocessor of your choice. You will find out all about these in the next chapter.

Will you ever.

oscilloscope

You don't absolutely have to have an oscilloscope to learn micros, but you'll miss most of the insight and most of the opportunities if you do not.

Since scopes are super expensive, it pays to try to rent or borrow one, either by signing up for a hands-on college or computer store course or by asking around at a micro club or bulletin board system.

I use a *Tektronix* 455 myself, but this is sort of a heavy. A scope from their newer 2200 series is a better choice. What you want in a scope good for learning micros on is one with at least a ten megahertz bandwidth, triggered sweep, two vertical inputs, and vertical delay. If you can afford it, sweep delay can also be very handy.

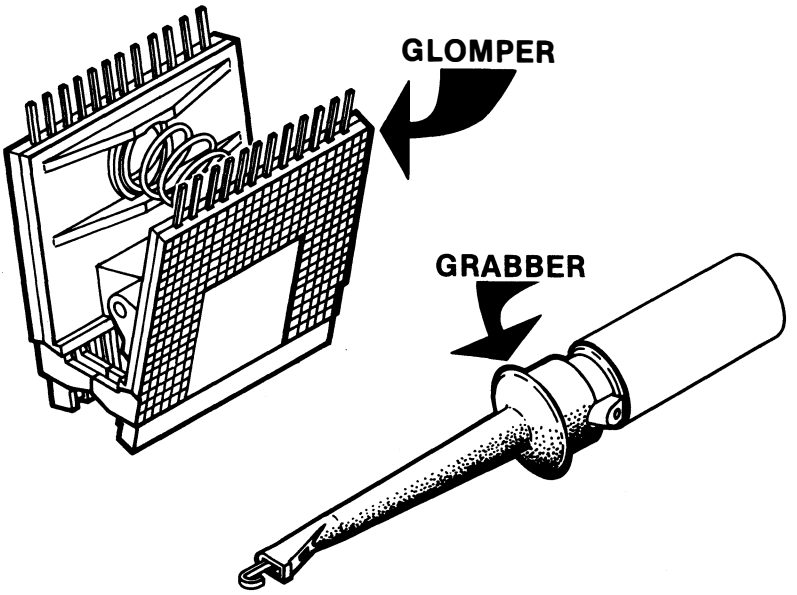
An oscilloscope lets you look into the actual working waveforms of a running microcomputer, giving you immediate details of timing and the relationships between various buses and signals. Scopes are also handy to check states of input and output ports, measure voltages, and so on. Oscilloscopes are absolutely essential when you repair, interface, or modify a microcomputer system.

So plan on learning what an oscilloscope is and how to use it as part of your micro learning experience. But don't run out and buy one. Borrow one or sign up for a course that gives you free access to one till you find out what scopes are and what they can do for you.

Another very useful tool is a general purpose volt-ohmmeter. The plain Radio Shack jobs should do and should cost less than $30. Do not get sucked into buying a digital voltmeter instead. Besides being more expensive, digital instruments aren't nearly as readable, useful, or convenient as a plain old meter-style VOM.

glompers and grabbers

Two other handy test aids are called the *glomper* and the *grabber* . . .



Glomper clips snap onto an integrated circuit and bring out all the pins to where you can conveniently reach them for measurement or scope viewing. Grabbers are tiny hook clips that let you

safely catch one lead of an integrated circuit or other small component without shorting adjacent pins. *AP Products* is one manufacturer of glompers, while *E-Z Hook* supplies many of the grabbers. Selections of these appear in most new-age electronic distributor catalogs.

Be sure to get the type of grabber that is small enough to safely grab a single pin on an integrated circuit. Larger ones simply won't do. By the way, any short on any pin of any integrated circuit is almost certain to wipe out the program in a micro, and some worst-case shorts, however brief, can also destroy the ICs themselves.

Always be careful!

electronic hand tools

A collection of the usual electronic hand tools is something you will want to pick up as you go along. Here are a few of the essentials . . .

ESSENTIAL HAND TOOLS

( ) Small soldering iron
( ) Diagonal cutting pliers
( ) Needle-nose pliers
( ) Small screwdriver
( ) Medium screwdriver

( ) Wire stripper
( ) Solder sucker
( ) Desoldering braid
( ) Small magnifier
( ) Small crescent wrench

( ) X-acto knife
( ) High intensity lamp
( ) 1/4-inch nut driver
( ) Third hand PC vise
( ) IC puller

You can buy most of this list as a set from *Heath* or *Jensen Tools,* but it is usually better to pick up what you need as you need it. Again, a school course, a club, or an electronic bulletin board can put you onto free tool use without your actually buying anything. This is a good way to get started, but you will almost certainly want to pick up your own tools as you go along.

workspace

Notice that the last entry on the micro toolkit list says you need *two* quiet workspaces. One is where the trainer is. The other is where you can quietly think and manipulate ideas and programs on paper.

It is extremely important for newcomers to micros to keep their grubby mitts off the microcomputer until they know exactly what they want to do with it. You don't just sit down in front of the thing and start punching in code. Instead, you have to plan out carefully exactly what you are going to do and how you do it . . .

> **Be sure to have TWO separate workspaces.**
>
> **The FIRST place is where you quietly design, develop, debug, and document your programs**
>
> **The SECOND place is where the micro lives and where you do actual coding and testing.**

Photographic darkrooms always have two separate workspaces. You must always keep the wet side and the dry side separate. The two different sides do different things in different ways. Mix them and you get a sloppy mess.

It's the same with micros.

And here's an essential rule . . .

The SOONER you start punching code into a microcomputer, the LONGER the task will take.

Plan before you punch.

In a later chapter we'll find out all about the *Micro Applications Attack,* a way to use micros to solve real-world problems. It turns out there is no need ever to go near a micro until something like the tenth step of a fourteen-step process.

Later on, you'll find yourself doing more and more development and debug work at the microcomputer, as you get into Editor/Assemblers, saving longer programs on disk, using emulators, etc.

But even then, you must keep two separate workspaces in your head. Micro as design and debug helper must remain totally separate from micro as running real-world applications programs. By

forcing yourself into a two-space attitude and two-dimension work habit ahead of time, you'll be way ahead of the game.

Well, we are just about ready to embark on the dark unknown of microprocessor programming. But before we do . . .

DOING IT:

( ) Assemble your micro toolkit.
( ) Arrange for a trainer.
( ) Get access to an oscilloscope
( ) Pick up the needed hand tools.

And now, as Von Neumann once said, "Let's get with the program . . ."

ON BEING A GENERALIST

The key feature that makes micros what they are today is that they are very *general* tools that can be customized to handle any *specific* task you can dream up. You have a "one size fits all" machine that can do almost anything for anyone. When the right software is added to that machine, it does one limited and specific something for a single someone.

Your software should also be as general as possible. Good software should do as many different things for as many different people in as many different ways as possible.

Let's look at two winning examples. These are, of course, *VisiCalc* and the *Adams Adventures.*

With hindsight, the idea behind *VisiCalc* is completely and totally obvious. Only no one thunk of it. Here was this drapery estimator adding up rows and columns on a spreadsheet. Over there was someone keeping church attendance records by adding up rows and columns on a spreadsheet. Stage left was a sales manager doing his forecasts by adding up rows and columns on a spreadsheet. Out in the field, a biologist was tallying observations by adding up rows and columns on a spreadsheet.

Now, no way would the biologist buy a drapery estimating program. And neither the drapery estimators nor biologists *by themselves* create enough of a market to be worth writing sanely priced software for. But just about everybody can use something that adds up rows and columns on a spreadsheet, no matter how specialized the information they are putting onto that spreadsheet.

The *Adams* adventures do pretty much the same thing. You see, there really is only *one* main adventure program. All that changes when you move from adventure to adventure is a data base file that is tacked onto the end of the program. Once your first adventure program is tested and debugged, you can go on and add new adventures forever, just by dropping in new data bases.

The *Adams adventures* main program also does its thing by being generally useful to many different data bases at once.

Anytime *you* write a program, always make it as general as you can and as flexible as you can, so many different people can use it many different ways to do many different things. Particularly in ways you wouldn't even dream of.

Think generalist. Act generalist. Be a generalist.

— seven —

The Discovery Modules

Having got this far, you are almost ready to start writing and debugging your own machine language programs.

We will use a technique called the "those #$!$# cards method" that will let you understand and use just about any microprocessor family from just about any micro school, present or future. We will take a "discovery" approach, working with a group of very simple yet fiendish modules. Each module builds on the previous one until you have fully explored most of the fundamentals of the micro of your choice.

Throughout, we will emphasize method rather than specific details. This way, you can easily apply what you do to most any micro family. We will break things up into two parts. The first part will show you what a machine language program is and how to write and debug simple code. That's what this chapter is all about. Later in Chapter 9, we will look at the *Micro Applications Attack* that shows you how to tie in simple program techniques with everything else needed to solve real world problems.

Beginners are always surprised to find out that punching code into the machine is only a tiny and trivial part of real world problem solving, something that happens only very late and plays a very minor role in an applications attack.

I feel very strongly that the only way to learn microcomputer programming is to start with machine language, rather than with an Editor/Assembler language.

Now, an assembler is not something inherently evil. An assembler is simply too powerful and too convenient a tool for a beginner. It hides from you the reality of what is actually going on in the micro on the gut level. It's sort of like flying a 747 jetliner instead of a trainer aircraft for your first solo flight.

Machine language programming is tedious dogwork. No doubt about it. And it takes bunches of patience and persistence, and it will be very frustrating. But doing your first programs in machine language rather than assembler language is *absolutely essential* for understanding and exploring a micro family.

Just note as you go along that anything really tedious or really bad involving machine language will get done "free" later with a good Editor and Assembler, but at the price of putting things between you and the machine that mask what is really going on.

Machine is a trip you must take on your own. The reasons to do so, of course, are that machine language programming can be insanely profitable, and that machine language is far and away the fastest running, most fully using, and most flexible way you can possibly interact with a micro of your choice. To repeat, truly great programs can ONLY be written in machine or assembly languages—there are no other alternatives. The "top thirty" programs for virtually all personal computers run in machine language, with practically no exceptions.

A crucial rule . . .

PLEASE!

Do NOT attempt to use an Editor or an Assembler until you have written, debugged, and fully tested not less than several hundred lines of hand-coded machine language instructions!

Many of our coding examples will use the 6502. First, I like this chip and know it best. Second, the 6502 is absolutely and undisputedly the funkiest microchip available anywhere ever. And finally, the 6502 is a very friendly chip to learn and use. It has very simple timing and hardware needs.

Since we are going to emphasize method rather than details, it won't really matter what chip you pick from which family for your first venture into microcomputing.

If you haven't done so already, put together the micro toolkit of the previous chapter, since you will need most of it here. A trainer is strongly recommended. If you must use a personal computer instead, be sure you can conveniently get it into machine language. It must have single step, trace, breakpoint, and debug abilities; an absolute reset into a machine language monitor; and at least one simple parallel 8-bit input/output port available for use. Naturally, you have to have the works open so that you can look at and measure individual pins on individual chips.

Before we write a program, though, we might want to ask the obvious question . . .

WHAT IS A PROGRAM?

A program is a series of machine language instructions that does some task. Get that? . . .

> **PROGRAM**—A series of machine language instructions that does some task.
>
> Hopefully, the task will be both useful and intended.

As a reminder, ALL microprocessors and ALL microcomputers can ONLY run machine language coded instructions on the gut level. Higher level languages place a machine language program called an *interpreter* or a *compiler* between the high level code and the hardware. This compiler or interpreter changes the high level code into the binary ones and zeros that machine language is all about.

A typical machine language program will have two different types of bytes in it, the *machine instructions,*and *data blocks.* The machine instructions are the op codes and the "with what?" and "where?" qualifiers needed to go with those op codes for certain address modes. Data blocks hold any information that the machine language instructions work with, such as text files, tables of addresses, graphics, color patterns, musical notes, or whatever.

Thus . . .

> **Machine language programs hold both MACHINE INSTRUCTIONS and DATA BLOCKS.**
>
> **The machine instructions are run by the microcomputer.**
>
> **The data blocks are accessed and used by the machine instructions as needed.**

One popular form of data block is called a *file.* A file is a fairly large block of information that is accessed as needed. A short file that only holds a few bytes is sometimes called a *stash.* We will see much more on files and stashes later.

By the way, it is finally time to do in the "data are" people once and for all. If you ever hear someone speak the words "data are" or "datum is," please immediately jump up and scream the word "FROBOZZ!" five times. If you see the same thing in print, send the

author five separate postcards, each with the single word "FROBOZZ!" on it. Then ask five friends to do the same thing.

An important point . . .

Only machine instructions can be "run" on a microcomputer.

A data block can be used only by a program. Try to "run" the data block and the system bombs.

Not every program needs and uses data blocks. Sometimes, small amounts of needed data will be built into the machine language instructions themselves. But programs that are better, longer, and more flexible will almost always have relatively short machine language blocks working with fairly long data blocks of one sort or another.

One tremendous advantage of the "short program with lots of data" route is that you can make the program do other things simply by changing the values in the data block. For instance, once you have designed and debugged the code for one Adventure, if you have used your data blocks right, all you have to do is change the blocks to change to a brand new and totally different Adventure. The fully debugged and tested instruction blocks will still work with the new data blocks.

As a simpler example, a traffic light program using data blocks can immediately be changed to a disco chaser program, a pendulum model, or a theater lighting control. Custom "instruction blocks only" programs would have to be rebuilt from the ground up each time.

Von Neumann and company

Just how do we arrange the data blocks and instruction blocks into a microcomputer? One obvious way is to put all the instructions in one place and all the data in a separate place, even if the data is only a single value. This is simple and obvious.

But it is not the way people think. People think by *mixing* actions, tests, and the data needed for those actions and tests *together* in sequence. Most microcomputers have their instructions arranged so the instructions and the data values needed to go with those instructions are combined in sequence.

This is called Von Neumann architecture . . .

VON NEUMANN ARCHITECTURE—A method of arranging the op-code instructions, tests, and any data values needed by those instructions together in sequence.

This "mix and match" of op codes, tests, and data values is used in virtually all microcomputers.

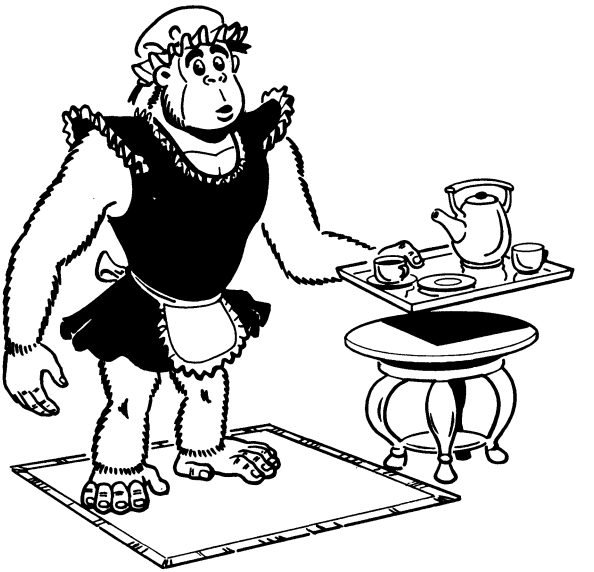
The big advantage of Von Neumann architecture is that it is extremely flexible. Instructions can even modify themselves, a process both dangerous and powerful at the same time. Certain locations can serve as data to parts of a program and as instructions to other parts of a program.

The disadvantages of the Von Neumann architecture are that it confuses beginning students and that instructions and data values that go with those instructions can need a variable number of bytes, depending on what that instruction is going to do and depending on the "How much?" or "With what?" or "Where?" data values that go with that instruction.

But, once again, this is simply the way people think.

Von Neumann architecture is used in micros since it most closely mimics the way people think and act.

Gorillas, too . . .



The next time you clean off a table after lunch, break down the task into the simplest steps you can possibly think of and repeat each step out loud as you do it.

Note in the following dialog how the actions, the data, and the tests are mixed together. Note also how you use them together one step at a time . . .

"WHAT IS THIS ITEM ON THE DINING ROOM TABLE?"
"It is a box of milk."
"DOES THE BOX OF MILK BELONG ON THE TABLE?"
"No."
"WHERE DOES THE BOX OF MILK BELONG?"
"In the refrigerator."
"WHERE IS THE REFRIGERATOR?"
"In the kitchen."
"I WILL GO TO THE KITCHEN REFRIGERATOR.
"IS THE REFRIGERATOR OPEN?"
"No."
"I WILL OPEN THE REFRIGERATOR."
"IS THERE ROOM FOR THE MILK?"
"Yes."
"I WILL PUT THE MILK IN THE REFRIGERATOR."
"I WILL CLOSE THE REFRIGERATOR."
"I WILL GO BACK TO THE DINING ROOM TABLE."
"WHAT IS THIS ITEM ON THE TABLE?"
"It is a shaker of salt."
"DOES THE SHAKER OF SALT BELONG ON TABLE?"
"Yes."
"IS THE SHAKER OF SALT IN ITS PROPER PLACE?"
"No."
"WHERE DOES THE SHAKER OF SALT BELONG?"
"In the table center with the pepper."
"I WILL PUT THE SALT IN THE TABLE CENTER.
"WHAT IS THIS ITEM ON THE TABLE?"

. . . and so on until you are finished.

Note several very important points. The first is that you break down a complicated task into single individual steps and then do each step one at a time in some reasonable order.

Microcomputers can also do only one simple thing at a time. Turns must be taken to let simple actions build up into results, and

then those results are built up one by one into tasks. Tasks are then completed one at a time to finish the job at hand . . .

Micros can do only one simple thing at a time.

Simple things are built up one by one to get a result.

Results are built up one by one to complete a task.

Tasks are completed one by one to finish the whole job.

The second point is that we mix actions, data values, and decisions as we go along. In this case "MILK" is a data value, "GO" is an action, and "DOES" starts a test.

If there was no milk on the table, chances are you could still clean up after lunch. The data values, such as "MILK," determine the exact actions that are to be done to complete the whole job.

So . . .

Micro instructions mix commands, data values, and decisions together in the order needed to get a result.

Results are then mixed together in the order needed to complete the entire job.

The third point is that you must keep exact track of where you are in a microcomputer program. Here is a very ridiculous example . . .

DOING IT:

If you were cleaning up a table, how would you handle . . .
An action called "MILK"?
An object called "DOES"?
A decision called "GO"?

You get into the same absurd situation if you don't keep track of *exactly* where you are in a program. Instead of picking up a valid op code, you may get some random data or qualifier value, and the program begins executing nonsense.

So . . .

For a program to work, you must know EXACTLY where to start and EXACTLY where to go to continue at all times.

The microcomputer has no way of second-guessing you. If you tell it something absurd to do, it will faithfully try to do it.

what does a program do?

The answer to that question is obviously that it depends on the program. But there are three main types of machine language programs. These are programs that get a *result,* programs that provide exact *time* sequences, and, finally, programs that *both* have to provide a result and simultaneously have to do it in an exact time frame . . .

TYPES OF MICRO PROGRAMS

( ) Programs that provide results.
( ) Programs that provide timing.
( ) Programs that must do both.

A calculation of some sort is a good example of a program where only the result matters. Say we need to calculate the square root of 273. What we are after here is a result, and it most likely won't matter much if it takes a fraction of a millisecond or a fraction of a second to do the job.

An algorithm, decision tree, or other set of rules usually helps in attacking any result-dependent program.

A traffic light or a pinball game are examples of programs where timing is most important. An ordered sequence of events spaced out in time is required in both cases. While we may be free to cal-

culate a square root along the way, it might not be needed, and any old way that does the timing for us will work.

A timing diagram is often the best choice for programs where time is of the essence.

The stickiest programs need an exact result that must take place in an exact time frame. One example is a microprocessor-controlled video display, where characters must be placed exactly on the screen in exactly the right time slot. Sometimes apparently simple programs end up taking too long to do by "obvious" programming methods, and these become the sticky type where both timing and results are important.

Almost always, the first attempt at any program will run far too slow. More often than not, some rethinking will get things up to acceptably fast speeds. Naturally, high level language freaks with their pitifully long execution times will blame everything on the hardware. Most of the time, though, all that is needed is some careful and creative use of machine language.

Another way to classify programs is by how they end . . .

MICRO PROGRAM ENDINGS

( ) Programs that go round and round till unplugged.
( ) Programs that return to some other program when they are finished.

Very few microprocessors are ever stopped. When a micro is doing something useful, it is *continuously* running some program. This may be a program such as a traffic light that cycles continuously or a program that does a calculation and then returns to a supervisory monitor or operating system.

Most interactive programs will spend much of their time waiting patiently for someone to press a key or stalling around for a printer to finish a page. Thus, most time on most interactive programs is spent simply waiting for something to happen.

If a program is to interact with people who have to make choices, that program should usually be *menu driven*. A menu driven program will have a master option selection list at its beginning. When one of these selections is picked, those actions needed for that selection are completed, and the program once again returns to the master menu for another selection. Even the exit, usually a "Q" for Quit option, should be included on this menu . . .

MENU DRIVEN PROGRAM—A program that starts with a master selection list and returns to that list after each task is complete.

Menu driven programs should be used for nearly all programs that are to interact with people.

Should the total job be horribly complex, each menu selection can lead to a sub menu, and if needed, each sub menu can lead to a sub-sub menu. An orderly process should exist to get back up one or more menu levels. At any time when a program finishes its task it should return to the upper menu level with everything back the way it was at the program start, so there are no suprises if menu options are picked in a strange order.

a "typical" machine language program

Most machine language programs start at some low address in the user RAM area and work their way up through user RAM, going from low addresses to high addresses. Remember that parts of RAM may be reserved for "system" uses and that your program should start somewhere near the bottom of the free area of user RAM. Use your simplified memory map as a guide.

Unless there is a good reason to do otherwise, the CPU reaches into the address space and gets an op code. It then evaluates that op code, and if it has to, gets another byte or two to answer "with what?" "where?" or "how much?" The CPU then goes on to the next available higher memory location and expects to find another op code there.

This process continues until the CPU gets some instruction that tells it that it is to go somewhere else to continue, rather than on to the next higher instruction. Examples of "somewhere else" commands are *jumps,* testing *branches, interrupts, breaks* used for debugging, and *subroutine* calls. More on these later.

The point here is that the CPU goes through memory in increasing order unless it is given an overwhelming reason not to. When the CPU gets to the end of the useful program at the upper end of its code, it will be told to jump back down to the beginning and start over again, or else it will return to a supervisory program, the monitor program, or an operating system.

Summarizing . . .

Machine language programs start at low memory addresses and work their way upward in sequential order.

One, two, or three bytes may be needed per instruction. The CPU will automatically "skip up" to the next legal instruction.

Jumps, branches, and a few other instructions can alter this normal low-to-high operation.

We can immediately see that it is extremely important for the microcomputer to know exactly where it is at all times. If the micro gets mixed up and thinks a "how much?" qualifier byte is really an op code, the whole program will bomb.

No, the micro won't stop. What happens is that some totally weird program starts executing, most likely plowing up everything of value that you put into the machine. This wayward program then continues until disaster strikes one way or another.

Here's how to prevent disasters . . .

TO AVOID PROGRAM BLOWUPS . . .

- ( ) **You must always have EXACTLY the program in the machine that you think you do.**
- ( ) **You must always EXACTLY initialize any program values and EXACTLY configure any hardware before you use either.**
- ( ) **You must always know EXACTLY where you are in any program.**
- ( ) **You must always start EXACTLY at the intended starting address of any program.**
- ( ) **Your jumps and branches must go EXACTLY where you intend them to.**
- ( ) **Your program space must be EXACTLY protected from incursion from any other use, including self-destruction.**

Fail to obey any of these rules and your program will bomb. And after it bombs, it will probably destroy all sorts of obscure locations in memory that you may wrongly continue to assume are okay.

Another rule . . .

> **If a program bombs, do not only fix the cause of the trouble.**
>
> **Always go back through the ENTIRE program and the ENTIRE data block area and make sure that no other damage was done by the bombing.**

We will find·out just how to do all these things as we go along. The important point here is that you have to pay extremely close . . .

Well, we have put it off as long as we can. Looks like we now have to turn to . . .

THOSE #$!$# CARDS

There is only one sure way to understand a microprocessor thoroughly and completely. And that way is actually to use each and every instruction in that micro's instruction set. Do this first in short and simple discovery modules and then later in actual programs.

This is what the "those #$!$# cards" method is all about. What you do is get yourself a thick pack of unruled 3 × 5 inch cards, all one color, for the micro of your choice. You complete one card for each and every instruction for each and every address mode. That translates to some 121 cards for the older versions of the 6502 and the better part of a thousand for the Z-80.

Each card should be simple and obvious to use but must have more detail than you could normally get off a pocket card. You should pick up each instruction as the need arises rather than trying to do things in alphabetical or some other order.

It is extremely important that you do each and every card by yourself and *by hand!* It is the action of creating the cards that forces you to relate to what an instruction is and how the microcomputer can use that instruction.

The "those #$!$# cards" method is very simple . . .

THE "THOSE #$!$# CARDS" METHOD

( ) Create a card for each and every instruction in each and every available address mode for the micro of your choice.
( ) These cards must concisely show you how and why each instruction can be used.
( ) You must do these cards in an "as needed" order, going from simple to complex.
( ) You MUST do each and every card BY YOURSELF and BY HAND!

We'll shortly look at what goes on a typical card and how to get started. We are going to look at a series of nine *"discovery modules."* Each of these modules will do something useful and interesting by itself. But more important, each module will build on what has gone before to let you fully explore the micro of your choice.

Hidden in each discovery module are some new techniques and new skills you will have to pick up before you can progress to the next module.

Here are the names of the modules . . .

THE "DISCOVERY" MODULES

(1) Tail Byter
(2) Figure Eight
(3) Square Deal
(4) Audio Tone
(5) Pitch Reference
(6) ".Y" Time Delay
(7) Nite Lite
(8) Text Outenblatter
(9) Burglar Interrupt

Note that none of these discovery modules are true programs. Sure, each one does some "gee whiz" thing using your trainer or personal computer. But real programs involve much, much more than just punching code into a micro and watching it react. We will save the actual real program writing for the Micro Applications Attack of Chapter 9.

Since it is super important that you do everything yourself on your own trainer, I am not going to give you code that will immediately run on any particular trainer. This way, you will have to think about what you are doing to get any results rather than just punching in some code you may not fully understand.

Canned code is worse than useless for learning micros. Avoid it!

So let's invent a fake "discovery trainer." Call it the MYTH-1 . . .

THE MYTH-1 DISCOVERY TRAINER

So that you will be forced to rewrite all code for the micro of your choice, we will use a fictional, MYTH-1 trainer for the detailed examples.

This fictional trainer uses the 6502 as the CPU.

The available 1K User RAM starts at location $2000.

There is one 8-Bit VIA style parallel port available. The "teach port" address is $C080. The actual port address is $C081.

Text is output to a printer or TTY by a subroutine at $F90D.

The keypad is read to the accumulator by a subroutine at $F67C.

NMI, RESET, and IRQ vectors are stashed at $02FA through $02FF.

The monitor includes a single-step feature, an LED display, and little else beyond the bare essentials.

As you go through the "those #$!$# cards" method, you may find many differences between your micro and the 6502 coding. Obviously, all op codes will be different for a microprocessor from a different school. You may also find that the index registers common to the 6502 and 6800 families have to be replaced with the indirect registers common to the 8080 school.

As this happens, be sure to look for "classes" of instructions that will do the same task our example does or that can solve a similar problem by a somewhat different method. You may also have to emphasize and explore some powerful instructions on your own.

But we are mainly interested in method. The "those #$!$# cards" method will work on ANY microprocessor from ANY family, simply by starting with the discovery modules and building on them.

Before we get started, though, we need to pick up some simple details on . . .

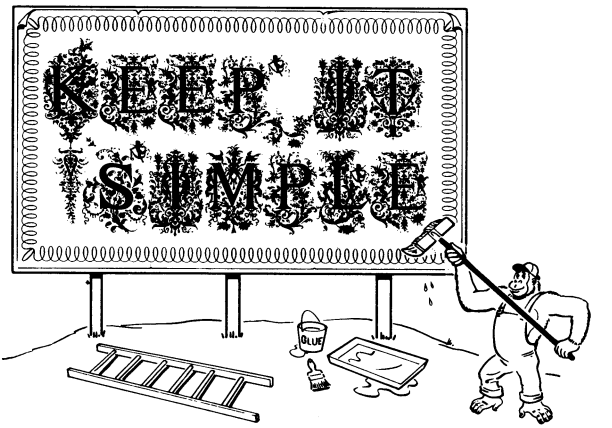
flowcharting

A flowchart is a special diagram that shows what program actions happen in what sequence. Again . . .

> **FLOWCHART**—A special diagram that shows what micro program actions happen in what sequence.

Old line dino people tend to go way overboard on flowcharting and use dozens of weird symbols and all sorts of nonsense conventions. They also spend far too much time drawing flowcharts instead of thinking creatively about what the program is really supposed to be doing.

The first and foremost rule of flowcharting is . . .



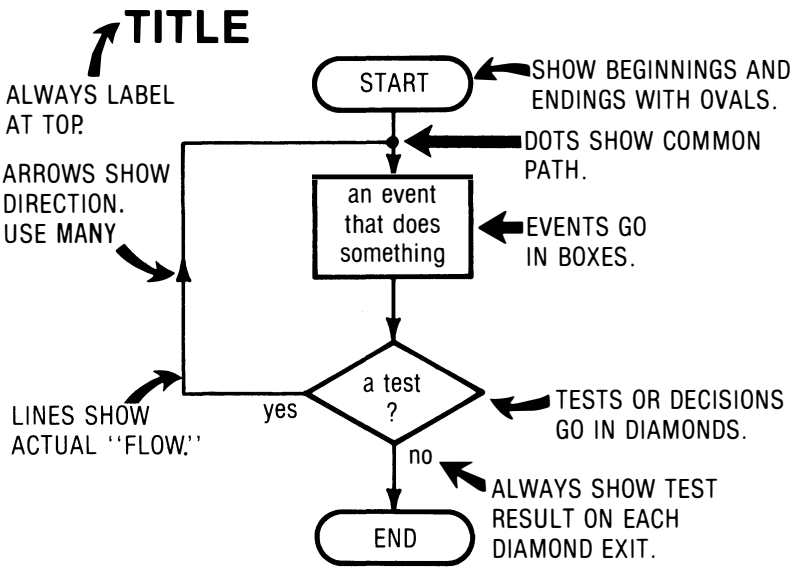
Sometimes you want to split up your flowchart into a "main" flowchart and separate "detail" flowcharts. These are otherwise known as . . .

BIG LUMPS FLOWCHART—Shows only main events in the simplest possible way.

CRUMBS FLOWCHART—Shows each and every micro action as an expanded detail of a single block on the big lumps flowchart.

You should avoid fancy flowcharts with lots of blocks and lines crossing each other. Use only the amount of detail needed to serve as an outline.

Here is a simplified flowchart showing the only symbols we need or should use . . .



The start or the finish of a flowchart is shown by an oval. Name this oval START, or ZORCH SUBROUTINE, or whatever. Any event

that does something goes into a rectangular box. Any decision that needs a "yes" or a "no" answer goes in a diamond-shaped box. If there are more than three possible answers to a test, use a stack of diamond-shaped boxes.

Lines with arrows connect as many event boxes as you need to as many decision diamonds as you use. Be sure to use lots of arrows so you leave no doubt about which direction the "flow" of your flowchart is. Always label and title your flowcharts. Include version and date.

Generally, you flow from top to bottom and from left to right. Detail is set aside elsewhere. If there are two or more paths to an event, these paths are joined with a dot above the event. Always enter an event at the top and exit at the bottom. Always enter a diamond from the top and exit right or left or down, depending on whether or not the decision leaves you in the mainstream. All diamond exits *must* be labeled. Use "yes" for the YES line, "no" for the NO line, or whatever you need.

Avoid crossing lines and keep nested loops one inside the other. Try to keep the "mainstream" flow straight downward, with "side trips" just that. Aim for a whole flowchart connected and in one piece. Do not use little numbered or lettered circles to show "splices."

As I mentioned before, avoid using a flowchart template, since there is too much useless garbage on it. Use a logic template instead. There are also flowcharting aids such as faint-blue printed master sheets, overlays, and mylar stick-ons. None of these should be needed here, although they do neaten and simplify documentation.

One thing that is essential, though, is quadrille or graph paper rather than lined or blank paper. This is a must for just about all micro work.

'Nuff said on flowcharts. Their use should become obvious as we go along. Let's see what goes on . . .

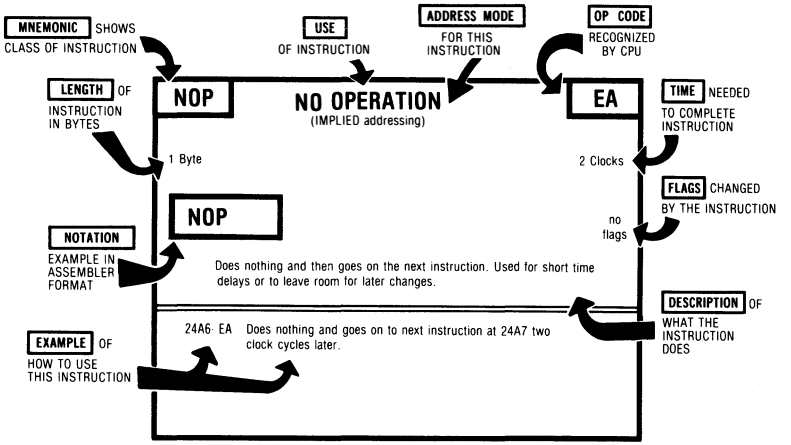
a typical card

For a given microcomputer family, you will want one card for each command in each and every address mode.

As a review, an op code is usually a 1-byte command that starts a course of action for the microprocessor's CPU. In the case of an obvious, or implied, addressing mode, that one byte is often all that is needed to complete the task. Fancier addressing modes will need a second, a third, or further bytes to qualify the original op code and answer such questions as "with what?" "how much?" "from where?" or "to where?"

Our first two cards will look at two instructions available on every microprocessor. One of these is an implied instruction, called "no operation" or NOP, while the second is an absolute instruction called "jump" or JMP.

Let's use NOP as an example of what we want to put on a card. Here is an annotated NOP card . . .



Let's first zero in on what is supposed to go on the card for any instruction. Then we'll find out what a NOP really is.

Starting in the upper left, we have the mnemonic. In this case, it's a "NOP." This should be in larger, bolder print, and shows us the *class* of instructions this particular one belongs to. I like to have only the mnemonic here. We'll pick the address mode and show the assembler notation elsewhere on the card. Keep abstract symbols to a minimum, since they often can confuse newcomers.

There can be lots of different cards for each mnemonic. You should end up with one card for each addressing mode that is available for that command.

A very brief description of what the instruction does should be centered on top of the card, again larger and bold. Try to provide more detail than the pocket card does. For instance, on the upcoming LDA or load the accumulator cards, use "PUT VALUE IN ACCUMULATOR" or "FILL ACCUMULATOR FROM PAGE ZERO." In other words, give a hint which addressing mode is used and what the intended use for the instruction is.

The op code goes at the upper right, also bold and larger. The op code is the only way the micro's CPU has of identifying which par-

ticular instruction this is to be. Op codes of related instructions are sometimes similar. Thus, on the 6502, three of the LDAs are coded as A9, A5, and AD to separate the immediate, page zero (absolute short), and absolute long modes. Each of these has a hex $A or "1010" code for the top four bits in its op code.

The addressing mode is centered under the instruction title. This tells us how we are going to get our package to or from Albuquerque. Very often, a single mnemonic will be able to do a bunch of different things in different addressing modes, as was the case with the LDA above. You should end up with one card for each mode of each instruction.

The number of 8-bit bytes needed for the instruction goes below the mnemonic. An implied addressing instruction usually needs only the mnemonic, while most other instruction modes need additional 8-bit bytes to answer "With what?" "Where to?" or "How far?"

How many CPU clock cycles are needed to complete the instruction? The answer to this appears under the op code. This value tells us how long each instruction will take to execute. This will turn out to be important later when we look at timing. Instruction times are called out in *CPU cycles*. The actual time to do a CPU cycle varies with the microcomputer, so you multiply the number of CPU cycles by the time per cycle to get the total execution time. Very often, 6502 micros will have a CPU cycle about one microsecond in length.

Note that it is real easy to mix up the cycle and byte numbers off the pocket cards. Be sure you get it right. On the 6502, a good memory jogger is the RTS or "ReTurn from Subroutine" command. This implied instruction takes one byte but takes a long time (six cycles) to complete.

Sometimes an instruction may need a variable number of clock cycles. On the 6502, any of the branches will need two clock cycles if the branch is *not* taken and three if it is. Another clock cycle may be needed if a page boundary is crossed. On other micros, the number of clock cycles can get incredibly complex, particularly on early devices in the 8080 school.

The box below the number of instruction bytes shows the notation we will need later on when we use an assembler or Editor and Assembler combination. The assembler picks an address mode by seeing what comes after the mnemonic. More precisely, the assembler demands an *op code* followed by an *operand*. The op code tells us what to do, and the operand tells us the address mode and answers the usual "where?" "with what?" or "how far?" questions.

Like so . . .

OP-CODE MNEMONIC—A three- or four-letter combination that tells us the exact instruction a CPU is to carry out.

OPERAND—A "modifier" that follows the op-code mnemonic to tell us "where?" "with what?" "how far?" or other addressing information.

The exact symbols needed for an operand vary with the microprocessor, as well as with the particular assembler or editor in use. Very often, though, the dollar sign is used to identify a hex value. The "number," "sharp," or "#" sign indicates an immediate value. Parentheses often mean an indirect address, where you go to the address in the parenthesis to get the address you are really after. If you are using register indirect addressing on the 8080 school, then an "I" will often get tacked onto the end of the mnemonic to show an indirect command. Other micro schools use "X" or "@" to show indirect addressing.

The number of hex digits in the address tells us the difference between absolute short and absolute long addressing. Commas will often tell us that indexed addressing is in use. What follows the comma is usually the index register. If you have a fancy enough assembler, quotes often mean to enter what follows as ASCII coded data values. This is handy to build files and data stashes that are to be used by your machine instructions.

Typical operand symbols are . . .

TYPICAL OPERAND SYMBOLS

$	- means a hexadecimal value
%	- means a binary value
	- (nothing) is in decimal
"	- means an ASCII coded string
#	- means an immediate value
( )	- means indirect addressing
,	- (comma) means indexed addressing

2 hex digits = absolute short
4 hex digits = absolute long

Once again, you should not use assembly language till well after you have done bunches of machine language programming. But it is important to get the notation right the first time, so that later on the

assembler will understand what you are asking it to do. The "punctuation" turns out to be *extremely* critical, so watch these details *very* carefully. You also have to be sure the notation you use is right for your choice of microprocessor and assembler.

Returning to our example card, the notation box should show a real example, rather than something obscure like "LDA $nnnn, R(k)." Use concrete examples and not vague symbols.

At the center right of our sample card, we see a description telling which flags are affected by this instruction. We will look at flags in depth later. For now, note that there are a handful of flags in most micros. Some instructions will affect certain flags and will not affect others. The flags can then be used to simplify tests and decisions, set modes, pick routes, or otherwise ease programming.

We then put in a simple description of what the instruction does and how it does it. Do this in plain old English. Do not, under any circumstances, use those stupid and totally undecipherable symbols seen in the micro data books.

Tell it like it is.

A line should separate the top and bottom halves of the card. Put one or two use examples below this line. Make up some legal starting address in user RAM and show the instruction doing something useful. Then explain, again in English, what is happening.

The first example should show the most obvious and simplest use of the instruction. A second example may be added to show some subtle point or bring out some use detail that may not be obvious. If there is some real gotcha, you can show this here. You can even continue special details on the back of the card, but don't if you can help it. You can also show simple diagrams on the bottom of the card. Rotate and Shift instructions are much easier to show than to describe.

Once again, the purpose of the cards is to force you to relate one-on-one to the instructions of the microcomputer of your choice. You should include enough, and only enough, information on the card to let you fully understand what the instruction does. Be absolutely sure you do each and every card *by yourself* and *by hand,* because creating the card is of far more value to you than actually using the card will be later.

As you complete a group of cards, use each and every card in an actual discovery module or other short program to make sure the instruction behaves exactly as you think it does. Then ask yourself what you can *really* do with this instruction that isn't obvious at first. The real power and real insight in micros happens when you start using instructions in new and creative ways.

Let's start building up our cards. We'll combine a micro topic or two with several new cards and then build on what we have to do in a discovery module.

We'll start out with . . .

NOP and JMP

Two very simple op codes and cards are the "do nothing" NOP and a "go somewhere else" command called JMP absolute. We will use these to get us started on our first discovery modules.

An instruction to do nothing at all sounds like a total waste. But this instruction, called a NOP (short for *No Operation*) ends up having lots of interesting and powerful uses in the micro world.

NOPs take up a minimum amount of time and then go on to the next instruction. One important use for NOPs is to purposely burn up machine clock cycles to provide an exact amount of time delay. This may be needed for time-critical programs or modules.

A second use for NOPs is to save room for some added complications later on. For instance, if you later need a subroutine call but don't want to get into it just yet, you put in three dummy NOPs for now and fire away. Later on, you can easily replace these with the real code you need.

NOPs are also a great debugging tool. If a program doesn't work, you often can patch over parts of the code with NOPs and see what goes away. Replacing one part at a time can often point to the problem area.

Let's look at the NOP card again . . .

NOP	NO OPERATION	EA
	(IMPLIED addressing)	
1 Byte		2 Clocks
NOP		no flags

Does nothing and then goes on to the next instruction. Used for short time delays or to leave room for later changes.

24A6– EA	Does nothing and goes on to next instruction at 24A7 two clock cycles later. No flags are changed.

NOP is an implied address mode instruction since we need no further information to complete the action. The assembler notation is just the NOP op code itself, since no operand or qualifier is in use.

A NOP takes up two CPU clock cycles in the 6502. We will shortly see that a CPU clock cycle is typically one microsecond on many 6502 trainers, with the Apple II running very slightly faster. So, one NOP delays two microseconds for us in most 6502 trainers. For a delay of four microseconds, you could use two NOP commands together. Six microseconds is done with three NOP commands, and so on. But we will find there are far better ways to gain long time delays than by using scads of NOPs.

Some other instructions will also behave as NOPs under certain circumstances. Many of the upcoming "test-and-branch" instructions default to a NOP if the test fails. On other microprocessors, a command, "move something to itself," such as an 8080 MOVAA or a 6800 school "branch never" command, also serve as no operations.

The NOP is admittedly not the most exciting of all microcomputer instructions. But we have to start somewhere, and there is a surprisingly large variety of timing, room-saving, and debug uses for this command.

Remember that a microcomputer starts at some low numbered memory location and works its way up through the instructions. It does this in ascending order, using up the one, two, or three bytes as needed to carry out a command. As long as the code is correct and as long as the commands do not tell us to do otherwise, the CPU will have its program counter work its way up through memory, usually in sequential order. The place in memory we use can be ROM for things like system monitors, operating systems, and often-used subroutine utilities. For changeable programs, user RAM is the place where the action takes place.

We may need a way to get from the "finish," or high end of the program, back to the "start," or low end of the program. Instructions called *branches* or *jumps* force the CPU to go somewhere else . . .

JUMP—A command that tells the CPU to go somewhere else to continue the program.
On the 6502, jumps are always unconditional.

BRANCH—A command that tells the CPU to go somewhere else to continue the program.
On the 6502, branches always involve a test.

In general, branches and jumps do pretty much the same thing. Either puts you somewhere else in a program. But the individual names may have very different and very special meanings for a specific micro family. For instance, on a 6502, the absolute jump commands do their thing regardless, or *unconditionally,* while the relative branch commands make a test and then may or may not go somewhere else, depending on whether the test *conditionally* passed or failed . . .

UNCONDITIONAL—A micro instruction that does its thing regardless of any thing else happening in the machine.

CONDITIONAL—A micro instruction that will make a test and then may or may not do its thing, depending on whether that test passed or failed.

IT MAY DEPEND...

There are two jump commands on the older 6502—the JMP absolute and the JMP indirect.

Here's the card for JMP absolute . . .

JMP	**JUMP SOMEWHERE ELSE**	**4C**
	(ABSOLUTE LONG addressing)	
3 Bytes		3 Clocks
JMP $25A6		no flags

Jumps unconditionally to a new position, whose position is shown by the second byte and whose page is shown by the third byte. Used to go somewhere else with no strings attached.

202B– 4C 06 A2 Jumps unconditionally to location $A206.
NOTE: SECOND Byte = position on page
THIRD Byte = page

This JMP command uses the absolute long addressing mode. We cannot simply say "jump." We must qualify that jump command with a "where to?" operand. On the 6502's absolute long addressing, you need a full 16-bit wide address, which means that it will take two additional 8-bit words to answer our "where?" question. In the 6502, as in most micro schools, the address is spelled out backward in the op code, with the low byte or position on the page as the second instruction byte and the high byte or page as the third instruction byte.

This backward addressing has several advantages. One is called *pipelining* and lets the CPU set up some of what it has to do ahead of time to speed up operations. Another advantage is that operations for absolute short and absolute long addressing start off the same way and proceed in the same direction.

When the CPU gets to a location where it expects to find a valid op code and it sees a "4C," it immediately whips out its own pocket card and decides it must do an absolute jump. After that, it looks in the next slot to find an address low value. It then looks into the next slot after that one to find the address high value. It then puts both values together in the correct order and jumps to that address.

Naturally, there must be something useful and interesting at the location that the microprocessor jumps to. Unless there is a valid op code in this location, the program will bomb.

One obvious use of absolute long jumps is to get from the end of the program back to the beginning. Another use is to combine different paths through a program into one common continuing path. Absolute long jumps can reach anywhere in the entire address space, so they can also be used to "amplify" a branch or other short jump to reach anywhere in the entire address space.

Another interesting use of a jump command is to make a *trap* . . .

TRAP—A jump or branch that goes to itself, hanging the program.

Now, obviously, you never want to leave a trap in a final program. But the trap is a powerful debugging tool that lets you do the first part of a program and see any results. If you have program problems, just add traps as needed to separate what is working from what is not.

I'll let you do the JMP indirect card on your own. Its a toughie to start with. Some hints. The 6502 JMP indirect, or 6C op code, finds an address low as its second byte and an address high as its third byte. It then goes to this address to find the *real* address low or position byte. It then goes one farther to the indicated address plus one to find the *real* address high or page byte. Finally, it goes to that "real" address.

Do it . . .

DOING IT:

If your trainer is from the 6502 school, complete the NOP, JMP absolute, and JMP indirect cards at this time.

If not, complete all cards for all do-nothing instructions and for all absolute jumps and all indirect jumps.

The jump indirect mnemonic often includes parentheses, such as JMP ($FDAC), but the notation changes from micro family to family. In this case, the microcomputer finds the address low from the "real" address value stored in FDAC and then looks in FDAD to find the "real" address value high. It then jumps to those new locations. Note that *five* bytes are involved. There are three bytes in the program module that include the jump indirect op code, the low address address, and the high address address. There are also two data block bytes stored elsewhere that actually hold the low and high bytes of the address we are to go to.

Indirect unconditional jumps are useful when you need to calculate an address, such as continuing with the options on a menu selection. They are also useful to let you put most of your program into ROM, while letting you calculate or otherwise change a pair of RAM address values as part of your fixed program.

A final important reason to master indirect addressing on the 6502 is that you must understand indirect addressing before you can go on to the super whiz-bang address modes that combine indirect, indexed, and absolute short or page zero addressing.

If your micro does not have an absolute indirect jump, it should have some other way to do the same thing, such one or more registers that can be used as an address pointer. Check it out.

We can now try out our first two cards on an actual discovery module . . .

DISCOVERY MODULE 1

TAIL BYTER

Write a program that does nothing three times and then repeats forever.

List, single step, and then run the program.

Remember, the discovery modules are NOT programs. Programs that solve any useful real-world problems take a very involved and long solution method called the *Micro Applications Attack* that we will see in Chapter 9. The discovery modules will run like a program, but we need only go from flowchart to code sheet to program.

Use this simplified attack . . .

TO DO A DISCOVERY MODULE—

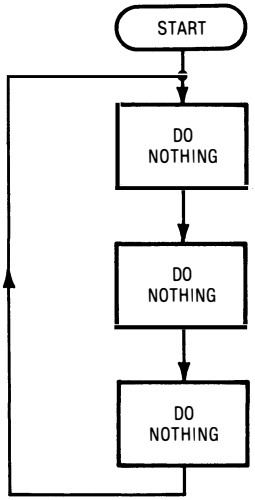
1. **Do a flowchart.**
2. **Do a coding form.**
3. **Enter the code.**
4. **Debug the code.**

Each discovery module will have all sorts of nasties fiendishly hidden in it. These include new skills you have to pick up or new things to learn. Our first discovery module requires you to learn how to use the machine language programming form and the hex dump form, how to bring up the micro, how to read and write to memory, how to list, how to single step, how to debug, and how to actually run a program on the trainer of your choice.

So, if you haven't already done so, get out your trainer, bring it up, and practice reading and writing memory locations. Use your simplified memory map to pick a good program starting point. Try writing to a ROM location. What happens? Why?

Here's the flowchart for the first discovery module . . .

TAIL BYTER:



We see that we want to have three events called "do nothing" in a row. Then we have to go back to the start and repeat. While we will use a jump to get back to start, we don't have to show this as an event. The return line, the arrows, and the path-combining dot at the top imply this jump for us.

Let's now get a machine language programming form and look at it . . .

ADDRESS	OP CODE	BYTE #2	BYTE #3	MNEMONIC	HOW?	NOTES

Starting at the left, we see a column that holds the address of each new instruction. The op code for that instruction goes next to the right. You use one line per instruction but only one box per byte. If you have a two-byte instruction, you put the op code in the first box and the second "how?" or "with what?" byte in the second

box. If you need three bytes, the third "where?" byte goes in the third box.

For instance, if your user RAM starts at $2000, you may need location $2000 for an implied instruction, or locations $2000 and $2001 for an immediate instruction, or locations $2000, $2001, and $2002 for an absolute instruction. The next available location for the next instruction on the sheet will likewise be bumped forward as far as needed.

The op-code column holds the mnemonic for the command. The HOW? column holds the operand if one is needed. The final column to the right holds notes. These notes, or *comments,* are people-type words in ordinary English that say the same thing the words in the flowchart boxes do. Avoid using any mnemonics or other computereze in this column.

Always do your programming forms from RIGHT to LEFT! Start with the people-type notes. Then figure out the op-code mnemonics and the "how?" operands needed to complete those commands. The answer to "how?" might instead answer "where?" or "with what?" Then go all the way to the left and calculate the next available address. You calculate the next available address by counting *filled* boxes from the previous address. Finally, use your cards to fill in the actual code values.

One more time.

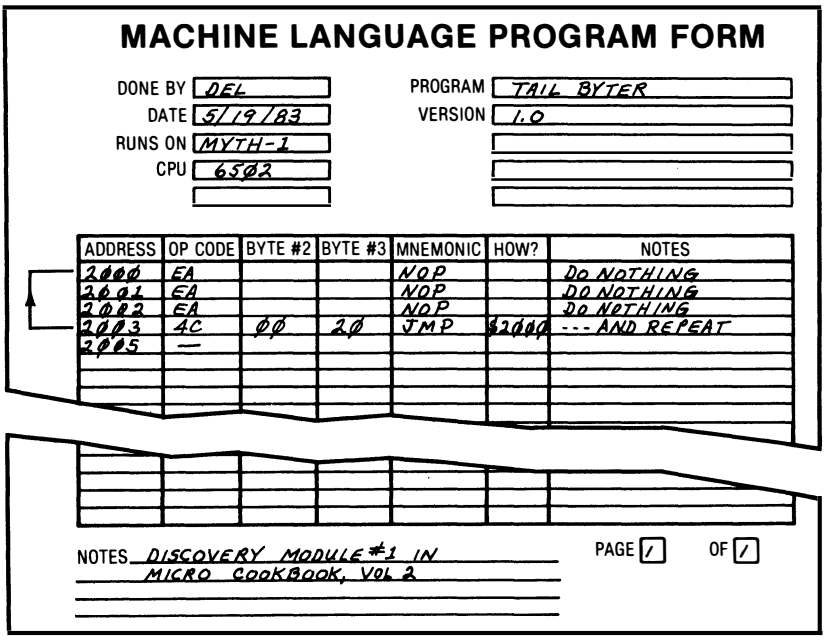
The action is on the right! It's those people-type notes or comments, followed by the assembly level mnemonic and operand, that you should spend most of your time on. Do NOT, repeat DO NOT, fill in the code values first! Always work from right to left.

Some rules . . .

PROGRAM FORM RULES

( ) Always work from RIGHT to LEFT!
( ) Keep comments in English, not computereze. The note, or comments, "field" is far and away the most important.
( ) Always use ONE line for ONE op-code command.
( ) Use the second and third byte boxes ONLY when the command needs a second or third byte.
( ) Calculate new addresses by counting one for each FILLED IN box per address space location.

Let's go through our first program form, filling in some detail. Here is what the final form will look like for our MYTH-1 imaginary trainer . . .



MACHINE LANGUAGE PROGRAM FORM

DONE BY: DEL
DATE: 5/19/83
RUNS ON: MYTH-1
CPU: 6502

PROGRAM: TAIL BYTER
VERSION: 1.0

ADDRESS	OP CODE	BYTE #2	BYTE #3	MNEMONIC	HOW?	NOTES
2000	EA			NOP		DO NOTHING
2001	EA			NOP		DO NOTHING
2002	EA			NOP		DO NOTHING
2003	4C	00	20	JMP	$2000	--- AND REPEAT
2005	—					

NOTES: DISCOVERY MODULE #1 IN MICRO COOKBOOK, VOL 2

PAGE 1 OF 1

This module is rather obvious. Three NOPs and a jump back to repeat over and over again. But let's go through how you build up your programming form, step by step.

The first thing to do is decide where you want your program to start. You can find this from your Simplified Memory Map or otherwise determine where your program is to go. Naturally, the first entry of a program has to be in a user RAM area that won't get plowed by something else. Much later on, you may want to permanently burn your program into EPROM, but even then you will probably need some RAM for things that are to change.

A lot of disk operating systems assume the first program location is in fact the start of a program. So it's a good idea to put your program or discovery module first into RAM, followed by any support subroutines, data blocks, files, stashes, or whatever.

We will start our program at $2000, the start of the MYTH-1's user RAM.

Now, we go to the flowchart and get the "do nothing" comment off the flowchart and put it on the programming form. We next decide that a NOP is a good mnemonic to do nothing, so we put this in the top slot of the MNEMONIC column. A glance at our NOP card tells us this is an implied instruction that needs no qualifications, so we leave the operand or HOW? column blank.

We then calculate the instruction address. In this instance, this step was already done when we decided where we wanted to start. Most of the time, though, we find the instruction address by looking at the *previous* instruction address and *adding* one, two, or three locations to it, depending on whether the previous command took one, two, or three bytes.

You only put in the machine code after everything else is filled in on the line. *You must have your note, mnemonic,* HOW? *and address fields complete before you look up any code.* You must also be sure you know *exactly* which address mode you are using at this point. The symbols and punctuation for your address mode also must be *exactly* correct.

Then, and only then, can you go to your cards and fill in the EA code needed. (Naturally, if your trainer is not from the 6502 school, your NOP will have a wildly different op code, rather than EA.)

The process continues in the same way for the second and third lines The only difference is in how we calculate our address. To calculate a new address, add the number of bytes of instruction code of the previous address to the previous op-code address. It's simple this time. $2000 plus one byte is $2001, the start of the second instruction. $2001 plus one byte is $2002, the start of the third instruction. *Don't forget that* $0A *comes after* $09 *and that* $10 *follows* $0F!

It's best to call out addresses out loud, "counting blocks" as you go on. This avoids any confusion on multiple byte addresses.

Note that the address mode can *change* how many bytes are needed for an instruction. Be sure you get the right mode with the needed number of bytes, because if you don't, the program will bomb by mixing up "how?" "where?" or "what?" bytes with valid op codes.

Our final line note says to do it again, and we decide an absolute jump is the way to go. Our mnemonic is JMP. In this case, the mnemonic by itself won't tell us all we need to know, so we put the address we want to jump to, $2000, in the operand, or HOW? column. Our address calculates just like before, since the previous instruction was one byte long.

On to the 6502 JMP card and the actual coding. We need a "4C" op code to do an absolute jump. This has to be followed by the address low or position byte, which in this case is $00. The third byte is the address high or page byte and is a $20. Note that the addresses appear in byte reverse order *only in the code blocks.* Addresses are shown frontward on the operand column and in any documentation.

On your trainer, there will be a different starting address and the JMP values will be different. The op codes will, of course, also be

different if you are not using a 6502 family microprocessor. Be sure to think about what you are putting on the sheet. Make sure all addresses relate to what you are trying to do and make sure you have EXACTLY the addressing mode you chose.

Note the arrow on the left margin. Always show program flow this way in a hand-coded machine language programming form.

I have shown you most of the program form in this example. It is very important to fill out ALL of the boxes on the form completely. Otherwise, old versions of nonworking programs will get mixed up with the good stuff, and you will be in deep trouble. ANY time you write down ANYTHING involving micros, always show the date, who you are, the program name, the version number, the CPU, and what machine it runs on.

In later examples, we will omit the boilerplate from the forms to save room. But remember that it is ABSOLUTELY essential that EVERY sheet of anything you do with micros is identified so well that you or anyone else can immediately tell what it is a year from now.

You can now punch the code into the machine. After you enter the code, you should dump it or list it to make sure that what you actually put into the machine is what you thought you put into the machine. *Listing* or *dumping* just gives you a location-by-location check to make sure the machine is filled the way you want it.

Still, the terms mean different things . . .

DUMPING—Viewing or printing the hex values in sequential memory locations.

Dumping works any place and any time on any contents.

LISTING—Viewing or printing values in sequential memory locations that are taken apart, or disassembled, op code by op code and presented in mnemonic form.

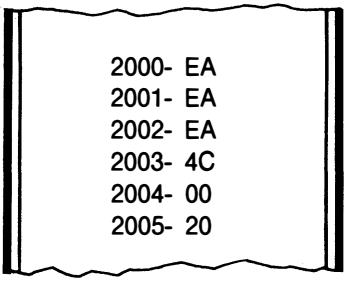
Listing, or disassembly, ONLY works on legal, correct program code and then ONLY works if it starts at the EXACT beginning of a a legal instruction.

If you get bunches of question marks during a listing, you either are trying to list something that is not a program or discovery module, or you have the wrong starting point, or you have mixed up

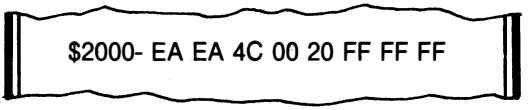
your address modes or otherwise dropped or picked up a byte somewhere.

All trainers let you dump memory. The fancy ones will also let you list. Lising is easier to read and simpler to use.

Here is how your program might dump onto a trainer's LED display . . .



Here is what a hex dump would look like on a personal computer's video display, if it is set up to show eight values per line . . .



The last three entries could be any old value, since we aren't using them and since they aren't needed by the program. Your hex dump form could be used to show eight or sixteen values per line. This is how the hex dump form looks, set up for sixteen values per line . . .

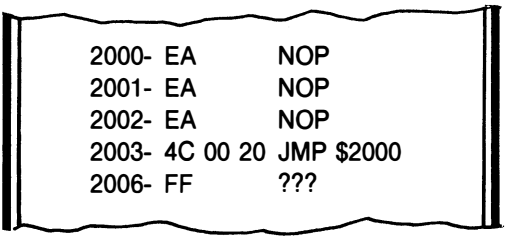
LINE ADDRESS	LINE ADDRESS PLUS															
	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F
2000	EA	EA	EA	4C	00	20										
2010																

Note that each hex value goes to the right of the next one, with the "line address plus zero" first on the left, the "line address plus one" next, and so on for sixteen values. The start of the next line will be address $2010, since $2000 + $10 = $2010.

Hex dump forms are used mostly to store data and file values or to hold complete programs in ready-to-enter form. The advantages of hex dumps are that they are compact and can hold anything.

Their disadvantage is that you can't easily tell what you have by looking at it.

Your program form should give you a listing of your program when you look at just the mnemonic and the "how?" columns. Here is an example of the program's listing, as seen on the screen of a personal computer . . .



The $2006 listing could be anything, and all sorts of meaningless garbage could follow the legal part of your program. Note there is no listing for the "00" value at $2004 and the "20" at $2005, since these are modifier, or operand, values that are an essential part of the instruction starting at $2003. Listing lists ONLY op codes. And, once again, listing works only on a legal program and only when you start at the beginning of a legal instruction. Any other listing gives you garbage.

Listing is sometimes called *disassembly,* since listing involves taking apart the code and breaking it down into individual mnemonics and modifying operands.

More buzzwords . . .

ASSEMBLY—Putting together a program by fitting the op codes and operands together in proper order to form a string of hex bytes.

DISASSEMBLY—Taking apart a string of hex bytes and trying to convert them into a legal sequence of op codes and operands.

Disassembly will work ONLY on a valid program and ONLY when begun from a legal starting point.

After you have listed or dumped your program to make sure you have it entered correctly, you should be ready to *debug* it. Debugging is any way of checking out a program to find problems . . .

DEBUGGING—Any method of checking out a program to find problems.

Common debugging techniques include listing, single stepping, trapping, isolation, tracing, falling back, and using breakpoints.

We will pick up on these debugging methods as we go along. *Listing* or *dumping,* of course, is the way to be sure that what you think is in the machine is really there. *Single stepping* lets another program run one line of your target program at a time. *Trapping* stops the program and hangs it at a selected point. *Isolation* is any way to decide what part of the program is causing the problem. Patches, NOPs, immediate subroutine returns, and other stunts can be used to change the program to isolate the part of it that is the culprit. *Tracing* is repeated single stepping at a slow speed or to generate a printed record. *Falling back* means going back to the last working version of the program and trying again. If you are on your first try, falling back consists of trying something simpler, taking smaller first steps, or trying a different tack. *Breakpoints* are forced software interrupts that jump you from your program to the monitor or some special code.

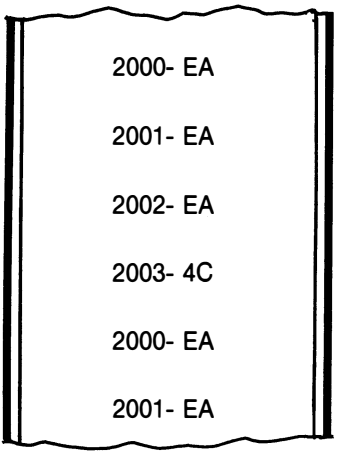
Most trainers have a single step feature. This is turned on by flipping a switch or pressing a special button. The Apple II has a single stepper built into its "old" reset ROM. Every time you press the S key, you get one line acted on and also get a printout or screen display of what all the working registers are up to.

But note that single stepping does NOT run your program. Single stepping runs a program sequence in the monitor that borrows one program line of yours at a time. To repeat . . .

SINGLE STEPPING—Running a monitor or operating system program that takes your target program and runs one line of it at a time, showing you results as you go along.

When you single step, you are running a monitor program. You are NOT running your target program.

As you single step your program on our MYTH-1 trainer, you should get the following addresses on the display . . .



. . . and round and round she goes.

If you have a fancy single stepper, you may get all sorts of other interesting stuff as well, such as mnemonics, operands, working register values, and so on. But the thing to watch is the addresses. If the addresses go round and round, rather than hopping off into some other address space, chances are your program is ready to run.

Note the difference between dumping and single stepping. In dumping, you check to see what is in each and every memory location. In single stepping, you actually execute one op code at a time, using up the one, two, or three bytes as needed to complete each command. Dumping *shows* you what is there. Single stepping actually *does* each command.

DOING IT:

If you single stepped the TAIL BYTER and got this sequence, what beginner's coding error did you make and how can it be fixed?

2000- EA
2001- EA
2002- EA
2003- 4C
0020- 9F

The value at address $0020 could be almost anything else, as well. Do you see the problem?

Oh yes. One very important point before we go on . . .

Any time you find a mistake in a program, you MUST go back and undo ALL possible damage that mistake did.

Instead of just fixing the error, ALWAYS go back and relist or redump the ENTIRE program and any related data blocks to be sure there is no hidden damage.

If you don't relist on each and every error, here is what happens. You find a mistake and fix it, and the program still won't behave! Why? Because single stepping or running the mistake caused some very strange op codes to be executed in some very dumb ways. Chances are these plowed some things that you didn't want plowed.

Anyway, you should now be able to run your program. The running rules change from trainer to trainer. On some you punch in an address and then hit a GO button. On others, you set a program counter pointer to some value, as is done with the "*" key on the AIM-65. On others, such as the Apple II, you type an address followed by a "G." Read the user manual that comes with the trainer to find out how to run a program.

So, you tell your trainer to run, and what happens?

Nothing!

What should happen is that the trainer should seem to quit in its tracks, perhaps putting the display off and hanging up any video cursor or prompts.

Why does this happen? Because our program tells the microcomputer to go round and round and do nothing else. There is nothing in our program that does anything to screens or LED displays. Some trainers, such as the KIM-1, will tell you the last adddress you worked on when you stopped the program. Start and stop your program several times. If the final address stays in the range of the address values of your code, then your program is probably working okay.

So if your first discovery module is working correctly, expect—absolutely nothing!

Another rule . . .

If a program is to do anything useful, there must be some output somewhere that can be viewed, used, measured, or put to use by another program.

Looks like computing for the sheer joy of computing won't hack it. Now, we could be sure our program is working correctly by tak-

ing an oscilloscope or logic analyzer with many inputs and plot out what the data bus and address bus are up to as a function of time. This always works, but it is a last resort and very painful way to check out a program.

Let's try to build a program that gives us a useful output by going on to . . .

DISCOVERY MODULE 2

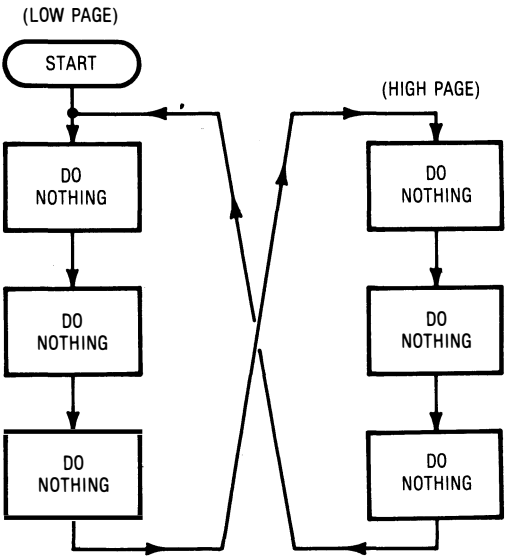
FIGURE EIGHT

Write a program that starts on an even page, does nothing three times, jumps to an odd page, does nothing three times again, and then returns to the initial page.

View address line A8 on an oscilloscope.

This should be easy enough. Three NOPs, a jump to a higher page, three more NOPs, and a return to a lower page. Although it is usually a bad idea, we'll purposely arrange our flowchart to look like a figure 8.

Here's the flowchart . . .



On our fictional MYTH-1 trainer, we will start on page 20 at location $2000 and on page 21 at location $2100. Since two pages of code are involved, use two separate machine language programming forms.

The code looks like this . . .

ADDRESS	OP CODE	BYTE #2	BYTE #3	MNEMONIC	HOW?	NOTES
2000	EA			NOP		DO NOTHING
2001	EA			NOP		DO NOTHING
2002	EA			NOP		DO NOTHING
2003	4C	00	21	JMP	$2100	GOTO HIGH PAGE
2006	—					

LOW PAGE CODE

ADDRESS	OP CODE	BYTE #2	BYTE #3	MNEMONIC	HOW?	NOTES
2100	EA			NOP		DO NOTHING
2101	EA			NOP		DO NOTHING
2102	EA			NOP		DO NOTHING
2103	4C	00	20	JMP	$2000	GOTO LOW PAGE
2106	—					

HIGH PAGE CODE

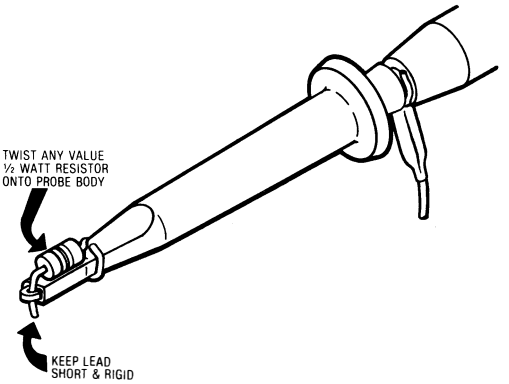
Be sure to separate your code any time there is any break in the addressing sequence. It's best to put each piece of code on a separate sheet. Reasons for this are so that a change in one part of the code won't involve recopying or moving another part, and so that all the code won't get mistakenly punched into the same memory area.

Single stepping should give you four stops on the even page and four stops on the odd page. Once again, when you run the program, everything should disappear.

But now we have something fairly convenient that we can view with an oscilloscope. Set your scope to 1 volt per division (0.1 with a 10X probe), DC coupling, AUTO triggering, and 2 microseconds per division horizontal scan rate. Touch the scope probe tip to make sure it is alive. You then do a further quick check on the scope by picking off +5 volts DC from the highest (counterclockwise-most from the notch) pin on any of the smaller non-memory 14- or 16-pin ICs or some other obvious point. Then grab one of the crystal pins or another known clock signal to check the scope display and locking sync. If you can't read DC or a clock, there is no point in continuing.

Then get out the trainer or personal computer schematic and locate address line A8. The safest place to pick this up is on an expansion connector by glomping onto an extra and empty connector. Another good route is to use a small grabber to catch one pin on one integrated circuit.

Here's a quick mod to a scope probe that lets you easily touch a single IC pin . . .



A PROBE MOD FOR IC PINS

Any old value half-watt resistor will work. Be sure the lead is just long enough to give you a convenient and rigid way to touch a single integrated circuit pin. Some probe accessory kits may have screw-on pin probe attachments. Use whatever you have available.

A gotcha . . .

If you briefly short ANY two points together on a trainer, you are almost certain to destroy your program.

If you briefly short exactly the wrong two points together even for an instant, you will also destroy the trainer.

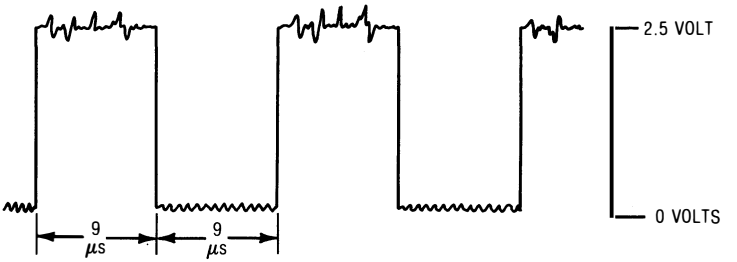
BUT - -

WATCH IT!

It is also a good idea to add a rigid ground terminal in a convenient place to accept the ground lead from your scope and for other measurements. Sometimes you might also like to add tiny wire tie points to other places in the circuit. If you do this, be sure you do not get solder on connector pins or do anything else dumb.

By the way, for best scope results, you must *calibrate* the probe to the particular scope input you are using. This is done by twisting the probe body or turning a small screw while viewing a square wave calibrating signal. Things are right when the square wave is square, without any rounding or overshoot. Do not interchange probes or mix them up once they are calibrated.

Anyway, you should get this picture on your typical 6502 machine when your program is run . . .



On other micro families, you may get holes chopped in the square wave or have other problems. Check the address bus at a memory chip pin or an expansion connector pin, rather than at the microprocessor chip, and see if it gets any better.

If your 6502 trainer has a 1-microsecond clock, you should get a square wave that is high for 9 microseconds and low for 9 microseconds, for a total period of 18 microseconds, or a frequency slightly above 55 kilohertz. More on time and frequency in the next discovery module.

Note that the address lines will be noisy, as will almost any other point you measure in a micro. There will be a small amount of ground noise and quite a lot of noise on the high level. This is normal. As long as you, or any electronic circuit, can cleanly and clearly tell a one from a zero, everything will still work fine.

Let us see what is happening and why on your program. We will now go through what is called *logic analysis* . . .

LOGIC ANALYSIS—Tracing out the operation of a microcomputer by showing EXACTLY what every line on the address, data, and control buses are up to.

This is often a last resort method, but it always works.

Why do we get a square wave? For now, we won't worry too much about what the data bus and the control bus are up to. But let's check out each and every address line versus time. Each address line will have the binary equivalent of its hex address on it . . .

WATCH A-8

ADDR.	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
2000	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
2001	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
2002	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
2003	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1
2100	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
2101	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	1
2102	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0
2103	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	1
2000	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Note that we have gone all the way around and then one more. The patterns should repeat. Now, let's look *vertically* to see what the address lines are up to. Lines A15, A14, A12, A11, A10, A9,A7, A6, A5, A4, A3, and A2 are forever low or zero and don't change. Line A13 is forever a one or high and doesn't change. But lines A8, A1, and A0 have patterns on them.

Our interest is line A8. Note the nice square wave. We get this square wave since our program forced this address line to be low for four instructions and then high for four more instructions. Later we will find that three NOPs and a JMP add up to nine 6502 clock cycles or nine microseconds. We will also get some waveforms on lines A1 and A0, but these may end up unsquare for reasons we needn't bother with just yet.

But we now have a program that gives us a measurable output when viewed on an oscilloscope. And we now see how logic analysis, painful as it is, can almost always show exactly what is happening inside the machine at any instant.

A question . . .

DOING IT:

Connect your scope to address line A1. You should get a faster rectangular wave here. Why?

Now, reset your FIGURE EIGHT program. The square wave turns into a royal mess, but something is obviously still running.

Why? What program are you viewing? Why is it so jumbled?

If you tried to use this square wave for anything, you might overload the trainer. You need better ways to get useful square waves out of micros. Those better ways involve *ports*. They also involve . . .

DISCOVERY MODULE 3

SQUARE DEAL

Write a program that puts a fast square wave out a port.

View the output waveform to make sure it is really square and not rectangular.

Here is sort of a chicken and an egg problem. You need to use ports to know about ports, and you need to know about ports before you can use them. You'll find all about ports in Chapter 8, where they logically belong, included with the rest of the I/O, or Input/Output stuff.

If this is your first trip through the micro cookbooks, stop here and pick up on Chapter 8. If not, let's continue.

We will assume that we have available a parallel port and that we will use the lowest output line for our square wave. We will further assume that this port has to be taught which lines are inputs and which are outputs. We do this on the MYTH-1 by using some teaching hardware at $C080. The actual port location is at $C081. More details on this in the next chapter.

The hidden nasties in this discovery module include finding out how to read and write values into and out of memory locations; how to initialize, or teach ports; and how to measure and use time, frequencies, and clock cycles.

Let's find out how we read and write to memory locations inside a discovery module or program . . .

LOADING AND STORING

With most microprocessors, there is no way that we can directly fill or empty any old location in the address space. We must initially use a working register to put values into, to receive values from hardware in the address space, or to replace values routed to other hardware in the address space.

So . . .

Most operations on most micros must go through one of its working registers.

The accumulator is usually your first choice for this.

Operations that fill the accumulator or other working registers are called *loads.* Operations that take copies of what are in a working register and put them into the address space are called *stores.* Operations that move copies from one working register to another are called *transfers* or *moves,* depending on the micro family chosen . . .

LOAD—The process of filling a working register with a value or a copy of the data in an address space location.

STORE—The process of taking a copy of the contents of a working register and putting that value into an address space location.

TRANSFER—The process of moving a copy of the contents of one working register to another. (6502)

MOVE—The process of moving a copy of the contents of one working register to another. (8080)

Let's look at a transfer first. Here's the 6502 way to put a copy of what is in the accumulator into the X register . . .

TAX	**PUT COPY OF A INTO X**	**AA**
	(IMPLIED addressing)	
1 Byte		2 Clocks
TAX		N and Z flags

Transfers a copy of what is in the accumulator into the X register. The accumulator is unchanged and the old X value gets destroyed.

Assume the accumulator holds an $F2.

2412– AA Puts an $F2 in the X register, leaves an $F2 in the accumulator, destroys the old X value, sets the N flag, and clears the Z flag.

As you do these cards, try to second guess what should go into each location before you actually find the right values. Since the TAX command needs no further information, you can guess that this is an implied instruction that is one byte long. You can also guess that this takes two clock cycles, since two cycles seems to be the minimum used for simple tasks. The symbol for the assembly notation will just be TAX, since this in an implied instruction.

But, unlike previous instructions, when you move something, you might like the flags to change. In this case, the N flag and the Z flag both change. You do not yet know what the N and Z flags are, but you will find out shortly. For now, all you worry about on the cards is whether the flags change and which ones are altered.

Note that the TAX command does not change the accumulator. It simply puts a copy of what is in the accumulator into the X register. Whatever used to be in the X register is destroyed, so if you needed the old X contents, you should have done something else with it before you used this command.

Remember also that our working registers usually hold 8-bit data values, expressed as hex $00 to $FF, decimal 0 to 255, or binary %0000 0000 to %1111 1111. It usually takes *two* hex digits to specify a data value, and it takes *four* hex digits to specify an address that can be anywhere in the 64K address space.

DOING IT:

If your trainer is from the 6502 school, complete the TAX, TXA, TAY, TYA, TSX, and TXS cards at this time.

If not, complete all cards for all instructions that move or exchange data between working registers.

The S in TSX and TXS is the *stack pointer.* You'll find out more on it later. For now, assume it is a dedicated use working register, eight bits of which can be copied to or from the X register.

Well, you just knocked off a bunch of cards, but none of them seems to help much with the discovery module problem of initializing a port or writing data to that port. These instructions simply move things around inside the CPU. They do not involve themselves with address space nor do they offer a way to put something new into the machine.

To put some *value* into the machine for the first time, you need to use the immediate addressing mode. The 6502 immediate addressing command used to fill the accumulator is . . .

LDA	PUT VALUE INTO ACCUMULATOR	A9

(IMMEDIATE addressing)

2 Bytes — 2 Clocks

LDA $F3

N & Z flags

Puts the value of the second instruction byte into the accumulator. Used to initially enter a fixed number into the machine.

2C34– A9 F3 Immediately places the value $F3 into A. Sets the N flag and clears the Z flag.

Note the key words "fixed value." There are other load instructions that go into memory and look at some location and then take whatever they can find there and put it into the accumulator or some other working register.

A rule . . .

Use the immediate addressing mode when you want to put a fixed number or value somewhere.

Use other addressing modes such as absolute long and absolute short when you want to go to a location, and take a copy of whatever is there, and put it into a working register.

You will find lots of different LDAs in the 6502 family. Eight of them for eight different address modes. The absolute long LDA is op coded AD and reaches anywhere in the instruction space to look at an address location, gets a copy of that location, and puts it in the accumulator. This is a 3-byte instruction, since we need an op code, followed by a low or position byte and a high or page byte.

The X and Y working registers can also be loaded in bunches of ways, but they are slightly less flexible than the accumulator.

The concept of "store immediate" doesn't make any sense, so there are no immediate stores. All store instructions take something that is in a working register and try to put it into memory somewhere. What was in that memory location before gets destroyed.

Remember that a memory location can hold RAM, ROM, I/O, or nothing. The CPU can try to write to any location, but it will succeed only if there is RAM or writeable I/O at that location.

Once again . . .

You can store only to a memory location that has RAM, I/O, or other writeable hardware in it.

Our most obvious 6502 store takes a copy of what is in the accumulator and stores it somewhere in the entire address space using absolute addressing.

Here are details . . .

STA	**PUT A INTO ADDRESS SPACE**	**8D**
	(ABSOLUTE LONG addressing)	
3 Bytes		4 Clocks
STA $2345		no flags

Takes a copy of what is in the accumulator and tries to store it in the address space location called out by the position or low-address second byte and the page or high-address third byte. Used to move data from the accumulator into the address space.

Assume the accumulator holds an $F4.

256A– 8D 45 23	Takes a copy of the $F4 in the accumulator and tries to store it at address $2345. Will succeed only if $2345 has writeable hardware in it.

An absolute store gives a way to take a copy of what is in a working register and put it anywhere in the address space. This will be useful to *initialize* a port or to write data to a port, as you will see shortly.

More cards . . .

DOING IT:

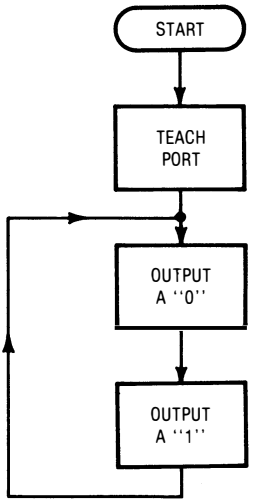
If your trainer is from the 6502 school, complete the LDA, LDX, and LDY immediate and absolute, and the STA, STX, and STY absolute cards at this time.

If not, complete all cards for all instructions that fill working registers with an immediate value and for all instructions that fill or empty working registers from an absolute long location in the entire address space.

Remember to use the programming manuals for the microprocessor's CPU as a dictionary or encyclopedia to find out about each card as the need arises. Do not try to do the cards in alphabetical order. Do not try to complete all address modes for a given mnemonic at the same time.

We now should be ready to do the flowchart for the port square wave . . .

SQUARE DEAL:



We see that we teach our port which lines are inputs and which are outputs. We do this only once at the beginning of the program. This process is called *initialization* . . .

> **INITIALIZATION**—Putting correct values into needed locations at the beginning of a program.

After we teach our port which lines are inputs and outputs, we go on and write a one to the port, then write a zero to the port, and then repeat the process over and over again. Note that we do NOT initialize again. We simply keep outputting ones and zeros. Check the arrow in the left margin of the upcoming code.

As the ones and zeros start streaming out the port, the port will first go high and then low and then high again, and so on forever. This should give us a square wave, or at least something, out the port.

As we have seen, there is no direct way to write to an absolute memory location or to a memory-mapped I/O port in most micro families. You have to put values into working registers and then store the register value in the final location. Normally, you will use the accumulator as your first choice of working register . . .

> **If there is any doubt about which working register to use, always try to use the accumulator first.**

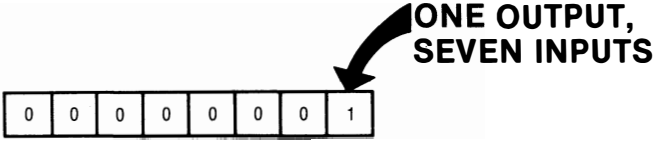
Naturally, if the accumulator is busy, you will either have to save its old contents elsewhere or have to use a different working register. This can become quite a juggling act in some micros. In the 6502, we have the X and Y registers to help us out, and we will see that we can easily shove accumulator values on to and off of a *stack* for temporary storage. We will also see that all of page zero is easily reached in an absolute short addressing mode. These locations can sometimes serve as an additional 256 working registers.

Now, if we only knew how to teach our port, we would be all set.

Full details on this appear in Chapter 8, but let's preview here. We've assumed our MYTH-1 trainer has a VIA type teachable 8-bit parallel port. The teaching address is $C080, the actual port is

located at $C081. To teach the port, place a one for each output line and a zero for each input line into the teaching location.

If we use the "zero" port line for an output and make all the other lines inputs, the teaching pattern will look like . . .



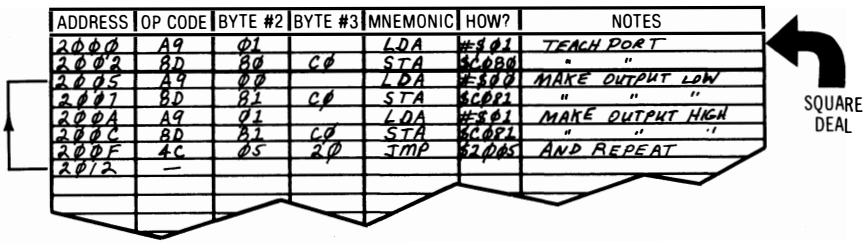
. . . which in this case is a hex $01. So we write this pattern to our teaching location $C080 by first filling the accumulator immediately with the value $01 and then by storing the accumulator into the teaching location.

Two gotchas. First, note that any ODD binary value into the teaching location will make the zero line an output. There are 128 possible values that can make any single port line an output and another 128 possible values that can make any individual port line an input. *Each bit stands on its own.* Teach a one and you get an output line. Teach a zero and you get an input line.

Second, be careful not to mix up the teaching location with the actual port location. On a VIA style port, the teaching location contains some hardware that will configure the port into your desired number of inputs and outputs. The port location is the actual funnel where data passes into or out of the microcomputer.

Note also that my comments here apply only to VIA style ports. Other ports will behave differently. See Chapter 8 for more details.

We are now ready for code . . .

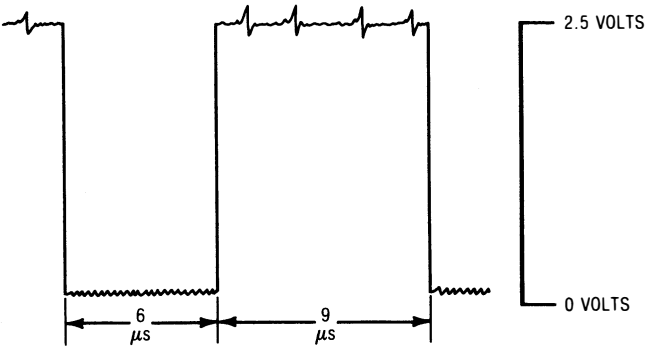


As usual, you fill in the blocks from RIGHT to LEFT. First load the accumulator with your input/output pattern and then store this in the port teaching location. Since this initialization is a separate task, we've drawn a heavier line across the form. Usually, you initialize

only once at the start of any program. Later jumps should skip this part of the code.

You then put a #01 immediately into the accumulator and then shuffle it off into the port. This is followed by an #00 and then a round-and-round jump.

Glomp your oscilloscope onto the port and you'll see . . .



Uh, oops. That's a waveform all right, and the waveform comes and goes with running and not running of the program. But this is *not* a square wave. It is a *rectangular* one! The low time is clearly less than the high time.

In any beginning micro class, there is always one whiz kid who runs out and gets this result way ahead of everyone else. It sure puts a crimp in his style when you say that isn't what you assigned.

Several points here. First, if you do not have an oscilloscope, you can use an ordinary VOM or voltmeter and you will get a voltage that is around 3 volts out of a NMOS port with this waveform when it is running. What you are doing is duty cycle averaging the lumps and reading the average voltage on the meter. While a scope is very useful for learning micros, you can use this voltmeter substitute in this module. The rest of the modules will give you other ways to measure or observe results.

The second point is that you should get a clean rectangular wave out on any trainer from any micro family. The address bus may get messed up with other stuff in different families, but you should get a clean rectangular wave out this port. If you don't, you have done something wrong.

DOING IT:

Here are some hex dumps of four mistakes beginning students have made with this module.

Disassemble the code, analyze the one mistake made each time, and show what the result would be.

```
2000- A9 01 8D 80 C0 A9 00 8D
2008- 81 C0 A9 01 8D 81 C0 FF
2010- FF FF FF FF FF FF FF FF

2000- A9 01 8D 80 C0 A9 00 8D
2008- 80 C0 A9 01 8D 80 C0 4C
2010- 05 20 FF FF FF FF FF FF

2000- A9 01 8D 80 C0 A9 00 8D
2008- 81 C0 A9 00 8D 81 C0 4C
2010- 05 20 FF FF FF FF FF FF

2000- A5 01 8D 80 C0 A5 00 8D
2008- 81 C0 A5 01 8D 81 C0 4C
2010- 05 20 FF FF FF FF FF FF
```

The last problem is a subtle one. More on it when we get to page zero addressing. Hint: What value would you expect to find stored in location $0000? In location $0001? Could the program still work? What are the odds?

Anyway, it looks as if we have to go back to the drawing board. We do not have a square wave out a port. Instead, we have a rectangular wave out a port.

Maybe now is a good time to look into . . .

TIME, FREQUENCY, AND CLOCK CYCLES

During a windstorm in an orchard, you might measure four plums per second falling off a tree. The *frequency* of plum fallings is four plums per second, while the average *time period* between plum fallings is one-fourth of a second, or 0.25 seconds.

Which leads us to . . .

FREQUENCY—The number of events per unit time.

TIME PERIOD—The time between repetitive events.

On an electronic waveform, a *cycle* is defined as "one trip around," or all values we go through before the waveform begins exactly repeating. On a rectangular or square wave, one way to show a cycle is from positive edge to positive edge. A cycle on a square wave thus has *both* a high time and a low time.

The frequency of a waveform is simply the number of cycles per second. The time period of a waveform is the time it takes to go through exactly one cycle, or to get back to the same starting point in the next cycle.

Frequency is measured in *hertz,* or cycles per second; in *kilohertz,* or thousands of cycles per second; in *megahertz,* or millions of cycles per second; and in *gigahertz,* or billions of cycles per second. Gigahertz frequency values are not common in microcomputer work, but they are very important for such things as microwaves and satellite television. Time periods are normally measured in *seconds;* thousandths of a second, called *milliseconds;* millionths of a second, called *microseconds;* or billionths of a second, called *nanoseconds.*

Thus . . .

FREQUENCY MEASURMENT UNITS

HERTZ—The number of cycles per second.

KILOHERTZ—The number of thousands of cycles per second.

MEGAHERTZ—The number of millions of cycles per second.

Here is how these frequencies are related . . .

1 hertz = .001 kilohertz = .000001 megahertz
1000 hertz = 1 kilohertz = .001 megahertz
1000000 hertz = 1000 kilohertz = 1 megahertz

Similarly . . .

TIME PERIOD MEASUREMENT UNITS

SECONDS (s)—The number of seconds per cycle.

MILLISECONDS (ms)—The number of thousandths of seconds per cycle.

MICROSECONDS (μs)—The number of millionths of seconds per cycle.

NANOSECONDS (ns)—The number of billionths of seconds per cycle.

And these are related by . . .

.000000001 s =	.000001 ms =	.001 μs =	1 ns
.000001 s =	.001 ms =	1 μs =	1000 ns
.001 s =	1 ms =	1000 μs =	1000000 ns
1 s =	1000 ms =	1000000 μs =	1000000000 ns

Time and frequency are inverses. The number of plums per second equals one divided by the time between plum fallings.

Most electronic people change all these numbers to scientific notation, and they always use powers of ten that are a *multiple of three*. Thus . . .

10^6 hertz = 10^3 kilohertz = 10^0 megahertz

and . . .

10^0 seconds = 10^3 milliseconds = 10^6 microseconds

Here is how you go about . . .

RELATING FREQUENCY TO TIME

TIME = 1/FREQUENCY
SECONDS = 1/HERTZ
MILLISECONDS = 1/KILOHERTZ
MICROSECONDS = 1/MEGAHERTZ

As we have seen, there is also a frequency multiple called *gigahertz* for a billion hertz, and gigahertz are the inverse of the period in nanoseconds. But, while nanoseconds are often used to measure micro speeds, the gigahertz term is not used or seen much in most micro work.

Note that there are no funny exponents involved if you work only in seconds and hertz; or if you work only in milliseconds and kilohertz; or if you work only in microseconds and megahertz. So, it pays to pick the units to fit the problem and avoid funny number hassles.

Some insights on how much is what. In one nanosecond, light travels almost one foot. There are more nanoseconds in three seconds than there are seconds in a century. Turning to frequencies, the power line has a frequency of 60 hertz, or 60 cycles per second. Most people can hear over a range of 30 hertz to 12 kilohertz. A bat may communicate at an ultrasonic frequency of 50 kilohertz. The carrier frequency of an AM radio station ranges from 550 kilohertz to 1500 kilohertz, or is roughly centered at one megahertz. Television frequencies start at 50 megahertz for channel two and work their way up.

A typical electronic logic gate can make some ten million or more decisions per second. That's just your everyday LS TTL gate. If we really want to get snappy, we can run hundreds of times faster with special circuits. The regular 6502 can typically carry out a half-million simple instructions per second, while a premium 6502B can quadruple that. The access time of a typical dynamic RAM is around one-fifth of a microsecond, or 200 nanoseconds.

On a microcomputer with a video display, one microsecond is often the time needed to put one character on the display. One character dot may take one-eighth of this time, leading to a video dot frequency of 8 megahertz. Screens eighty characters wide will halve the character time and double the video bandwidth.

The time period of the rectangular waveform you just got in your port code is found by adding the high time to the low time to get the total cycle time. Depending on the trainer, you probably got a high time of 9 microseconds and a low time of 6 microseconds, for a total period of 15 microseconds. This corresponds to a frequency of 1/15 megahertz, or 1000/15 kilohertz, or 66.7 kilohertz. This is in the high ultrasonic part of the frequency spectrum.

How do we know just how long an event will take on a microcomputer? This depends on several different things.

Most microcomputers define a *clock cycle* as the time interval needed to complete one internal CPU event . . .

CLOCK CYCLE—The time interval needed to complete one internal CPU event.

Clock cycles are very simple on the 6502. A minimum of two identical clock cycles are needed to complete a simple instruction. In other micro schools, clock cycling may get very complicated and a dozen or more clock cycles may be needed to complete even a simple instruction.

The *clock frequency* is a reference frequency that is either sent to a microprocessor's CPU or else is internally generated by it. The frequency of the reference going into the clock pin sets the time of one clock cycle.

But different microprocessors use this clock frequency in different ways. For instance, a 6502 using a 1-megahertz clock can do some things faster than a Z-80 using a 4-megahertz clock. This happens because it takes many more clock cycles to complete certain Z80 instructions than are needed to complete similar 6502 instructions.

So, you can't automatically decide that a 4-megahertz CPU is faster than a 1-megahertz one, particularly if they are from different families. You also cannot usually go by the number stamped on the crystal on a trainer or personal computer. Very often, a crystal of much higher frequency is used to generate the timing for color video displays and other high frequency needs. This crystal frequency is suitably divided down to form the clock input to the microprocessor.

On most 6502 trainers, the microprocessor CPU clock frequency is 1 megahertz, resulting in a clock time of one microsecond. The Apple II runs very slightly faster, having a 1.023 megahertz clock and a resultant 0.978 microsecond clock cycle . . .

On most 6502 trainers, the clock frequency is one megahertz, and the clock cycle time is one microsecond.

The Apple II is very slightly higher in frequency and has a very slightly shorter cycle time.

So far, so good. The cycle time is the time needed to complete one internal CPU event. But it takes at least *two* internal CPU

events to complete even a simple 6502 instruction, and more complicated instructions may take many more.

To get the time per instruction, multiply the clock cycle time by the number of clock cycles per instruction . . .

FINDING INSTRUCTION TIMES
The instruction time equals the clock cycle time MULTIPLIED by the number of clock cycles per instruction.

For instance, if a JMP absolute instruction takes three clock cycles, and if the trainer you are using has a 1-microsecond clock cycle, it then takes 3 microseconds to complete a JMP instruction.

To find the total task time, add up the times of all the individual instructions . . .

FINDING TASK TIMES
The time to do a task is found by ADDING all the individual instruction times needed to do that task.

Task times turn out to be very important on any microcomputer program that is involved in providing timing. Task times also get critical if the entire job takes too long and you have to find a faster way to get similar results.

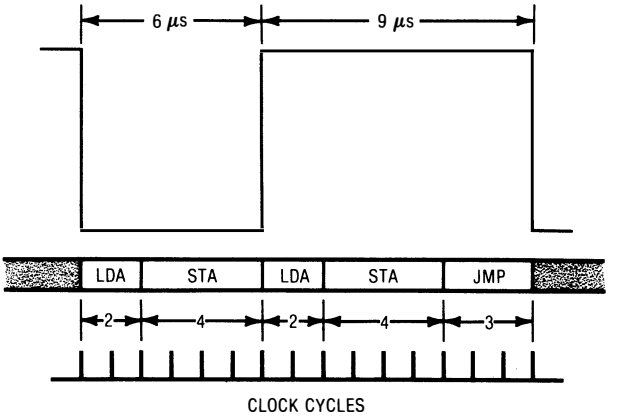
Let's try out this timing information on our rectangular wave and see what it shows us. We will assume a 1.0 microsecond clock.

The initialize time takes 6 microseconds, since it takes two clock cycles to immediate load and four clock cycles to absolutely store. But since this initialize process is done only once, it doesn't affect the square wave we get out. All it does is delay negligibly the time the rectangular wave starts.

From the time the clock goes low, we have a load immediate and a store absolute that add up to 6 microseconds. But from the time the clock goes high until it goes low again, we have a load immediate, a store absolute, and a jump absolute, or 2 + 4 + 3 = 9 microseconds. The rectangular wave is low for 6 microseconds and high for 9 microseconds.

So after the first trip around, we get a rectangular waveform, because the jump takes time and because the jump happens only when the output is high.

Let's redraw our rectangular waveform, along with the instruction times that make up the waveform . . .



We can make our rectangular wave into a square wave by adding another 3 microseconds of delay during the time the output is low. We might try using a NOP, but this gives us only 2 microseconds of extra delay. Two NOPs together will overdo it and give us 4 microseconds of extra delay.

Too little or too much.

But you already have one instruction on your cards that takes exactly three clock cycles to execute. So how would you . . .

DOING IT:

Add one instruction to your port SQUARE DEAL discovery module to produce an exact square wave.

Test and demonstrate it.

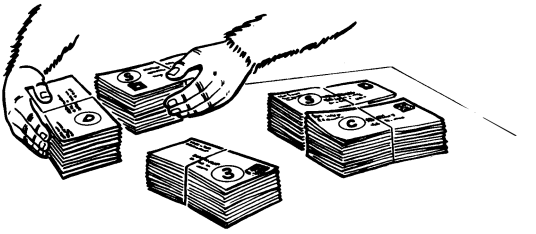
That ought to just about do this module. The hidden nasties here included finding out about loads and stores, accumulator action,

time, frequency, and clock cycles, calculating exactly how long a program takes, more scope work, and learning about and using ports.

Before we go on to the next discovery module, let's find out what we need to know about . . .

FLAGS

Have you ever gotten some mail with a mysterious and bright sticker on it, say a green 3, a red D, a yellow C or an orange S?



The reason you got this sticker on your mail is that it happened to be the top piece in a post office bundle, or else the top piece in a bulk mailing bundle.

The orange S sticker stands for STATE. This tells the postal employees that everything in the bundle is for the same state. You only have to read the sticker and the state on the top label to know the entire bundle is intended for a certain state.

The yellow C is a CITY sticker for large cities that have more than one zip code. The red D stands for DIRECT or DIGIT, meaning that everything in the bundle goes to the same zip code. There's a fairly rare blue F for FIRM if the entire bundle goes to one large company.

And finally, there's that green 3, which stands for SCF, or *Section Center Forwarding* and which means that the three most significant zip code digits are the same. Post offices aren't allowed to exchange mail with each other. They always have to send up to the SCF office and then the SCF office has to send it back down again. While this greatly simplifies the mail flow, it can lead to absurdities. For instance, if you are in the Thatcher shopping center and want to pay a bill owed to a store in the Safford shopping center across the street, the bill has to make a 190-mile round trip through Globe to get there, since Globe is the 855 SCF.

Anyway, what these stickers do is save having to look at each piece of mail in the bundle. Instead, this one sticker is a marker, or *flag,* that tells what the present *status* of the entire bundle is.

Idiot lights on cars are another example of flags. These red lights do not tell you exactly what is right or wrong, but they get your attention when and if it is needed.

The attention you pay to an idiot light may depend on what you are driving and what you are doing at the time. If it's late afternoon, you are fifteen minutes from home, own an American car with a water-cooled engine, and the GEN light goes on, you can probably drive on home without any immediate problems. But if you have an old air-cooled VW bug, that GEN light means you also have lost your engine cooling and that you must stop right now and check the fanbelt.

Microprocessors also have flags . . .

FLAG—A 1-bit indicator in a micro that tells you something about the status of the machine that may be useful.

All microcomputers have flags. Sometimes, you can completely ignore them. Other times, they are a very convenient and very handy way of finding out what is happening or what has happened. A flag on a package of mail eliminates tearing the bundle apart to find one piece of information. Similarly, micro flags greatly simplify finding out certain things about your program.

One very important use of flags is to make decisions. If a flag is set, you go on to something new. If the flag is not set, you repeat what you just did. Or, you can let a flag show which of two new routes to pick. Or you can let a flag select certain operating modes of your machine.

The number and type of flags varies with the micro, although there are typically three to eight flags available for your use.

Just as you can group eight warning lights on a car dashboard, you can also group your flag bits together into a single 8-bit word. This 8-bit word behaves totally differently than most 8-bit words though, since it is made up of eight totally independent bits, each doing its own and private thing.

When we group together 1-bit flags into a single word for convenience, we form a dedicated use register called a *processor status register* . . .

PROCESSOR STATUS REGISTER—A dedicated use working register that holds flags as individual bit values.

One use of a processor status register is to take a sort of snapshot of what all the flags are up to at any instant. This can be handy if the micro gets interrupted. By saving where it is in a present program, and by also saving the processor status register that holds our flags, we can later pick up where we left off.

The processor status register is also very useful for debugging, since it shows you what all the flags are up to with one check. It also gives you a way to simultaneously set all flags the way you want them.

More often than not, however, each and every flag will behave totally differently and you will be involving yourself with only a single flag at a time. It is only for unusual things like initializing, debugging, or interrupts that you will work with all the flags at once in the processor status register.

On a 6502, the flag or processor status register is also called the P register. One way to remember this is . . .

On a 6502, the P register is the Processor Status Register that holds all the flags for us.

Remember this as the . . .

Phlag

. . . register.

Each flag does a different thing and behaves differently in a micro. Some are automatically changed on many instructions. Others can be set or reset with software. Others will be altered only by the CPU as the result of certain internal operations. Some flags on some micros can also be hardware controlled by the outside world.

There are two rules to flag use. The first rule is that the flag remembers the last thing that affected it as long as power is applied, regardless of how many new instructions or actions go by that do not affect it.

But, even if flags will remember things for us a very long time, the second rule tells us that it is always a good idea to use a flag state

immediately after it is altered. This prevents any later instructions from surprising you and is particularly important on CMP or compare instructions. So . . .

Flags remember their state from the last time they were altered.

BUT—

You should always use a flag immediately after you alter it.

Let's look at some typical flags and see what they do and how they work. There are three flags that are very common and very important and are found in just about every microprocessor. These are the Z flag, the N flag, and the C flag, short for *zero, negative,* and *carry.*

The Z for *zero* flag goes to a one on any zero result that affects the zero flag. It goes to zero otherwise . . .

ZERO FLAG—Sets or goes to a one on any zero result that affects the flag.

Clears or goes to a zero on any non-zero result that affects the flag.

The Z flag works automatically with most micros. Anytime you do anything that affects this flag, it will keep track of whether you have a zero result.

This is useful in counting down something to zero, in testing for a register overflow, and in finding out whether two things are equal in comparisons and other logic operations.

There is usually no direct way to hardware or software set or clear the Z flag by itself. Its operation is usually fully automatic. Z flags are normally used with conditional moves to let you alter program flow on a zero result.

The N or *negative* flag also works automatically and is available on most micros. Any time you do anything that affects this flag, it will keep track of whether the most significant, or leftmost, bit is a one or a zero . . .

NEGATIVE FLAG—Sets or goes to a one on any affecting result whose leftmost bit is a one.

Clears or goes to a zero on any affecting result whose leftmost bit is a zero.

What the N flag really does is take a copy of the MSB, or most significant bit, of any result that activates this flag, and saves a copy of this bit for you. The N flag is called an S flag on certain microprocessors. The S here stands for *sign*.

The reason this is called an N flag is that, in 2's complement signed binary arithmetic, a positive number will have a zero MSB and a negative number will have a one MSB. Thus the N flag is zero for certain positive numbers and a one for certain negative numbers.

But regardless of whether or not you are involved with 2's complement arithmetic, the N flag always sets on an MSB set and always clears on an MSB cleared on any instruction that affects this flag.

The N flag is also usually automatic, and there is normally no way to software set or software clear the N flag.

The third of the three most important and most common micro flags is called the C or *carry* flag. The C flag can be software set or cleared and also keeps track of carry and borrow activities in some arithmetic commands . . .

CARRY FLAG—A general-use flag settable or clearable by software instructions.

Also will go to a one if a carry is needed after an addition or will go to a zero if a borrow is needed after a subtraction.

The carry flag can do lots of different things. Its intended use is to keep track of carry and borrow operations during straight binary arithmetic. Normally, you *clear* the carry before starting an addition. Should the carry set after an addition, this means you have to add one to the next most significant decade or whatever you are adding.

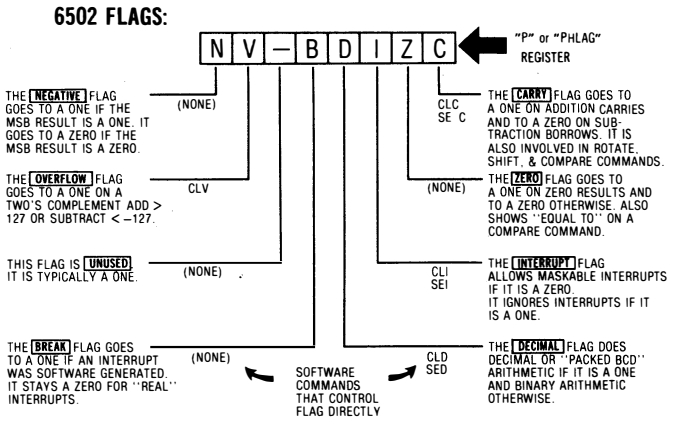
Similarly, you normally *set* the carry before starting a subtraction. If the carry clears after a subtraction, this means you have to subtract, or borrow one from the next most significant decade or whatever you are subtracting.

Besides this, if you are not doing arithmetic, you can use the carry flag as a 1-bit reminder of where you are in a program. For instance, one way to do some task twice in a micro is to go through the task with the carry cleared the first time, then go through it a second time with the carry set. This only works, of course, if there is no arithmetic or other instructions that affect the carry flag in your task.

Certain other instructions alter the carry flag. There are *shifts* and *rotates* that will let you move bits from working registers or address space RAM into the carry flag. You can then test this flag and react in one of two different ways. The carry flag also may be used to hold "greater than" results during a *comparison.*

While these three flags are far and away the most useful and the most common, there are other specialized flags available on different micro families.

The 6502 has seven flags. Here's a rundown of how they are arranged in the P or Phlag register and what they do . . .



Starting at the left, we have our N, or negative, flag which sets on an MSB one and clears on an MSB zero and is typical of most any micro.

This is followed by the V, or *overflow,* flag that sets or clears on a "plus or minus 127" overflow or underflow in 2's complement signed binary arithmetic. This flag can be software cleared. Interestingly, it can also be hardware set with a pin on the 6502 CPU, although this intriguing feature is rarely used.

The next flag is not used, but most often will read as a one. It is labeled "-" and presumably was reserved for future expansion.

The B flag stands for *break.* There is a powerful software debugging command in the 6502 called a BRK and coded hex $00. If you

get to this code in your program, the CPU will immediately interrupt itself and go to wherever it normally would on an outside-world interrupt. The break flag can tell the experienced 6502 programmer whether an interrupt was a "real" outside world event or an "artificial" programmer-created break. Operation of the B flag is automatic. More on BRK later.

The D flag stands for *decimal.* There are two ways to do arithmetic in the 6502: the usual or binary method and a "by decades" decimal method using a code called *packed BCD.* This sounded like a great idea once, but nobody seems to use this decimal mode any more because it can cause all sorts of unusual troubles if you aren't careful.

There are software commands to set and clear the D flag. Setting it puts you into the decimal mode. Clearing it puts you into the normal or binary mode. Most 6502 monitor and operating systems will automatically put you into the binary mode on reset or startup, but it is always a good idea to reaffirm binary with a CLD command as the first instruction in your program.

If you ever get some really weird results, check to see if this decimal flag got changed somehow. The Apple II has a strange DOS bug, known as the "$48 problem," caused by accidental setting of the D flag.

The next flag is the I or *interrupt* flag. Remember that an interrupt is some outside world event that demands the computer's attention. The I flag lets you allow or disallow certain interrupts. These are called *maskable* interrupts. If the I flag is *set,* interrupts are *not* allowed. If the I flag is *cleared* or zero, the interrupts *are* permitted.

The Z for zero, and C for carry flags are the same as you will find on most other micros.

DOING IT:

Investigate how all your flags work at this time.

Other micros will have other flags for other uses. You might find a *half decimal* flag that lets you convert binary math into packed BCD results. We might say that this is the opposite of the D flag in the 6502. The D flag puts you into decimal ahead of time; the half-decimal flag lets you repair a binary answer after it is complete. The half decimal flag is sometimes called an *auxiliary carry* flag.

Once again, decimal arithmetic inside micros is very much on the wane and is being dropped in many newer chips.

There might be some general use flags that you can access from hardware pins as well. Newer 16-bit micros may include an X or *extend* flag that decides which part of the address space is in use, an S for *supervisory* flag that decides whether the program code is to be protected, and a T or *trace* mode that gives you debugging options. You might also get more than one interrupt flag.

Many flags can be software controlled. Here is how to clear the carry on the 6502 . . .

CLC **CLEAR THE CARRY FLAG** **18**

(IMPLIED addressing)

1 Byte — 2 Clocks

CLC	clears C. flag

Forces the carry flag to the zero or cleared state. Normally used before addition to be sure you add what you think you do. Also used as a general purpose flag when no arithmetic is involved.

21B3– 18 Clears the carry flag and goes onto the next instruction at $21B4.

On the 6502, you can software set or clear the C, D, and I flags. You can only software clear the V flag. There is, however, a seldom used pin on the CPU that lets you hardware set the V flag.

More cards . . .

DOING IT:

If your trainer is from the 6502 school, complete the CLC, SEC, CLD, SED, CLI, SEI, and CLV cards at this time.

If not, complete all cards for all instructions that set or clear flags.

Some microcomputer systems also have system level flags. These system level flags are sometimes called *soft switches* . . .

SOFT SWITCH—A system level flag that does things like picking video display modes or text formats.

The difference between a flag and a soft switch is that the flag applies to the CPU in the micro*processor*, while a soft switch is used for some high level mode picking that involves the entire micro*computer* system. Soft switches can be built from flip-flops or latches and are always outside the CPU chip.

There are many obvious soft switches used in the Apple II computer. Four of these interact to pick video display modes. Another four can output to the outside world by way of the game paddle connector. Two more are used for speaker and cassette outputs.

You'll find flags to be a great convenience that very much simplifies your design of microcomputer programs. You will also find that if you don't fully understand what flags are and how they work, they will surely return to haunt you later.

One of the most common and most important uses of flags is to let the computer make a decision and alter its course of action based on that decision. It does this by means of . . .

THE IF INSTRUCTIONS

We have already seen that there are conditional and unconditional instructions. The unconditional instructions, such as a JMP absolute long, do their thing regardless of what else is happening.

Conditional instructions are what make a computer "smart." With conditional instructions, the computer can test and alter its own future course of action based on where it is now. The ability to test and make decisions is central to microcomputer intelligence.

Conditional instructions usually involve flags. If the flag is in one state, the instruction decides "YES!" and goes somewhere else to continue. If the flag is in the other state, the instruction decides "NO!" and essentially behaves as a NOP, going on to the next instruction as though nothing happened.

On the 6502, all the conditional instructions are called *branches*, and their mnemonics always start with a "B." These branch instructions use relative addressing and hop so many squares forward or backward using 2's complement signed binary.

Other micro families have conditional jump instructions that also test flags, such as a JNZ (short for "jump if not zero").

There is one very big advantage of relative branches over absolute jumps. Relative branches are relocatable and let you easily move your code around in the address space, where absolute jumps require recoding when you try to move your program somewhere else in memory.

Any of these conditional jumps or branches can be called IF instructions . . .

> **IF INSTRUCTION**—An op-code instruction that tests a flag and then picks one of two actions based on the condition of that flag.
>
> Conditional branches and conditional jumps are typical IF instructions.

Let's look at an example. Here's the 6502's BNE (short for Branch IF Not Equal) . . .

BNE — **BRANCH IF *NOT* EQUAL** — **D0**

(RELATIVE addressing)

2 Byte — 3 clocks if taken, 2 clocks if not.

BNE $267A — no flags

Tests the Z flag. If the Z flag is a one or set, does nothing and goes on to the next instruction. If the Z flag is a zero or cleared, jumps forward or backward by the 2's complement amount held by the second instruction byte.

267B– D0 FD	Goes to $267D IF the Z flag is set and goes to $267A IF the Z flag is cleared.
267B– D0 03	Goes to $267D IF the Z flag is set and goes to $2680 IF the Z flag is cleared.
NOTE —	Assembler notation shows absolute address, but second op-code byte is in 2's complement binary.

What this instruction does is check the zero flag. If the zero flag is *not* equal or set, that branch is taken. If the zero flag *is* equal or set, that branch is not taken, and the program continues as if nothing had happened.

The direction the relative branch goes in is decided by the value of the second byte. If the second byte has its MSB cleared (values $00 to $77), then the branch goes *forward* through memory. If the

second byte has its MSB set (values $80 to $FF), then the branch goes *backward* through memory.

A forward branch is usually used to *skip over* part of the code, while a reverse branch is often used to *repeat* previous instructions as part of a loop.

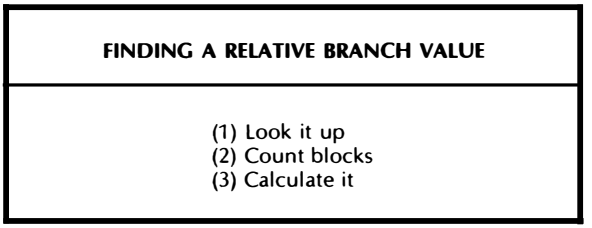
With a 6502 relative branch, you can go plus or minus 127 bytes from where you are. If you have to go further, you use a branch to a JMP absolute instruction. This lets you reach any part of the address space with a test and branch.

One of the most painful things about tapping the power of the relative branch involves . . .

calculating relative branches

When you decide you want a relative branch, you have to tell the instruction how far to go and in which direction. This sounds easy enough, but it can be a real hassle if you are new to micros.

There are at least three good ways to find relative branches . . .



Once again, the problem is to put some value into the second byte of a relative branch instruction that makes us go just far enough in the right direction to pick up exactly the point at which we want to continue computing.

The simplest way is to look up the answer somewhere. Almost any assembler will automatically figure out the 2's complement value for you. All you do is tell the assembler which address you want to go to, and it does the dogwork, even down to filling in the actual value into your program.

Many personal computers and trainers also will help you with this task. For instance, the old PAIA 8700 has a built-in branch calculator in its monitor, and the Apple II will do hex arithmetic directly from its keyboard.

There is an automatic circular slide rule calculator built into Chronicle 6 of *The Hex Chronicles* (Howard W. Sams 21802) that instantly gives you the answer, following the simple use instructions.

Any of these methods work quickly and simply.

Our second method is a "by rote" one called the *block counting* method. Here is how it works . . .

BLOCK COUNTING METHOD

Always remember that the WORST possible value you could ever put in a relative branch second byte is $FF! So—

(1) Draw a branch arrow and show the EXACT square you wish to branch to.
(2) Lightly put a FF and a "?" in the second branch byte.
(3) Count OCCUPIED blocks to find the branch value.

For FORWARD branches, count frontward "FF-00-01- . . ." once each occupied block.

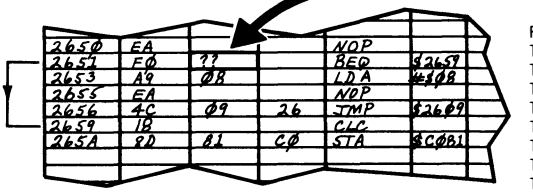
For BACKWARD BOXES, count backward "FF-FE-FD- . . ." once each occupied block.

The method is fully automatic and it easily works by rote. You complete your program except for the second byte of your branch, making sure that you show the "branch taken" absolute address as an operand following the mnemonic. You then lightly put the *worst* possible value—$FF—in the second byte of the op code and simply count occupied blocks frontward or backward.

For forward branches you count "FF," "00," "01," "02" . . . and so on till you hit the square you are trying to reach. Each time you touch a square, call out loud one higher value. When you get to the correct square, replace the FF with the correct value.

Like so . . .

COUNTING BOXES FOR A FORWARD BRANCH



THIS IS THE "PROBLEM" SQUARE. WE HAVE TO PUT THE RIGHT VALUE ($00 to $7F) HERE TO GO FORWARD EXACTLY THE RIGHT NUMBER OF SQUARES.

FF IS ALWAYS WRONG. SO TOUCH THIS SQUARE AND SAY FF.
THEN, TOUCH A9 & SAY 00.
THEN, TOUCH 08 & SAY 01.
THEN, TOUCH EA & SAY 02.
THEN, TOUCH 4C & SAY 03.
THEN, TOUCH 09 & SAY 04.
THEN, TOUCH 26 & SAY 05.
THEN, TOUCH 18 & SAY 06.

BUT 18 IS WHERE WE WANT TO GO, SO WE PUT 06 INTO THE ?? SQUARE. THAT'S IT!

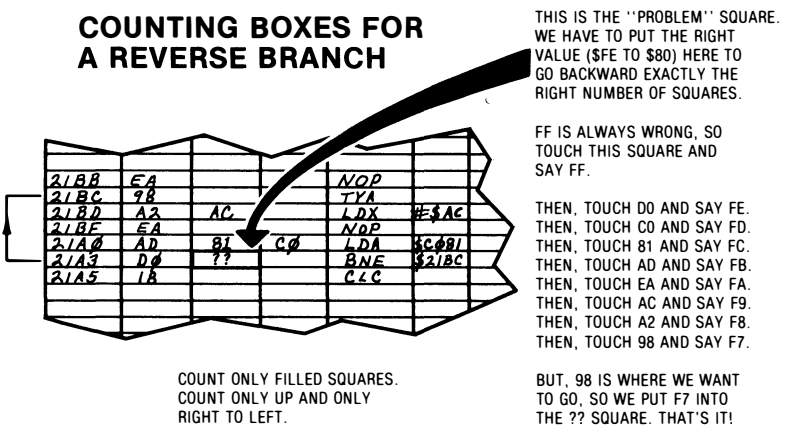
COUNT ONLY FILLED SQUARES. COUNT ONLY DOWN AND ONLY LEFT TO RIGHT.

If you don't yet know how far forward you want to go, just take a guess for the operand address, and put a "??" in the magic square. Later, when you get on down the programming form to the address you really want to forward branch to, just go back and replace the guess operand with the real address, and count the blocks to replace the ?? with the needed value. One big advantage of assemblers is that they use *labels* that let you name things before you know where they really are.

Be sure to show those branch arrows on the left margin of your programming form.

A "backward" branch means you are going to go "uphill" on your programming form. It also means that the "branch taken" value is less than your present address, and that the branch value will be a number from $FF down to $80 in 2's complement form.

Here is how you do a backward branch by the block counting method . . .



And that's all there is to it. Don't think about it. Just put an absolutely wrong FF into the magic square and count occupied blocks. The method does, of course, get tedious if the branch goes more than a dozen occupied blocks in either direction. But most branches are short ones.

If all else fails, you have to find the branch by the method that the math freaks would have forced on you in the first place. This requires both thought and an insight into what is happening.

Anyway, here is the "real" way to find a relative branch value . . .

THE OFFICIAL "MATH FREAK" WAY

(1) Write down the absolute address of the op code at the BRANCH TAKEN address.
(2) SUBTRACT the absolute address of the op code at the BRANCH NOT TAKEN address from the above.
(3) If the answer is negative, convert the answer to 2's complement signed binary.
(4) Put the answer in the magic block that follows the branch op-code byte.

Whichever method you use, you are allowed to go only plus 127 or minus 127 bytes from where you start. Any branch beyond this is out of range, and instead of going frontward, you end up going backward, and vice versa . . .

On 6502 relative branches—
FORWARD branches above $7F are a no-no!
REVERSE branches below $80 are a no-no!

So, finding a branch value is no big deal. But you must very carefully check each branch value to make sure it goes exactly where you think it does or your program will bomb. Branches must always go to a valid op code and never to an operand or a data file value, or you will end up in deep trouble.

timing

The timing on a relative branch is interesting. If the 6502 relative branch is *not* taken, the timing is the same as a NOP, or two clock cycles. But if the branch *is* taken, the branch takes one additional clock cycle, for a total of three. The extra cycle is needed to change the program counter to its new value.

So, if timing is critical, you have to add up the branch taken values separately as three cycles and the branch not taken values as two cycles each.

Oh, yes. There is an even more subtle timing gotcha. It doesn't happen very often, but it sure can foul you up if your timing is very crucial.

Most of the time, branches move forward or backward to addresses on the same page. If you cross a page boundary, the branch will still work the way you expect it to, but it will take an additional clock cycle. This happened to me on a very touchy timing program in the *Enhancing Your Apple II* series and it was very hard to pin down. If your timing seems inexplicably off by one, check this page crossing detail.

Otherwise, the extra cycle is so rare that you can ignore it.

IF revisited

IF instructions usually come in pairs . . .

> **IF instructions usually come in pairs.**
>
> **If the right one don't get you then the left one will.**

For instance, we'll find there's a BEQ, or branch if equal, that does the exact opposite of the BNE. This one takes the branch on a zero result and does nothing on a non-zero result.

If you find a decision backward from how you would like things to be, check into the opposite or complementary instruction and op code, and it will usually be just what you need.

There are eight relative branch instructions on the 6502, a pair for each of the C, N, V, and Z flags. Looks like card time again . . .

> **DOING IT:**
>
> If your trainer is from the 6502 school, complete the BCC, BCS, BNE, BEQ, BMI, BPL, BVC, and BVS cards at this time.
>
> If not, complete all cards for all IF instructions that test a flag and alter the program flow as a result of that test.

Other micro schools may have conditional jumps rather than conditional branches. Most conditional jumps in most other micro families use absolute long or absolute short addressing modes rather than relative addressing. On the 8048, there are separate conditional jumps available for each page of memory.

On some newer 16-bit micros, you also have *long branch* relative jumps that can hit any point in a 64K address space. These long branches are calculated the same way you do the short ones, only the forward values can range from $0000 to $7FFF, and the backward values from $FFFF down to $8000.

You can, of course, fake a long branch on the 6502 by doing a branch to a jump absolute address. You lose relocatability, though, since you will have to modify the absolute jump address if you move the program elsewhere.

By the way, and again on the 6502, it is very easy to get BMI and BNE mixed up in your head. Use Branch if Not Equal for the Z flag instruction, and Branch if MInus for the N flag instruction. As a memory jogger, remember to "Never think NEgative."

Also, note that the "odds" are different on these different branch instructions. With random results, a BNE gets taken an average of 255 out of 256 times. A BEQ gets taken an average of 1 out of 256 times. But a BPL or a BMI each will get taken, on average, about half the time or 128 out of 256 tries.

If you aren't careful, you can easily miss a BEQ and end up never taking the branch. This happens if, say, you have an odd number in a register and keep counting it down by twos. You go from $01 to $FF and miss the BEQ every time.

Most beginning student's problems with 6502 programs involve not understanding how to do a relative branch or ending up with a branch that goes to the wrong place. If your branch goes to the wrong place, the wrong op code will be picked up on the next instruction, and the program will bomb and plow everything up. It's usually a good idea to single step IF instructions very carefully to be sure they go where you want when you want.

Time for another discovery module . . .

DISCOVERY MODULE 4

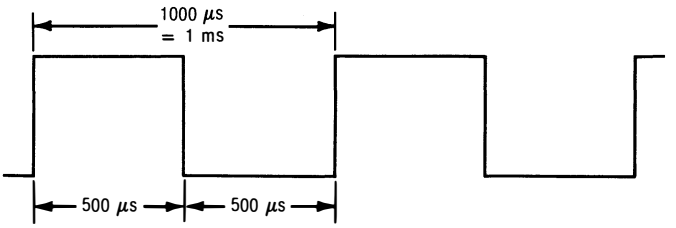
AUDIO TONE

Write a program that puts an audio tone out a port.

Use a pair of loops in your program.

Looks like we are about to build a music synthesizer. We'll guess that "an audio tone" means a 1-kilohertz square wave for now, and that it looks like this . . .

1 KHZ SQUARE WAVE



This square wave has a frequency of 1 kilohertz and a period of 1 millisecond. It is high for half a millisecond, or 500 microseconds, and low for another 500 microseconds.

One obvious way to handle this waveform is simply to add NOPs to our existing ultrasonic square wave of the previous discovery module. To get from 9 microseconds to 500 microseconds takes another 491 clock cycles. We can delay 491 clock cycles with 244 NOP instructions and a compensating JMP. Naturally, we will have to do this twice, once while the clock is high and once while it is low.

The program ends up over 500 bytes long! And, if you think this is bad, just wait till you try doing an 0.1 second square wave or a 1 second time interval. The number of NOPs you need will turn out to be ridiculous.

This "brute force" method is sometimes called *straight line coding* . . .

> **STRAIGHT LINE CODING**—A program in which each instruction gets used only once and in linear order.

Straight line coding ends up being the simplest and fastest you possibly can do but at a terrible penalty in program length. Sometimes, particularly if speed is important, straight line coding can solve problems faster and better than any other method. It may, in fact, be the only possible way to meet stringent timing needs.

But we certainly aren't interested in doing things as fast as we can if we purposely are trying to stall for half a millisecond at a time.

Instead of using 244 NOPs in a row, could we maybe use *one* NOP over and over again 244 times?

To do this, we use a very important programming concept called a *loop* . . .

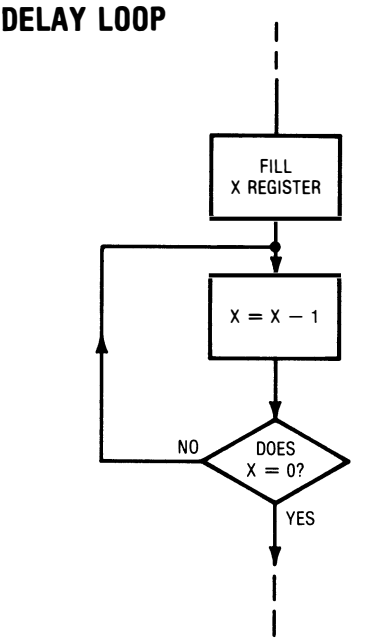
> **LOOP**—A portion of a program in which the same instructions get used over and over again till a task is completed.

Say we fill a working register with some number and then knock one off that number. We then test to see if we got to zero. If we didn't, we knock another number off. We keep this up till we hit zero. Instead of using lots of NOPs to burn up clock cycles, we use a pair of "knock one off and test for zero" commands, a total of five bytes of code.

This particular type of loop is called a *delay loop,* because its main goal in life is to take up a certain number of clock cycles . . .

> **DELAY LOOP**—A loop whose main use is to burn up clock cycles to hit an exact time delay value.

Here is the flowchart for a simple delay loop . . .



What you do is fill a register with a value and then knock one off that value over and over again until you reach zero. The net result is a long time delay that reuses the loop instructions over and over again as long as needed.

Loops have lots of other uses. Some loops search through a text file to print out a message. Some patiently wait for someone to press a key on a keyboard. Some do a calculation exactly the number of times that is needed for a certain accuracy.

Loops are extremely important and are the preferred way of handling many programming problems, since loops can be very compact and can use the same code and the same working registers over and over again.

There are some rules for intelligent use of loops in your programs . . .

LOOP USE RULES

(1) Loops always take longer than straight line coding because of unavoidable overhead time.
(2) Loops almost always need far fewer bytes of code than does straight line coding.
(3) Loops must have at least one exit. Preferably, loops should also have no more than one exit.
(4) Loops within loops within loops are permitted, but one loop should never try to cross another.
(5) It is usually easier and shorter to count a loop down to zero rather than up to some number.

Let's check into these rules in more detail. The reason that loops always take longer than straight line coding is that there usually are some instructions that are needed to set up the loop, such as putting a number into a working register to set the number of loop trips. Even more important, a test is always needed to exit from a loop, and this test always takes time. So, if speed is very important, a loop may not be the best answer.

Sometimes you can speed up a loop by "sharing" the overhead in as many ways as you can. For instance, if your loop only does one task, all the overhead gets charged to that one task. But if you do thirty-two different tasks with your loop, each task gets charged only one thirty-second of that loop's overhead.

If a loop has no exit, you will stay stuck in the loop forever. One way to get stuck is to branch back to the instruction that fills the counter with the initial value, rather than to the location that knocks one off the value each trip. The loop sees the register full, empties by one, checks for zero, and then goes back and fills the

register again, repeating forever. Another way to hang a loop is to check for a value that never shows up, like looking for a zero value when only odd numbers will result.

Some "high level" loops are intended more or less to repeat forever. The service loop in an adventure that first reads the keyboard, then parses the words, then checks the action, and then goes back to read the keyboard again, would continue over and over again until the game is won or until disaster strikes. But even this type of loop should have a Q (for Quit) option or some other means of exit.

It is considered very poor form for a loop to have more than one possible exit. This can happen if you are either checking for a value match or are exiting with a default "no match" value. There are other cases as well. Loops should have one clear and obvious means of exit to continuing code.

If you try this sort of multi-exit looping with a higher level language, you usually end up with a "NEXT WITHOUT FOR" error message or something similar fouling up the works.

Keep your loops simple and modular, with one obvious place you end up after the loop is complete.

You can *nest* loops . . .

NESTING—Putting blocks of code within one another, such as a loop inside a loop, or a subroutine inside a subroutine.

One way to get a very long time delay is to put a loop *inside* a loop. This makes the inner loop go completely through all its paces for each count of the outside loop. This way, you end up with the *product* of the inner loop time and the outer loop trips.

Most micros will let you nest loops and subroutines to any reasonable depth. But you should never let two loops "cross" each other. Crossing happens when you test one loop and branch through the other one. This leads to disaster nearly every time.

So, unless you know exactly what you are doing, always design your loops to have one and only one exit point, and design each loop so it always finishes before it falls back to the next outer nested loop.

We've already seen that you usually do better to count a loop down to zero rather than up to some value. The first and main reason is that it is shorter, easier, and faster to BNE than it is to CMP, or compare against some fixed value. The second is that the loop is "cleaner" if you try to change it, since the "fill" value is at the beginning rather than in the middle of the code. Finally, a count-

down loop always holds the number of trips remaining in it. This can be helpful for debugging.

Okay. We want to use a pair of loops to do a pair of time delays to extend an ultrasonic square wave so it becomes an audio tone. So, what do we count, and how far do we count it?

Now, we could go out ahead of time and calculate exactly what we need, but it is far better to . . .

Rather than do any involved math ahead of time, just stab any old data values into your code and get the code working first.

LAZY = SMART

I've seen it happen time and time again. Someone spends hours meticulously calculating exactly what they think they want and then eventually get around to punching in code. The code ends up not working, and all that time and effort is lost. Or the code shows a much better way to do the same job.

And, again, all that work goes down the drain.

Since the odds are very high that new code will not behave the way you expect it to and that any code at all will teach you something, there is no point in doing involved calculations ahead of time.

We will find out how to get an exact time delay out of a loop after we get the loop working and debugged. This is far and away the best order to do things.

For the counter in a loop, you can count anything countable. This can be a working register or any RAM location in the entire address space. Since the accumulator may be needed for other uses, it is usually not a good idea to use the accumulator for the counter in a loop.

On the 6502, the X register is often a good choice for a loop counter . . .

On the 6502, the X register is often a good choice for the counter in a loop.

You should already have done a card on LDX immediate.

This puts a value into the X register, just as LDA immediate fills the accumulator with a fixed value.

Whatever register or RAM location you use for a loop counter, it must be able to be *incremented* or *decremented* . . .

INCREMENTING—Adding one to the value in a register or RAM location.

DECREMENTING—Subtracting or removing one from the value in a register or RAM location.

For instance, if you have an $06 in the X register, incrementing once gives you an $07, while decrementing once instead drops you to $05. Note that incrementing a register with $FF in it *overflows* to $00, and that decrementing a register with $00 in it *underflows* to $FF, going round and round.

Now, you can increment or decrement almost any register or RAM location by moving a copy of the contents to the accumulator, adding or subtracting one to it, and returning it to the original location. But this is inefficient and sloppy. What you want to do is use a register or RAM location that can be counted up or down "in place" with no hassles.

Another card . . .

DEX	DECREMENT X REGISTER	CA

(IMPLIED addressing)

1 Byte — 2 Clocks

DEX — N and Z flags

Removes one count from what was in the X register. Sets the Z flag on a zero result and the N flag if the MSB is a one. Used to count down the X register, often as part of a loop.

Assume that the X register holds an $06.
2A18– CA Changes X register to $05, clears N and Z flags.

Assume that the X register holds an $00.
2A18– CA Changes X register to $FF, clears Z flag and sets N flag.

You can also decrement the Y register and separately increment both the X and Y registers. The 6502 also has eight very powerful increment and decrement commands that will work anywhere in the address space, using absolute long, absolute short, and indexed instruction modes. Naturally, you can increment a location in the address space only if it contains RAM. A very few I/O devices will also be incrementable and decrementable, provided the target location can both be read from and written to and provided that that location holds and saves the previous value for you.

Always check for compatibility before you try incrementing or decrementing anything in the main address space. Note that decrementing something in the address space takes lots of clock cycles, since you have to get the value, add one to it in the CPU, and then once again replace that value in the address space. For instance, a DEX only takes two clock cycles to do but a DEC absolute long takes six. These read-modify-write DEC and INC instructions are super powerful, but you have to give them enough time to work.

More cards . . .

DOING IT:

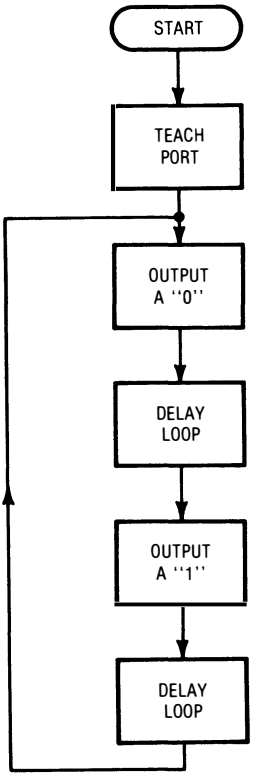
If your trainer is from the 6502 school, complete the DEX, INX, DEY, and INY implied commands and the INC and DEC absolute long cards at this time.

If not, complete all cards for all implied or absolute long instructions that increment, decrement, set or clear working registers or RAM locations.

Interestingly, the older 6502s have no immediate way to increment, decrement, or clear the accumulator. Presumably this was done to encourage you to use the X register as a counter. You can, of course, clear the accumulator by loading a $00 into it in the immediate mode. You can also increment by adding one and decrement by subtracting one, but these take more than one byte and may burn up extra clock cycles.

Here's the flowchart for our 1-kHz audio tone square wave . . .

AUDIO TONE:



And here is the actual code shown on the machine language programming form . . .

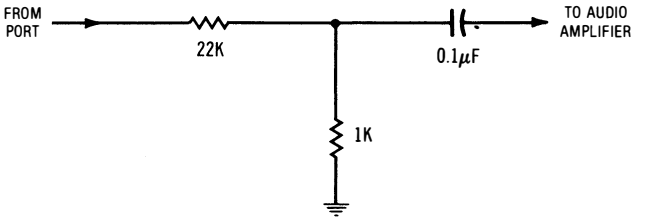
ADDRESS	OP CODE	BYTE #2	BYTE #3	MNEMONIC	HOW?	NOTES
2000	D8			CLD		VERIFY BINARY
2001	A9	01		LDA	#$01	TEACH PORT
2003	8D	80	C0	STA	$C080	" "
2006	A9	00		LDA	#$00	OUTPUT A ZERO
2008	8D	81	C0	STA	$C081	" " "
200B	A2	62		LDX	#$62	STALL 491 µs
200D	CA			DEX		" "
200E	D0	FD		BNE	$200D	" "
2010	4C	13	20	JMP	$2013	EQUALIZE 3 µs
2013	A9	01		LDA	#$01	OUTPUT A ONE
2015	8D	81	C0	STA	$C081	" " "
2018	A2	62		LDX	#$62	STALL 491 µs
201A	CA			DEX		" "
201B	D0	FD		BNE	$201A	" "
201D	4C	06	20	JMP	$2006	AND REPEAT
2020	—					

The code is simply our old square wave code, with extra room made in it for two delay loops, one loop to stall when the output is high, and one loop to stall when the output is low. The three-cycle timing compensation of the "jump to nowhere" probably won't be important or needed for most audio uses, but I have included it anyway.

You can check this code with an oscilloscope. If you have no scope, a voltmeter should read 2.5 volts when the code is running, as it averages out the ones and zeros for you. This assumes a NMOS or CMOS parallel port. A TTL port will probably give you a reading of 1.3 volts or so when the code is running.

But it is much more fun to *listen* to the code running, since it is an audio tone. See if you can't dig up an old mangy test instrument called a *signal tracer* somewhere. If not, add this "listener" probe to any old audio amplifier, cassette recorder with a monitor mode, or whatever . . .

LISTENER PROB:



The reason for this "listener" probe is to knock the 5-volt square wave down to something that won't overload the sensitive input of your usual audio amplifier. The tone should, of course, be there only when the program is running. Try stopping and starting the program to verify that you are actually generating the tone with your software.

A shielded audio cable is the best way to connect this circuit to your amplifier. Otherwise, you may get bunches of power line hum. Be sure the ground lead of the cable connects to the common of the microcomputer.

By the way, you might be able to drive a speaker directly with some ports on some micros, but the results won't be very loud. Try it and see what happens.

After the tone is working, you can go on to get the frequency correct and do any math that is involved. This way, any surprises or new ideas that running the code gives you do not waste your time and effort.

Let's see. A DEX takes two clock cycles, and a BNE takes three clock cycles if the branch is taken, and two clock cycles if not. Apparently, if we are making N trips through the loop, there will be 5N - 1 clock cycles used up, since all but the last trip will take 3 + 2 = 5 cycles and the last trip will take 2 + 2 = 4, and since 4 = 5 - 1.

The LDX will add two clock cycles to the loop time, so the final formula for this particular loop will be 5N + 1 cycles, where N is the *hex* number we put in the X register.

Math details will, of course, change with the number of instructions in the loop and how you use them, as well as the clock frequency and the micro chip in use. Just add everything up as usual.

For our 1-kHz squarewave, and a 6502 trainer with a 1-microsecond clock cycle, we want a delay of 500 microseconds when the clock is high and a delay of 500 microseconds when the clock is low. But, we already used up 2 + 4 + 3 = 9 clock cycles in the code outside the loop with the load immediate, the store absolute, and the jump. This leaves 491 microseconds. Plug in the math . . .

$$5N + 1 = 491$$
$$N = 98$$

So apparently we want decimal ninety-eight trips through the loop to exactly hit a 1-kilohertz square wave. But . . .

DOING IT:

Put a 98 into the loop code following LDX and the frequency ends up way too low.

Why? What did you forget to do?

The correct value, for a 1-kHz square wave using the math above, is $62. Do you see why?

Never forget this all-important detail.

Sometimes you will not be able to hit the precise frequency you are after. This happens if the number of loop trips ends up too high for one count and too low for another. If this occurs, you can sometimes add NOPs to pick up multiples of two clock cycles and JMPs to pick up multiples of three clock cycles, as needed.

A better way to pick up three clock cycles of delay is to do a branch to the next location with the flags correct for taking the

branch. This method is relocatable, but an absolute jump is not. It can be real tricky to pick up one clock cycle of delay. Sometimes you can do this based on the *difference* between a two-cycle and a three-cycle delay. Newer 65C02s do have one-cycle NOPs.

Another thing you can try is making the square wave slightly asymmetrical and taking up different totals of clock cycles on the high and low sides. This will introduce a very slight amount of FM frequency modulation and a very slight amount of second harmonic distortion into your waveform.

Most of the time, though, the micro can give you far more accurate timing than you need. If this tone is for a burglar alarm, what difference does a frequency error of 0.1 percent make? For that matter, what difference does a 20 percent frequency error make?

Always adjust your timing and clock cycles to be only somewhat better than the accuracy of the results you are after. Any more is a waste of time and effort.

The hidden nasties in this discovery module include finding out about IF instructions, calculating relative branches, incrementing and decrementing registers, finding delay values, and using loops.

Before we go on to the next discovery module, we need to pick up details on . . .

THE STACK

Have you ever taken a close look at one of those automatic tray feeders at the cafeteria? . . .

The tray feeder obeys the rule that the last tray on always is the first tray off. We never have to worry about addressing a given tray, for there is always one on top of the stack ready for use.

I have avoided using the term RAM to mean "random access memory," becase just about everything in a microcomputer has random access. All that random access means is that you don't have to climb over the contents of some other addresses to get to the one you are really after. Things were not always this simple. Very old dino computer memories consisted of serial shift registers that went round and round. You literally had to catch what you needed when it came by on the next time around. So RAM today is not the big deal it once was.

But there are times when you might want *not* to have random access. Instead, there is an alternative to random access in most micros called a *stack* . . .

STACK—A computer memory area that acts on a "last in, first out" basis.

The advantage of a stack memory is that you don't have to address it. You simply push things down onto the top of the stack or pull them back off the top as you need them. Using a stack to store and retrieve things in order is short, quick, and easy.

One use of a stack is as a handy temporary stash to hold program values. A stack can also be used to move things between registers, particularly between the accumulator and the processor status or phlag register.

But in their most important use, stacks are untouched by human hands. The CPU uses the stack to remember things it needs to save for later on. For instance, in a subroutine, you have to save the page and position bytes for the address you were on when you started the subroutine. On an interrupt, you also have to remember the return address, but you will want to save all the flags by pushing the phlag register onto the stack as well. If there are other things that you want to save, you can also add these on your own to the essentials saved by the CPU.

Some newer microcomputers have two stacks. One of these is the *system* stack that saves things the CPU needs, while the other is the *user* stack that saves things the programmer wants to hold on to. But most older micros have a single stack that holds, in order, a mix of user data and system saves.

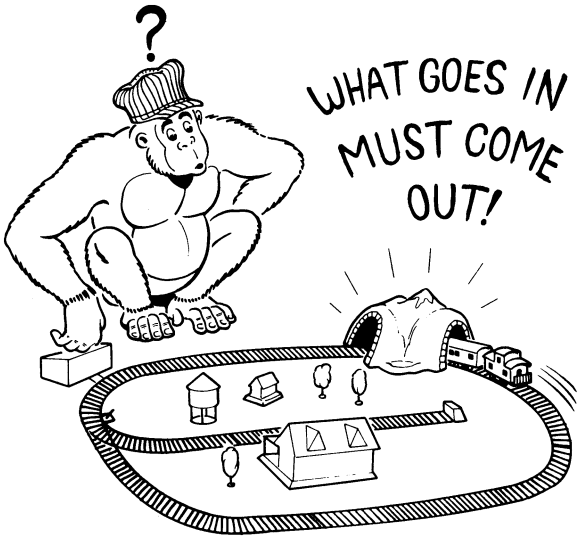
While you could build a separate and fancy hardware circuit that behaves as a "last in, first out" memory, this is seldom done. Instead, an area of plain old RAM is set aside and has its access restricted so that it appears to behave like a stack . . .

Most micros do not have a true last-in, first-out stack.

Instead, an area of RAM is set aside and made to appear like a true stack.

There are several advantages to faking a stack out of RAM rather than using a real "last-in, first-out" memory. The first is that no special hardware is needed. Second is flexibility. Third is power, because you can manipulate the stack both in its intended way, and also by any of the usual address space instructions. A fourth advantage of a faked stack is that you do not have to move any data from location to location. Instead, you simply move a separate pointer to show the new location.

The foremost rule of stack use is . . .



Normally, the only way you have to address a stack is with the time sequence with which you put things onto the stack and

remove them. Thus at all times you must keep exact track of what is on the stack and in what order.

Let's list some other stack rules . . .

STACK USE RULES

(1) Stacks work on a LAST-IN, FIRST-OUT basis.
(2) Stacks must be initialized so you know where they start.
(3) Stacks have a certain capacity. You must not overfill them.
(4) You must not fill a stack faster than you empty it.
(5) You must not empty a stack faster than you fill it.
(6) You must keep track of the order in which things are put on the stack.
(7) The stack must normally be protected from non-stack program access.

Most of these rules are fairly obvious. You use a stack just like the tray feeder at the cafeteria. The last thing you put in is the first thing you get back.

Since in RAM, stacks are usually faked, you always have to know where to start the stack. This stack start is usually done by initializing the stack pointer to some value. More on this shortly.

All stacks on all micros have a maximum stack size or *capacity*. The capacity is just like the capacity of the trays on the feeder. More details on this shortly, But our main concern here is that you are allowed to put only so many values into the stack without the stack overflowing and causing problems.

If you put something on a stack and never take it off, this takes one away from the stack capacity and keeps you from getting at anything older on the stack. Do this often enough, and the stack overflows. Thus, you must be sure to remove every item you put onto the stack.

And you can remove an item only once. If you put things onto the stack and do not use them, the stack destroys by overflow. If you remove things from the stack that you didn't put there in the first place, the stack destroys by underflow. It is real easy to save something on a stack as part of a loop and then exit the loop, leaving the value stuck on the stack on the loop exit. Every time you go through the loop, one more item gets stuffed onto the stack, and the program sooner or later bombs.

If you have a long program that uses the stack only every once in a while, stack overflow or underflow may not take place for a long

time and may be very hard to pin down. Always be sure that when you put something on the stack, it is the first thing to get removed, and that it always gets removed exactly once.

You also have to make sure that what is really on top of the stack is what you think is there. For instance, if you save something on the stack, go to a subroutine, and then get something back off the stack, instead of getting what you thought you saved, you get part of the subroutine address, because the CPU used the stack to save that address for you. This is one big advantage of having separate user and system stacks—the two can be kept separate. Regardless, you always have to be certain you know what is on the stack and in which order.

Finally, you must normally protect your stack from "illegal" or RAM-style access. Any program that writes to the memory area set aside from the stack will bomb the program, since it messes up all the values, particularly return addresses, stored in the stack. You can, however, if you are very careful, use "illegal" stack access to do all sorts of mind-blowing things, like popping a subroutine, or finding out inside a subroutine or interrupt which part of the main program called it.

So much for the rules and when to break them.

stack size and location

To review, a stack is a last-in, first-out memory that is faked by setting aside an area of system RAM. This forms a handy "address-free" stash that is usually available both to you for program use and to the CPU for system purposes, such as saving return addresses for subroutines.

The size and location of a stack depends on the microprocessor chosen and on the system used. On smaller and dedicated micros such as the 8048, the stack is only a few words long, and you are very much limited in what you can put there.

Some microprocessors, including the 6800 family, let you put the stack anywhere in memory and make it any size you want. The advantage of this is flexibility. The disadvantage is that if the stack runs away, it can take the entire machine with it. This type of stack also needs a full width or 16-bit stack pointer, so the stack will be harder to initialize and control as well.

The 6502 uses a nice compromise between these "too much" and "too little" approaches to stack design. On the 6502, the stack can be up to 256 bytes long and always occupies memory page one, or addresses $0100 through $01FF. In normal stack use, you should initialize your stack to $01FF and work your way down in memory as you push things onto the stack.

Thus . . .

The 6502 stack fits on page one of memory and can be up to 256 bytes long.

Usually, the stack is started at $01FF and works down through RAM.

The stack must, of course, be in RAM. Because the stack always goes on page one and because the powerful upcoming absolute short addressing mode can best use RAM on page zero, all 6502 microcomputer systems normally have RAM at the *bottom* of their address space.

A 256-byte stack is more than enough for practically all program uses and, since there is no way the stack can ever leave page one, there is no way it can plow anything but itself.

If you are absolutely sure your stack will be very short, and if you are very careful of your stack use, you can use the low bytes on page one for other uses. But this is dangerous, since a collision of the stack with other locations is certain to plow your program.

Card time once more . . .

PHA	**PUSH ACCUMULATOR ON STACK**	**48**

(IMPLIED addressing)

1 Byte — 3 Clocks

PHA — no flags

Takes a copy of what is in the accumulator and pushes it into the next available stack location. Then decrements the stack pointer by one.

Assume A holds an $06 and S holds an $FD.

28BD– 48 Puts a $06 in location $01FD and then decrements stack pointer S to $FC.

By the way, *pushing* consists of entering something onto the stack, and *pulling* or *popping* consists of removing something from the stack . . .

PUSHING—Entering something onto the stack.

PULLING—Removing something from the stack.

You PUSH things onto the stack and PULL things off the stack. There is also a term, *popping*. Popping a stack is almost the same thing as pulling the stack, but popping is used in a special sense. It means throwing away the things you are taking off the stack so you can get at what is underneath. For instance, if you pop the stack twice on the 6502, you cease being a subroutine and continue just as if you were in the main program.

A dangerous yet powerful technique.

Anyway, PHA simply puts a copy of what is in the accumulator onto the stack. If you want to save the X register onto the stack, you do a TXA and then a PHA, since there is no direct PHX command on older 6502s. The Y register, and anything else anywhere in the address space, can be saved by putting what is to be saved first into the accumulator and then pushing it onto the stack. Thus, anything in the machine can be put onto the stack or removed from the stack.

We use a *stack pointer*, or S register, to keep track of where we are in the stack. The S register is a dedicated use 8-bit register inside the CPU that saves the stack's position address for us. The 6502's CPU always and automatically adds an $01 to the stack pointer address, so that the stack absolute address is always in the range $0100 through $01FF, yet can still be stored as a single 8-bit word.

On the 6502, the stack pointer is normally intialized to $01FF, and the stack works down through memory . . .

On the 6502, a dedicated use S or stack pointer register remembers the next available stack location for us.

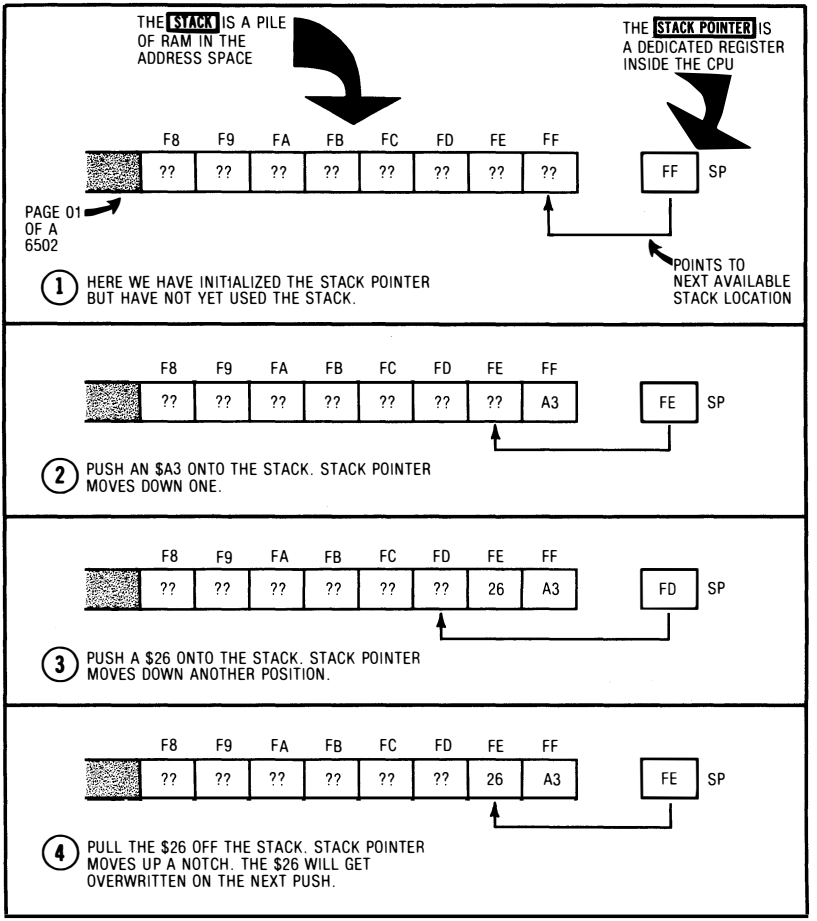
The CPU always adds an absolute $01 to the 8-bit S register so that the stack pointer always points somewhere between $0100 and $01FF.

The stack pointer is normally initialized to $01FF and the stack works its way down through page one.

The stack pointer must never be allowed to go below $00 or above $FF. Going below $00 means you have completely filled all available 256 locations with stack values and have *overflowed*.

Exceeding $FF means you have emptied all locations and have *underflowed*.

Here is how the stack and the stack pointer interact . . .



Remember that the stack is a bunch of cleverly disguised RAM on page one, while the stack pointer is an 8-bit dedicated use register inside the CPU. You initialize the stack pointer to $FF, which becomes $01FF when the CPU gets done playing with it. As you push things onto the stack, the stack pointer decrements itself to $FE, $FD, and so on down. As you pull things off the stack, the stack pointer increments itself to $FD, $FE, $FF, and so on up.

Note that nothing moves around in RAM. Once you shove something onto the stack, it stays in the *same* RAM location until it is overwritten. Only the value in the stack pointer changes.

In fact, even *after* you pull a stack location, the data in that RAM location does not change. It is only when you rewrite to that location with a new PUSH command that any location changes. Thus, your stack may be able to save a partial past history of stack access for you.

You should already have done cards on the TXS and TSX transfer commands. If you want to initialize the stack pointer, you put some value, often $FF, into the X register and then do a TXS. This moves the value from X to S. If you want to find out where the stack is pointing now, you do a TSX, putting a copy of the stack pointer into X. You mentally have to add an "$01" in front of this value to find the actual page one address.

System monitors or other operating systems will usually do this stack initialization for you. But if your program is to be the only one ever run in the machine, then you will have to carefully initialize the stack pointer very early in your program.

DOING IT:

If your trainer is from the 6502 school, complete the PHA, PLA, PHP and PLP cards at this time.

If not, complete all cards for all instructions that push things onto or pull things off of any stack areas.

Also find out how to initialize and read the stack pointers.

Another stack trick that you can do is push things onto the stack with one command and pull them off with another, *moving* data in the process. For instance, if you do a PHA followed by a PLP, you will move a copy of the accumulator into the processor status register, or our phlag register. This lets you simultaneously set all your flags on the 6502. If you do a PHP followed by a PHA, you have moved all of your flags into the accumulator where you can look at them.

One final time. The stack is an area of RAM set up in an oddball way so you can have "address free" or last-in, first-out access. You as programmer can push data values onto the stack or pull them off. The CPU can also push and pull the stack and does so most often to

save return addresses for subroutines and both return addresses and flags for interrupts.

The stack pointer, on the other hand, is simply an 8-bit dedicated use register in the CPU that keeps track of the next available stack location. The 6502's CPU automatically adds a free "$01" to this pointer so that this pointer always leads to an absolute address of $0100 through $01FF. Normally, the 6502 stack pointer is initialized to $FF, which the CPU translates to $01FF. The stack builds down from $01FF, with each pointed location being the next available for use.

Here is our next discovery module . . .

DISCOVERY MODULE 5

PITCH REFERENCE

Create a musician's A-440 pitch reference using a subroutine.

Once again, it's a square wave out a port. It turns out that practically any real-world use of micros will involve square waves and similar waveforms out a port or into one. So hitting away at this fundamental need six ways from Sunday is very worthwhile.

Except for a subtle gotcha, more on which later, all we need do for a musician's pitch reference is to change our existing 1-kilohertz square wave into a 440-hertz square wave. The pitch standard for international note A above middle C is 440 hertz. You can do this simply by increasing the delay values in the earlier program.

But, what is this subroutine nonsense?

Look back over your earlier code. Do you see how we have used exactly the same code *twice*? The delay loop when we are high is exactly the same code as when the output is low. Wouldn't it be nice if we could use the same block of code over again in two different places in our program?

Hence subroutines . . .

SUBROUTINE—A block of working code that can be accessed automatically from several different places in a program.

There are two main uses of subroutines. Subroutines *shorten* programs by letting code be used over. Subroutines *neaten* programs by separating high level code from low level details . . .

SUBROUTINE USES
Subroutines SHORTEN programs by letting code be used over again in several different places. Subroutines NEATEN programs by separating high level code from low level details.

The shortening part is obvious. If we need a delay loop in ten different places in our program, we can instead use one subroutine and ten subroutine calls, greatly shortening the program.

But it is the neatening feature of a subroutine that is far more important. What this neatening does is keep the nitty-gritty details of your code out of the mainstream of the big program. For instance, you can now separate the details of how you delay from the main part of the program that simply wants to delay. With subs, there's no need to rewrite "high level" code whenever some tiny detail changes.

Neatening, of course, doesn't only look nice. It greatly simplifies debugging and developing programs and very much eases documentation and explaining to others how the program works.

So it is not only proper to have subroutines that are called only once by the main program; it is also a darn good idea that should be strongly encouraged.

Naturally, nothing is ever completely free. If you use subroutines, there is some unavoidable overhead in your subroutine call and return that burns up CPU clock cycles. So, using subroutines will always be slower than using straight line coding. If you want speed at all costs, do not use subroutines.

But for practically all other needs, you should use subroutines. And sub-subs. And sub-sub-subs. And so on. Break things up and reuse things where and whenever possible.

Another trick with subroutines is that you can use existing code that is already in the machine. Most systems will already have lots of subroutines inside their monitors that give you delays, ways to output text, ways to handle I/O, means of lighting displays, and so on. By "stealing" these existing *utility* subroutines, you can simplify your code and shorten your program. Even if you can't directly use these subs, you can lift them out, modify them, and put them into your own code with little time and effort.

One mind-blowing use of subroutines involves something called *re-entrant* coding. Re-entrant coding lets a subroutine call *itself* over and over again. This is useful in repetitive calculations, such as series approximations. When you use re-entrant coding, you do, of course, have to provide for an orderly exit and must not exceed the stack capacity. But it's very heavy stuff that can do an awful lot with a surprisingly few bytes of code. FORTH freaks are often very much into re-entrant subs . . .

RE-ENTRANT CODE—Any part of a program that reuses itself as one of its own subroutines.

A very dangerous and tricky concept that can dramatically shorten programs.

Subroutines are the greatest.

You use a subroutine by calling it from your main program. A subroutine call differs from a jump in that the jump, when taken, never expects to return. But, on a subroutine call, the CPU very carefully keeps a record of where you are in the main program by stuffing the program counter onto the stack. Later, when the sub is finished, the return process will look at the stack and automatically know where to return.

A card . . .

JSR	**JUMP TO SUBROUTINE**	**20**
	(ABSOLUTE LONG addressing)	
3 Bytes		6 Clocks
JSR $236A		no flags

Jumps temporarily to subroutine whose low or position address is shown by the second byte and whose high or page address is shown by the third byte. Used for subroutine access.

2004– 20 6A 23	Jumps temporarily to 236A for subroutine code. Resumes at $2007 after normal sub return.

The JSR command needs more clock cycles than most other instructions, since it has to remember where to return to after the subroutine is complete. To do this, the JSR instruction pushes the program counter high and the program counter low onto the stack and then resets the program counter to the starting address of the first op code in the subroutine.

Note that the CPU saves *only* the return address for you. If you want to save anything else, such as the flags, the accumulator, the X register, or the Y register, or any address space value, you have to do it yourself with extra code.

To get back from a subroutine, we use an RTS, short for ReTurn from Subroutine. Here's the card . . .

RTS | **RETURN FROM SUBROUTINE** | **60**

(IMPLIED addressing)

1 Byte — 6 Clocks

RTS — no flags

Returns automatically to the next instruction in the main program that called the subroutine. Fully automatic.

2AD4– 60 Automatically returns to next instruction in the main program that called this subroutine.

The RTS is a 1-byte implied instruction that is fully automatic. The CPU looks into the stack to find the return address and then goes to the next instruction in the calling program.

An obvious rule . . .

ALL subroutine code must ALWAYS end with an RTS.

CAN'T GO BACK WITHOUT A ROUND TRIP TICKET!

If you do not end your subroutine with an RTS, the program stays in the subroutine and moves down one address level in the stack as well.

Now, it is possible to have different *entry* points in a subroutine. Sometimes, you will let one subroutine *fall through* to another. But you always must use an RTS when you are finished with your subroutine code and want to return to the main program.

DOING IT:

If your trainer is from the 6502 school, complete the JSR and RTS cards at this time.

If not, complete cards for all instructions that access and return from subroutines.

Let's look at some sneaky tricks you can do with your subroutine code.

If you are having problems debugging a program, just put an RTS temporarily as the first code line in your sub. Then single step. If the main program works, then the problem is in the sub, and vice versa. This separates the "high level" problems from "detail" ones. Be sure to remove this "immediate return" when you are done with it, or it will haunt you forever.

Be careful if you are single stepping a subroutine all by itself, for when the RTS comes up, the CPU may not know where to return to and may put you into the monitor or plow things up for you. Always stop at the RTS if you did not JSR to the sub in the first place.

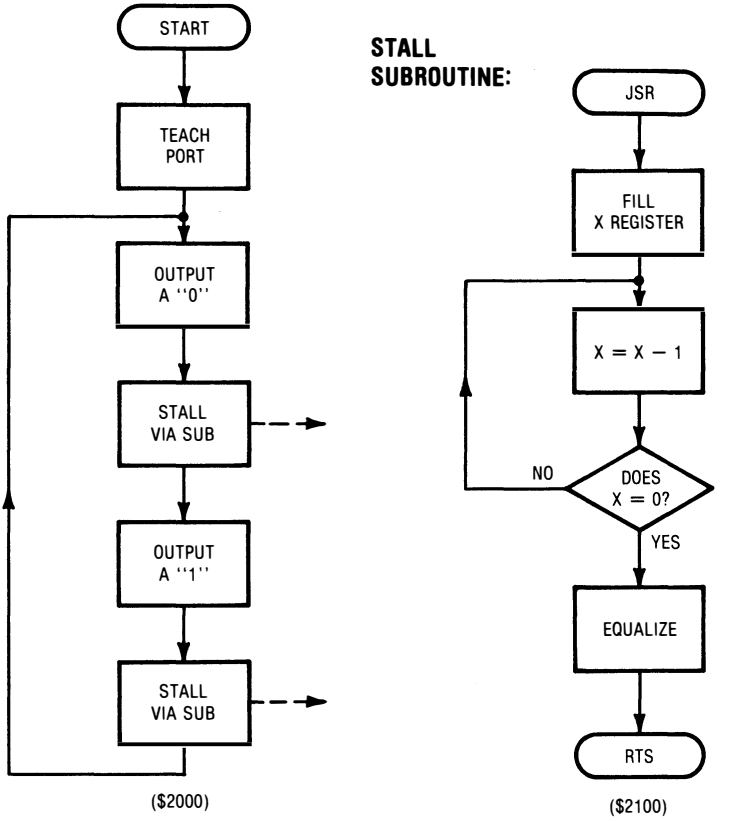
If you are in a subroutine and want to know which part of the program called the sub at the present time, check the top two stack locations for the calling address page and position. You can also pop these two locations to continue as if you were back into high level code, or change them to return somewhere else.

Force feeding return addresses on a subroutine return can be a very powerful indirect addressing scheme, one that even will work on a micro that normally cannot handle indirect addresses. To do this, shove the position byte and then the page byte that you want to go onto the stack. Then do an RTS. You end up at your new address at the same code level you started with. Note that the two pushes and the two pulls of the RTS cancel out, leaving the stack as it was originally.

When you are first writing a program, it's usually a good idea to put each subroutine on a separate page of memory, well away from the main program. This lets you change details of one sub without affecting any of the others. Later on, you can reassemble everything into minimum address space, but there is no point at all in doing this till after you are sure you have all the code working exactly as you want it to.

Here is the flowchart for our pitch reference module . . .

PITCH REFERENCE



Note that our flowchart is now in two pieces. We have the high level flowchart that includes a box called "STALL VIA SUB." And we have the low level flowchart that gives us full details on how we

use a subroutine to do a delay loop. When you look at the code for this sub, you will recognize the same loop that we had before, except for some extra NOPs needed to hit an exact delay value.

We have also added, in parentheses, addresses that show which code goes where in the machine. This helps in relating code sheets to flowcharts, but it also means you have to redo the flowchart if you move the program somewhere else.

Our program code is now in two pieces. We will put the program on page $2000 and the subroutine on page $2100. Here is the code for the program . . .

MAIN PROGRAM

ADDRESS	OP CODE	BYTE #2	BYTE #3	MNEMONIC	HOW?	NOTES
2000	D8			CLD		VERIFY BINARY
2001	A9	01		LDA	#$01	TEACH PORT
2003	8D	80	C0	STA	$C080	" "
2006	A9	00		LDA	#$00	OUTPUT A ZERO
2008	8D	81	C0	STA	$C081	" " "
200B	20	00	21	JSR	$2100	STALL VIA SUB
200E	4C	11	20	JMP	$2011	EQUALIZE 3 µs
2011	A9	01		LDA	#$01	OUTPUT A ONE
2013	8D	81	C0	STA	$C081	" " "
2016	20	00	21	JSR	$2100	STALL VIA SUB
2019	4C	06	20	JMP	$2006	AND REPEAT
201C	—					

And here is the subroutine code . . .

SUBROUTINE

ADDRESS	OP CODE	BYTE #2	BYTE #3	MNEMONIC	HOW?	NOTES
2100	AD	E1		LDX	#$E1	DELAY 1111 µs
2101	CA			DEX		(VIA LOOP)
2102	D0	FD		BNE	$2101	" "
2104	EA			NOP		EQUALIZE 4 µs
2105	EA			NOP		" "
2106	60			NOP		AND RETURN
2107	—			RTS		

The subroutine code is nothing but one of our earlier delay loops, modified with extra NOPs, and followed by the RTS needed to return to the main program. Instead of rewriting the code twice in the main program, we write the delay loop once in a subroutine, and then call the sub twice in the main program.

Be sure to load *both* the program and the subroutine in your micro when you test or use either of them. If you jump to a subroutine address and there is no subroutine there, the program will continue as if a sub really was there and will start executing nonsense code, eventually plowing up the works.

In this particular example, we just break even on total bytes using our subroutine, compared with writing two separate delay loops. This happened here because the sub is very short and is used only twice, and because there are those new overhead bytes needed to call and return from a sub. Had we needed the sub in three places, or had the sub been longer, we would have been ahead on code length. Your average subroutine will usually save you bunches of bytes of code.

We also have definitely neatened up our program and separated the high level code from the delay details, so we gain much on structure and appearance even if we only break even on bytes.

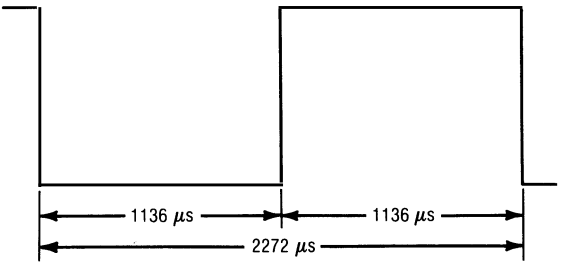
timing details

There is one extra complication in timing a program that uses subroutines.

You have to allow six CPU cycles for each subroutine call and six CPU cycles for each subroutine return, for a grand total of twelve extra cycles.

Let's look at a 440-hertz squarewave again . . .

STANDARD PITCH A-440:



The exact time period of a 440-hertz square wave is 2272.72 microseconds, or 1136.36 clock cycles per half cycle. But the closest we can hit this is 1136 cycles on a trainer with a 1-microsecond CPU cycle time. Is this accurate enough?

Musicians speak of a *cent* as one hundredth of a *semitone*. A semitone is the pitch difference between two adjacent piano keys. A 1-cent musical pitch accuracy is roughly 0.06 percent in frequency. Only the very best musicians can recognize a 1-cent difference, and a 3-cent pitch accuracy is usually tolerable.

In our case, the pitch is high by 1136.36/1136 or 1.0003. This is an error of +.0003. Converted to percent, this is 0.03 percent, or half a cent in pitch and twice as good as the best musician.

So, a delay of 1136 cycles per side will be more than good enough. The high level program code takes nine cycles, and the subroutine overhead takes twelve, so our loop inside the subroutine must take 1136 − (9 + 12) = 1115 clock cycles.

The loop formula is still 5N + 1 cycles, but we can't hit 1115 exactly. Let's try for 1111 clock cycles . . .

$$5N + 1 = 1111$$
$$N = 222$$

Of course, that is in decimal, so we convert to hex and end up with a loop value of $DE.

Since we are four cycles shy, we throw in two NOPs for good measure, and we precisely hit what we need, a nearly perfect 440-hertz square wave out a port for use as a pitch reference.

a lesson learned the hard way

But surprise, surprise. You proudly haul your pitch reference off to a musician and he sneers at it! He also mumbles something vague about the timbre being "too bright" or some such nonsense.

Later, after cooling off, you find a musical type who is also scientific. He explains that a pitch reference should be an exact sinewave, since any higher harmonics will mask the true pitch. This is particularly sticky on instruments like the piano. The overtones on a piano are not true harmonics because of the string lateral stiffness, and its overtones are sometimes as strong as or stronger than the fundamental.

The lesson learned the hard way is . . .

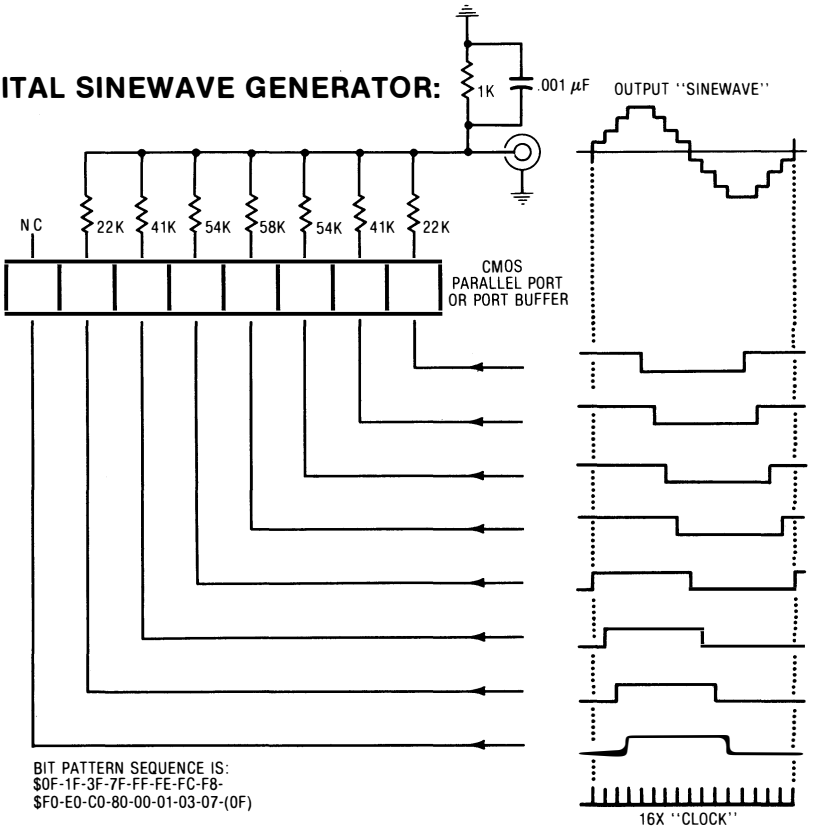
No matter how "perfect" your program is, if the final user doesn't like it, it is utterly and totally worthless!

So back to the drawing board. Since this is a single-frequency square wave, you can probably tack three resistors and three capacitors onto the end to form a simple RC low pass filter that gets the harmonics down to something liveable.

But, naturally, once your musicians actually use your pitch reference, they will want other notes for other instruments or different piano strings. So a heavy brute force filter may not be the answer.

Instead, you can digitally synthesize sinewaves and route them out your port. Here is one waveform you can generate and the output port resistor network involved . . .

DIGITAL SINEWAVE GENERATOR:



What you do is take all eight port lines and provide just the right pattern in just the right way to synthesize a good sinewave. This particular one will have its first two "loud" harmonics as the fifteenth at 1/15 amplitude and the seventeenth at 1/17 amplitude,

both of which are easily stomped into nothingness with a small capacitor. You "clock" the pattern, or advance the waveform, at sixteen times the desired output frequency.

Full details on this appear in the *CMOS Cookbook* (Howard W. Sams 21398).

There are some upcoming shift and rotate instructions that you might like to use for a sinewave generator. You can also use files and a D/A converter, rather than custom resistor values for table lookup.

Let's do it . . .

DOING IT:

Create a musician's pitch reference that a musician can actually use.

HINT: Go find a real, live musician before you start.

Remember, while doing all this, that you are competing with a $2.40 pitch pipe and had darn well better have more to offer.

ABSOLUTE SHORT ADDRESSING

Now is probably a good time to explore *absolute short addressing*. Many microcomputers have one or more ways to reach a small and known area of the total address space with commands that are both faster and shorter than absolute long. Absolute short addressing has speed and program length advantages but often limits itself to one area of memory and can cause turf fights over valuable locations.

On the 6502, one important use of absolute short addressing is to make available another 256 locations that are almost as fast and easy to use as the working registers in the CPU.

The 6502 school lets their absolute short addressing go by the name of *page zero* addressing . . .

PAGE ZERO ADDRESSING—A type of absolute short addressing that assumes the address is on page zero, or somewhere in absolute address space locations $0000 through $00FF.

Page zero addressing is also available in the 6800 micros, where it goes by the grossly misleading term *direct* addressing, whatever that means. Sounds good on the data sheet, though.

Almost every 6502 instruction provided in absolute long addressing is also available in absolute short or page zero addressing.

With page zero addressing, you need only the position byte following the op code, since the leading "$00" is assumed by the CPU. Thus, page zero instructions are only two bytes long rather than three.

Also, since the CPU doesn't have to wait around to look at the third byte of the op code, page zero instructions usually execute faster than absolute long instructions by at least one clock cycle.

Another card . . .

STA **STORE ON PAGE ZERO** **85**

(PAGE ZERO addressing)

2 Bytes 3 Clocks

STA $36

no flags

Puts a copy of what is in the accumulator into the page zero location shown by the second byte. Shorter and faster than absolute long addressing.

Assume the accumulator holds an $6F.

2CB3– 85 36 Stores a $6F in location $0036. No flags are changed.

Note that the assembler notation is the mnemonic followed by two hex numerals. The STA page zero command workings are obvious, but the related page zero LDA command seems to cause newcomers all sorts of problems.

The LDA page zero command has an assembler format of LDA $35 and an 85 op code. This command takes a copy of whatever happens to be in location $0035 and puts it in the accumulator. The data value put in the accumulator can be any 8-bit value at all that happens to be in this address location, ranging from $00 to $FF.

On the other hand, the LDA immediate command has an assembler format of LDA #$35 and an A9 op code. This command uncon-

ditionally puts the value hex three-five, or decimal fifty-three into the accumulator. A reminder . . .

DON'T get LDA immediate and LDA page zero mixed up!

LDA immediate puts an 8-bit value into the accumulator.

LDA page zero goes to an address and puts a copy of whatever it finds into the accumulator.

In assembler notation, immediate always MUST have a # symbol in front of the operand!

Watch this particular detail very carefully. One place it shows up is on the AIM-65 trainer where you can just punch in programs in assembler notation. If you forget the "#" in front, all your immediate loads become page zero loads with variable rather than fixed values.

More cards . . .

DOING IT:

If your trainer is from the 6502 school, complete the page zero versions of all of your present absolute long addressed cards at this time.

If not, go through your existing cards and do a new one for every card that has available absolute short or other simplified addressing modes.

If you have nothing comparable to page zero addressing in your particular micro, then the micro will have some other way of doing interesting addressing. You might check into register indirect addressing at this time or otherwise explore the op codes to find alternate ways of reaching the address space that are both faster and shorter than absolute long.

Our next module will use absolute short addressing . . .

DISCOVERY MODULE 6

".Y" TIME DELAY

Write a subroutine that provides a time delay of first one-tenth second and then write another one that delays *.Y* or "*Y*-tenths" second.

Demonstrate your subs first with a 5-hertz square wave and then light an LED lamp for 0.3 seconds on and 2.0 seconds off.

There are several new things to pick up in this module. One nasty involves using a new addressing mode, page zero for the 6502. Another involves a "loop within a loop" and finally a triple-nested loop. We also have to figure out how to pass a variable to a subroutine and, finally, how to make code easier for the "high level" user to apply.

What is wrong with our plain old delay loop? This loop is fine for fairly short time delays, but it can't reach a tenth of a second. No way.

Let's see why. The absolute maximum value we can put in our delay loop is $FF for 255 delay cycles, right?

Wrong.

We can get super sneaky and do very slightly better than that. If you put a #$00 as your loading value into your loop, you immediately decrement it to $FF, take the branch, and round and round you go. Thus, we can have a full 256 trips around the loop.

We saw that a simple delay loop takes 5N + 1 cycles, so, plugging in the math, our total delay in this case is

```
(5 * 256) + 1 = 1281 clock cycles
```

Add this to the 9-cycle square wave main program time and the 12-cycle subroutine overhead time, and we end up with a flat-out 1302 clock cycles per square wave half, or a total square wave period of 2604 microseconds. This translates to a minimum frequency of 384 hertz, which is bunches above our 5-hertz goal.

So the math tells us we can't hack a tenth of a second with a single delay loop. Now we could add NOPs or whatever inside our

loop to make it longer, or we could use our loop over and over again by repeating the code two, three, or more times. But these will only buy a little more time, not the major increase we need.

Instead, we will use a *loop-within-a-loop*, or a nested loop . . .

> **LOOP WITHIN A LOOP**—A nested pair of loops. The inside one has to go completely around for each count of the outer loop.

If you do one loop and then do a separate second loop, you end up with the *sum* of the two loop delay times. But, if you put a loop inside a loop, you end up with the *product* of the inner loop delay time, multiplied by the number of trips around the outer loop. Products, of course, rack up much faster than sums. Instead of 1281 clock cycles for a loop, we can get over 256 * 1281 clock cycles, or over a one-third second delay by going this route.

Time out for a side trip . . .

"user friendly" code

Two very important things to work toward whenever you do any microcomputer programming is to make the code *user friendly* and *designer friendly* . . .

> **USER FRIENDLY CODE**—Programs that are as easy as possible for the final user to put to use.
>
> **DESIGNER FRIENDLY CODE**—Pieces of programs that are as easy as possible to put together, adopt, and modify.

You have probably heard lots about user friendly code. For a good example of what *not* to do, just take any public domain program out of any old user library and study it.

User friendly code must, first and foremost, be written in machine language so that it can most fully utilize all of the available resources of the target microcomputer in the most flexible and creative way at the fastest possible speed. Check into the top thirty programs for any personal computer, and you will find almost all of the greatest, best, and most profitable programs will be written

either totally in machine language or will make very extensive and creative use of machine language subroutines and modules.

So the main reason to write user friendly code is that it will make you filthy rich.

User friendly code is just that. It interacts with the person running the program in the simplest, most convenient, and most effective way. This means you must use sound, color, and graphics anywhere they can help and improve the program but not to the point where they become obnoxious or tedious. The user should need only an absolute minimum of keystrokes. Extra RETURN or ENTER keyhits are so much of a no-no these days that we won't even talk about them.

User friendly code should normally be menu driven, so that the user can move from one selection to another in the simplest and most obvious manner. The code must be totally self-prompting and self-tutoring. The code must be consistent. What happens in one part of the program should also happen in the same way in other parts of the program, and those ways should be reasonable and predictable.

Naturally, user friendly code must be unlocked and available to the user for backup, modification, and customizing. User friendly code should provide all source code to all users. In fact . . .

The ONLY justifiable reasons for NOT giving a user a completely unlocked program and complete source-code documentation are—

1. **You are so ashamed of the code that you don't want anyone to see it;**

OR

2. **You have so blatently overpriced your work that you don't want anyone to laugh at you.**

Error trapping on user friendly code must be complete. No matter how stupid or dumb the input or how random the keystrokes, the code must safely and sanely return to a reasonable point in the program. Under NO conditions should the final user EVER be presented with a cryptic computerese notation: "SYNTAX ERROR LINE 5436," or "DOS CODE ERROR $08." And under NO circumstances should the final user be dropped into the monitor or into some high level language without a full and polite explanation of what happened and why.

To end up with user friendly code, the micro has to be *transparent*, in much the same way that a great book is transparent if it transfers ideas or thoughts to you without calling attention to the ink and the pages and the words. More ideas on transparency appear in *The Incredible Secret Money Machine* (Howard W. Sams 21562). The microcomputer should be involved only in relating to the user and solving a problem for the user. The micro should not, under any circumstances, bring to the user's attention that it is in fact a computer and that a computer is involved at all.

Many of the user friendly software rules are survival skills, for any software that is not user friendly is certain to get unbought out of existence.

Designer friendly software is a somewhat different concept, and not too many people are yet paying much attention to it.

When you build up a program, you normally build it up out of machine language program modules. These modules should let you work with the highest level system concepts at all times. This in turn, makes the final 'high level' program as easy to put together and use as can be.

So far, we have used only simple modules. What can we do to make these modules as useful as we can in final programs?

There are lots of things that will help. First and foremost is to use lots of subroutines. Subs neaten and separate the high level code from the low level code in any application. Changes in details don't change the main program this way when you use subs. Each sub should have one or more obvious entry points and one—repeat, one—and only one exit. The state of all registers and all flags in the machine must be known and consistent on sub exits.

Data files are another way to build designer friendly code. We will see much more on data files shortly. What a data file lets you do is put anything changeable into a separate block. To change the program action or results, you change only the data file entries instead of mucking around with the code. An obvious example is an adventure. If you use extensive data files, you can simply change the files to create a whole new program without any low level code debugging or changes.

A third and most important route to designer friendly code is to use high level system concepts when and wherever you can. Make the subroutines and modules work in ways that the high level designer can use directly.

A big example will drive this home. Suppose we need a long time delay in a final program, such as a 10.2-second delay. We could put a custom loop inside a loop inside a loop to give the needed 10,200,000 cycles of delay, and it certainly would work. But any time

we needed a long time delay after that, it would be back to square one. We would have to redesign everything.

Suppose, instead, we build a subroutine that gives us exactly so many *tenths* of a second of time delay, rather than so many machine clock cycles. Wouldn't this be much easier for the high level designer to use? The designer simply would rip off the module and pass the number of tenths of a second of delay needed to the sub.

That fast and that easy.

A rule . . .

> **For designer friendly code, make all code modules work with "high level" concepts that make sense to the final program designer.**
>
> **A time delay that stalls in 0.1-second units, rather than so many machine cycles, is an obvious example.**

The reason for making all your modules work with "high level" concepts and constants is that it makes the final program much simpler and easier to debug. The final program designer does not have to go clear back to square one and mess around with individual machine language instructions. Instead, the designer picks up building blocks that can be used directly to do system-level things conveniently and quickly.

Designer friendly code is also easy to adapt to new problems and easy to modify for custom changes.

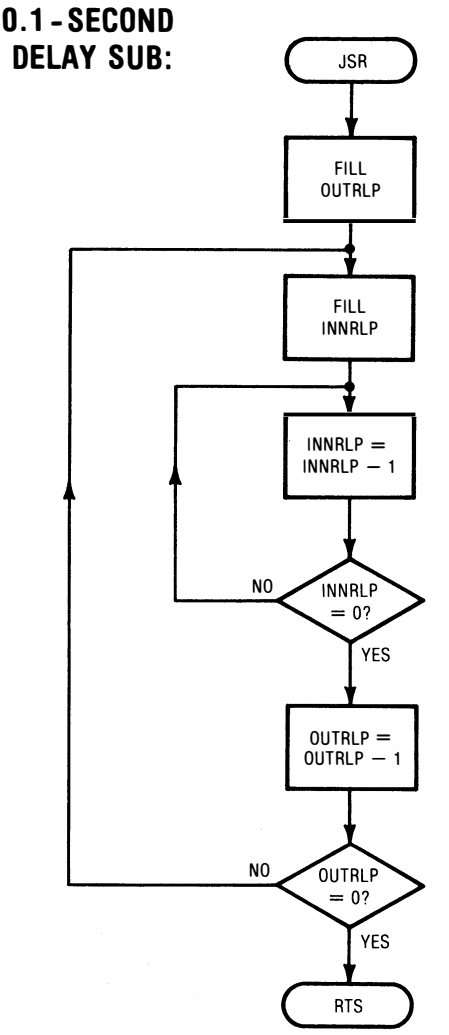
So we will find out how to do a 0.1-second time delay, and then we will adapt this to a *.Y* or *Y/tenths* of a second delay. And anytime you need a long time delay in any high level program from now on, you simply rip off this module rather than going back through all the gory details.

Naturally, there are lots of different ways to con a microcomputer into delaying a tenth of a second. One way involves interrupts. You can let the power line interrupt your program sixty times a second and count every sixth cycle. Some micro peripheral chips include *timer* circuits that can do the same thing for any desired time interval. Special *real-time clock* chips are also available for date and "people time" entry. You can also look around in the existing monitor and see whether there is any delay code utility subroutine ready to go. Another way is to use the accumulator and the stack together

to give a delay loop. The WAIT code beginning at $FCA8 in the Apple II monitor is an elegant example of this.

In this module, however, we will explore using a loop within a loop of page zero variables for our 0.1-second delay, and we will then later count out *Y* of these tenth of a second delays.

Here's one possible flowchart for a 0.1-second delay subroutine . . .



We could use the accumulator and the X and Y registers together to handle a long delay, but chances are there are more important things for these working registers to do. So, we will set aside two locations down on page zero using absolute short addressing. We will call these two locations OUTRLP and INNRLP for "outer loop" and "inner loop." We will make the inner loop go completely around once for each *single* count of the outer loop. The outer loop will go once around for the *entire* delay, which will give us the *product* of the inner loop time and the number of outer loop trips. This product we will then adjust to 0.1-second.

In our MYTH-1 trainer, we'll assume that location $50 can be used for INNRLP and $51 for OUTRLP. On your trainer, you will have to find "safe" locations for your absolute short variables. Note that we put the faster running, or "less valuable" variable first, and the slower or "more valuable" variable second, so we are consistent with the "position-page" coding used in absolute addresses and other 16-bit stores. Always try to have as much consistency in your programs as possible.

Here's some code . . .

0.1-SECOND DELAY SUB

ADDRESS	OP CODE	BYTE #2	BYTE #3	MNEMONIC	HOW?	NOTES
2100	A9	6F		LDA	#$6F	FILL OUTER LOOP
2102	85	51		STA	$51	" " "
2104	A9	70		LDA	#$70	FILL INNER LOOP
2106	85	50		STA	$50	" " "
2108	C6	50		DEC	$50	INNER MINUS ONE
210A	D0	FC		BNE	$2108	INNER EMPTY ?
210C	C6	51		DEC	$51	OUTER MINUS ONE
210E	D0	F4		BNE	$2104	OUTER EMPTY ?
2110	60			RTS		DONE -- RETURN
2111	—					

In this delay subroutine, we first "fill" both OUTRLP and INNRLP with magic numbers that will give us the correct total delay. Remember, when you are first debugging code, it is best just to punch in any old number and get the code working more or less the way you want it to. Later, after the code is debugged, you can go back and change the magic numbers to exactly what you need.

The reason for this two-step process is that any calculations you make ahead of time will probably be wrong anyway and end up a waste of time, particularly if you change the code or find problems with it. Again . . .

At the start of a program, just punch any old reasonable numbers into locations that set timing values.

Save the exact values until after you are sure the code is working and doing what you want.

And, as a reminder, don't forget that you can simplify and shorten things for debugging. For instance, you can temporarily replace the first op code of a subroutine with an RTS command. This lets you find out whether a problem is in the main code or in the sub. And you can put very low magic numbers into loops, such as #$01 or #$02, so you can watch the loops go around only once or twice, rather than making hundreds of trips.

Anyway, we fill OUTRLP and INNRLP with the magic constant values. Then we start counting INNRLP down, just as we did with the previous single timing loop. When INNRLP hits zero, we knock one count off OUTRLP and then reload INNRLP and count it all the way down again. The process repeats over and over. Finally, when OUTRLP hits zero, we are done with the subroutine and exit to our main program. This can be our 5-hertz square wave program, or any other that needs a 0.1-second delay.

Note that the inner loop goes completely around for each single decrement of the outer loop. This gives you the product of the inner loop time multiplied by the number of trips set by the outer loop. Note also that the loops are in fact nested and do not cross.

DOING IT:

In this subroutine, one relative branch value is $FC. In the previous single-loop delay sub, the relative branch value was $FD, yet both programs only back up one instruction at this point.

Why the difference?

Always be sure you know EXACTLY where your relative branches go to. If a relative branch misses your op code and hits an operand, it will try to execute the operand as if it were a legal op code and

then will proceed to whomp off into left field and bomb everything. Naturally, the CPU is doing exactly what you told it to do.

You can also get into trouble by looping to the wrong address. For instance, if you loop back to the TEST instruction, you will keep testing forever with no change in results. If you loop back to the FILL instruction, you will *never* exit the loop, since you fill it, knock a count off, fill it again, knock a count off, and so on forever.

To do a 5-hertz square wave, just use the pitch reference main program with this new "0.1" subroutine. A 5-hertz square wave will have a period of 0.2 second, so it will be high for 0.1 second and low for 0.1 second.

Several points. Be sure to defeat the "auto" triggering on any scope when you view slow waveforms or you will not be able to lock to what you want to see. A 5-hertz square wave on a meter should have obvious kicks to it and any old VOM can be used for a test. If you use a speaker or a signal tracer, you should get continuous clicks that sound something like the rotary dial on a telephone.

timing again

How do we find out how long a loop inside a loop will take?

Obviously, we count up all the clock cycles and multiply that number by the time per clock cycle to get the total delay. But the math looks messy.

Ugly, even.

Any time and any place you find messy math, ask yourself if some quick and dirty *approximation* will do the job instead. Even if it won't, *always approximate first* to be sure your final answer is reasonable.

And it pays to never spend time or effort being more accurate than you really have to. If our 0.1-second delay is for a traffic light or a sprinkler system, being off by a percent or two is no problem. If it is to track sidereal time for a telescope, you'll need to be a lot more accurate.

Generally, however, things other than the microcomputer op code will gang up on you and ruin "perfect" accuracy for you. For instance, if you are using the power line interrupt method for timing, you will end up with a short term accuracy of 0.1 percent or so, no matter how accurate your math is. The actual frequency of your clock will also get into the game. F'rinstance, the Apple II normally has a clock cycle that's only 0.978 microseconds long, rather than a full microsecond. Forget this timing detail and all frequencies end up low by 2.2 percent.

Any crystal on any trainer will have some tolerance. Don't expect better than .05 percent accuracy unless you calibrate the crystal against a standard like WWV or unless you are using an expensive,

precision crystal and temperature regulating oven. A few very low cost trainers do not even use a crystal. Instead they use a clock oscillator whose frequency is set by a resistor and a capacitor or by a ceramic resonator. If you have one of these, don't expect much in the way of accuracy or unit-to-unit repeatability.

I guess what I am really saying is . . .

> **Never make things more accurate than they really need to be.**
>
> **Never waste your time and effort on things that won't matter anyway.**

Let's look at three different methods of calculating the OUTRLP and INNRLP values, in order of increasing hassle and accuracy.

Method 1 on calculating magic numbers is to punch any old value in and then change the numbers around by comparing the results against the time base on an oscilloscope. Three percent accuracy and no hassle. Or count the loop over and over again and compare 200 trips against 20 seconds on a kitchen clock.

In Method 2, we approximate the results without going into the gory details. A quick look at the inner loop code tells us that we have a 3-cycle branch taken time and a 5-cycle decrement time, for a total 8-cycle inner loop time. This is different from the single loop time of five cycles since we are decrementing a page zero location rather than the X register. You can decrement the X register very quickly in two clock cycles since it is inside the CPU. But to decrement a page zero location, you have to go to page zero, get the value, put the value in the CPU, decrement the value, and then return the new value back to page zero again. All of this assumes, of course, that there is RAM in the page zero location to be decremented.

Anyway, if there are N counts in the inner loop, we will use up roughly 8N clock cycles in the inner loop. The outer loop won't normally take up much of the time, so we can often ignore this "overhead." Then, if there are M counts in the outer loop, our total delay time will be slightly more than 8*M*N.

Continuing with our quick and dirty timing approximation. If 8*M*N is to be 100,000 clock cycles of 1 microsecond each for a total of 0.1 second, then M*N will be 12,500 cycles. Now, within limits, we can have any values of M and N as long as their product equals 12,500 and the values of M and N range between 0 and 255. A quick thing to do is to let M equal N and then, if M * N = 12,500,

either value will be the square root of 12,500, or decimal 112. This equals hex $70 and will give an approximate delay of 12544 * 8 = 100352 cycles. Just for kicks, we will knock one count off the one loop to get under 100,000 cycles and use $70 for the INNRLP magic number and $6F for OUTRLP.

Method 3 should be used ONLY if you are sure you need a precise number of clock cycles. This rarely happens. One place where I ran into it is in exact field sync for the Apple II, where you need to delay exactly 17030 clock cycles to get to the next field, no ifs, ands, or buts. Details on this in Enhancement 4, Volume 1, of the *Enhancing your Apple II* series (Howard W. Sams 21846).

At any rate, if we have to count clock cycles, we have to count clock cycles. The inner loop actually takes 8N − 1 cycles for the loop and five cycles to fill the inner loop magic number. The filling process needs two cycles for an immediate load and three for an absolute short store on page zero. Thus our total inner loop time is apparently 8N + 4 clock cycles. Now if there were no inner loop, the outer loop time would use up 8M + 4 clock cycles. Add the inner loop and we have to reuse the inner loop M times. And, we have to allow for the twelve clock cycles needed for the JSR and RTS overhead. So, our total loop time will be . . .

$$\text{Loop Cycles} = 8M + 4 + M(8N + 4) + 12$$

which simplifies to . . .

$$\text{Loop Cycles} = 8MN + 12M + 16$$

Now this looks a lot like our quick and dirty approximation of 8*M*N. Note that the first term is ridiculously bigger that the second term, which in turn is very much bigger than the third term. That is what good approximations are all about. If we try our values of M = 111 and N = 112, we get a total number of loop cycles of 100,804. This is within a tenth of a percent and is good enough for many uses.

Note that our 5-hertz square wave will be even slightly lower in frequency, since we haven't subtracted the time needed for the "high level" code instructions. In this example, the difference will be utterly negligible, and besides, our goal is to provide a subroutine with exactly 0.1-second delay.

But what if we have to be exact? In that case, you have to keep trying different values of M and N to find either the exact value you want or some number slightly below it. You then make up the difference by throwing in NOPs or whatever to hit things exactly.

It is a lot of fun to write a program that calculates values for you and then runs through all the numbers it knows. To save on time and paper, you tell your program to give you numbers only near the final values you want.

Try it . . .

DOING IT:

Write a program for a personal computer that calculates the exact OUTRLP (M) and INNRLP (N) values for a delay of 0.1 seconds using a 1.0-microsecond clock.

If you miss, take the nearest low value and equalize it with NOPs or whatever.

How close can you come? Can you hit exactly 100,000 cycles without any equalizing code?

Stuff like this is lots of fun. It's called *numeric analysis.* This is most useful for calculating timing loops, musical notes, approximating complicated functions, and so on. Numeric analysis often works on a trial and error basis. Rather than actually solving some messy equation, you just punch in values and see what you get.

Our formula will work on this particular subroutine only for a trainer that has an exact 1-microsecond clock cycle. In other uses, the math changes but the idea stays the same.

Well. We now have a loop within a loop that gives us an "exact" or at least a "good enough" tenth of a second delay. And working with a 0.1-second module is very designer friendly, compared to messing around with clock cycle counting and low level code. All you do is rip off this ready-to-go sub and you have a hassle-free tenth of a second delay ready to go.

But it would be even more designer friendly to build a subroutine that would give you *any* number of tenths of a second delays. This is even better and more convenient and can become a universal timing module for solving lots of different high level problems.

Time out for another side trip . . .

passing variables to a subroutine

The way we can do a .Y-second subroutine is to put some value into the Y register or another working register, and then jump to the

subroutine. We say the value in the working register gets passed to the subroutine . . .

> **PASSING VARIABLES**—Any method of taking values in a main program and letting subroutines or interrupts use them and vice versa.
>
> Also any scheme to move values between high level languages and machine level code modules.

Passing variables is fairly easy to do. You put something somewhere and then tell whatever needs it where to go to look for it. When the sub is done and has a result for you, you put that result somewhere else and let the main program find it.

Simple.

Yet profound. It is very important to be able to move things between subroutines, interrupts, main programs, and high level code in an orderly way without surprises.

There are many different ways to pass variables . . .

WAYS TO PASS VARIABLES

( ) Use working registers
( ) Use the stack
( ) Use absolute short RAM
( ) Use absolute long RAM

The working registers are an obvious choice, as long as those registers are not needed for other things and as long as you understand just which register gets affected which way.

You can also use the stack to pass variables, but be sure you *exactly* empty the stack in precisely the same way you filled it. This gets messy fast, since the CPU will pile addresses on top of your pushed stack values on a subroutine call and demand a return address when you go back. But variables can be saved on the stack by creatively messing with the stack pointer.

Stacks are normally used in just the opposite way, though. Stacks are often used to *save* working registers and other values to prevent any damage to locations that get used by the subroutine or whatever.

For instance, say you have something useful in the X register that you want to keep for the main program to use later. Say further that you have a subroutine that needs the X register to do its particular thing. What you do is save the initial X value onto the stack, go ahead and use the X register as needed by the subroutine, and then come back to the main program and restore the original X value.

The first choices to pass things back and forth between programs and subroutines are your working registers. If you run out of spare registers, then use absolute short addresses. To do this, let the main program store values into absolute short locations, such as the page zero addresses on the 6502. The subroutine then uses these values as needed and adds results to other page zero locations.

Actually, any location in the entire address space can be used to pass things back and forth, as long as that location is protected from other uses. For instance, if you are using both machine language and a higher level language such as BASIC, you can find out where the variables are stashed in the BASIC program and read these locations as needed by the machine language parts of the program.

This sort of variable passing can get tricky. You have to know exactly where everything is at all times. Thus, the high level language must define its variables in exactly the right order.

Some rules . . .

PASSING RULES

( ) Variables that are to be passed must be put somewhere reachable by both passer and passee.

( ) Variables and registers that are NOT to be passed must be protected against destruction by the passee.

( ) No location or variable can have more than one use at any one time.

Sometimes you can speak of *global* variables that can be used by any part of the program and *local* variables that can be used only by a single subroutine or other code sequence . . .

GLOBAL VARIABLE—A value stored in a location that can be used by any part of a complex program.

LOCAL VARIABLE—A value stored in a location that is only used by one small part of a complex program.

It is very important to be sure that the code in a subroutine or an interrupt does not damage things it is not supposed to. This can happen very easily if, say, both the main program and the subroutine have definite about ideas what to do with the accumulator or a working register. After the subroutine or interrupt return, the wrong value may be stuck in the wrong place and the main program will get into deep trouble. We will see an example of the bad scene that can result when we look at interrupts.

So, on any fancy subroutine, it always pays to save everything onto the stack or into a "safe" memory area before anything else happens and restore everything back off the stack or from memory to the way it was just before the return. This way, the main program gets everything back the way it thought it was . . .

On any fancy subroutine, it is ALWAYS a good idea to save all flags and working registers as the FIRST thing the subroutine does and then restore all flags and working registers as the LAST thing the sub does before its return.

On the 6502, only the program counter high and low values get saved automatically for you on a subroutine call. If you want to save the accumulator, you do a PHA. If you want to save the X and Y working registers, you do a TXA or a TYA and then a PHA. Flags are saved with a PHP. You have to be careful to undo the stack in exactly the opposite way you filled it. If the flags were the first one on, they must be the last one off. Note that the accumulator has to be used to save and restore X and Y, so the "real" value in the accumulator should be the first thing pushed onto the stack and the last thing pulled off.

On non-6502 micros, you would do the same thing, saving any flags and working registers that you want to protect from subrou-

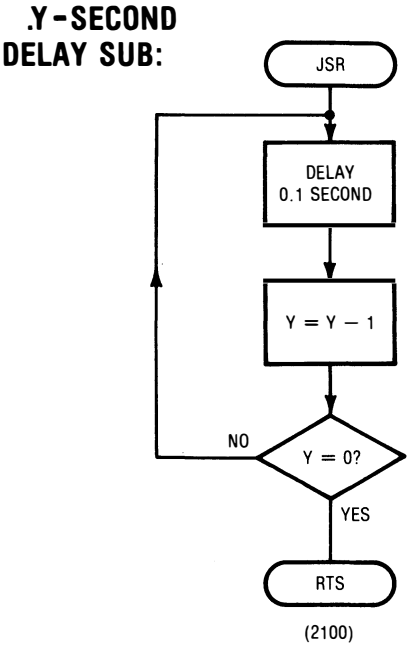
tine damage. Always save flags *first,* so values pulled off the stack don't hurt them.

You also have the option of saving to "safe" memory locations. But watch the trap of more than one part of your code saving to the same memory area and fouling up previous saves. The Apple II monitor routines of IOSAVE and IOREST are deadly this way.

There are exceptions to the "save everything" rule. Saves take time and use up space. If either time or space is so important that nothing else matters, then you may have to save only the stuff that absolutely must not be destroyed. And if the subroutine is very simple and you are very sure *exactly* what is going to change in just which way, you can also skip saving everything in sight.

Back to our discovery module. To do a .Y subroutine that delays from 0.1 to 25.6 seconds for us in 0.1-second increments, we fill the Y register in the main program with the delay value and then go to a new subroutine that uses the 0.1-second delay over and over again Y times.

Here is the flowchart for a .Y-second delay sub . . .



(2100)

All we have to do here is rip off the code for the 0.1-second delay and remove the bottom RTS. Then we decrement Y, test Y for zero, and repeat the 0.1-second delay Y times. When Y hits zero, we finally return. The number of tenths of a second delay you get is set by the number you put into the Y register, which in turn sets the number of trips through the ".1" code.

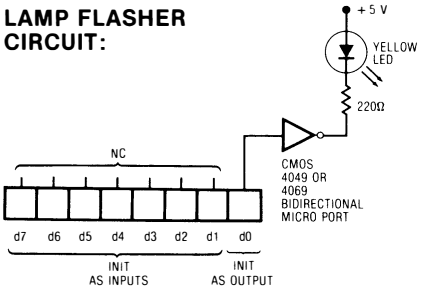
Here is the actual code . . .

.Y-SECOND DELAY SUB

ADDRESS	OP CODE	BYTE #2	BYTE #3	MNEMONIC	HOW?	NOTES
2100	A9	6F		LDA	#$6F	FILL OUTER LOOP
2102	85	51		STA	$51	" "
2104	A9	70		LDA	#$70	FILL INNER LOOP
2106	85	50		STA	$50	" "
2108	C6	50		DEC	$50	INNER MINUS ONE
210A	D0	FC		BNE	$2108	INNER EMPTY?
210C	C6	51		DEC	$51	OUTER MINUS ONE
210E	D0	F4		BNE	$2104	OUTER EMPTY?
2110	88			DEY		ONE LESS Y
2111	D0	ED		BNE	$2100	DONE "Y" TENTHS?
2113	60			RTS		RETURN
2114	—					

Now let's use the code for an 0.3-second on, 2.0-second off light flasher. We will have to connect some sort of lamp driver to the output port, per the details upcoming in Chapter 8.

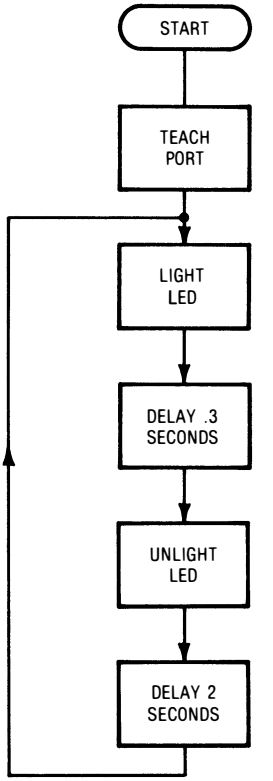
Here is one possible circuit . . .



Our main program is only slightly longer than the earlier delay programs. We use a .Y-second delay subroutine, but we have to pass Y a value from the main program before we go to the .Y-second delay subroutine.

The flowchart . . .

LED FLASHER:



($2000)

and the high level code . . .

LED FLASHER

ADDRESS	OP CODE	BYTE #2	BYTE #3	MNEMONIC	HOW?	NOTES
2000	A9	01		LDA	#$01	TEACH PORT
2002	8D	80	C0	STA	$C080	" "
2005	A9	01		LDA	#$01	LIGHT LED
2007	8D	81	C0	STA	$C081	" "
200A	A0	03		LDY	#$03	SET DELAY
200C	20	00	21	JSR	$2100	DELAY VIA SUB
200F	A9	00		LDA	#$00	UNLIGHT LED
2011	8D	81	C0	STA	$C081	" "
2014	A0	14		LDY	#$14	SET DELAY
2016	20	00	21	JSR	$2100	DELAY VIA SUB
2019	4C	05	20	JMP	$2005	REPEAT FOREVER
201C	—					

First, we *fill* the variable to be passed in the main program and then *use* the variable in the subroutine.

DOING IT:

If you fail to fill the Y register in the main program, after a trip or two around, you will default to a very long time delay of 25.6 seconds.

Why?

As a hint, remember that a working register will always hold its *last* value until something new gets put into it.

The second point on our light flasher is that we are now using one subroutine to do *two* different things for us in two different parts of the program. This is designer friendly code nearly at its best. Instead of worrying about tenths of a second, we now worry only about total delays. To get a delay, just put the delay value into Y, call the sub, and you are home free.

Sure beats counting cycles each time.

But even this is not optimum. We will get even more designer friendly when we use the files of Discovery Module 8.

Some further comments. Be sure to load both your program and your subroutine each time you use them. They obviously will need each other to work properly. Also, note that the Y values will be in hex and not in decimal. Two seconds will be twenty tenths of a second, or hex $14. Finally, a casual glance at our code shows us filling the accumulator with an $01 value twice in a row. This is not wasteful, for first, it is a coincidence to this program, and second, we need a *new* value of $01 after every trip around. Watch details like this and don't try to cut corners.

Note that our .Y subroutine destroys what was in the accumulator, uses Y to pass variables, and ignores the X register. If the old contents of A are valuable, you'll have to save them somewhere, such as in the stack or memory, before using this subroutine.

Before we go on, we ought to look into some . . .

bit twiddling

So far, all of our op-code commands have used *entire* 8-bit words. It was all or nothing. Either the entire word got loaded or stored, or nothing at all happened. It sure would be nice if we could do anything we like to any number of bits in any bit position at any time.

We can handle sixteen bits at a time by using 8-bit words in pairs. We have already seen several examples of this in the absolute long addressing where one 8-bit byte handles the page address and a

second 8-bit byte handles the position-on-the-page address. By working with pairs of words, anything that you can do with 16-bit words in a 16-bit micro can also be done with pairs of 8-bit words in an 8-bit micro. All it takes is some extra commands and extra time.

We can also go the other way and mess around with individual bits. *Bit twiddling* consists of isolating and working with individual bits . . .

BIT TWIDDLERS—Op-code commands in a micro that can isolate and work on single bits in a word.

Bit twiddling gets very important whenever you want to sense what a single input line is up to or want to alter only one line of eight ports routed to an output. With bit twiddling, you can mix and match input and output pins on some ports and sense and change them at will. Most 8-bit micros have lots of different bit twiddling commands available in many different address modes.

There are three basic types of bit twiddling commands. These are *logic commands, sideways shovers,* and *testers* . . .

TYPES OF BIT TWIDDLERS

LOGIC COMMANDS—Perform traditional digital logic on a bit-by-bit basis.

SIDEWAYS SHOVERS—Change bit positions by shifting or rotating.

TESTERS—Compare bits against known values or set flags for certain bit conditions.

Let's look at some examples of each type of bit twiddler.

Logic commands are instructions that do any of the traditional "Boolean" functions, such as AND, ORA, EOR, and so on. On an 8-bit microcomputer these instructions will do a separate bit-by-bit logic action on each bit in each position. The logic is usually done by working the accumulator against a fixed and immediate mask or by working a mask in the accumulator against some location in the main memory. So, you can think of one 8-bit micro instruction as, say, eight separate hardware AND gates, each of which works on a single pair of bits.

The *sideways shovers* are used to move bits from left to right or right to left. *Shift* and *Rotate* commands are typical. These instructions can also move bits into the Carry flag where they can be

tested. Moving all the bits to the left one place is the same as multiplying the straight binary number in that word by two. Moving all the bits to the right one place is the same as dividing the straight binary number in that word by two. Sideways shovers are also useful for converting hex to ASCII and for packing pairs of 4-bit words into a single 8-bit location.

Some newer microprocessors have a sideways shover that can move you any number of bits right or left with a single instruction. This ability is called *barrel shifting*. Barrel shifting can be faked through repeated shifting or rotating of a single bit position at a time.

The *testers* let you test individual bits to see whether they are ones or zeros. The results of these tests usually end up setting or clearing one or more flags. You can then test the individual flags and use IF instructions to alter what is happening. The CMP, or compare, function lets you find out if one number is equal, larger, or smaller than another. The BIT test, and similar commands on other micros, directly lets you investigate what a single bit is up to without damaging anything else in the machine.

Let's look at some examples of bit twiddlers. Here's the 6502 card for the AND immediate logic command . . .

AND	**AND ACCUMULATOR WITH MASK**	**29**

(IMMEDIATE addressing)

2 Bytes — 2 Clocks

AND #$20 — N and Z flags

Performs a bit-by-bit logical AND of the contents of the accumulator against a mask in the second instruction byte. The result ends up in the accumulator. AND instructions can be used to force certain bits in a word to zeros.

Assume the accumulator holds an $60.

2B4D– 29 20 ANDs the $20 of the mask against the $60 in the accumulator, giving us a $20 result in the accumulator, while clearing the N and Z flags.

Here is what happens: there is a bit-by-bit logical AND done on each position in the accumulator, working against the same position in the mask. The results of each individual AND are then placed back in the accumulator.

Remember that the AND rules say that 0 AND 0 = 0; 0 AND 1 = 0; 1 AND 0 = 0; and finally 1 AND 1 = 1.

Each bit position is treated separately. What is happening in, say, bit 6 has no effect on bit 2, and vice versa.

It is only when there is a one in both the accumulator location AND the mask location that you get a one result in any bit position. Thus, the AND instructions will force *zeros* into any position that does *not* have a one in the mask.

If you force zeros in only a few locations, you make room for new things that can be put in these bit slots. For instance, an AND #$0F will clear the top four bits and convert an ASCII numeral to a 4-bit hex or BCD number.

If you force zeros into all but one location, you end up testing a single bit to see if it is one or zero. For instance, if we have a #$01 mask, we can get an $01 result only if there is a one in the LSB of the accumulator. So, if there was a one here, we get a one result, which clears the Z flag. If we had a zero here, the AND operation will give us a zero result, which sets the Z flag. You can then BNE or BEQ to alter what you are going to do next.

You can do the same thing to any other bit position. The eight magic mask values to check any bit location are $01, 02, 04, 08, 10, 20, 40, and 80, going from right to left. Each masks out its own bit for special treatment.

Note that the AND instruction destroys what is in the accumulator. If you wanted the other bits for something, you should have saved them elsewhere first. The upcoming test instructions are not as destructive and can be more useful.

There are usually many different address modes available for the AND instruction. Among others, the 6502 has page zero and absolute long AND addressing. With AND instructions that work with the address space, the value in the accumulator is ANDed against the memory location, and the result goes in the accumulator. This time, the value stashed in main memory is not destroyed but, once again, the old accumulator value is, since the old value gets replaced by result of the logical AND.

The mask works differently in both cases . . .

In an AND immediate, the mask is found in the second byte of the instruction and the result ends up in the accumulator.

In an AND against a location in the main address space, the accumulator usually holds the mask and the result ends up in the accumulator.

Time to do the other logic instructions . . .

DOING IT:

If your trainer is from the 6502 school, complete the AND, ORA, and EOR cards for immediate, page zero, and absolute addressing at this time.

If not, complete all cards for all instructions that perform logic operations in immediate, absolute short, absolute long, or any other address modes that you have already have used.

The immediate ORA instruction does a bit-by-bit logical OR of the accumulator against the mask and puts the results in the accumulator. This time, *any* one in either the accumulator or the mask forces a bit location to *one*. So, OR instructions can be used to *force ones* into certain bit locations.

OR instructions can also be used to combine bit groups into an entire 8-bit word. For instance, if you had a hex $04 in the accumulator and ORed it immediate with a mask value of $30, you would end up with a $34 in the accumulator, which is the full ASCII equivalent of the 4-bit hexadecimal $4.

As with AND, the OR instructions usually can also work anywhere in the address space. The mask starts in the accumulator and the result ends up in the accumulator. The value in main memory is not altered.

The EOR, or Exclusive OR, instruction works in much the same way that AND and ORA do, only it *changes* individual bits. If the bits being logically acted on are the *same*, you end up with a *zero* result. If the bits being logically acted on are *different*, you end up with a *one* result.

One use of the EOR instruction is to complement 8-bit words. If you do an EOR #$FF, you change each and every bit in the word in the accumulator to its complement, making each one a zero and each zero a one.

A very fancy use of EOR is to add and remove images from a graphics screen. If you do this just right, and if you keep a separate

copy of the new image somewhere else, you can "un-EOR" the display and remove the new image without hurting the background. This is essential for animation and other applications which have something moving against a background.

Other uses of the EOR instruction involve binary addition, controlled complementing, change detection, and driving liquid crystal displays.

As with AND and OR, you can also go into the absolute address space and do an EOR. The mask usually sits in the accumulator and is destroyed, leaving the result in the accumulator.

Remember . . .

AND logic functions force ZEROS into certain bit locations.

ORA logic functions force ONES into certain bit locations.

EOR logic functions force CHANGE into certain bit locations.

With these three logic functions, you can twiddle any bit location in any word into a one, a zero, leave it whatever it was, or make it into whatever it was not. You can do this with a word in the accumulator worked against a fixed set of masks, or you can put masks in the accumulator and work these masks against anything in the address space. Either way, the result ends up in the accumulator.

You can also use these three logic instructions as building blocks to do any logic function, although this is not very common. For instance, you can AND and then EOR #$FF to get the NAND logic function. The EOR #$FF simply acts as an output inverter.

The only disadvantage of these logic commands is that they destroy what was in the accumulator in the process of getting a result. If you want to do something different with some of the other accumulator bits later, you have to save a copy of the word as it was before you start doing any logic commands on it.

Here's a card for a sideways shover . . .

ROR	ROTATE RIGHT THRU CARRY	6A

(IMPLIED addressing)

1 Byte — 2 Clocks

RORA — N, Z, and C flags

Takes what is in the accumulator and moves each bit one to the right. The rightmost bit goes into the carry flag and the old carry value goes in the most significant bit.

Assume that the accumulator holds a $61 and the carry flag is cleared.

2C31– 6A Leaves a $30 in the accumulator and the carry flag set, since everything gets shifted one to the right.

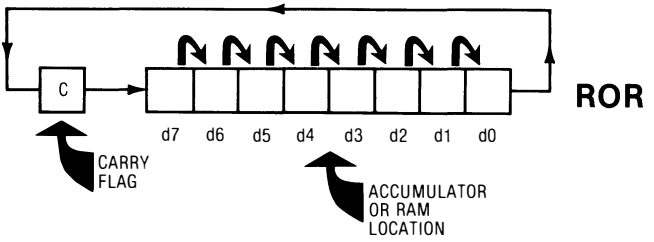
And the bits go round and round. LSB goes into the carry flag. Carry goes into the MSB. MSB moves one to the right. All others move one to the right, just like musical chairs. Some micro families give you the choice of going through the carry flag or just going round and round through the bits only.

Several gotchas. You can rotate any location in the address space, provided it has RAM in it that can be both read from and written to. This rotate scheme takes a while but is very powerful, and no register values are destroyed in the process. The N, Z, and C flags are all changed, giving you a powerful way to alter and test memory locations without using the accumulator or another working register.

On the 6502, there are a few quirks involving the shift and rotate instructions. The ROR immediate instruction that rotates the accumulator has to be entered in some assemblers as the four-letter mnemonic RORA. Thus, a page zero rotate would be ROR $06, an absolute one would be ROR $FE06, and an accumulator rotate would be RORA.

In a masterpiece of PR flak, the 6502 people separate out the four "accumulator" mode shift and rotate instructions and classify them as a separate addressing mode. *Accumulator* addressing is simply an implied instruction that works only with the accumulator.

Shift and rotate instructions are usually best shown with a small sketch, like this . . .



There is a ROL command that also shoves things round and round, only this time around things move to the left, with the MSB going into the carry flag, the old carry value going into the least significant bit on the right, and so on. Each bit moves one to the left, through the carry.

You can also shift without rotating. The 6502 uses different sounding names for two commands that do the exact opposite of each other. The ASL, or *Arithmetic Shift Left,* command shifts everything to the left, feeding zeros into the MSB each time, and putting the LSB into the carry flag. The LSR, or *Logical Shift Right,* command shifts everything to the right, feeding zeros into the LSB each time, and putting the MSB into the carry flag. The N and Z flags pick up the usual result in the usual way.

If we were to have an ASR, or *Arithmetic Shift Right,* command, it would have to preserve the MSB so that 2's complement arithmetic would still work. This command is available on some micros but not on the 6502. ASR is sticky to use, so be very careful with this one.

One obvious use of rotate commands is to move a bit value into the carry flag where it can be saved or tested with IF instructions. Note that after nine identical rotate commands, you end up back where you started. This is one way to output serial bit patterns and not destroy the original word.

Shift commands can be used to multiply or divide a straight binary word by two. Shift to the left to multiply and to the right to divide. One place where you multiply by two is in picking pairs of addresses out of a table. Your pointer to this table can be doubled by an ASLA command to point automatically to the start of a new position-page address pair.

Note that eight shift commands in a row empty the byte to all zeros. Shifts are destructive. Rotates are not necessarily destructive, as long as you go exactly once around.

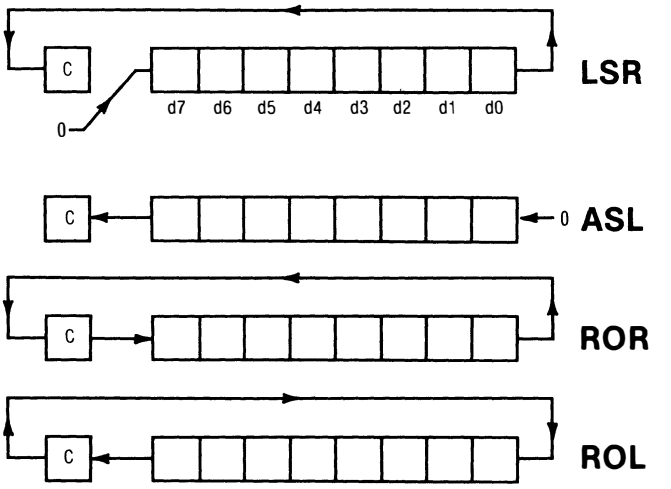
More cards . . .

DOING IT:

If your trainer is from the 6502 school, complete the ROR, ROL, ASL, and LSR commands for implied (accumulator), page zero, and absolute addressing at this time.

If not, complete all cards for all instructions that move bits sideways in a word, using those address modes you already know.

Here's a family portrait of the 6502 sideways shovers . . .



To recap, there are four sideways shover classes of instructions available in the 6502 that may be used on the accumulator or may act on any location in the address space.

Two rotate commands move everything round and round through the carry, one clockwise, and one counterclockwise. Two shift commands move the bits in the word to the right or the left. A zero is put in the one end of the word, while the other end of the word overflows into the carry flag.

Our final class of bit twiddlers can be used to test bits or entire words. These are very powerful, but you have to be very careful and be sure you know exactly how they work.

Let's start with a CMP for compare card . . .

CMP **COMPARE ACCUMULATOR AGAINST MASK** **C9**

(IMMEDIATE addressing)

1 Byte — 2 Clocks

CMP #$04 — N,Z, and C flags

Compares the value of the accumulator against the mask pattern of the second byte. Sets the Z flag if the two are equal, resets Z otherwise. Sets the C flag if the accumulator is greater than or equal to the mask, resets C otherwise.

Assume that the accumulator holds an $A3.

2C41– C9 02 Sets the carry flag since the accumulator is greater than the mask. Clears the Z flag since the accumulator does not equal the mask.

You could also think of the CMP instruction as an EXCLUSIVE NOR logic command. We call it a test instruction since CMP is almost always used as part of a test.

This instruction compares the accumulator against the mask. If the two are equal, the Z flag sets. Checking for equality is the most common use of the CMP instruction. If the accumulator is bigger than or equal to the mask, CMP sets the carry flag. If the accumulator is less than the mask, the carry flag clears. You can easily remember this by thinking of carrying a large mouth bass. Then remember that the "Big A SetS."

The carry flag can check to see if the accumulator *is smaller than* the mask. The zero flag can check to see if the accumulator *equals* the mask. The two together can be used for a "greater than" check.

Compare instructions are almost always followed by an IF instruction, such as a conditional branch. Since other instructions can change the flags, it is a good idea to always put the IF instruction immediately following the compare instruction in your code . . .

Compare instructions should be immediately followed by their IF instruction.

Think of a test as a two step operation. First, we compare against a mask or a memory location. Certain flags will change on the comparison. Then we do some conditional branch or jump based on the flag results of that compare.

One use of "greater than" tests is for range checks . . .

DOING IT:

Show how a pair of compare instructions can check the value in the accumulator to be sure it is an ASCII character from uppercase *A* through *Z*.

Range checks are very important any time you could end up with the wrong value somewhere. In this example, we might have a twenty-six selection menu. We want to pick only legal letters, and we want to either ignore anything else or perhaps provide some error message.

More cards . . .

DOING IT:

If your trainer is from the 6502 school, complete the CMP, CPX, and CPY cards for immediate, page zero, and absolute at this time.

If not, complete all cards for all instructions that compare a register against something using all address modes that you already know.

When you compare against a memory location, pretty much the same thing happens as when you compare against a mask. If the

accumulator is equal to the contents of the memory location, the Z flag sets. If the accumulator is larger or equal, the C flag sets. In this case, the accumulator is usually the mask, and memory is what you are comparing the mask against.

Unlike the other logic commands, the CMP does not destroy what is in the accumulator. Thus you can compare what is in the accumulator over and over again until you find the match you need. While the CMP command will not test a single bit in a single position for you, it will test for a complete match of all eight bits in all eight locations at once.

On the 6502, the CMP instruction also subtracts the mask from the accumulator and lets this result set or clear the N flag. This specialized test result is confusing to most beginners and is not used nearly so often as the test for equal or greater than. Again on the 6502, compare instructions involving the X and Y registers are also available in the immediate, page zero, and absolute addressing modes.

The advantage of the CMP is that it lets you test a whole word for equal or smaller than. The disadvantage is that CMP by itself does not normally let you test a single bit at a time.

To test single bits at a time, we have, of all things, a BIT test . . .

BIT **TEST BITS IN MEMORY** **2C**

(ABSOLUTE LONG addressing)

3 Bytes — 4 Clocks

BIT $C056 — N,V, and Z flags

Tests the bits in the memory word selected by the position or low value in the second instruction byte and the page or high value in the third instruction byte. Puts a copy of the memory MSB into the N flag. Puts a copy of the next-to-MSB into the V flag. Does a non-destructive AND against the accumulator and sets the Z flag on zero result.

Assume an $04 in the accumulator and an $85 in $C056.

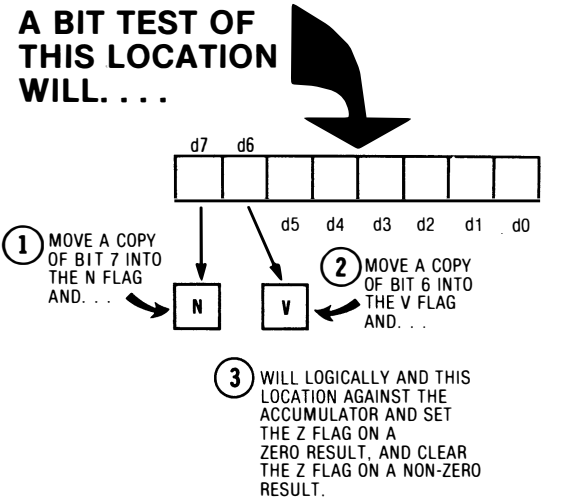
211E– 2C 56 C0 Tests contents of $C056. Sets N and Z flags but clears the V flag.

Wow. This one is sort of hairy. On the 6502, our BIT test has three different actions. Details will change for other micros.

First, and most handily, the BIT reaches into memory and unconditionally puts a copy of memory bit B7 (the most significant bit, or MSB) into the N flag.

It then goes one bit to the left, picks up B6, and puts a copy of B6 into the V flag. Note that the positions of the V flag and the N flag in the processor status or phlag register are the same as those bits tested in your target word.

Huh? A sketch makes it obvious . . .



In this first way to use the BIT test, everything happens free and automatically. You don't have to worry about what is in the accumulator. Since only bits 6 and 7 get moved into the flags, though, it always pays to put the things to be tested on the highest two bits of any word . . .

The 6502's BIT test quickly and simply tests bits 6 and 7 of a memory location.

It pays to put any bits that need testing in these locations.

This automatic BIT test operation is very handy when you interface I/O peripherals to the 6502, since many peripherals automati-

cally provide service requests and other command information on the high pins. If you are doing your own I/O using mixed inputs and outputs, it pays to put the inputs on the high pins and the outputs on the low pins.

Once you have isolated your bits 6 and 7 in the flags with a BIT test, you can branch on bit 7 with a BPL or BMI and separately branch on bit 6 with a BVS or a BVC.

The second way to use the BIT test does tie up the accumulator. You can test any of the bits in a word by putting a mask value in the accumulator and then BIT testing. The AND of the mask value in the accumulator taken against the location you are BIT testing sets or resets the zero flag. Unlike the AND instruction, the contents of the accumulator are not destroyed.

For instance, if you want to know what bit B6 is up to, you simply use the BIT test by itself and then check the V flag. If you want to know what bit B2 is up to, you store a $04 in the accumulator and then do a BIT test. The third bit to the left in the accumulator gets ANDed with the third bit to the left in the memory, and the Z flag is *cleared* if both bits are ones, since 1 AND 1 gives a non-zero result. The $04 in the accumulator is not hurt and stays there for further use. Note that this is backward from the 6502 CMP instruction, where equality *sets* the Z flag on a match.

There is a third and sneaky use for a BIT test. Sometimes you only want to address a memory location rather than read from it or write to it. This happens in the Apple II with its soft switches and in other systems where you only want to flash an address to let some hardware respond. One important use of this is resetting keyboard strobes. The BIT test gives you a quick and dirty way to flash an address without any worry about changing any register or memory values. The N, Z, and V flags will change during the address flashing, so be careful, or you can do a PHP, BIT, PLP sequence to restore the flags to the way they were.

To summarize, the BIT test is a powerful way to test individual bits in a memory word. On the 6502, bits 6 and 7 automatically get copied into the V and N flags. If you put a mask value in the accumulator, the BIT test will AND on this mask, letting you alter the Z flag, clearing the flag on a one match. The usual IF instructions then let you decide what to do as a result of the condition of the bit you just tested. Finally, the BIT test automatically flashes a memory address for special uses. More on this in the next chapter.

BIT tests work differently on different micros, but there is usually some way to isolate the bits for test . . .

Some micros will combine their test and branch actions into a single instruction. The new 65C02 does this on some exciting new op codes, but the old 6502 does not.

DOING IT:

If your trainer is from the 6502 school, complete the BIT absolute and BIT page zero cards at this time.

If not, complete all cards for all instructions that test or change individual bits in memory or working registers, using those address modes you already know.

If you have a very old or very strange micro, there may not be individual bit tests. But you can always do something to test bits, such as saving the word and then rotating the word through the carry flag until the bit you need sits there, or else by isolating a bit position using an AND mask. As with almost everything in the micro world, there are many different ways to do things, and no one way is necessarily the best or only way to handle a problem.

Pay particular attention to the Z flag on the micro of your choice. CMP and BIT tests in different families may have different use rules for when the Z flag goes to a one, so always check. Once again, CMP *sets* the 6502 Z flag on equality, and BIT *clears* the 6502 Z flag on a bit match.

Just for kicks, the next time you want to torment a COBOL freak, just ask them to show you the code sheets needed to read the third bit over in a port. The BIT test in COBOL takes page after page of garbage to replace five or less bytes of machine language code!

Let's see how to use some commands that mess with individual bits in our next discovery module . . .

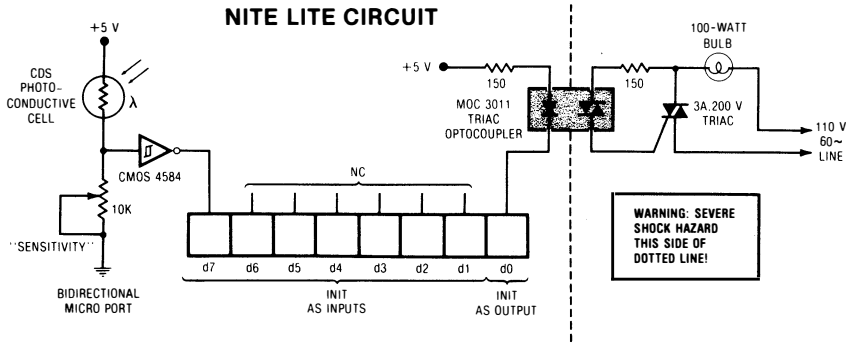
DISCOVERY MODULE 7

NITE LITE

Write a program that lights a 100-watt light bulb when a photocell thinks it is dark outside.

Once again, you will need some more I/O details from Chapter 8. We need a photocell and a conditioning Schmidt trigger circuit for an input. We'll separately need an optocoupler, a triac driver, and a light bulb for an output.

Let's use a circuit something like this . . .



We route the output of the photocell circuit into an input on one bit line of a single port. We route the output of the microcomputer to an output bit line of a port. We'll assume you have a bidirectional I/O port, and that you will input on the MSB line B7 and output on the LSB line B0 of the same port of our MYTH-1 trainer.

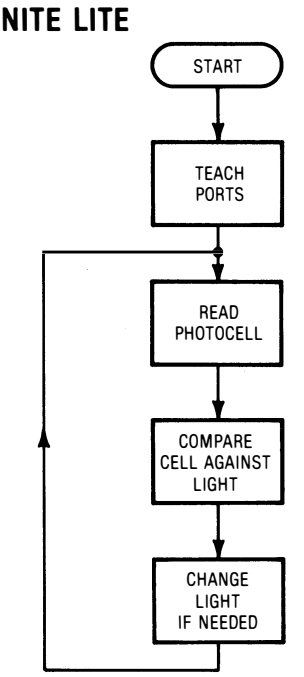
Note that some ports on some trainers may need a bias resistor or even a buffer to drive an optocoupler reliably, per details in the next chapter. Note further that a severe shock hazard exists on the right hand side of the nite lite circuit, since it is connected directly to the AC line. Think!

Let's do our nite lite with some logic instructions and some sideways shovers. You read the port and then move a copy of the port code sideways till the input bit aligns with the output bit position.

This takes two ROLA commands, one to move the MSB into the carry flag, and a second to move carry into the LSB. Now that the input bit and the output bit are in the same position, we can compare them. We can EOR these two to see if any change is needed and update the output. We will also very carefully *exclude* the other bits from the EOR with a suitable AND mask, so that no other bits are allowed to change.

A *lit* photocell tells us it is day by giving us a *zero* to the input port. If the output port is also a zero, this means we do *not* want to light the bulb. An EOR of zero versus zero gives us zero, and we make *no change* to the output. Similarly, if it is night and the lamp is on, a one EORed against a one still says to make no change. Only when there is a difference between input and output do we make the change.

Here's one possible flowchart . . .

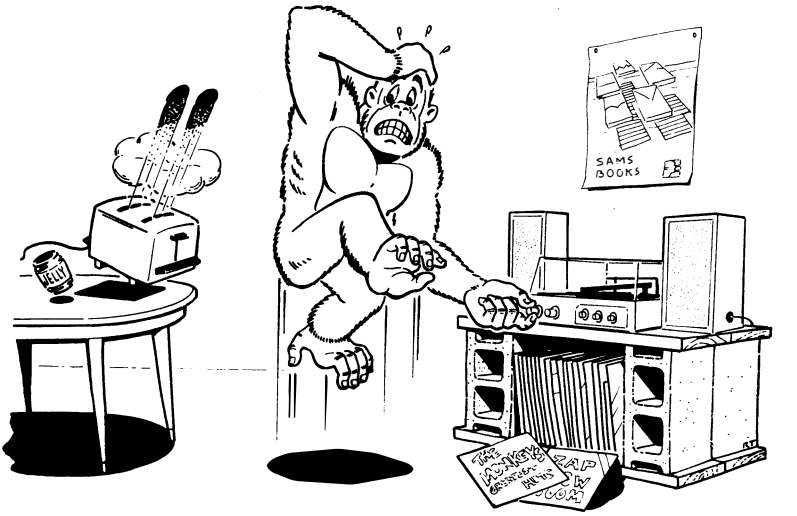


One very important thing when you are manipulating individual bits is to make sure that no other bits in other bit positions get upset in the process. If we have only a single input line and only a single output line, this is no problem. But more often than not we have to be super careful to make sure nothing else gets disturbed.

Our EOR logic will do this, since we will end up changing only a single output bit, and then only if the change is needed.

Fail to do this and the windshield wipers will turn on when you switch to high beam. Or the toast will pop up when you change FM stations. Or something else equally poor.

And ungood.



Here's the code for the nite lite . . .

NITE LITE CODE

ADDRESS	OP CODE	BYTE #2	BYTE #3	MNEMONIC	HOW?	NOTES
2000	A9	01		LDA	#$01	TEACH PORT
2003	8D	80	C0	STA	$C080	" "
2005	AD	81	C0	LDA	$C081	READ PORT
2008	2A			ROLA		ALIGN INPUT
2009	2A			ROLA		TO OUTPUT
200A	29	01		AND	#$01	MASK OUTPUT
200C	4D	81	C0	EOR	$C081	CHANGE IF NEEDED
200F	8D	81	C0	STA	$C081	AND REPLACE
2012	4C	05	20	JMP	$2005	REPEAT FOREVER
2014	—					

We first teach the port that line 0 is an output and line 7 is an input. In the interest of safety and good practice, we'll also make all the other port lines inputs.

Then we read our port by loading it into the accumulator. We then rotate the mask two steps to the left with a pair of ROLA commands. One to get into the carry, and one to get out of the carry and into the LSB. This aligns the input bit in the accumulator with the output bit still in the port. We then EOR the accumulator against the port, and the resulting bit in position zero will be a one if the port needs to be changed and a zero if it does not. An AND immediate #$01 then masks out all the other bits in the accumulator and forces them to the no-change zero state. We then once again EOR the accumulator against the port. This time, the output bit will change if needed, but all other port lines will stay the way they were. The corrected port value in the accumulator then gets restored back to the port, completing the update process.

Note that our flowchart has no IF diamond in it, since the program flow always goes in the same direction. We do *not* test and branch. Instead, the EOR command will change or not change the output as we wish, but it always does so in the same program sequence.

A final jump lets the program repeat forever. Note that we have not changed any of the other bit lines. Only the output line changes, and then only if it needs changing.

Some variations . . .

DOING IT:

Show five other ways of doing the nite lite, using other commands that manipulate individual bits in a word. Use as many different bit twiddling instructions as you possibly can.

Show how you keep your other port bit values intact while you do this.

Which is the "best" program? Why would you use the others?

As usual, there is no single best way to write a program. It depends on your style, on what has to be done, and on the micro you chose to work with. Obvious goals in any program are to keep the code short, have it run fast, and make it easy to understand.

These three goals usually fight each other and you often have to seek a balanced middle ground.

Did you think of just connecting the photocell to the optocoupler without using the micro at all? Or just using your micro to "borrow" some supply power? This is clearly the cleanest and simplest solution with the shortest program. But this teaches us nothing about bit twiddling, and if we make any change at all, it's back to square one.

Nonetheless, it always pays to ask, "Do we really need a microcomputer for what we are trying to do?" The most elegant solution is often the simplest of all.

Time out for some foolishness . . .

DOING IT:

Let the light bulb's light shine onto the photocell in dim room light.

What happens? Why?

Now, rewrite one of the nite lite modules so that the light goes ON in the daytime and OFF at night.

With this new module, a dark room, and feedback from bulb to photocell, show how you can light a light bulb with a match, and then blow it out by cupping your hands around the bulb and blowing on it.

Both of these involve *feedback* from output to input. One is *negative* feedback, while the other is *positive*. Which is which? What does this tell you about any micro interface application where outputs may affect inputs?

The hidden nasties in this discovery module included learning about instructions that manipulate individual bits for logic, sideways shoving, and testing. You also have to do a 1-bit A/D conversion to interface a photocell to a port and do an output isolating and amplifying optocoupler triac interface that will drive a big light bulb. More details on I/O in the next chapter.

Let's take a peek at our next discovery module . . .

DISCOVERY MODULE 8

TEXT OUTENBLATTER

Write a program that places the name of one of sixteen animals onto a video screen or out to a printer or terminal.

Use a text file and a pointer stash. Use the keyboard or keypad on your trainer to select each animal by typing the first letter of its name.

The best possible designer friendly code will use *files*, so that we need only to change data values in files and do not have to rewrite the program for each new use.

Of course, everyone knows what files are and how to use them properly, right?

Uh, that's not quite it. He's not even wrong this time.

A file is a block of data accessed by a program . . .

FILE—A block of data accessed by a program.

Depending on who is using one for what, files can go by different names. A *text file* holds messages or message-related things like printer or DOS commands. A *table lookup* is a file used for scientific or math purposes where one number gets substituted for another. Error correction of non-linear analog signals is one use of table lookups. A *code converter* is a table lookup used for code conversion, such as, say, going from Baudot to ASCII or from QWERTY to DVORAK. An *address block* is a file of addresses that tell us where to go next, possibly in response to a menu selection. A *pattern file* is a series of dot patterns needed to put colors onto a video screen or combinations of ones and zeros out a port. These are sometimes called *shape tables* or *sprite maps*, depending on how they are used. A *pointer stash* is a file to access another file holding addresses or index values that show where each new thing starts in a second and longer file.

Like so . . .

TEXT FILE—Holds a message or a string of printer or disk commands.

TABLE LOOKUP FILE—Gives a new value in exchange for an old one.

CODE CONVERTER FILE—Exchanges an input code for a different output code.

ADDRESS FILE—Holds a series of addresses that give a choice of where to go next.

POINTER FILE—Shows where to access a second file to get a certain message or other longer sequence.

PATTERN FILE—Keeps graphics images or other shape information until needed.

Higher level languages will also use files for mailing lists, inventory, word processing, employee records, data base management, and so on. Many of these are variations on the text file. Regardless of the language, all file concepts are pretty much the same. Let's worry only about the more fundamental file concepts here.

One very important thing to remember about any file is that a file is NOT a program and is NOT capable of running by itself. Files are *separate* blocks of data that are used elsewhere by some program . . .

Files are NOT programs and they are NOT capable of running!

Files are blocks of data used by machine language code placed elsewhere in the address space.

Thus, if you have a program that involves itself with files, the machine language code will be in one place and the file will be somewhere else in the address space. More often than not, the file will shortly follow the machine language code. The machine language code NEVER jumps or branches to the file, and there is NEVER a time when the microprocessor tries to execute op code in a file. The contents of a file are used by loading a location into a program or else by receiving a stored value generated by a program.

If you access your file one word at a time in sequential order from some starting point, you have a *sequential access file*. If, instead, you can get at any file entry at any time, you have a *random access file* . . .

SEQUENTIAL ACCESS FILE—A file where the values are used in sequential order, from beginning to end.

RANDOM ACCESS FILE—A file where any value can be used at any time in any order.

The telephone book is a good example of a random access file, since most people get only the number they want, rather than starting at the beginning of the book and reading every number

till they get to the one they are after. This paragraph is an example of a sequential access file. Most people will start at the beginning of the paragraph and read it rapidly from left to right and slowly from top to bottom, picking off the letters and the words in sequential order. Note that the meaning of this paragraph will change dramatically if you don't read these words in the order that I intended you to. Try it.

It doesn't matter to the file whether it is accessed randomly or sequentially. The program that is accessing the file will decide whether to do a sequential access, a random access, or a combination of the two. A text file for a word processor will usually be output strictly sequential, for we want the words always to come out in the same order. When we edit this file, though, we will interrupt the sequential access long enough to go where we want in the file and make any changes we like. A text file for an adventure will be a mix of sequential and random access. Normally there will be dozens of rooms and hundreds of response phrases that can show up in any order. But each of these responses will be printed out one letter at a time in sequential order, when and if needed.

Files are very important in the micro world because they are so designer and user friendly. If you set up and debug a program that uses files, you can make the program do many other things simply by changing the file values. The working code does not change, and debugging becomes far simpler and much quicker.

Files also force you into thinking much more generally and into working with code that does many different things in many different ways. As an example, you could write a traffic light program that does not use any files. But any changes at all in that program would require you to go back to square one and start all over again with new code. If instead you write a program to generate any number of light patterns combined with any number of matching delay values and then set up a *pattern file* for the lights and a *delay file* for the time delays, you will not only solve your particular traffic light problem but solve any and all traffic light problems at the same time.

Further, by using the files, you can easily change the traffic light program to a pendulum model, or a disco chaser, or a theater lighting system, or whatever. The general program that uses files is almost always far more convenient and easier to use than a dedicated program that does only one single task.

And doing things in the most flexible and general sort of way, of course, is what micros are all about.

Most microcomputers have some way that you can pick values out of a file. Sequential values are normally handled with an *indexed* instruction mode or else by a *register indirect* process.

Either access method makes it quick and easy to get values out of a file.

Here is an example of a 6502 indexed instruction . . .

LDA | **LOAD A INDEXED BY X** | **BD**

(ABSOLUTE INDEXED addressing)

3 Bytes — 4 Clocks

LDA $0900,X — N and Z flags

Adds the absolute base address found in the second or position byte and the third or page byte to the index value in the X register. Then goes to that address and puts a copy of what is there into the accumulator. Used for rapid file access with short code length.

Assume that the X register holds a $06 and that $0906 holds a $A3.

2C34– BD 00 09 goes to location $0900 + 06 = $0906, reads the A3 there, and puts it in the accumulator. The Z flag is cleared and the N flag is set.

Indexed addressing gives us a very simple way to pick values out of a file. It is the same as the counterperson in the donut store who can pick one donut off the rack at a time in sequential order.

There are two parts to an indexed address. The first is called the *base address* and is built into the instruction. The second part is called the *index value* and is held in a working register . . .

BASE ADDRESS—The starting point of an indexed instruction.

The base address is usually built into the op code.

INDEX VALUE—The offset or "add-on" value that is added to the base address to get the final or "real" address.

The offset is usually held in a working register.

You need *two* separate pieces of information to find an indexed address. First you find where to start and then you find how much to add to the starting point to get to where you really want to go.

The big advantage of indexed instructions is that code using indexed instructions is far shorter and far more convenient to use, compared to brute force coding.

For instance, suppose we wanted to pick off the contents of a dozen locations ranging from $0900 through $090B and route them to a port. Here is the brute force way to do it . . .

```
LDA $0900
STA PORT

LDA $0901
STA PORT

LDA $0902
STA PORT

LDA $0903
STA PORT

LDA $0904
STA PORT

LDA $0905
STA PORT

LDA $0906
STA PORT

LDA $0907
STA PORT

LDA $0908
STA PORT

LDA $0909
STA PORT

LDA $090A
STA PORT

LDA $090B
STA PORT
```

I have just snuck some *assembler notation* in on you here. LDA $0906 tells us we want to absolutely load the accumulator from

location $0906, and will have the 6502 machine language coding of $AD $06 $09. We know this is absolute addressing since there are four digits in the address and since there is nothing following in the way of commas, parentheses, or index register names.

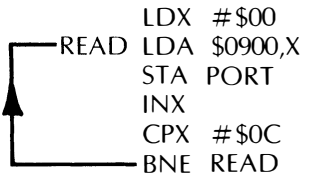
STA PORT tells us to route the accumulator out a port. Which port? Ahead of time you tell the assembler where your port is. One way is to tell your assembler "PORT EQU $FD0C," short for "anytime from now on that I use the label "PORT," I really mean location $FD0C." So, if we taught the assembler that PORT was $FDOC, a command STA PORT would mean $8D 0C FD in 6502 machine language. Once again, since the address called PORT has been defined as four digits and no commas, register values, or parentheses, we know to use absolute addressing.

Note how much more meaningful is an address labeled PORT, compared to $FD0C.

Anyway, back to the file reading code. We see that it takes six dozen bytes to read twelve data values and then output them to a port. Six bytes are needed per word, three for the load and three for the store. And, this in fact turns out to be the fastest possible way to read a file with the 6502.

But it certainly is *not* the shortest.

Let's try the same thing using indexed addressing . . .



Well, even if it looks like gibberish, it is obviously much shorter gibberish than before. Let's see. First we load the X register with a zero value, and then load the accumulator with what is at $0900 plus zero, or simply the contents of address $0900. We then shove this out the port.

Next, we add one to X. We then check to see if it went one past where we wanted to stop. If not, we repeat the loop. The *label* READ following the BNE tells the assembler to run back through all the code till it finds a READ label at the start of some code. The assembler automatically figures out all the relative branch nonsense that you have done by hand so far. We go through the loop twelve times. Each time, we advance the index by one and go to the next location in the file. Each file value goes out in turn . . .

DOING IT:

Convert the two previous assembly language sequences into machine language code.

You should end up with fourteen bytes using the indexed load instructions and seventy-two bytes using brute force absolute loads. In this case, that's nearly a 6:1 improvement.

But, should you read a hundred values out of your file, the indexed method still only takes fourteen bytes, compared to six hundred bytes needed for the brute force.

So, indexed instructions dramatically shorten the length of code needed and make it much easier to read files as well. It is also far simpler and much more flexible to change an indexed loop than to do brute force coding.

There is, however, a speed penalty. One load and store done the brute force way takes only eight CPU cycles, while one load and store done indexed can take up to sixteen CPU cycles, allowing for loop overhead. So, you pay a 2:1 speed penalty for the convenience and flexibility of indexed code. This, of course, is typical of any loop. You have to pay the loop overhead every time you go through the loop. This speed penalty is often negligible, especially when outputting text, since printers and video screens communicate at fairly slow rates.

DOING IT:

If your trainer is from the 6502 school, complete the LDA,X and LDA,Y absolute and page zero cards at this time.

If not, complete all cards for all of the simpler indexed instructions available on your micro.

If your micro lacks indexed instructions, do cards on other ways of reading file values, such as register indirect.

Some microcomputers may not have indexed instructions. Instead, they may offer some other way of doing the same thing. An addressing mode called register indirect can often do the same as an indexed instruction can.

In register indirect addressing, the accumulator or whatever is loaded from an address specified by another register. Since it is usually easy to increment, decrement, or compare this address register, you can read sequential file locations just as you can with an indexed instruction.

On the 6502, there are no register indirect commands available. Instead, there is a very rich variety of powerful indexed instructions that mix and match address modes. There is "absolute,X" addressing. There is also "absolute,Y" addressing, "page zero,X" addressing and "page zero,Y" indexed addressing available. There are even more powerful modes that mix and match indexed, indirect, and page zero. More on these shortly.

Here are some hints on using files in programs . . .

FILE USE HINTS

1. Always set up a sample file before you write any program code.
2. It is a good idea to start a file on an even page boundary, unless space is a severe problem.
3. You should always do a range check to make sure you do not read or write outside the boundaries of a file.
4. Generally, the more files, the better. Use pointer files to access bigger files.
5. Use the shortest, most flexible, and most powerful address modes you can to access files, unless speed is a severe problem.
6. Make the allowable meanings on file values as general as you possibly can.
7. Special layout forms can ease file design. So can any powerful assembly program.

These points are fairly obvious.

If you make a model file ahead of time, it will show you how much room it will take and get you thinking about how you are going to organize and access the file. In longer machine language

programs, the files will usually take up the lion's share of the address space. More often than not, you will have short and compact code that accesses long and detailed files.

Even file boundaries, such as a file starting at location $0900 or $0A00, are a good idea to keep structure in your programs. This also simplifies keeping track of index values. Even boundaries can be slightly faster, for an extra CPU cycle may be needed if an indexed instruction crosses a page boundary. If space is extremely important, you may want to crunch the files back together end to end. But do this only late in the game and only if absolutely necessary.

A *range check* is a calculation that makes sure your attempt at reading a file or writing to it is in bounds. If you read outside a file, you will get a bad data value for your program. If you write outside a file, you could destroy the program as well. So, always be sure there is *no way* any file read or write can get out-of-range address data.

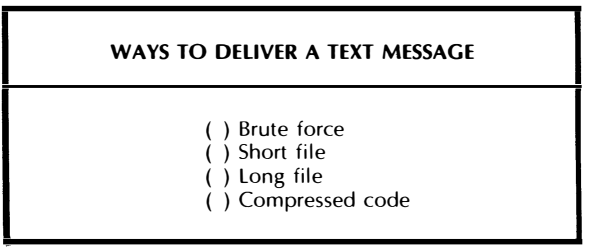
The more files you use, usually the more general your programs will be. Always think of the most versatile form of stuff you put in files. For instance, in a traffic light program using files, think of "pattern" and "delay" values in two separate files. You can do vastly more with a program that manipulates patterns and delays than just emulating traffic lights.

Special forms will help you lay out files. We will see an example shortly. If it takes a custom layout chart to simplify things for you, then do it. Create your own custom form that works. Xerox is cheaper than time.

text messages

One of the most important uses of files is to deliver *text messages*. These can be user instructions or the responses in an adventure or prompts in a business program. Almost all programs need some messages put somewhere at some time. Text files are an obvious way to do this.

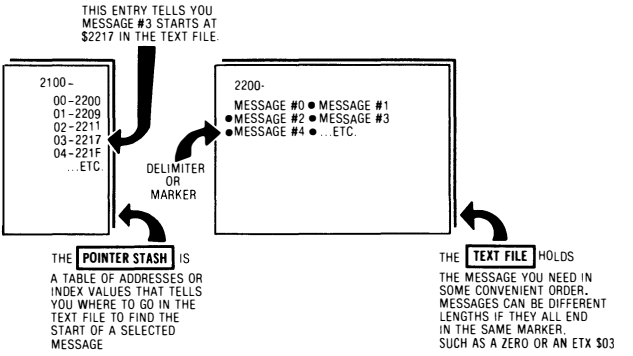
How you do a text file depends on the length of the material you want to output and how many total different messages are needed. Four obvious routes to text messages are called *brute force, short file, long file,* and the *compacted* or *compressed code file* . . .



You can use *brute force* coding for single and very short messages. For instance, if you want to output the word "*dog*," you could load the accumulator immediate with the ASCII character *d*, and send that to a port, screen, or printer. Then you could output the character *o*, followed by a *g*. Brute force coding is simple, obvious, and horribly painful for long messages.

In *short file* text messages, you keep the total number of characters to be sent under several hundred, and use indexed addressing that can handle a block of up to 256 characters at a time. If you need somewhat more than 256 characters, you can go to several blocks of 256 characters. Two files are usually involved. One is the *text file* that contains all the messages end on end, and the second is a *pointer stash* that shows where each message begins . . .

USE A PAIR OF FILES TO SEND SHORT MESSAGES



Each message ends with a *marker* or *delimiter* of some sort, such as a 00 NUL or an ASCII "End of Text" ETX or $03 command . . .

MARKER—Something to show the end of a text message, such as an ASCII control character.

DELIMITER—A pair of identical somethings that show the beginning and end of a text message, such as a pair of slashes.

Usually a marker goes only at the *end* of the message. Delimiters, though, are usually found in *pairs*, one at the start and one at the end of a message. Thus "/message goes here/" has a non-printing delimiter slash at each end.

There are lots of possible markers. For our example, we'll use the ETX, or End Of Text, marker. If you use the delimiter method instead, you put some symbol at the beginning and the end of the text. The first symbol is remembered and text is output until the second delimiter arrives. Delimiters are often used in word processing search-and-replace commands and in assemblers that provide ASCII text strings. Markers are common in adventures. Sometimes these delimiters will be used to separate /old/new/ in replacement commands.

Other markers could include a NUL or 00, which tests the Z flag "free," or a shift of the last text character from High ASCII to Low ASCII, which shortens the file by one character per message.

Anyhow, most of the text messages in a short file will have different lengths, so a separate pointer stash is used to point to the start of the first message, the start of the second message, and so on. We will see how to use a pointer stash in this discovery module.

If the messages are very long, or if you have a micro that can handle 16-bit–wide indexed instructions, you can go to "full width" or 16-bit file access. This is called the *long file* method. Your base address is sixteen bits and your index is sixteen bits, letting you hit any spot in the entire address space. On the 6502, this is best done with the upcoming indirect indexed addressing methods.

Normally, you put ordinary ASCII characters into your text file. Sometimes you will use ASCII with the most significant bit set and sometimes with the MSB cleared. This depends on the system. The Apple II normally uses the alternate ASCII code with the MSB set.

The advantages of ordinary ASCII characters are that everyone can use and understand them. This can become very important in interfacing, say, a word processor to a phototypesetter. And, of course, any attempt to hide the contents of a file through sneaky coding is not only futile but also a red-flag challenge and open invitation to others to tear into your code.

But what if your message is so long that it won't fit into your address space if it is ASCII coded? Why did it take so long to discover that the entire original *Colossal Cave Adventure* text would easily fit into an ordinary Apple all at once?

The solution to these "make it fit" questions is to use *text compression* . . .

TEXT COMPRESSION—Any scheme to make long text files take up less room than they would if encoded as ASCII characters.

An ASCII-coded character takes up one 8-bit word. This is obviously inefficient, since there aren't 256 different characters you normally would like to display. If cramming long messages into minimum space is very important, you can go to any of a number of text compression schemes.

For instance, in the *Zork* adventures, three characters are stuffed into two bytes. This is done by letting upper case characters be one 5-bit code subset, the lower case characters a second, and numbers a third. Special commands let you get between the subsets, something like the old "figures" and "letters" case shifts in the Baudot code. Since three characters now fit into two bytes, your compression efficiency is 67 percent compared to ASCII. A 10K message file takes only 6700 bytes this way.

In the *Adams* version of the classic *Colossal Cave Adventure,* pairs of characters are given unique code values using up byte values that ASCII does not need. Every time a pair of characters is replaced by a single byte, you save 50 percent of the file space you would otherwise need.

There are also worthless compression schemes involving special variable-length codes called *Huffman* codes. The more often the letter is used, the shorter its coding. But *Huffman* codes do not see too much micro use because it is extremely hard to quickly manipulate variable bit-length words with a micro.

The theoretical limit to text compression for long ordinary English texts is something like two bits per character, or around 25 percent of the space that ASCII needs. Some dictionary and speller programs actually come close to this limit. You might beat this limit by exactly matching the code compression used to the specific message to be delivered.

You should consider using text compression if you clearly do not have the room in your micro for the whole text. Another advantage of text compression is that it can eliminate the need for repeated disk access, or even can let you cut down on the total number of diskettes needed.

A much simpler text compression scheme, often overlooked, is to substitute single byte code values for things that will repeat often.

For instance, only one code value byte is needed to store hundreds of different "town, state, zipcode" strings. A single T on the bottom line of an address in a mailing list can point to a second file containing "Thatcher, AZ 85552." This is an 18:1 text compression! It's also simple and easy to do.

Anyhow, for this discovery module, we will use a standard ASCII coded short file with a companion pointer stash.

Let's put a dozen animals into our file into alphabetical order, using a custom form. Like so . . .

ANIMALS FILE

BASE ADDRESS 23 00 FILE ANIMALS 1.0

INDEX VALUE	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
MESSAGE	A	A	R	D	V	A	R	K	e_{tx}	B	U	F	F	A	L	O
CODE	41	41	52	44	56	41	52	4B	Ø3	42	55	46	46	41	4C	4F

INDEX VALUE	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
MESSAGE	e_{tx}	C	A	M	E	L	e_{tx}	D	O	L	P	H	I	N	e_{tx}	E
CODE	Ø3	43	41	4D	45	4C	Ø3	44	4F	4C	5Ø	48	49	4E	Ø3	45

INDEX VALUE	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
MESSAGE	L	E	P	H	A	N	T	e_{tx}	F	O	X	e_{tx}	G	N	U	e_{tx}
CODE	4C	45	5Ø	48	41	4E	54	Ø3	46	4F	58	03	47	4E	55	Ø3

INDEX VALUE	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
MESSAGE	H	Y	E	N	A	e_{tx}	I	M	P	A	L	A	e_{tx}	J	A	C
CODE	48	59	45	4E	41	Ø3	49	4D	5Ø	41	4C	41	Ø3	4A	41	43

INDEX VALUE	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
MESSAGE	K	A	L	e_{tx}	K	A	N	G	A	R	O	O	e_{tx}	L	L	A
CODE	4B	41	4C	Ø3	4B	41	4E	47	41	52	4F	4F	Ø3	4C	4C	41

INDEX VALUE	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
MESSAGE	M	A	e_{tx}	M	O	O	S	E	e_{tx}	N	I	G	H	T	E	N
CODE	4D	41	Ø3	4D	4F	4F	53	45	Ø3	4E	49	47	48	54	45	4E

INDEX VALUE	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
MESSAGE	G	A	L	E	e_{tx}	O	C	T	O	P	U	S	e_{tx}	P	Y	T
CODE	47	41	4C	45	Ø3	4F	43	54	4F	5Ø	55	53	Ø3	5Ø	59	54

INDEX VALUE	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F
MESSAGE	H	O	N	e_{tx}												
CODE	48	4F	4E	Ø3												

We will start our animals file at a base address $2300 and use standard ASCII with the most significant bit cleared. Our end of message marker will be an ASCII End Of Text or $03. Here is a repeat of the first few locations . . .

2300-	$41	"A"
2301-	$41	"A"
2302-	$52	"R"
2303-	$44	"D"
2304-	$56	"V"
2305-	$41	"A"
2306-	$52	"R"
2307-	$4B	"K"
2308-	$03	ETX
2309-	$42	"B"
230A-	$55	"U"
230B-	$46	"F"
230C-	$46	"F"
230d-	$41	"A"
230E-	$4C	"L"
230F-	$4F	"O"
2310-	$03	ETX

You get these ASCII file values from back in Volume 1 or out of the *Hexadecimal Chronicles* (Howard W. Sams 21802). We have shown all capital letters. For lower case letters, you would add $20 for each lower case value desired.

Now, we could access this file directly, starting at $2300 for the Aardvark message and starting at $2309 for Buffalo and at $2311 for Camel and so on. But, since the names all have different lengths and since the starting points are mostly oddball values, it is better to set up a separate *pointer stash*.

The pointer stash tells us the starting address of all the animals in the file from Animal #00 through Animal #0F.

Looking at our pointer stash on the next page, we see that Animal number 0, the Aardvark, has a pointer value of $00. Animal number 1, the Buffalo, has a pointer value of $09. This pointer tells us to start at the base address of $2300 plus an index value of $09, or 2309 for a Buffalo. To print the word Buffalo, we first decide we want Animal number 1. Then we go into the pointer stash to find the index value of the start of the message. Then we get the message out starting at $2309 and continuing till we get a marker.

Here is what the pointer stash and the flowchart for the program look like . . .

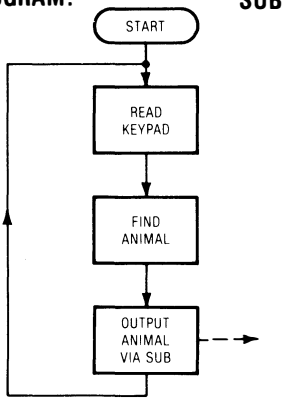
ANIMALS POINTER STASH

BASE ADDRESS 22ØØ

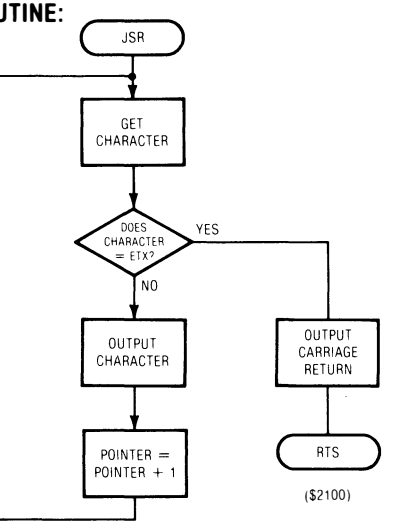
ADDRESS	ANIMAL SELECTION	INDEX POINTER	MESSAGE
ØØ	ØØ	ØØ	AARDVARK
Ø1	Ø1	Ø9	BUFFALO
Ø2	Ø2	11	CAMEL
Ø3	Ø3	17	DOLPHIN
Ø4	Ø4	1F	ELEPHANT
Ø5	Ø5	28	FOX
Ø6	Ø6	2C	GNU
Ø7	Ø7	3Ø	HYENA
Ø8	Ø8	36	IMPALA
Ø9	Ø9	3D	JACKAL
ØA	ØA	44	KANGAROO
ØB	ØB	4D	LLAMA
ØC	ØC	53	MOOSE
ØD	ØD	59	NIGHTINGALE
ØE	ØE	65	OCTOPUS
ØF	ØF	6D	PYTHON
1Ø	—	—	—

ANIMALS FLOWCHARTS:

MAIN PROGRAM:



PRINT SUBROUTINE:



And here is some code . . .

ANIMALS MAIN PROGRAM

ADDRESS	OP CODE	BYTE #2	BYTE #3	MNEMONIC	HOW?	NOTES
2000	20	7C	F6	JSR	$F67C	READ KEYPAD
2003	29	0F		AND	#$0F	MASK TO 16
2005	AA			TAX		FIND ANIMAL POINTER
2006	BC	00	22	LDY	$2200,X	" "
2009	20	00	21	JSR	$2100	OUTPUT ANIMAL
200C	4C	00	20	JMP	$2000	REPEAT FOREVER
200F	—					

ANIMALS PRINT SUB

ADDRESS	OP CODE	BYTE #2	BYTE #3	MNEMONIC	HOW?	NOTES
2100	B9	00	23	LDA	$2300,Y	GET CHARACTER
2103	C9	03		CMP	#$03	AN ETX?
2105	F0	07		BEQ	$210E	"
2107	20	0D	F9	JSR	$F90D	NO, OUTPUT CHAR
210A	C8			INY		POINTER+1
210B	4C	00	21	JMP	$2100	GET ANOTHER CHAR
210E	A9	0D		LDA	#0D	OUTPUT CR
2110	20	0D	F9	JSR	$F90D	" "
2113	60			RTS		AND GO BACK
2114	—					

To go with our code, of course, we need a pair of files. These are the pointer stash and the actual animals file. Here's a repeat of these in hex dump format . . .

ANIMALS POINTER STASH

LINE ADDRESS	LINE ADDRESS PLUS															
	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F
2200	00	09	11	17	1F	28	2C	30	36	3D	44	4D	53	59	65	6D
2210	--															

ANIMALS TEXT FILE

LINE ADDRESS	LINE ADDRESS PLUS															
	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F
2300	41	51	52	44	56	41	52	4B	03	42	55	46	46	41	4C	4F
2310	03	43	41	4D	45	4C	03	44	4F	4C	50	48	49	4E	03	45
2320	4C	45	50	48	41	4E	54	03	46	4F	58	03	47	4E	55	03
2330	48	59	45	4E	41	00	03	49	4D	50	41	4C	41	03	4A	41
2340	4B	41	4C	03	4B	41	4E	47	41	52	4F	4F	03	4C	4C	41
2350	4D	41	03	4D	4F	4F	53	45	03	4E	49	47	48	54	45	4E
2360	47	41	4C	45	03	4F	35	54	4F	50	55	53	03	51	59	54
2370	48	4F	4E	03												

We see that our main program sits on page $2000, and we have a text outputting subroutine on page $2100. The pointer stash starts at $2200, and the main Animals file begins at $2300. You could move things much closer together, of course, but it is a good idea to keep file areas larger than you think you'll need and always to start them on even page boundaries.

Several points here. We are assuming that our MYTH-1 trainer has an output port or a video display subroutine at location $F90D. Anytime we want a character to appear on screen or go out to a printer, we do a JSR to this location. Your own trainer, of course, will have a different location that does the same thing. This location receives a character value in the accumulator, does whatever it has to do to put it on a screen or out a serial printer line, and then returns to your calling program.

We read the keypad of our trainer with a subroutine located at $F67C. This subroutine automatically scans the keyboard until a key is pressed and then returns with a hexadecimal digit $00 through $0F in the accumulator. Again, there should be something similar on your micro. If you would rather input letters, and if your trainer has a full keyboard, note that subtracting $C1 from high ASCII values gives you an A=0, B=1, C=2, and so on.

Now, you could have read the keypad yourself as part of your calling program, but if you check into the monitor of almost any trainer, you will find all sorts of ready to go *utility subroutines* that can be adapted to your program . . .

Most trainers and all personal computers have ready-to-go utility subroutines in their monitor that do all kinds of useful things for you.

USE THEM

And, while you are at it . . .

DOING IT:

Make a list of all the available utility subroutines in a micro trainer and personal computer of your choice.

Then show how you use these utility subs and how you pass variables to and from them. What registers are used? What is destroyed? What ends up where?

Our program is short and simple. We start with a JSR to read the keypad. The keypad returns with a number 0-F in the accumulator, but just to be darn sure, we do a range check by ANDing against an $0F mask. This forces the answer to be in the safe range of the sixteen values of the pointer stash.

Next, we transfer this animal number to the X register and load the Y register indirectly from the pointer stash at $2200. This puts the starting index value for any animal into the Y register for us. The LDX,Y indexed command is one of the sneakiest ones available on the 6502. If your micro can't do this, then get the index value out of the pointer stash, put it into the accumulator, and then move it where you can use it. TAY ought to be helpful here.

The animal number goes into the X register and picks a value out of the pointer stash. The pointer stash value then goes into the Y register, where it becomes the index value to begin printout of the actual animal. Thus, you put your keypad value into X and use this to get Y from the pointer stash. You then use Y to find the start of the message.

Then we reach into the main animals text file at $2300 plus Y and get the first character of our animal. We do this with an indexed "LDA,Y" command that goes to the sum of the base address and the Y register.

Next, we test to see if this is the last character. If it is not, we output the character by jumping to the utility subroutine in the monitor that outputs characters for us.

Again, if it is not the last character in the animal string, we increment Y to tell the index to move on to the next character. Then we branch back and get the next letter from the file and output it.

We keep reading the file and outputting characters until we get to an ETX marker. At that time, we stop outputting characters, provide a carriage return, and then jump back to reading the keyboard again.

If you press a "0" on the keyboard, you should get an Aardvark on the screen. A "1" gives you a Buffalo, and so on. This is a random access file, since you can get any animal in any order by hitting the numbers at random.

If this was a real part of a real program, the main program might pick the animal by loading the X register with an animal number. Then it would jump to a similar message printing subroutine. At the end of the subroutine, it would return to the main program until it was time to print another animal.

If your trainer offers absolutely no way to display characters (Not even a Teletype output? Come on now!), switch to a personal computer for this particular discovery module. The Apple II does this one beautifully, but no, I won't tell you here how to put characters on the display or read the keyboard. Find it out for yourself. Or

check into the *Enhancing Your Apple II* series, particularly Enhancement 3, in Volume 1 (Howard W. Sams 21846).

DOING IT:

Rework your files so that your module now gives you vegetables instead of animals.

Do NOT make any program changes!

Several points here. First, the NUL, or 00 marker, is easier to use than the ETX that I've shown, because it is testable "free" with a BNE branch.

Second, the file values do not have to be in any particular order as long as each pointer correctly points to the start of its correct text string. Thus, you can easily add to the end of your file.

Third, note that you can also put prompting or default strings into your file for easy error checking. Use messages like "Please try again," or "That's not a letter, Turkey!" Try it.

The hidden nasties in this enhancement include learning new and powerful indexed instructions or their equivalents, finding out how to design and use files, discovering how to work with existing utility subroutines, and picking up some hands-on practice using the ASCII code.

Should you not want to use pointer stashes, there are lots of alternate ways to set up your files. You can stash a number that tells how long each file is as it comes up. Or, you can put all the files that are the same length in the same area in memory. Another possibility is to count markers, but this gets old fast and takes forever. You can even make each file entry a fixed record length and count records. But this gobbles up memory. You can also shorten long files into a shorter and separate *key,* and then mess only with the key if you need sorts or whatever. This is sometimes called ISAM, short for the *Indexed Sequential Access Method.*

getting fancy

So far, we have only looked at the simpler indexed instructions that are available on the 6502. There are also two very complicated and very mind-blowing classes of instructions that combine indexed addressing, page zero addressing, and indirect addressing. These two instruction classes are the innermost keys to the 6502's success. Let's see just what they do.

The two names are very confusing. One is called *indirect indexed* and the other is called *indexed indirect*. To keep them apart, note *where* the word "index" appears in the name and use the terms *"pre-indexed"* and *"post-indexed" instead* . . .

INDEXED INDIRECT (Pre-Indexed)—A very rarely used 6502 address mode that lets you jump to any of a long list of address pairs on page zero.

LDA (06,X) is an example.

The index picks the address pair. The CPU then loads from the address stored in 06 + x (position) and 07 + x (page) to continue the program.

INDIRECT INDEXED (Post-Indexed)—A very common and extremely powerful 6502 address mode that lets you jump to an address that is the sum of a page zero base address and a Y index register value.

STA (06),Y is an example.

The op code picks the base address located in 06 (position) and 07 (page). The Y index gets added to the base address, and the data is stored at that address.

Here's one way to remember which is which . . .

Confusing, no? Let's try the oddball pre-indexed op code first. Take LDA ($06,X) as an example. Note the parentheses that are *around the whole works*. This says we go to the *address* that is the sum of $0006 *plus* the X index value, get what is stored there, and save that value as an address low or position byte. Then we go on to the very next location and get and save that value as an address high or page byte. We then load or store indirect to or from that address. Adds, subtracts, compares, and logic are also done the same way.

What this lets you do is calculate a bunch of address pairs and then selectively use one of the pairs to find something. This is handy in processing certain forms of interrupts and has other sneaky uses. Unfortunately, page zero is usually far too valuable for space gobbling uses like this. So forget indexed indirect unless you really want to get into the 6502 in a big way.

But the opposite address mode of indirect indexed, or post indexed, is so powerful and so useful that you must learn to use it if you are to become 6502 literate. Say we have a STA (06),Y command. Note the parentheses *only around the page zero address.* This says to find the low address in $06 and the high address in $07, *then* add the Y value to it, and then store the accumulator in that location.

What this lets you do is calculate a base address so that one program can service many different files in many different locations. This also solves the dilemma of reaching more than 256 bytes at a time, for all you have to do is increment the base address *pair,* and you can continue all the way through the entire 64K address space.

A simpler use for indirect indexed is to let Y = 0. Then you have faked a load or store indirect, something not otherwise available on older 6502s. Once again, you can also add, subtract, compare, or do logic.

Anytime you work with several blocks of files in a 6502 program, or involve yourself with calculated addresses, you will need the indirect indexed instruction. So . . .

DOING IT:

If your trainer is from the 6502 school, complete all indexed indirect and all indirect indexed cards for those commands you already know.

If not, complete all cards for all exotic and powerful address modes useful for working with files. Do cards only for those commands you already know.

The ultimate test of whether or not you are a 6502 machine language freak is whether you can understand and use the indirect indexed or post-indexed addressing mode to its full power. Can you handle it?

A test . . .

DOING IT:

Create a file several thousand characters long, made up of the typical adventure responses such as "The Iron Statue greedily wolfs down the antique eggplant."

Then create a pointer file that holds pairs of absolute addresses that point to the start of each response.

Then create a "full width" text file access program that uses the 6502's indirect indexed commands to exchange an input message number for an output text message.

Do this, and you are into the very core of 6502 machine language programming.

As an example of how to do "double wide" file manipulation, pick a pair of page zero addresses, say $06 and $07, to hold an indirect indexed address for you. Go to your pointer file and get your low starting address and put it in $06. Go to your pointer file again and get your high starting address and put it in $07. Set the Y register to #$00. Then do an indirect indexed LDA ($06), Y to get the first character out of your text file. Test each character for a marker. If this character is not a marker, output the character and increment the address pair at $06 and $07. Do this by always incrementing $06 and then incrementing $07 only on a zero result of the $06 increment.

Alternately, you can increment Y for each additional character in a response, provided that each *individual* response is less than 256 characters long.

Be sure to reset Y to 00 when you finish.

"Double wide" file maneuvers will start you on *any* address in the 64K address space and will output a message of *any* length. There is no 256-character file length limit as there was with the earlier short file work. Also, you can now use *one* subroutine to handle many different files, just by changing the addresses in $06 and $07 to point somewhere in new file.

There are lots of other uses of text files. For instance, you can put control commands into a text file and use these commands to read a disk drive, to turn a printer on or off, to change modes of an intelligent daisywheel, or to imbed typesetting commands.

While there are many different types of files and ways to use them, if you understand the basic short and long text files, you should be able to handle anything else in the way of files as the need arises.

It pays to go out of your way to work files into all of your programs in one way or another. Files are so powerful and so general and so convenient that they should be one of your main programming tools.

INTERRUPTS

Many microcomputer programs simply run on forever, looping to themselves. Others will do something as a subroutine or as a service to another program and then return to a main control program or to an operating system or a system monitor. But sometimes we like to set up programs so that an outside world event can get their attention.

One common way that an outside world event can get the microcomputer's attention is with an *interrupt* . . .

> **INTERRUPT**—An outside world event that requests or demands the microcomputer's attention.

An interrupt is a programming tool that combines hardware and software to let outside world things "borrow" the microcomputer's CPU for a while. Normally, one program will be running until it is interrupted. At that time a new program begins that *services* the interrupt. After that service program is finished, the old program picks up where it left off.

Interrupts differ from subroutines in that a subroutine is code that is run when called from *part of a running program.* An interrupt is code that starts running *when an outside world event requests or demands attention.*

As an example of an interrupt, suppose you want to water some trees for an hour every second day. Now, you could set up a microcomputer program that runs continuously, patiently waiting 44 hours, outputting a single ON command for six CPU cycles to a latching solenoid water valve, waiting another four hours, and then outputting a single OFF command to the same valve, again taking another six CPU cycles.

This is obviously dumb, because the computer is completely tied up waiting nearly forever for something that it can handle very quickly.

Instead of going this route, suppose you build an external hardware timer. This could be a bits-and-pieces CMOS divider chain, a single IC real-time clock, or even—heaven forbid—a backgear timing motor and a microswitch. By using interrupts, the computer can do many other things until it is time to water. Then the computer gets interrupted from what it was doing long enough to service the interrupt and water the trees. It then promptly goes back to what it was originally up to.

So interrupts are one very efficient way to handle random outside world events, particularly those that are unpredictable, changing or don't show up very often.

There are at least three different types of interrupts. These are called *masked interrupts, non-maskable interrupts,* and *system resets* . . .

MASKED INTERRUPT—An interrupt line on a microcomputer that can be ignored or accepted under software control.

NON-MASKABLE INTERUPT—An interrupt line on a microcomputer that demands immediate attention and must not be ignored.

SYSTEM RESET—A special, top priority non-maskable interrupt that gets a microcomputer started on power-up, automatically initializing essentials, and going to a valid starting point in a program or monitor.

Maskable interrupts are the ones most commonly used. With a maskable interrupt, there will be an interrupt flag that lets you accept or ignore interrupts. If you decide to accept an interrupt, the instant the interrupt arrives, the microprocessor finishes the program instruction it is working on and then shoves enough information on the stack to remember where it was. The microprocessor then jumps to a point that services the interrupt. At the end of the interrupt, a return command unwinds the stack and picks up where it left off in the main program.

If you ignore the interrupt, you can go on computing as long as you like. If the interrupt is still there or still trying when you decide to accept it, you service it at that time. Otherwise, the interrupt gets ignored. Important maskable interrupts should be *hardware latched to provide handshaking,* so they will still be there when you get around to using them.

A *non-maskable* interrupt cannot be ignored. You must service it as soon as you can safely stop your program and save some return information on the stack. Non-maskable interrupts are less commonly used. Often, the single step and debugger code in a system monitor will usurp the non-maskable interrupt line on a 6502 trainer, preventing you from conveniently using this line for your own purposes.

System reset is a special, no-holds-barred interrupt that drops everything and returns you to an orderly start-up sequence. Depending on how it is set up, the system reset can put you in the monitor, into an advanced operating system, or into the correct starting place in a program of your choice.

There are two important uses of system reset. The first is to get the system up and started on the right foot when it is first powered up. Remember that a microcomputer is always executing some code somewhere. If that code is random garbage, you get garbage-quality results. System reset bypasses the garbage and gets you started on the correct program at the correct point.

The second use of system reset is to recover from programs that have bombed or otherwise gone astray. Some system resets are "gentle" and try to save as much of your code and program conditions as they can. Others are violent and put you back on square one. Properly designed system reset code should give you the option of a *cold start* that sends you back to square one or a *warm start* that gently returns control back to your program for you.

Each micro family has a different approach to how many interrupts there are and how they are used. These approaches can often be classified into one of two possible systems. One system is called the *polled interrupt* and the other is the *prioritized interrupt* . . .

POLLED INTERRUPT—A software intensive interrupt system where there is one interrupt line and software to ask each possible interrupt source if it needs attention.

PRIORITIZED INTERRUPT—A hardware intensive interrupt system where there are many interrupt lines and hardware to decide which line is the most needful of service at any time.

One example of a polled interrupt is that old "stop the train!" cord . . .

Just as any passenger can pull the emergency stop cord, a polled interrupt allows any outside world event that needs servicing to get the CPU's attention. If there is only one thing in the system that can interrupt the CPU, then we are home free, for we know what is doing the interrupting.

On the other hand, if there is more than one possible source of interrupts, we have to go to some software routine that goes to each interrupt source in turn and asks that source "Was it you that wanted attention?" This is called polling, just as a delegation at a political convention is polled by being asked one at a time what their vote was.

The actual polling is done by reading bit lines on a port or by talking directly to a peripheral chip. Details vary with the application.

Usually, some interrupts will be more important than others. Some things must be serviced immediately, while others may be able to wait around for a while. For instance, a "send me more text" interrupt from a printer can usually be near the bottom of the pile, because a slight delay in hard copy output probably is no big deal. To place different values on different polled interrupt sources, you simply interrogate the most critical lines first and give them top priority.

Polled interrupts can use a hardware idea called daisy chaining to enforce which interrupt gets serviced first. To do this, a more important interrupt will "pull the plug" on lesser ones when it needs some action. The daisy chain is reconnected after the important interrupt is serviced. The Apple II has a seldom-used hardware

daisy chain on its I/O connectors set up so that slot 0 is top dog and slot 7 is tail-end Charley. Hardware plugged into any slot can break the part of the chain routed to higher numbered slots, and its own requests can in turn be broken by lower numbered slots.

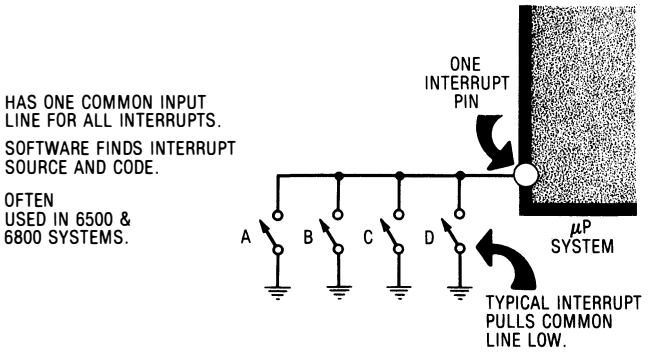
The advantage of polled interrupts is that they leave you with very simple hardware, often only one wire and a single package pin. The disadvantage is that you have to use software to sort out who is doing the interrupting if you have more than one possible interrupt source.

Prioritized interrupts do the opposite. In a prioritized interrupt system, there are lots of package pins, each of which accepts an interrupt line. The interrupt lines are given an interrupt priority. Often, the lower the number, the more important the interrupt. For instance, a system reset might be put on interrupt line 0 since a reset normally is the most important possible outside world event.

The advantage of prioritized interrupts is that each individual interrupt is ready to go with no additional polling software needed and the priorities of each interrupt are always known. The disadvantage is that this method ties up a lot of hardware and package pins that can be put to better uses, especially if you are not going to use lots of interrupts.

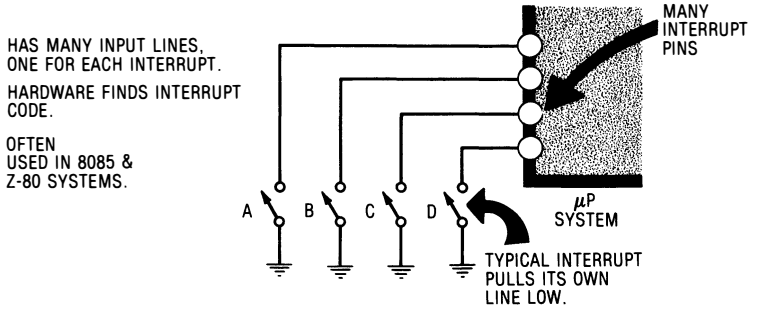
Let's look at the difference in connections between polled and prioritized interrupts. Here is the setup for polled interrupts . . .

POLLED INTERRUPTS



And here is how you handle prioritized interrupts . . .

PRIORITIZED INTERRUPTS



The 6502 and 6800 schools tend to use polled interrupts with their few package pins and use software testing to find an interrupt source. The 8080 and Z80 schools often add a system chip called an 8259 that gives you eight or more prioritized interrupt lines ready to go. But note that you can design a 6502 system with prioritized interrupts, or an 8085 system with polled interrupts, if you want. It's just not mainstream.

DOING IT:

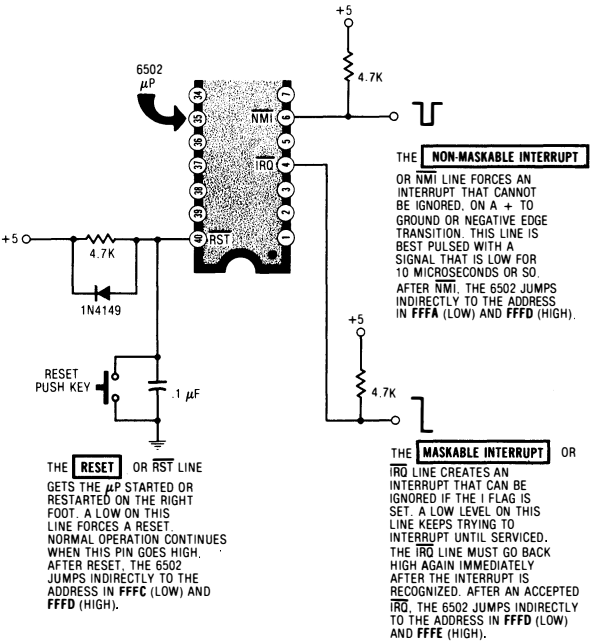
Find out how many of what type of interrupt lines are available on the microprocessor of your choice, including their signal polarities, use rules, and address locations. Then, find out how many interrupts are available on the trainer of your choice, how these lines are used and addressed.

Note that there is a big difference between the way the micro*processor* handles interrupts and the way a complete micro*computer* system handles them. Let's look at a specific example.

On the 6502, there are three interrupt lines that go into the CPU package—called $\overline{\text{RST}}$, $\overline{\text{NMI}}$, and $\overline{\text{IRQ}}$. All of these lines are normally held high. To activate them, you briefly bring one line low as needed. For instance, to do a system reset, you bring the $\overline{\text{RST}}$ line low for a fraction of a second. To do a maskable interrupt, you briefly bring the $\overline{\text{IRQ}}$ line low. To do a non-maskable interrupt, you briefly bring the $\overline{\text{NMI}}$ line low. Rules and conventions change with other micros.

Here's a diagram of the interrupt pins on the 6502 . . .

THERE ARE THREE DIFFERENT INTERRUPTS ON THE 6502



The resistor and capacitor on the RST line gives you an automatic reset interrupt on power-up. When you first apply power, the capacitor is slow to charge, and the system is held reset until the $\overline{RST}$ line goes high enough to release the interrupt. At that point, the computer starts off on a normal program. Should supply power be removed, the capacitor quickly discharges through the diode so it is ready to go on a fresh restart.

The technical details of the 6502's IRQ and NMI differ slightly. IRQ must be held low till serviced. NMI is edge sensitive, with a high to low transition, or negative edge, causing a NMI request. Fine points like this, of course, change with each microprocessor.

One important and two-edged rule . . .

Requests on interrupt lines MUST be short enough that they do not interrupt themselves or get serviced more than once!

BUT

Requests on interrupt lines MUST last long enough that they can be recognized!

Suppose you yank an interrupt line low and hold it low. The computer interrupts itself. And then it interrupts the interrupting program. And the program keeps interrupting itself forever, or at least till whatever is yanking the interrupt line low goes away. This is clearly ungood.

Suppose instead that you only briefly pulse the interrupt line low and your program is presently ignoring interrupts. The interrupt gets missed. This is also ungood.

How you attack this detail depends on your system, how many interrupt sources you have, and whether you ever need to temporarily prevent interrupts from happening.

For instance, if you are always going to accept interrupts and only have one interrupt source, you can simply use a brief pulse of ten microseconds or so to pull the $\overline{\text{IRQ}}$ line low. This always starts your interrupting program. The first command in the interrupting program can be a SEI, or set interrupt, which acts as a lockout to prevent interrupts for a time that is at least long enough to let the interrupt go away. The interrupt flag and mask is later released with a CLI command.

As a second example, if you have only one interrupt that may have to wait till you can handle it, you *latch* a set-reset flip-flop or some other memory element and hold the interrupt line low till it is acknowledged. When you acknowledge the interrupt, one of the first things you do with the interrupt code is unlatch the interrupt line. This is a typical example of *handshaking*.

If you have more than one interrupt source and have times when you don't want to be interrupted, you have to add external hardware to keep track of who is on first. Details vary with the system.

We'll see an example of how to handle a single interrupt in our next discovery module. If you are confused on handling interrupts, just find some way to obey the rule in the previous box and everything should work out correctly.

Note that some microcomputer operations must not be interrupted under any circumstances. For instance, an interrupt while writing to a disk drive can destroy the diskette's contents.

Interrupting *any* timing loop will lengthen the loop time by the amount of time needed to service the interrupt. Watch this detail.

What happens on an allowed interrupt?

The CPU first finishes up the instruction it happens to be working on. It then does a jump indirect to an address stored somewhere. There is a different address stored somewhere for each interrupt or reset line that goes into the CPU. Details vary with the micro family.

On the 6502, there are three interrupts. An NMI or non-maskable interrupt jumps to the address stored in $FFFA low and $FFFB high. The RST or reset interrupt jumps to the address stored in FFFC low

and FFFD high. Finally, an IRQ or maskable interrupt jumps to an address held by FFFE low and FFFF high.

Like this . . .

6502 CPU INTERRUPT ADDRESSES

$\overline{\text{NMI}}$— jumps to the address held in FFFA low and FFFB high.
$\overline{\text{RST}}$— jumps to the address held in FFFC low and FFFD high.
$\overline{\text{IRQ}}$— jumps to the address held in FFFE low and FFFF high.

Note that these 6502 locations must be ROM or permanent memory. Reset must know where to reset to, even on a newly powered system. This is one of the main reasons that 6502 systems put RAM on the bottom of the address space and ROM on top.

But, still in a 6502 system, you probably will want to change where your IRQ programs and your NMI programs sit. To handle this relocation problem, the monitor or the operating system will give you a RAM location that you can change. The fixed ROM location puts you into a monitor routine that does a jump indirect to the new RAM location. This solves the dilemma of having your interrupts always at fixed locations and yet flexibly programmable.

Some fancy systems have what they call *cold starts* and *warm restarts* . . .

COLD START—A way of bringing up a micro that safely puts you in the monitor or operating system.

WARM RESTART—A way of bringing up a micro that returns you as gently as possible to your running program.

On a *cold start*, you go into the monitor or operating system. This gets you off on the right foot. One of the first things the monitor does is check some RAM location for some magic value. If that value is not there, the monitor concludes that this is a cold start and proceeds from there. The monitor then puts the needed magic value into the special RAM location. So long as the power stays on

and the magic value stays in the RAM location, any future resets will be warm restarts.

On a *warm restart*, you can be returned safely to your program without dropping into the monitor or operating system.

The locations and rules for interrupts on a microcomputer system are usually much different from those of a microprocessor's CPU. For instance, interrupt lines will now be available on pins on expansion sockets and connectors. You will also have RAM locations set aside that let you change the interrupt and reset vectors. This lets you reset directly to your operating program, rather than the monitor, and lets you decide where your interrupt program code is going to sit. What really happens is that the CPU goes to ROM to find a fixed monitor location and then does a jump indirect to wherever your RAM told it to go.

Often on a 6502 system, the RAM addresses needed will also "just happen" to end in $FA through $FF. Thus, on the KIM-1 trainer, the RAM interrupt vectors sit from $17FA through $17FF, while on the Apple II, they lie from $03FA through $03FF, in just the order you would expect.

If your system is not from the 6502 school, the general idea will be the same but the locations and use rules will change. You might have many interrupt lines of progressively lower priority. Each of these will *vector* via an address stored in ROM to a monitor or other *service* program that will then do a jump indirect via some RAM locations that you can change to go where you want.

If you have more than one interrupt source, note that interrupts can interrupt other interrupts, and you have to keep track of who is on first and who just struck out.

In simpler micro systems, you are better off not using interrupts at all unless they really free up CPU time or solve some other big problem. A single interrupt is preferable to many, but each system is different and may take a different approach.

We have already seen how there is a SEI flag that sets the mask to prevent interrupts in a 6502, and a CLI flag that clears the interrupt flag or mask to allow interrupts. You should already have these cards on hand.

Doing an interrupt is more or less automatic and is only slightly more complicated than using a subroutine. When you service an interrupt, the CPU has to finish what it is working on and then must store the return address low and high onto the stack so it knows how to pick up where it left off. This is the same as the subroutine jumping process.

But that's not all. An interrupt servicing sequence in the CPU *also* has to save the processor status or phlag register. By knowing the stopping address and the condition of all the flags, you are well on

your way to picking up exactly where you left off. Should you want to save the accumulator or other working registers, you can do so on your own, as the earliest part of the interrupt program code. You then restore what you saved in reverse order as the final part of the interrupt code. You can save what you need onto the stack or elsewhere in system RAM.

Instead of a subroutine return, we now have an interrupt return . . .

RTI	RETURN FROM INTERRUPT	40

(IMPLIED addressing)

1 Byte — 6 Clocks

RTI — All flags restored

Stops the interrupt program and returns to the main program by restoring the program counter and the flags off the stack.

Assume an interrupt occurred just before instruction $04F5.

$2AA2– 40 Resumes the main program at $04F5 with all flags restored.

Which leads us to an obvious but very important rule . . .

Just as the last instruction in a SUBROUTINE has to be an

RTS

the last instruction in an INTERRUPT has to be an

RTI

Interrupts don't necessarily have to come from the outside world. They can also come from add-on cards or from companion peripheral chips external to your microprocessor, and they can even be faked by an experienced programmer.

For instance, if you add a peripheral chip that has a *timer* on it, the timer can demand the CPU's attention whenever it wants to. This is one very useful way to handle short to medium sized time delays without tying up your CPU. A *real-time clock* chip can also use the interrupt line to demand attention after some time delay. Real-time clocks more often involve themselves with longer time delays and events that have to be locked to people-type times or dates.

Interrupts can also be used to tie two or more micros together and are one possible way to let several different users take turns interacting with the same microprocessor system.

Finally, an experienced programmer can force a "fake" interrupt to stop a program somewhere or to debug or test code. This is called a *break*. More on this shortly.

But first, let's get some practice using interrupts with our final discovery module . . .

DISCOVERY MODULE 9

BURGLAR INTERRUPT

Use interrupts to model a simple burglar alarm.

The main program is to delay 20 seconds after starting to simulate letting you leave.

Each grounding of the interrupt line is to sound an alarm for five seconds and light an LED.

Anytime you work with interrupts, you must have a *main program* and some *interrupting code*. This is somewhat like using subroutines, where you put the main code somewhere and some subroutine code somewhere else.

But, unlike subroutines, the computer has to be told ahead of time where the interrupt code will be found. The rules will change with the microprocessor and the microcomputer system, but one of the first things your main program has to do is teach the computer where to go if it gets interrupted.

To repeat . . .

When you use interrupts, you need code for the main program and separate code for each interrupting program.

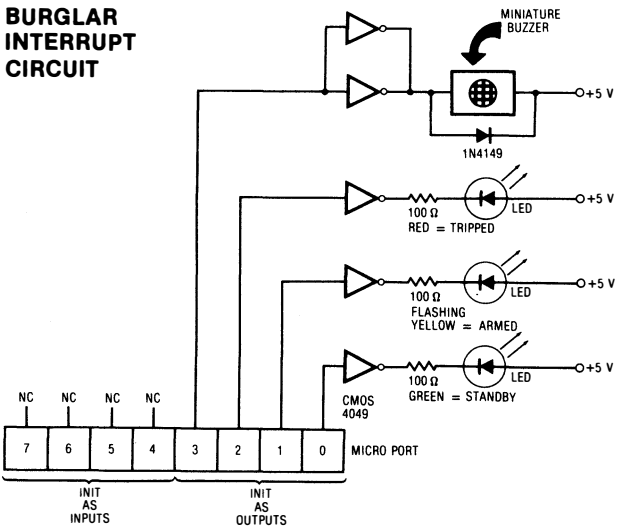
Your main program also must tell the micro where to find the start of the interrupting code.

Let's think about our hardware first. We will assume that a momentary grounding of the interrupt line will trip the alarm. To get fancy, we will use a green LED, a yellow LED, and a red LED, along with a small electronic buzzer.

When you first run your program, only the green light should go on. No alarms should be allowed for the first 20 seconds so that you can leave the building without tripping the alarm. After the first 20 seconds, the lights should switch to a flashing yellow, meaning the alarm is armed but not tripped.

If the alarm is tripped, the buzzer should sound for 5 seconds, and a red LED should be permanently lit. After the 5 seconds, the red light should stay on, and the flashing yellow should continue. If the alarm is tripped again, the same action should repeat.

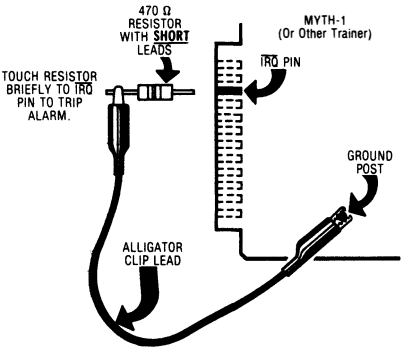
The interface circuit looks like this . . .



More details on interface circuits in the next chapter. We use CMOS inverters here to "amplify" the port signals to make them "loud" enough to drive the LEDs and the buzzer. Note that the polarity of the LEDs is critical and you have to provide series current limiting resistors. The diode around the buzzer prevents spikes and must point in the "backward" direction shown. The buzzer should be a small electronic one that works on minimum current. You may have to parallel two or more CMOS inverters to get enough buzzer drive.

We will simply touch a grounding wire to the interrupt request line of our trainer to simulate an alarm trip . . .

TRIPPING THE ALARM:



What you do is ground the $\overline{IRQ}$ pin on the trainer's I/O connector through a small resistor. Briefly touching the resistor to the $\overline{IRQ}$ pin should trip the alarm.

Several important points.. First, if your trainer does not have a good ground post or terminal, it is best to provide one. Do this both for this module and as a convenient place to tie your scope ground. Second, the actual location of the IRQ pin will, of course, change with your choice of trainer. Be sure to read the schematic or user's manual to find the right pin on the correct connector.

Some 44-pin connectors on some trainers are labeled 1 through 22 on one side and A through Z on the other. On the lettered side of a 22-pin connector, there is no G, I, O, or Q pin. Note that the pin on the top of a connector usually has a different signal on it than the same pin on the bottom. Do not short these out with a clip lead or you can destroy your trainer!

Be sure to keep the resistor leads very short, and be sure never to ground any connector pin directly.

Hence, this warning . . .

> **If you use a wire to ground the interrupt line, be VERY careful not to short any adjacent pins or traces!**

The best place to find the $\overline{IRQ}$ line is on one of the expansion or I/O connectors on the trainer, rather than trying to directly short the pins at the microprocessor chip.

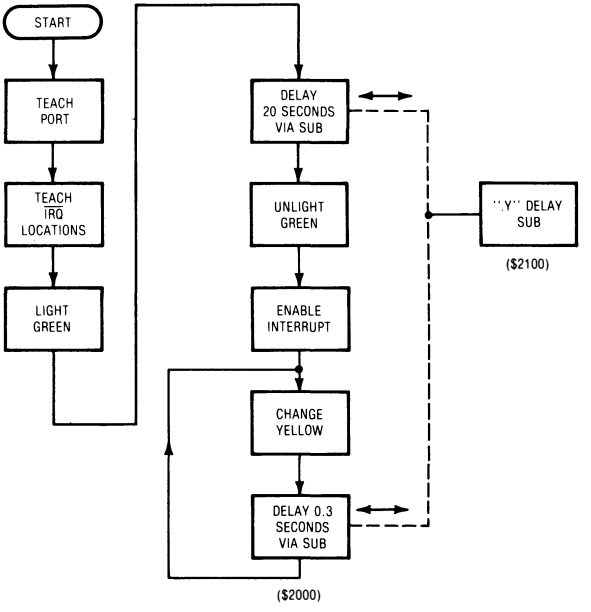
Depending on where you short pins on a trainer, you can do anything from destroying the program to destroying the machine.

Be careful!

We will use a main program starting at $2000, a .Y-seconds delay subroutine starting at $2100, and our interrupt code that will start at $2200.

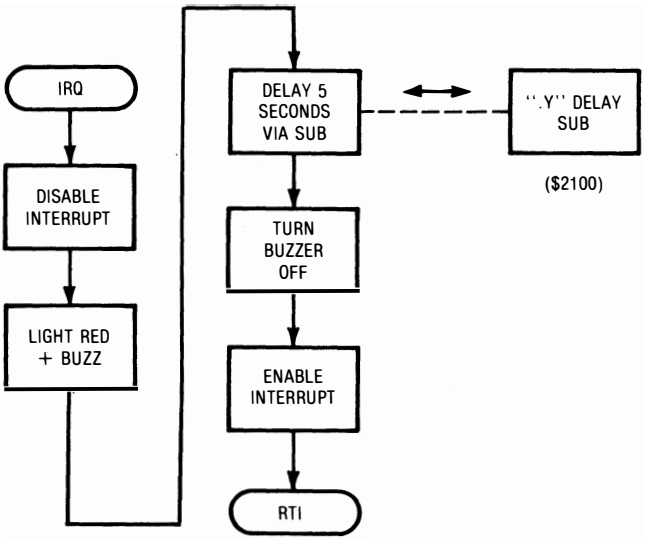
Here is the flowchart for the main program . . .

BURGLAR INTERRUPT
MAIN FLOWCHART:



And the interrupt code flowchart . . .

BURGLAR INTERRUPT IRQ FLOWCHART:



($2200)

And this is the main program code . . .

BURGLAR MAIN CODE

ADDRESS	OP CODE	BYTE #2	BYTE #3	MNEMONIC	HOW?	NOTES
2000	78			SEI		LOCK OUT ALARM
2001	A9	0F		LDA	#$0F	TEACH PORT
2003	8D	80	C0	STA	$C080	" "
2006	A9	00		LDA	#$00	CONNECT IRQ CODE
2008	8D	FE	02	STA	$02FE	" "
200B	A9	22		LDA	#$22	" "
200D	8D	FF	02	STA	$02FF	" "
2010	A9	01		LDA	#$01	LIGHT GREEN
2012	8D	81	C0	STA	$C081	" "
2015	A0	C8		LDY	#$C8	DELAY 20 SECONDS
2017	20	00	21	JSR	$2100	VIA SUB ".Y"
201A	A9	00		LDA	#$00	UNLIGHT GREEN
201C	8D	81	C0	STA	$C081	" "
201F	58			CLI		ALLOW ALARM
2020	AD	81	C0	LDA	$C081	CHANGE YELLOW
2023	49	02		EOR	#$02	" "
2025	8D	81	C0	STA	$C081	" "
2028	A0	03		LDY	#$03	DELAY 0.3 SECONDS
202A	20	00	21	JSR	$2100	VIA SUB ".Y"
202D	4C	20	20	JMP	$2020	CONTINUE FLASHING
2030	—					

This code starts at location $2000. You will also need a .Y delay subroutine like we used before, starting at location $2100. Refer back to Discovery Module 6 for the correct code. Do not forget to load your main program, your subroutine, and your interrupt code! Forget any single part and your program bombs for sure.

Our program first initializes itself by ignoring interrupts and by teaching the CPU where the interrupt code is to begin. We will assume our imaginary trainer has to have the IRQ interrupt address stored at $02FE low and $02FF high. This crucial detail will, of course, change with different trainers.

Initialization continues by teaching a port that it has a green LED line zero, a yellow LED line one, a red LED line two, and a buzzer line three. Note that we have no port inputs to this program. We are going to input an alarm on the interrupt line instead.

The program then turns on the green light and starts a 20-second delay. We will use the .Y-second time delay to stall for us. Using both an interrupt and a subroutine in the same program should clear up the differences between the two, but it does introduce a very sticky problem, on which we will see more shortly.

After the timeout is complete, the interrupt is enabled, and the green light is turned off. Next, the program goes into a flashing yellow mode, and continues this way until interrupted or shut off. We flash yellow by carefully changing *only* port line one every 0.3 seconds. We want the yellow to flash regardless of whether the red trip light is on or off, and we must not change any bit but the yellow one. The yellow flasher also uses the subroutine for delay values.

Here is the interrupt code . . .

BURGLAR INTERRUPT CODE

ADDRESS	OP CODE	BYTE #2	BYTE #3	MNEMONIC	HOW?	NOTES
2200	78			SEI		LOCK OUT ALARM
2201	A9	0C		LDA	#$0C	LIGHT RED + BUZZ
2203	8D	81	C0	STA	$C081	" " "
2206	A0	32		LDY	#$32	DELAY 5 SECONDS
2208	20	00	21	JSR	$2100	VIA SUB ".Y"
220B	A9	04		LDA	#$04	TURN BUZZER OFF
220D	8D	81	C0	STA	$C081	" "
2210	A0	01		LDY	#$01	FIX DELAY SUB
2212	58			CLI		ALLOW ALARM
2213	40			RTI		AND GO BACK
2214	—					

An interrupt arrives when the $\overline{\text{IRQ}}$ line is briefly grounded. If the 20-second startup timeout is complete, the microcomputer will then go to the start of the interrupt code by checking the address in

the magic interrupt slots. For our MYTH-1 trainer, the interrupt checks into slots 02FE and 02FF and finds a $2200 address pair there that tells us where the interrupt code begins.

The first thing the interrupt code does is set the I flag to prevent the interrupt from interrupting itself. Then we permanently light a red LED, and temporarily turn the yellow LED off. We also start the buzzer buzzing at this time.

After our buzzer starts up, we timeout 5 seconds using the .Y subroutine, and then shut the buzzer back off. Presumably, you want to scare the burglar away without buzzing all night. The interrupting code finally releases the I flag and returns to the main program with an RTI.

After release, the main program resumes what it was doing, namely flashing the yellow light. In fancier problems, the main program would most likely go back to doing some other high level task that might be totally unrelated to the interrupting code.

You start the program by running the main code. Only the green light should light. Twenty seconds later, green should switch to flashing yellow. You trip the alarm by momentarily grounding the $\overline{\text{IRQ}}$ line through the safety resistor. The program is working if alarm trips are ignored for the first 20 seconds, and then the next trip should light the red LED and make the buzzer squawk. The buzzer should sound off for 5 seconds, but the red LED should stay lit. Further alarm trips should only buzz the buzzer.

Note how the CPU's operation switches automatically from main code to interrupted code and back again. Note also how either block of code seems to be free to use a subroutine. See how the sub automatically returns to whatever called it, while the interrupt automatically goes back to the main program.

or does it?

Ah, but there's a bug in the works. And not one you could call a feature either . . .

DOING IT:

Replace the LDY #$01 at the end of the burglar interrupt code with a pair of NOPs and rerun your program.

The yellow bulb should now stay "stuck" and refuse to flash until half a minute after each alarm.

Why?

Remember how we said there were *local* variables and *global* variables? If you ever get the two mixed up, you are in deep trouble. *Local variables must be kept local, and global ones must be available at any time for any use without interference.*

What happens in the burglar interrupt is that the interrupt code uses the subroutine for the 5-second buzzer timeout. It exits the sub with a zero in the Y register.

But the main program might have been using the *same* subroutine *at the time it was interrupted.* Instead of the 0.3-second yellow delay, you end up going all the way around and then some, for a full 25.6 seconds of delay.

Your interrupt has destroyed three values needed by your main program. These are the number of tenths remaining in the Y register and the two subroutine counter locations INNRLP and OUTRLP. Try to pick up where you left off and funny things happen.

You must be very careful that interrupts do not change anything you need your main program . . .

You MUST be VERY CAREFUL that an interrupt's code does not change or destroy any values located anywhere that the main program might need!

In this example, it's the shared subroutine that causes the problem. In other cases, you might be using page zero locations for some use in the main program and for something else in the sub. Other times, you will alter registers or flags in an unexpected way.

Here are three possible ways around shared location problems . . .

WAYS TO KEEP VARIABLES LOCAL

( ) Keep everything separate
( ) Save and restore
( ) Minimize the damage

Keeping everything separate is one sure way to keep local variables truly local and safe. This means that the interrupt code should

use its own variables in its own unique locations and should not change or alter any registers needed by the main program, *including the accumulator*. In this module, we could write a separate delay subroutine, but we would still have a problem with the accumulator.

The *save and restore* technique tells you to save all registers and variables that you may need somewhere else as the first thing you do with your interrupt code. It's easiest to shove the accumulator, X, and Y onto the stack. You can do the same with other variables, such as INNRLP and OUTRLP, or you can move copies of these to safe locations.

You should then unwind all the saves and put everything back where it belongs as the *last* thing the interrupt code does. Save and restore is far and away the best possible technique to use. But there's a time penalty, since extra bytes and clock cycles are needed to carry out the save and restore process.

Minimizing the damage is a very dangerous technique. The four things damaged by the subroutine are the accumulator, the Y register, INNRLP, and OUTRLP. These last three aren't damaged by the main interrupt code, but are damaged by the subroutine the interrupt uses, giving the same nasty result.

Since our burglar alarm is a somewhat sloppy use of a micro, we can let INNRLP and OUTRLP end up too big or too small, and all it does is give you a slightly longer tenth of a second the first time back around. But the Y value is another matter, since we would get the full timeout of 25.6 seconds with an interrupt return with a Y value of zero. So, as a sloppy fix, we reset Y to 0.1, making the first yellow flash somewhat short and then the rest normal.

And the program seems to work all right with this sneaky two-byte LDY #$01 patch.

But the fact remains that our interrupt has altered four values that are supposed to be local to the rest of the program.

So . . .

Do NOT minimize damage to "fix" shared variable problems.

All this does is drive the real problems deeper and makes them vastly harder to find.

Instead . . .

DOING IT:

Rewrite the burglar interrupt code to save and restore the A, Y, INNRLP, and OUTRLP values.

some further interruptions

It turns out that there are up to five parts to a program that uses interrupts. Here they all are . . .

INTERRUPT PROGRAM PARTS

(1) The main program.
(2) Code early in the main program that connects the interrupt vectors.
(3) Any subroutines used by the main program.
(4) The interrupting code.
(5) Any subroutines used by the interrupting code.

Needless to say, if you forget to load any one of these parts, your program will not work. Be sure to load all parts of a program anytime you use it.

And, if any problems show up, always reload *all* five parts to make sure that what you think is in the machine is really there.

If you have several different interrupts, you have to set up some way to find out who is doing the interrupting. If you have a polled interrupt system, then you have to start reading some port inputs somewhere to find out just who is trying to get your attention. In a prioritized interrupt system, your interrupts will normally arrive on different package pins, and each will jump to the address you taught it to go to at the start of your main program.

Want to do more with interrupts? How about this . . .

DOING IT:

Work out a safe way to interrupt your computer either 60 or 120 times a second from a power line reference, using one of the circuits of the next chapter.

Then show how you can build a light dimmer, a clock, and an irrigation timer.

To sum up, interrupts are outside world events that demand or request the computer's attention. Interrupts can in fact be outside world signals or they can come from peripheral or timer chips, or even from the programmer for debugging. Important types are the maskable interrupt, the non-maskable interrupt, and the system reset.

There are two common ways of handling multiple interrupts that depend on the microprocessor in use. In a polled interrupt, there is one interrupt pin, and the interrupts are software checked via port lines to see who demands attention. In a prioritized interrupt, there are many hardware input lines and each goes to its own special address. Each interrupt is vectored to a special address in a special location that begins the interrupting code. The interrupting code ends on a Return From Interrupt, or RTI command.

Interrupts are best used where long time delays or random outside world events must interact with the microcomputer system. Use of interrupts frees up the microprocessor chip from single-minded tasks of waiting around till something happens.

The hidden nasties in this module involve using interrupts, telling the difference between interrupts and subroutines, properly initializing "magic" address locations in a system, keeping variables local, and picking up more practice with simple hardware interfacing.

breaks and breakpoints

What could you do if you were able temporarily to *force* an interrupt exactly when and where you wanted to in any program? This would be a very powerful debugging tool, since you could then stop the program, and check to see what each and every register and address space location of interest was up to. Single stepping gets old fast when you have to go through long loops or find out what is happening well into a program. With this *forced interrupt*, though, you could stop the program at any place and at any time.

One way you can do this is to add a fancy address decoder. When you hit the instruction that contained this address, the output of the decoder would reach around and trip the interrupt line. You might interrupt yourself back into the system monitor. You can then go to special code that does special things for you. You might even execute some custom code, such as a "new" op code, and then pick up where you left off in your main program.

A fake or programmer-created interrupt is such a needed tool that just about any microcomputer system will do it for you without any extra hardware at all. We call this *software-driven interrupt* a *break*, and the place in the code where the break takes place is called a *breakpoint* . . .

BREAK—A software-driven interrupt used for debugging or testing.

BREAKPOINT—The place in the program where the break code is put.

One older use of the word *break* involves World War II fighter planes. The command here means to drop what you are doing since you are about to be shot down. In ham and CB radio communications, the *break* command means that someone else wants to interrupt or break into the conversation.

A break in a computer program means essentially the same. Stop the program immediately and go to the monitor or some other interrupting program useful for testing or debugging.

Practically all microcomputers have a special op code to force a software-driven interrupt. What you do is replace a legal op code with the break op code. When the microprocessor gets to this point in a program, it immediately acts as if it were interrupted.

The 6502 break code is a double zero. Like so . . .

BRK **FORCE SOFTWARE INTERRUPT** **OO**

(IMPLIED addressing)

1 Byte — 7 Clocks

BRK — Sets B and I flags

Forces an interrupt and jumps to the address stored in $FFFE low and $FFFF high. Sets B and I flags.

Assume $FFFE = $07 and $FFFF = $FE

$2AAA– $00 Forces a software break by jumping to $FE07. Sets B and I flags.

Now, if you do not have any external interrupts in your system, BRK will be their only possible source, and you are home free. Just use the IRQ address to point to where you want to go on a break. Some micros, such as the 6800, keep the BRK and IRQ vectors separate for you. The 6502 does not.

But, what if you do have "real" interrupts in use as well as "fake' software-driven break interrupts? Each microcomputer has to have some way to tell the two apart.

On the 6502, the BRK code sets a special flag that is called the B or Break flag. If you have to, you can read this flag as the first part of your interrupting code. If it is a "real" interrupt, the break flag is cleared, and you go on and service the interrupt as it is supposed to be. If you have more than one "real" interrupt, you then poll by reading inputs ports lines till you find the culprit. If you have a "fake" interrupt, the break flag is set, and you can now go on with your debugging. Other CPUs behave differently.

There is no quick way to read the break flag on the 6502. Instead, you move the flag register into the accumulator by bouncing it off the stack with a PHP and PLA command pair. You then do an AND #$10 to mask out the B flag. A zero result means the break flag was zero. You only have to go this route if you are trying to tell breaks from real interrupts.

Another card . . .

DOING IT:

If your trainer is from the 6502 school, complete the BRK card at this time.

If not, complete all cards for all instructions that let you do breakpoints or other software-driven debugging aids.

For practice in using breakpoints, replace the first instruction in the delay loop of Discovery Module 4, the audio tone using a subroutine, with BRK, and set up the IRQ vectors to point to your monitor. Single step the program, and you should enter the monitor where you used to enter the subroutine loop.

If you want to do something with a breakpoint and then return to your main code, you will start executing code at the breakpoint *plus two* on the 6502. Thus, if you want to continue, *the next byte after a breakpoint is skipped*. Watch this detail if you get fancy with your breakpoints.

One sneaky use for breakpoints lets you add your own custom op codes to the microprocessor of your choice. What you do is enter a BRK, followed by two or three code bytes. You teach the micro to BRK to some code of your own, rather than the system monitor, and then process the code bytes as needed. For instance, you can add

all of the fancy 6809 op codes to the 6502, or provide most any other "op-code" command you like.

Op code patches like these are sometimes called *macros*. Macros work like real op-code instructions but may take more code and may need much longer to execute.

Be sure to learn how to use breaks and breakpoints since they are such a powerful and friendly tool. If your trainer is old enough or cheap enough to lack a single step feature, the breakpoint will be absolutely essential to debug code. In any trainer or personal computer, break debugging is most helpful and very valuable.

what? no math?

We have now gone through nine discovery modules that should have led you through practically all the op codes of the micro of your choice. Some of you may have noticed that we did not seem to pick up much in the way of micro arithmetic along the way. Why not?

Mostly to drive home the point that math really isn't all that important for many different microcomputer uses.

Now, if you want to get into a heavy math trip with micros, you certainly can. Micros offer incredible new vistas for math freaks as well as powerful and elegant tools for attacking problems. So, have at it. There is nothing to stop you. But please don't let any personal math hangups keep you from learning about and understanding microcomputers.

To review, we've seen that there are four levels of arithmetic normally done with microcomputers. These are the *bit level*, the *word level*, the *multi-word level*, and the *algorithm level*.

Bit level math is plain old logic, and we now know how to do things like AND, OR, and EOR bits together. Word level involves the addition and subtraction of single 8-bit words to get a 0 to +256 result in straight binary or a -127 to +127 result in 2's complement signed binary.

For more accuracy, we can do addition with pairs or even groups of words, taking a carry or a borrow from the least significant word pair and adding or subtracting it to the more significant pair of words, and so on. This way you can get any amount of precision you need.

The algorithm level lets you combine the usual addition and subtraction instructions with shifts, logic, and all the other microcomputer op codes to get a fancy result. Once again, most micros do not have any immediate way to multiply, divide, find square roots, or do trig stuff. Instead, you repeat all the simple op codes over and over again in a *pattern* needed to get you a final result.

There are lots of alternatives to brute force machine-level coded math . . .

ALTERNATIVES TO BRUTE FORCE MATH

( ) Steal the plans
( ) Use table lookup
( ) Use higher level language
( ) Add dedicated hardware

Almost every ordinary math operation has already been worked out for most popular microcomputers. Many of your typical programming books will be very happy to either bore you, confuse you, or overwhelm you with all the gory details. Public domain program libraries literally overflow with poor to barely useful ways of handling nearly any math problem. Many micro magazines will also cover math programs in depth.

Standard algorithm libraries are expensively available from such heavies as the ACM and IEEE as well.

So, if your main goal is to get a math result, rather than build personal skills in mathematical programming, then go steal the plans from someone else. Then use them. Don't reinvent the wheel.

As to the morality of this, just play fair. Using one source is plagiarism; two or more is dedicated research. As long as your personal value added dominates your programs, feel free to adapt and use the ideas of others. Ideas and concepts are worth a dime a bale in 10-bale lots. It is only the final *conversion* of ideas into a useful program or product that is profitable or genuinely useful.

It's not creative unless it sells.

Table lookup is a very powerful way of doing oddball math things like trig functions, logs, and so on. What you do is use a file to get immediately from problem to solution. Table lookups are extremely fast, compared to any method of calculating anything complex. For instance, say you wanted to plot a circle on a graphics screen. Normally, you will only need 8-bit accuracy for this, since most screens are less than 256 lines high. So, instead of taking forever while calculating the square-root-of-the-sum-of-the-sine-and-cosine-squares, just look up the values in the table and use them directly. Then scale the answer to size with a second table lookup that acts as a multiplier. Some sneaky tricks are needed to cut any tables you use down to reasonable sizes, but that just adds to the fun.

If your micro has BASIC or another higher level language available, then consider using that high level language to do any of the fancy math stuff for you. This works out very well if you only need an occasional calculation and can live with the pitifully slow speed of high level languages. Sometimes a mix of high level language and machine language subs will give you the best combination of speed

and flexibility. Other times, your machine language programs can directly borrow the math subroutines from the higher level language.

Another alternative to math-intensive micro problems is to add some custom hardware. Two examples. You can add a fast multiplier chip to do a hardware multiply or divide for you. You could also add a fancy calculator chip or "math function" integrated circuit and unload the fancy calculations from your micro's CPU. Check *TRW* for multipliers, and *National, Intel,* or *AMD* for dedicated math chips.

Well, we have weasled around on this long enough. It turns out that there are usually only two micro instructions available that do arithmetic. These are *add* and *subtract*. Both are usually available in many different address modes.

The *add* usually works by getting a value from someplace, adding that value to whatever is already in the accumulator, and putting the result back into the accumulator. *Add* instructions are usually provided with lots of different address modes, so you can either add an immediate constant value to the accumulator or add anything in the address space to the accumulator.

Some micros give you a choice of using or not using the carry flag with your add commands. If you do not use the carry, there is no way to extend the addition beyond eight bits, but the carry flag is not altered, and can be safely used for other things. The 6502 always uses the carry flag for its addition and subtraction instructions.

Here is a 6502 card for add immediate . . .

ADC **ADD VALUE TO ACCUMULATOR** **69**

(IMMEDIATE addressing)

2 Bytes — 2 Clocks

ADC #$06 — C,N,V and Z flags

Adds the value of the second byte of the instruction to the accumulator and the carry flag. Then puts the result in the accumulator. The carry flag sets on a result > 255. N and Z flags work normally. The V flag sets on a result > 127. Used to add a constant value to the accumulator.

Assume that the accumulator holds an $AE.

2C34– 69 9F — Adds $9F to the $AE already in the accumulator, leaving us with $4D in the accumulator, sets C and V flags, and clears N and Z flags.

The add immediate instruction takes what is left in the carry flag from previous work and adds that to the accumulator. Then it adds what is in the second instruction byte to the accumulator. Then it puts the results of this addition back into the accumulator. The N and Z flags behave in the usual way. The carry flag will set on a result greater than $FF, and the V flag will set on a result greater than $7F.

If you are working in straight binary, you can ignore the V flag. The C flag carry result will be important to you only if you are using two or more words for "double width" addition. We have already seen that you must know the state of the carry flag before you add . . .

The state of the carry flag MUST be known before doing any addition or subtraction!

Usually, you CLEAR the carry flag before you do an ADDITION.

Usually, you SET the carry flag before you do any SUBTRACTION.

If you add with the state of the carry flag unknown, you will get an answer that could be either correct or else high by one. You can also start adding with a *set* carry flag and add *one less* than normal to get a correct result, but such sneakiness can return to haunt you later.

The V or oVerflow flag is important only when you are doing 2's complement signed binary arithmetic, and then only on the most significant 8-bit word you are using. On an 8-bit add, the V flag will set on a result greater than 127. Detailed examples of 2's complement arithmetic are shown in Chapter 1 of the *6502 Programming Manual*.

There is also an SBC or SuBtract with Borrow instruction in most microcomputers. The carry flag is still used, with the usual assumuption that *a set carry is a cleared borrow*. This one usually works by doing an internal 2's complement and then adding. As long as you are subtracting small numbers from big numbers, and as long as you expect a positive or zero result, you can subtract in straight binary the same way you add. It is only when you expect or need negative results that you have to go to 2's complement signed binary.

Let's wrap up the cards . . .

DOING IT:

If your trainer is from the 6502 school, complete the ADC and SBC cards for all address modes at this time.

If not, complete all cards for all instructions that do addition or subtraction at this time.

We are purposely not going into details of double width 2's complement arithmetic here. Subtraction is fairly rare in micros. Even when it is done, it often involves a single 8-bit word and a positive result.

By the way, the original 6502 does not have a DEA or Decrement the Accumulator command. Some other micros do. If yours does not, you can fake a DEA command by doing an SEC and then an SBC #$01. The result will be one less than what was in the accumulator before. Remember that a set carry flag is treated as a cleared borrow and vice versa.

Unless you have missed a card or two somewhere along the way, this should have got us through all the instructions for the 6502. In your particular micro, there may be cards and even address modes left over.

So . . .

DOING IT:

If your trainer is from the 6502 school, pick up any cards you may have missed at this time.

If not, investigate all remaining address modes and then do all remaining cards.

You should now have a complete set of all cards for your micro and a complete understanding of how to use each card. You will probably have discovered by now that the value in the cards is more in *creating* them than in *using* them later. In writing a card by hand, you are forced to think about exactly what the card does and how it is used. That, of course, is the whole point in the "those #$!$#

cards" method and the secret to thoroughly learning and understanding machine language programming.

If you want to change microprocessor families, just go back and repeat everything with a new set of cards of a different color. Once again, this method works with *any* microcomputer from *any* family, present or future.

As a reminder, our discovery modules aren't true programs, for there is far more that goes into writing a program than simply punching some code into the machine. We will find out one good way to write real programs in Chapter 9. This method is called the *Micro Applications Attack*. We'll also find out where to go next.

Lets now put all of our interface and I/O stuff together in a big pile and call it Chapter 8.

things they never tell you in computer school

SOME PROGRAM MUSTS

What does it take to write and successfully market a software program today? I count eight absolute musts:

(1) **The program MUST run in machine language.**
Check into the "top thirty" listing of any major microcomputer, and you'll find programs that run in machine language sweep the listings, thirty to zip. This is because machine language is far and away the fastest, most powerful, and most flexible way of doing things. BASIC need not apply.

(2) **The program MUST be written by someone who is a top-notch programmer and knows the application area cold.**
If you are only a programmer, you will write a program that is totally worthless when people actually try to use it to solve real-world tasks on their terms. If you are only an expert in your field, then you will write a program that self-destructs and trips all over itself. In short, if you have never sharpened a *Pulaski,* don't try to write forest fire simulation software.

(3) **The program MUST be user friendly.**
This means that an absolutely minimum number of keystrokes are used to do anything and that all commands are in a logical and self-consistent order. It means menu-driven programs, or other easy-to-use structures that smoothly and swiftly move you from program

area to program area. It means talking in English, rather than computerese. It means NEVER dropping your user into an operating system, losing a file, or hanging the machine. It means unlocked and readable standard disk codes and formats, easily and conveniently available to other programs. User Friendliness means full and total recovery from the dumbest possible inputs entered at the worst possible time.

(4) **The program MUST be unlocked.**
There are only four things that locking and copy protection does for a program: (1) It hacks off and inconveniences the legitimate users of the program; (2) It raises the price of the program; (3) It diverts time and energy that should have been spent improving and testing the main program . . .
. . . and, of course, (4) It very dramatically increases the number of bootleg copies of the program in circulation, because of the superb entertainment and unbeatable education that cracking copy protection provides.

(5) **The program MUST be fully documented and disclosed.**
Decent documentation, including complete source code listings for all machine language programs, is an absolute must. Users demand the absolute and inherent right to modify and adapt a program to suit their needs. They must be provided with all the information they can possibly use and then some.

(6) **The program MUST be beta tested.**
There is no way that a program can be written by a single author and immediately marketed. Instead, a large group of knowing but more or less unbiased outsiders must thoroughly test the program to see what bugs exist in the program and find out how people outside of the thought patterns of the original author are going to interact with the software. Two beta test methods include trusted reviewers and captive clubs.

(7) **The program MUST be well supported.**
The program author must be willing and able to directly handle any program problems that crop up, for at least two years after program release. This means, at the very least, continually monitored telephone voice and modem hot lines, along with published upgrade and update information that is either free or priced at printing and distribution costs.

(8) **The program MUST be fun to use.**
This is what makes all of CP/M so dreary and so downright awful. Remember that EVERYTHING run on a computer is a game. The only difference between the *Zork* and *Visicalc*adventures is that the high score in *Visicalc* appears in *The Wall Street Journal* instead of *Softline*. A winning computer program MUST be fun to use. Since the program is, by definition, a game, that game must be fun to play and must be winnable. If your users aren't enjoying themselves, all is lost.

Index

A

Absolute indirect addressing, 81
Absolute long addressing, 81
Absolute short addressing, 73, 90, 208
Accumulator, 22
Accumulator indirect addressing, 81
ADC card, 300
Address
  base, 84, 261
  bus, 37, 39
  flasher, 319
  toggler, 319, 321
Address modes, 10, 63, 86
  indexed indirect, 277
  indirect indexed, 277
Address space, 10, 15, 20-21
  rules, 13
Addressing
  absolute indirect, 81
  absolute long, 70, 89
    commands, 72
  absolute short, 73, 90, 208
  accumulator indirect, 81
  block move, 95
  immediate, 68, 88
    commands, 69
  implied, 65, 87
    commands, 66
  indexed, 83-86
  indirect, 80-81, 92
  page zero, 208
  register indirect, 81
  relative, 76-77, 91
    commands, 78
    relocatable, 95
Addressing—cont
  virtual, 95
A/D input converter, 399
  multiple slope, 402
  types, 401
Algorithm, 304
Amplifiers, circuit level, 373
AND card, 239
Animal project, 269
Apple II, 50, 54, 275, 284, 349
Approximation, 227
Architecture, 31
  microcomputer, 32
  microprocessor, 32
  Von Neumann, 117
Assembler form, 104
Assembly, 146
Audio Tone (Discovery Module 4), 186

B

Barrel shifting, 239
Base address, 84, 261
Baud rates, 363
BIT card, 248
Bit twiddlers, 238
Blocks, data, 115
BNE card, 180
Bottom-up programming, 423
Bounce, 396
Branch, 135
Break, 302
Breakpoint, 302
BRK card, 302
Burglar Interrupt (Discovery Module 9), 283

Bus
address, 37, 39
control, 41, 43
data, 37-38
lines, 43
multiplexed, 39
Byte
high address (page), 17, 38
low address (position), 17, 38

C

Cards, those #$!$#, 125, 130
ADC, 300
AND, 239
BIT, 248
BNE, 180
BRK, 302
CLC, 178
CMP, 246
DEX, 192
JMP absolute, 136
JSR, 208
LDA, 158, 261
NOP, 135
PHA, 202
ROR, 243
RTI, 290
RTS, 209
STA, 159, 217
TAX, 156
Carry flag, 175
Chips
serial I/O, 365
"more than a port," 368
Circuit level amplifiers, 373
Circuit level interface, 312, 371, 391
Circuits
CMOS, 315, 354
integrated, safety rules, 317
LSTTL, 315, 354
signal levels, 316
CLC, 66
CLC card, 178
Clock
cycle, 168
frequency, 168
CMOS circuits, 315, 354
CMP card, 246
Code
designer friendly, 220
position independent, 79
user friendly, 220
Coding, straight line, 187
Cold start, 288
Commands
absolute long addressing, 72
immediate addressing, 69
implied addressing, 66
logic, 238, 242
relative addressing, 78
teaching, 329
Conditional instruction, 136
Control bus, 41, 43
Contact bounce, 396
Converter, A/D input, 399
Converter, D/A, 387
CPU, 34, 46

D

D/A converter, 387
companding, 390
multiplying, 390
Darlington transistors, 377
Data blocks, 115
Date bus, 37-38
Data files, 222
Debouncing, 393
Debugging, 147, 210
Decoding, 44
Decrementing, 192
Delay loop, 188
Delimiter, 267
DEX card, 192
Dice project, 360
Direct I/O, 14
Disassembly, 146
Discovery Modules, 126, method,140
1. Tail Byter, 140
2. Figure Eight, 150
3. Square Deal, 155
4. Audio Tone, 186
5. Pitch Reference, 206
6. .Y Time Delay, 219
7. Nite Lite, 251
8. Text Outenblatter, 257
9. Burglar Interrupt, 283
Dumping, 144

E

8080, 345
8085, 56
8048, Imsai, 53, 59
Electronic hand tools, 109

F

Fancy ports, 327
Figure Eight (Discovery Module 2), 150
File, 115
Files, kinds of, 258
  data, 222
  random access, 259
  sequential access, 259
  use hints, 265
Flags, 172
  carry, 175
  negative, 174-175
  6502's, 176-177
  zero, 174
Flowchart, 127
Forms
  assembler, 104
  hex dump, 105, 145
  machine language programming, 103
  simplified I/O diagram, 345, 443
Frequency, 165
  clock, 168
  units, 165
Frobozz, 115-116

G

Glomper, 108
Grabber, 108

H

Halt, 43
Hand tools, electronic, 109
Handshaking, 281, 287, 399-341
Hex dump forms, 105, 145
HP 5036, 345

I

If instruction, 180, 185
Immediate addressing, 68, 69, 88
Implied addressing, 65, 66, 87
Imsai 8048, 53, 59
Incrementing, 192
Index value, 84, 261
Indexed addressing, 83-85, 93
Indexed indirect address mode, 277
Indexed sequential access method, 276
Indirect addressing, 80-81, 92
Indirect indexed address mode, 277
Initialization, 161, 296, 327, 344
Input conditioning, 392, 396
Instruction
  conditional, 136
  machine, 115
  unconditional, 136
Instruction times, 169
Integrated circuit safety rules, 317
Integrated circuit signal levels, 315
Intel 8212, 335-339
Interface, 311
  circuit level, 312, 371, 391
  input and output, 366
  micro level, 313, 314
  people level, 313
  system level, 313
Interrupt, 43, 280
  addresses (6502), 288
  masked, 281
  non-maskable, 281
  polled, 282, 284
  prioritized, 282, 285
  program parts, 300
I/O
  diagram, simplified, 345, 443
  direct, 14
  memory mapped, 14

J

JSR card, 208
Jump, 135
JMP absolute card, 136

K

Keyboard, scanning, 353

L

LAN controllers, 369
LDA card, 158, 261
Listener probe, 195
Listing, 144
Load, 156
Logic analysis, 153
Logic commands, 238, 242
Loop, 188
  delay, 188
  use rules, 189
  within loop, 220

LSTTL circuits, 315, 354

M

Machine language programming, 114
  form, 103
Marker, 267
Masked interrupt, 281
Memory map, 32
  detailed, 48, 54
  simplified, 48
Memory mapped I/O, 14
Menu driven program, 121-122
Micro Applications Attack, 407-422
Micro level interface, 312, 314
Micro toolkit, 99
Mnemonic, 132
Modules, Discovery, 126, 140, 150, 155, 186, 206, 219, 251, 257, 283
Move, 156
Multiplexed bus, 39
MYTH-1 discovery trainer, 126

N

Negative flag, 174-175
Nesting, 190
Nite Lite (Discovery Module 7), 251
Non-maskable interrupt, 281
NOP, 130
NOP card, 134
NPN transistors, 376
Numeric analysis, 230

O

Op code, 131
Open collector outputs, 370-371
Operand, 132
  symbols, 132
Optocoupler, 383, 394
OR instructions, 241
Oscilloscope, 107, 163
Output conditioning, 392, 396
Output isolation, 383
Outputs, open collector, 370-371

P

Page zero addressing, 208
Parallel ports, 312, 325, 329
Passing variables, 231
People level interface, 313
PHA card, 202
Phlag register, 173
Pipelining, 71, 137
Pitch Reference (Discovery Module 5), 206
Pointer, 25
Pointer, stack, 28, 204
Pointer stash, 271
Polled interrupt, 282
Ports
  input and output, 325
  latched output, 333
  parallel, 313, 325, 329
  serial, 312, 325, 362
  simple and fancy, 327
Port lines, minimizing, 352
Position independent code, 79
Processor status register, 173
Prioritized interrupt, 282, 285
Program, 115, 120
  blowups, 123
  counter, 27
  form rules, 141
  menu driven, 121-122
Programmer's model, 32, 55
Programming
  bottom-up, 423
  machine language, 114
  stickiest box, 415, 423
  top-down, 423
Protecting diode, 379
Protocol, 339
Popping, 203
Pulling, 203
Pushing, 203

Q

Q option, 190

R

RAM, 14
Random access file, 259
Reading, 12
Re-entrant code, 208
Register indirect addressing, 81
Relative addressing, 76-77, 78, 91
Relative branch timing, 184
Relative branch value, 181
  block counting method, 182-183

Relative branch value—cont
official math freak method, 184
Resolution, 388
Resource sheet, 97
ROM, 14
ROR card, 243
RTI card, 290
RTS card, 209
Registers, types of, 21
address, 26
data direction, 344
flag, 29
index, 23
phlag, 173
processor status, 173
Registers, working, 10, 20-21

S

Scanning keyboard, 353
Schmidt triggers, 397-398
Serial I/O chips, 365
Serial ports, 312, 325, 362
Sequential access file, 259
Settling time, 388
Setup time, 334-335
Sideways shovers, 238, 245
Signetics 490, 381
Simple ports, 327
Simplified I/O diagram, 345
form, 443
Single stepping, 147
6551, 366
6800, 58
6502, 57, 114, 168, 288
6530, 348, 368
6522, 342-345, 348, 351
Sneak path, 357
Soft switch, 179, 319, 322
Spike protector, 378
Sprague 2813, 381
Square Deal (Discovery Module 3), 155
STA card, 159, 217
Stack, 198
use rules, 200
Stack pointer, 28, 204
Start, cold, 288
Stash, 115
Stickiest box programming, 423
Store, 156
Straight line coding, 187
Subroutine, 206
uses, 207
Subroutines, utility, 274
SYM-1, 51, 347
System level interface, 313
System reset, 281

T

Tail Byter (Discovery Module 1), 140
Task times, 169
TAX card, 156
Teaching commands, 329
Testers, 238
Text compression, 268
Text Outenblatter (Discovery Module 8), 257
Time measurements, 166
Time multiplexing, 357
Time period, 165
Toolkit, micro, 99
Top-down programming, 423
Trainers, 101
MYTH-1, 126
Transfer, 156
Trap, 137
Triac, 385
Tri-state drivers, 331

U

Unconditional instruction, 136

V

Value, index, 261
Variables
global, 233
local, 233, 298
passing, 231
rules, 232
Von Neumann, 117

W

Warm restart, 288
Working registers, 10, 20-21
Writing, 12

X

X-Y matrix circuits, 357

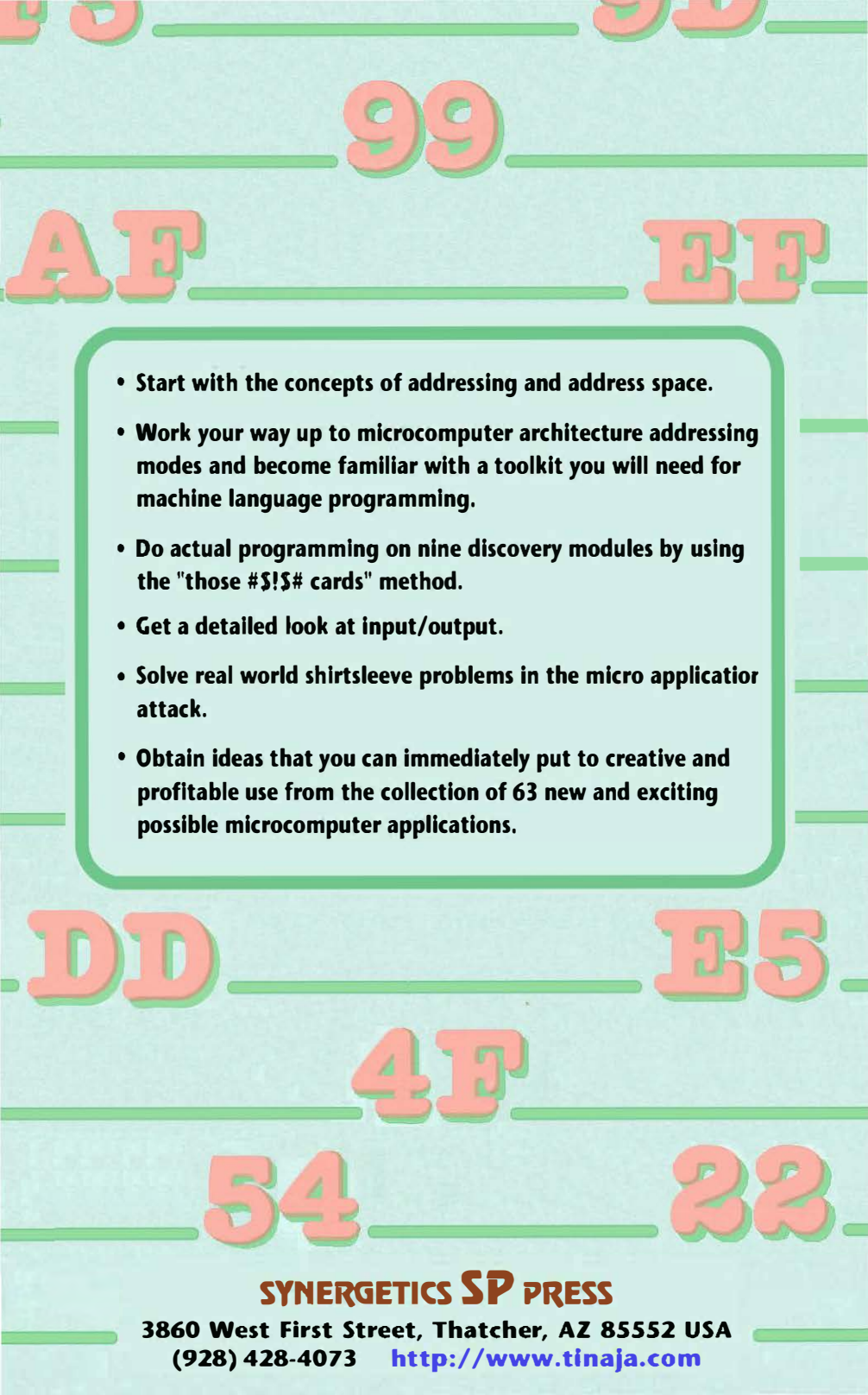
Y

.Y Time Delay (Discovery Module 6), 219

Z

Z-80, 52, 97
Zero flag, 174

This book continues as...
< http://www.tinaja.com//ebooks/MLP2cb.pdf >



- Start with the concepts of addressing and address space.
- Work your way up to microcomputer architecture addressing modes and become familiar with a toolkit you will need for machine language programming.
- Do actual programming on nine discovery modules by using the "those #$!$# cards" method.
- Get a detailed look at input/output.
- Solve real world shirtsleeve problems in the micro applicatior attack.
- Obtain ideas that you can immediately put to creative and profitable use from the collection of 63 new and exciting possible microcomputer applications.

SYNERGETICS SP PRESS
3860 West First Street, Thatcher, AZ 85552 USA
(928) 428-4073 http://www.tinaja.com