POLY 88 MICROCOMPUTER SYSTEM

VOLUME II: OPERATION AND SOFTWARE

TABLE OF CONTENTS

POLY 88 Microcomputer System Manual
Vol. II: Operation and Software

A: Introduction to the POLY 88.

The POLY 88 system is designed to be, not only a powerful problem-solver, but also a source of satisfaction and enjoyment. Sophisticated computer users know that computers are interesting as well as useful. To derive the greatest possible value, both practical and aesthetic, from the system, it is important to have both a ready ability to interact with the computer at the level of keyboard and screen and a good sense of what is going on inside the computer at the level of actual electronic events. This volume is intended to show the user how to operate the system, and to convey some measure of awareness of what is going on inside the computer as the user operates it.

You may be quite advanced in your familiarity with computers, or you may just be getting started. Our discussion should be understandable and helpful to the beginner, yet interesting and worthwhile to the expert. We will be moving quickly through the fundamental concepts of symbol systems -- binary and hexadecimal math and ASCII code -- used by computer users, and some useful "languages," especially assembly language, with which we communicate with computers. Then we will consider how a computer accepts, stores, and manipulates data to produce results. Then we will be operating the POLY 88 to see how it works.

If you already have some experience with computers, you will be mainly interested here in reading about those things that are unique to the POLY 88 -- its architecture and its monitor. You will probably just want to skim through Section A.

Section B will give you some hands-on experience with the POLY 88, while Section C and the appendix present in tabular form the information you will want to refer to often.

If your experience with computers is more limited, you will want to go through Section A with care. It discusses binary math, etc., in a way intended to provide some insight into what actually goes on inside the computer. Computer users should be able to picture in their minds what the computer is going through in response to their commands. Indeed, the computer usually performs its operations so fast and replies to its operator so promptly that the operator may have no sense of anything going on between his pushing a key and the appearance of the response on the screen. This "lack of sympathy" is undesireable, because it causes operators to miss much of the aesthetic value of the computer, and in fact prevents them from making full use of the computer (especially a micro-computer like the POLY 88).

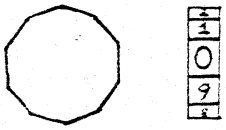
The relatively inexperienced computer user may find that our discussion is occasionally hard to follow. He or she will want to re-read and interpret, refer to other texts, and discuss the subject with other people. The authors of this text, however, are determined to make it understandable to the beginner, yet interesting to those who are more advanced.

1. Symbol Systems

a. Number Systems

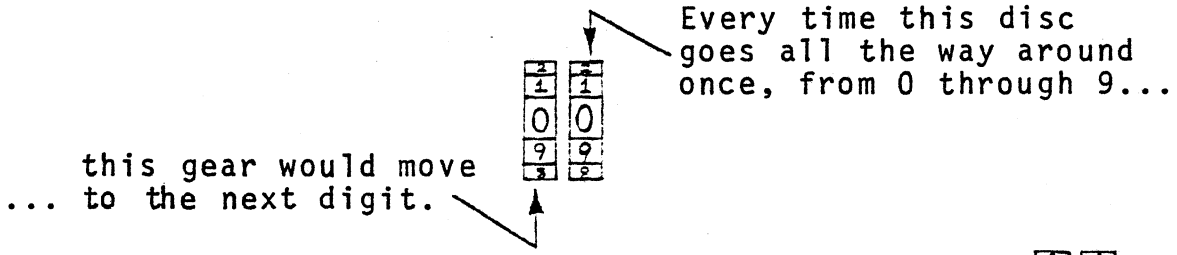
i) Decimal and Binary

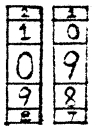
Binary math is the symbol system that most closely approximates the actual operations of an electronic digital computer. A mechanical computer might use devices having ten different "states," like a disc with the digits 0 through 9 around its edge:



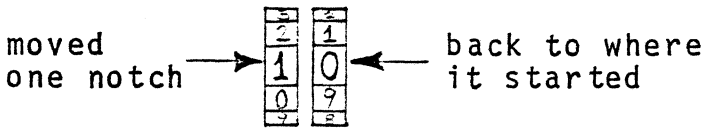
(or discs with ten notches, gears with ten teeth, etc.). The ten "states" of such a disc would be the ten different digits that the disc presents to the view of the human user. In fact, there are such devices, and they might be thought of as being purely "decimal" in operation -- decimal meaning ten.

We could put several such discs together and have them count things. Each disc would be geared to the one next to it, so that when the disc on the right went all the way around once, through all ten digits from 0 through 9, the gear on the left would move one "notch" to its next state.



So when the counter had counted nine things, it would read 

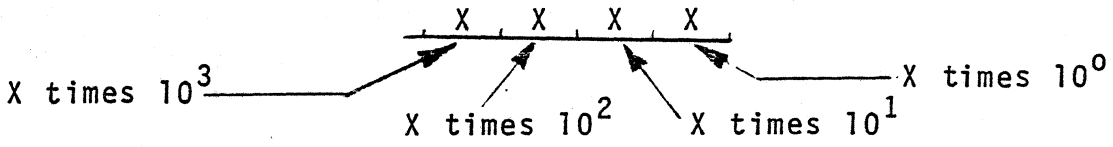
and when it counted one more, it would read:



An alternative would be to have just one large disc with many numbers on it -- the numbers 0 to 100, say. But our "decimal discs," each geared to move one number when the gear to its right went all the way around once, would be smaller and handier, and would be able to count very large numbers. In fact, we increase its capacity to count by a factor of ten every time we add another disc. A two-disc counter could count from 0 to 99, while a three-disc counter could count from 0 to 999, and so on.

Actually, this hypothetical counter using decimal discs is a

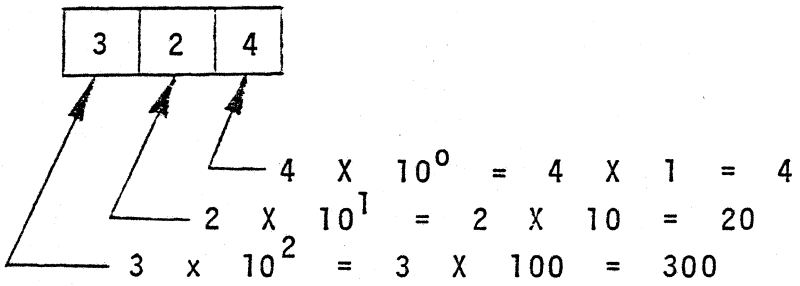
mechanical analog of the decimal number system itself. Each disc corresponds to a "place" in a decimal number, and the fact that we would want each disc to move by one number when the disc to its right had gone all the way through its ten digits shows that each "place" in a decimal number represents a power of ten, with each place being one power higher than the place to its right.



In the decimal system, this number:

324

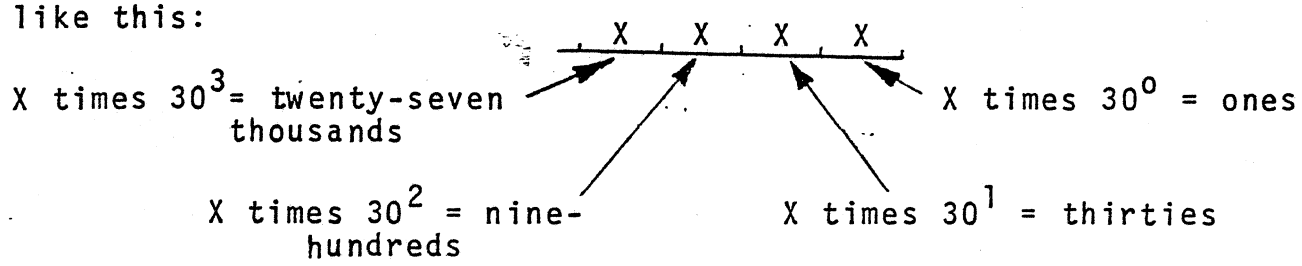
means "three hundred and twenty-four" because each digit in the number represents a power of ten, thus:



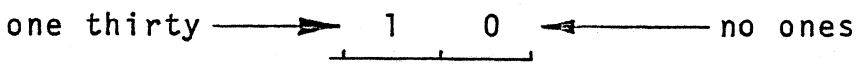
The right-hand place (called the "least significant") in the number is the 10^0 or "ten to the zeroeth" place; the digit occupying that space tells you how many 10^0s the number contains. Since $10^0 = 1$, this position is called the ones place or ones column. (The zeroeth power of any number is one.) The 4 occupying this column means "four ones." Moving to the left, the next place shows 10^1 or ten. The 2 occupying this place means "two tens." The next place (in this case, the "most significant") indicates 10^2, or one hundred. Three hundreds, two tens, and four ones -- 324. To express numbers involving

thousands, tens of thousands, etc., we just keep adding places to the left. The left-most place is always the most significant -- it indicates the highest power of the base. Decimal has ten different digits -- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 -- and so can indicate from 0 to 9 ones in the ones column, from 0 to 9 tens in the tens column, and so forth. Humans, having ten fingers, find decimal or ten-based counting very natural, and many people just cannot believe that other bases are better for some purposes.

Nevertheless, the fact is that <u>any</u> value can be used as a base in a number system. Consider <u>thirty</u>: We say "thirty" and write 30 to represent this quantity. But just as we could use some other word than "thirty" to represent the quantity, so we could apply some other number system. For instance, the base of the number system could itself be thirty. That system would work like this:



Thirty includes "no ones" and "one thirty," with no nine-hundreds, etc. So in a thirty-based system, the value thirty would be expressed:



In a number system based on three, thirty would be expressed:

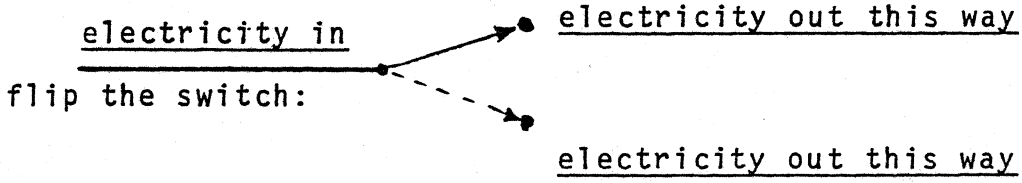
1 0 1 0

(Why?)

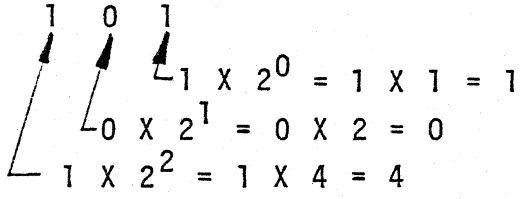
Just what number is selected to be the base of a number system is a matter of convenience. Creatures having ten fingers find

ten a convenient base. But when you start making devices to help you in computation, it becomes convenient to use other bases as well.

Computers use solid-state electronic devices to perform computations. The simplest, smallest, cheapest solid-state device can take on just two electronic "states." (Recall that the decimal disc had ten states -- the ten different positions it would be in to show each of its ten digits to a viewer.) Such a device is often thought of as a switch. The switch might offer two possible pathways:



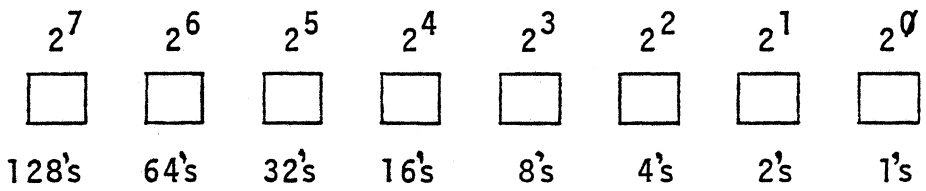
The two stable states of the switch might simply be OPEN and CLOSED -- OFF and ON. The two states can be called YES and NO, or TRUE and FALSE -- or they can be called 1 and 0. This immediately suggests a number system based on just two digits. And in fact, since a computer actually performs its operations by means of many tiny solid-state devices that have two possible stable states, computers are said to "use binary" in their operation. In binary each position or column in a number represents a power of two, rather than a power of ten. This binary number



says " 1 one, 0 twos, and 1 four, " or five. In decimal -- 5. In binary -- 101. (We will continue to use written words like "one, two, three" to discuss these other systems.) We will also slash all Øs to distinguish them from 0's, as does the POLY 88.)

Obviously, binary numbers are usually longer than decimal numbers. So why use them? Because solid-state electronic devices find them convenient. To be more exact, the binary system is the simplest number system that can convey any and all data. The only simpler system would be to the base one, and that would not be a "system" at all, but just a tally -- ten marks to indicate the number ten. The simplicity of binary allows computer design to be as simple as possible, since the simplest physical system with more than one stable state has two.

Any quantity can be represented in binary. Just keep adding powers of two to the left. Binary digits, or "bits," are frequently grouped in groups of eight:



Eight bits is a "byte." The largest number expressible in a byte, obviously, is 11111111, which is (starting from the right) one one, one two, one four, one eight, and so forth. In decimal it would be 1 + 2 + 4 + 8 + 16 + 32 + 64 + 128, or 255. Larger numbers, of course, are built up from several bytes.

Many computers, the POLY 88 among them, always treat values in eight-bit bytes. For instance, the POLY 88 stores data in its memory in the form of eight-bit bytes. Recall that a binary digit or bit, which is always either 0 or 1, corresponds to a tiny solid-state device which is in one or the other of its two stable states. When you store a data quantity in the POLY 88, you are actually manipulating the states of many such devices. Let us call these two states the "zero state" and the "not zero state." When you store the quantity five in the POLY 88's memory,

you affect the state of eight (microscopically small) devices. Five in binary is 101B, or, as an eight-bit byte, 00000101B. The eight affected devices will be in this overall state:

Zero State	Zero State	Zero State	Zero State	Zero State	Not Zero State	Zero State	Not Zero State

Another way to show the state of the eight affected devices is:

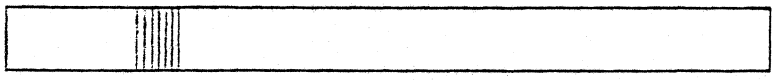
Ø	Ø	Ø	Ø	Ø	1	Ø	1

This is a very convenient way, because here we have our binary number 00000101B over-laid onto the representation of the eight affected devices. We will be using this representation often, because to get the greatest use and the greatest aesthetic satisfaction out of your computer, it is important to "think binary" (at least at first)and visualize the actual events going on down at the level of the bi-stable devices. The "memory" of the computer consists of many, many groups of eight such devices, and can be thought of as many such rows (bytes) as this:

Ø	Ø	Ø	Ø	Ø	1	Ø	1

turned on edge and grouped together like this:

Memory



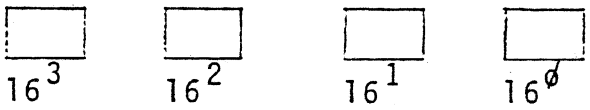
each location in memory
contains one byte -- eight bits

Binary expresses the actual state of the bi-stable devices or "flip-flops" that make up the computer's memory. As we will be discussing later, binary also expresses the "vectors" or the pathways that lead to the bytes that make up the memory.

ii) Octal and Hexadecimal

Though electronic devices find binary convenient, it is cumbersome for humans. So computer operators use other number systems that are more like decimal in their compactness, namely "octal," based on eight, and "hexadecimal," based on sixteen. Since eight and sixteen are themselves powers of two, numbers in these systems are fairly easy to compare with or convert to binary numbers. Decimal could be used, but it would not convey any sense of the actual state of operations down at the level of the solid-state devices, where the abstraction of numbers has its reality in the form of the electrical state of each device.

Octal uses the digits Ø through 7, with each position indicating up to seven times a power of eight. Hexadecimal is more important for our purposes, since operating the POLY 88 involves its use. Hexadecimal uses all ten of the familiar digits, plus the first six letter of the alphabet: Ø123456789ABCDEF.
Each position in a hexadecimal number indicates a power of sixteen, thus:



Clearly, very large numbers can be compactly expressed in hexadecimal. This number, for instance:

FØØ

Says "no ones, no sixteens, and fifteen two-hundred-fifty-sixes." In decimal -- 3,84Ø. (The hex number above would ordinarily be written FØØH and said "F zero zero hex." We will always put an H after a hex number and a B after a binary number; a number with no letter after it is always decimal.)

In binary, by the way, the above hex number is 111ØØØØØØØØB. There is no sense in which the decimal number 3,84Ø is the "real" expression of this value. Any binary number up to four bits can be expressed as one hexadecimal number, thus:

Binary	Hexadecimal	Decimal
ØØØØB	ØH	Ø
ØØØ1B	1H	1
ØØ1ØB	2H	2
ØØ11B	3H	3
Ø1ØØB	4H	4
Ø1Ø1B	5H	5
Ø11ØB	6H	6
Ø111B	7H	7
1ØØØB	8H	8
1ØØ1B	9H	9
1Ø1ØB	AH	1Ø
1Ø11B	BH	11
11ØØB	CH	12
11Ø1B	DH	13
111ØB	EH	14
1111B	FH	15

Recall that in the POLY 88, each memory location consits of eight bi-stable devices, so the contents of any memory location can be expressed in eight binary digits or bits. This eight-bit value can in turn be expressed in two hexadecimal characters. For instance, if you think of the number 1111ØØØØB as two groups: 1111 ØØØØ, you see that it equals FØ in hex (FØH).

b. ASCII. Binary numbers can be used (like any numbers) in a code to express things other than quantities. One code for us to consider is the American Standard Code for Information Interchange, or ASCII. ASCII provides a means for putting information into a computer and getting it out again in a form that makes sense to the human. In ASCII, the characters the human writes or reads -- upper and lower case letters and the ten digits, plus punctuation marks -- are all assigned numberical values. Every character on a typewriter keyboard, for instance, is assigned a binary equivalent. When the human strikes a key on the computer's input keyboard, the keyboard in turn sends to the computer one byte -- an 8-bit binary number -- corresponding to that key. This is necessary because the computer itself "understands" nothing but binary; it conducts all its operations in terms of the bistable state of electronic connectors.

ASCII also enables the computer to put out characters that make sense to the human. If the computer and the human are communicating strictly by means of a typewriter, the process described above simply reverses. The human puts in his/her information by striking the appropriate keys. The keyboard electronics interpret this according to ASCII code into binary bytes, which are then sent on to the computer. When the computer completes its operation, it sends a series of bytes back to the teletype, which interprets them according to the ASCII code and causes the appropriate keys to strike. Some bytes correspond to functions other than key strikes -- carriage shift, carriage return, etc.

The POLY 88 uses a keyboard input and video output. (There is also a tape input/output. The tape can be recorded according to the

ASCII code -- actually in a dual-tone code corresponding to the binary representations of the ASCII characters.) The human types into the keyboard his/her input. The keyboard sends this input on to the computer in the form of the bytes assigned to each key by ASCII. When the computer finishes its operations, it sends the information in ASCII to its own video electronics, where the information is converted into the form that will cause the appropriate characters to appear on the screen. You will find a chart showing the ASCII-to-binary code in the appendix.

2. Computer Languages

a. Machine language

As said earlier, the data a computer deals with exist in the form of states of sets of bi-stable devices. Actually, not only data items but also every operation the computer can perform corresponds to a binary number. At the most fundamental level of the computer hardware, events take place in the form of changes in the state of individual devices having two stable states. Therefore, binary "statements" can exactly "express" what is actually happening inside the computer.

The heart of a computer is its central processor, usually consisting of one or more integrated circuits or "chips." Built into the physical structure of these chips are microscopic pathways and electronic devices that enable the computer to perform its basic operations. The POLY 88 has one such chip, the Intel 8080A microprocessor, designed to allow the computer to perform 72 fundamental types of functions. These functions are called "machine instructions," and together are called the instruction set. Each instruction corresponds to a binary number, and the set of all such numbers is called the machine language.

b. Assembly language

Binary exists for the convenience of the computer -- it allows computer design to be as simple as possible. Hexadecimal and ASCII are ways that statements which make sense to the human can

be related to "statements" that "make sense" to the computer. Assembly language is a convenience to the human -- it expresses computer operations in a form that makes sense to him.

Assembly language assigns a word to each instruction. When the human writes a program, he/she uses assembly language to express what the computer is supposed to do. For instance, the human may write down something like this:

```
INR  A
```

This instruction, which means "INcrement Register A," causes the value stored in a certain location to increase by 1. This instruction can now be converted into a form the computer can use. Each word in assembly language can also be expressed as a two-digit hex number (called the "opcode") which represents the binary instruction actually executed by the machine. (For a list of all assembly language words--the instruction set--and their hex equivalents, see the appendix.) The human can now rewrite his program in hex. The instruction above, for instance, would be:

HEX	ASSEMBLY LANGUAGE
3C	INR A

This conversion process is called assembling. Now we can type our hex directly into the keyboard. From there on, the POLY 88 converts the input into the binary form it requires.

Assembling by hand, by the way, can be avoided. You can program the computer so that it will convert assembly language to the form it needs. Such a program is called an assembler. When using an assembler, you begin by putting the assembler program into the computer. Then you type assembly language directly into the keyboard, and the computer interprets that input appropriately.

c. High level languages

There is an even more "human" way to write a program. Operations which seem to the human to be single steps (like "multiply") may actually require the computer to obey many instructions. The human may find it convenient to be able to use symbols that do not correspond directly to machine instructions. In assembly language there is a word or symbol corresponding to every instruction that the computer will obey in performing its operations. In a high level language, on the other hand, one word or symbol may imply many instructions.

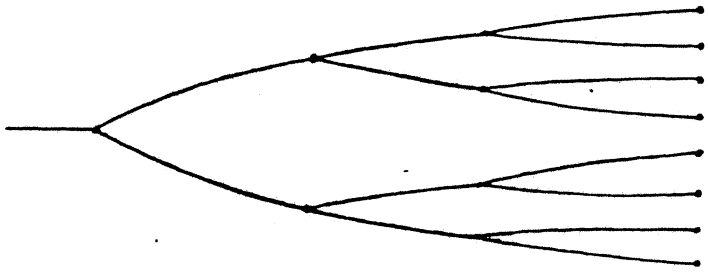
When you write your commands for the computer in assembly language (and then convert them to hex, or enter them directly into the computer by using an assembler), you are thinking in terms of the actual instructions built into the computer's central processor, and so you will probably be making the best use of the abilities of that particular computer. Nevertheless, it can sometimes be a great convenience to be able to write a program in a high-level language, using terms like "multiply" that imply many different instructions and would have to be expressed in several assembly language terms. The high-level language most appropriate for small computers like the POLY 88 is called BASIC. To communicate with your computer in BASIC, you would first program it to convert your statements in BASIC into the appropriate sequence of machine instructions.

3. Computer Theory

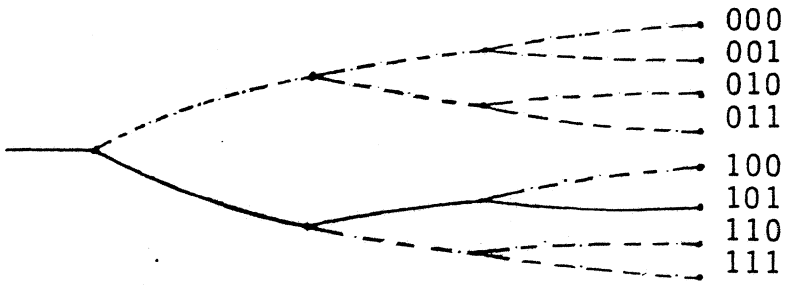
a. Address and Memory

Any computer works by performing a series of manipulations (or "program") on data stored in the computer's memory. Computer memory consists of the state of the thousands or millions of small, solid-state "flip-flops" or bi-stable devices within it. The POLY 88 has from 10,000 to half a million such individual devices in its memory, depending on options.

To perform its functions, the computer must be able to locate any one of the data items stored in its memory whenever it is needed. So the computer must keep track of where it puts each item. To do so, it assigns an "address" to each item in memory. In the POLY 88, the memory "bits" (each one corresponding to the state of a single flip-flop) are divided up into bytes of eight bits each, each byte having its own address. To get to a given address, the computer searches along a wiring pathway that has sixteen decision points -- sixteen places where the path can fork to the left or right. A schematic of this pathway would look, in part, like this:



We can assign the binary digits Ø and 1 to left and right turns at each decision point. Consider the pathway represented by the unbroken line:

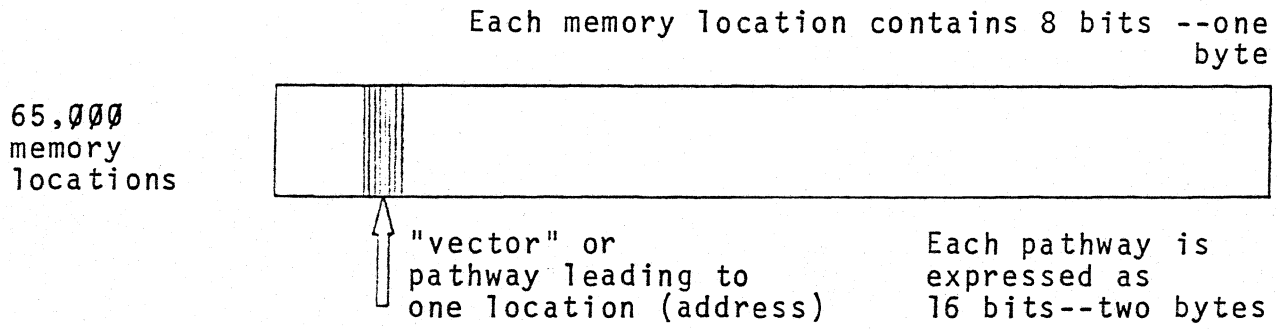


To follow this pathway, you turn first right, then left, then right. Defining left as Ø and right as 1, we can give this pathway the unique name 1Ø1, which means "right-left-right." Three right turns would be 111; the pathway involving three left turns would be ØØØ. All the other pathways would have the designations shown. These binary numbers can be thought of as the "addresses" to which these pathways lead. Since we have

either a left or a right turn at each of these decision points, we have a total of 2^3 or eight unique pathways in all. You might have noticed something else interesting. The addresses also turn out to number the pathways in sequence in binary notation. The top one, ØØØ, equals zero. The next one down equals one, the next one is two, the next one is three, and so forth. The last one, 111, is seven. So all eight pathways are neatly numbered, from zero through seven. That is what we mean when we say that each memory address is a unique pathway that can be expressed by a binary number. Each digit, Ø or 1, in the binary number corresponds to the state of a bi-stable device that is serving as a "switch" that turns the pathway to the left or right.

As you can see from the figure above, the total number of different pathways doubles at every decision point. Since there are 16 such points in all in the POLY 88, it can store items in 2^{16} or 65,536 different possible memory locations. Each "location" is actually a unique pathway, determined by the states of sixteen bi-stable devices, leading to a unique set of eight binary devices. The states of these eight devices constitute an item, or part of an item, stored in memory (one byte of memory). So -- 2^{16} or 65,ØØØ possible pathways leading to 65,ØØØ locations, each of which can contain any one of 2^8 or 256 different binary values.

All addressed locations taken together make up the "memory space" of a computer. Any one address is sometimes called a "vector" in this space because it "points" to a single byte in memory. The memory space and address pointer can be depicted like this:



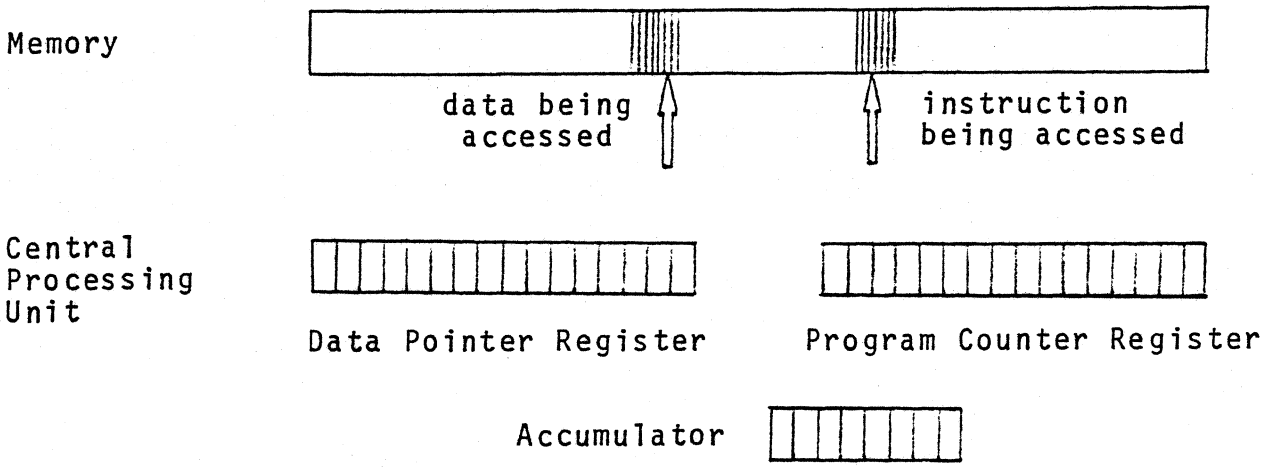
with the bar representing the bytes of 8-bit groups within the memory, and the large arrow indicating the 16-bit address that specifies the pathway leading to each memory location.

The items stored in memory can be used in any one of several ways. An item can represent a program instruction to the machine. Bytes in memory can represent ASCII characters, and thus can enable the machine to communicate with its human operator. An item can be a numerical value, to be manipulated in a program--like your bank balance. In the same way, a pathway leading to these items can be defined in two ways: as a program pointer (called "program counter") when it leads to memory items defined to be instructions; or as a data pointer when it leads to quantities to be manipulated by the program. Whether a certain stored item is to be considered an instruction or a quantity to be manipulated by a program is up to the operator. The computer does not care either way. In fact, if the operator accidentally tells the computer to treat a data quantity as though it were an instruction, the computer will gladly do so. To it, anything the program counter points to is an instruction--whether the operator meant it to be an instruction or not. And anything the data pointer points to is an item of data to be manipulated. Trouble sometimes occurs when the operator accidentally causes the program counter to point to a memory location that does not contain program. The computer executes the item as an instruction, and from that point begins to produce nonsense or "garbage."

b. Central Processor Architecture

A computer consists of memory space, containing stored program instructions and data to be manipulated in programs, and a central processing unit, which manipulates data in response to program instructions. The CPU opens pathways to memory items, takes items from memory and temporarily stores them, transforms data by means of mathematical and logical operations, and sends results out to memory or to an output device.

The organization of a processor is called its "architecture; an extremely simple computer could work like this:



Here we have a bar representing a large amount of memory, with a data pointer pointing to a quantity stored at one address, and a program counter pointing to a program instruction which tells the CPU what to do. (Don't lose sight of the fact that these "arrows" that are "pointing" at bytes of memory are actually pathways leading to groups of eight bi-stable devices.) Also, there are some "registers." The data pointer register contains the two-byte number that corresponds to the memory address the data pointer is pointing to. The program counter register states the memory address of the instruction that the CPU is currently executing. This hypothetical computer performs the operation pointed to by the program counter, using the data pointed to by the data pointer. The result of the operation gets stored in the accumulator. Note that, because the data pointer and program counter registers hold addresses, they contain two bytes. When the computer finishes the operation indicated by the program counter, the program counter automatically moves to the next instruction in the program. The program counter may "jump" on command to a new address at a considerable distance from the previous location, but for one moment let's visualize it as just moving one slot to the right at each step. The

computer then does whatever that instruction tells it to do. Each time an instruction is completed, the program counter moves to its next instruction.

The operation of this simple computer can be visualized as follows. The program counter points to an instruction that tells the CPU, "Move data pointer to <address> ." The CPU opens a line to the slot bearing that address. It also move the program counter to the next instruction. The next instruction says, "Take the quantity located in the slot indicated by the data pointer, and move it to the accumulator." The CPU does so. Next instruction: "Move the data pointer to <new address> ." CPU obeys. "Add the quantity in that slot to the quantity in the accumulator." CPU does this. "Move the data pointer to <new address> ." Obeys. "Take the quantity from the accumulator and put it in the memory slot that the data pointer is now pointing to."

Note that this series of operations involves three quantities in memory: two quantities that were already in memory, and a third quantity, the sum of the first two, which is now also stored in memory.

Now, about those other two CPU registers. Every time the computer obeyed an instruction, the program counter register changed to reflect the address of the next instruction. Since we are visualizing this simple computer as performing a series of instructions in sequence, let us say for the moment that after each instruction is performed, the value in the program counter register goes up by 1, to move the program counter one slot up. (Actual instructions can consist of several bytes and therefore occupy several consecutive addresses.) We could put an instruction into the program that says "Jump the program counter all the way to another part of the program." The value stored

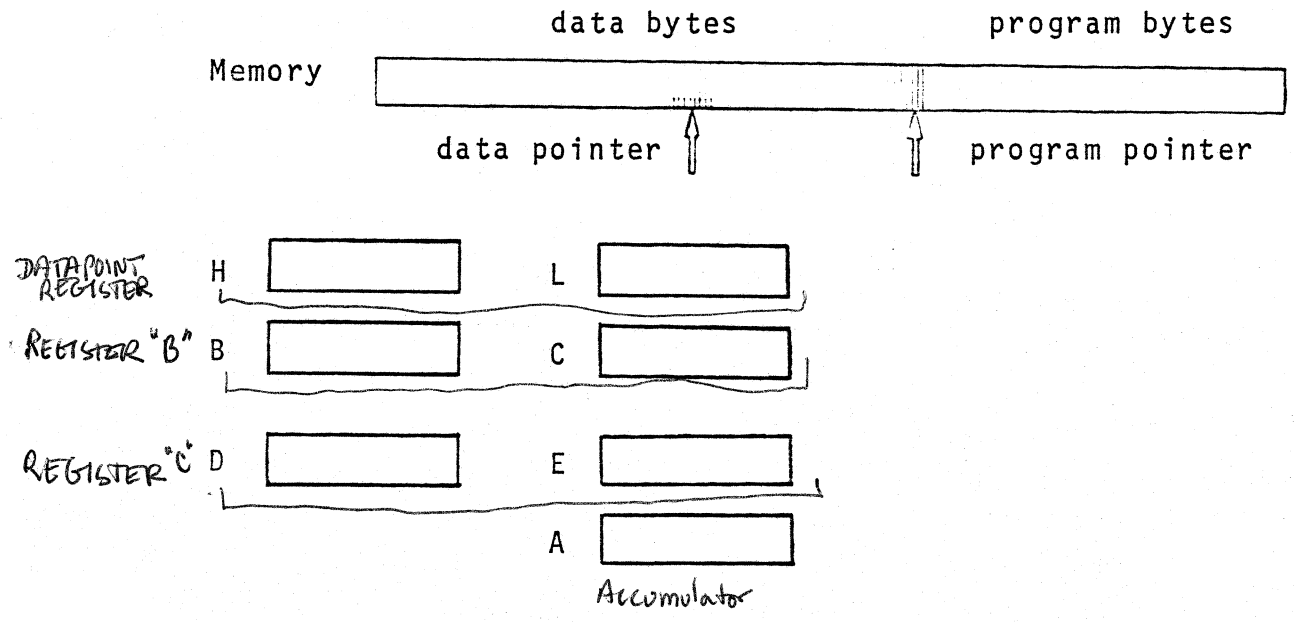
in the program counter register would change to reflect the address of the new instruction. The same thing is true of the data pointer register -- the value in the data pointer register goes up or down as the data pointer moves to correspond to the memory address of each accessed memory item.

Why the accumulator? Because a computer performs its operations by taking one very small step at a time. To take a quantity out of memory, then take another one from memory and add it to the first, then put the sum into a new location, takes the computer three steps. It needs the accumulator to store the results of intermediate steps.

This simple computer can do anything that any real computer can do. But because it has just the three basic registers, it does everything slowly. To increase the speed of the computer, we add registers.

The POLY 88 has several additional "working registers." The working registers are like the accumulator in that they temporarily store values being used in computations.

We will add the working registers to our conceptual depiction of what is called the <u>architecture</u> of the POLY 88:



So far, we have the memory space; a data pointer register; a program counter register; working registers (two pairs, each one holding one byte); and an accumulator. Let us consider the working registers.

The working registers, like the accumulator, temporarily store values the computer is using in the course of its operations. Thus the computer can move a value from memory to any register, including the accumulator, from any register to any other register, and from any register to memory. It can perform some operations on the contents of the accumulator. It can also perform operations involving the contents of the accumulator and other registers (add them, for instance).

You have probably noticed that, because we used the letter A to designate the accumulator, we designate the working registers B, C, D and E. Each register holds one byte; the registers can also be used in pairs, B with C and D with E, to hold 16 bit quantities. The register pairs are called "pair B" and "pair D."

You might also note that we are using the letters H and L to designate the two bytes of the data pointer register pair. This simply means that the left register contains digits of relatively high significance, the right register digits of lower significance. In any number, the significance of digits increases as you move to the left -- in decimal, tens are more significant than ones, hundreds are more significant than tens, and so forth. The letters H and L could apply to any of the register pairs; by convention, however, only this register pair is described in that way.

There remains just one basic feature of computer architecture to add: the stack pointer. In order to talk about the stack pointer, we will have to go a bit deeper into the subject of programming.

A program is a pre-determined series of instructions for the

computer to follow in solving a specific problem. Recall that the heart of the computer is its central processing unit, consisting of one or more chips into which are built the electronics providing for all the logical operations of the computer. This central processor lets the computer deal with all the various kinds of problems it is built to deal with. But the central processor does not tell the computer when to perform any given operation, or on what, and so it does not enable the computer to deal with any specific problem. That requires a program.

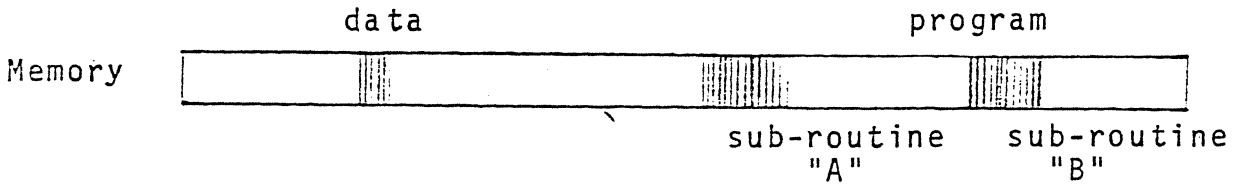
The fundamental instructions built into the central processor are called collectively the "instruction set." A program also consists of instructions, which the user stores in memory. The computer uses the electronics of the central processor as required to obey the program instructions it encounters in memory.

A very simple program starts out with instruction #1 and moves along in a straight line through a series of instructions to the end. But almost no program is that simple. Most programs incorporate strings of instructions that the computer performs repeatedly, returning to the beginning of the string and performing it again until some condition is satisfied. These repeating strings are called "loops."

Another possibility is a separate part of a program that performs some specific task that may be called for at several different times in the execution of the program. The computer moves to that part of the program whenever it is instructed to, and performs the instructions it contains; then it returns to the main-stream program. These repeatable sub-portions of the program are called "sub-routines."

Obviously, the terms "loop" and "sub-routine" are not really mutually exclusive. A sub-routine could be a loop. The point is that both terms indicate a departure from the straight-ahead, left-foot-right-foot progress of the program.

We can depict a sub-routine in this way:



Sub-routines "A" and "B" consist of instructions for operations that are needed several times in the execution of the program. Every time sub-routine "A" is needed, the computer comes upon the instruction "Call sub-routine ⟨first address in A⟩ ." And off it goes. When it needs "B," the computer comes upon the instruction "Call sub-routine ⟨first address in "B"⟩ ."

This is where the stack pointer comes in. Say the program counter departs from the main-stream program and goes to sub-routine "A." The computer performs the sub-routine as required. Now it has to get back to the right place in the main program. The stack pointer records the address to which the program counter must return in order to resume the main-stream program. In the example we are discussing, this will be the address in the main-stream from which the program counter originally departed -- plus three.

Why plus three? Because if the program counter returned from sub-routine "A" to exactly the same point in the program from which it departed, it would once again come upon the instruction "Call first address in "A" ." So back it would go to the beginning of sub-routine "A". Finished, it would return to the same instruction -- and go back to "A" again -- and again -- till some merciful human pulled the plug. So when it return from the subroutine, it must begin at the instruction following the "CALL" (one-byte) and the two-byte address of the sub-routine.

Now, our example is still very simple. So far it just involves jumping to either "A" or "B" from the main-stream program, then

back. But often a program will involve calling a sub-routine, then calling from the midst of the sub-routine to another, and so forth. So the program counter may move from the main-stream program to "A," then before finishing "A" it may move to "B," then "C," etc. To return, it may have to go from "C" to "B," then to "A," then back to the main-stream program. The actual path a computer follows to go from question to answer can be very, very complicated.

That is why the stack pointer has a stack. The stack pointer keeps inserting into a portion of memory the addresses that the program counter has to return to as it finishes with sub-routines. The first address that goes into this "stack" is the address from which the program counter departed when it jumped to sub-routine "A," plus three. If it goes to "B" before it finishes with "A," then the next entry in the stack is the address in "A" to which the program counter must eventually return. The stack pointer keeps putting these addresses in, in the required order, till it is necessary to start returning.

To summarize, the computer stores into its memory both data items and program instructions, in the form of bytes. It takes data items from memory and puts them into the working registers, where they can be manipulated. Addresses, or pathways leading to data bytes and program bytes, are represented in the program counter, data pointer, and stack pointer registers. These addresses can themselves be stored into memory and recalled as required.

c. Instruction Set

At the heart of the POLY 88 is a small "chip," about the size of the nail of your little finger, called a central processing unit or CPU. This integrated circuit, the Intel 8080 A, incorporates many microscopically small solid-state electronic devices that enable the computer to perform its various operations. Basically, in fact, all processing is the job of the CPU, while the rest of the computer components provide input, output, storage, and access.

We will now take a look at all 72 kinds of operations or "instructions" the Intel 8080 CPU can perform. First we will consider some general concepts.

Some of the operations the CPU performs are familiar -- addition and subtraction, for instance. Others equally important, and in fact more fundamental, are "logical operations," in particular those called complementation, AND, OR, and XOR (exclusive OR).

COMPLEMENTATION

Addition and subtraction treat binary quantities as quantities -- as numbers built up of one or more digits, to be treated as wholes. Logical operations, on the other hand, treat the bits of binary values one at a time. One of them, complementation, simply changes every Ø to a 1 and every 1 to a Ø. These two numbers are complementary:

```
1Ø1Ø1ØØØ
Ø1Ø1Ø111
```

One of the CPU's operations is "complement"--that is, the CPU can be told to complement any number, and will respond by changing every Ø to a 1 and every 1 to a Ø.

TWOS COMPLEMENT

For simplicity of design, the central processor uses addition to subtract. It does this by converting a number to be subtracted into its "twos complement," which in effect reverses its sign, then adding it to the number to be subtracted from.

Let's say that the subtraction problem is:

```
  11Ø11ØØ1
- Ø1ØØ1ØØ1
```

The number to be subtracted is Ø1ØØ1ØØ1. To do this, we will instead add the twos complement of this number, which is in effect its negative counterpart (in a number of fixed length--here, eight bits). We begin by constructing its twos complement.

First we complement this number -- invert every bit. It becomes 10110110. That is the "ones complement," or just the complement of 01001001. Then we add 1 to give the twos complement.

```
  10110110
+ 00000001
----------
  10110111
```

We can test to see if this is really, in effect, the negative equivalent of the original number by adding it to the original number and seeing if the result is zero. In so doing, we will also see the point of considering only numbers of fixed length.

```
  01001001
+ 10110111
----------
1 00000000
```

ninth bit not considered part of sum

one byte

The fact that we are considering numbers of fixed length means that the carry out of the most significant place is not considered part of the sum, so zero can result from the addition of two nonzero bytes.

Adding 10110111 to 11011001 yields:

```
  11011001
  10110111
----------
1 10010000
```

Carry not part of sum

result

Among binary number of a fixed length of eight bits, there are 256 different possible combinations. These 256 combinations can be considered to be the positive numbers from 0 through 255. Equally well, they can be considred the 256 values from -128 to + 127. In this latter case, the binary expressions for the values from 0 to +127 would all be exactly the same as when only

positive numbers are being represented. Then converting all of these values to their twos complements yields the remaining binary values which can be considered their negative counterparts. Note that, in this case, all the positive numbers would begin with a Ø in the most significant place, and all the negative numbers would begin with a 1.

ØØØØØØØ1B = 1
Ø1111111B = 127 (decimal)

11111111B = -1 (twos complement of 1)
1ØØØ0001B = -127 (twos complement of 127D)

Note that Ø (ØØØØØØØØB) and -128 (1ØØØØØØØB) are their own twos complements. All other values have their own unique but dissimilar twos complement.

The existence of these two sets of values, positive values starting in every case with a Ø and negative values starting in every case with a 1, means that the most significant bit can be considered not only part of the value but also the sign of the value. This is called "signed twos complement notation." The following discussion of computer operations or "instructions" must be understood in light of twos complement representation.

LOGICAL OPERATIONS

One logical operation, complementation, treats the bits of a single binary value. The other logical operations of the CPU compare the bits of one binary number with the bits of another. Let's take two binary numbers having just one bit each:

Compare these bits.

The comparison checks to see if the two bits conform to some "rule," and leaves a record to indicate whether or not they do conform. A rule might be: "The two bits are the same." If they are the same, we can leave a 1 as a record; if they are not the same, we can leave a Ø. Note that, in a comparison of two bits, the first bit can be either a Ø or a one, and the second bit can also be a Ø or a 1, so there are 2^2 or four possible cases:

Ø◄——►Ø : 1.
1◄——►1 : 1.
1◄——►Ø : Ø.
Ø◄——►1 : Ø.

(Since the rule says nothing about the order of the bits, we can consider the last two cases identical.)

A different rule will produce different results for the same cases. Supposing the rule is " One or the other of the two digits is a 1." Now the four cases produce:

Ø◄——►Ø : Ø.
1◄——►1 : 1.
1◄——►Ø : 1.
Ø◄——►1 : 1.

Another way of saying that two bits compare in conformance with a rule is that the comparison is <u>true</u>. Using <u>false</u> and <u>true</u> instead of Ø and 1 is very interesting, because it shows how fundamental these comparisons are to logic, and therefore why these comparisons are called logical functions.

In the second example above, the rule was that one or the other

of the two bits (or both) had to be 1. If at least one bit was 1, the result of the comparison was also 1. If neither was 1, the result of the comparison was Ø. Defining Ø as false and 1 as true, we can restate that rule thus: If one bit or the other is true, then the result is also true. For brevity:

> If A is true OR B is true, then
> C is true.

This rule provides a model for a certain kind of logical syllogism -- the kind in which a certain conclusion always follows if either one of two conditions is met. For instance:

If the batteries are dead, OR the bulb is burned out, the flashlight will not work.

Here, either one of the conditions if true is sufficient to make the conclusion true. There is, of course, another kind of relationship, in which both one condition AND the second must be true for the result to be true.

If you have enough money, AND if the store is open, you can buy what you want.

This AND relationship provides the model for the most famous syllogism of all: "All men are mortal; Socrates is a man; therefore Socrates is mortal." Stated like the previous example: "If all men are mortal, AND if Socrates is a man, then Socrates is mortal." Note that syllogisms need not assert the truth of any particular fact, but only offer a model to predict the "truth relationships" of possibilities. Another way of making the same syllogism would be:
All men mortal? True. AND Socrates a man? True: Socrates mortal? True.

To summarize the AND relationship:

false AND false: false
false AND true : false
true AND false: false
true AND true : true

Defining Ø as false and 1 as true, we can summarize the AND relationship in this "truth table":

	Ø	1
Ø	Ø	Ø
1	Ø	1

AND

We have already seen the rule that if one OR the other (or both) of the two conditions being compared is true, then the result is true. Here is the OR truth table:

	Ø	1
Ø	Ø	1
1	1	1

OR

One other rule concerns us here: The XOR or "exclusive OR" rule, in which one or the other (but <u>not</u> both) conditions must be true for the result to be true.

For instance, suppose that Mr. Smith employs an equal number of female and male people:

If Smith hires one female, XOR if Smith hires one male, he will have an unequal number of female and male employees.

Here is the XOR truth table:

	Ø	1
Ø	Ø	1
1	1	Ø

As we said before, the POLY 88 CPU compares the bits of two numbers by means of one of these rules in performing its logical operations. Here's how two numbers are compared in these ways:

```
          10110010
          11011111        AND
result    10010010
```

The top number is compared bit by bit with the number below it, to see if the upper bit AND the lower bit are 1.

```
10110010
11011111        OR
--------
11111111
```

The top number is compared bit by bit with the number below it, to see if the upper bit OR the lower bit is 1.

```
10110010
11011111        XOR
--------
01101101
```

The top number is compared bit by bit with the number below it, to see if the upper bit XOR the lower bit is 1.

BRANCHING

Computers are valuable primarily because they can do repetitive tasks very rapidly. To be able to repeat the same task a required number of times, the computer must be able to decide whether it is to repeat a task or move on. The computer repeats a task

until some condition is satisfied, then moves on to something else. If it could not make such a decision, the computer would have to be told whether to repeat or move on -- the operator would have to make that decision, and the computer would be far less useful than it is.

This decision can be -- and is -- divided into two parts: a test and a branch. The test determines which of two conditions exists. The test may be of whether two values are equal, or of whether one value is at least as large as another; it may be of whether a particular single bit is Ø or 1, etc. These tests always involve at least one of the values currently stored in a CPU register.

The branch is the point at which the computer moves in one of two directions. Which way the computer goes depends on the result of a previous test. We need a way to record the results of the test for use in the branch. This the computer does by setting the value of a particular bit to 1 or resetting it to Ø to indicate which of two conditions was found to exist. These bits are called "flags."

These decisions, called conditional branches, always involve two instructions:

TEST. Which of two conditions exists? Set a particular flag to 1 or reset it to Ø depending on which condition exists.

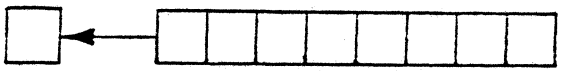
BRANCH. Is a particular flag Ø or 1? Go on to one or another of two different instructions depending on the status of the flag.

Following a branch instruction, the computer always moves on either to the next instruction in sequence, or to an address stated in the branch instruction itself, where it will encounter another instruction.

In the POLY 88 there are five flags.

CARRY FLAG

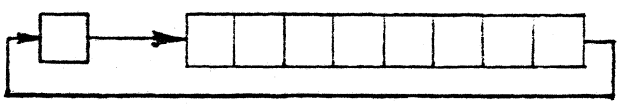
When a number is added to the value in the accumulator, the result may include a carry out of the left-hand bit, the bit of highest significance. This carry "sets" the carry flag to 1.



accumulator

When an addition does not result in a carry out of the most significant accumulator bit, the carry flag is Ø.

The carry flag can be set to Ø or 1 by other operations. For instance, the instructions RAR (rotate accumulator right) and RAL (rotate accumulator left) affect the carry flag. In RAR, the least significant bit in the accumulator moves into carry, the bit that was in carry goes into the most significant place in the accumulator, and all other accumulator bits move one place to the right.

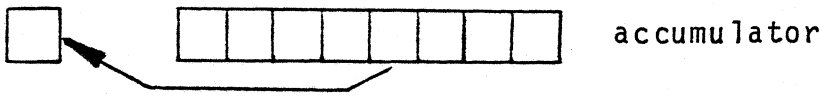
RAR 

RAL is the opposite of RAR.

The carry flag can be affected by logical operations as well as addition, subtraction, and rotation.

AUXILIARY CARRY FLAG

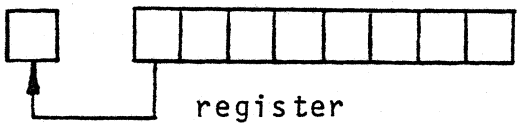
A carry out of the "third bit" (fourth place -- 2^3) sets the auxiliary carry flag:



The auxiliary carry flag cannot be tested directly, and exists only to enable the DAA instruction for decimal conversion.

SIGN FLAG

The sign flag is set by certain instructions to duplicate the most significant bit of the value in the affected register.



Recall from our discussion of twos complement that the most significant bit in a register can be interpreted as the sign of the data quantity when the quantity is considered to be twos complement.

ZERO FLAG

The zero flag is set to 1 at the end of certain operations if the byte resulting from the operations is all zeroes; the zero bit is reset to Ø if the result is not zero.

A result that consists of eight zeroes plus a carry out of the seventh bit sets the zero flag to 1, and also sets the carry flag to 1.

PARITY FLAG

"Parity" refers to whether the number of 1s in a byte is even or odd. Byte parity is checked after certain operations. If the

number is even, parity is "even" and the parity flag is set to 1; if there is an odd number of 1s, parity is "odd" and the parity flag is reset to Ø.

INSTRUCTIONS

Following is a complete list with discussion of all the operations built into the central processor of the POLY 88. The discussion divides the operations into groups of related instructions. Each operation is identified by a "mnemonic" which corresponds to an instruction in machine language ("opcode"). For a chart showing all assembly mnemonics and the associated opcodes, see appendix.

CARRY FLAG INSTRUCTIONS. Two instructions affect the carry flag alone:

CMC (complement carry). Complement the carry flag -- Set it to Ø if it is 1 or to 1 if it is Ø.

STC (set carry). Set the carry bit to 1.

SINGLE REGISTER INSTRUCTIONS. These instructions affect the contents of one memory address or any one of the CPU registers -- one byte. If memory, the instruction affects the byte addressed by pair H.

INR (increment register or memory). Increment the affected register or memory byte by 1 -- add 1 to it.

DCR (decrement register or memory). Decrement register or memory byte by 1. This instruction is the opposite of INR -- it is identical to it except that it reduces the affected byte by 1. All flags may be affected.

CMA (complement accumulator). Complement the byte in the accumulator -- change every 1 to Ø and Ø to 1. No flags are affected.

DAA (decimal adjust accumulator). Adjust the byte in the accumulator to form two groups of four bits, each representing one decimal digit. This instruction is rather complicated, treating as it does the awkward relationship between binary and decimal. It is used -- infrequently -- when a decimal output is desired. DAA adjusts the first four bits and second four bits of the accumulator byte separately. First, the less significant four bits of the accumulator byte are compared to 1ØØ1 to see if they are greater than nine. If they are (or if the auxiliary carry flag is set to 1), then the accumulator is incremented by six -- which reduces the value of the four bits to nine or less. Next, if the four more significant bits of the accumulator byte now represent a number greater than nine (or if the carry flag is set to 1), then these four bits are incremented by six, so that they will represent a value of nine or less. Note that either of these two adjustments may have produced a carry. A carry out of the four less significant bits sets the auxiliary carry flag to 1; otherwise, it is reset. A carry out of the accumulator byte sets the carry flag to 1; otherwise, it retains its previous value. All other flags may be affected.

NO-OPERATION INSTRUCTION:

One instruction results in no operation.

NOP (no operation). Move on to the next instruction in sequence. No flags are affected.

DATA TRANSFER INSTRUCTIONS:

These instructions transfer data between registers or between memory and registers.

MOV (move). Move one byte of data from an indicated register or memory to another individual register or memory. The data also remains in its original location.

Format example: MOV B,A. "Move the byte in A (accumulator) into register B." Note that the format states the affected register first. Data cannot be moved from one memory address to another in a single operation. Data moved out of memory is always taken from the location addressed by H & L. No flags are affected.

STAX (store accumulator). Store the contents of the accumulator into the memory location addressed by register pair B or pair D. No flags are affected.

LDAX (load accumulator). Store the contents of the memory location addressed by the indicated register pair (pair B or pair D) into the accumulator. No flags are affected.

REGISTER OR MEMORY TO ACCUMULATOR INSTRUCTIONS.

These instructions operate on the accumulator using a byte taken from a register or from memory. Memory is taken from the memory location addressed by the data pointer (H & L). Results are left in the accumulator.

ADD (add register or memory to accumulator). Add the byte in one register or in memory to the value in the accumulator. ADD A doubles the accumulator. All flags may be affected.

ADC (add register or memory and carry flag bit to accumulator). Add the byte from a specified location, plus the value of the carry flag, to the value in the accumulator. All flags may be affected.

SUB (subtract register or memory from accumulator). Subtract the byte in a specified register or memory location from the value in the accumulator. SUB A subtracts the accumulator from itself, leaving it (and the carry flag) at zero. All flags may be affected.

SBB (subtract register or memory and carry flag bit -- "borrow"-- from accumulator). Subtract the byte taken from a specified location, plus the value of the carry flag, from the accumulator. All flags may be affected.

ANA (AND register or memory with accumulator). AND the specified byte with the accumulator. ANA is often used to zero part of the accumulator. Carry, zero, sign, and parity flags may be affected.

XRA (XOR register or memory with accumulator). XOR the specified byte with the value in the accumulator. XRA A zeroes the accumulator. Then a MOV from A to a register zeroes that register. All flags may be affected.

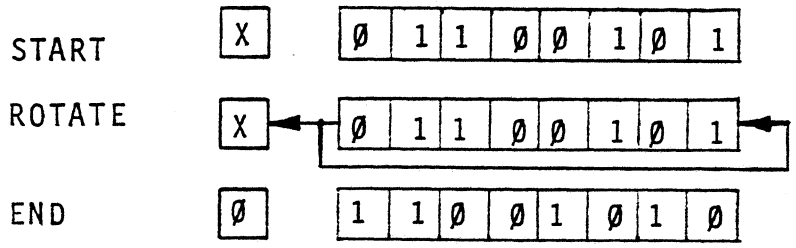
ORA (OR register or memory with accumulator). OR the specified byte with the value in the accumulator. This instruction is often used to set part of the accumulator to 1s. Flags affected: carry flag is zeroed; zero, sign, and parity flags may be affected.

CMP (compare register or memory with accumulator). Compare the specified byte to the contents of the accumulator. In effect, this determines if the specified byte is smaller than, equal to, or larger than the accumulator byte. Flags: the zero flag is 1 if the quantities are equal, and Ø if they are unequal. The carry flag is 1 if the register or memory byte is larger than the accumulator byte, and Ø otherwise (but when the two compared values differ in sign, the sense of the carry flag is reversed). All other flags may also be affected.

ROTATE ACCUMULATOR INSTRUCTIONS.

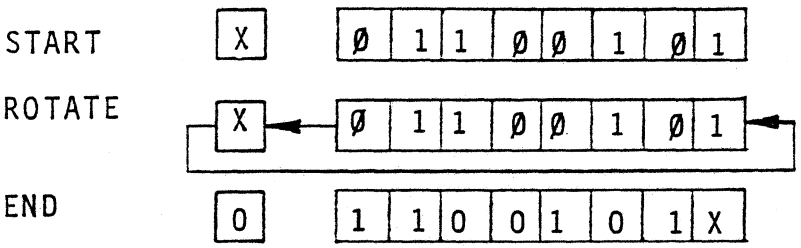
These instructions rotate the contents of the accumulator -- move a bit from one end, and shift the other bits one place. Rotation can be to the left or to the right, and involves the carry flag bit (but no other).

RLC (rotate accumulator left and into carry). Move the most significant bit in the accumulator (left-hand bit) into the carry flag and into the least significant place in the accumulator. All other bits shift one place to the left.



RRC (rotate accumulator right and into carry). Here, move the least significant bit from the accumulator into carry and into the most significant place; the opposite of the instruction above.

RAL (rotate accumulator left, through carry). Move the most significant accumulator bit into carry, and the carry flag bit into the least significant place; shift all other accumulator bits left.

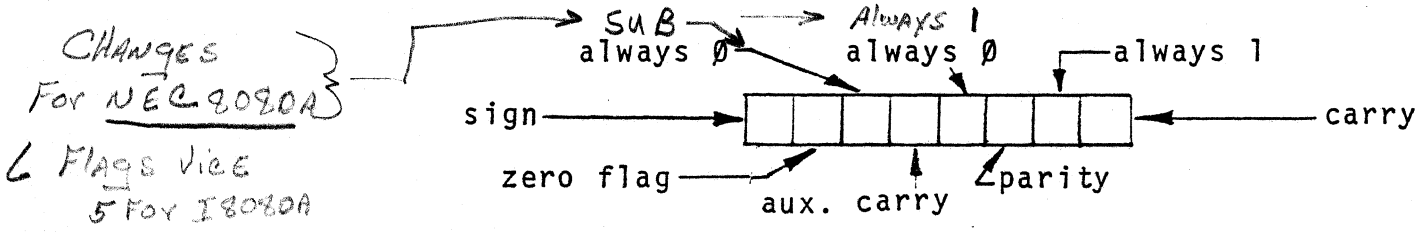


RAR (rotate accumulator right, through carry).
Move the least significant bit to carry, and move carry into the most significant place; the opposite of the instruction above.

REGISTER PAIR INSTRUCTIONS.

These instructions operate on the register pairs.

PUSH (push data onto stack). Store the value in the specified register pair into the two bytes of memory addressed by the stack pointer. Such data is said to be "pushed" onto "the stack." The more significant byte goes into address SP-1, the less significant into address SP-2. Indicating PSW (processor staus word) stores the current accumulator value at SP-1 and a byte incorporating all the flags in SP-2:



The stack pointer is left pointing to the address where the second byte has been stored Flags are not affected.

POP (pop data off stack). Store data from the stack into the indicated register pair. The byte of data at SP is stored into the less significant register; the byte at SP+1 goes into the more significant register. If register pair PSW is indicated, the byte at SP goes into the accumulator, and the byte at SP+1 provides the bits of the flags. This instruction is the opposite of the one above.

DAD (double add). Add the two-byte value in the indicated register pair (B, D, or H) to the two-byte value in pair H, and leave the result in pair H. Flag affected: carry.

INX (increment extended register pair). Increment the value in a register pair by 1 -- add 1 to it. No flags are affected.

DCX (decrement extended register pair). The opposite of the above.

XCHG (exchange registers). Move the value in pair H to pair D and vice versa. No flags are affected.

XTHL (exchange H & L with stack). Exchange the value in L with the value in the memory location addressed by the stack pointer and exchange the value in H with the value in that memory address plus one (SP + 1). No flags are affected.

SPHL (load SP from H & L). Load the value in register pair H into the stack pointer register. That value is now the stack address pointed to by the stack pointer. No flags are affected.

IMMEDIATE INSTRUCTIONS.

These instructions operate on one or two bytes of data, included in the instruction itself. The data immediately follows the opcode (hence "immediate").

LXI (load extended immediate). Load the indicated register pair with the two bytes immediately following. The first byte goes into the lower-order register, the second into the higher-order register. No flags are affected.

MVI (move immediate). Move the following byte into the specified register or into the memory location addressed by the data pointer. This instruction resembles LXI except that it enters only one byte of data (and therefore can be used to load a memory location).

No flags are affected.

ADI (add immediate to accumulator). Add the folowing byte to the value in the accumulator, and leave the result in the accumulator. All flags may be affected.

ACI (add immediate, plus the carry bit, to accumulator). Add the following byte, plus the value of the carry flag bit, to the value in the accumulator, and leave the result in the

accumulator. All flags may be affected.

SUI (subtract immediate from accumulator). Subtract the following byte from the value in the accumulator, and leave the result in the accumulator. All flags may be affected. This instruction is the subtraction equivalent of ADI above.

SBI (subtract immediate, and "borrow," from accumulator). Subtract the byte immediately following, and the value of the carry flag bit, from the value in the accumulator, and leave the result in the accumulator. This is the subtraction equivalent of ACI above. All flags may be affected.

ANI (AND immediate with accumulator.) AND the byte immediately following with the value in the accumulator, and leave the result in the accumulator. Carry, zero, sign, and parity flags may be affected.

XRI (XOR immediate with accumulator). XOR the byte immediately following with the value in the accumulator, and leave the result in the accumulator. The carry flag is set to Ø. Zero, sign, and parity flags may also be affected.

ORI (OR immediate with accumulator). OR the byte immediately following with the value in the accumulator, and leave the result in the accumulator. The carry flag is set to Ø. Zero, sign, and parity flags may also be affected.

CPI (compare immediate data with accumulators). Compare the following byte to the value in the accumulator. The zero flag is set to 1 if the two values are equal ~~and Ø if they are unequal~~ and Ø if they are unequal. The carry flag is set to 1 if the immediate data value is larger than the accumulator value, and set to Ø otherwise. (But if the two values differ in sign, the

sense of the carry flag is reversed.) All other flags may be affected.

DIRECT ADDRESSING INSTRUCTIONS.

These instructions involve the contents of memory addresses; the addresses are included as part of the instruction. The instruction states the address "backwards" -- first the less significant address byte, then the more significant. These instructions do not affect flags.

STA (store accumulator direct). Store the value in the accumulator into the memory location addressed in the instruction.

LDA (load accumulator direct). Load the contents of the memory location addressed in the instruction into the accumulator. No flags are affected. This instruction is the opposite of STA above.

SHLD (store H and L direct). Store the contents of register pair H into the memory location addressed in the instruction. No flags are affected.

LHLD (load accumulator direct). Load the contents of the memory location addressed by the instruction into the L register, and the contents of the next higher address into the H register. This is the opposite of SHLD above.

JUMP INSTRUCTIONS

These instructions cause the computer to "jump" to another part of a program rather than continue to perform instructions in sequence. None of these instructions affects flags.

PCHL (load program counter with H & L). Load the contents of register H into the more significant byte of the program counter, and the contents of register L into the less significant byte. The next instruction executed will be the one now addressed by the program counter. Note that this instruction does not itself contain an address. All other jump instructions do.

JMP (jump). Execute the instruction located at the address given in the instruction, and continue sequentially. This is called an "unconditional jump." All the following jump instructions are "conditional."

JC (jump if carry). Jump to the instruction addressed by this instruction if the carry flag is set to 1. If the carry flag is Ø, move on to the next instruction in sequence.

JNC (jump if no carry). Jump to the instruction addressed by this instruction if the carry flag is set to Ø. If the carry flag is 1, move on to the next instruction in sequence. This instruction is the opposite of the above.

JZ (jump if zero). Jump to the instruction addressed by this instruction if the zero flag is set to 1. If the zero flag is set to Ø, move on to the next instruction in sequence. Compare to JC. Note that "if zero " means that the register in question is all zeroes, so that the zero flag is set to 1.

JNZ (jump if not zero). Jump to the instruction addressed by this instruction if the zero flag is set to Ø. If the zero flag is set to 1, move on to the next instruction in sequence. This instruction is the opposite of JZ above. Compare to JNC.

JM (jump if minus). Jump to the instruction addressed by this instruction if the sign flag is set to 1 ("minus"). If the sign flag is set to Ø, move on to the next instruction in sequence. Compare to JC and JZ above.

JP (jump if plus). Jump to the instruction addressed by this instruction if the sign flag is set to Ø ("plus.") If the sign

flag is set to 1, move on to the next instruction in sequence. This instruction is the opposite of JM above. Compare to JNC and JNZ.

JPE (jump if parity even). Jump to the instruction addressed by this instruction if the parity flag is set to 1 ("even parity"). If it is set to Ø, move on to the next instruction in sequence. Compare to JC, JZ, and JM above.

JPO (jump if parity odd). Jump to the instruction addressed by this instruction if the parity flag is set to Ø ("parity odd"). If the parity flag is 1, move on to the next instruction in sequence. This instruction is the opposite of JPE above. Compare to JNC, JNZ, and JP above.

CALL SUBROUTINE INSTRUCTIONS

Like jump instructions, call instructions cause the computer to depart from sequential execution of instructions. Also like jump instructions, they usually are "conditional" -- they usually operate only if some condition is met. And as with jump instructions, execution of instructions continues in sequence starting with the instruction at the address called (stated in the call instruction). The two types of instructions also resemble one another in that the address included is stated "backwards" -- first the less significant address byte, then the more significant. Also, these instructions do not affect flags.

The two kinds of instructions differ in that a call instruction "pushes" an address onto "the stack" -- namely, the address of the instruction to which the computer will "return" when it has finished the subroutine. See Section A.3. for a discussion of the stack.

CALL. Go to the instruction addressed by this instruction, and begin sequential execution there. This is an "unconditional call,"

and corresponds to an unconditional jump. All other call instructions are conditional, and correspond to the conditional jump instructions, each triggered by the state of one of the flags.

CC (call if carry). Go to the instruction addressed by this instruction if the carry flag is set to 1. If the carry flag is Ø, move on to the next instruction in sequence.

CNC (call if no carry). Go to the instruction addressed in this instruction if the carry flag is set to Ø. If the carry flag is 1, move on to the next instruction in sequence. This instruction is the opposite of CC above.

CZ (call if zero). Go to the instruction addressed by this instruction if the zero flag is set to 1. If the zero flag is Ø, move on to the next instruction in sequence. Compare to CC.

CNZ (call if not zero). Go to the instruction addressed by this instruction if the zero flag is set to Ø. If the zero flag is 1, move on to the next instruction in sequence. This instruction is the opposite of CZ above. Compare to CNC.

CM (call if minus). Go to the instruction addressed by this instruction if the sign flag is set to 1 ("minus"). If the sign flag is Ø, move on to the next instruction in sequence. Compare to CC and CZ above.

CP (call if plus). Go to the instruction addressed by this instruction if the sign flag is set to Ø ("plus"). If the sign flag is 1, move on to the next instruction. This instruction is the opposite of CM above. Compare to CNC and CNZ above.

CPE (call if parity even). Go to the instruction addressed by this instruction if the parity flag is set to 1 ("even parity"). If the parity flag is Ø, move on to the next instruction in

sequence. Compare to CC, CZ, and CM above.

CPO (call if parity odd). Go to the instruction addressed in this instruction if the parity flag is set to Ø. If the parity flag is 1, move on to the next instruction. This instruction is the reverse of CPE above. Compare to CNC, CNZ, and CP above.

RETURN FROM SUBROUTINE INSTRUCTIONS.

These instructions get the computer back from subroutines to the instruction following the call instruction that caused it to depart. Specifically, they "pop" an address previously "pushed" onto "the stack" off of the stack and into the program counter, causing the computer to next execute the instruction located at that address. Execution then continues sequentially from there. Each return instruction is associated with a previous call instruction, i.e. the program counter always returns eventually to the point in a program that it previously departed from (to an instruction following a call instruction). Therefore the number of returns executed is always equal to the number of calls executed (unless the machine halts).

Since these instructions always "pop" addresses in the order opposite that in which they were "pushed," they can be said always to operate on the "next available address" in the stack, so that the address need not be stated in the instruction.

Like "jump" and call instructions, all but one of the return instructions are conditional upon the state of one of the flags. Flags are not affected by return instructions.

RET (return). Return to the most recently pushed address. This is an "unconditional return."

RC (return if carry). Return to the next address on the stack if the carry flag is Ø. If the carry flag is 1, move on to the

next instruction in sequence.

RNC (return if no carry). Return to the next address on the stack if the carry flag is Ø. If the carry flag is 1, move on to the next instruction in sequence. This instruction is the opposite of RC above.

RZ (return if zero). Return to the next address on the stack if the zero flag is 1. If the zero flag is Ø, move on to the next instruction in sequence. Compare to RC above.

RNZ (return if not zero). Return to the next address on the stack if the zero flag is Ø. If the zero flag is 1, move on to the next instruction in sequence. This instruction is the opposite of RZ above. Compare to RNC above.

RM (return if minus). Return to the next address on the stack if the sign flag is 1 ("minus"). If the sign flag is Ø, move on to the next instruction in sequence. Compare to RC, RZ above.

RP (return if plus). Return to the next address on the stack if the sign flag is Ø ("plus"). If the sign flag is Ø, move on to the next instruction in sequence. This instruction is the opposite of the instruction above. Compare to RNC, RNZ above.

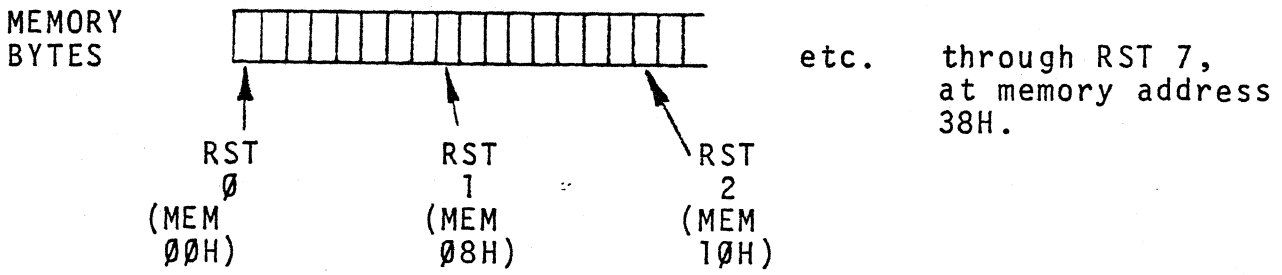
RPE (return if parity even). Return to the next address on the stack if the parity flag is 1 ("even parity"). If the parity flag is Ø, move on to the next instruction in sequence. Compare to RC, RZ, RM above.

RPO (return if parity odd). Return to the next address on the stack if the parity flag is Ø ("odd parity"). If the parity flag is 1, move on to the next instruction in sequence. This instruction is the opposite of RPE above. Compare to RNC, RNZ, RP above.

RESTART INSTRUCTION.

One special instruction, RST, resembles the call instructions in that it pushes a return address onto the stack and sends the computer off to another location. The address of the instruction following the RST instruction sequentially is pushed onto the stack, so that the computer will eventually return to its point of departure. Note that the RST instruction pushes the address of the instruction following RST -- otherwise the computer would return to the RST instruction itself and be trapped in an endless loop.

RST sends the computer (i.e. the program counter) off to one of eight pre-determined memory locations, each the first of a sequence of eight bytes, making up the first sixty-four bytes of memory.



Actually, the eight bytes associated with each RST can be reached by means of other kinds of instructions -- jump and call instructions -- and need not comprise individual routines. In the POLY 88, all sixty-four of these bytes are used in the monitor (discussed later).

The CPU executes an RST at one of two times. An RST instruction may be written into a program, in which case the instruction is in effect a "call" instruction in shorter form -- one byte instead of three. More usually, the CPU executes an RST when the running of a program is interrupted "from the outside". For instance, loading onto tape is a very slow process for the POLY 88, which

can output data much faster than the tape recorder can properly record it. So the computer outputs data to the tape on an interrupt basis -- it occupies itself with other tasks until the output port electronics indicate that it is time to output another data item to the tape. This forces a restart, which puts a book marker into the program so the computer will be able to get back to its point of departure, and sends the program counter off to a predetermined location to begin execution of a brief routine that causes the computer to output a data item to the tape.

INTERRUPT FLIP-FLOP INSTRUCTIONS.

Sometimes it is important not to permit interruptions of a program. For that reason, interrupts can be disabled--input or output electronics can be prevented from forcing a restart. Whether or not interrupts are disabled depends upon the state of a single flip-flop, called the interrupt flip-flop. When the flip-flop is set to 1, input or output electronics can force a restart until the interrupt flip-flop is reset to Ø, from which time interrupts are disabled till the flip-flop is once again set to 1. No flags are affected.

EI (enable interrupt). Set the interrupt flip-flop to 1.

DI (disable interrupt). Reset the interrupt flip-flop to Ø.

INPUT/OUTPUT INSTRUCTIONS.

These instructions cause the computer to input data from or output data to a device external to the computer -- like a keyboard. To be precise, the instruction causes the CPU to open an input or output port, which is assumed to provide a connection with some device. No flags are affected.

IN (input). Load one byte from the designated input port into the accumulator.

OUT (output). Send the byte in the accumulator out to the designated output port.

HALT INSTRUCTION.

This instruction brings computer operations to a stop. It first increments the program counter -- adds 1 to it -- so that the computer will resume with the next instruction. No flags are affected.

HLT (halt). Increment the program counter, then stop.

SYSTEM PHILOSOPHY

The POLY-88 system represents a departure from the usual microcomputer system organization in that it contains, in its minimal configuration, several sophisticated I/O devices. These devices - a keyboard, a memory mapped video display, and a universal serial port - are at fixed addresses and are accessed and controlled in identical ways on every POLY-88. The result of this standardization is that the power of the elementary machine can be increased manyfold by the use of dedicated read only memories (ROMs) resident in the CPU as hardware. The ROM can assume that the special dedicated devices exist and can use them. Other systems have no way to know where the devices are located or even if they exist. The ROM in the POLY-88 CPU is called the monitor ROM, and is supplied with every POLY-88 system sold. It contains bootstrapping functions, front panel simulator functions, I/O device drivers for the self-contained devices, utility programs, and initialization routines that configure the system when power is supplied or the system is reset externally.

The ROM also provides another extremely important function: it sets up conventions for the logical expansion of compatible software systems that are well engineered from the ground-up. Unlike the owners of large computers, many microcomputer owners intend to develop their own software systems. The monitor ROM allows these systems to share resources such as I/O handlers, supervisors or special purpose subsystems, but most importantly, it allows users to share their programs. The standardization of I/O handling means Polymorphic Systems can publish software which will run on any POLY-88 immediately, without modification of sections of the binary program to cope with the various I/O methods used on each system.

The standardization does not, however, limit the use of the POLY-88 with other types of I/O devices such as TTY's, serial CRT's, paper tape reader/punches etc., because the ROM allows the standard devices to be re-allocated once the system is up and running. Any type of I/O device can be substituted for the standard devices by loading a simple driver routine for that device and installing its address in one of the monitor's "wormholes".* The wormholes are used to communicate between any user program and a standard I/O device of any type. Thus, once the driver program is written for the special device to be used, it will work for any program that communicates through the wormholes, and it will work on any POLY-88. The wormholes and system standardization are dealt with in a later section.

Another advantage of the dedicated I/O devices on the POLY-88 system is reliability. Specifically, the dedicated memory mapped video display allows the ROM to generate a simulated front panel which replaces the usual array of switches and light emitting diodes. The hardware front panel used on most minicomputers and many microcomputers is a rather expensive and complex device. It was intended, in minicomputer systems, to be used only in emergencies or when "bring-up" (bootstrapping) the system. Microcomputers, however, since they are frequently used for program debugging and are bootstrapped very frequently, have a tendency to wear out front panel hardware quickly. Furthermore, the hardware switches and lights have no access to the all-important CPU registers in microcomputers as they do in minicomputers. The front panel switches on most microcomputers

* The term "wormhole" has been used to describe a hypothetical astronomical situation where a black hole connects to the "other side" of the universe. When this happens, information can pass through the wormhole, in only one direction, much as "assumptions" pass down the monitor's wormholes.

can be used for little else than examining and loading memory contents in binary and starting program execution at a certain address. Some microcomputers use serial output devices with ROM based monitors for program debugging. These systems have extremely limited I/O speed, and therefore result in a tedious interaction between man and machine that the POLY-88 eliminates by putting all important information on the screen at all times.

The front panel display of the POLY-88 shows all of the CPU registers, the "workspace" of the CPU, and a memory "window". The "workspace" is the areas of memory pointed to by the double registers, including the program counter and stack pointer. The workspace display shows, therefore, the program area, stack area, and data areas pointed to by HL, DE and BC. The memory "window" is an 8 X 8 block of memory that is displayed, with addresses, on the bottom of the screen. It can be used to view a selected area of memory or to point to data areas being modified. The philosophy behind the front panel display is that it is best to use the computer's high output capability to effectively answer all the programmer/debugger's questions about the machine rather than require him to ask.

Another application of the "anser-the-question-before-it-is-asked" philosophy is in the POLY-88 bootstrap tape loader. The resident ROM contains a complete audio cassette tape loader which reads absolute binary programs in a sophisticated format called POLYFORMAT. The files in this format are broken into short blocks, each with a name and number recorded with it. These names and numbers are displayed on the dedicated video display whenever tape is being read. They give the operator an indication of what file he is reading, where along the length of the file he is, and whether or not the tape is even being read properly. The names and record nubmers effectively make the data on the cassette visible, so that files can be separated from each other and located.

The POLYFORMAT has other advantages also. Its block structure allows the tape to be stopped in the event of an error and restarted before the erroneous block. Some recording formats require a file be read entirely without errors, or the whole loading must be restarted. The names on the records allow the computer to identify needed data on the tape such as relatively complex constructs like subroutines in a library that must be linked in a relocatable linking loader. The names also allow files to be packed closely together, without time-wasting leaders. POLYFORMAT includes definitions of several types of blocks (or records) such as: absolute binary (for programs), data (for text etc.), end (stops load), auto-execute (jumps into given address), and comment (displays a message for operator). The comment record is another example of the visibility philosophy. By placing comment records at the beginning and end of a file, the tape is made even more visible.

USING THE ABSOLUTE TAPE LOADER

The tape loader mode of the monitor may be entered at any time by resetting the CPU (either by depressing the front panel reset button or by applying power to the system). When tape mode is entered, the system video display is cleared and a small block appears in the upper left corner of the display. The small block is the cursor symbol used by the display driver program "DSPLY" which is resident in the monitor ROM with the loader. In order to load a POLY FORMAT absolute binary tape, the loader needs to know which encoding scheme to use and the name on the desired file. The encoding scheme can be indicated by typing either a "B" or a "P" on the system console keyboard. These stand for BYTE standard, the encoding scheme used in PolyMorphic published software, and POLY-PHASE, PolyMorphic's special very high speed encoding scheme. The loader transmits all necessary configuration information to the 8251 USART and the 5307 programmable baud rate generator on the CPU card

GO TO Pg. 67

according to the encoding specification. It selects the zero designated mini-card, which should be the audio cassette interface mini-card being used for the load.

When this is done, the cursor moves down to the next line and the loader expects a 1 to 8 character file name to be entered followed by a carriage return. The cassette motor control line is turned on, and the tape is read, comparing the record names found with the name of the desired file. As records are discovered with the correct name, they are accpeted. Reading continues until an END or AUTO-EXECUTE type record is encountered. (See section on tape format). If an END record is found, the cassette motor control line is turned off and the loader waits for another encoding specification (B or P followed by file name) or a "continue" command (just a "C" is typed-the old* EXECUTE type record is found, the cassette motor is turned off and the loaded program is executed by jumping to the address indicated in the AUTO-EXECUTE record.

Each record encountered, whether it is accepted or not, is acknowledged by having its name listed, followed by its record number, on the system console display. Thus if it is desired to simply examine the contents of a certain cassette tape, the tape loader can be told to search for an impossible name, such as no name at all. It will continue indefinitely, searching for the nonexistent name on a record, each time showing the name and hexadecimal record number of each record it finds. The record number is recorded in the RCD# field of the record.

Occasionally, while a file is being loaded, a COMMENT type record will be encountered and the message it contains will be displayed on the system display. All files should have a COMMENT record at the beginning for documentary purposes, and it is the appearance of this COMMENT message on the system video display that indicates the loader has recognized the desired file and will load it. COMMENT records are very useful,

*name is used).

Go To Pg. 57

as they can indicate such things as the portion of a large program that has loaded (the message serves as a flag some distance down the tape), or that a program has finished loading and is executing (an AUTO-EXECUTE followed the COMMENT).

All mass storage systems must cope with errors in some way, and magnetic tape is far from an exception. A very long program has a relatively high probability of loading incorrectly simply due to noise and other factors which create "soft" or read-only errors. If the lost data can be re-read, the soft error is unlikely to occur again, and the loading can continue. The POLY-FORMAT allows an erroneous record to be re-read without starting at the beginning of the file. The record structure of the POLY-FORMAT is such that each record is completely self-consistent. This means that if the cassette tape is rewound beyond the erroneous record and the loader and recorder are restarted, the loader will find the first complete record (if it is restarted in the middle of a record) and will reload records up through the record that was lost. This process may be repeated until a difficult record loads properly - a very time consuming proposition if considered with a file structure which requires restarting of the load at the beginning of the file. An error is indicated to the operator by a question mark on the video screen and stopping of the cassette motor. If the motor control is not being used (some recorders have motor voltages and polarities inconsistent with the audio cassette motor control drivers) then the tape will continue to play after the loader has stopped at the error. In this case, it may be necessary to rewind the tape some distance, and it will be helpful to check the record numbers to find the original spot. A depression of a C key on the keyboard will resume reading of the tape. If the motor control is being used, it will be impossible to rewind the tape until the motor control is again

turned on, and again, this can be done by simply depressing the C key on the keyboard. The name and format of the file being loaded is retained and need not be re-entered.

FRONT PANEL MODE

Instead of a hardware front panel, the POLY-88 uses a program which drives a simulated front panel display onto the system video terminal. The display, shown in appendix C, is present whenever the monitor is in the front panel mode, and is updated each time a command is executed, or the registers are modified in any way. The display is thus always a reflection of the contents of the registers and memory at any instant, just as if it were hardwired into the CPU and the memory address and data busses. Visible in the display are:

(Vol II Pg 81)

*contents of CPU registers: program counter (PC), stack pointer (SP); accumulator (A) and general purpose registers.

*contents of memory areas pointed to by general purpose registers: program area, stack area, and the areas pointed to by BC, DE, and HL.

*a moveable memory window which shows 64 contiguous locations and their addresses.

*the status of the carry, sign and zero flags decoded into an easy to read form.

The front panel display driver program is complemented by a command interpreter program. In most cases, a single keystrike on the system console keyboard will allow the operator to:

*interrupt a running user program to bring up the front panel.

*single-step, run with breakpoints, or return to full speed execution of the user program.

*move the memory window to a given location.

*enter single bytes or long strings of bytes in hexadecimal into memory with instant verification of entered data and easy backup for error correction.

*trace byte-reversed address pointers in memory by moving the memory window to the given address.

*move the memory window to point at the program, stack or data areas currently being used by the user program.

The commands in Appendix B are primitives and should be used in combination to provide a powerful interactive system for manipulating memory data and debugging machine language programs. There is, for example, no command for setting the contents of any given general register. Instead, there is a command for pointing the memory modify window at the save area on the system stack where a given register was stored. This allows the contents of the register to be modified using the rest of the commands, such as the Jumbo (J) command, which allows entry of a full address in its normal byte order instead of the byte-reversed order of 8080 addresses. Another example might be using the I (indirect) command after pointing the window at the register save area on the stack. This will point the window at the memory area that the register points at.

In other words, if the register was the program counter, a sequence of "SPI" would leave the window displaying the program area. The program area could then be modified using the full power of the front panel commands.

In order to fully introduce each command and its possible combinations with others, the following text will take the reader step-by-step through the procedure of loading a simple program, correcting entry mistakes, checking its logic using the I command and the X (single-step) command, and finally, running it.

USING THE FRONT PANEL MODE

Suppose we wish to construct a simple program in an available location in RAM. The demonstration we will use is a video display test which loads each location of VTI memory with the low-order address byte of each location. This has the effect of

displaying all possible characters and graphics patterns on the screen in a cyclic group of 256 characters. The display is thus repeated four times vertically.

STARTS AT C80H

The program looks like this in symbolic assembly language:

```
21 00 F8 ---------LXI     H,0F800H   ;start at top of system screen
75 ------ LOOP:   MOV     M,L        ;put out each location's low addr byte
23 ---------------INX     H          ;next location
7C ---------------MOV     A,H        ;get high addr byte for comparison
FE FC ------------CPI     0FCH       ;is it off the screen yet?
C2 83 0C ---------JNZ     LOOP       ;no - keep going
76 ------ HLTAGN: HLT                ;yes - OK, were done, stop.
C3 8B 0C         JMP      HLTAGN     ;sometimes interrupts break a HLT, so
                                     ;go back to the HLT when they do.
```

In hexadecimal machine code:

21 00 F8 75 23 7C FE FC C2 83 0C 76 C3 8B 0C

assuming that we want to load it at C80H, which is a free space in the system RAM. The problem now is to correctly load this hex into the RAM at that address and then send the CPU off executing it.

Turn on the POLY-88, and push the front panel reset button. The machine should have a CPU card with 4.0 monitor installed as per appendix A, and a video terminal interface card with its address switches at 0F800H, also as per appendix A. The screen should show only the small cursor block in the upper left corner. This is the prompt character (actually no prompt char. per se')*for the tape loader system. Since we want to use the front panel mode, push the control Z (hold down CTL before and while pushing Z). Instantly, the front panel display should appear on the screen. Appendix C shows an example of the front panel display with an explanation of the various data areas in it. For now, the part that interests us is the memory modify window at the bottom. The window is a 64 byte section of memory which shows, in hex, the locations before and after the "current" window position. The byte actually at the current window position is indicated by a right-arrow at the left

* but only a cursor symbol.

center of the block. The address of this byte and the leftmost byte in each row is displayed at the far left of the screen, also in hex.

Now, to enter the test program into the RAM, we first point the window at the desired address:

LC80(ret) Look AT 0C80H (CR) [Address 3200 D]

where "(ret)" means carriage return (CR on some keyboards, RET on others). The display should now show 0C80 in the address next to the arrow. To enter the bytes of our program, we can simply type the hex for each instruction followed by a space. When hex is being entered, the termination character for each byte is interpreted as a valid command. In this case, the space indicates that the window pointer is to be incremented, hence each byte goes into a succeeding location. The program entry looks like this:

21(space)00(space)F8(space).....76(space)C3(space)8B(space) 0C(space) where the "(space)" means, of course, a blank space from the space bar.

Suppose an error had been made while entering data into memory. The window may be moved back one location with the backspace command, control H. The erroneous byte may be reentered, and after the usual space, the rest of the program bytes may be entered. If the error is detected before a termination character is typed, it is only necessary to continue typing in the hex for the correct code until the last two hex characters shown at the bottom of the screen are correct. The hex input routine used by the command interpreter shifts hex nybbles into a two byte register from the bottom, so when it returns to the calling program on receipt of the termination character, it leaves only the last four nybbles in the

double register. In the case of hex input to the window location, only the bottom two nybbles are actually used by the program. Any previous hex digits are ignored. The use of the last hex characters typed in is built into all the other commands which expect a hexadecimal input of some kind.

One of the commands which expects a hex input is the J command. J stands for Jumbo, a mnemonic which indicates a double word is to be entered at the current window location. The J command is followed by up to four hex nybbles or characters and a carriage return, thus:

JF8ØØ(RET)

The contents of the two locations at and following the window are modified to contain the "byte-reversed" double word that was entered. Actually, as mentioned above, only the four last characters typed in for hex are used, so if an error is made on entry, just keep typing until all four of the last characters are perfect. Since the register that the bytes are shifted into is started out with all zero contents, a small hex number need not be typed in with leading zeroes (unless, of course, it is being re-typed after an error). The way to enter an address of ØC8Ø, then, is to type the J followed by C8Ø followed by a carriage return; JC8Ø(ret). One important fact about the J command is that it does not move the window pointer. The reason for this is to allow the use of the I command immediately after a J. Sometimes this combination can be useful.

The I command is the "indirect" operator. It takes the two bytes at and following the window location and puts them into the window pointer. It "jumps" to the address currently shown in the memory at the window pointer. This is a very useful function for tracing programs that do JMP's or CALL's by placing the window pointer over the address of the JMP or CALL and then typing I. It is also immediately after a "J" as a check

of the address entered. If the address is correct, the window will show the data that are supposed to be pointed to.

It should be pointed out that the I and J commands work with double words in memory that are stored in what is known as "byte-reversed" format. The 8080 puts the high order byte of an address stored in memory into the high order register of a pair when POP or LHLD type instructions are executed. PUSH and SHLD instructions operate similarly. Addresses in JMP and CALL instructions also follow this rule. Although it seems logical to arrange addresses this way, it is normal to enter data into memory incrementing addresses between bytes entered. This, unfortunately, means that addresses are typed in low order byte first. Addresses are also displayed backwards in the normal representation of data in memory: addresses increasing to the right. The seemingly backwards storage of addresses has come to be called "byte-reversed".

Now that the program has been entered correctly, we would like to run the program. The first thing to do is to set up the program counter to point at the first instruction of the program. To do this we will use one of the S commands: SP, which will point the window at the area on the system stack where the program counter is stored. Now, of course, the actual program counter could not be stored on the stack, because the program which we are running that displays the front panel and interprets our commands is moving the PC up and down in the monitor. The program counter we will modify is the one that will be restored into the "real" program counter when we want to execute the program. In other words, as far as we are concerned, the actual program counter is stored right there in memory, along with the values of all the other registers. Since the stack may have any value in it when we preseed control Z, the locations actually used to store the register values are unknown. The monitor however, keeps track of these locations

and will point us at any one we want if we use the S type of command.

So, to set up the program counter, we point the window at the proper place on the stack with an SP command, and then do a J:

SPJC80(ret)

The front panel display at the top of the screen should now show the 0C80 we just entered in the PC register. The area to the right of the PC double word shows the memory pointed to by the PC, which is the program area. It should show part of the program we have loaded. The arrow at the bottom of the front panel display points upwards at the actual locations that all the register pairs point to. The 21 hex that was the first opcode of our program should be visible above this arrow in the field next to the PC.

The memory window should also show the new PC value, except it will be backwards because of the "byte-reversed" address format. The window should show 80 followed by an 0C. Now, to check this value, let's see if the PC actually points at the program. Press I and the window should show the first instruction again: the 21H. For one last check before we run the program, type a (ret) and the window will scroll up one row (8 bytes). We could move backwards one row by typing line feed (LF). This gets us closer to the address in the JNZ instruction that we want to test. Space down to the locations following the JNZ (following the C2) and press I. The window should point at the address we called "LOOP" in the symbolic assembly program. The instruction at this address was a MOV M,L, which has hex opcode 75. The 75 should be in the memory window after the I command is typed.

If this last test works, we are ready to step the program

through one cycle of its loop to see what happens. The program counter is still set up to 0C80, so press the single-step command key, X. The program counter will advance to 0C83 and the HL register pair will be loaded with F800, the data from the second two bytes in the LXI H instruction. On the next single-step, the first byte will be transmitted to the video screen, but since the front panel display is replaced on the screen after the byte the instruction transfers, we do not see anything happen. The next instruction increments HL to F801. Next, A gets the contents of H, then it is compared with FC hex in the CPI 0FCH instruction (FE FC). Finally, since F8 does not equal FC, and the zero flag is not set, the JNZ instruction goes back to 0C83 to continue the loop.

The program seems to work when single stepping, so the final test is to execute it at full machine speed. This can be done by pressing G. The entire screen should fill with the test pattern of consecutive ASCII characters and graphics patterns, in a cyclic replication four times down the screen. The single-stepping lost the first character in the upper left corner of the screen, though.

When the test pattern is verified, the front panel mode may be re-entered as it may be at any time by typing the control-Z. The front panel will appear, showing the program counter just as it was, halted at the end of the program on the 76 (HLT) instruction. The HL pair should have the last screen address used: FC00H.

4.0 UTILITY PROGRAMS

In addition to the directly operator-apparent features of the 4.0 monitor, there are also some software-apparent features which make I/O handling and other difficult or routine operations easy in machine language. Program patching is easy on the 4.0 monitor becasue the most common patching area for programs being moved from system to system is in the I/O section, and the 4.0 monitor contains patch routines for direct substitution into most programs. These include routines for transmitting characters from the accumulator to the console display from the keyboard to the accumulator, and from the USART port to the accumulator. Several hexadecimal conversion routines are included which will handle nybble, byte and doubleword conversions from the keyboard. Other utility functions include lower to upper case folding of characters input from the keyboard (lower case characters are converted to upper case before passing on to user), screen manipulations such as clear, tab, space and carriage return, and USART setup programs for Polyphase, Byte Standard or user defined USART mode.

The following discussion will be found useful to beginning programmers who wish to use the utility routines to simplify their programming. Advanced users will want to study the section on software conventions and methodology in order to make their complex programs more general , flexible and compatible with both PolyMorphics published software and other user's software. The utility programs in the 4.0 monitor, although designed to be universally available to all users of POLY-88 systems, may occupy different address areas in later versions of the 4.0 (4.1 etc.). This would make programs written using the utility programs obsolete, and would require adjustment of the addresses in all programs that were to be transferred from one verson to the next. To avoid this mass adjustment of addresses in user code, PolyMorphics will not publish any version of the 4.0 monitor with the utility routines relocated unless it is absolutely necessary to do so. However, the possibility does

GO TO K
pg. 68

exist, so it is wise for programmers who intend to use the utility programs to be aware that this may happen. In any case, the utility routines are intended only to allow users to construct small software systems without the need to generate all the standard utility functions themselves.

The vital utility functions - the I/O functions - are made available to all levels of programming without reservation. The standard character transmission operations are made available through an ingenious device known as the "wormhole vector", which puts the I/O routines at absolutely stnadard positions in the CPU resident system RAM memory. Thus, no matter what version of monitor is being used, or even whether the monitor has been replaced by a disk operating system or time-sharing system, the wormholes in the vector will remain fixed and programs using them for I/O need not be changed. It is possible, then, for anyone to write programs which will last through many cycles of monitor or supervisor redesign by simply using the wormholes for I/O rather than any fixed-address I/O subroutines.

The concept of standardizing the addresses of various important pieces of information has been extended in 4.0 to include many things beside the addresses of the character I/O handlers. The storage areas for such widely accessible variables as the present video screen starting, ending and cursor addresses are defined by the 4.0 in an absolute, fixed manner, so that any program can modify or examine them with confidence in their locations. The definition of all the standard locations and mthods of use of the variables in the system RAM are what give the POLY-88 its software flexibility (an interesting apparent contradiction: the absolute rigidity results in increased flexibility). The techniques for using the wormholes and other "system variables" (see glossary for definition of "system variables" and other terminology used here) are described in detail in later sections. It is the intention of this section,

however, to describe the use of the utility functions by programmers who would rather see their systems work than to generate a masterpiece of programming generality and portability.

First, let us look at the basic I/O functions. Almost all programs need some sort of communication with the keyboard and system video display. The first two wormholes are defined for this purpose. Wormhole zero (WH0, pronouned whoo) is the console input wormhole, while wormhole one (WH1, pronounced whee) is the console output wormhole. Notice that these wormholes are defined by their logical rather than their physical function. Either of them may be changed to operate with any actual physical console input or output device desired, but as far as the programmer is concerned, all they do is communicate with the computer operator through some sort of character-oriented device. The total effort that must be expended to talk to the operator is thus, to do a CALL to the appropriate wormhole. The character will be taken from the accumulator or placed in the accumulator by the subroutine that the wormhole "contains" (checkout how the wormholes actually work if you want to, but for now, pretend each one contains a complete subroutine in its four bytes). All the wormholes work the same way, by transferring one character at a time from the accumulator the addresses of all the wormholes are shown below.

Wormhole	Address	Logical device accessed (example)
WH0	0C20	console input (generally a VTI keyboard)
WH1	0C24	console output (generally a VTI screen)
WH2	0C28	system input (binary from a mag.tape)

WH3	0C2C	system output (binary to a mag tape)
WH4	0C30	aux. syst. input (secondary mag or paper tape)
WH5	0C34	aux. syst. output (secondary tape)
WH6	0C38	text input ("saved" pgm listing from tape)
WH7	0C3C	text output (printer or tape for listings)
WH8	0C40	undefined input
WH9	0C44	undefined output

The first three wormholes are loaded by the monitor with the proper data to make them act as subroutines when the system is powered up or when the front panel reset button is pushed. The subroutines in the monitor are "installed" automatically in WH0, WH1 and WH2, but the other wormholes, since they require very complex I/O handlers, must be installed later, with I/O routines in RAM. The actual routines that the monitor installs are: the DSPLY program - for driving a PolyMorphics VTI-64 or 32 video display, KI - for getting characters from the keyboard port of the VTI board, and USRTI - for getting characters from the 8251 USART on the CPU. As described later, these default allocations may be changed easily. Let us illustrate the use of the wormholes with a program which echoes the characters typed on the VTI keyboard port onto the VTI display:

```
NEXTCH:   CALL WH0        ;get a character from console device
          CALL WH1        ;put on console display
          JMP  NEXTCH     ;go back for more
```

Notice that even this simple loop allows complete control of the video display through the DSPLY routine which recognizes many of the standard control codes.

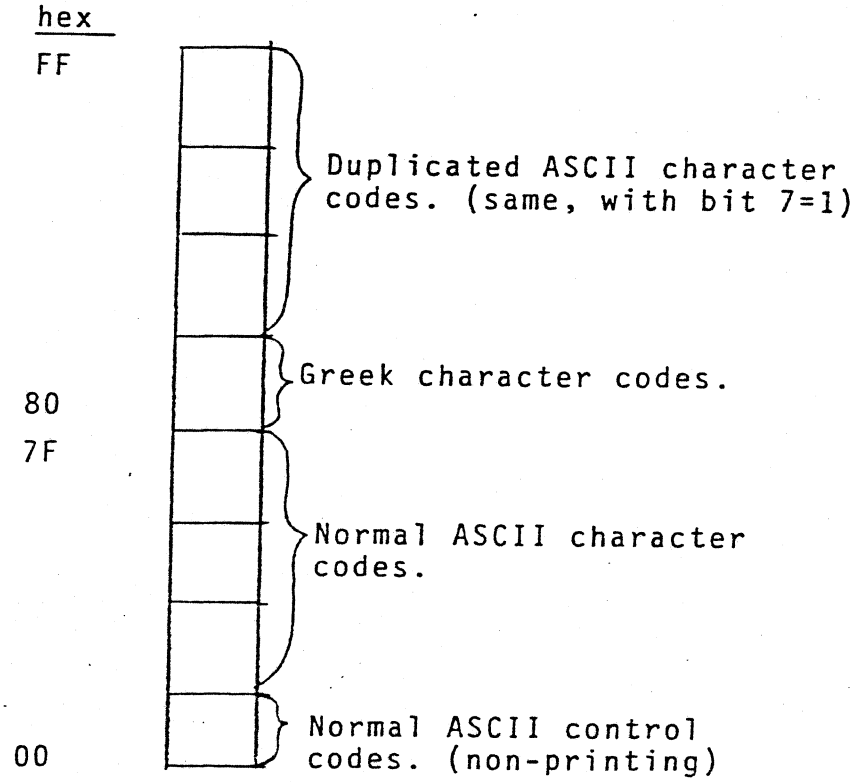
DSPLY, KI and USRTI

More specifically, the codes that DSPLY recognizes are:

ASCII code (hex)		keypress	function
DEL	7F	rubout	moves cursor back, deleting char.
CR	0D	return	skips a line and puts cursor at left side of screen. "car. ret."
FF	0C	CTL/L	clears screen, leaving cursor at "home" - upper left. "form feed".
VT	0B	CTL/K	moves cursor to "home" position, in upper left corner."vertical tab".
HT	09	CTL/I	skips cursor to next horizontal position evenly divisible by eight. "tab" function.

DSPLY also does a little rearranging of the character set given to it. If the character is a control code, as are several of the above, it is not printed on the screen. However, if it is a control code but has a high bit 7 (the top bit) then it will print as a greek character, but will not work as a control character. It is thus impossible to use the graphics capability of the VTI card by transmitting characters to the screen through DSPLY. Any graphics character comes out as a regular ASCII character, as if bit 7 were low. Note that DSPLY corrects for the backwards polarity of bit 7 as the VTI card expects it. Normally, a high bit 7 will display graphics, and a low bit 7, ASCII, when bytes are transferred directly to the VTI as if it were memory. The DSPLY program will take either polarity in bit 7, and will always generate characters rather than blocks. The map of the DSPLY input expectations looks like this:

Mapping of DSPLY expected input codes.



The other two default wormhole subroutines operate in a more elementary manner. Neither the KI nor USRTI wormhole subroutines map their character codes in any way. KI gets 8 bit characters from the VTI keyboard without zeroing bit 7. It is possible using KI to load binary data or special function codes from an auxiliary keyboard from the VTI keyboard port. Normal ASCII characters are expected to have zeroes in bit 7, so, if bit 7 of the keyboard is not grounded, then it should be zeroed by the software. Bit 7 here means, as usual, the highest bit, not the next to highest. USRTI operates exactly as KI except that it fetches bytes from the USART.

To test the USART and a cassette tape system, the following direct echo loop can be loaded into a free spot in RAM:

```
LOOP:     CALL  WH2     ;get a byte from USART
          CALL  WH1     ;display its ASCII representation on scrn
          JMP   LOOP    ;go back for more
```

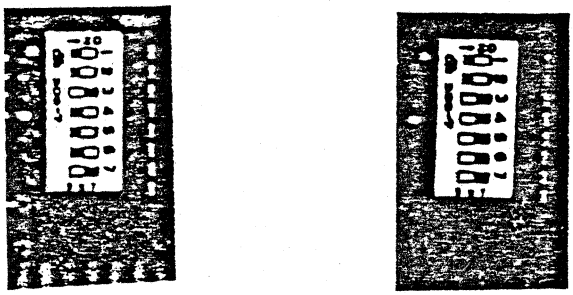
Before this program will work, the USART must be configured to read from the tape. To do this, reset the system with the front panel button, and proceed as if a tape were being loaded. Put in a dummy name and a carriage return. Then bring up the front panel with a CTL/Z, and run the loop. When the cassette is played into the USART through an audio-cassette interface, the characters on the tape will appear on the screen. Many of the characters will be control codes and will clear the screen or return the cursor, but some of the patterns on the tape will be discernible, such as the string of lower case f's that represent the leader of hexadecimal E6's on a POLYFORMAT record.

APPENDIX A. INSTALLING 4.0

A number of hardware changes are necessary to convert the P-88 to 4.0 monitor compatibility. After installing the 4.0 monitor ROM (read only memory) in the right-most ROM socket on the CPU card, the following points of possible incompatibility should be checked and corrected as needed:

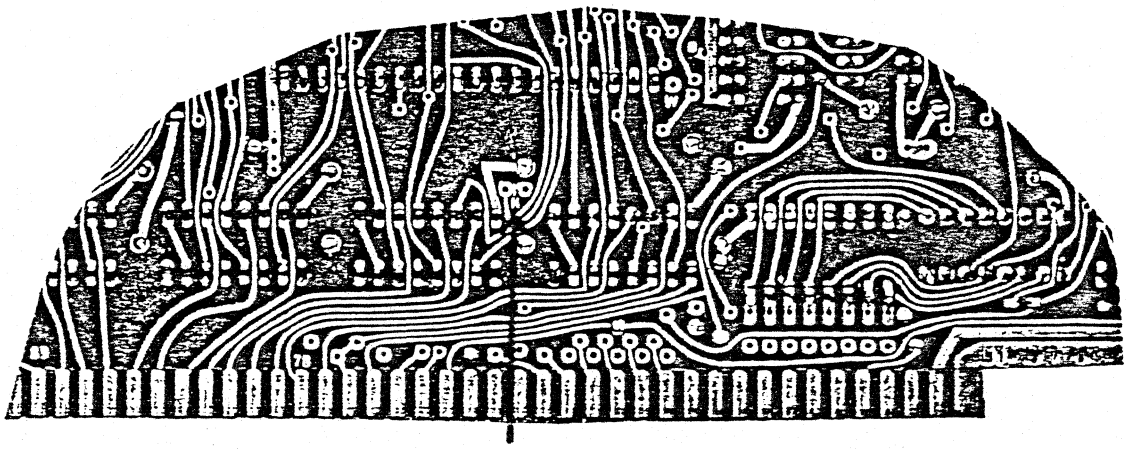
1. The system video screen, although it may be moved once the system is running, is initialized to run at F800 hex. In order to change the address on the video card, configure the address selection switch as shown below. The movement of the video address allows greater expansion of contiguous program memory.

OLD SWITCHES NEW SWITCHES



2. A short trace labelled "K" on the back of the CPU card is normally cut for the earlier monitors since they do not use interrupts from the USART, and this trace connects the USART interrupt to VI3 (vector interrupt three). This trace should be reinstalled if it h as been cut by soldering a short piece of bare wire into the two pads on either end of the trace as shown in Fig. A.2.

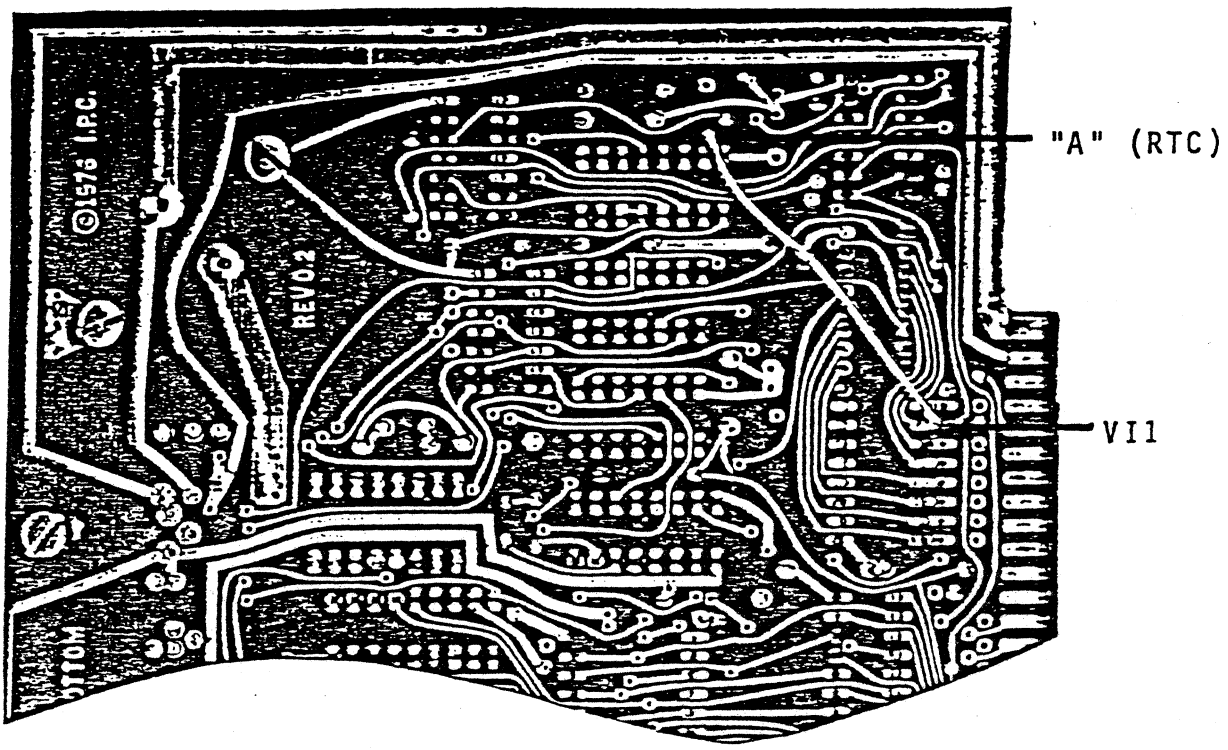
Figure A.2. "K" TRACE REINSTALLATION



Bottom edge of back of CPU card shown.

3. The 4.0 monitor uses the 60 Hz real time clock interrupt which is normally not connected for the earlier monitors. To connect the clock, run an insulated 2" jumper wire on the back of the CPU card from the "A" pad to interrupt pad 1. The "A" pad is on the right center of the CPU card

Figure A3. RTC JUMPER



(looking at the back), however, the "A" is on the front of the card, next to a 74LS109. Again on the back of the card, vector interrupt 1 can be found in a group of eight pads arranged in a horizontal line at the bottom right. Interrupt 1 is second from the right in the upper series of eight.

5. Earlier monitors had keyboard driver routines which zeroed the high order (bit 7) bit of the data coming from the keyboard. The 4.0 monitor leaves this bit unchanged when data is obtained from the keyboard via the keyboard driver routine. This is so the high order bit may be used for inputting binary directly through the keyboard or for increasing the number of valid key codes on the keyboard to include special functions. An example of this might be the use of the keyboard driver routine in a text editing system where a special cursor control keypad transmits a high bit 7, and the normal keyboard transmits the normal zero bit 7.

   The net effect of this change is that all keyboards must transmit a zero bit 7. To do this, make sure that this bit is grounded on the keyboard itself, or ground the "B" pad on the VTI board. This pad connects to bit 7 of the input to the 8212 keyboard data latch. It is near the strobe-polarity jumper pads in the upper right corner (looking frontwise at board). Version 1.0 and later video cards will all omit the "B" pad, because their keyboards are expected to supply a zero bit 7.

APPENDIX B. MONITOR COMMANDS

The following list comprises the set of primitive operators or commands available in the 4.0 monitor in the front panel mode. Front panel mode may be entered at any time by striking the control Z key on the system console keyboard. Further commands on the system console keyboard have effects which are immediately reflected in the front panel display. When front panel mode is to be exited, the interrupted program may be restarted transparently, since the entire status of the CPU is saved on the current system stack upon entry to the monitor.

Key or key sequence	Effect
control Z	Interrupt currently executing program. Front panel mode is entered. Status of CPU (PC,SP,regs.,flags) is saved on the system stack.
X	Execute the next instruction of the interrupted program and return to front panel mode to display results.
G	Go to the next instruction of the interrupted program and do not return.
Lxx..xx(CR)	Look at address xx..xx with the memory modify display. The variable-length address (up to four last hex digits accepted) is placed into the memory modify display pointer.
space	Move the memory modify display pointer forward one and redisplay everything.
BX (control H)	Move the memory modify display pointer back one and redisplay everything.
CR (carriage ret.)	Move the pointer forward 8 positions. This has the effect of scrolling the display up one line.

Key or key sequence	Effect
LF (line feed)	Move the pointer back 8 positions. This has the effect of scrolling the display down one line.
xx xx(any command)	The last two hex characters before the command are entered into the location pointed to by the memory modify pointer. The command is then executed.
Jxx xx(CR)	Jumbo data word (double-word) is entered in byte-reversed format at and following the memory modify pointer. The last four hex characters before the carriage return are used.
I	Indirect display. The two bytes at and following the memory modify pointer are placed into it in reverse order, so that if they represent the address in a JMP instruction, the pointer will be moved to that address.
SP (Program Counter) SH (HL) SD (DE) SB (BC) SA (Accumulator/flags)	Stack modify. The memory modify pointer is moved to that address on the stack where the indicated register pair was stored on program interrupt. If the location at the memory modify pointer is modified, the register display will show the contents of the appropriate register as having changed, and when the G command is executed, program execution will continue with the new value. To enter a double-word, the J command may be used. A single byte may be inserted in one register of a pair by simply entering it for the lower register and by spacing once over the lower register to enter it into the upper register. Data at the address pointed to by a register pair may be modified by using the I command to move the memory modify pointer to the appropriate area of memory.

Key or key sequence	Effect
U (or other illegal command)	Update the display. This can be used to watch dynamically changing events such as the real time clock counter being incremented in system memory, or an I/O buffer filling.
T	Tape system is entered. This is the same tape system entered on power up or reset from the front panel reset button, except that the system constants that are initialized by either of these latter entry methods are left intact. These include the video screen address, console wormholes and interrupt service routine addresses for the USART, keyboard and RTC.

APPENDIX C. FRONT PANEL DISPLAY

Shown on system video screen

```
PC 008C 0C 0C 7E B7 C2 8B FE 8C
SP 0FFA FF 8C 00 FB 7C 2F 31 A0
HL 0C0C 49 48 D5 10 08 56 C6 DA
DE 0C51 21 00 88 A7 BA DC 0F 1F
BC 0000 A0 19 70 31 00 10 06 FF
AF FF86 CNZ       ↑

1FE3    FF FF FF FF FF FF FF FF
1FEB    FF FF FF FF FF FF FF FF
1FF3    FF FF FF FF FF FF FF FF
1FFB    FF FF FF FF FF 0I CA CD
2003 →  00 AA FE 3A 40 21 CE 8F
200B    76 C2 3C 03 2A 27 0C 3A
2013    26 4F 3A 29 2A 44 0C C9
201B    B9 83 B2 16 F0 C8 33 BA
```

Explanation of display

```
PC=008C hex   (PC)=B7 hex
SP=0FFA       (SP)=FB
H=0C  L=0C    (HL)=10
D=0C  E=51    (DE)=A7
B=00  C=00    (BC)=31
A=FF
flag byte=86
```

Carry, sign and zero flags are shown as all high ("CNZ") for explanatory purposes only, as this is an impossible condition and does not correspond to the flag-byte shown. A low flag is displayed as a blank, eg. " " means C=S=Z=0.

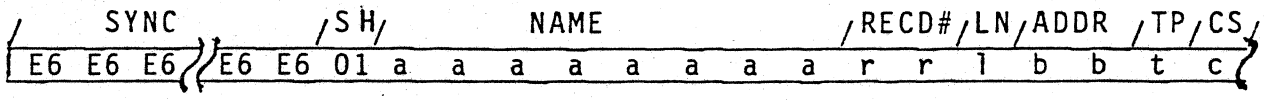
Memory window is displaying data around 2003 hex. If data is entered it will replace the 00 to the right of the arrow. Addresses increase from top to bottom and left to right. Location 2003 contains 00, 2004 contains AA etc.

The display shown above appears on the POLY-88 system console whenever the monitor is in the front panel mode. It is updated each time any command is executed, so it always reflects the contents of the memory and registers accurately.

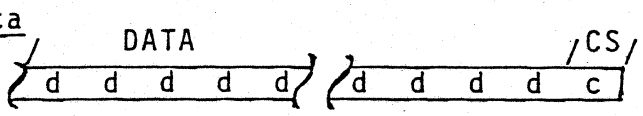
The top of the screen shows the simulated front panel itself, and the bottom shows the memory window. To the right of each register pair is a display of the locations on either side of the address in the register pair. The up-arrow near the middle of the screen points to the actual location that the register pair points to. This is the location that would be modified or read if the associated register pair is used as a pointer (LDAX B, LDAX D, or MOV A,M type of instructions). Since the A register and the F (flag) byte are never used as concatenated bytes in an address, they do not have a memory

APPENDIX D. POLYFORMAT DEFINITION

The next page shows five examples of POLYFORMAT records, one of each of the currently defined types. Each record has the same basic structure as shown below:



Data

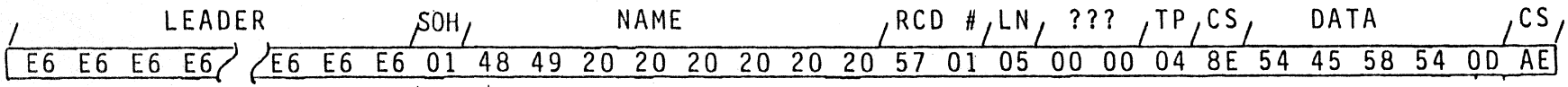


Each record consists of a HEADER followed by a possible DATA field. The fields in the header above have the following meanings:

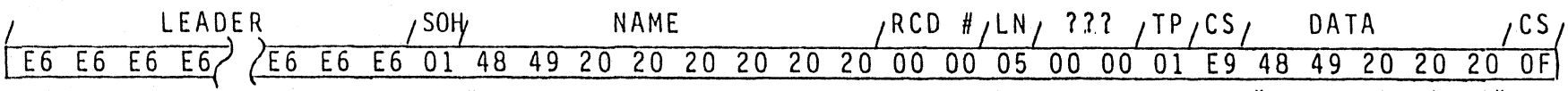
field designator	purpose/description of field	field name
a	eight character record name	NAME
r	record number (0 to 65536)	RECD#
l	length of data 1 to 256 bytes	LN
b	bias address or absolute addr.	ADDR
t	one of five record types	TP
c	checksum modulo 256 neg.sum	CS
d	data bytes binary 8 bits	DATA
E6	hexadecimal E6 sync chars.	SYNC
01	ASCII "start of header", SOH	SH

All records begin with the SYNC characters (exactly 16 of them) followed by an ASCII SOH character. NAME and RECD# are on all types of record, but record types without data have undefined LN fields. The ADDR field is defined for absolute binary types and auto-execute, in which cases it indicates loading or branching

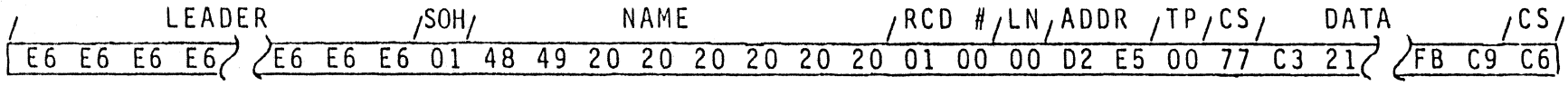
DATA (text,relocatable object etc.)



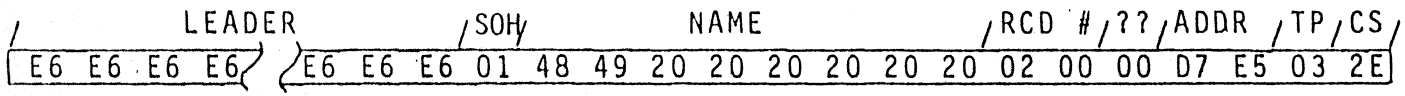
COMMENT (message to operator during load)



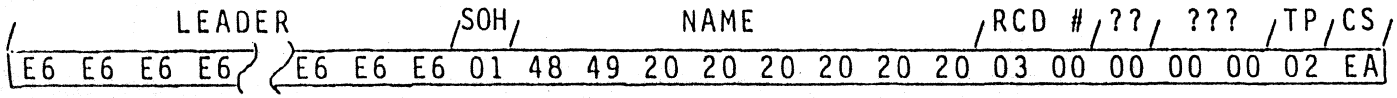
ABSOLUTE BINARY (core-image for binary object code)



AUTO-EXECUTE (jump to address given in ADDR field)



END (stop tape loading - indicates no more data)



address respectively. Both RECD# and ADDR are in the standard 8080 byte-reversed format, so that they work with LHLD and SHLD instructions. In other words, the upper byte in memory contains the most significant byte of the double word. The type field has the following meanings:

TP (type) field	Record type indicated
00H	ABSOLUTE BINARY. Used for program storage where data must be reloaded into the same place it was copied from originally. The address of the area is contained in the ADDR field. Data is copied starting at this address for the number of bytes given in the LN field.
01H	COMMENT. The data in the data field is echoed to the system video display as a message to the operator. The checksum on the data may be ignored.
02H	END. This terminates a file. Any physical device capable of being turned off is stopped, if it is the device supplying the data. LN,ADDR fields are undefined. No data follows the header.
03H	AUTO-EXECUTE. The address in the ADDR field is jumped to. LN is undefined and no data follows the header.
04H	DATA. Information in the DATA field is loaded into a buffer somewhere and used by a program. Data in this record has no address associated with it as object code programs do, hence the ADDR field is undefined. Data is 8 bits/no parity as is the ABS. format and might be ASCII text (bit 7=0) or relocatable object code etc..Length of data part of record is given in LN.

The length field specifies from 1 to 256 bytes in the data field, but it is given the following meanings:

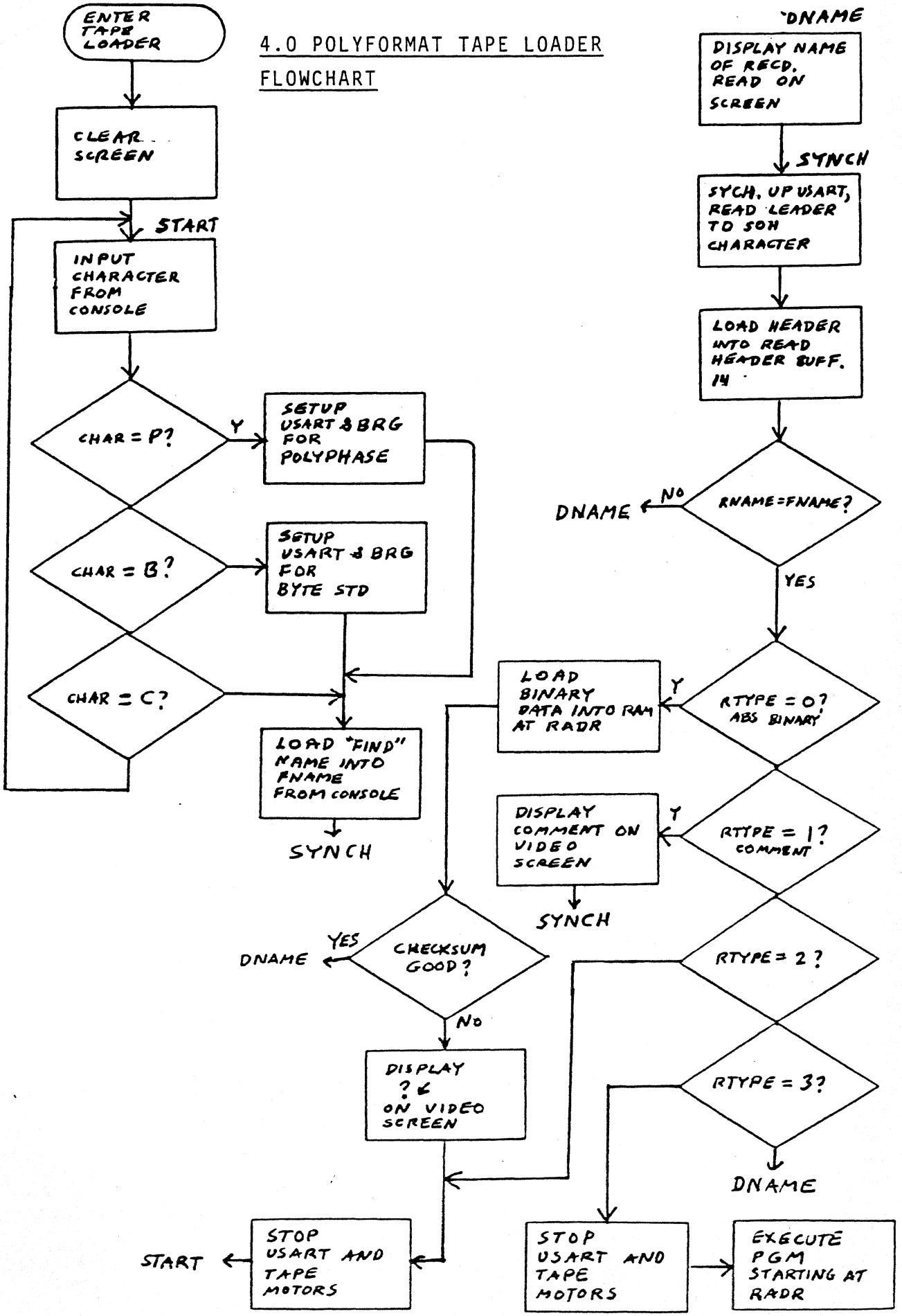
LN (length) field	Actual data block length (bytes)
1 to 255	1 to 255
0	256

This is so that the value of the length field corresponds with the actual number of bytes transferred, but also so that the records may be an even ¼K block maximum. The seemingly special case of zero length (0 LN) field for a 256 byte data block is actually natural, since it represents the overflow condition (100H with the 1 dropped). It is easy to write program loops which work with this definition with no special logic to detect the 256 byte case.

Checksums are applied to both the header and the data block separately. The checksum is the negative sum of the bytes preceding it. When it is added to the preceding bytes by a loader, the result should be zero. A valid header must have a correct checksum, or it will be ignored. If a data block following a valid header has a bad checksum, a checksum error is generated and loading of the file stops until the erroneous record can be re-read correctly. Record types without data blocks do not need a second checksum following the header checksum. Header checksums do not include the SYNC characters or the SOH character.

Records on a magnetic tape are separated by an inter-record-gap or IRG of sufficient length that the tape may be stopped between records and restarted without loss of the next record. This is to allow controlled loading and storing of data to match the processing speed of a program. Records may be stored in immediately successive positions on a tape for the ABSOLUTE type record if it is not desired to read the data back slowly under program control. In this case, data can, and must be loaded back into RAM directly, at full speed.

4.0 POLYFORMAT TAPE LOADER FLOWCHART



APPENDIX G. GLOSSARY OF TERMS

awareness — the condition of knowing in a given program that a certain system variable can and should be changed to a certain value without dangerous effects. An assembler, for example, should change no system variables - it is completely "unaware". This way, an "aware" program, called a supervisor, can run the assembler with the system RAM configured any way desired. The assembler could be used to communicate with any physical device that the supervisor desires, merely by letting the supervisor put the address of the physical device's driver routine into the appropriate wormhole.

filter program — A program which can be installed in a wormhole by putting its address in that wormhole, and which performs some intermediate processing on I/O information before passing it on to its intended destination. An example is a pager for the listing generated by an assembler. The pager would be installed in a wormhole by a supervisor, and then the assembler would be called. The assembler would proceed unaware that the pager was counting lines and leaving spaces on each page for a page number that it generates also. The pager would pass the paged text listing on down to the physical device driver routine that was originally in the wormhole. Filter programs are also " unaware", as are user programs, since they do not know what physical device they are filtering. They depend on a supervisor to install them.

ISR or Interrupt Service Routine — A program which executes each time a specific interrupt of the 7 possible interrupts - occurs. It usually communicates with a physical device and moves data between the device and a buffer area in RAM which is shared by the ISR and the wormhole program which will transfer the data on to the user. The ISR also zeroes a flag byte, also in a shared (thus standardized) location. The wormhole program simply waits for this flag to be zeroed, and when it is, it knows that the data has been placed in the associated

CASSETTE TAPE FORMAT CRITERIA

Characteristics of the medium

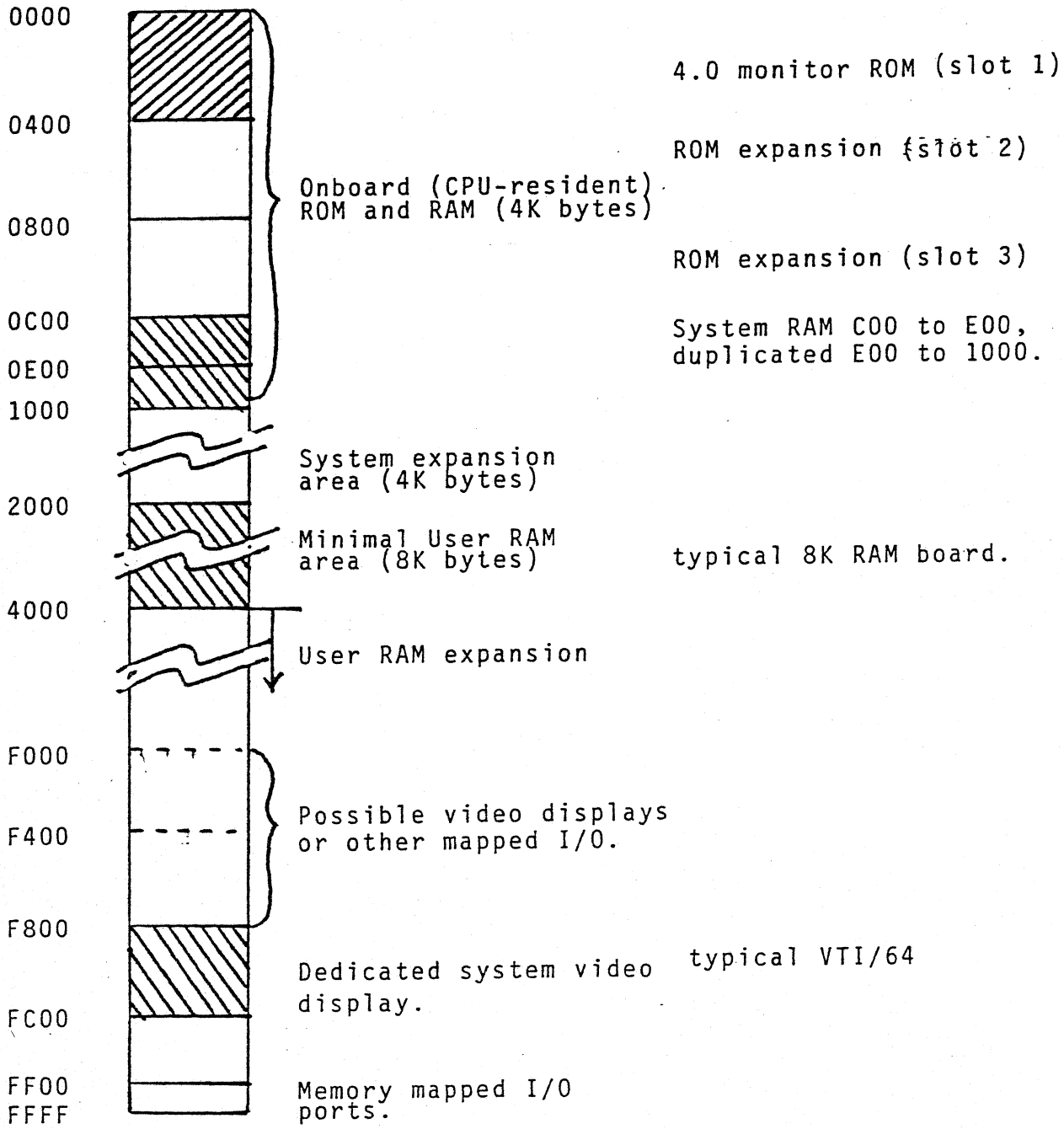
1. fixed speed of character transfer
2. operator cannot visually identify position of tape to find leaders or identify files
3. cassettes are long enough for many files
4. no position control is available other than start/stop without operator intervention. The operator, however does not know what is coming from tape, and cannot perform positioning functions accurately, so he is of limited value.

Format criteria

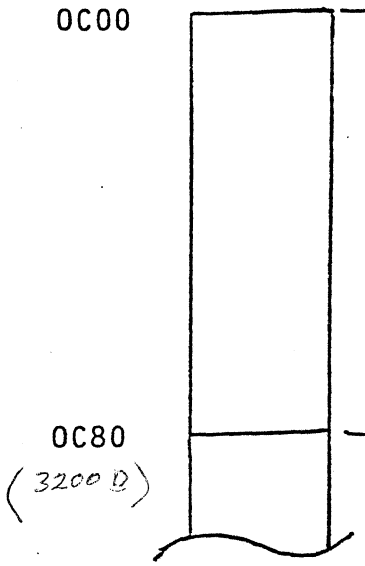
1. Transfer speed regulatable over long term for program controlled acceptance of data input or output.
2. Read error recovery--operator assistance acceptable on error detection, but it must be possible to skip already read data. This makes direct loading into memory for binary object code tapes faster since an error does not mean the entire tape must be reread. Also makes buffered transfers recoverable because previous data may not be available.
3. Files identifiable by the machine so that tape libraries are possible and files may be packed closely together.
4. Contents of tape should be made visible to operator so he can scan a tape and find a file or a blank space or so he will know what portion of a file has loaded etc...
5. Multiple file types should be avilable for special kinds of data such as absolute binary object code, which should be loaded directly into memory, documentation or "comment" files, which should be displayed for the operator, ASCII or binary data which is to be operated on by a program of some kind and therefore should be transferred through a buffer.
6. Synchronous or asynchronous byte format should be usable with no change in the file format
7. Any transmission speed should be usable
8. Format should work on a teletype or floppy disk as well as cassette tape.
9. High efficiency in information packing for fast data rate.

APPENDIX E. MEMORY MAP

Total address space allocation.

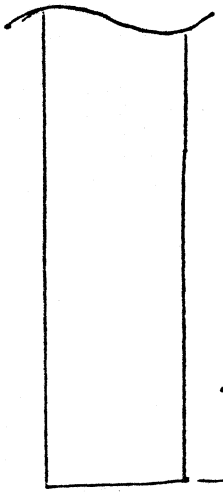


CPU resident system RAM.



System variable space. Defined for all systems as reserved for storage of interrupt, wormhole, and other shared information.

User program space (in systems with no other RAM). May be used by operating systems for I/O drivers, etc., in large systems.

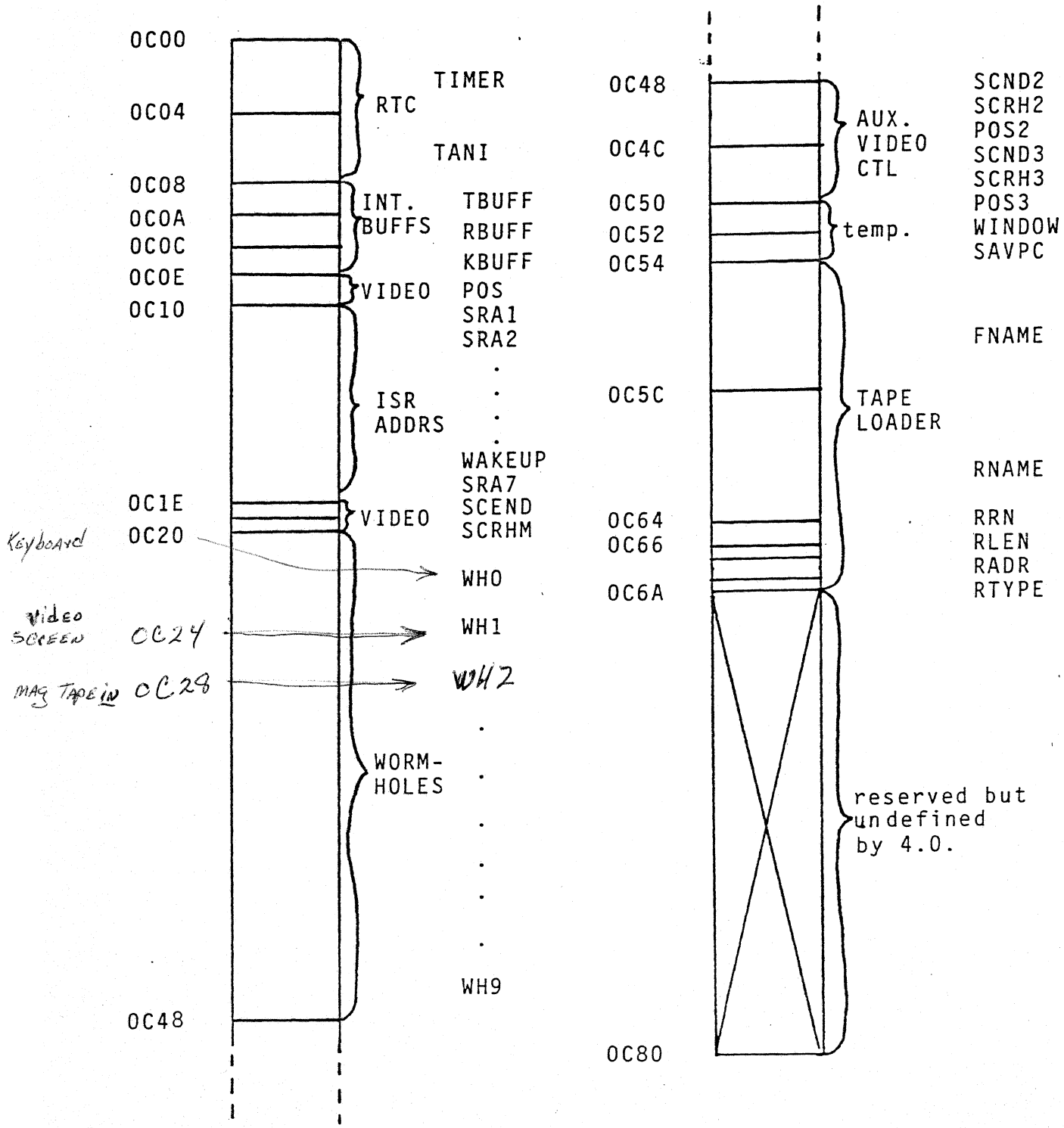


Meeting point between system programs and stack depends on size of both - either can expand until conflict occurs. Big systems usually move stack to a larger area to allow more system programs.

Initial system stack. May be moved by large systems which need system RAM for system functions.

OFFF
(4095 D)

System variables in System RAM.



buffer. When the wormhole program is done, it de-activates the flag by putting a non-zero value in it. Interrupts occur at random times, so they must save the contents of all CPU registers and restore them when done.

link program — A part of a supervisor program which connects a filter program to the physical devices that the filter program replaced. It is installed in the appropriate wormhole by the supervisor program before the user program is executed. When the user program calls the filtered wormhole, the link program is executed. The link program restores the wormhole to its original contents and calls the filter program. The filter program executes, transferring filtered data to and from the original source/sink. When it returns, the link program restores its own address into the wormhole and returns into the user program. When the user program is finished, it returns back into the supervisor program which restores the original contents of the wormhole which had the link program address. It is evident (with thought) that a supervisor program may act as a link program for a supervisor above it, and that a link program is a simple-minded supervisor program.

logical device — An imaginary I/O device which behaves in a known way for a program which communicates with it. For example, the unknown device which communicates with the computer operator is called the logical console device. It can be used by an program without regard to what physical device is actually being used for communications with the operator (video display, TTY, graphics CRT, etc....). This means that the logical devices must be standardized: methods of access must be uniform, and the conceptual function of each device must be uniform. Standardization is achieved in the POLY-88 by such means as the use of wormholes and the declaration of conventions for allocation of system variables in the CPU resident RAM memory.

physical device	An actual piece of hardware capable of performing input or output or both, via some standard method or format. It is used to describe the difference between the data sources/sinks that an executing program is "aware" of (see glossary definition of "awareness") and the data sources/sinks that are actually being used. Programs are normally concerned with "logical devices" (e.g. "logical tape reader" or "logical console key board"), whereas the system and the computer operator are concerned with the actual physical device that data is flowing to (e.g. audio cassette or paper tape reader).
supervisor program	Any program which changes the contents of any wormhole or other system variable in the CPU resident RAM is acting as a supervisor to all programs which use the system in the new context (see dictionary definition of context). The supervisor is responsible for saving the old contents of the system RAM that it changes, and for restoring it when its job is done. It executes a CALL to the "user" program it is supposed to run. When the user program is finished, it will return and the supervisor can restore the system RAM. Supervisor generally discover how to change the system RAM by talking to the operator through the system console.
system variables	Any of the shared storage areas in the CPU resident RAM memory from C00H to C80H. System variables are used to "configure" the system because they control the connections of physical to logical devices through the wormholes, the address of current system video display and system keyboard, the addresses of auxiliary video displays, the addresses of the interrupt service routines for each interrupt, and the status of the real time clock.
user program	A completely unaware program. It is designed to perform some processing function and so should be maximally flexible in its I/O. For this reason, it does not change any system variable, but assumes that any system variables that it references have been set up properly by a supervisor before hand. When it is done, it returns with a RET instruction and the supervisor that called it restores the system variables it changed or runs another user program.

wormholes — One of 10 areas in the CPU resident RAM memory that contain CALL followed by RET instructions. The address of the CALL is the address of a routine which can fetch or transmit a single byte from/to a physical device. Each wormhole is defined to be the access port for a given logical device such as the console input, console output or binary input or output etc.. The wormholes start at 0C20H and go up, each taking 4 bytes. Each can be called and will transmit/fetch the character in A. See Appendix H for details.

wormhole programs — A program called by a wormhole which is supposed to communicate with a logical device such as the logical console input device. In an interrupt driven system, it usually wastes time waiting for a flag in a known RAM location to indicate that data is available or can be transmitted to the physical device currently connected. When the flag is activated, the transfer takes place and the wormhole program returns to the user via the wormhole. The wormhole program thus guarantees that exactly one byte will be transmitted before it returns to the user. No registers other than the accumulator may be affected.

APPENDIX I

ASCII CONTROL CHARACTER USAGE

HEX	ASCII	CTL KEY	USE
ØØ	NUL	@	USED TO DETECT BREAK CONDITION
Ø1	SOH	A	
Ø2	STX	B	
Ø3	ETX	C	
Ø4	EOT	D	
Ø5	ENQ	E	
Ø6	ACK	F	
Ø7	BEL	G	
Ø8	BS	H	BACKSPACE
Ø9	HT	I	HORIZONTAL TAB
ØA	LF	J	LINE FEED (NOT RECOGNIZED BY CRT DRIV.)
ØB	VT	K	VERTICAL TAB - MOVES CURSOR TO HOME POS.
ØC	FF	L	FORM FEED - CLEAR SCREEN ON CRT
ØD	CR	M	CARRIAGE RETURN - NEWLINE ON CRT
ØE	SO	N	
ØF	SI	O	
1Ø	DLE	P	
11	DC1	Q	
12	DC2	R	
13	DC3	S	
14	DC4	T	
15	NAK	U	
16	SYN	V	SYNC CHARACTER - IGNORE EXCEPT AS BCC
17	ETB	W	
18	CAN	X	CANCEL LINE
19	EM	Y	KEYBOARD INTERRUPT
1A	SUB	Z	SAVE CPU STATE & GO TO MONITOR
1B	ESC	[	HALT EXECUTION & RETURN TO INTERPRETER
IC	FS	\	
ID	GS	]	
IE	RS	↑(^)	
IF	US	→(_)	
>F	DEL	RB	RUBOUT - DELETE LAST CHARACTER AND BACKSPACE

APPENDIX J. VECTOR INTERRUPT SYSTEM

Interrupts on 4.0 vector through the locations from zero through 38 hex as they do in any 8080 system, however these locations in the monitor ROM contain instructions which push all general registers and jump to an address stored in the corresponding element in the SRA table (Service Routine Address table). The SRA table is in System RAM, so can be changed by a user supervisor program which wants to install its own Interrupt Service Routine (ISR) into any of the vectors. The chart below shows the history of the vectoring for each of the monitor versions. 4.0 fixes permanently the address of the SRA table at C10. Note that there is no SRA6, but instead an address known as the "wakeup" address, which is the jump address for RTC timeout. The RTC acts like a piece of hardware wich increments the negative count in TIMER (four bytes at C00) until it reaches zero, when the routine at the WAKEUP address is jumped to.

VECTORED INTERRUPT	EQUIVALENT RESTART INSTRUCTION	INTERRUPT VECTOR ADDRESS ASSIGNED BY MONITOR VERSION		
		2.0	2.2	4.0
VI0 sing.step.	RST 7	N.A.	N.A.	C1C (SRA7)
VI1 RTC	RST 6	1030	0FFC	C1A (wakeup)
VI2 KBD	RST 5	N.A.	N.A.	C18 (SRA5)
VI3 USART	RST 4	1020	0FF9	C16 (SRA4)
VI4	RST 3	1018	0FF6	C14 (SRA3)
VI5	RST 2	1010	0FF3	C12 (SRA2)
VI6	RST 1	1008	0FF0	C10 (SRA1)
VI7 (same as RESET)	RST 0	Used to initialize the system		

The service routine should use the original flag byte and single byte buffer that KSR (Keyboard Service Routine) used for the VTI keyboard port. If it does not, the address in the console input wormhole should be changed to point to a routine which can communicate with the proper flag and buffer. The flag and buffer used by KSR is called KBUFF and is located at C0C, with the flag byte at that address, followed by the buffer. The flag indicates data is valid in the buffer when it is zero.

If it were desired to connect multiple keyboards to the system, this could be done either by using more interrupt vectors, assigning a separate service routine to each, or it could be done by doing "polled" I/O to determine which keyboard interrupted through a shared interrupt. Polling requires an interrupt service routine which can talk separately to each keyboard port to find the one that interrupted, and then service only that one. It puts the data obtained into a buffer corresponding to the interrupting keyboard and activates (zeroes) the corresponding flag. In a time sharing system, the operating system would swap addresses into and out of the console input wormhole (WHO) each time a user's program was restarted. Each user's wormhole would effectively contain a special routine which would communicate only with the buffer corresponding to his keyboard.

The monitor provides a routine which reads from the USART, and which uses another buffer of the same configuration as the keyboard buffer. It is called RBUFF and is located at C0A in system RAM. When a higher-level program installs routines which can both read and write through the USART, they should use the buffers and flags defined in system RAM, just as the keyboard service routine did with the keyboard buffer. This way, all wormhole programs which wait for the flags to go to

zero will work no matter what interrupt service routine actually changed the flag. The ISR may be handling more than one device on the particular interrupt that it is installed in, and may be placing the characters it obtains into a long buffer (longer than the single byte buffer used to talk to the wormhole programs), but in any case, if the same flag and byte-buffer are used, no incompatibilities between ISR and wormhole program will result.

The key board flag is particularly important, since many large applications programs may test it to determine whether a key has been pressed. If different ISR's use different flags, then these applications programs will not work.

On the subject of standards, the interrupts service routines all use a standardized register save sequence, which consists of pushing PSW (accum and flags), then B, D, and H. All interrupts are forced to use the sequence because it is done automatically in the ROM before the ISR is jumped to. For the convenience of the ISRs, a section of code in the monitor called IORET can be used to restore all the registers in the same sequence, and to enable interrupts before returning to the interrupting program. A simple jump into the start of IORET will do the rest. In the monitor, the USART ISR, called USRTSR falls through into IORET to avoid the need to jump.

There are only seven vector interrupts because the location that V17 would interrupt through is the same address as RESET uses, specifically, address 0000. V17 would have the same effect as resetting the CPU, which of course initializes the system and brings up the tape loader mode.

Several of the other VI's have been dedicated also. They are allocated by 4.0 for the single step interrupt, the system keyboard and the 8251 USART (in addition to the RTC interrupt mentioned above).

The single step interrupt is generated by a few flip-flops on the CPU which count two instructions after they are activated and then interrupt. This allows the single stepping feature on the POLY-88 which is so useful in program debugging. The monitor pops the values of the registers from their save area on the system stack, leaving the address to be stepped, activates the single-step hardware, and does a return. The RET instruction is counted by the hardware as one instruction, then the single instruction in the user program that gets executed is counted as another before the interrupt is generated. The interrupt is vectored to SAVE - an address in the monitor - by SRA7, which is initialized to contain this address. SAVE then pushes all the registers back down the stack and returns to the front panel mode for further operator commands. 4.0 leaves the vector address of VIO changeable through SRA7 in order to use the single-stepping feature in more advanced program debugging systems.

The sytem keyboard interrupt is initially set up for a VTI keyboard port, but by changing the address in SRA5, any other interrupt serviced physical device service routine may be installed.

```
;
;FINAL VERSION 4.Ø MONITOR NOV 22,1976
;
;COPYRIGHT 1976 BY
;POLYMORPHIC SYSTEMS
;A DIVISION OF
;INTERACTIVE PRODUCTS CORPORATION
;737 S. KELLOGG AVE.
;GOLETA, CA  93Ø17
;
;********** 4.Ø MONITOR **********
;
;WRITTEN BY              D.L.FAIMAN
;                        D.W.SALLUME
;                        R.L.DERAN
;
;POLY-88 RESIDENT MONITOR ROM VERSION 4.Ø.  RUNS FROM ØØ
;TO 3FFH IN FIRST CPU ROM SOCKET.  FRONT PANEL RESET OR
;THRU ZERO GETS A "POLY-FORMAT" ABSOLUTE TAPE LOADER
;WHICH WILL RUN THE CPU RESIDENT USART TO READ BYTE-STAN
;('B') OR "POLY-PHASE" ('P') AUDIO CASSETTE TAPES.
;CONTROL-Z AT ANY TIME ON SYSTEM KBD BRINGS UP FRONT PAN
;DISPLAY WITH MEMORY MODIFY WINDOW IN HEX.  COMMANDS THE
;ALLOW MEMORY,REGISTER MODIFICATION AND SINGLE-STEP/EXEC
;OF INTERRUPTED PGM.
;
;UTILITY PGMS AVAILABLE FOR USER ARE:
;DSPLY  PUTS CHAR ON SYSTEM VIDEO SCRN. RECOGNIZES
;       ALL STANDARD CTL CHARS AND SCROLLS TEXT UP.
;       ACCESSIBLE BY CALL TO WORMHOLE 1 (WH1) AT ØC24H.
;KI     GETS CHAR FROM SYSTEM CONSOLE KBD.
;       ACCESSIBLE BY CALL TO WORMHOLE ZERO (WHØ) AT ØC2
;USRTI  GETS CHAR FROM USART BACKGROUND LOAD
;       TYPE ROUTINE.
;HEXO   PUTS OUT LOW NIBBLE OF A IN HEX THRU WH1
;BYTE   PUTS OUT A IN HEX THRU WH1.
;DEOUT  PUTS OUT D,E IN HEX THRU WH1.
;HEXC   GETS A HEX NUMBER UP TO 2 BYTES LONG IN H,L FROM
;       WHØ, ECHOING ON WH1.  NON-HEX CHARACTER
;       TERMINATES INPUT. THE TERM. CHAR IS LEFT IN A.
;MOVE   TRANSFERS #BYTES IN BC FROM ADDR IN HL TO ADDR I
;       TERMINATES WITH HL=HL+BC,DE=DE+BC,BC=Ø.
;IORET  A SECTION OF CODE THAT TERMINATES AN
;       INTERRUPT SERVICE ROUTINE BY RESTORING
;       ALL REGS IN STANDARD ORDER, EI, RET.
;TIME   EXECUTES 6Ø TIMES A SECOND AUTOMATICALLY, INCREM
;       ING FOUR BYTE LOCATION CALLED TIMER.  IF TIMER=Ø
;       ALL REGS SAVED AND ROUTINE AT ADDR IN LOCATION
;       WAKEUP IS JUMPED TO.  THIS ROUTINE MAY TERMINATE
```

```
                ;       BY USING IORET.
                ;
                ;******SYSTEM RAM ALLOCATION******
                ;
                ;DEDICATED LOCATIONS.
                ;
0C00                    ORG     0C00H   ;BEGIN OF ONBD RAM
0C00            TIMER:  DS      4       ;INCREMENTED BY 60HZ CLOCK.
                                        ;LOW BYTE=LEAST SIGNIFICANT.
0C04            TANI:   DS      4       ;TIME AT NEXT INT. USED BY
0C08            TBUFF:  DS      2       ;USART TRANSMIT BUFFER
                                        ;FLAG AND ONE BYTE BUFFER.
                                        ;USED BY EXTERNAL DUMPERS.
0C0A            RBUFF:  DS      2       ;USART RECEIVE BUFFER FLAG
                                        ;AND ONE BYTE BUFFER.
                                        ;USED BY MONITOR TAPE LOADER.
0C0C            KBUFF:  DS      2       ;KBD BUFF-FULL FLAG AND BUFFER
0C0E            POS:    DS      2       ;CURSOR POSITION FOR DSPLY
                ;INTERRUPT SERVICE ROUTINE ADDRESS TABLE.
0C10            SRA1:   DS      2       ;VI7
0C12            SRA2:   DS      2       ;VI6
0C14            SRA3:   DS      2       ;VI5
0C16            SRA4:   DS      2       ;VI4: USART INT.,FIRST INITED SR
0C18            SRA5:   DS      2       ;VI3: KBD INT.
0C1A            WAKEUP: DS      2       ;ADDRESS JUMPED TO WHEN CLOCK TIME
                                        ;OUT IS IN THIS LOC. IT FUNCTION
                                        ;EXACTLY LIKE AN SRA, BUT IS ACT
                                        ;JUMPED TO BY THE CLOCK SOFTWARE
                                        ;EFFECTIVELY SIMULATES A HARDWAR
0C1C            SRA7:   DS      2       ;SINGLE-STEP INTERRUPT. NORMALLY
                                        ;AT ITS INITED ADDR WHICH BRINGS
                                        ;UP FRONT PANEL DISPLAY.  IT CAN
                                        ;CHANGED FOR EXTRNAL PGMS TO USE
                                        ;DEBUGGING OR FANCY PGMING.
                ;VIDEO SCREEN ADDRESS PARAMETERS USED IN DSPLY.
                ;INITED TO F800H,FCH, BUT MAY BE CHANGED AFTER
                ;SYSTEM START UP.
0C1E            SCEND:  DS      1       ;SCREEN END FOR DSPLY.
0C1F            SCRHM:  DS      1       ;SCREEN HOME FOR DISPLAY
                ;WORMHOLE VECTOR. THIS TABLE IS FULL OF CALL-RETURN PAIR
                ;WHICH CALL SYSTEM I/O PROGRAMS. THEY MAY BE CHANGED AFT
                ;INITIALIZATION TO CALL ANY PHYSICAL DEVICE DRIVER IT IS
                ;DESIRED TO INSTALL.  THIS IS NORMALLY DONE WITH LINK PR
                ;AS DESCRIBED IN MANUALS.  THE FIRST TWO WORMHOLES ARE I
                ;TO THE SYSTEM CONSOLE KBD AND SYSTEM VIDEO DISPLAY.
                ;ALL USER I/O SHOULD BE DONE THRU THE WH'S TO INSURE
                ;COMPATIBILITY WITH ANY SUPERVISOR SYSTEM AND TO ALLOW
                ;DYNAMIC REASSIGNMENT OF I/O FOR A FIXED USER OBJECT PGM
0C20            WH0:    DS      4       ;CONSOLE IN: INITED TO CALL SYS.
0C24            WH1:    DS      4       ;CONSOLE OUT: INITED TO CALL DSP
0C28            WH2:    DS      4       ;SYSIN: USED EXTRNLY BY TAPE OR
                                        ;INPUT DRIVERS. A CALL GETS BYTE
FFEA            INITLEN EQU     SRA4-$  ;LENGTH OF INITILIZED MEMORY
0C2C            WH3:    DS      4       ;SYSOT: USED BY TAPE OR DISK OUT
```

```
                                        ;DRIVERS. A CALL OUTPUTS BYTE IN
0C30            WH4:    DS      4       ;SYSIN2: SECONDARY SYSTEM INPUT.
0C34            WH5:    DS      4       ;SYSOT2: SECONDARY SYSTEM OUTPUT
0C38            WH6:    DS      4       ;READ: TEXT TYPE INPUT USED
                                        ;BY EXTRNL TAPE,DISK,KBD DRIVERS
0C3C            WH7:    DS      4       ;LIST: TEXT TYPE OUTPUT TO LINE
                                        ;PRINTERS OR TEXT FILES ETC.
0C40            WH8:    DS      4       ;AUX WH: IN
0C44            WH9:    DS      4       ;AUX WH: OUT
                ;VCB BLOCKS TO BE TEMPORARILY SWAPPED FOR SCRHM,SCEND AN
                ;IN ORDER TO USE DSPLY ON OTHER THAN SYSTEM VIDEO DISPLA
                ;A LINK PGM DOES SWAPPING, AND IS CALLED FROM WH1 IN-
                ;STEAD OF DSPLY. IT THEN CALLS DSPLY, AND RESTORES
                ;CONTENTS OF SCRHM,SCEND AND POS BEFORE RET..
                VCB2:                   ;SECONDARY VDO SCRN CONTEXT
0C48            SCND2:  DS      1
0C49            SCRH2:  DS      1
0C4A            POS2:   DS      2
                VCB3:                   ;TERTIARY VDO SCRN CTX.
0C4C            SCND3:  DS      1
0C4D            SCRH3:  DS      1
0C4E            POS3:   DS      2
                ;
                ;TEMPORARY LOCATIONS USED BY FRONT PANEL MODE.
                ;
0C50            WINDOW: DS      2       ;MEM. MODIFY WINDOW POINTER.
0C52            SAVPC:  DS      2       ;USED BY S.S.WHEN STACK UNAVAILA
                ;
                ;TEMPORARY LOCATIONS USED BY TAPE LOADER.
                ;
0C54            FNAME:  DS      8       ;"FIND" NAME FOR TAPE LOADER.
0C5C            RNAME:  DS      8       ;READ RECD NAME -FOUND ON TAPE.
0C64            RRN:    DS      2       ;READ RECD NUMBER.
0C66            RLEN:   DS      1       ;READ RECD LENGTH.
0C67            RADR:   DS      2       ;READ RECD ADDR (BIAS).
0C69            RTYPE:  DS      1       ;READ RECD TYPE.
                ;FREE ONBOARD RAM ONWARDS.
                ;
                ;***MONITOR PROGRAM***
                ;
                ;VECTOR INTERRUPT LOCATIONS
0000                    ORG     0
0000 310010     RESET:  LXI     SP,STACK ;FRNT PANEL RST OR POWER UP GET
0003 C30002             JMP     RST1    ;RST1 INITS SYST RAM,ENDS IN TAP
                ;
                ;THESE TWO INSTRUCTIONS CAN BE CALLED TO GET THE
                ;ADDDRESS OF THE CALLING PROGRAM INTO H,L.  THIS
                ;IS NECESSARY IN WRITING SELF RELOCATING CODE.
                ;
0006 E1                 POP     H
0007 E9                 PCHL
0008 F5         VI6:    PUSH    PSW     ;STANDARD REGISTER PUSH SEQUENCE
0009 C5                 PUSH    B
000A D5                 PUSH    D
```

```
000B E5              PUSH    H
000C 2A100C          LHLD    SRA1     ;GET SERVICE ROUTINE ADDRESS
000F E9              PCHL             ;GO EXECUTE IT. IT WILL RTRN THR
0010 F5       VI5:   PUSH    PSW      ;SAME AS ABOVE
0011 C5              PUSH    B
0012 D5              PUSH    D
0013 E5              PUSH    H
0014 2A120C          LHLD    SRA2
0017 E9              PCHL
0018 F5       VI4:   PUSH    PSW
0019 C5              PUSH    B
001A D5              PUSH    D
001B E5              PUSH    H
001C 2A140C          LHLD    SRA3
001F E9              PCHL
0020 F5       VI3:   PUSH    PSW
0021 C5              PUSH    B
0022 D5              PUSH    D
0023 E5              PUSH    H
0024 2A160C          LHLD    SRA4
0027 E9              PCHL
0028 F5       VI2:   PUSH    PSW
0029 C5              PUSH    B
002A D5              PUSH    D
002B E5              PUSH    H
002C 2A180C          LHLD    SRA5
002F E9              PCHL
0030 F5       CLOCK: PUSH    PSW      ;THE CLOCK INT ALWAYS GOES TO TH
0031 C5              PUSH    B        ;TIMER COUNTER ROUTINE.
0032 D5              PUSH    D
0033 E5              PUSH    H
0034 C34000          JMP     TIME
0037 00              DB      0
0038 F5       SS:    PUSH    PSW
0039 C5              PUSH    B
003A D5              PUSH    D
003B E5              PUSH    H
003C 2A1C0C          LHLD    SRA7
003F E9              PCHL
0040 D308     TIME:  OUT     8        ;ENABLE INT FOR NEXT 60HZ CYCLE
0042 21000C          LXI     H,TIMER  ;COUNTER LOCATION
0045 3E04            MVI     A,4      ;4 BYTES TO INCR
0047 34       TIME2: INR     M        ;INC ONE LOC.
0048 C26400          JNZ     IORET    ;DONE,STANDARD RETURN
004B 23              INX     H
004C 3D              DCR     A
004D C24700          JNZ     TIME2
0050 2A1A0C          LHLD    WAKEUP   ;CNTR IS ZERO,SO INITIATE  WAKEU
0053 E9              PCHL             ;GO TO THE WAKEUP TASK
                ;USART INTERUPT SERVICE ROUTINE FOR INPUT
                ;ITS ADDRESS IS INITED INTO SRA4.
0054 DB01     USRTSR: IN     1
0056 E602            ANI     2
0058 CA6400          JZ      IORET
```

```
005B DB00           IN      0
005D 210A0C         LXI     H,RBUFF
                    ;
                    ;THIS CODE IS SHARED BY ANY SERVICE ROUTINE
                    ;WHICH HAS FOUND A CHARACTER.  HL SHOULD HAVE THE
                    ;BUFFER ADDRESS, FLAG FIRST-THEN ONE BYTE BUFF.
                    ;ONLY USED BY INPUT ISR'S.
                    ;
0060 3600     IOPUT:MVI     M,0     ;ZERO THE FLAG: WE GOT THE CHAR
0062 23             INX     H       ;MOVE UP TO DATA BUFFER
0063 77             MOV     M,A     ;PUT CHAR IN DATA BUFF
                                    ;FALL THRU TO IORET
0064 E1       IORET:POP     H       ;THIS CODE SECTION CAN BE JUMPED
                                    ;FOR STANDARD SEQ.REG.POPS AND E
                                    ;AS FOR ANY INTERRUPT SERVICE RT
0065 D1             POP     D
0066 C1             POP     B
0067 F1             POP     PSW
0068 FB             EI
0069 C9             RET
                    ;WORMHOLE END OF KBD CHAR FETCH RTNS.
                    ;WHEN CALLED BY WH0,WAITS FOR VALID DATA FLG IN KBUFF,
                    ;THEN RETURNS CHAR FROM KBUFF+1 IN A.
006A E5       KI:   PUSH    H       ;WE CAN ONLY WRECK A, SO SAVE H
006B 210C0C         LXI     H,KBUFF
006E 7E       KI1:  MOV     A,M     ;GET FLAG
006F B7             ORA     A       ;IS IT ZERO?
0070 C26E00         JNZ     KI1     ;NO,TRY AGAIN
0073 35             DCR     M
0074 23             INX     H
0075 7E             MOV     A,M
0076 E1             POP     H
0077 C9             RET
0078 E5       USRTI:PUSH    H
0079 210A0C         LXI     H,RBUFF
007C C36E00         JMP     KI1
                    ;
                    ;****** VIDEO DISPLAY DRIVER ******
                    ;
                    ;DSPLY IS THE FAMOUS "TELETYPE SIMULATOR" WHICH
                    ;DRIVES A POLYMORPHICS VTI AND SCROLLS TEXT WHEN
                    ;THE SCRN FILLS.  IT RECOGNIZES ALL IMPORTANT CONTROL
                    ;CHARACTERS AND USES THE SCRHM,SCEND, AND POS LOCATIONS
                    ;IN SYSTEM RAM WHICH ARE INITIALIZED TO GIVE A SCREEN
                    ;AT 0F800H TO 0FBFFH ON POWER-UP OR FPRST.  AT
                    ;THE SAME TIME, ITS OWN ADDRESS IS PUT IN WH1 FOR THE
                    ;DEFAULT SYSTEM CONSOLE DISPLAY DRIVER.
                    ;
007F F5       DSPLY:PUSH    PSW     ;A WH PGM MUST SAVE ALL REGS
0080 C5             PUSH    B
0081 D5             PUSH    D
0082 E5             PUSH    H
0083 2A0E0C         LHLD    POS
0018          CTLX  EQU     18H     ;ASCII "CAN" CLEARS CUR. LINE
```

```
0086 FE18            CPI     CTLX
0088 CAD000          JZ      CLINE
008B 11B900          LXI     D,SCRL  ;ALL DSPLY CHARACTER
008E D5              PUSH    D       ;HANDLING SUBROUTINES EXCEPT
                                     ;CLINE RETURN TO SCRL TO CHECK
                                     ;SCRN OVERFLOW AND SCROLL
                                     ;IF NECESSARY.
008F 367F            MVI     M,07FH  ;BLANK THE CURSOR SO WE
                                     ;WON'T HAVE TO LATER.
                                     ;USE A GRAPHICS BLANK
                                     ;INSTEAD OF A SPACE.
007F         RB      EQU     7FH     ;ASCII "DEL" OR RUBOUT
                                     ;BACKSPACE AND DELETE CHAR.
0091 FE7F            CPI     RB
0093 CAE400          JZ      RBR
0096 FE20            CPI     20H     ;IF CHAR IS ABOVE 20H, PRINT IT
0098 D2B400          JNC     NORM    ;ELSE IT'S NOT A NORMAL CHAR,
                                     ;SO TEST IT FOR VALID CTL CODE
000D         CR      EQU     0DH     ;CARRIAGE RETURN. MOVES DOWN
                                     ;A LINE, LEFT-ZEROES CURSOR.
009B D60D            SUI     CR
009D CAE600          JZ      CRR
000C         FF      EQU     0CH     ;FORM FEED. CLEARS SCRN, HOMES
                                     ;CURSOR.
00A0 3C              INR     A
00A1 CAEF00          JZ      FFR
000B         VT      EQU     0BH     ;VERTICAL TAB. JUST HOMES CURSOR
00A4 3C              INR     A
00A5 CAF900          JZ      VTR
0009         TAB     EQU     09H     ;TAB. MOVES CURSOR RIGHT TO
                                     ;NEXT EVEN/8 POSITION.
00A8 3C              INR     A
00A9 3C              INR     A
00AA C0              RNZ             ;IF NOT A TAB, RETURN
00AB 7D              MOV     A,L     ;BACK UP CURSOR TO EVEN/8
00AC E6F8            ANI     0F8H
00AE 6F              MOV     L,A
00AF 010800          LXI     B,8     ;MOV UP 8 POSITIONS
00B2 09              DAD     B
00B3 C9              RET
00B4 F680    NORM:   ORI     80H     ;WE HAVE A PRINTING CHAR, SO MAP
                                     ;IT INTO CHAR AREA OF VTI SPACE.
00B6 77              MOV     M,A     ;PUT IT IN REFRESH MEM.
00B7 23              INX     H       ;MOVE UP A POSITION.
00B8 C9              RET             ;GO TO SCRL.
                ;
                ;SCRL SCROLLS TEXT UP THE SCRN IF NECESSARY,
                ;THEN FALLS INTO CURP, WHICH RESTORES THE CURSOR
                ;AND RETURNS TO USER THRU IORET.
                ;
00B9 3A1E0C  SCRL:   LDA     SCEND
00BC BC              CMP     H
00BD C2DC00          JNZ     CURP
00C0 2A1E0C          LHLD    SCEND
```

```
00C3 7C              MOV     A,H
00C4 95              SUB     L
00C5 54              MOV     D,H
00C6 2E40            MVI     L,40H
00C8 1E00            MVI     E,0
00CA 4D              MOV     C,L
00CB 47              MOV     B,A
00CC CD0001          CALL    MOVE
00CF 2B              DCX     H
                ;
                ;CLINE CLEARS THE CURRENT LINE
                ;
00D0 3E3F       CLINE:  MVI     A,3FH
00D2 57              MOV     D,A
00D3 B5              ORA     L
00D4 6F              MOV     L,A
00D5 367F       WIPE:   MVI     M,7FH
00D7 2B              DCX     H
00D8 15              DCR     D
00D9 C2D500          JNZ     WIPE
00DC 36FF       CURP:   MVI     M,0FFH
00DE 220E0C          SHLD    POS
00E1 C36400          JMP     IORET
00E4 2B         RBR:    DCX     H       ;RUBOUT ROUTINE
00E5 C9              RET
00E6 014000     CRR:    LXI     B,64    ;CARRIAGE RETURN RTN.
00E9 7D              MOV     A,L
00EA E6C0            ANI     0C0H
00EC 6F              MOV     L,A
00ED 09              DAD     B
00EE C9              RET
00EF CDF900     FFR:    CALL    VTR     ;FORM FEED ROUTINE
00F2 367F       FF1:    MVI     M,7FH
00F4 23              INX     H
00F5 BC              CMP     H
00F6 C2F200          JNZ     FF1
00F9 2A1E0C     VTR:    LHLD    SCEND   ;VERTICAL TAB RTN.
00FC 7D              MOV     A,L
00FD 2E00            MVI     L,0
00FF C9              RET
                ;
                ;MOVE MOVES -BC BYTES FROM THE AREA STARTING AT
                ;(HL) TO THE AREA STARTING AT (DE)
                ;
0100 7E         MOVE:   MOV     A,M
0101 12              STAX    D
0102 13              INX     D
0103 23              INX     H
0104 0C              INR     C
0105 C20001          JNZ     MOVE
0108 04              INR     B
0109 C20001          JNZ     MOVE
010C C9              RET
                ;
```

```
                        ;KEYBOARD INTERUPT SERVICE ROUTINE FOR SYSTEM CONSLE
                        ;KEYBOARD AT 0F8H.  ITS ADDRESS  IS INITED INTO SRA5.
                        ;IT WATCHES FOR CTL/Z.  IF IT FINDS ONE, IT FALLS
                        ;THRU INTO SAVE, THUS ENTERING FRONT PANEL MODE.
                        ;KBD ADDR IS FIXED BY SCREEN ADDR, WHICH IS 0F800H
                        ;SINCE A POLYMORPHIC VTI USES SAME DECODER FOR KBD
                        ;AS FOR VIDEO REFRESH MEMORY.
                        ;
  010D DBF8             KSR:    IN      0F8H
  010F 210C0C                   LXI     H,KBUFF
  001A                  CTLZ    EQU     01AH
  0112 FE1A                     CPI     CTLZ
  0114 C26000                   JNZ     IOPUT
                        ;
                        ;****** FRONT PANEL MODE ******
                        ;
                        ;SAVE IS THE ENTRY POINT INTO FRONT PANEL
                        ;MODE IF REGISTERS HAVE BEEN ALREADY PUSHED IN
                        ;STANDARD SEQUENCE.  IT PUSHES PC AND SP
                        ;ON TOP OF REGISTERS FOR DISP TO USE.
                        ;
  0117 210A00           SAVE:   LXI     H,10
  011A 39                       DAD     SP
  011B E5                       PUSH    H
  011C 2B                       DCX     H
  011D 56                       MOV     D,M
  011E 2B                       DCX     H
  011F 5E                       MOV     E,M
  0120 D5                       PUSH    D
                        ;
                        ;WARM IS THE ENTRY POINT TO FRONT PANEL MODE
                        ;IF PC AND SP HAVE ALREADY BEEN PUSHED DOWN
                        ;ON TOP OF REGISTERS IN STANDARD SEQUENCE
                        ;
  0121 CD9203           WARM:   CALL    CLEAR
  0124 FB                       EI
  0125 CDF900                   CALL    VTR       ;SET HL TO BEGINNING OF SCREEN
  0128 015101                   LXI     B,337     ;OFFSET FROM SCRHM FOR UPARROW
  012B 09                       DAD     B
  012C 369C                     MVI     M,9CH     ;VTI CODE FOR UPARROW
  012E 017501                   LXI     B,175H    ;OFFSET FROM UPARROW FOR
                                                  ;RIGHT ARROW
  0131 09                       DAD     B
  0132 369A                     MVI     M,9AH     ;VTI CODE FOR RIGHT ARROW
                        ;
                        ;DISP IS THE ENTRY POINT TO FRONT PANEL MODE IF
                        ;THE SCREEN ALREADY SHOWS A FRONT PANEL DISPLAY.
                        ;IT DOES NOT CLEAR THE SCREEN, SO WILL NOT
                        ;BLINK WHEN SCREEN IS UPDATED.
                        ;
  0134 3E18             DISP:   MVI     A,CTLX    ;ERASE LAST COMMAND ON SCRN
  0136 CD7F00                   CALL    DSPLY
  0139 3E0B                     MVI     A,0BH
  013B CD7F00                   CALL    DSPLY
```

```
013E 3E06             MVI     A,6
0140 210000           LXI     H,0
0143 39               DAD     SP
0144 016D01           LXI     B,MSG
0147 F5               PUSH    PSW
0148 37       DISP1:  STC
0149 CD7C01           CALL    FLDSY
014C CD7C01           CALL    FLDSY
014F CD9703           CALL    BLK
0152 5E               MOV     E,M
0153 23               INX     H
0154 56               MOV     D,M
0155 23               INX     H
0156 CDD103           CALL    DEOUT
0159 F1               POP     PSW
015A 3D               DCR     A
015B CA8701           JZ      FLAGS
015E F5               PUSH    PSW
015F C5               PUSH    B
0160 EB               XCHG
0161 01FDFF           LXI     B,-3
0164 09               DAD     B
0165 EB               XCHG
0166 CD7F03           CALL    HEXO8
0169 C1               POP     B
016A C34801           JMP     DISP1
016D 50435350 MSG:    DB      'PCSPHLDEBCAFCMZ'
0171 484C4445
0175 42434146
0179 434D5A
                      ;
                      ;FIELD DISPLAY PUTS OUT CHAR IN ADDR IN B IFF CY SET.
                      ;IFF CY ZERO, PUT BLANK TO VDO DISPLAY.
                      ;IN EITHER CASE, B IS INCREMENTED.
                      ;
017C F5       FLDSY:  PUSH    PSW
017D D49703           CNC     BLK
0180 0A               LDAX    B
0181 DC7F00           CC      DSPLY
0184 F1               POP     PSW
0185 03               INX     B
0186 C9               RET
0187 CD9C03   FLAGS:  CALL    TABBER
018A 7B               MOV     A,E
018B 0F               RRC
018C CD7C01           CALL    FLDSY
018F 07               RLC
0190 07               RLC
0191 CD7C01           CALL    FLDSY
0194 07               RLC
0195 CD7C01           CALL    FLDSY
                      ;
                      ;MMOD PLACES THE MEMORY MODIFY DISPLAY
                      ;ON THE SCREEN   WINDOW POINT TO THE BYTE
```

```
                        ;TO BE MODIFIED
                        ;
0198 CD8D03     MMOD:   CALL    CROUT
019B CD8D03             CALL    CROUT
019E 2A500C             LHLD    WINDOW
01A1 01E0FF             LXI     B,-32
01A4 09                 DAD     B
01A5 0E08               MVI     C,8
01A7 EB                 XCHG
01A8 CDD103     MMOD1:  CALL    DEOUT
01AB CD9C03             CALL    TABBER
01AE 3E7F               MVI     A,7FH
01B0 CD7F00             CALL    DSPLY
01B3 CD7F03             CALL    HEXO8
01B6 0D                 DCR     C
01B7 C2A801             JNZ     MMOD1
                        ;
                        ;COMD CALLS HEXC TO GET HEX NUMBERS TO BE ENTERED INTO M
                        ;THEN EVALUATES THE NON-HEX TERMINATING CHARACTER TO SEE
                        ;IT IS A COMMAND
                        ;
01BA CDAA03     COMD:   CALL    HEXC
01BD 79                 MOV     A,C
01BE B7                 ORA     A
01BF C45302             CNZ     STORE
01C2 78                 MOV     A,B
01C3 213401             LXI     H,DISP
01C6 E5                 PUSH    H
01C7 21E801             LXI     H,MTBL
01CA CDDB01             CALL    LOOKUP
01CD 1600               MVI     D,0
01CF D23E02             JNC     ADDIT
01D2 CDDB01             CALL    LOOKUP
01D5 D8                 RC
01D6 210002             LXI     H,RTN
01D9 19                 DAD     D
01DA E9                 PCHL
                        ;
                        ;LOOKUP COMPARES THE ACCUMULATOR
                        ;AGAINST THE ENTRY IN A TABLE
                        ;POINTED TO BY HL AND RETURNS WITH
                        ;THE BYTE FOLLOWING IT IN E.
                        ;CARRY FLAG SET IF NO MATCH
                        ;TABLE MUST END IN 0FFH, EACH ENTRY IS 2 BYTES
                        ;
01DB BE         LOOKUP: CMP     M
01DC 23                 INX     H
01DD 5E                 MOV     E,M
01DE C8                 RZ
01DF 23                 INX     H
01E0 5E                 MOV     E,M
01E1 1C                 INR     E
01E2 C2DB01             JNZ     LOOKUP
01E5 23                 INX     H
```

```
01E6 37                 STC
01E7 C9                 RET
                ;
                ;MTBL IS THE TABLE OF COMMANDS FOR
                ;MEMORY MODIFY
                ;
01E8 20         MTBL:   DB      ' '     ;SPACE MOVE POINTER FORWARD
01E9 11                 DB      17
01EA 08                 DB      8       ;BACKSPACE MOVES BACKWARDS ONE
01EB 0F                 DB      15
01EC 0D                 DB      13      ;RETURN MOVES FWD 8 BYTES
01ED 18                 DB      24
01EE 0A                 DB      10      ;LINE FEED MOVES BACK 8 BYTES
01EF 08                 DB      8
01F0 FF                 DB      -1      ;END OF 1ST HALF TABLE
01F1 47                 DB      'G'     ;GO
01F2 6C                 DB      G-RTN
01F3 53                 DB      'S'     ;SET REGISTER
01F4 7C                 DB      SETR-RTN
01F5 58                 DB      'X'     ;EXECUTE (SINGLE STEP)
01F6 59                 DB      X-RTN
01F7 49                 DB      'I'     ;INDIRECT
01F8 49                 DB      IND-RTN
01F9 54                 DB      'T'     ;TAPE LOADER
01FA 14                 DB      TAPE-RTN
01FB 4C                 DB      'L'     ;LOAD MEMORY POINTER
01FC 37                 DB      LOD-RTN
01FD 4A                 DB      'J'     ;JUMBO - LOAD 2 BYTES
01FE 2C                 DB      DOUBLE-RTN
01FF FF                 DB      -1
                ;
                ;RST1 IS THE INITIALIZATION ROUTINE
                ;IT SETS UP THE WORMHOLES USED IN THE
                ;MONITOR, THEN CHECKS FOR A SECOND ROM
                ;AND CALLS IT IF IT IS THERE
                ;
0200            RTN     EQU     $       ;ALL ROUTINES REFERENCE TO HERE
0200 11160C     RST1:   LXI     D,SRA4
0203 21EA03             LXI     H,INITER
0206 01EAFF             LXI     B,INITLEN
0209 CD0001             CALL    MOVE    ;INIT WORMHOLES
020C 3A0004             LDA     400H    ;GET 1ST BYTE OF 2ND ROM
020F 3C                 INR     A       ;IF IT WAS NOT 0FFH
0210 C40004             CNZ     400H    ;CALL IT
0213 FB                 EI              ;TURN ON INTERRUPTS
                ;
                ;TAPE IS THE BOOTSTRAP LOADER ROUTINE
                ;IT EXPECTS TO GET A B,P, OR C OR IT WILL
                ;WAIT FOR ONE
                ;
0214 CD9203     TAPE:   CALL    CLEAR
0217 CDA103     START:  CALL    LCFLD   ;GET A CASE-FOLDED B OR P
021A 4F                 MOV     C,A     ;SAVE THE B OR P FOR ECHOING.
021B FE50               CPI     'P'     ;POLYPHASE
```

```
021D CABA02             JZ      POLY
0220 D642               SUI     'B'     ;BYTE STANDARD
0222 CAC702             JZ      BITE
0225 3D                 DCR     A
0226 CAF302             JZ      HEAD    ;CONTINUE
0229 C31702             JMP     START   ;NOT A B OR C OR P : TRY AGAIN
                ;
                ;
                ;
                ;DOUBLE GETS A HEX NUMBER FROM THE CONSOLE
                ;AND LOADS IT INTO THE NEXT 2 BYTES IN MEMORY
                ;(OR A REGISTER) LOW ORDER BYTE FIRST
                ;J<HEX #>(CR)
                ;
022C CDAA03     DOUBLE: CALL    HEXC
022F EB                 XCHG
0230 2A500C             LHLD    WINDOW
0233 73                 MOV     M,E
0234 23                 INX     H
0235 72                 MOV     M,D
0236 C9                 RET
                ;
                ;LOD LOADS WINDOW WITH THE NUMBER
                ;FOLLOWING THE L COMMAND
                ;L<HEX NUMBER>(CR)
                ;
0237 CDAA03     LOD:    CALL    HEXC
023A 22500C     SVW:    SHLD    WINDOW
023D C9                 RET
                ;
                ;ADDIT MOVES THE MEMORY POINTER (WNDOW)
                ;BY FORWARD OR BACKWARDS BY
                ;THE VALUE IN E (D=0)
                ;E IS OFFSET BY 16
                ;
023E 2A500C     ADDIT:  LHLD    WINDOW
0241 01F0FF             LXI     B,-16
0244 09                 DAD     B
0245 19                 DAD     D
0246 C33A02             JMP     SVW
                ;
                ;IND LOADS THE NEXT 2 BYTES IN MEMORY
                ;INTO WINDOW
                ;
0249 2A500C     IND:    LHLD    WINDOW
024C 5E                 MOV     E,M
024D 23                 INX     H
024E 56                 MOV     D,M
024F EB                 XCHG
0250 C33A02             JMP     SVW
                ;
                ;STORE STORES THE BYTE IN L (ENTERED
                ;FROM CONSOLE) INTO MEMORY
                ;
```

```
0253 7D       STORE:  MOV     A,L
0254 2A500C           LHLD    WINDOW
0257 77               MOV     M,A
0258 C9               RET
                      ;
                      ;X EXECCUTES ONE INSTRUCTION POINTED TO
                      ;BY SAVPC AND RETURNS TO RST7
                      ;
0259 E1       X:      POP     H       ;2 DUMMY POPS
025A E1               POP     H       ;GET PC
025B 22520C           SHLD    SAVPC   ;SAVE IT
025E E1               POP     H
025F E1               POP     H       ;GET REGISTERS
0260 D1               POP     D
0261 C1               POP     B
0262 F1               POP     PSW
0263 E3               XTHL    ;RESTORE PC
0264 2A520C           LHLD    SAVPC
0267 E3               XTHL
0268 FB               EI
0269 D30C             OUT     12      ;ENABLE SINGLE STEP LOGIC
026B C9               RET             ;GO TO USER PGM.
                      ;
                      ;G ACTS THE SAME AS X BUT DOES NOT
                      ;ENABLE SINGLE STEP LOGIC, AND
                      ;THEREFORE DOES NOT RETURN
                      ;TYPING CTL-Z WILL RETURN FROM THE
                      ;PROGRAM BEING EXECUTED AND SAVE
                      ;ALL REGISTERS
                      ;IF A RST7 IS ENCOUNTERED THIS WILL
                      ;ALSO HAPPEN
                      ;
026C E1       G:      POP     H
026D E1               POP     H
026E 22520C           SHLD    SAVPC
0271 E1               POP     H
0272 E1               POP     H
0273 D1               POP     D
0274 C1               POP     B
0275 F1               POP     M
0276 E3               XTHL
0277 2A520C           LHLD    SAVPC
027A E3               XTHL
027B C9               RET
                      ;
                      ;SETR POINTS TO ONE OF THE 8080 REGISTERS
                      ;AS SAVED IN MEMORY
                      ;MAY BE USED WITH JUMBO COMMAND TO
                      ;SET A REGISTER PAIR, OR WITH OTHER
                      ;COMMANDS TO SET INDIVIDUAL REGISTERS
                      ;S<P/H/D/B/A>
                      ;
027C CDA103   SETR:   CALL    LCFLD   ;GET REGISTER DESIGNATION
027F 218F02           LXI     H,RTAB  ;AND LOOK UP POSITION
```

```
0282 CDDB01           CALL    LOOKUP
0285 D8               RC                ;RETURN IF NOT VALID REGISTER
0286 210200           LXI     H,2
0289 54               MOV     D,H       ;SET D-0
028A 39               DAD     SP        ;ADD 2 TO STACK POINTER (FOR CAL
028B 19               DAD     D         ;ADD REG. POSITION
028C C33A02           JMP     SVW       ;PUT IN WINDOW
                      ;
                      ;RTAB IS A TABLE OF THE 8080 REGISTERS
                      ;AND THEIR RELATIVE POSITIONS AS
                      ;STORED ON THE STACK
                      ;
028F 50       RTAB:   DB      'P'
0290 00               DB      0
0291 48               DB      'H'
0292 04               DB      4
0293 44               DB      'D'
0294 06               DB      6
0295 42               DB      'B'
0296 08               DB      8
0297 41               DB      'A'
0298 0A               DB      10
0299 FF               DB      -1
                      ;
                      ;GET IS USED BY THE LOADER TO GET
                      ;C BYTES AND STORE THEM AT HL TO HL+C-1
                      ;
029A AF       GET:    XRA     A
029B 47               MOV     B,A
029C CDA502   GET1:   CALL    TI
029F 77               MOV     M,A       ;PUT BYTE IN MEMORY
02A0 23               INX     H         ;INCREMENT POINTER
02A1 0D               DCR     C
02A2 C29C02           JNZ     GET1
                      ;
                      ;TI GETS A CHARACTER FROM WH2
                      ;AND KEEPS A CHECKSUM IN B
                      ;D IS USED AS A TEMPORARY
                      ;
02A5 CD280C   TI:     CALL    WH2
02A8 57               MOV     D,A
02A9 80               ADD     B
02AA 47               MOV     B,A
02AB 7A               MOV     A,D
02AC C9               RET
                      ;
                      ;SETUP PUTS IMMEDIATE BYTES INTO BAUD RATE GEN.,
                      ;AND THEN USART CTL PORT.  THE TERM CHAR IS 00H,
                      ;WHICH IS ALSO XMTED TO USART, LEAVING IT IDLING.
                      ;WHEN DONE, JUMPS TO LOC. AFTER
                      ;IMMED BYTES.
                      ;
02AD E1       SETUP:  POP     H         ;GET ADDR FOLLOWING CALL TO SETU
02AE 7E               MOV     A,M       ;GET DATA FOR BRG
```

```
02AF D304               OUT     4       ;PUT IN BRG
02B1 23         SET1:   INX     H       ;NEXT BYTE
02B2 7E                 MOV     A,M
02B3 D301               OUT     1       ; USART PORT
02B5 B7                 ORA     A       ;WAS THAT 00H?
02B6 C2B102             JNZ     SET1    ;NO,NEXT BYTE
02B9 E9                 PCHL            ;JUMP TO THE 00H, EXEC. IT AS A
                                        ;CONTINUE
                ;
                ;POLY AND BITE CONTAIN SETUP
                ;INFORMATION FOR POLYPHASE OR BYTE
                ;OPERATION OF THE USART
                ;
02BA CDAD02     POLY:   CALL    SETUP   ;THE NEXT BYTES GOTO BRG AND USA
02BD 05                 DB      005H    ;TO BRG: SELECT DEV 0, 2400 BAUD
02BE AA                 DB      0AAH    ;FAKE SYNCH CHAR IF USART EXPECT
02BF 40                 DB      040H    ;INTERN. RESET. GETS USART TO MO
02C0 0C                 DB      00CH    ;MODE CODE FOR SYNCHRONOUS, 8 BI
                                        ;NO PARITY, INTERN. SYNC, 2 SYNC
02C1 E6                 DB      0E6H    ;FIRST SYNC CHAR. TO SRCH FOR
02C2 E6                 DB      0E6H    ;2ND SYNC CHAR.
02C3 00                 DB      000H    ;LEAVE COMMAND AT 00H (IDLE), RT
02C4 C3CF02             JMP     NAMER
02C7 CDAD02     BITE:   CALL    SETUP
02CA 06                 DB      006H    ;TO BRG: SEL. DEV. 0, 300 BAUD
02CB AA                 DB      0AAH    ;FAKE SYNCH CHAR IF USART EXPECT
02CC 40                 DB      040H    ;INT. RST. GETS US TO MODE LEVEL
02CD CE                 DB      0CEH    ;MODE: ASYNCH., 8 BITS, NO PARIT
                                        ;2 STOP BITS, 16X CLOCK SCALING
02CE 00                 DB      000H    ;FOR NOW COMMAND IS IDLE
02CF 21540C     NAMER:  LXI     H,FNAME
02D2 79                 MOV     A,C     ;ECHO THE B OR P
02D3 CD240C             CALL    WH1
02D6 CD8D03             CALL    CROUT   ;OUTPUT CR
02D9 0E09               MVI     C,9
                ;
                ;NAM0 GETS THE NAME OF THE FILE
                ;TO BE LOADED FROM THE CONSOLE
                ;
02DB CD200C     NAM0:   CALL    WH0
02DE CD240C             CALL    WH1     ;ECHO CHAR.
02E1 FE0D               CPI     13
02E3 CAEC02             JZ      NAM     ;DONE IF CR
02E6 77                 MOV     M,A     ;STORE IN MEMORY
02E7 23                 INX     H
02E8 0D                 DCR     C
02E9 C2DB02             JNZ     NAM0    ;DONE IF 9 CHARACTERS
02EC 3620       NAM:    MVI     M,20H   ;FILL OUT WITH BLANKS
02EE 23                 INX     H
02EF 0D                 DCR     C
02F0 F2EC02             JP      NAM
                ;
                ;HEAD SEARCHES FOR A RECORD HEADER
                ;AND STORES IT AT RNAME
```

```
                ;COMP THEN COMPARES THE NAME AGAINST
                ;THE NAME IT IS SEARCHING FOR
                ;GETS NEXT HEADER IF NOT MATCH
                ;AFTER DISPLAYING NAME AND RECORD NUMBER
                ;IF CHECKSUM ERROR GOES TO ERROR
                ;
02F3 214203     HEAD:   LXI     H,DNAME
02F6 E5                 PUSH    H       ;ANYTHING RETURNING AFTER HERE
                                        ;WILL DISPLAY THE RECD NAME AND
02F7 3E96       HEAD6:  MVI     A,096H  ;START USART READING, ENTER SEAR
                                        ;IF SYNCHRONOUS, AND START MOTOR
02F9 D301               OUT     1       ;TO USART CTL PORT
02FB CD280C             CALL    WH2
02FE FEE6       HEAD7:  CPI     0E6H    ;SYNC CHAR.
0300 C2F702             JNZ     HEAD6   ;RESYNC USART
0303 CD280C             CALL    WH2
0306 FE01               CPI     001H    ;SOH CHAR.
0308 C2FE02             JNZ     HEAD7
030B 215C0C             LXI     H,RNAME
030E 0E0E               MVI     C,14
0310 CD9A02             CALL    GET     ;GET HEADER
0313 C27003             JNZ     ERROR
0316 215C0C             LXI     H,RNAME ;COMPARE NAMES
0319 11540C             LXI     D,FNAME
031C 0E08               MVI     C,8
031E 1A         COMP:   LDAX    D
031F BE                 CMP     M
0320 C0                 RNZ             ;NOT MATCH
0321 13                 INX     D
0322 23                 INX     H
0323 0D                 DCR     C
0324 C21E03             JNZ     COMP    ;FALL THRU IF MATCH
0327 2A670C             LHLD    RADR    ;GET LOAD ADDRESS AND
032A 3A660C             LDA     RLEN    ;LENGTH IN PREPARATION FOR LOADI
032D 4F                 MOV     C,A
032E 3A690C             LDA     RTYPE   ;CHECK RECORD TYPE
0331 B7                 ORA     A
0332 CA6C03             JZ      GETD    ;DATA, LOAD INTO RAM
0335 3D                 DCR     A
0336 CA5F03             JZ      COMNT   ;COMMENT, DISPLAY IT
0339 3D                 DCR     A
033A CA7503             JZ      STOP    ;END OF FILE, STOP TAPE
033D 3D                 DCR     A
033E C0                 RNZ             ;NOT TYPES 0-3, TRY AGAIN
033F D301               OUT     1       ;STOP TAPE
0341 E9                 PCHL    ;AUTO-EXECUTE, GO TO PGM.
                ;
                ;WERE DONE LOOKING AT OR READING A RECD, SO DISPLAY
                ;NAME AND RECD# OF LAST SEEN RECD.
                ;
0342 015C0C     DNAME:  LXI     B,RNAME
0345 1608               MVI     D,8
0347 37         DNAM2:  STC
0348 CD7C01             CALL    FLDSY
```

```
034B 15              DCR     D
034C C24703          JNZ     DNAM2
034F CD9703          CALL    BLK
0352 2A640C          LHLD    RRN     ;DISPLAY R/N
0355 EB              XCHG
0356 CDD103          CALL    DEOUT
0359 CD8D03          CALL    CROUT
035C C3F302          JMP     HEAD
                     ;
                     ;COMNT DISPLAYS COMMENTS ON THE SCREEN
                     ;
035F CD280C          COMNT:  CALL    WH2     ;ECHO TAPE ON CRT FOR A COMMENT
0362 CD7F00          CALL    DSPLY
0365 0D              DCR     C
0366 C25F03          JNZ     COMNT
0369 C3F702          JMP     HEAD6
                     ;
                     ;GETD GETS DATA AND RETURNS IF CS2 IS GOOD
                     ;OTHERWISE IT STOPS THE TAPE
                     ;AND PRINTS A "?"
                     ;
036C CD9A02          GETD:   CALL    GET
036F C8              RZ
0370 3E3F            ERROR:  MVI     A,'?'
0372 CD7F00          CALL    DSPLY
0375 CD8D03          STOP:   CALL    CROUT
0378 AF              XRA     A
0379 D301            OUT     1       ;OUTPUT NUL TO STOP USART
037B E1              POP     H       ;CLEAN UP STACK
037C C31702          JMP     START
                     ;
                     ;***** UTILITY SUBROUTINES *****
                     ;
                     ;THESE SUBROUTINES MAY BE USED EXTERNALLY, SO WE WANT
                     ;THEM IN KNOWN LOCATIONS.
                     ;
                     ;HEXO8 OUTPUTS 8 BYTES FROM THE ADDRESS POINTED
                     ;TO BY D,E LEAVING D,E PONITING TO THE NEXT
                     ;LOCATION IN MEMORY.  IT PUTS THE BYTES OUT
                     ;IN HEX WITH A SPACE BETWEEN THEM AND A
                     ;CARRAGE RETURN AT THE END OF THE LINE.
                     ;
                     ;D2 IS THE ACTUAL LOOP WHICH HEXO8 USES
                     ;IT PUTS OUT THE NUMBER OF BYTES IN B
                     ;INCRAMENTS D AND THEN PUTS OUT A
                     ;CARRAGE RETURN.
                     ;
037F 0608            HEXO8:  MVI     B,8
0381 CD9703          D2:     CALL    BLK
0384 1A              LDAX    D
0385 CDD603          CALL    BYTE
0388 13              INX     D
0389 05              DCR     B
038A C28103          JNZ     D2
```

```
                        ;
                        ;CROUT, CLEAR, BLK, AND TABBER
                        ;OUTPUT A CARRIAGE RETURN, FORM FEED, BLANK,
                        ;OR HORIZONTAL TAB TO THE CRT DRIVER, RESPECTIVELY
                        ;
038D 3E0D       CROUT:  MVI     A,CR    ;PUT CAR RETRN ON CONSLE DSPLY
038F C37F00             JMP     DSPLY
0392 3E0C       CLEAR:  MVI     A,12    ;CTL-L (FORM FEED)
0394 C37F00             JMP     DSPLY
0397 3E20       BLK:    MVI     A,' '   ;SPACE
0399 C37F00             JMP     DSPLY
039C 3E09       TABBER: MVI     A,9     ;CTL-I TAB
039E C37F00             JMP     DSPLY
                        ;
                        ;LCFLD (LOWER CASE FOLD). GETS A CHAR FROM WH0.
                        ;IF AN LC CHAR WAS FOLDED, CY=0,ELSE CY=1.
                        ;WATCH RUBOUTS! THEY'RE FOLDED TO 5FH FROM NORMAL 7FH.
                        ;
03A1 CD200C     LCFLD:  CALL    WH0     ;GET CHAR
03A4 FE60               CPI     060H    ;IF UPPER CASE,FORGET IT
03A6 D8                 RC              ;UC LETTERS ARE LESS THAN 60H
03A7 D620               SUI     020H    ;FOLD THIS LOWER CASE LETTER ON
03A9 C9                 RET
                        ;
                        ;HEXC INPUTS VARIABLE LENGTH HEX FROM WH0, ECHOING ON DS
                        ;TERM CHAR IS ANY NON-HEX CHAR, AND IT IS RETURNED IN B.
                        ;DIGIT COUNT RETURNED IN C.
                        ;
03AA 210000     HEXC:   LXI     H,0     ;ZERO CONVERSION BUFFER
03AD 4D                 MOV     C,L
03AE CDA103     NXNYB:  CALL    LCFLD   ;GET A CASE-FOLDED CHAR FROM WH0
03B1 47                 MOV     B,A     ;SAVE THIS CHAR SO IT CAN BE USE
                                        ;IT WAS THE TERM CHAR.
03B2 FE30               CPI     '0'     ;RETURN IF LESS THAN ASCII 0
03B4 D8                 RC
03B5 CD240C             CALL    WH1     ;ECHO EACH CHAR. VALID OR NOT
03B8 D630               SUI     '0'     ;CHANGE ASCII INTO BINARY 0-15
03BA FE0A               CPI     10
03BC DAC703             JC      NXNB1
03BF D607               SUI     7
03C1 FE0A               CPI     10
03C3 D8                 RC              ;RETURN IF NOT HEX
03C4 FE10               CPI     16
03C6 D0                 RNC             ;RETURN IF NOT HEX
03C7 0C         NXNB1:  INR     C       ;COUNT # OF HEX CHARACTERS
03C8 29                 DAD     H       ;SHIFT HL
03C9 29                 DAD     H       ;OVER
03CA 29                 DAD     H       ;FOR NEXT
03CB 29                 DAD     H       ;DIGIT
03CC B5                 ORA     L       ;OR IN NEW DIGIT
03CD 6F                 MOV     L,A
03CE C3AE03             JMP     NXNYB
                        ;
                        ;DEOUT OUTPUTS DE TO THE SCREEN
```

```
                        ;AS 4 HEX DIGITS
                        ;
03D1 7A         DEOUT:  MOV     A,D
03D2 CDD603             CALL    BYTE
03D5 7B                 MOV     A,E
                        ;
                        ;BYTE OUTPUTS THE ACCUMULATOR
                        ;AS 2 HEX DIGITS TO THE SCREEN
                        ;
03D6 F5         BYTE:   PUSH    PSW
03D7 0F                 RRC
03D8 0F                 RRC
03D9 0F                 RRC
03DA 0F                 RRC
03DB CDDF03             CALL    HEXO
03DE F1                 POP     PSW
                        ;
                        ;HEXO OUTPUTS 1 HEX DIGIT TO
                        ;THE SCREEN - THE UPPER HALF
                        ;OF A IS MASKED WITH ZEROS
                        ;
03DF E60F       HEXO:   ANI     15
03E1 C690               ADI     90H
03E3 27                 DAA
03E4 CE40               ACI     40H
03E6 27                 DAA
03E7 C37F00             JMP     DSPLY   ;OUTPUT HEX DIGIT AND USE RETURN
                        ;
                        ;****** INITIALIZATION PARAMETERS ******
                        ;THE FOLLOWING INFORMATIO IS USED ON FPRST OR POC
                        ;TO SETUP THE STARTING SYSTEM CONTEXT.
                        ;
1000            STACK   EQU     01000H  ; USED IN AN LXI,SP
                        ;
                        ;THE FOLLOWING BLOCK IS COPIED DIRECTLY OVER
                        ;SYSTEM RAM STARTING AT SRA4.
                        ;
03EA 5400       INITER: DW      USRTSR  ;VI3 USART INTERUPT
03EC 0D01               DW      KSR     ;VI2:  THE STANDARD KBD INT
03EE 6400               DW      IORET   ;WAKEUP: NOTHING FOR NOW
03F0 1701               DW      SAVE    ;VI0: SINGLE STEP INT GOES BACK
03F2 FC                 DB      0FCH    ;VIDEO SCRN ENDS AT FC00H-1
03F3 F8                 DB      0F8H    ;VIDEO SCRN HOME AT F800H
03F4 CD6A00             CALL    KI      ;WORMHOLE 0: INIT TO KBD AT F8H
03F7 C9                 RET             ;STANDARD PART OF ANY WORMHOLE
03F8 CD7F00             CALL    DSPLY   ;WORMHOLE 1:  INIT TO VIDEO DSPL
03FB C9                 RET
03FC CD7800             CALL    USRTI   ;WORMHOLE 2: INIT TO USART AT 0
03FF C9                 RET
0000                    END
```

SYMBOLS SORTED BY NAME

ADDIT	023E	BITE	02C7	BLK	0397	BYTE	03D6	CLEAR	0392
CLINE	00D0	CLOCK	0030	COMD	01BA	COMNT	035F	COMP	031E
CR	000D	CROUT	038D	CRR	00E6	CTLX	0018	CTLZ	001A
CURP	00DC	D2	0381	DEOUT	03D1	DISP	0134	DISP1	0148
DNAM2	0347	DNAME	0342	DOUBL	022C	DSPLY	007F	ERROR	0370
FF	000C	FF1	00F2	FFR	00EF	FLAGS	0187	FLDSY	017C
FNAME	0C54	G	026C	GET	029A	GET1	029C	GETD	036C
HEAD	02F3	HEAD6	02F7	HEAD7	02FE	HEXC	03AA	HEXO	03DF
HEXO8	037F	IND	0249	INITE	03EA	INITL	FFEA	IOPUT	0060
IORET	0064	KBUFF	0C0C	KI	006A	KI1	006E	KSR	010D
LCFLD	03A1	LOD	0237	LOOKU	01DB	MMOD	0198	MMOD1	01A8
MOVE	0100	MSG	016D	MTBL	01E8	NAM	02EC	NAM0	02DB
NAMER	02CF	NORM	00B4	NXNB1	03C7	NXNYB	03AE	POLY	02BA
POS	0C0E	POS2	0C4A	POS3	0C4E	RADR	0C67	RB	007F
RBR	00E4	RBUFF	0C0A	RESET	0000	RLEN	0C66	RNAME	0C5C
RRN	0C64	RST1	0200	RTAB	028F	RTN	0200	RTYPE	0C69
SAVE	0117	SAVPC	0C52	SCEND	0C1E	SCND2	0C48	SCND3	0C4C
SCRH2	0C49	SCRH3	0C4D	SCRHM	0C1F	SCRL	00B9	SET1	02B1
SETR	027C	SETUP	02AD	SRA1	0C10	SRA2	0C12	SRA3	0C14
SRA4	0C16	SRA5	0C18	SRA7	0C1C	SS	0038	STACK	1000
START	0217	STOP	0375	STORE	0253	SVW	023A	TAB	0009
TABBE	039C	TANI	0C04	TAPE	0214	TBUFF	0C08	TI	02A5
TIME	0040	TIME2	0047	TIMER	0C00	USRTI	0078	USRTS	0054
VCB2	0C48	VCB3	0C4C	VI2	0028	VI3	0020	VI4	0018
VI5	0010	VI6	0008	VT	000B	VTR	00F9	WAKEU	0C1A
WARM	0121	WH0	0C20	WH1	0C24	WH2	0C28	WH3	0C2C
WH4	0C30	WH5	0C34	WH6	0C38	WH7	0C3C	WH8	0C40
WH9	0C44	WINDO	0C50	WIPE	00D5	X	0259		

SYMBOLS SORTED BY VALUE

RESET	0000	VI6	0008	TAB	0009	VT	000B	FF	000C
CR	000D	VI5	0010	CTLX	0018	VI4	0018	CTLZ	001A
VI3	0020	VI2	0028	CLOCK	0030	SS	0038	TIME	0040
TIME2	0047	USRTS	0054	IOPUT	0060	IORET	0064	KI	006A
KI1	006E	USRTI	0078	DSPLY	007F	RB	007F	NORM	00B4
SCRL	00B9	CLINE	00D0	WIPE	00D5	CURP	00DC	RBR	00E4
CRR	00E6	FFR	00EF	FF1	00F2	VTR	00F9	MOVE	0100
KSR	010D	SAVE	0117	WARM	0121	DISP	0134	DISP1	0148
MSG	016D	FLDSY	017C	FLAGS	0187	MMOD	0198	MMOD1	01A8
COMD	01BA	LOOKU	01DB	MTBL	01E8	RST1	0200	RTN	0200
TAPE	0214	START	0217	DOUBL	022C	LOD	0237	SVW	023A
ADDIT	023E	IND	0249	STORE	0253	X	0259	G	026C
SETR	027C	RTAB	028F	GET	029A	GET1	029C	TI	02A5
SETUP	02AD	SET1	02B1	POLY	02BA	BITE	02C7	NAMER	02CF
NAM0	02DB	NAM	02EC	HEAD	02F3	HEAD6	02F7	HEAD7	02FE

COMP	031E	DNAME	0342	DNAM2	0347	COMNT	035F	GETD	036C
ERROR	0370	STOP	0375	HEXO8	037F	D2	0381	CROUT	038D
CLEAR	0392	BLK	0397	TABBE	039C	LCFLD	03A1	HEXC	03AA
NXNYB	03AE	NXNB1	03C7	DEOUT	03D1	BYTE	03D6	HEXO	03DF
INITE	03EA	TIMER	0C00	TANI	0C04	TBUFF	0C08	RBUFF	0C0A
KBUFF	0C0C	POS	0C0E	SRA1	0C10	SRA2	0C12	SRA3	0C14
SRA4	0C16	SRA5	0C18	WAKEU	0C1A	SRA7	0C1C	SCEND	0C1E
SCRHM	0C1F	WH0	0C20	WH1	0C24	WH2	0C28	WH3	0C2C
WH4	0C30	WH5	0C34	WH6	0C38	WH7	0C3C	WH8	0C40
WH9	0C44	SCND2	0C48	VCB2	0C48	SCRH2	0C49	POS2	0C4A
SCND3	0C4C	VCB3	0C4C	SCRH3	0C4D	POS3	0C4E	WINDO	0C50
SAVPC	0C52	FNAME	0C54	RNAME	0C5C	RRN	0C64	RLEN	0C66
RADR	0C67	RTYPE	0C69	STACK	1000	INITL	FFEA		

THATS ALL, FOLKS!

****** SMALL DUMPER FOR 4.0 ONBOARD RAM ******

SMD IS A SIMPLE ABSOLUTE DUMPER WHICH RUNS ENTIRELY WITHIN THE ONBOARD MONITOR RAM FROM C6AH TO D9CH. ITS STARTING ADDRESS IS C6A HEX. WHEN RUN, IT CLEARS THE SCREEN AND EXPECTS AN ENCODING SPECIFICATION AND FILENAME JUST AS THE 4.0 RESIDENT LOADER. AFTER THESE ARE INPUT, THE STARTING AND ENDING HEX ADDRESSES ARE INPUT AS SHOWN IN THE FOLLOWING EXAMPLE WHERE THE SMD IS USED TO COPY ITSELF:

(SCREEN CLEARED, CURSOR IN UPPER LEFT)

```
B
SMD

C6A,D9D                    (D9D USED FOR SAFETY)
C6A
D6A
D6A                        (THIS LAST IS AN ENDRECORD)
```

(SCREEN CLEARS AGAIN, READY FOR ANOTHER DUMP)

BEFORE DATA IS DUMPED, THE CASSETTE RECORDER SHOULD BE SETUP WITH THE PROPER PLUG IN THE MICROPHONE JACK. THE BYTE/BIPHASE CASSETTE CARD HAS TWO PLUGS FOR WRITING - ONE FOR BYTE AND ONE FOR BIPHASE. THE READ PLUG (LABELLED USUALLY "EAR" OR "SPKR") SHOULD NOT BE PLUGGED IN. SOME CASSETTES DO ODD THINGS WHEN BOTH THE MIC AND EXTERNAL SPKR JACKS ARE PLUGGED IN. ALSO MAKE SURE THAT ENOUGH TAPE RUNS BEFORE TYPING THE FINAL CARRIAGE RETURN ON THE END ADDRESS SPECIFICATION SO THAT NON-RECORDABLE LEADER GETS A CHANCE TO PASS BY BEFORE DUMPING STARTS.

THE ONBOARD DUMPER WAS HAND OPTIMIZED TO FIT INSIDE THE FREE SPACE ON SYSTEM RAM, BUT THE SYSTEM STACK ALSO RESIDES THERE. THIS MEANS THAT THE STACK MAY OVERRUN THE DUMPER, ERASING PART OF IT. IF THE DUMPER HAS BEEN IN RAM WHILE BASIC HAS BEEN RUN , FOR EXAMPLE, THE STACK HAS PROBABLY SQUASHED IT AT SOME TIME. IF THERE IS DOUBT, CHECK THE BYTE AT D99H. IT SHOULD BE A C9 (RETURN INSTRUCTION). IF IT IS NOT, OR YOU JUST WANT TO MAKE SURE, RELOAD THE DUMPER JUST BEFORE USING IT.

WHEN THE DUMPER IS DUMPING, EACH RECORD WILL BE DISPLAYED AS A HEX NUMBER ON THE SCREEN. THE HEX NUMBER REPRESENTS

THE ADDRESS OF THE DATA BEING DUMPED ON EACH RECORD. THAT ADDRESS IS PUT ON THE HEADER OF THE RECORD SO THE 4.0 RESIDENT LOADER WILL KNOW WHERE TO PUT IT WHEN IT IS READ BACK IN.

THE LAST RECORD IS AN "END" TYPE RECORD. IT IS PUT ON AUTOMATICALLY. IT WILL DISPLAY AS A RECORD WITH DUMP ADDRESS EQUAL TO THE ADDRESS OF THE RECORD BEFORE IT. OPTIMIZATION OF THE DUMPER'S CODE REQUIRES SOME STRANGENESSES SUCH AS THIS, BUT IN ANY CASE, THE LAST RECORD (DUMP FINISHED) WILL BE SIGNALLED BY THE SCREEN CLEARING. THIS PUTS THE DUMPER BACK IN ITS INITIAL MODE, JUST AS IF IT HAD BEEN RESTARTED AT C6AH. MORE DATA MAY BE DUMPED IF DESIRED.

```
                                ; ******* ONBOARD DUMPER FOR 4.0 *******
                        ;
                        ;       THIS IS A POLYFORMAT DUMPER FOR ABSOLUTE
                        ;DATA WHICH RUNS FROM C6A TO D9F (OR SO), START ADDRESS
                        ;C6AH.  WHEN RUN, IT ACTS LIKE 4.0 MONITOR TAPE LOAD IN
                        ;THE WAY IT ACCEPTS ENCODING SPECIFICATION (B OR P) AND
                        ;FILE NAME.  THEN IT EXPECTS TWO HEX NUMBERS FOR
                        ;START AND END DUMP ADDRESSES.  EACH RECORD DUMPED SHOWS
                        ;ADDRESS USED IN HEX ON SCREEN.  WHEN DONE, IT PUTS OUT
                        ;AN "END" TYPE RECORD AND CLEARS SCREEN, READY.
                        ;FOR ANOTHER DUMP.
                        ;
                        ;ORIGINAL 2.2 DUMPER SYSTEM WRITTEN BY DAVID FAIMAN
                        ;REWRITTEN,DOCUMENTED AND CONVERTED TO ONBOARD FOR 4.0
                        ;BY R.L.DERAN
                        ;
                        ;
0C20                    WH0     EQU     0C20H
0C24                    WH1     EQU     0C24H
0C16                    SRA4    EQU     0C16H
02AD                    SETUP   EQU     02ADH
03AA                    HEXC    EQU     03AAH
03D1                    DEOUT   EQU     03D1H
0C5A                            ORG     0C5CH-2
0C5A                    LENGTH: DS      2
0C5C                    WNAME:  DS      8
0C64                    WRN:    DS      2
0C66                    WLEN:   DS      1
0C67                    WADR:   DS      2
0C69                    WTYPE:  DS      1
0C6A 21450D             START:  LXI     H,TISR
0C6D 22160C                     SHLD    SRA4
0C70 3E0C               STAR2:  MVI     A,0CH   ;FORM FEED
0C72 CD240C                     CALL    WH1     ;CLEAR SCREEN
0C75 CD200C                     CALL    WH0
0C78 CD240C                     CALL    WH1
0C7B FE42                       CPI     'B'
0C7D CA920C                     JZ      BITE
0C80 FE50                       CPI     'P'
0C82 C2700C                     JNZ     STAR2
0C85 CDAD02             POLY:   CALL    SETUP
0C88 05                         DB      005H
0C89 AA                         DB      0AAH
0C8A 40                         DB      040H
0C8B 0C                         DB      00CH
0C8C E6                         DB      0E6H
0C8D E6                         DB      0E6H
0C8E 00                         DB      000H
0C8F C39A0C                     JMP     NAMER
0C92 CDAD02             BITE:   CALL    SETUP
0C95 06                         DB      006H
0C96 AA                         DB      0AAH
```

```
0C97 40              DB      040H
0C98 CE                DB      0CEH
0C99 00                DB      000H
                ;
                ;       NAMEING ROUTINE
                ;
0C9A 210000     NAMER:  LXI     H,0
0C9D 22640C             SHLD    WRN
0CA0 0E08               MVI     C,8     ;BLANK NAME FIELD
0CA2 21630C             LXI     H,WNAME+7
0CA5 3620       NAM:    MVI     M,020H
0CA7 2B                 DCX     H       ;BACKUP H TO WNAME
0CA8 0D                 DCR     C
0CA9 C2A50C             JNZ     NAM
0CAC 23                 INX     H
0CAD 0E08               MVI     C,8
0CAF CD180D             CALL    CRLF
0CB2 CD200C     NAM0:   CALL    WH0
0CB5 CD240C             CALL    WH1
0CB8 FE0D               CPI     00DH    ;CR
0CBA CAC30C             JZ      DUMPC
0CBD 77                 MOV     M,A
0CBE 23                 INX     H
0CBF 0D                 DCR     C
0CC0 C2B20C             JNZ     NAM0
0CC3 AF         DUMPC:  XRA     A
0CC4 32690C             STA     WTYPE
0CC7 CD180D             CALL    CRLF
0CCA CDAA03     SIZE:   CALL    HEXC
0CCD 22670C             SHLD    WADR
0CD0 78                 MOV     A,B
0CD1 CD240C             CALL    WH1
0CD4 EB                 XCHG
0CD5 CDAA03             CALL    HEXC
0CD8 CD180D             CALL    CRLF
0CDB 7D                 MOV     A,L
0CDC 93                 SUB     E
0CDD 6F                 MOV     L,A
0CDE 7C                 MOV     A,H
0CDF 9A                 SBB     D
0CE0 67                 MOV     H,A
0CE1 225A0C             SHLD    LENGTH
0CE4 CDF60C             CALL    DUMPR
0CE7 3E02       ENDC:   MVI     A,2
0CE9 32690C             STA     WTYPE
0CEC 3D                 DCR     A
0CED 32660C             STA     WLEN
0CF0 CD540D             CALL    DUMP
0CF3 C36A0C             JMP     START
                ;
                ;       DUMP DATA RECORDS
                ;
0CF6 215B0C     DUMPR:  LXI     H,LENGTH+1
0CF9 7E                 MOV     A,M
0CFA B7                 ORA     A
0CFB CA100D             JZ      OVER
0CFE 35                 DCR     M
```

```
0CFF AF             XRA     A
0D00 32660C           STA     WLEN
0D03 CD540D           CALL    DUMP
0D06 2A670C           LHLD    WADR
0D09 24               INR     H
0D0A 22670C           SHLD    WADR
0D0D C3F60C           JMP     DUMPR
0D10 2B       OVER:   DCX     H
0D11 7E               MOV     A,M
0D12 32660C           STA     WLEN
0D15 C3540D           JMP     DUMP
                ;
0D18 3E0D     CRLF:   MVI     A,0DH
0D1A CD240C           CALL    WH1
0D1D C9               RET
                ;
                ;       ROUTINE TO OUTPUT A RECORD
                ;
0D1E 0600     PUT:    MVI     B,0     ;CLEAR CHECKSUM
0D20 4F               MOV     C,A     ;PUT LENGTH OF RECORD IN C
0D21 7E       PUT0:   MOV     A,M
0D22 23               INX     H
0D23 F5               PUSH    PSW
0D24 80               ADD     B
0D25 47               MOV     B,A
0D26 F1               POP     PSW
0D27 CD340D           CALL    TO
0D2A 0D               DCR     C
0D2B C2210D           JNZ     PUT0
0D2E 78               MOV     A,B
0D2F 2F               CMA
0D30 3C               INR     A
0D31 C3340D           JMP     TO
                ;
                ;       TAPE OUTPUT ROUTIME
                ;
0C08          TBUFF   EQU     0C08H
0D34 E5       TO:     PUSH    H
0D35 21080C           LXI     H,TBUFF
0D38 F5               PUSH    PSW
0D39 7E       TO1:    MOV     A,M
0D3A B7               ORA     A
0D3B C2390D           JNZ     TO1
0D3E 23               INX     H
0D3F F1               POP     PSW
0D40 77               MOV     M,A
0D41 2B               DCX     H
0D42 34               INR     M
0D43 E1               POP     H
0D44 C9               RET
                ;
                ;       TISR IS A SIMPLE USART READER WHICH WILL
                ;       RE-TRANSMIT THE CHARACTER IN TBUFF IF IT HAS NOT
                ;       BEEN REPLACED BY THE WORMHOLE ROUTINE.  IT
                ;       DOES NOT CHECK THE FLAG, BECAUSE IT ASSUMES
                ;       THAT THE PROGRAM CALLING THE WORMHOLE IS FASTER
                ;       THAN THE USART AND SO IT ALWAYS HAS A VALID
```

```
                ;       CHARACTER FOR US TO TAKE.
                ;
0D45 AF         TISR:   XRA     A
0D46 32080C             STA     TBUFF
0D49 3A090C             LDA     TBUFF+1
0D4C D300               OUT     0
0D4E E1         IORET:  POP     H
0D4F D1                 POP     D
0D50 C1                 POP     B
0D51 F1                 POP     PSW
0D52 FB                 EI
0D53 C9                 RET
                ;
                ;       DUMP PUTS OUT ONE COMPLETE RECORD.
                ;       IT TURNS ON USART AND MOTORS,WAITS A WHILE
                ;       FOR AN IRG, PUTS OUT 64 SYNCH CHARACTERS,
                ;       DUMPS A RECORD ACCORDING TO THE WRITE CONTROL
                ;       BLOCK  AT WNAME (IT ALSO PUTS THE WCB
                ;       ON THE RECORD AS HEADER), INCREMENTS THE RECORD
                ;       NUMBER, STOPS USART AND MOTORS, AND RETURNS.
                ;
0D54 3E21       DUMP:   MVI     A,021H
0D56 D301               OUT     1
0D58 2A670C             LHLD    WADR
0D5B EB                 XCHG
0D5C CDD103             CALL    DEOUT   ;DISPLAY THE ADDRESS WE'RE DUMPI
0D5F CD180D             CALL    CRLF
0D62 21FF8F             LXI     H,08FFFH
0D65 2B         DELAY:  DCX     H
0D66 7C                 MOV     A,H
0D67 B7                 ORA     A
0D68 C2650D             JNZ     DELAY
0D6B 0E40               MVI     C,64
0D6D 3EE6               MVI     A,0E6H  ;SYNC CHARACTER
0D6F CD340D     DUMP0:  CALL    TO
0D72 0D                 DCR     C
0D73 C26F0D             JNZ     DUMP0
0D76 3E01               MVI     A,001H  ;START OF HEADER
0D78 CD340D             CALL    TO
                ;
                ;       DUMP HEADER AND DATA RECORDS
                ;
0D7B 3E0E               MVI     A,00EH  ;LENGTH OF HEADER RECORD
0D7D 215C0C             LXI     H,WNAME
0D80 CD1E0D             CALL    PUT
0D83 3A660C             LDA     WLEN
0D86 2A670C             LHLD    WADR
0D89 CD1E0D             CALL    PUT
0D8C 21640C             LXI     H,WRN
0D8F 34                 INR     M
0D90 AF         OFF:    XRA     A
0D91 CD340D             CALL    TO      ;THESE PUSH OUT LAST BYTES FROM
0D94 CD340D             CALL    TO      ;THE USART AND WH BUFFER PIPELIN
0D97 CD340D             CALL    TO      ;TURN OFF MOTOR AND TRANSMITTER
0D9A D301               OUT     1
0D9C C9                 RET
0000                    END
```

1. Page 67 of Volume II is out of order -- it should follow page 56 to make the text follow in a logical manner.

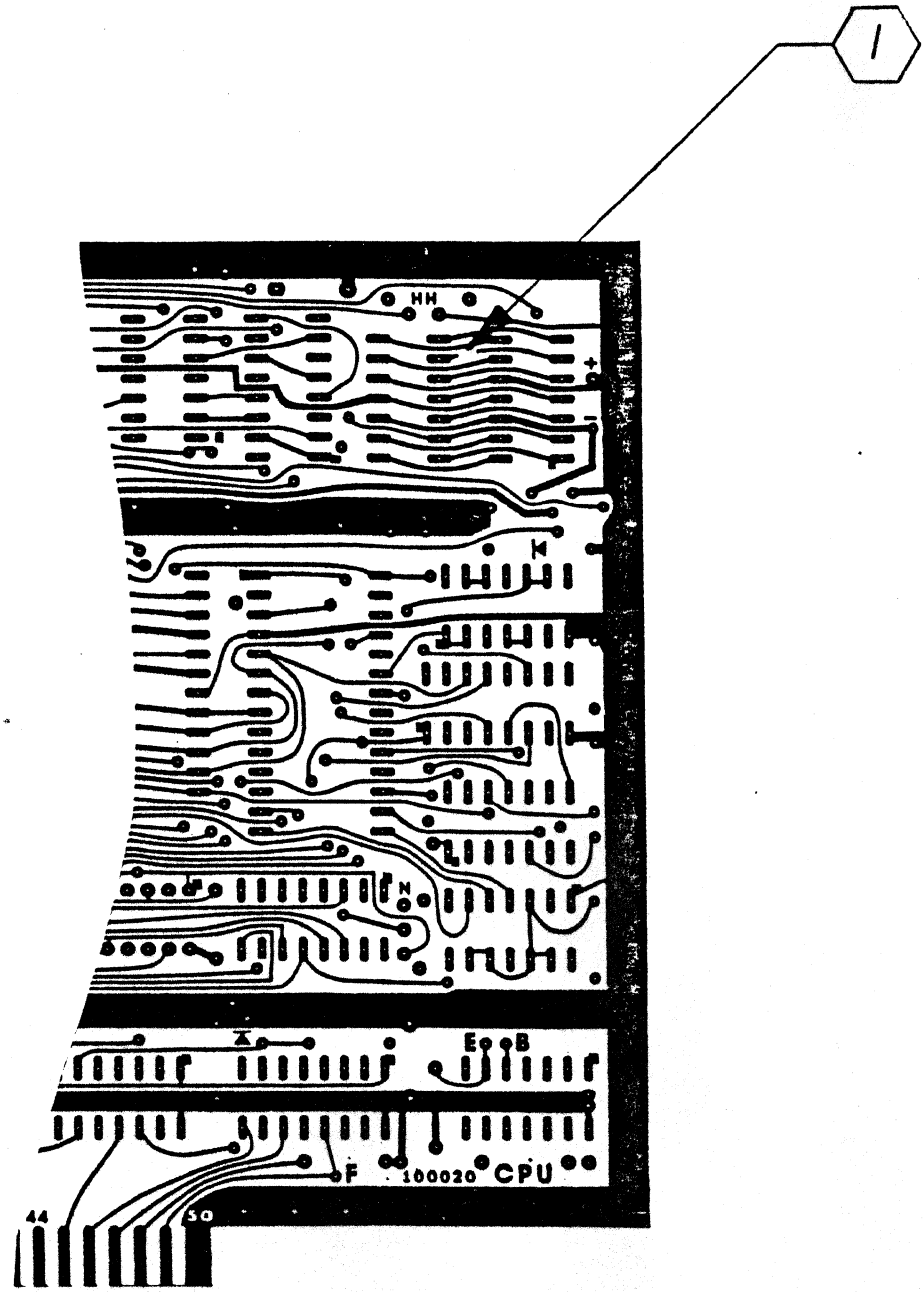
THE FOLLOWING MODIFICATION IS REQUIRED TO THE CPU BOARD FOR REVISIONS UP TO 0.3.

1. Cut trace from Port #1 pin 6 to Port #2 pin 6
2. Jumper Port #2 pin 6 to IC30 pin 5.

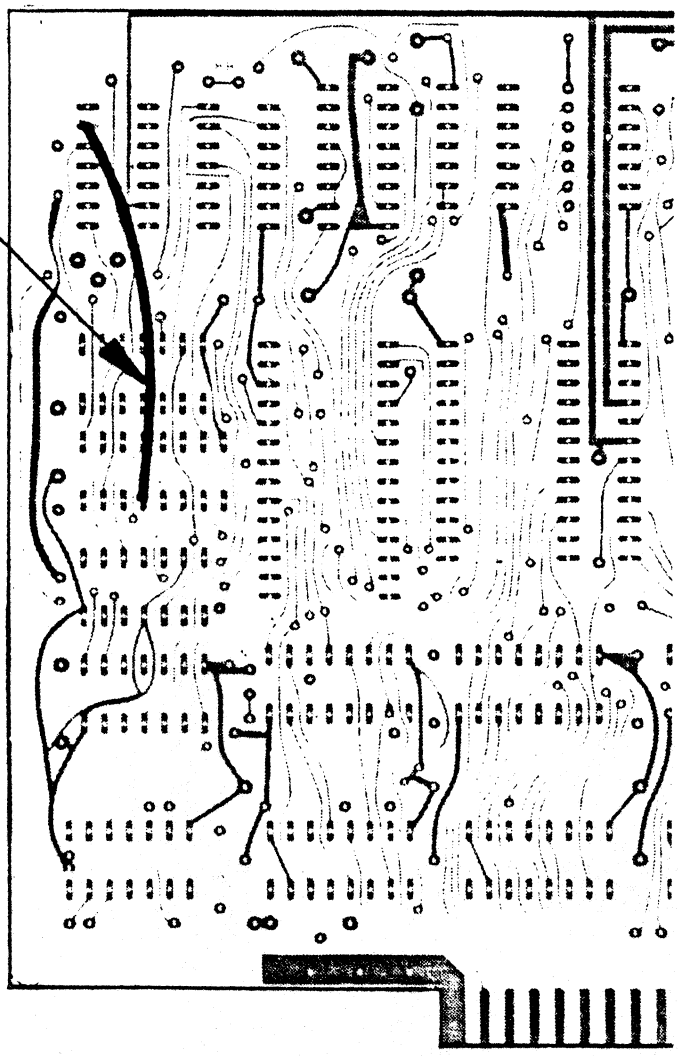
Note: Port #1 for printer cable only

Port #2 for cassette cable only

NUMBER:
105252



NUMBER:
105252



NUMBER:
105252