— SIX —

Addresses and Address Spaces

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

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

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

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

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

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

Buzzwords…

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

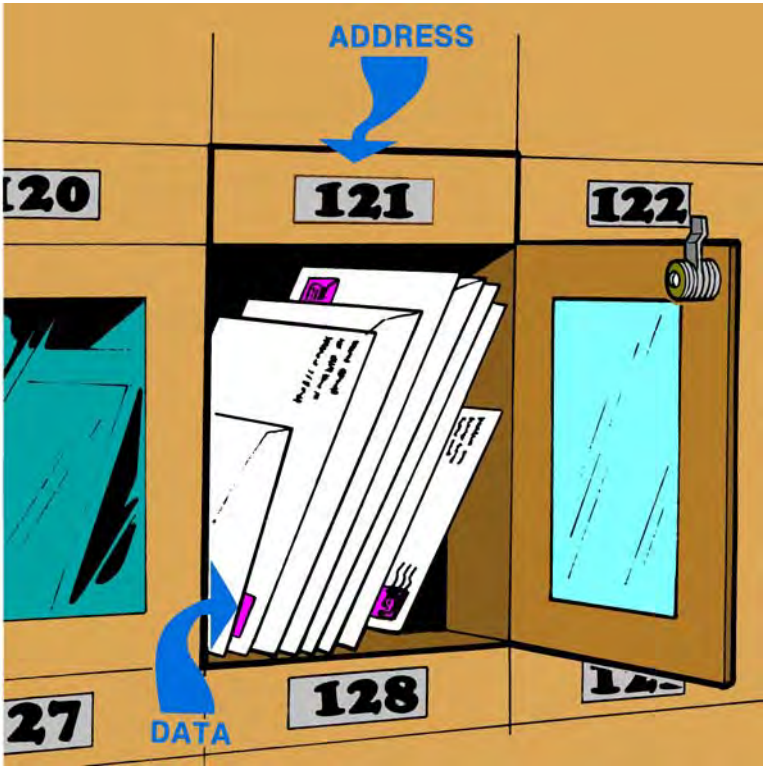
WORKING REGISTERS–Temporary stashes available inside the micro's CPU that involve themselves with practically everything the CPU does.

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

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

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

A typical box looks like this...



Repeating, an address is a location, and data is what goes in that location. Each address in a microcomputer has to be unique. No mixups are allowed. The addresses in the address space are often identified by a hex number. Typical working registers are usually identified by name or by a single letter.

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

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

So…

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

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

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

Reading DOES NOT alter the contents of the location.

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

Writing DOES alter the contents of a location.

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

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

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

Thus…

SOME ADDRESS SPACE RULES

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

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

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

These four hardware types are…

ADDRESS SPACE HARDWARE

- RAM
- ROM
- I/O
- nothing

programming restrictions. A direct I/0 micro does not prevent you from using memory mapped I/0 on it, and this is exactly what most people end up doing.

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

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

Sometimes, the unused locations can be used to simplify decoding. For instance, if you have extra address space to burn, you could give a single I/0 port 256 consecutive locations, any of which could be used to reach the port. This can greatly simplify any of decoding hardware but can become a dangerous trap.

ADDRESS SPACE

Let's take a closer look at the address space of a typical micro. We have seen the address space is the "reach" of a microprocessor, made up of all the possible locations into which we can put RAM, ROM, I/0, or nothing at all. We also now know that the micro has working registers that are usually outside the address space but inside the microprocessor chip itself.

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

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

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

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

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

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

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

SIZE	LINES	USES
4096	12	dedocated micros
65536	16	personal computers
262144	16*	business systems
1672216	24	heavy applications

*bank switched

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

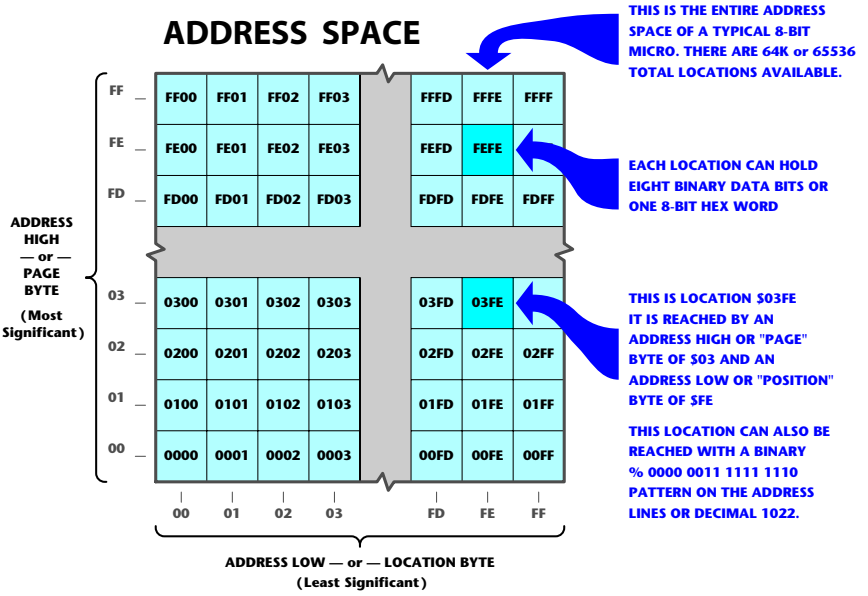
Each location in a micro's address space must be reachable by a unique address. Good old straight binary seems to be the standard and best way to reach a single location in the address space. For convenience, this binary address is normally called out in the form of a hexadecimal word.

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

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

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

Something like this .....



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

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

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

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

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

To recap .....

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

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

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

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

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

To repeat...

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

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

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

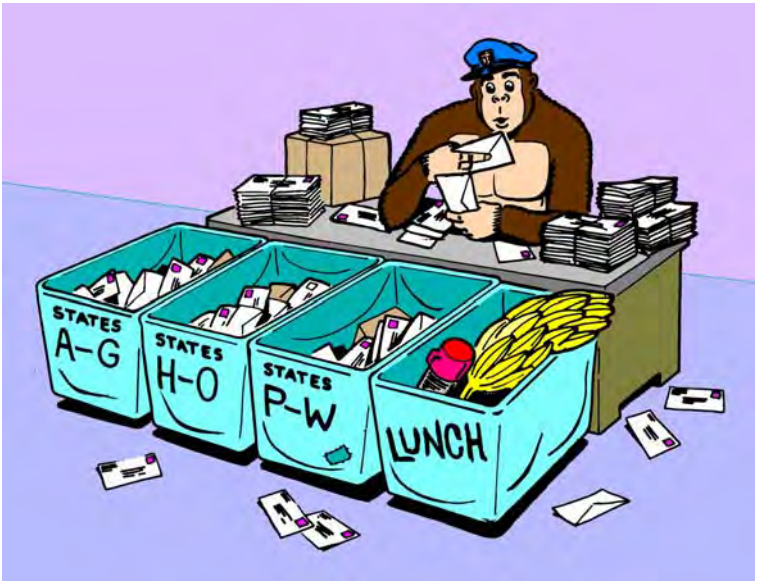
To review, the address space is the reach of a microprocessor's CPU that calls out the maximum number of places into which you can put RAM, ROM, 1/0, or nothing at all. On an 8-bit micro, each of these locations can hold one 8-bit data word. This data word can be anything you like and used anyway you want.

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

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

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

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



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

WORKING REGISTERS

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

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

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

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

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

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

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

TYPES OF WORKING REGISTERS

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

Such as an accumulator or A register.

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

Such as index or address register.

DEDICATED USE—These have one special purpose and cannot be used otherwise.

EX: A program counter or flag register.

The three main kinds of working registers include completely general ones that can involve themselves with just about anything.

You can use these any way you like. Then there are the intended use registers that have one big purpose in life. You can use them to do their thing, or else you can use them for other stuff if you don't need their specialty.

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

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

Let's look at some typical working registers and see what they can do. Later we will pick up more specific details on how certain registers in certain families work.

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

> **ACCUMULATOR**–A general-purpose working register that often holds CPU activity actions.

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

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

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

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

In the 8080 family, there is a main accumulator, along with B, C, D, and E registers. The 6800 instead offers a pair of accummulators

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

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

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

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

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

You may also find other general-use use registers in your micro. These may be secondary accumulators or simply places to store things. They may or may have all the power an accumulator does. It depends on microprocessor and you want to do with it.

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

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

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

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

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

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

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

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

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

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

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

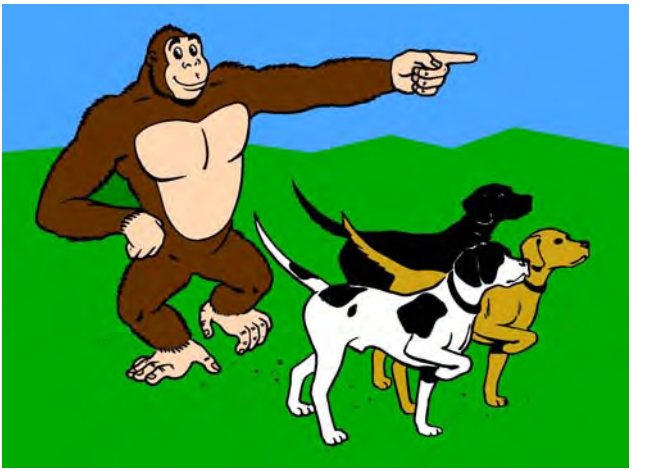
There are two popular widths of index registers. An 8-bit index register can only count down from 255 or reach 256 locations in a file from a given starting address. The X andY registers of the 6502 family are typical. You can also have 16-bit wide index registers,

such as the X register in the 6800 family. A 16-bit index register can reach any point in the 65536 location address space.

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

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

Ther are a whole class of working registers which can serve either for intended or dedicated uses, depending upon the way they are connected. These working registers are called **pointers**...



Actually, the dogs are used to point to somewhere else. We are not so interested in the dog itself as in where the dog is pointing...

> **POINTER**–A memory location that holds an address rather than data.
>
> Pointers are used to show a location where data is to come from or go to.

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

The most common pointer is called an address register...

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

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

Using an address register has two advantages. The big one is that you can calculate or change where you want to store things as you go along, rather than being stuck with absolute values locked into the program. 0 second major advantage is that a faster and shorter instruction can be used to store something at an address pointed to by an address register, compared with spelling out exactly where in the address space you are to go.

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

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

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

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

Let's now look at some dedicated working registers. These are pretty much locked into doing one job and are not available or usable for anything else. One of the most important dedicated working registers is a pointer called the program counter…

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

> The stack pointer is nowhere near the stack. The stack pointer is a dedicated working register in the CPU.
>
> The stack itself is a bunch of RAM locations out in the address space somewhere.

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

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

> **FLAG REGISTER**–A dedicated-use working register that holds all of the present conditions of the micro for us.
>
> Flag registers may also be called PROCESSOR STATUS registers.

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

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

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

The most useful thing about your flags is that you can use them to control what your micro is to do. Each flag is a single bit and each behaves per some rule or rules. Some flags you can set or clear by yourself. Others are controlled only by the CPU.

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

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

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

To sum up, working registers are special RAM locations inside the microprocessor chip. These registers can serve as temporary stashes where the CPU can work on problems and keep track of where you are and where you are going. The number of working registers will change with the micro family.

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

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

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

One final dedicated-use register is the flag register or processor status register. It keeps track of the exact micro condition.

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

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

ARCHITECTURE

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

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

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

The arrangement of resources within a micro is its…

> **ARCHITECTURE**–The arrangement of resources within a microcomputer or microprocessor.

You'll find two different architectures. One of these is the micro **processor** architecture that decides what is available inside the chip itself. The second is the micro**computer** architecture that tells you what the whole system has in the way of memory, I/0, user access, and other available resources.

For instance, the architecture of the microprocessor tells you how many working registers are available, what the total address space is, what width buses are provided, and similar chip-level features. The architecture of the microcomputer tells you how much RAM, ROM, and I/0 are now in the address space, the system meanings put on certain locations, and how you can reach specific address slots.

Something like this…

PROCESSOR ARCHITECTURE–The arrangement of things inside the CPU, including the number of working registers, bus structures, and processing details that are available.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Our address space contains RAM, ROM, I/0, and unused empty areas. RAM is memory that you can change quickly and often but is usually not permanent. ROM is memory that is more or less permanent. I/0 is input and output that give our micro ports for real-world access. While the newest RAM and ROM are getting more and more like each other, it still is safer to assume that they are different devices. You normally use RAM for things that change (such as a program or data file) and ROM for stuff that has to be permanent (such as a monitor or operating system).

The RAM, ROM, and I/0 are usually grouped into blocks. Micro systems often use much more RAM than ROM. These blocks also depend on the size of available chips. For instance, a 16K block of RAM is a very common module size for older personal computers. For more RAM, you usually add increments of 16K to bring the total to 32K, 48K, or more. ROM tends to be in blocks of 2K or 4K and expands in multiples of 2K. The reason that the ROM blocks seem smaller is that most ROM is byte wide, containing a full 8-bit word at each internal address location. Most RAM is a single bit wide, so that a 16K block of RAM may take eight different chips, one for each bit in the 8-bit word. If you wanted a full 16K of ROM, you would also need eight chips, only this time it would take eight chips of 2048 bytes each. Newer micros use 64K or larger RAM or ROM.

The area in the address space reserved for I/0 is usually very much smaller than that reserved for RAM and ROM. Few micro systems need more than a handful of I/0 ports. With a 4K space set aside for I/0, you could have 4096 different ports of eight bits each. This is vastly more than you normally would ever use.

Much of our address space could remain empty. When you are building a simple system such as a solar panel controller, you will still use the same microprocessor everyone else uses, but you use only small amounts of RAM, ROM, and I/0. The unused area in the address space is ignored. You can use this for later expansion, or you can use empty spaces to simplify the decoding process.

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

Memory map locations will often be dependent upon the chosen microprocessor. In the 6502 school, it's easy to get to memory page zero (addresses $0000-$00FF) and the handy stack storage area

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

So, the 6502 school will want you to put RAM in low addresses, starting at the bottom, and ROM in the high addresses, working down from the top. The I/0, by default, goes in the middle. Chips in the 6800 family have similar needs with RAM low and ROM high.

8080 school wants you to do the exact opposite. Low addresses are saved for interrupt and reset vectors that normally must go in ROM. The tradition here is to put ROM on the bottom, RAM on top, and, once again, I/0 in the middle. The 8048 family usually has a small address space of 4K, split into a low 2K of ROM and a high 2K of RAM. If you are using less, you build ROM from the bottom up and RAM from the top down.

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

Here are some address space rules…

- The RAM, ROM, and I/0 are usually in large blocks set by the chip sizes. 16K is a typical older RAM block while 2K or 4K are typical ROM blocks.
- The arrangement of the RAM and ROM blocks depends on the micro family. The 6502 and the 6800 families put RAM on the bottom and ROM on the top. The 8080 and 8048 families do the opposite.
- The address space need not be completely filled. Dedicated controllers will leave many empty locations. Other uses may save space for later expansion. A technique called bank switching can be used to overfill the address space, working with as many larger modules as needed.

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

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

> The DATA BUS is used to pass information between the CPU and the address space.
>
> The DATA BUS is eight bits wide on an 8-bit microcomputer.
>
> The DATA BUS is bidirectional since it must work both ways.

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

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

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

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

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

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

Summing up…

> The ADDRESS BUS is used to select an address in the address space for later access by the microprocessor's CPU.
>
> The ADDRESS BUS width is decided by the size of the address space. A 64K address space needs sixteen address lines in the bus.
>
> The ADDRESS BUS is always under control of the CPU and thus is unidirectional.

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

The address buses and data buses are normally separate and on separate pins. A few micros use the same pins to route data and addresses on a time-shared basis.

This is called a **multiplexed bus**…

> **MULTIPLEXED BUS**–A bus that can have data and addresses on it at different times.
>
> System timing must be able to sort out all the addresses and data as needed.
>
> Multiplexed buses are used by some VCIW's and by many 16-bit microprocessors.

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

address and data commands. Another disadvantage of multiplexing is that you may have to use special and nonstandard chips made by one manufacturer rather than using industry standard parts.

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

Another version of a multiplex bus gives you either eight bits of data or the eight low bits of an address. Separate address lines are provided for higher addresses.

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

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

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

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

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

A reminder…

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

Another control line is the system clock. This is a high frequency signal that is the master crank for the microprocessor. We'll look at more details on this when we get to system timing. The system clock lets you lock everything together. This gets important when you are using fancy peripheral chips or are address-pin multiplexing dynamic RAM. Sometimes several different phases of the system clock will be available and may be labeled **cp1** and **cp2** . Different phases are used for different timing needs. Clock phase timing can be very critical. Overlap must be zero.

The **reset** line is another member of the control bus. The reset line gets everything restarted properly on power up. When the microprocessor's CPU is reset, it goes into a known state of a known program and picks up from there. Some fancy I/0 chips also need to be reset to get started on the right foot. If you had a port that output random data on startup, you might simultaneously be giving "forward" and "reverse" commands to an entire steel rolling mill or bring about other unpleasantries. The reset bus is also your panic button to stop a wayward microsystem when it is up to no good.

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

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

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

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

CONTROL BUS details will vary with the chosen microcomputer in use.

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

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

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

Here's a rundown of typical control bus lines…

TYPES OF WORKING REGISTERS

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

HALT—A line used to stop the CPU temporarily.

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

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

RESET—A line that gets the CPU started on power-up or restored after an error.

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

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

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

Details of use and names will vary with the microprocessor family you pick. For instance, you'll find a single R/W control line in the 6500 and 6800 families, while the 8080, Z80, and 8048 families use two separate read and write lines.

ADDRESS SPACE DECODING

Our address space is the total reach of the microprocessor's CPU. Into this address space we put blocks of RAM, ROM, I/0, or nothing. How do we know which block we are going to access?

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

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

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

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

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

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

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

The address space consists of blocks. Each block can be RAM, ROM, I/0, or nothing at all. These blocks are arranged to suit the needs of both the microprocessor and the system designer.

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

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

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

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

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

THE MEMORY MAP

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

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

There are two memory map types. A **simplified memory map** paints the big picture. In particular, the simplified map should show you how big the address space is, indicate where the user RAM is located, point out where the monitor is, and identify the general area of 1/0 locations. The purpose of a simplified memory map is to get you started understanding and using a micro system.

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

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

Obviously, any given location can be used for only one thing at any one time. Thus…

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

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

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

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

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

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

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

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

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

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

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

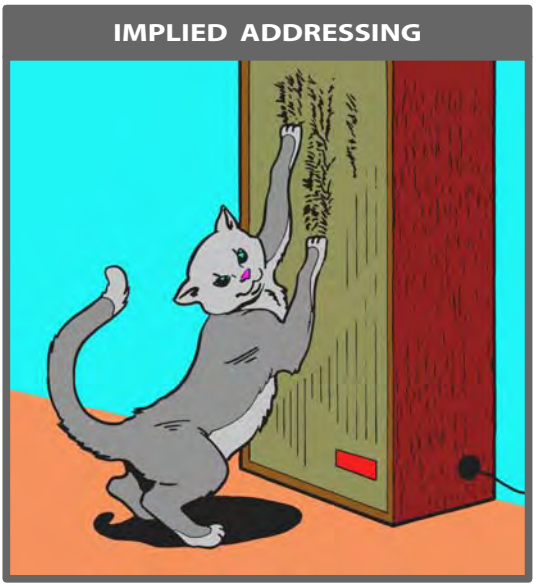
One classic example is a loop. Say something needs to be done ten times. By looping the same code ten times over, you can shorten the program almost to a tenth of its "brute force" form. But each trip through the loop means unavoidable loop overhead that takes time. As we said, short code often equals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

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

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

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

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



You just got home, and there, sharpening its claws on your best hi-fi speaker grille, was the cat. Now, exactly what you say probably isn't printable, but it will be short and to the point, leaving no doubt whatsoever **which** cat and which speaker grille you are referring to.

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

> **IMPLIED ADDRESSING**–An saddress mode where it is completly obvious what will happen where, without anuy further information needed
>
> An implied instruction usually needs only one byte of op code.

For instance, we now know we have carry flags in most micros. A command to "clear the carry flag" is often shortened to the mnemonic CLC, will clear this flag for us. On a CLC command, the carry flag gets cleared. There is no question about which flag or what we are going to do to the flag.

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

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

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

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

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

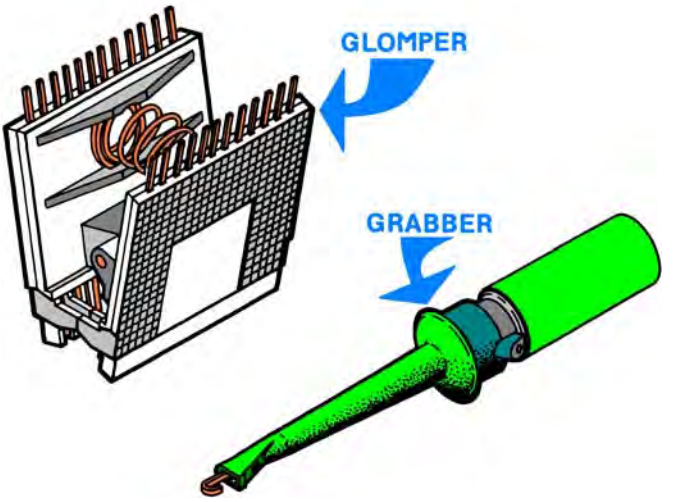
An oscilloscope lets you look into the actual working waveforms of a running microcomputer, giving you immediate details of timing and the relationships between various buses and signals. Scopes are also handy to check states of input and output ports, measure voltages, and so on. Oscilloscopes are absolutely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

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

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

glompers and grabbers

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



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

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

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

Always be careful!

electronic hand tools

A collection of the usual electronic hand tools is something you will want to pick up as you go along. Here's a very few of the bare essentials you'll need to get started…

ESSENTIAL HAND TOOLS

- ( ) **Small soldering iron**
- ( ) **Diagonal cutting pliers**
- ( ) **Needle-nose pliers**
- ( ) **Small screwdriver**
- ( ) **Medium screwdriver**
- ( ) **Wire stripper**
- ( ) **Solder sucker**
- ( ) **Desoldering braid**
- ( ) **Small magnifier**
- ( ) **Small crescent wrench**
- ( ) **X-acto knife**
- ( ) **High intensity lamp**
- ( ) **1/4-inch nut driver**
- ( ) **Third hand PC vise**
- ( ) **IC puller**

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

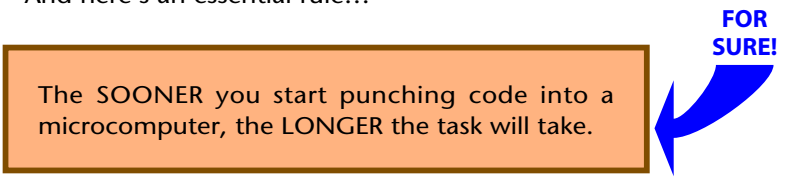
Notice that the last entry on the micro toolkit list says you need two quiet workspaces. One is where the trainer is. Another is where you can quietly think and manipulate ideas and programs on paper.

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

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

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

And here's an essential rule…



Plan before you punch.

Later we will find out the **Micro Applications Attack** as a way to use micros to solve real-world problems. It turns out there is no need ever to go near a micro until something like the tenth step of a fourteen-step process.

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

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

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

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

DOING IT:

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

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

ON BEING A GENERALIST

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

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

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

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

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

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

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

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

Think generalist. Act generalist. Be a generalist.

— SEVEN —

The Discovery Modules

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

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

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

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

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

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

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

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

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

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

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

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

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

Before we write a program, though, we might want to ask…

WHAT IS A PROGRAM?

A program is a series of machine language instructions that does some desired task…

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

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

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

Thus…

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

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

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

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

An important point…

> Only machine instructions can be "run" on a microcomputer.
>
> A data block can be used only by a program. Try to "run" the data block and the system bombs.

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

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

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

VON NEUMANN AND COMPANY

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

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

This is called a **Von Neumann architecture**.