— 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…

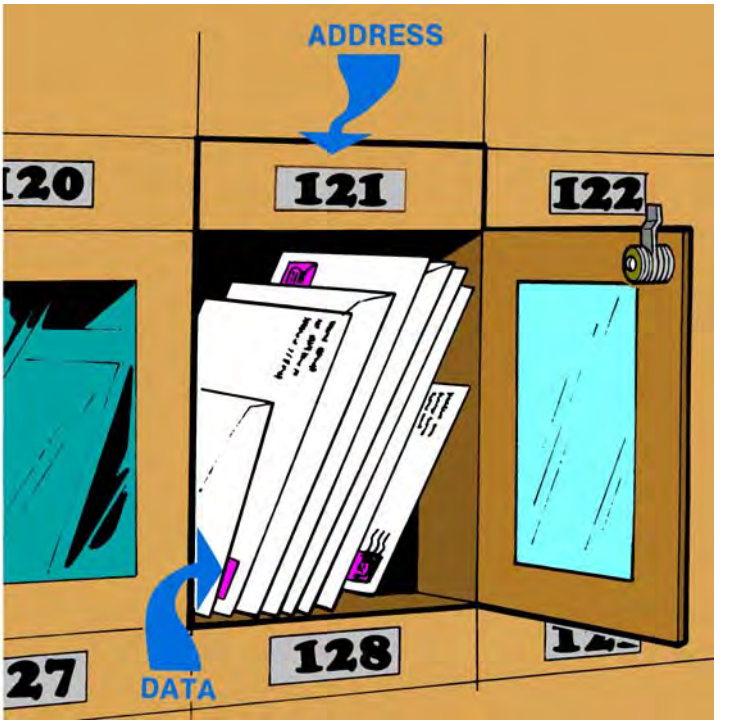
ADDRESS SPACE–The "reach" of a microprocessor's CPU. The total number of available locations the CPU can communicate with.

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

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

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

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

I/0 stands for Input and Output. We can pass information to and from the real world through suitable I/0 **ports**. When the port is an **input only** port, we can only read this location with the micro. If the port is an **output only** port, we can only write to this location with the micro. If the port is **bidirectional**, we can read from those port lines set up as inputs and write to those port lines set up as outputs. Other locations can be used to control any bidirectional port, keeping track of which lines go where. These other locations are often found nearby in the address space. More details on this in Chapter 8.

The I/0 ports that are located in the address space just like RAM and ROM are called memory mapped 1/0…

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

The big advantage of memory mapped I/0 is that anything you can do to a RAM or ROM location, you can also do to a suitable I/0 port, since the CPU does not know what is out there in the address space. Most microprocessor systems support memory mapped I/0.

There is another type of I/0 called direct I/0 that is provided on some earlier 8080 school chips. While direct I/0 can be faster and more easily decoded, it is limited to one micro family and has some

programming restrictions. A direct 1/0 micro does not prevent you from using memory mapped 1/0 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 the 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 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.

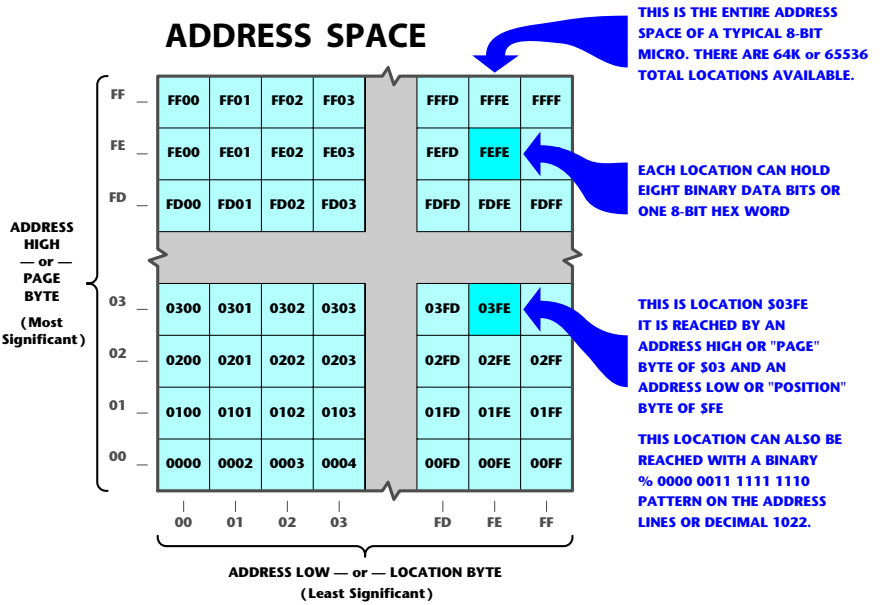
Each location in a micro's address space must be reachable by a unique address. Good old straight binary seems to be the standard and best way to reach a single location in the address space. For convenience, this binary address is normally called out in the form of a hexadecimal word.

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 row and column cross on the selected bay is the box we are after. Mathematicians would call a two-or three-dimensional grouping of boxes a **matrix**.

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

Something like this .....



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

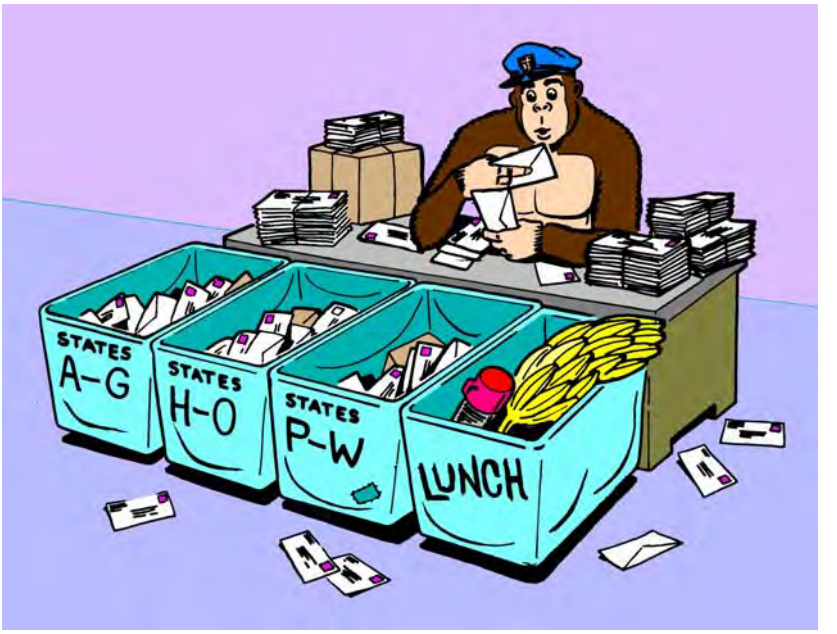
To review, the address space is the reach of a microprocessor's CPU that calls out the maximum number of places into which you can put RAM, ROM, 1/0, or nothing at all. On an 8-bit micro, each of these locations can hold one 8-bit data word. This data word can be anything you like and used anyway you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

> 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.

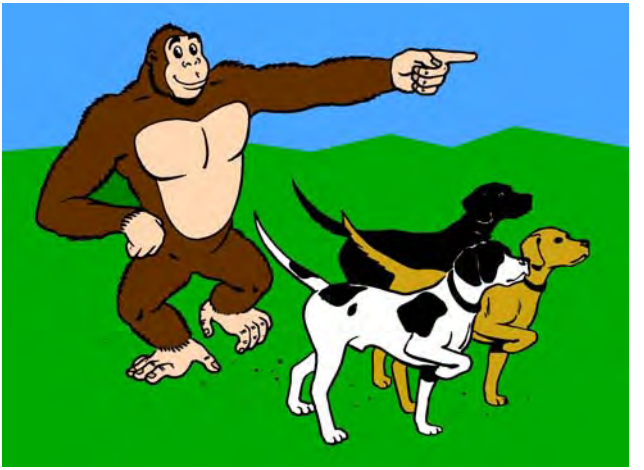
There are two popular widths of index registers. An 8-bit index register can only count down from 255 or reach 256 locations in a file from a given starting address. The X andY registers of the 6502 family are typical. You can also have 16-bit wide index registers,

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 that can serve either intended or dedicated uses, depending upon how 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.