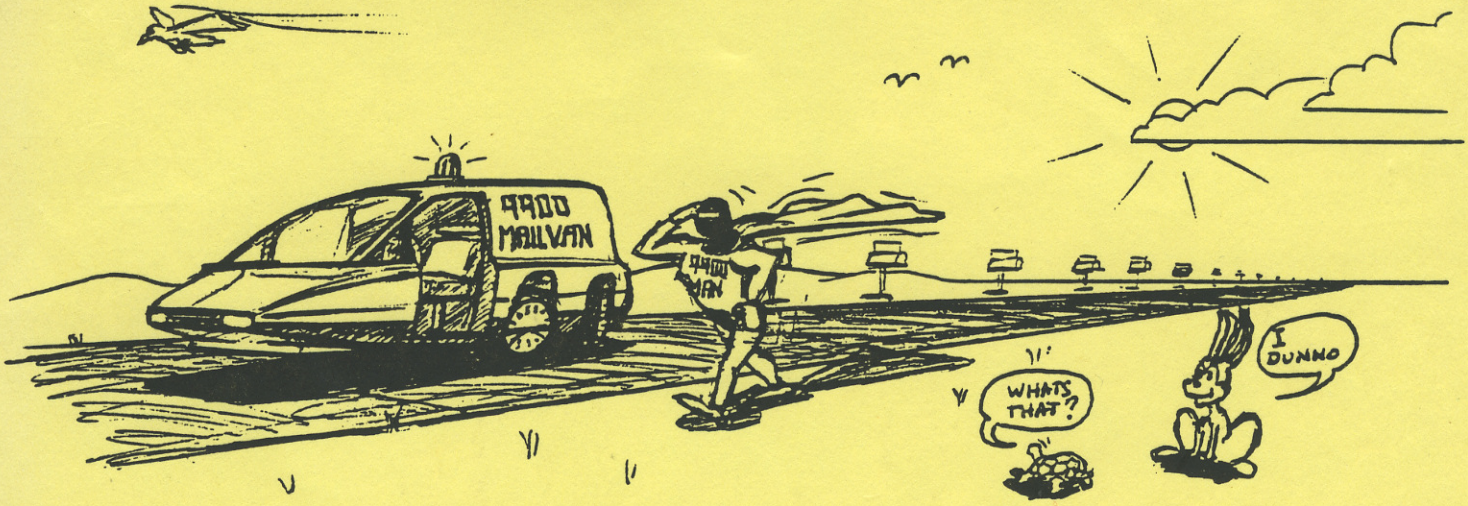


Easy Assembly

A collection of articles on TMS 9900 Assembly Language for the TI 99/4A

By Bob Webb

2-15-92

A S S E M B L Y L A N G U A G E

OH NO! What is this article doing in POMONA VALLEY 99ER NEWSLETTER? Well, I am a new member to your Club. My name is Bob Webb and I work on electronically controlled machines for Bell and Howell. My Wife Julie and Son, two year old Michael, are allowing me this time to enjoy my favorite hobby. This brings me to Assembly Language. My hobby has been, for many years, to try and unravel the ins and outs of this obscure part of our 99/4A's.
This language took me 3 solid years to learn. I jumped in to Basic and Extended Basic pretty quick with no real pain.
Since these languages posed no real problem I bought the Mini-Memory module and proceded to study Assembly Language.

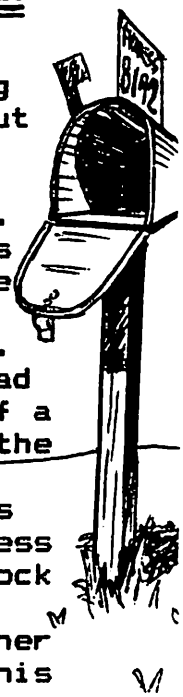
O U C H !

That brick wall was hard!
What in the HECK were they trying to say in that manual?
The module remained in a box for about a year after that.
Simple English was not a high priority to those Manual Writers.
So, I decided to move on and continued to write in extended basic with all of my friends.
But, I could not help wondering what else was in that Silver and Black Console. I gave it another try. The Secrets in the IC Chips were too intriguing to ignore.
My Mini-Memory module was dusty but it worked the first time I plugged it in. The LINES program was more bizzare looking to me now due to my experience in writing Basic programs. How does that program work? I could not concieve of a way to emulate the same thing in Basic. It was doing amazing things!

After buying every Book I could find on Assembly Language for the 99/4A and studying them, the Fog began to lift.
Three years later I am still in the dark as to many details regarding the CRU and Peripherals.
But, You cant confuse me on what else is inside our Silver and Black Consoles.

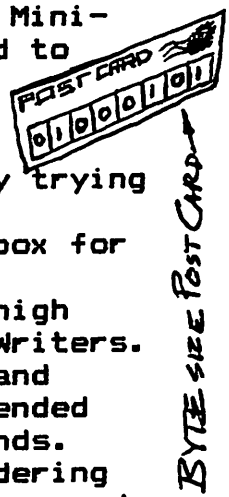
With my help I know I can save you years of frustration by presenting a simple sort of Road Map of our computers innards.
I dident have this kind of help.
My intention is not to teach the Language to you. I think it is more important to first explain what the language is and how it controls your machines hardware (Keyboard, Memory, T.V., Etc.).

THIS WILL BE LESSON NUMBER ONE.

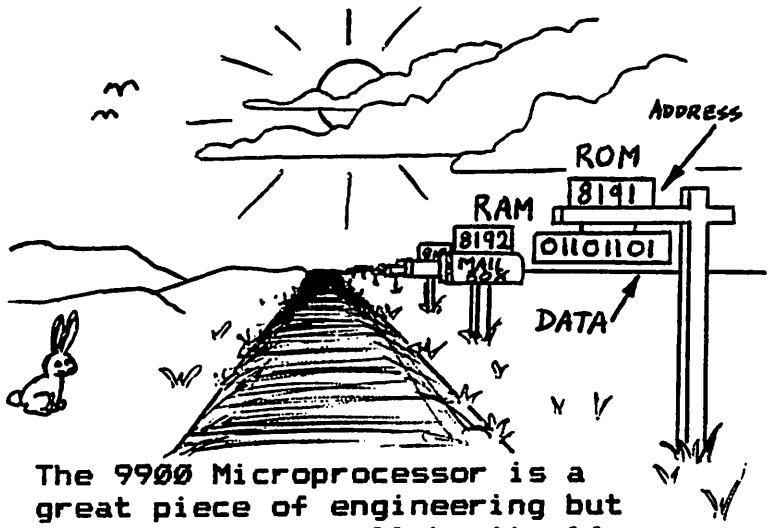


Inside your console there are many chips. It looks confusing and near impossible to sort out what chip does what.

Dont think about it like that.
Your 9900 microprocessor looks at all of those chips from the inside and all it sees is a single, One Lane Country Road.
No one else drives on this road but him. Without the threat of a collision he drives at about the speed of light.
He wears a watch and only goes down the road to another address at certain intervals. This clock controls the timing of his movements. There are a few other devices that are located on this road at specific addresses.
Those devices will be explained later.
This paved country road has a Mail Box at nearly every address.
Each Mail Box can hold one BYTE of information. No more, No less.
As you may know Computers operate with a language called Machine Code. We Speak English, our TI Speaks in Machine Language.
The Alphabet of Machine Language is only 2 characters long.
ZERO 0, and ONE 1.
This Alphabet is known as the BINARY numbering system.
BI meaning two. Each of the Boxes contains a Post Card. It will hold a BYTE of data. This means it has 8 Squares drawn on it. Each square will hold one ZERO 0 or one ONE 1.
When you turn on your computer all of the Post Cards in all of the Mail Boxes have ZERO's drawn in all of the Digit Places or squares.



The Mail Boxes are all on one side of the road. The other side of the road is nothing but a lush green field. We will only need to concentrate on the side with the Mail Boxes. These Boxes are real. They do exsist and are known as parts of RAM chips.
We all know our program will evaporate when the computer is turned off. On the Post Cards, the ONES and ZEROS are like 8 light bulbs, either on or off, with one bulb in each of the digit positions. If you turn off the power, all the bulbs go out in the Post Cards, in all Boxes. The information that those bulbs represented is lost forever because they were not saved in some way. We save our collection of ONES and ZEROS with our Disk Drives and Cassette recorders.
RAM stands for Random Access Memory. The 9900 Microprocessor can randomly read or write to those locations.

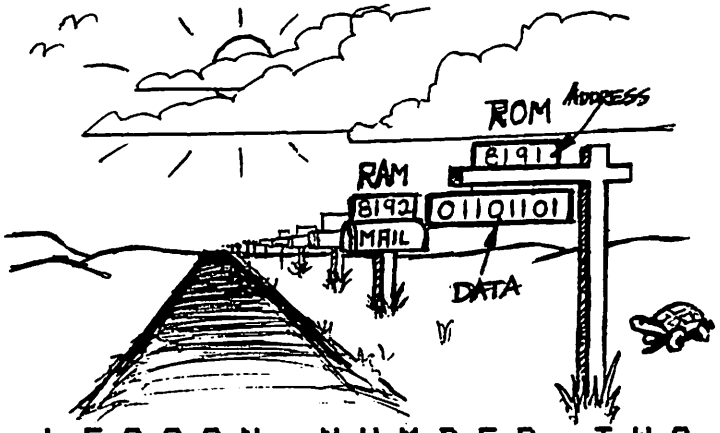


The 9900 Microprocessor is a great piece of engineering but it is very dumb all by itself. It needs to follow a list of instructions to do any task. Even when you first power the Console up it needs to have some program to follow or it will just sit there waiting for its first instruction.
Where is its first instruction located? We know that there is only one road. We know that there are mail boxes along the way. But how many Addresses are there and where do they begin?

They begin, quite naturally, at Zero. The First Address on the road is Zero, the Second is One the Third is Two and so on.
When your Console is first turned on the 9900 Microprocessor looks for the data at address Two on that country road.
It knows to look at Address Two and use the Binary number there as its first address to look for its program. Much like a GOTO statement in Basic. TI did this first because changes are made durring production when mistakes are found in earlier Consoles. Later Verions of this program might jump to another address in memory. This is called a VECTOR TABLE. But wait, the Mail Boxes contents are zeroed out each time power is removed. That is right. So, that is why ROM is needed.
At Address Two, the Mail Box is replaced by a sign. The sign is painted with permanent ink. In our Consoles Addresses Zero through 8,191 have signs placed where the Mail Boxes normaly reside. Texas Instruments placed those signs there. This is the so called BOOT STRAP program. The 9900 Microprocessor must pull its boots on before it can walk. The 8,192 signs along the road do things like clear the screen and place the familiar color bar picture up. Part of Basic resides there as well(more on Basic later).
ROM stands for Read Only Memory. Texas Instruments made about seven versions of this ROM.
The One Lane Country Road has Addresses starting from Zero going all the way up to 65,535!(With 32K)
8 X 65,536 = 524,288
If each Mail Box holds 8 digits, and there are 65,536 Mail Boxes, that means there are over a half a million ZEROS and ONES inside our Consoles!
What else is along this road?
What is GROM and GRAM?
What do they mean when they say we have 16K of RAM for our programs?
Answers to these and many other questions in later lessons. BYE!

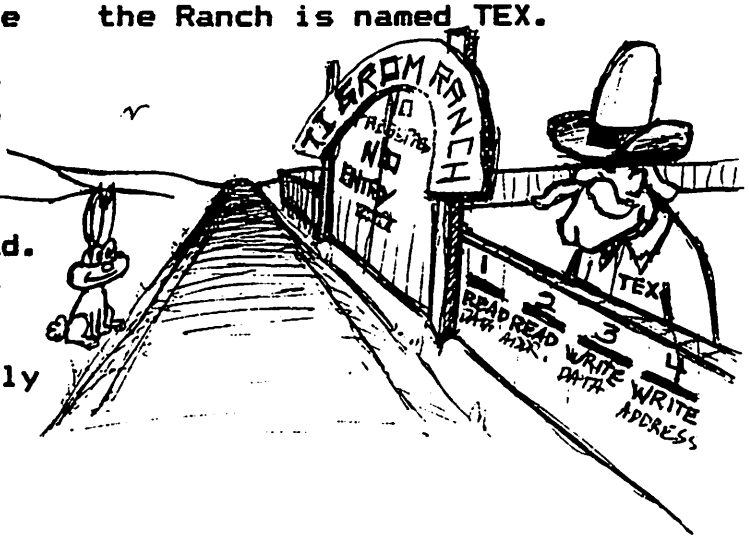
A S S E M B L Y L A N G U A G E © WEBB 1991 =:-)

Hello again. Last months lesson was real simple. This time I'll dig a little deeper. Now, we all know that our 9900 Microprocessor does not actually travel down an Old Country Road. But, for our discussion I think it will make other points clear if we continue to use this analogy.
If you have any questions, or want to enlighten me on any points, please write. If you include a self addressed, stamped envelope I will try to write back in a prompt manner.
Write to: BOB WEBB
P.O. BOX 3023
ARCADIA, CA. 91007

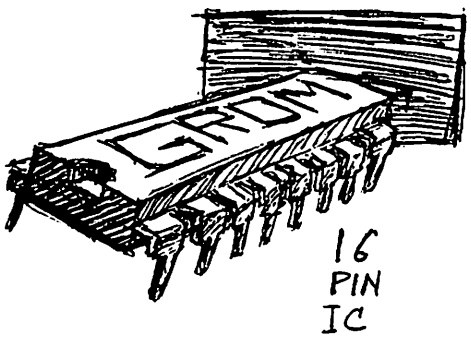


L E S S O N N U M B E R T W O

As you will recall, our 9900 CPU has a clear and simple view of the insides of our Consoles.
He looks at all of the Integrated Chips of our computers from the inside and see's only a Long One Lane Country Road.
The addresses, on only one side of the road, range from Zero to 65,535 (when we have our 32k memory installed).
The first 8,192 addresses have signs placed along side the road. These signs have 8 bits of data painted on them with permanent ink. This first 8k block of memory is known as ROM (read only memory). ALL PROGRAMS and DATA inside the computer are in Machine Code. The program stored in this ROM area is in Machine Code, zeros and ones.
ROM, represented by signs at each of the first 8,192 addresses was written by Texas Instruments in Assembly Language.
The BASIC Language built into our consoles is stored in Machine Code But, it was written by TI in a Language called GPL, or GRAPHICS PROGRAMMING LANGUAGE. In order for us to be able to run this BASIC Language program we must have a built in GPL INTERPRETER program.
The actual BASIC Language program is stored in special IC Chips, isolated from our CPU, called GROM.
The first program that is executed in any computer is known as the BOOT STRAP program. Our computers BOOT STRAP program is located in this first 8k of ROM.
This Machine Code program was written in Assembly Language.
It is a GPL INTERPRETER program that reads programs written in GPL. All GPL programs must be stored in those special IC Chips called GROM Chips. In order for our 9900 CPU's to be able to read those GPL programs it must have a way to communicate with the GROM Chips. Texas Instruments built a huge Texas size Ranch along that Little Old Country Road. On that big Ranch they placed 3 GROM's. This Ranch is so big that there are 4 MAIL BOXES. The Owner of the Ranch is named TEX.



Old TEX named it GROM RANCH.
The console BASIC is written in GPL and is stored in GROM Chips number 1 and 2. GROM Chip number Ø has the real BOOT STRAP program that has all of the initializing routines, start up TI COLOR BAR screen and MODULE SELECTION LIST screen. GROM RANCH can be thought of as a second road with its own address space. This road starts at GROM Address zero and goes to GROM Address 24,575 in the first three GROM Chips.
The Module Port in our Consoles is officially called a GROM Port. That is because Texas Instruments Modules can contain one or more GROM chips as well. This extends the GROM address space.
So we can see that GROM RANCH can be modified simply by putting a Module in the Port. A module can have up to 5 more GROM Chips inside. The Editor Assembler Module has only one GROM Chip. But the EXTENDED BASIC Module has 5 more GROM's.



Why did Texas Instruments invent GPL and GROM's? (GROM RANCH). An Assembly Language Operating System or BOOT STRAP program would have been simpler and cheaper to produce. G R E E D !! This was done so that they could charge other companies for the right to use the GPL Language for their own Game Modules. This idea backfired. This type of system is called Closed Architecture.
Other computers had an Open type Architecture and thrived. APPLE Computers had an Open System as well as IBM. This one element of our computer along with some other odd marketing ideas killed any chance for our computer family to grow. Software Companies did not want to pay the high royalty, or licensing fee's. So, they did not write many programs for our machines. Or, if they did write programs they were written in Assembly Language and bypassed the GPL interpreter. This made TI very angry. So, they Closed the Architecture even more. They came out with the Version 2.2 Console. If your Game Module did not have the special GROM Chips it would not run. MEAN OLD MR. TEX!
Their own need to control every aspect of the 99/4A finally wiped out their chances at success.

$ EXCESS PROFITS BREEDS RUINOUS COMPETITION.

Well, that's enough about that. Let's study GROM Chips a bit (Pun Intended).
GROM Chips are like ROM. They are permanent Read Only Memory Chips. In fact, GPL Language is almost identical to Assembly Language. It differs in one main aspect. The GROM Chips AUTO INCREMENT themselves as they are read. As the CPU fetches data out of GROM RANCHes Mail Boxes it does not need to tell the GROM what GROM Address space to read next. The GROM automatically fetches the data from its own next address. The GPL interpreter program causes the CPU to read the binary data in the GROM's, READ, MAIL BOX. The GPL interpreter reads the GROM program from the Mail Box at GROM RANCH and only stops when the GROM program code includes some type of jump, or goto type statement.

Normal ROM Chips do not Auto Increment. You must tell the CPU what address you would like to access almost every time you give it an instruction.
This wastes time if you are only going to read the data in the chip. The CPU must make that many more trips up and down the road collecting the next address to go to. So, GROM's are quick.
GRAM Chips are like RAM Chips. They Auto Increment just like GROM. I understand that TI never used any GRAM Chips in the Console or Modules of the 99/4A.
We Programmers can do some fun things with the GROM Chips. The COLOR BAR TITLE SCREEN uses its own large style Character Set.
With some programming savy we can use that Character Set in our own Assembly Language programs. We write a Post Card or two to TEX at GROM RANCH and ask him to send out that Character Set through the GROM READ DATA MAIL BOX.
Or if we want to use other parts of the BOOT STRAP program all we have to do is get a copy of the German Book called "TI 99/4A INTERN", by Heiner Martin. It is a copy of the Assembly Language GPL Interpreter program in ROM. It also includes the GPL programs in the first 3 GROM Chips in our Consoles. When I bought it I did not know what it was. I buy TI books on impulse. But later, when I began to understand more about Assembly Language it gave me deep insight into our machines ROM and GROM Operating System.

(C) 1985 by Verlag fur Technik und Handwerk Gmbh,
D-7570 Baden-Baden,
Postfach 11 28, West Germany
(ISBN 3-88180-009-3)
Printing: F.W. Wesel, Baden-Baden

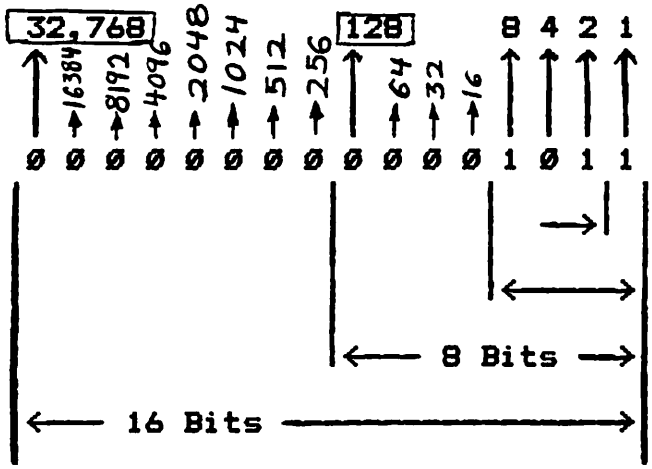
This is not light reading. It is a lot of raw code and a little commentary. But if you are a nut like me and want to know what is in our Consoles than this is for you. We can not alter anything in GROM. Remember it is just like ROM We can only read it. But there are great subroutines built into GROM that we can use.
We communicate with GROM RANCH in BINARY CODE written on POST CARDS. Each Post Card holds combinations of Zero's and One's representing Data or Instructions for TEX.
8 bits, or rather a Byte, of data fits on one Post Card. The CPU is like a lightening quick Mail Man. He delivers Post Cards encoded with data to and from GROM RANCH. The CPU and TEX do not talk to one another directly. They only use the Post.
Here is a description of GROM RANCHes 4 Mail Boxes.

CPU ADDRESS WHERE MAIL BOX IS LOCATED.	GROM FUNCTIONS ASSOCIATED WITH THE MAIL BOX AT THAT LOCATION. TEX IS ON THE OTHER SIDE OF THE FENCE.
38,912	= GROM/GRAM READ ADDRESS
38,914	= GROM/GRAM READ DATA
39,936	= GROM/GRAM WRITE DATA
39,938	= GROM/GRAM WRITE ADDRESS

If you write to GROM nothing will be written. Only GRAM can be written to. So, the Write Data and Write Address Mail Box would be used only if you found a way to install GRAM in the Console or the GROM Port.

Now, if the Editor Assembler, Extended Basic, or Mini Memory Module is installed in the GROM Port this is our first chance to access the 9900 CPU's Address space directly. Most of the time our computers are speaking GPL. Now it's our turn to speak!

The 9900 CPU only understands Machine Code. The position of the ZERO or ONE in an eight Bit Byte represents a given amount. Just like our Decimal System does. In the Decimal numbering system 11 means eleven. in Binary 11 means three! Why is that? The first digit, or Bit, in the first position, on the right, represents one in both the Decimal and the Binary system. The second digit in Decimal represents one group of ten ones. The second digit in Binary represents one group of two ones. So if a number one is in the second position, in Binary, it represents two. If a zero is in the second position it means no group of two, or zero(of course). In the Binary System each position to the left is double the amount just before it. So, this number in Binary means 11:



8 + 2 + 1 = 11

Now you are asking why did he draw 16 Digits? He told me last time that the Post Cards in the Mail Boxes, and Painted Signs, only held 8 digits, or Bits. Yes! Each address along the road is capable of holding only one Byte, meaning 8 Bits. The reason I drew 16 Bits of data is because the 9900 CPU is capable of reading 16 Bits, or a WORD, of data at a time. This is where the term 16 Bit Microprocessor comes from. The CPU can read a Byte of data or a Word of data upon command. Most of the time we will talk about a Word of data because more work gets done in a shorter period of time. If our CPU was only capable of reading a Byte at a time it would be called an 8 Bit Microprocessor (another name might be Commodore VIC-20). Wait a minute Bob! You said each address only holds a Byte of data! Yes. If you tell the CPU to grab a Word of data from address 2, it studiously goes to address 2 and picks up its first Byte. We will call this Byte the MOST SIGNIFICANT BYTE, or MSB. Then he moves automatically to address 3 and grabs the next Byte. We will call this one the LEAST SIGNIFICANT BYTE, or LSB. When the CPU reads ROM or RAM the contents at those addresses are not harmed or altered. The CPU mearly reads the data and stores it elsewhere.

The CPU always thinks of this Word of data as having the MSB on the left and the LSB on the right, unless you tell it otherwise. So, our Word of data looks like this:

```
Address 2 and 3
Binary                  Dec
0000000000100100 = 36

MSB      LSB
00000000 00100100
Address2 Address3
```

In fact all of memory is used by the CPU this way. It Thinks and acts in terms of Words. I think we now understand why our memory stops at the 65,536th memory address. The CPU can only look at 16 bits at a time. If every one of those 16 Bits were one, or on that would represent the number 65,536. 65,536? Yes! Remember the first address is zero. So, there are exactly 65,536 addresses!

Boy, Oh Boy! What does all of this mean? It means Machine code is a pain in the posterior for mere mortals like you and me. I think it would be wise to buy a $20. Scientific Calculator if you want to experiment at all. I use the Texas Instruments TI-35 PLUS. It converts Decimal to Binary and back. It also converts Decimal to Hexadecimal. This is a Base 16 Numbering System, just like Decimal and Binary. The reason I bring this up is because most Assembly Language requires that you write your programs in this format. It is not hard to learn. And in a short while you will begin to think in Hexadecimal just like a real nerd!
Hexadecimal is like Decimal but instead of counting from one to ten you count from zero to fifteen. For a total of 16.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	HEX
										11		13		15		DEC
1	2	3	4	5	6	7	8	9	10		12		14		16	

If you write numbers in Hex how do you know its Hex and not Dec? You put a greater than symbol in front of it like this:

>0024

If you write in Binary you will usually put a lower case b behind it like this:

0000000000100100 b

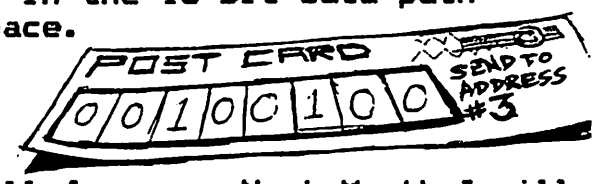
If you write numbers in Decimal you will write it like you have since the first grade:

36

The sharper people out there will have already noticed that all three of the numbers above are the equivalent to Decimal 36.

Decimal 36, or Hexadecimal >0024 happens to be the number at address 2 and 3 in most of our Consoles, if not all of them.
This is the beginning address of the GPL Language program in ROM. Our CPU's use this address like a GOTO statement in BASIC. When we first apply power to our consoles the CPU performs a LEVEL ZERO INTERUPT which is a RESET. Then it reads the data at address 2 and 3, to collect a full Word, and JUMP's, or GO's TO, Address >0024. This is all well and good but why does it start at the third address in memory and not 0? It does. I was saving this until I thought you could follow along.

-------> Address ZERO and ONE contain the Hex number >83E0, in all Consoles. This number is also an address. Beginning at this address, >83E0, is another block of memory that the CPU sets aside as a Scratch Pad, or Workspace. It needs an area to store numbers and addresses while it performs calculations and operations on them. The Block of memory is 16 Words long. I will be going into detail on this little workspace later. Just know that for now there is a 16 Word Workspace set aside by the CPU in an area of the fastest RAM in our Consoles. This RAM has a 16 Bit data path. All of the other blocks of RAM are in our 32k cards in the Peripheral Expansion Box and are sent down the P E Boxes Firehose connector in an 8 Bit data path. There is no way to change this. It is hard wired this way. Our computers can perform MATH quickly in the 16 Bit data path Work Space.



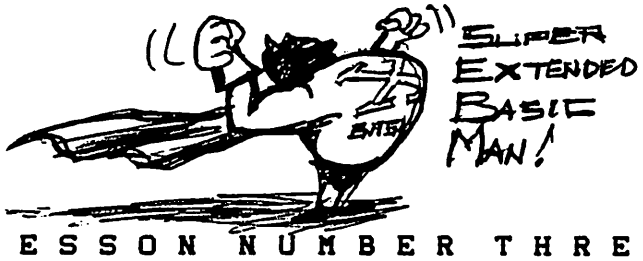
Thats all for now. Next Month I will get to the really fun stuff! VIDEO DISPLAY PROCESSOR RAM!!!! BYE!

A S S E M B L Y L A N G U A G E

 =:')

Howdy doo! This will be the last formal lesson on memory mapping. Next month we will use all that we have learned about the insides of our computer and examine the 9900 CPU!
If you have any questions, or want to enlighten me on any points, please write. If you include a self addressed, stamped envelope I will try to write back in a prompt manner. Write to:

BOB WEBB
P.O. BOX 3023
ARCADIA, CA. 91007



L E S S O N N U M B E R T H R E E

How can we look at the data stored in the Mail Boxes and on the signs? We could write an assembly program that does that. But we do not know how to write one. So, we must use a language that we know.
Extended Basic has two commands, CALL PEEK and CALL LOAD that can access the data along the Old Country Road. CALL PEEK allows us to look at all of the data from address >0000 to >FFFF. These are the hexadecimal numbers that represent address 0 to address 65,535 in decimal form. Remember that the greater than symbol ">" means that the number is in HEX format. CALL LOAD allows us to write data to those Mail Boxes along the old country road. We can only access the data on the private road on GROM RANCH if we communicate with TEX through his four Mail Boxes. If you want to experiment with CALL PEEK or CALL LOAD remember that you can do no harm to the chips inside your console.

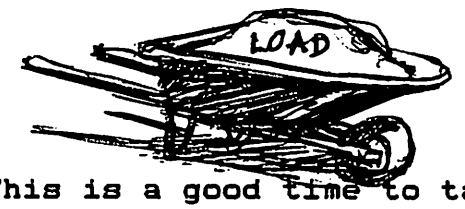
You can do real damage to disks that are left in a drive if you accidentally write to them. You may want to remove any from from the drives when you begin experimenting. After the disks are removed you can play around inside your computer all day.
The easiest and safest thing to do first is to read the data that is there. To do this you use the CALL PEEK command.
The most important thing to remember is that to access any address larger than 32,767 you must subtract 65,536 from it.
Lets say that I want to read the data stored in ROM at address ZERO, I would write this line into my Extended Basic Program:

```
100 CALL PEEK(0000,X)
110 PRINT X
```

Lets say that I want to place a value of 60 in address 41,215. This address is larger than 32,767 so we do this:
41215 - 65536 = -24321
Our program would then look like this:

```
100 CALL LOAD(-24321,60).
```

Armed with this information and our Memory Map we can find out what area of memory we are changing when we use all of those useful call loads.



This is a good time to talk about the GROM port. The GROM port also has a direct link to our country road.
From Address >6000 to Address >7FFF is an area set aside for assembly language programs. Modules from Companies other than Texas Instruments use this area for their programs.

Some modules have no GROM chips in them. I wrote about that last month. This is the area set aside for those programs. The older consoles will run a program in a module that is strictly assembly. The Version 2.2 Consoles will only run a program in this area if the module also has a GROM chip as well as the program ROM chip.
When no module is in the GROM port this area is all zero's. No chip is there, so you can not read or write to anything in this space.
Most of the time a module has only ROM. However, the MINIMEMORY module has RAM and ROM. With this module in the port you can write assembly language programs that reside between address >7000 and >7FFF. This 7k byte RAM area is called MEDIUM MEMORY. Why is that you ask? Well, because there are other RAM area's when you have your 32k card installed. These 2 RAM area's are called HIGH MEMORY and LOW MEMORY. Most Assembly programs are written for the HIGH MEMORY area and the EDITOR ASSEMBLER MODULE. The HIGH MEMORY area is between address's >A000 and >FFFF. You can write assembly programs for the MINIMEMORY module that reside in this HIGH MEMORY area. The LOW MEMORY RAM area is used for programs and something called the REF\DEF TABLE. This is where an assembly language program is named and an entry point for the program is stored.
All assembly programs must have an Entry Point stored in a directory so that the CPU can find it and then start executing the first commands. The MINIMEMORY module has its own REF\DEF TABLE located at the bottom of the RAM area.
With the Line By Line Editor installed you will find 3 entries in the last few bytes of RAM.
You will find the ASCII equivalent of the words LINES, OLD and NEW.
Between these program Entry Point names will be a WORD of data. This WORD is the Entry Point Address to the program listed in the TABLE.
When you write an assembly language program for the MINIMEMORY module you must install the Name and Entry Point to your program.

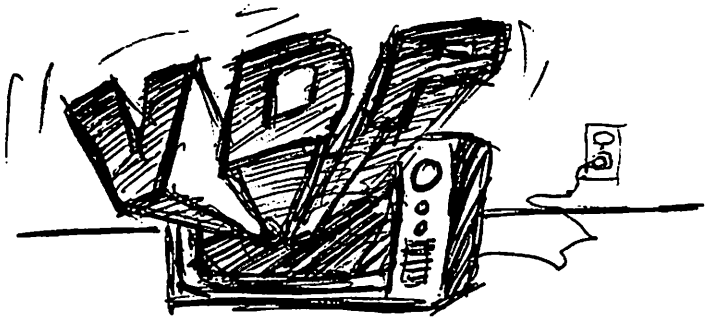
The MINIMEMORY manual is not clear on this point and I hope this will help you.
Speaking of help, 2 books have helped me the most. The first is called:

Fundamentals of TI-99/4A
Assembly Language
by, M.S. Morley
published by TAB books inc.

The second is called:
Introduction to Assembly Language for the TI Home Computer
by, Ralph Molesworth
published by Steve Davis publishing.

These two books helped me a great deal. They both offer a good set of lessons but they both lack a little something. Together they form a good beginning.
The EDITOR ASSEMBLER manual is mandatory if you want to get serious about the subject.
I actually like reading the manual now that I understand what everything is.
With the small amount that you have learned here you too can probably enjoy it as well.

The sound chip write address is >8400. The Memory Map on the other page shows that the sound chip write address goes from address >8400 to address >87FF. Only >8400 is actually used. The rest of the area is not used. It has not been Decoded. Texas Instruments, for reasons only it knows, wasted the rest of this space. Other areas in memory are the same way. The Speech Module Read Write addresses are also not decoded fully. 768 BYTES are wasted between address >8000 to >82FF. This area contains 3 identical copies of the SCRATCH PAD RAM that is used by the CPU as its personal workspace. Nothing can be done with it. Programs can not be written here. Pages 404 through 406 in the Editor Assembler Manual describes the Scratch Pad area.



The last and most interesting part of our Computers, as far as I am concerned, is the area along the old country road called the VIDEO DISPLAY PROCESSOR. It is very similar to the GROM RANCH. But we write to this area all of the time. The VDP RAM area is located along the road and is accessed just like the GROM area. There are 4 Mail Boxes identical to GROM RANCHES.

CPU ADDRESS WHERE MAIL BOX IS LOCATED.	VDP RAM FUNCTIONS ASSOCIATED WITH THE MAIL BOX AT THAT LOCATION.
>8C02	VDP READ/WRITE ADDR.
>8C00	VDP WRITE DATA ADDR.
>8802	VDP STATUS (MSB)
>8800	VDP READ DATA ADDR.

The VDP RAM area is like another road going off at a 90 degree angle from our country road. Once again we have to communicate with the RAM area through these Mail Boxes. The STATUS Mail Box is new to us. But its name gives us a clue to its function. The VDP is actually another CPU. Its use is dedicated to creating and presenting images to the Television or Monitor Screen. The VDP RAM road starts at address >0000 and goes to address >3FFF. The addresses from >0000 to >02FF is the actual image on the screen. If you change a byte in here you will automatically change a spot on the screen. The rest of the area is for your character shapes, Basic program, color of the characters, and other CPU and disk drive housekeeping. Study the Memory Map and next month we will look at the CPU BRAIN!

A S S E M B L Y L A N G U A G E © WEBB 91 MEMORY MAP

CONSOLE. Address		Description	PE BOX. & PORT
>0000 >1FFF		CONSOLE ROM	
>2000 >3FFF		LOW MEMORY RAM 1/2 OF 32K CARD 8K BYTES	
>4000 >5FFF		PERIPHERAL EXPANSION ROM AREA.	
>6000 >7FFF		MODULE PORT ROM/RAM AREA	
>8000 >82FF		3 COPIES OF SCRATCH PAD RAM (wasted space)	
>8300 >83FF	>8300 >83FF	SCRATCH PAD RAM AREA TO	
>8400 >87FF	>8400	SOUND CHIP WRITE ADDRESS	
>8800 >8802 >8BFF >8C00 >8C02 >8FFF	>8800 >8802 >8C00 >8C02	VDP READ DATA ADDRESS VDP STATUS (MSB) VDP WRITE DATA ADDRESS VDP READ/WRITE ADDRESS	
>9000 >93FF >9400 >97FF	>9000 >9400	SPEECH MODULE READ ADDRESS SPEECH MODULE WRITE ADDRESS	
>9800 >9802 >9BFF >9C00 >9C02 >9FFF	>9800 >9802 >9C00 >9C02	GROM/GRAM READ DATA ADDRESS GROM/GRAM READ ADDRESS ADDRESS GROM/GRAM WRITE DATA ADDRESS GROM/GRAM WRITE ADDRESS ADDRESS	
>A000 >FFFF		HIGH MEMORY RAM 1/2 OF 32K CARD 24K BYTES	

A S S E M B L Y L A N G U A G E © WEBB 1991

Hello. This lesson is about the most important part of our machines. The master CPU. I hope you have retained a little bit of the Memory Map in your mind. This lesson will recall some of that material. if you have any questions, or want to enlighten me on any points, please write. If you include a self addressed, stamped envelope I will try to write back in a prompt manner. write to:

BOB WEBB
P.O. BOX 3023
ARCADIA, CA. 91007

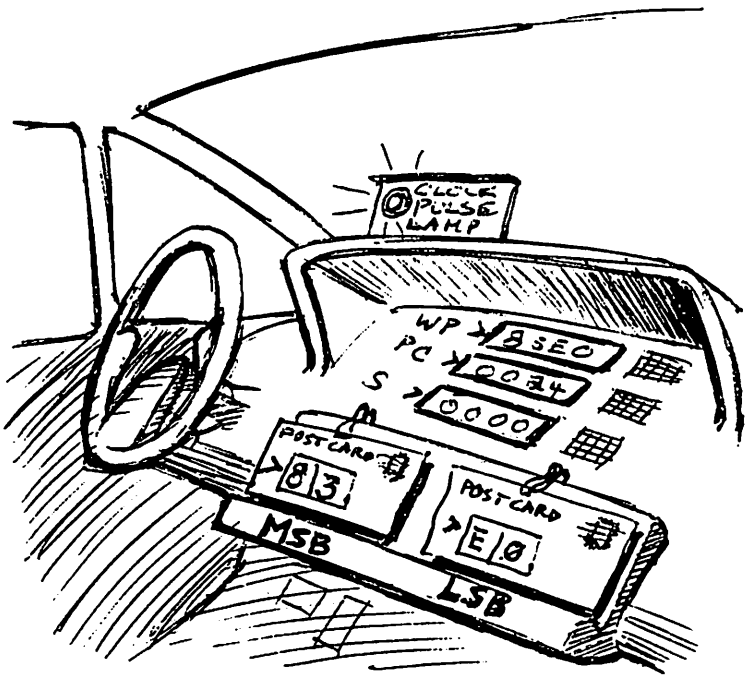
He comes from a proud family of Microprocessors and takes great pride in that heritage. He has younger Cousins now that have much faster Clock speeds and larger Data Busses but none the less holds his head high. He has proven his worth to about 3 Million consumers. 9900 MAN has a Rigid set of rules he lives by. The first rule is that he must follow his Master Clock at all times. Everyone in this world of his must do the same. On the Instrument Panel in the VAN is a Master Clock Pulse Indicator. Every time the Master Clock Ticks that Indicator Light flashes and 9900 MAN tromps on the throttle to reach his next destination address.

L E S S O N N U M B E R F O U R

The sleek, ELECTRON MAIL VAN, depicted above is the vehicle that our Hero, 9900 MAN, drives. you will note that it is not unlike Mail Vans found all over the United States. It has the open slide door so that 9900 MAN can reach into the Mail Boxes with little effort. However this beauty travels at near the speed of light. Our Country Road is private. So, he is the only one that tears along this route. He loves his vehicle and always keeps a full tank of electrons.

Besides the Master Clock Indicator he has 3 other Major Readouts. These readouts in reality are known as the 3 Hardware Registers. 9900 MAN pays strict attention to these 3 Readouts.
Here are the names of the readouts:
1. PROGRAM COUNTER REGISTER
2. WORKSPACE POINTER REGISTER
3. STATUS REGISTER
Each Readout is 16 Bits long, or 2 Bytes, or a Word. We all know that 9900 MAN reads only BINARY but we will think of them as being in HEXADECIMAL. Lets recall some of the information in our past lessons on the Memory Map.

On the Instrument Panel each of the readouts has a keypad next to it so that 9900 MAN can enter the new number. There are two clip boards that hold 2 Post Cards. This is so that 9900 MAN can write new Post Cards. You see, he writes most of the Post Cards himself. He then puts them in their proper Mail Box.



In lesson number two we found out that when you first power up the Console the CPU (also named 9900 MAN), performs a LEVEL ZERO INTERUPT which is a RESET. The term SET means to make data at an address a 1. The term RESET means to make data at an address a 0. So, this LEVEL ZERO INTERUPT forces 9900 MAN to race along ALL RAM Mail Boxes and place Post Cards with all ZERO's on them inside. After all RAM memory is RESET 9900 MAN always looks for his first instructions at address >0000, >0001, >0002, and >0003. The first two addresses contain the Most Significant Byte and the Least Significant Byte. 9900 MAN is a 16 Bit CPU so he always takes a full WORD of data each time he performs a fetch.

The first WORD of data at addresses >0000 and >0001 is, >83E0. 9900 MAN knows that the first word of data he collects is an address. This address is the first address of his Scratch Pad area. He always needs to have a place in memory set aside so that he can make notes for himself and perform math. This area is known as the WORKSPACE REGISTER AREA. It is 16 WORD's in length which means it starts at address >83E0 and continues to >83FF. This is how it would look:

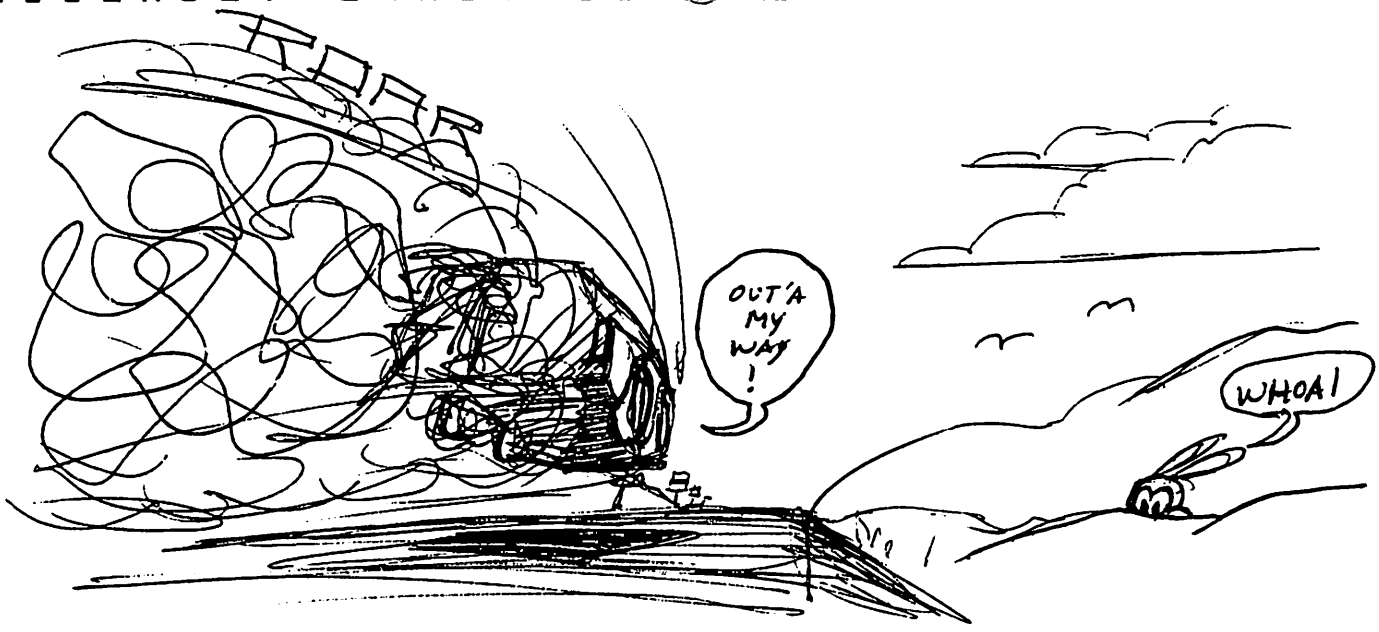
```
>83E0 MSB 1ST WORD.    REGISTER >0
>83E1 LSB
>83E2 MSB 2ND WORD.    REGISTER >1
>83E3 LSB
>83E4 MSB 3RD WORD.    REGISTER >2
>83E5 LSB
>83E6 MSB 4TH WORD.    REGISTER >3
>83E7 LSB
>83E8 MSB 5TH WORD.    REGISTER >4
>83E9 LSB
>83EA MSB 6TH WORD.    REGISTER >5
>83EB LSB
>83EC MSB 7TH WORD.    REGISTER >6
>83ED LSB
>83EE MSB 8TH WORD.    REGISTER >7
>83EF LSB
>83F0 MSB 9TH WORD.    REGISTER >8
>83F1 LSB
>83F2 MSB 10TH WORD.   REGISTER >9
>83F3 LSB
>83F4 MSB 11TH WORD.   REGISTER >A
>83F5 LSB
>83F6 MSB 12TH WORD.   REGISTER >B
>83F7 LSB
>83F8 MSB 13TH WORD.   REGISTER >C
>83F9 LSB
>83FA MSB 14TH WORD.   REGISTER >D
>83FB LSB
>83FC MSB 15TH WORD.   REGISTER >E
>83FD LSB
>83FE MSB 16TH WORD.   REGISTER >F
>83FF LSB
```

Bob, why did you say REGISTER after each WORD of data? This is a way of naming each WORD of data stored here. When you write an Assembly Language program, you will want to tell the CPU to add REGISTER 0 to REGISTER 1.

This WORKSPACE REGISTER AREA is where you will spend a lot of time. All programs use this area. All Math and Logic functions are done here. So, after the reset is performed 9900 MAN picks up >83E0 at the first 2 addresses and puts that number into his Instrument panel WORKSPACE POINTER readout. You can see in the illustration that >83E0 is now always on display in the WP readout. This is to remind 9900 MAN where his WORKSPACE is. He can now zip right there without any trouble. The next built in function is for him to fetch the next 2 bytes of data from addresses >0002 and >0003. That number in most Consoles is, >0024. 9900 MAN knows this to be the address of his first instruction. That instruction at >0024 and >0025 (one WORD), is the so called ENTRY POINT in the ROM BOOTSTRAP program. He fetches the number >0024 from address >0002 and >0003 and enters it into his PROGRAM COUNTER readout. Once this number is entered 9900 MAN sits and waits for his first CLOCK PULSE. When the lamp flashes he tromps on the gas pedal (ELECTRON PEDAL?) and runs down the old country road to the address indicated in the PC. He reads the first number painted on the permanent sign and to him it is an instruction. 9900 MAN understands many instructions. these are listed in the Editor Assembler Manual. The instruction is a number of course. 9900 MAN reads the number and knows what it means. As an example lets say that the number tells 9900 MAN to LOAD IMMEDIATE the Hexadecimal number >0017 into WORKSPACE REGISTER >3. 9900 MAN writes the number >0017 down onto 2 Post Cards. He then sits and waits for the Clock Pulse Lamp to flash. When it does he blasts down to addresses >83E6 and >83E7 and places those 2 Post Cards inside them. The MSB Post Card going to the Mail Box at >83E6 and the LSB going into >83E7. When he has finished this task he looks down at the PROGRAM COUNTER Indicator. The PC has automatically incremented itself by 4 addresses. He Turns the ELECTRON MAIL VAN around and faces the other way down the Old Country Road. Once again he sits and waits for the Clock Pulse Lamp to flash. ZAP it goes and he is on his way to address >0028. He has just completed his first use of the WORKSPACE AREA. The number >0017 is now stored in REGISTER >3. You should now be asking me why the PC was incremented by 4 addresses. The Instruction was 1 WORD in length. That accounts for the first 2 addresses. The second 2 addresses was the number >0017. It must be stored next to the instruction. When you write an Assembly Language Program you use the Editor program. This is merely a stripped down version of TI-WRITER. You write this program as a kind of "Letter of Instructions". When you are satisfied that there are no errors in your "Letter" you save it to DISK. Then you start the Assembler Program and it asks for the name of your "Letter", or SOURCE PROGRAM on the DISK. Once you give it the name it starts up the DISK DRIVE and reads your SOURCE PROGRAM. The Assembler takes each line of your program and converts it into MACHINE CODE. That MACHINE CODE is then saved to DISK. The MACHINE CODE file is saved under a name you give it. This file is known as the OBJECT FILE. This is the raw BINARY program that 9900 MAN can read. Here is the ASSEMBLY LANGUAGE line:

```
     LI  R3,>17
```

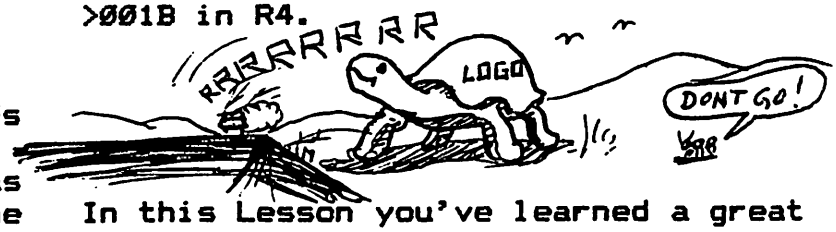
LI=LOAD IMMEDIATE into REGISTER >3 the Hexadecimal number >0017. The Assembler adds the >00 MSB.



Here we sit. 9900 MAN has stopped at the permanent sign at address >0028. Lets continue our Example and say address >0028, >0029, >002A and >002B contain the equivalent of: LI R4,>04
This is the same instruction as before. Only this time we are going to LOAD IMMEDIATE >0004 into REGISTER >4. 9900 MAN writes the number onto 2 Post Cards. Once again, the MSB on the left and the LSB on the right Card. after completing the task the Clock Pulse Lamp flashes and 9900 MAN flattens the gas pedal. He must have great neck and stomach muscles. The acceleration forces time after time must take a toll on him. He screetches to a halt at WORKSPACE REGISTER >4. (Addresses >83E6 and >83E7). Quickly placing the 2 Post Cards into the Mail Boxes he looks down at the PC Indicator. It has been incremented by 4 again. The new address is >002C. The Lamp flashes and he charges to the address indicated in the PC. The instruction he finds on the next set of signs might be the ADD function. Lets say it goes like this:

A R4,R3

The letter "A" means ADD. So, the instruction is telling 9900 MAN to ADD the contents of REGISTER >3 to the contents of REGISTER >4. In this operation the number >0017 will be added to the number >0004 located in REGISTER >4. The number >0004 will be replaced with the SUM of the 2 numbers. >0004 will be lost forever unless we save it in another REGISTER or RAM memory space. In this case we dont care if we lose the number and we allow the SUM to replace it.
The number >0017 in REGISTER >3 will be untouched. 9900 MAN merely reads the number there and makes a note of it on his Post Cards. The SUM would be >001B. The result of this operation would leave the Number >0017 in R3 and >001B in R4.



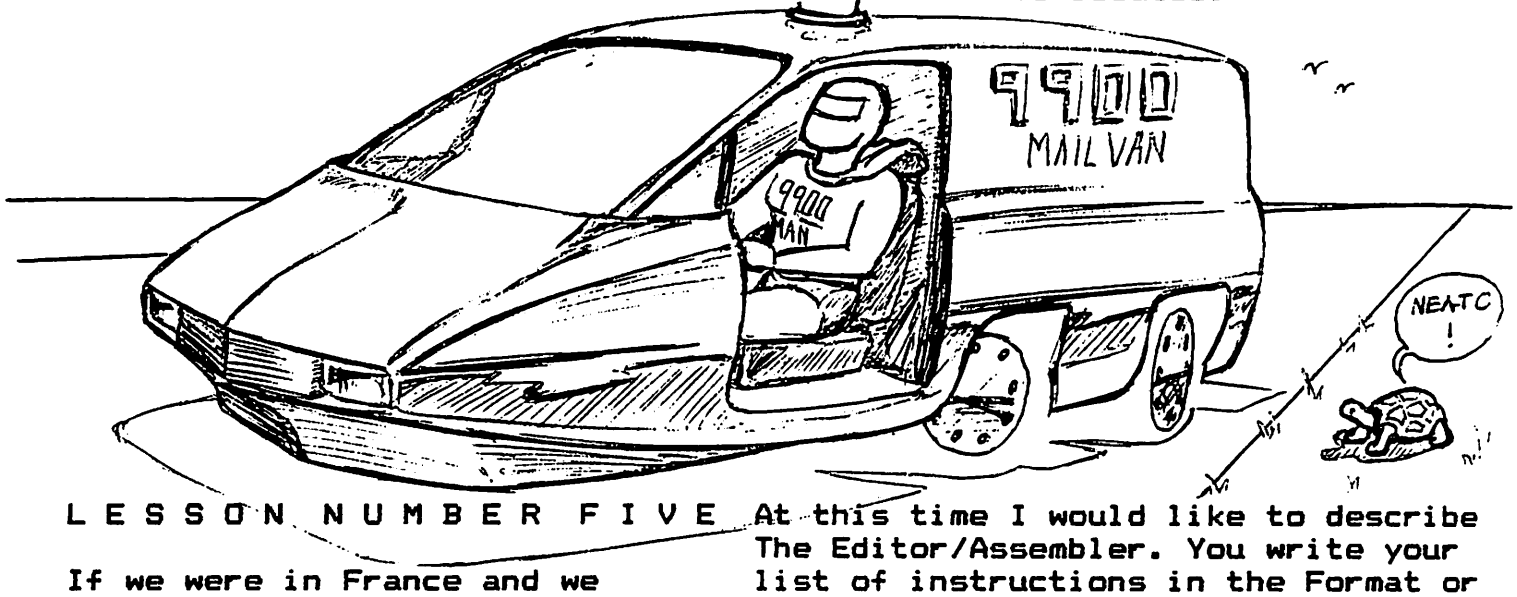
In this Lesson you've learned a great deal about the 9900 CPU.
In next months lesson we will continue discussing the CPU and delve into the format of ASSEMBLY LANGUAGE.
I hope you have gotten something from these lessons.
Respectfully yours,

BOB WEBB.

Howdy doo. I hope you have gotten something out of these lessons. I have been able to reinforce my understanding of these subjects by having to write about them. Once again, if you have any questions, or want to enlighten me on any points, please write. If you include a self addressed, stamped envelope I will try to write back in a prompt manner. Write to:

BOB WEBB
P.O. BOX 3023
ARCADIA, CA. 91007

One way to produce good results is to learn the Computers Language. The best language to learn would be Machine Code. When you learn Assembly Language you naturally begin to understand Machine Code. However Machine code is cold and impersonal (All zero's and ones or Hexadecimal). It lacks the human touch. Assembly Language is a sort of Interpreter between our two worlds. We can write things like JMP SCREEN or TEXT 'HELLO' and the assembler program will interpret that into Machine Code. We dont have to look up Hex codes for the letters in HELLO or the memory address where the label SCREEN is located.



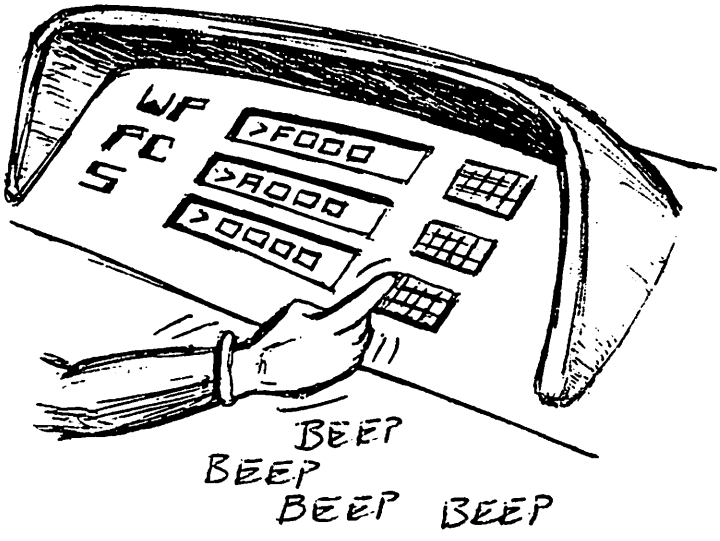
L E S S O N N U M B E R F I V E

If we were in France and we gave a little French child $10Ø to do some errands for us, we should write that list of instructions in French. This would ensure the best results. That child has grown up speaking and reading French. If we write that list of instructions with a poor understanding of French the results could be very bad. That child might read the list and just sit on the steps of the hotel trying to understand them. The same thing happens to our computer when we give it a list of bad instructions. It might just lock up or produce bad results.

At this time I would like to describe The Editor/Assembler. You write your list of instructions in the Format or rather the language our Microprocessor understands. The better we understand this language the easier it becomes to sit down and write code.
I think it is only fair to warn you that with Assembly you cant just sit down and key in code. If a program is very small it still requires some planning. A simple thing like putting a letter on the screen gets to be quite a job. Your Assembly Language programs will always seem very large. However the end result, The Machine Code Instructions in RAM, will take up very little space. With this space savings you also get SPEED. This is why we want to tackle this Language.

Last month I showed you the LI instruction.
LI R3,>17 This means to Load Immediate into REGISTER >3 the Hexadecimal number >ØØ17.
Since our computers operate on even Memory Addresses the assembler adds the MSB >ØØ to the >17. It then becomes >ØØ17 when it is entered into memory.
You will recall that the REGISTER is located in an area of memory we call the WORKSPACE REGISTER AREA. The WORKSPACE REGISTERS for the BOOTSTRAP programs in ROM and the GROM programs are located in an area known as SCRATCH PAD RAM. This RAM area is on a 16 bit data path and is the quickest RAM in our computer. It is too small to put a program there but we can use the area for our programs WORKSPACE REGISTER. Yes thats right, we must make our own WORKSPACE REGISTER AREA for our program. The instruction to do this is LWPI. Load Workspace Pointer Immediate.
This instruction is usually the first instruction in a program. It looks like this:
LWPI >FØØØ
This instruction makes a 16 WORD area of CPU Memory into a WORKSPACE REGISTER AREA.
You may put any RAM address you like after the instruction.
This example starts the WORKSPACE AREA at address >FØØØ.
This makes REGISTER >Ø MSB at address >FØØØ and its LSB at >FØØ1. REGISTER >2 MSB is at >FØØ2 and LSB at >FØØ3. And so on. Remember that LOW and HIGH MEMORY are inside the Peripheral Expansion Box's 32K Card. This means that the data path to the 9900 CPU is only 8 bits wide down the black FIREHOSE CONNECTOR from the PE box. Our WORKSPACE REGISTER AREA is twice as slow here than if it were in the SCRATCH PAD RAM area >83ØØ to >83FF. This is not great news. We want SPEED after all. And if we are forced to do our math calculations in the slower RAM area we are not true 9900 Assembly Language programmers! Well all is not lost. It turns out we can use this area for our WORKSPACE REGISTERS if we are very careful and follow the guidelines spelled out in the Editor Assembler Manual on pages 4Ø4 to 4Ø6. One of the problems occurs when your program is executed, or started, from a BASIC program by CALL LOAD and CALL LINK, and your program returns control to BASIC, only >83ØØ to >8317 can be used by your program. Further, if you pass parameters, or variables, with the CALL LINK instruction then only >83ØØ to >83ØF is available to you. Now, you can use all of >83ØØ to >8349 if you decide to use BASIC to load and transfer to your Assembly Language Program and stay there. You only need >83ØØ to >831F for your WORKSPACE REGISTER AREA. So if you are careful you can use this area. Dont be afraid to use regular RAM for your REGISTERS. Only a programmer looking for optimum SPEED would need to use this area. We only need lightening SPEED for our program. So regular RAM is just fine.
If you remember last months description of the 99ØØ MAIL VANS Instrument Panel you will recall the three Digital Readouts.
1. The WORKSPACE POINTER REGISTER.
2. The PROGRAM COUNTER REGISTER.
3. The STATUS REGISTER.
With the LWPI instruction we have entered a new number into the WORKSPACE POINTER REGISTER.
9900 MAN now goes to the new area in memory for the computations with Registers. Any data left in the old WORKSPACE REGISTERS is left alone and remains untouched until you decide to jump back in there. All of my programs start with LWPI. If you are looking at Machine Code in RAM Memory the LWPI instruction is >Ø2EØ. It is rare in most programs I have looked at.

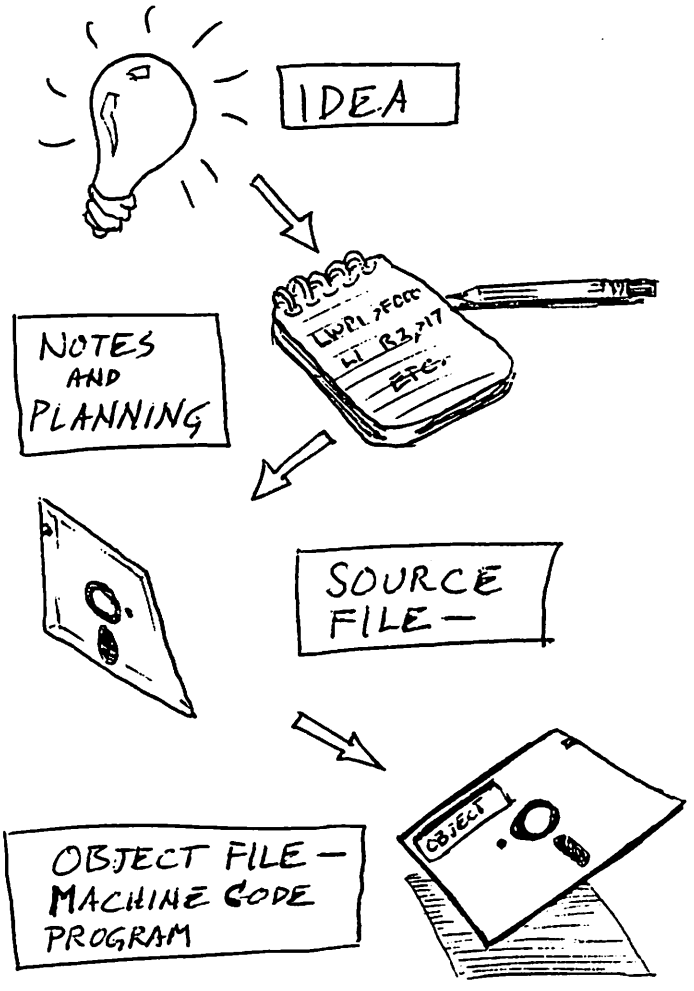
When you look at Machine Code in memory and run across >ØØ2EØ you can be sure it is an ENTRY POINT into the program. An ENTRY POINT is the first instruction to be read and acted upon by 9900 man. All programs have an ENTRY POINT and many of them use >ØØ2EØ as a first instruction.
Now you know a MACHINE CODE instruction.
O.K.. You can now put an instruction in a program that 9900 MAN will use to change his WP number in the MAIL VAN.



When we decide to write a program we can start with the Mini Memory Module or the Editor Assembler Module. Small experiments are best done in the Mini Memory Module. It is easy to use the Line by Line Assembler. However there is no record of your source code. You can not call back the Source code and trouble shoot it because the Line by Line Assembler, true to its name, only assembles source code one line at a time. After you type in LWPI >8320 and press ENTER that line is instantly turned into Machine Code and placed in Memory. To examine that instruction later you must first locate the memory space the instruction resides in and then you must decifer what the Machine Code means. Not a task for a beginner. So a good choice to start with is the Editor Assembler Module. Plug it in and select it from the Module Menu. You will then be facing the Modules Main Menu. At this time we are only interested in the Editor selection. Find the Editor Assembler Disk with EDIT1 on it and load it into Disk Drive one. Select Editor on the Main Menu then LOAD on the next. EDIT1 will load automatically. After it is loaded "FILE NAME?" will show up with a flashing cursor. If you have a SOURCE PROGRAM on disk now is the time to load it. However, we dont have such a file so just press the FUNCTION 9 combination. That prompt will disappear and you will be faced with 5 options, LOAD, EDIT SAVE, PRINT, and PURGE. Select EDIT. You will be faced with
*EDF (VERSION 1.2) or other version number.
Press FUNCTION S (Left Arrow).
You should now see numbers on the left side of the screen. Press ENTER several times. You may have guessed by now that you are in the EDITOR. You can write letters with this EDITOR. In fact this is a great word processor. It is dedicated to writing SOURCE CODE but it is equaly good at writing to your Mom.
Press FUNCTION 9 again and you will be in the command line. Be sure that ALPHA LOCK is on. you can now select E(DIT,F(IND,R(EPLACE,M(OVE,I(NSERT, C(OPY,S(HOW,D(ELETE,A(DJUST,T(AB, and H(OME?.
Lets say that we have now finished writing our SOURCE PROGRAM with all of those helpful comments next to the instructions. We now want to Save it. Press FUNCTION 9 twice. you will be back at the 5 selection Menu. Select SAVE. Lets call our SOURCE FILE, DSK1.SOURCE. This will help us identify it later.

Wow. We have our Letter type file named SOURCE. This is our record. We can load it back into the editor and change it any time. with this SOURCE file we are ready to run it through the all important ASSEMBLER program. Press FUNCTION 9 again and you should be back at the Main Menu. Put the Editor Assembler Disk back in the drive.
Select Assembler. The Screen will show * ASSEMBLER *. And prompt with LOAD ASSEMBLER?. With the Alpha Lock key on, press "Y" and the Assembler Program will load. It will then ask for the SOURCE FILE NAME?. Our file is named DSK1.SOURCE. press ENTER and the Disk Drive will come on for a moment. It will look for that name on the disk. Be sure you have inserted that disk in before pressing ENTER. Then it will ask you OBJECT FILE NAME?. This is the name your program will have. Lets say that it is DSK1.OBJECT. Type that in and press enter. Then it will ask LIST FILE NAME?. If you want to print the SOURCE file on you printer put in PIO or whatever your printer name is. If you do not want a listing just press enter and it will move to the next selection. OPTION.
If you have any special requests for the listing of the file you may enter R,L,S and or C. Any combination will do. If you do not want any options just press ENTER. Look at page 33 and 34 of the EDITOR ASSEMBLER MANUAL for OPTIONS and this procedure.
BOOOOOM. The Disk Drive Takes off and suddenly the screen goes blank and prints out ASSEMBLER EXECUTING. Then it is all over. ØØØØ ERRORS shows up on the screen, hopefully, and PRESS ENTER TO CONTINUE shows up.
If there are ERRORS it will show you the number and tell you what line number they are on.

We now have an OBJECT FILE named OBJECT and we can run it with the start name we specified in the SOURCE FILE PROGRAM. More on that later. For now we are just happy to know how to operate the Assembler. If you want to practice typing in a program page 342 to 344 in the Editor Assembler Manual is a great beginner program.
Next Month I will talk about that program and what all of that means.



If you do not type in that program dont worry. It will be explained in next months lesson. It will give you a sense of accomplishment though if you try and succeed. It is a nice simple example of using the monitor and moving things about. See you next month.

ASSEMBLY LANGUAGE

Hello again. This will be the last of my beginners series of lessons. I am studying BIT-MAP mode right now and hope to show how it works soon. BIT-MAP mode is something you have seen in graphic drawing programs. Each pixel on the screen can be turned on or off, and each row of eight pixels can have two colors. This is not as good as a different color for every pixel but it can be used in stunning ways. More on that later. If you have any questions, or want to enlighten me on any points, please write. Also, if you would like the full set of 6 lessons please include $7.00 with your request to cover postage and copying. The Post Office charges $.95 just for Postage! Jeepers! Send your request for the series to:

BOB WEBB
P.O. BOX 3023
ARCADIA, CA. 91007

LESSON NUMBER SIX

Well, we want to write our first letter of instructions to 9900 MAN. We will use the EDITOR program and EDITOR/ASSEMBLER MODULE to do so. 9900 MAN relys upon the ASSEMBLER program to translate our ENGLISH LANGUAGE (ASCII DV/80 FILE) LETTER OF INSTRUCTIONS into BINARY MACHINE CODE INSTRUCTIONS. Like BASIC and EXTENDED BASIC the way you write the instructions requires a certain format.

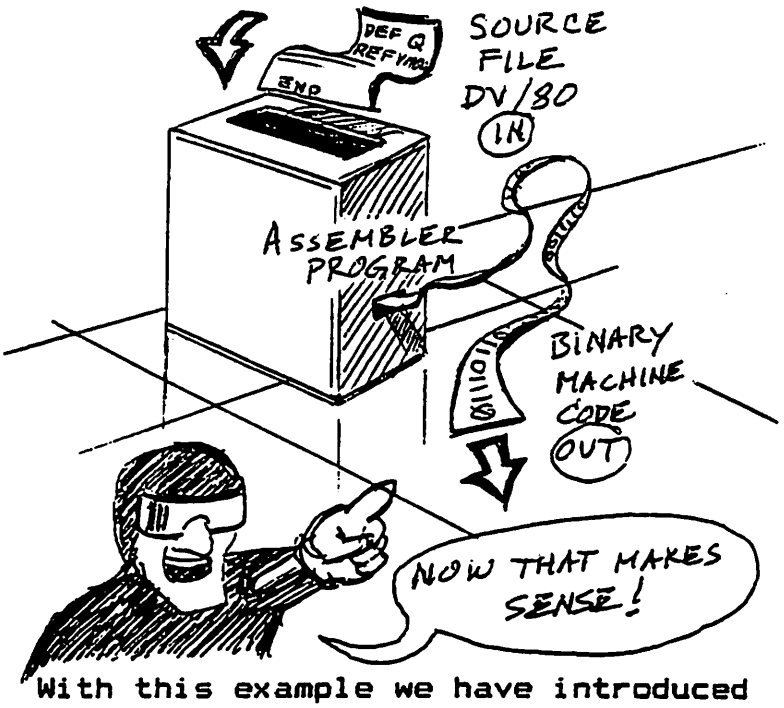
The ASSEMBLY LANGUAGE FORMAT is simple. There are 4 vertical columns of information used. Only the center 2 are needed. The other 2 are optional. Here is a sample program lifted from the EDITOR/ASSEMBLER manual. Pages 342 to 344.

```
                    1 TO 4 CHARACTER'S
                    <------------------>
                    1                 4
          123456 1234 12345678901 2345-to-8901
0000            DEF   BUBBLE
0001            REF   VMBW,VMBR,VSBW
0002 BBLE       DATA  >3C7E,>CFDF,>FFFF,>7E3C
0003 COLOR      DATA  >F333
0004 BBL        BYTE  >A0
0005 SPACE      BYTE  >A8
0006 LOC        DATA  >01DA,>020D,>0271,>02A5
0007            DATA  >02D6,>02E1,>0000
0008 MYREG      BSS   >20
0009 BUBBLE     LWPI  MYREG
0010            LI    R0,>394
0011            LI    R1,COLOR
0012            LI    R2,2
0013            BLWP  @VMBW
0014            LI    R0,>D00
0015            LI    R1,BBLE
0016            LI    R2,8
0017            BLWP  @VMBW
0018            CLR   R0
0019 LOOP1      MOVB  @SPACE,R1
0020            BLWP  @VSBW
0021            INC   R0
0022            CI    R0,>300
0023            JNE   LOOP1
0024            MOVB  @BBL,R1
0025            LI    R2,LOC
0026 LOOP2      MOV   *R2+,R0
0027            MOV   R0,R0
0028            JEQ   SCROLL
0029            BLWP  @VSBW
0030            JMP   LOOP2
0031 VDPBF1     BSS   >20
0032 VDPBF2     BSS   >20
0033 SCROLL     CLR   R0
0034            LI    R1,VDPBF1
0035            LI    R2,>20
0036            BLWP  @VMBR
0037            LI    R0,>20
0038            LI    R1,VDPBF2
0039            LI    R2,>20
0040 LOOP3      BLWP  @VMBR
0041            AI    R0,>20
0042            BLWP  @VMBW
0043            AI    R0,>40
0044            CI    R0,>300
0045            JL    LOOP3
0046            LI    R0,>2E0
0047            LI    R1,>VDPBF1
0048            BLWP  @VMBW
0049            JMP   SCROLL
0050            END
```

ENTRY POINT

I HAVE ADDED THE LINES TO HELP YOU SEE THE STRUCTURE.

I have only shown 3 vertical columns of data and one column of line numbers. The line numbers are there so I can refer to them. Also to show what it would look like on your computer screen. The fourth column of information would be on the far right of the other three. This column is for comments. Much like a REM statement in BASIC.
I have chosen this program from the EDITOR/ASSEMBLER manual because it is simple and gives a great deal of satisfaction when running. I could have written my own program for this lesson. But each of you probably have the manual and this introduction will show you how it is laid out.
I have changed the program from the book in only two ways. I have deleted the comments and split the DATA statement at line number 0006. This was done to make it fit on the first page. You will note that when I did this I was not following the manuals strict layout with comments following the asterisks. And, that because this was done, the line numbers would not be the same as the manuals. Line numbers are only used to debug programs. They are not reffered to in any way in any assembly program. When you get around to assembling your program if there are any errors the screen will show you what line number is a problem.
The assembler will give you an idea of what the problem is by saying something like SYNTAX ERROR 0003. (0003 being the line number). But in no way do you refer to a line number in the program itself. This brings me to our first column. Instead of using line numbers for a GOTO type of instruction, ASSEMBLY uses a LABEL. A LABEL is a point to start at or to go to.
In assembly JMP is used instead of GOTO. In our example there is an instruction at line number 0030 that says JMP LOOP2. At line number 0026 in the first column is LOOP2. If we were to write this JMP instruction in BASIC and used the line numbers it would be GOTO 26. So JMP is just like GOTO in BASIC but instead of line numbers we use LABELS.



With this example we have introduced the functions of column 2 and 3. The JMP instruction is in the OPCODE column. OPCODE is a fancy name for instruction. This column is the work horse of assembly. All instructions reside in this column. No data or labels are ever put here.
You will also note that there is a space between the LABEL column and the OPCODE column. This space is needed to tell the assembler that the label is finished and the opcode is next. The assembler expects up to six characters in the LABEL column, followed by at least one space, and up to four characters in the OPCODE column. You can have more than one space between columns. The third column is the OPERAND column. This is where you tell the assembler what you want added, moved, changed or read. The instruction is JMP and the OPERAND

is LOOP2. The manual calls these columns, in order, LABEL FIELD, OPERATION FIELD, OPERAND FIELD, and COMMENT FIELD.
The Source Statement Format part of the manual goes from page 46 to 48. Now that you have a Global view of the layout we can discuss some of the details needed to make your program work. The ASSEMBLER program needs to know the name of your program. It always needs to see a DEF statement at the top of your list of instructions. The ASSEMBLER takes the name in the OPERAND FIELD and places it in LOW MEMORY in an area called the REF/DEF table. This is a VECTOR table. An area in memory 9900 MAN knows to go to to find the starting point of your program. It is very important to note line number 0009. The LABEL at 0009 is BUBBLE. Also please note on line number 0000 the Instruction DEF BUBBLE. The ASSEMBLER reads DEF BUBBLE and places the name in the REF/DEF VECTOR TABLE in LOW MEMORY and then looks for that label in the LABEL column. It then knows that line number 0009 is the ENTRY POINT into the program. Or, rather, the first instruction to be executed.
If you will recall from one of the earlier lessons the instruction LWPI. This means to LOAD WORKSPACE POINTER IMMEDIATE. 9900 MAN will take the address following this instruction and place it in his WP digital readout on the Dash Panel in his 9900 MAIL VAN.
After the needed DEF instruction to the ASSEMBLER comes the REF Instruction. In Assembly it is mandatory that you tell the computer that you are going to use one of its, built in, SUBROUTINES in advance of using it. The ASSEMBLER does not like surprises.

An analogy to EXTENDED BASIC would be, you must tell the computer in the first lines of instructions the Commands you are going to use. You would be forced to say, in this program I am going to use the PRINT statement and the DISPLAY AT statement. If you do not put that in the first line an error will be generated and you will not be able to use them. This is how you must treat the built in subroutines in Assembly. What kind of subroutines are there? In our program the VMBW, VMBR and VSBW subroutines are used. So they must be declared in the first instructions. Once again this REF OPCODE is only for use by the ASSEMBLER program. It reads REF VMBW. And knows that anywhere that term VMBW shows up in our text it is to equate that term with the ENTRY POINT into the subroutine program. This subroutine is an assembly language program burned into a GROM chip inside the EDITOR/ASSEMBLER module. The VMBW program, or subroutine, is declared to the ASSEMBLER and it then links your program to the VMBW program. When you use the BLWP or "BULLWHIP" command you are actually leaving your program for a while and running this small subroutine and then branching back to where you left off. Just like the GOSUB command in BASIC. There are many BRANCH routines in Assembly. BLWP stands for BRANCH AND LINK WORKSPACE POINTER IMMEDIATE. This is used if you are branching out from your program, accessing the Mail Boxes at TEX'S GROM RANCH, or sending or receiving Mail to the VDP RAM area. You dont have to declare BLWP because that is an OPCODE. But you must declare the subroutines built in to ROM or GROM. These subroutines make communicating with TEX at GROM RANCH or sending data to the screen in VDP an easy task instead of a long drawn out process. VMBW stands for VIDEO DISPLAY PROCESSOR RAM MULTIPLE BYTE WRITE. What a mouthfull!

All that means is that it helps you write multiple bytes of data anywhere in VDP RAM. Here is how it works.
Lets say you want to write two bytes of data to an area in VDP RAM. It is not important what this area is used for. Just know that it has something to do with putting an image on the screen of your monitor.
We can use the example starting at line number 0009 or, to speak like an Assembly Programmer, at the LABEL "BUBBLE".
After we set up our Workspace Register Scratch Pad area with the LWPI instruction we can see the first line of information needed for the VMBW program.
LI R0,>394 (>394=VDP ADDRESS)
LOAD IMMEDIATE >394 into REGISTER ZERO. Register Zero is 9900 MAN's Scratch Pad Ram space. It is a place he holds information while he manipulates it.
We dont know where that WORKSPACE AREA is. Because earlier we told the assembler to place it anywhere it likes with the command at line number 0008, MYREG BSS >20.
BSS is used to tell the assembler to set aside an area of memory X number of bytes and give that area a name. In this case the name is MYREG. Standing for MY REGISTER. You could name it anything you like. Pick a name that you will help you remember it. The BSS Instruction set aside >20 bytes of memory for the SCRATCH PAD RAM. You could make it an exact address if you like.
LI R1,COLOR
LOAD IMMEDIATE >F333 into Register 1. Earlier the LABEL COLOR was was given to the number >F333. Now anytime you use the name COLOR anywhere in the program the ASSEMBLER knows that you mean >F333. In this case it means we make the color of the objects, that will be defined later on in the program, with a foreground color of >F(HEX for WHITE). And a background color of >3 (HEX for LIGHT GREEN). Then foreground >3 and background >3.
LI R2,2
LOAD IMMEDIATE into Register 2 the DECIMAL NUMBER 2. Since there is no greater than symbol ahead of the number it knows that the number is in DECIMAL.
BLWP @VMBW
"BULLWHIP" to the VMBW subprogram. 9900 man then branches to the subprogram and begins to execute those instructions. The BULLWHIP command does what is called a CONTEXT SWITCH. When it branches to the subprogram it branches to another set of WORKSPACE REGISTERS. These registers are dedicated to the subprograms in ROM and GROM exclusively. You better not use these other SCRATCH PAD RAM area's for your programs or you will crash. This context switch takes place so that your registers will be left alone while the subprogram is running. It uses its own space and not yours. No other computer in the world does this that I know of. This is one of the great features of our machines.
The subprogram runs and takes the data you put in your own Registers 0,1 &2. It then does all of the work loading those 2 bytes of data into VDP RAM. Change the number from 2 to 500 and you can see the power of this little subroutine. The subroutine branches back into the program where it left off (line number 0014). And continues to the next Instruction.
The last but not least OPCODE is the END command. If you type the LABEL BUBBLE in the OPERAND column the program will start automatically. You wont have to type the word BUBBLE to start the program. Thats it!
Now you know it all. HA HA. Study your manual and Good Luck! Thanks for your letters of support and may GOD BLESS US ALL! BYE FOR NOW!

KEYBOARD READER

This small program is one of my most used programs. I can never remember the number associated with a key press or ASCII symbol. So, I threw this thing together. Let me caution you before I continue. Do not run this program until you have saved it. Once you start it, the only way to stop it is to turn your computer off. Function Quit and Function 4 are disabled so you can't break back in or kill it. This was done so that all combinations of key presses could be viewed. If you dont want these features delete line numbers 160 to 190. You must delete line 170 if you dont have memory expansion hooked up or a syntax error is generated.

```
100 ! KEY TO NUMBER PROGRAM
110 ! extended basic & 32k
120 ! by Bob Webb, 6-1991
130 ! Caution: you will have to
140 ! turn off computer to end.
150 !
160 ! CALL LOAD disables quit
170 CALL INIT :: CALL LOAD(-31806,16)
180 ! ON BREAK NEXT disables ftcn 4
190 ON BREAK NEXT
200 !
210 CALL CLEAR
220 BLANK=0
230 DISPLAY AT(5,5):"KEY TEST PROGRAM"
240 DISPLAY AT(7,5):"Press Any Key."
250 DISPLAY AT(9,5):"It's Number will"
260 DISPLAY AT(10,5):"be displayed."
270 !
280 ! MAIN LOOP
290 !
300 CALL KEY(0,K,S)
310 BLANK=BLANK+1
320 IF BLANK>1000 THEN 410
330 IF S=0 THEN 300
340 DISPLAY AT(12,4):K
350 DISPLAY AT(12,10):CHR$(K)
360 BLANK=0
370 GOTO 300
380 !
390 ! BLANK SCREEN
400 !
410 CALL CLEAR
420 CALL KEY(0,K,S)
430 IF S=0 THEN 420
440 GOTO 210
```

Once this program is running, press any key.
It's number will be displayed. If an ASCII symbol is associated with that particular key press it will be displayed just to the right of the number.

This program does not break any new ground. However you might find a part of it to be of use. I have added one of my favorite little details to it.
If no key is pressed for a given amount of time it jumps to a screen saver type of subprogram.

The BLANK variable is a counter. This clock ticks away and if a key is pressed it is reset to zero and begins again.
If no key is pressed it jumps down to line 400 and stays there until a key is pressed.

This second program can be added to your own program. It has the same kind of screen saver loop in it as the first. after the GOSUB statement you can test for which key was pressed (IF K=13 THEN X). Happy Computing, and long live our 99/4a!

```
100 ! KEY LOOP - extended basic
110 ! by Bob Webb, 6-1991
120 CALL CLEAR :: DISPLAY AT(10,7):"TEST"
130 GOSUB 180
140 CALL CLEAR :: DISPLAY AT(10,7):"ENTER"
150 ! 13 IS THE ENTER KEY
160 GOSUB 180 :: IF K=13 THEN 140
170 GOTO 120
180 ! PRESS ANY KEY LOOP
190 FOR BLANK=1 TO 200
200 CALL KEY(0,K,S):: CALL HCHAR(24,16,32)
210 IF S=1 THEN 250 :: CALL HCHAR(24,16,107)
220 NEXT BLANK :: CALL CLEAR
230 CALL KEY(0,K,S):: IF S=0 THEN 230
240 GOTO 190
250 RETURN
```

```
*       This program run's right.
*       It had a bug in the scroll
*       routine. The Editor/Assembler
*       Manual is wrong. I fixed the
*       bug. It was in the SCROLL loop.
*       This program is on pages 342
*       to 344. The bug is on the 2nd
*       line on page 344. The Manual
*       adds when it should subtract.
*
*       Assemble with the R option.
*
        DEF   BUBBLE
        REF   VMBW,VMBR,VSBW
*
BBLE    DATA  >3C7E,>CFDF,>FFFF,>7E3C
COLOR   DATA  >F333
BBL     BYTE  >A0
SPACE   BYTE  >A8
LOC     DATA  >01DA,>020D,>0271,>02A5
        DATA  >02D6,>02E1,>0000
MYREG   BSS   >20
LINE    DATA  >20
*
* Set up colors
* BUBBLE is the entry point into the program.
*
BUBBLE
        LWPI  MYREG
        LI    R0,>394                   color table 20 and 21
        LI    R1,COLOR                  load colors >F3 and >33 (white & green)
        LI    R2,2                      2 bytes to load
        BLWP  @VMBW                     move to VDP RAM
*
* Set up character definition
*
        LI    R0,>D00                   character >A0 location
        LI    R1,BBLE                   definition of bubble character
        LI    R2,8                      8 bytes to move
        BLWP  @VMBW
*
* Clear screen
*
        CLR   R0                        start at VDP RAM >0000 (first char pos)
LOOP1   MOVB  @SPACE,R1                 move space character
        BLWP  @VSBW                     move one space at a time(char position)
        INC   R0                        points to next location on screen
        CI    R0,>300                   end of screen >20 = 768th char pos)
        JNE   LOOP1                     jump if not equal to loop1
*
* Place bubbles on the screen
*
        MOVB  @BBL,R1                   load character code for bubble
        LI    R2,LOC                    load pointer to address for bubble
LOOP2   MOV   *R2+,R0                   load real address
        MOV   R0,R0                     check if finished loading
        JEQ   SCROLL                    finished. start scrolling the screen
        BLWP  @VSBW                     write bubble on the screen
        JMP   LOOP2
*
```

ENTRY POINT

```
* Scroll screen
*
VDPBF1 BSS   >20
VDPBF2 BSS   >20
*
*    Fixed SCROLL routine.
*
*    Pages 342 to 344 in the
*    Editor/Assembler Manual.
*
*
*    At the top of the program I
*    added this....
*
*   LINE   DATA >20
*
*    And in the SCROLL routine I
*    added this....
*
*          S     @LINE,R0
*
*    The Editor/Assembler Manual
*    shows an AI (ADD IMEDIATE)
*    instruction when they should
*    have subtracted. (Top of page
*    344).
*
```

ANY QUESTIONS? COMMENTS?

WITH A SELF ADDRESSED STAMPED ENVELOPE WRITE TO:

BOB WEBB
P.O. BOX 3023
ARCADIA, CA. 91007

```
SCROLL
       CLR   R0                    VDP address >0000 = first char position
       LI    R1,VDPBF1             CPU buffer address
       LI    R2,>20                number of bytes to move
       BLWP  @VMBR                 move >20 from VDP RAM (32 chars=1 line)
*
       LI    R0,>20                VDP address >20
       LI    R1,VDPBF2             CPU buffer address
       LI    R2,>20                number of bytes to move
LOOP3  BLWP  @VMBR                 copy the line
       S     @LINE,R0              move to lower VDP memory (SUBTRACT >20 FROM R0)
       BLWP  @VMBW                 write back to the lower line
       AI    R0,>40                read next line
       CI    R0,>300               check if end of screen (char pos 768)
       JL    LOOP3                 if not, copy more
*
       LI    R0,>2E0               write the last line
       LI    R1,VDPBF1             CPU buffer where the first line is
       BLWP  @VMBW                 move CPU to VDP
*
       JMP   SCROLL                keep scrolling
*
       END   BUBBLE
```

REMOVE THIS IF YOU DO NOT WANT THE PROGRAM TO AUTOSTART. IF YOU REMOVE IT YOU MUST THEN TYPE "BUBBLE" TO START IT.