Astrocade Machine Language Programming Tutorial

"The Bit Fidder's Corner"
By
Andy Guevara

The author of this tutorial, Andy Guevara, programmed the Machine Language Manager cartridge for the Astrocade. This tutorial complements that cartridge, but has a general focus so this information can be used without reinterpretation by Astrocade assembly programmers, or those wishing to learn about the machine.

The complete five serialized machine language tutorials here were originally published in the "Arcadian," a newsletter for the Bally Astrocade computer/console. The serial series appeared in 1983 and 1984 in volume 5, pages 42, 43, 47, 114, 132, 133 and volume 6.

Part 1

Hi there! This is the first installment of what I hope will be a long and prosperous relationship between you, me, and the ARCADIAN.

The aim of this column is simple--to dispense as much "inside" knowledge as possible about the workings of the Astrocade. This means down to the bit level if necessary. I plan to cover a lot of the material referred to in the on-board subroutines manual put out by the ARCADIAN. You might want to pick up a copy. By the time we're done, we ought to be able to do just about anything a microcomputer is supposed to be able to do.

First, some preliminaries: I won't be using BASIC very much in my examples. The reason for this is that the Astrocade innards are not programmed in BASIC. BASIC as a language, is itself a program and a series of subroutines that sits between us and the Z-80 microprocessor. So, as a rule, I will be talking primarily in Z-80 machine language. I know there's not a lot of you who will understand it right off, so the first couple of installments are set up to familiarize you with the terminology and conventions used in programming at the byte level.

I see that it's time for a sales pitch... Since I won't be using BASIC, how am I going to try out examples, you might ask? Well, the answer is simple. We at The Bit Fiddlers have developed a cartridge we call the Machine Language Manager.

So as to help explain what it is, let me first explain what it isn't. The MLM is not a language cartridge like BASIC is. As I said earlier, BASIC is a program. It translates your BASIC statements to machine code (that is, instructions the Z-80 can understand) on a line-by-line basis, interpreting each statement as it goes. This is why there are line numbers; so the interpreting process doesn't get lost. This also explains why BASIC programs are relatively slow.

The MLM, not being a language; doesn't need line numbers. It works directly on the Z-80 memory. It's what is known in the trade as a Monitor program. Its purpose is to directly enter and change values or instructions the Z-80 will understand. This way you get to tell the Z-80 precisely what to do.

What this means is, in order for you to use the MLM, you're going to have to learn to use Z-80 machine language. But then, that's why this column is here... to show how the Astrocade works--in machine language.

A little more on the MLM. We've put in a few helpful capabilities, such as a formatted listing, ability to change the register contents, cassette tape storage routines for use with the original 300 Baud interface, and a print routine for those of you who have connected a printer to your unit. There are also routines in the cartridge to clear the screen, change the amount of memory available for your programs, and output single characters or whole lines to the screen or printer.

So these are the advantages:

1. You get faster-running programs
2. Programs take up less space than their BASIC equivalents
3. Memory can be rearranged to allow over 3K Bytes of storage
4. You get 4 colors for either side of the Right/Left boundary instead of 2
5. You get direct access to the on-board subroutines for animation, character generation, graphic effects, timing, and sound effects. In other words, you have the capability to produce cartridge quality programs that are storable on cassette tape! Trust me.

But back to what I set out to do.

The Z-80 talks in, and responds to Bytes. Fine, what's a Byte? Well, a Byte is made up of 8 Bits. Bit is short for Binary Digit. So the Z-80 talks in Binary.

Think of it this way: At the Z-80 data port there are 8 ON-OFF switches lined up side by side. There are only two states each switch can be in: ON or OFF. This is how Binary (Base 2) arithmetic works.

To give the Z-80 a particular instruction code, we can set the switches to a particular combination of ON and OFF states. Let's assign the number 1 to the ON state and 0 to the OFF state. Now we can do it in terms of a binary code, such as 01100110.

Well, there's a better way yet. We can translate this binary number into one we can better understand. For example, 00000001=1 Base 10. Simple enough. Let's drop the leading zeros for now. OK, let's add 1+1 in Base 2: 1+1= ? Since values can only be 1's and 0's, we have to put a 0 and carry a one into the next column. The answer then is 10 in Binary, 2 in Base 10.

Well what's all this mean? It means we don't have to keep track of Z-80 instructions in configurations of 8 individual bits. We can do it by converting to numbers.

Binary goes like this:

0000	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8

and so on, to 255 for all 8 hits being turned ON. But this is a little unwieldy if we have to go back and forth to the Binary form. So let's try a different approach.

Break the 8 bit configuration into two 4-bit subcodes, like 0110 0110. Now each subcode, called a Nibble (honest!), can be a number from 0 to 15. Let's further constrain it by saying each nibble can only have a 1-digit representation. That would sake 0110 0110=66 in the new code.
But what about the numbers 10 to 15? Well, since we can't have 2-digit nibbles, we'll assign the letters A through F for these values. Welcome to the Hexadecimal world!

In this system, each four bits represents one Hexadecimal column. That is: F+1=10 in Base 16. It equals 16 in Base 10.

The reason for all of this is that almost every book ever written for the Z-80, or any other microprocessor, leans heavily on Hex numbers. At 2 digits per byte you can see why, in terms of printer's ink alone!

OK, we know that Z-d0 instructions are coded in Bytes made up of 2 Hex digits. But how do we get the codes into memory? And where do we put them? How does the Z-80 know where to go to get at them? Answers to these and other burning questions will be in next month's column.

One last thing. I'd like to hear from you and what you think about the column. Is it at too low a level or too high? Also, are there any requests? Do you have a particular problem or application that you would like addressed? Drop me a line!

The format of the column is still flexible and YOU are the ones to benefit. See you next time.

Part 2

Hello again! Last month we covered a little bit of binary and Hexadecimal code and talked a little about what bytes are. This month, knowing that the Z-80 responds to these 8 bit lumps called bytes, we'll go over how and where to put them into memory.

But first, to help you along with Hex numbers, I've put in this handy-dandy table.

Hexadecimal Columns

6		5		4		3		2		1	
HEX	= DEC	HEX	= DEC	HEX	= DEC	HEX	= DEC	HEX	= DEC	HEX	= DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	727,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	7,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,O64	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,810	F	240	F	15
0 1 2 3		4 5 6 7		0 1 2 3		4 5 6 7		1 2 3 4		4 5 6 7	
BYTE				BYTE				BYTE			

To show how it works, we'll do a few Hex to decimal conversions, since a lot of people don't believe that Hexadecimal is a real number system. As you recall, Hex is a base 16 number system. This means there are 16 distinct numbers per column. In base 10, the numbers for any column are 0 thru 9. So it is for Hex... except we have to invent characters for the numbers above 9. To make them memorable, we'll make those characters A thru F.

Take a look at the columns under the heading "1" on the table. The numbers under the heading 'HEX' are the Hexadecimal numbers (clever, right?). Notice the corresponding values on the right. It's all the same until you get to A. Like with decimal, when you get to the largest value you can have in a column, to make the value one larger, you put a zero then add one to the next column, on the left. Like going from 9 to 10. Except it's F to 10. Anyway, let's do a few examples.

Since we're working with numbers 4 digits long, we'll only use the columns labeled 1 thru 4. Let's take the number 4000H. The 'H' is there to identify it as a Hex number. To convert, take the leftmost digit, and find it in the table under '4'. To the right of the digit will be its decimal equivalent. You should have found 16,384. The remaining three digits are zeroes, and equate to zeroes in decimal. Adding up all the equivalences gives us 16,384, as you might have guessed. As you probably already know, this is the value of the first address in RAM memory. But more on that in a bit. Let's do FFFFH.

This one's easy too. Since F is the last entry in every column, we take the last corresponding number in each of the four columns and add them together:

```
    61,440 + 3,840 + 240 + 15 = 65,535
```

which is the highest 4 digits can go. The same operation applies for any Hex number conversion. So, let's get on to other things.

As luck would have it, each byte in memory has its own unique address. This address is a number, two bytes (sixteen bits) long. Being sixteen bits, the address can be a value from zero to 65,535 (affectionately known as 64k) in decimal, which is zero to FFFF in Hex. The designers of the Astrocade decided to assign certain address ranges to specific purposes. For instance, the system programs in ROM (Read Only Memory-you can't change it), start at address 0000 and proceed to address 1FFF in Hex, which is 8191 in decimal. Starting at address 2000H and going on up to 3FFFH is address space available for plug-in cartridge ROM. There is no physical memory at this range of addresses until a cartridge is put into the slot.

RAM memory (standing for Random Access Memory, which doesn't really say that it's alterable, although it is) starts at address 4000H (16384 decimal). This is where all user programs, variables, and graphics go. It continues on to 4FFFH which is the last address used by the Astrocade. All of this RAM area, believe it or not, is mapped to the TV screen. It's where you put information when you want it to show on the screen. There are tricks to hide it when you don't want it to show, but we'll get to that in our segment on graphics.

When you add external memory to the system, you have to make it respond to the addresses above 5000H so that you don't interfere with memory inside the Astrocade. This added memory is not mapped to the screen, so it's entirely available for programs and data. We'll explore the implications of that idea later also.

Got all that? I didn't think so, but let's forge ahead anyway. You can see that unless we have external RAM, we can only use the address space from 4000H to 4FFFH for our programs. All the rest is previously assigned. We'll get into particular addresses a little later when we start putting together programs of our own.

This seems like a pretty good place to close it up. Next time, we'll take a look at the Z-80's instruction set, and get a feeling for how the registers are used. Who knows? Before long we may be able to figure out what Mike Skala was talking about in his column on graphics. Hang in there!

Part 3

Hello again! Time to forge ahead in our quest for Z-80 knowledge. Last time we went over the Hexadecimal number system. Today we're going to cover the Z-80 registers and a few of the instructions, so we can get a feel for how the little monster is programmed.

I'm sure you're all aware of the variables A thru Z as used in BASIC. Well, the Z-80 has several internal storage places of its own that are not unlike the BASIC variables. These are the registers. They are 8 bits wide, meaning they can hold values from 0 to FF (255 for you decimal types). Most are general purpose, but a few are reserved for special uses.

The registers are used primarily for storing information for quick access, and for doing most of the arithmetic functions of the instruction set. Another use is to hold an address for accessing data from memory. In this case, two registers are paired together to form a 16 bit address word.

The names of the registers are: A,B,C,D,E,H, and L. Clever huh? Any of these can be used to hold an 8 bit value. The A register is the most often used because the answers to 8 bit additions, subtractions, and other computations end up in the A register. For 16 bit manipulations, the pairs BC, DE, and HL are used, with the answers going to the HL pair.

The HL pair is important for another reason. When moving data from one location in memory to another, or loading one of the other registers with data from memory, the HL pair can be used as an intermediate addressing register. That is, I can put an address in HL, and load the A register with the data located at the address held in HL. Further, if I need a string of values, like in a table, I could put the starting address of the table in HL and load A from (HL). The parentheses mean "the location addressed by." Remember that. Getting the rest of the values from the table is then just a matter of incrementing the value in H and L and performing the same load instruction.

The H and L registers are so named because they hold the High and Low bytes of an address in memory. The other names were simply done for convenience.

Well, that ought to be enough on the registers. Let's see how the instructions look. Going back to the small example above, we loaded the A register from a place in memory addressed by the HL register pair. The instruction for this looks like:

```
LD A,(HL)
```

This is the standard format for Load type instructions. The destination of the data comes first, then the source. In this case, the above instruction is read "Load A from the memory location addressed by H and L."

There are other ways to put data into a register. We can get it from another register:

```
LD A,B
```

or we can get it from memory directly:

```
LD A,(4A00H)
```

or we can load the value immediately:

```
LD A,4
```

These are what are referred to as "addressing modes." Let's do this the right way:

`LD A,4` is the IMMEDIATE mode
`LD A,B` and
`LD A,(4A00H)` are the DIRECT mode
`LD A,(HL)` is the INDIRECT mode.

In the IMMEDIATE mode, we get the data from the program itself. It's like A=4 in BASIC. In the DIRECT mode, we get the data from another place, either another register or a memory location (in this case memory address 4A00H). This is like A=B, or A=%(18944). The INDIRECT mode of addressing takes us one step further away by having to know what is in the HL pair first. This is like A=%(B), where we had set B sometime earlier. There is also an INDEXED INDIRECT mode, but I don't want to overload you just now.

Remember that our Z-80 only recognizes numbers. That means that we can't use the shorthand instructions above directly. What we have to do instead is supply the Hex numbers that correspond to the instructions.

For instance,

```
LD A,4
```

converts to

```
3E 04
```

Simple, right? I didn't think. so. Every instruction the Z-80 knows is represented by numbers that are 1, 2, 3, or 4 bytes long. All told, there are 696 instructions in the set, so you can see why the shorthand notation is used.

From here you can see the need for a good list of the Z-80 instructions (or OP-CODES as they're called). Most (if not all) of the books on the Z-80 have a list in alphabetical order of the Op-codes (also known as mnemonics, the first 'm' is silent), and a list in numerical order of the corresponding Hex numbers.

In future columns, I will explain groups of instructions rather than try to go through all 696 of them. And in cases where I show a program, I will also include the Hex numbers (Object code) for the instructions I use.

See you next time.

Part 4

I'm sure by now that a lot of you have gotten some of the 'inside' information on the ARCADE's on-board subroutines. It was one of the first things I sent away for. The question is, how many of you were able to make heads or tails of the information?

Well, that's what the aim of this column is: explanations of some of the routines and how they are used.

This all assumes, of course, that you have a working knowledge of Z-8O machine code. If you have that much, then experimenting using BASIC or the Machine Language Manager should be easy for you.

Probably the most informative of the documents put out by the ARCADIAN was the on-board ROM description written by Dave Nutting & Associates. It was almost written in English. I say 'almost' because some things were assumed to be common knowledge, and others had missing information or explanations that fell short.

Since this is the heart of the ARCADE, we'll be covering the subroutines in a little more detail than the Nutting manual. Hopefully, we won't have to dig into the source code to find all the answers.

First, some preliminaries. All of the on-board Subs have to be called by using the code "FF". The ARCADE is designed to recognize this value as a system call. For my own use, in the assembler that I use, I've defined the labels "SYSTEM" and "SYSSUK" to be the value FF (255 Decimal). The Nutting manual uses these two labels in describing each of the subroutines. The two labels indicate how the arguments (additional information) are to be delivered to the subroutine.

In all cases, the subroutine number refers to the 'SYSTEM' convention. The subroutine number is what the ARCADE operating system uses to get to the right subroutine. The method used here is to first load the registers with the right arguments, then issue the system call sentinel, SYSTEM (that is, FF), then the number of the subroutine.

For instance, to call subroutine number 1A (Hexadecimal for 26), using the SYSTEM convention, we would first load all the pertinent information into the registers, then put FF as the opcode, followed by the value 1A. It assembles thusly:

```
11 00 4A     LD   DE,4A00H
01 00 11     LD   BC,1100H
3E 00        LD   A,0
FF           DEFB SYSTEM
1A           DEFB FILL
```

The Hex object code is shown on the right. With these instructions the subroutine 'FILL' is called, with execution returning to the very next instruction after the subroutine number. We'll get to what this specific subroutine does later.

The SYSSUK version of this same call looks like this:

```
FF           DEFB SYSSUK
1B           DEFB FILL+1
00 4A        DEFW 4A00H
00 11        DEFW 1100H
00           DEFB 0
```

Notice that you don't have to go through the expense of loading the arguments individually. They are "sucked" into the proper registers by the ARCADE operating system. The order that they are put in is therefore very important. Also notice that the subroutine number is 1 greater than before. This will always be true, and is how the operating system knows the difference between SYSSUK and SYSTEM. SYSSUK requires the subroutine number to be one more than its usual value. And SYSSUK will always use an ODD subroutine number.

The advantage of the SYSSUK structure is more compact code. The disadvantage is the inflexibility. It is better used for absolute cases than for iterative loops. Like for borders that will always be in the same place, rather than objects that will move across the screen.

A last important point. When returning from a system subroutine, unless otherwise stated, the registers will be set to the loaded values. That is, the values that were loaded before a SYSTEM call, or the values picked up by a SYSSUK call. The exception is when a specific result is to be returned in a register or registers.

So, now that that's out of the way, let's begin. The very first subroutine, 00, is pretty interesting. This is the system interpreter, not to be confused with the BASIC interpreter. What this routine does is allow the stringing together of several subroutines under one "call." There are no arguments associated with this routine. It looks like this:

```
FF           DEFB SYSTEM
00           DEFB INTPC
```

The subroutines you want to string together then follow the above without using the FF sentinel. For instance, using the FILL example from above, we could fill three different areas of the screen like so:

```
FF          DEFB SYSTEM
00          DEFB INTPC

1A          DEFB FILL
00 40       DEFW 4000H   ;WHERE TO START
10 00       DEFW 10H     ;HOW MANY
00          DEFB 0       ;WITH WHAT

1A          DEFB FILL
20 40       DEFW 4020H
10 00       DEFW 10H
11          DEFB 11H

1A          DEFB FILL
30 40       DEFW 4030H
50 00       DEFW 50H
FF          DEFB 0FFH

02          DEFB EXIT
```

The last byte is the subroutine for leaving the interpreter and returning to normal machine language. There are no arguments associated with EXIT either.

To allow the interpreter to work a little more efficiently, there are subroutines for jumping and calling from within the interpreted string.

Subroutine 04, RCALL, is for calling a standard machine language subroutine from within an interpreted string. For instance,

```
04          DEFB RCALL
20 4E       DEFW 4E20H
```

would call a machine language subroutine at address 4E20H. That subroutine would end with a standard RET (C9), and control would return to the next subroutine in the string.

Subroutine 06, MCALL, is similar to RCALL, except that the call is to another interpreted string of subroutines. If we had an interpreter string at address 4E20H, then the call would look like:

```
06          DEFB MCALL
20 4E       DEFW 4E20H
```

The called string would have to end with subroutine 08, MRET. This would make sure the return was to the correct place.

The final interpreter-related subroutine is number 0AH, 10 for you Decimal types. This is the MJUMP routine, which does a direct jump out of one interpreter string into another. The form is:

```
0A          DEFB MJUMP
20 4E       DEFW 4E20H
```

and, of course, the target of the jump has to be an interpreted string itself.

Well, that about wraps it up for now. I don't intend to take all the routines numerically down the line like today. Instead, I'll skip around and try to explain how groups of subroutines work together. This way we'll take some of the mystery out of animation, joystick control, and interrupts and such.

By the way, for those of you who don't recognize some of the assembler directives I use:

`DEFB` stands for 'Define Byte"
`DEFW` stands for "Define Word"
`DEFM` stands for "Define Message"
`DEFS` stands-for "Define Storage".

We'll use the last two in examples in upcoming articles.

See you next time.

Part 5

Well maybe not ALL of them, but enough to get you on your way...

The largest single source for machine language techniques for the Astrocade is the system description known affectionately as the Nutting Manual. This one document has been the source of almost all the machine language routines used in the latest generation of games (and, from the beginning, in all cartridge games).

Granted, you do need to know something about Z-80 machine language before tackling this manual. But if that's not a problem, then this is what you need to get that space war game of yours off the ground.

Curious? Let's take a look at L&M'S Ms. Candyman... What's the first thing you notice? The music of course! There's a complete music processor in the ROM chips on board the Astrocade just waiting to be tapped. And it's done in a much easier way from machine code than from BASIC. There is also information on making rectangles of any size, much like the Box command. And for drawing patterns on the screen from memory.

Probably what you're most interested in are the routines used to move objects around the screen. Well, there are two ways to do this. The first is to make your own character set with characters of your own choosing. Like a gremlin instead of an "A." Then it's a matter of "printing" them wherever you choose. The second method is to use the vectoring capability of the Astrocade. By setting up a block of memory for specific information, you can let the system take care of moving your character around the screen, keeping track of the movement limits you've defined, and whether or not he's supposed to bounce off walls, etc. That's the one we used. By using the system functions as often as possible, we only needed to develop our own framework to direct the activities of the game. Would you believe that we don't control the notions of the Jokers in Ms. CM? The Astrocade does it for us! All we do is set the speed and stand back...

Did you know that the SNAP function is built into the system? From machine code it's easy to get to and use. Of course the SHOW function is there too. There's also the capability to scroll any part of the screen, and define the size of the scroll window.

For keeping track of data, there are routines for handling tables of bytes, words, or nibbles easily. I use these a lot.

Here's a very little known fact... On board the system, there is a set of floating point math routines.

And they're not even described in the Nutting manual! I always thought this strange, since the calculator programs haven't been giving back integer answers. A little sleuthing into the source code of the unit showed these routines. And if you're real nice, I might explain how to use them. Suffice to say that about half of the 200+ page manual is the source code of all that is in the ROM set of the Astrocade. This has been most helpful when the descriptions of routines were less than adequate (which is about half the time).

Once you get the hang of it though, you'll find the manual indispensable. I have!

-- Andy Guevara 1983/84

-- These tutorials converted to text, and then to RTF, by Adam Trionfo, January 13, 2002