4.

I promised to throw you in the swim during that last lesson, but sorry I had to leave you swimming at the end of it. Here's a short review:

The 6809 microprocessor contains several registers. Each register is in effect a memory slot inside the processor, but each register has a uniquely defined task. The A and B accumulators are 8-bit arithmetic logic units, or ALUs, capable of performing simple arithmetic and logical operations. The X and Y registers are 16-bit registers used mainly to index, that is to point to, addresses within the processor's memory range. The PC, the program counter, points to the memory address containing the next instruction that the processor is the act upon.

The address range of the 6809 runs from **$0000** to **$FFFF**, a total of 65,536 locations. When the power is turned on, the processor fetches the information stored in the top two bytes of memory, concatenates it, and places it in the program counter. The processor obtains its first instructions from there, the instruction decoder begins translating the instructions into actions, and the computing begins.

As an example of this much of the 6809's architecture, I presented a short program. In that example, the X register was given the address of — that is, indexed to — the first character of an ASCII message stored in memory, and the Y register was indexed to the first display location in video memory. The A accumulator loaded a value from memory indexed by X, and stored that value in memory indexed by Y, causing an ASCII character equivalent to the stored value to appear on the screen.

At the end of the lesson, I had introduced the flags, formally known as the condition code register, whose purpose is to provide simple indications about the most recent instructions executed by the 6809 processor. In this case, by comparing the value in the X register to a known value, and subsequently checking the condition codes, it is possible to determine when the complete message has

Machine language programming actually begins in this lesson. You'll be needing your editor/assembler EDTASM+ now, so be sure to have your copy before beginning this session.

* What is the address range of the 6809 processor, in hex.

$0000 to $FFFF

* How many bytes does the A accumulator hold?

One byte.

* How many bytes does the X register hold?

Two bytes.

* X and Y are what kind of registers? Why?

Index registers; because they index an address in memory.

* What does the program counter (PC) indicate?

The memory address containing the next instruction the processor is to act upon.

* What is the formal name for the flags?

The condition codes, or the condition code register.

Mnemonics

* There is a set of verbal descriptions of processor commands; what are these descriptions called?

Verbal descriptions of processor commands are called mnemonics.

* How is "mnemonics" pronounced?

It is pronounced nuh-MON-ix.

* What do mnemonics represent?

Processor commands.

* What is the proper name for a processor command?

A processor command is an operation code, or opcode.

* One processor command is written LDX. What does this mean?

LDX means "load X register".

* What is LDX?

LDX is an opcode meaning "load X register".

* What is STA? What does STA represent? What does STA mean? What action does it cause?

STA is a mnemonic; it represents an opcode; the opcode means "store A accumulator"; it causes the contents of the A accumulator to be stored in memory.

* Describe CMPX. What is it? What does it represent? What does it mean? What action does it cause?

CMPX is a mnemonic; it represents an opcode; the opcode means "compare X register"; it causes the value of the X register to be compared with another value..

been displayed. I used an example in BASIC to outline the process, and finished by having you load and examine a mnemonic source code. Load that program again — it follows on this tape — and then I'll talk about mnemonics and source code, and what they mean.

> Program #8, an EDTASM+ program. Insert the EDTASM+ cartridge, and turn on the power to your computer. When the cursor appears, type L and press ENTER. The computer will search (S) and find (F). When the cursor reappears, display the program. Type P#:* and press ENTER. If the right-hand side of the program is not similar to the listing, or if an I/O error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix.

```
00100                LDX     #$0600
00110                LDY     #$0400
00120 LOOP           LDA     ,X+
00130                STA     ,Y+
00140                CMPX    #$0800
00150                BNE     LOOP
00160                RTS
00170                END
```

We'll spend a session learning to use the editor/assembler a little later. For the moment, print this listing on the screen by typing P followed by ENTER. What you see should almost look familiar from the descriptions of the processor instructions you've been getting from me.

What you're looking at are mnemonics, somewhat verbal descriptions of processor commands. I'll read the commands in the third column. Load X, Load Y, Load A, Store A, Compare X, Branch if Not Equal, Return from Subroutine. One more time, just for familiarity. Load X, Load Y, Load A, Store A, Compare X, Branch if Not Equal, Return from Subroutine. These commands are called operation codes, or Op Codes.

LoaD X register
STore A accumulator
CoMPare Y register
ReTurn from Subroutine

In the fourth column you'll see the Operands, those values and indications *used* by the Op Codes. I'll read the third and fourth columns together, which provides a complete description of each 6809 processor instruction in turn. Here goes.

- Load X with the immediate value hexadecimal **0600**

- Load Y with the immediate value hexadecimal **0400**

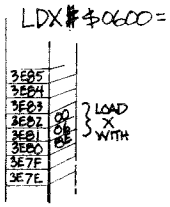
- Load A with the value from memory indexed by X, and increment X by one

- Store A to the value in memory indexed by Y, and increment Y by one

- Compare X to the immediate value of hexadecimal **0800**

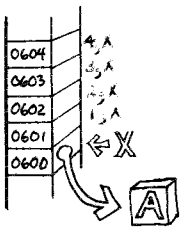
- Branch if the result of the previous computation was not zero, that is, if not equal, back to the instruction labeled LOOP.

- Return from subroutine. The return is used here only because this program is a machine-language subroutine we have used from BASIC. This RTS gets the processor back to BASIC.



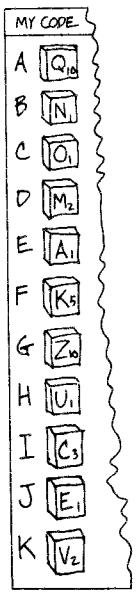
I've used some new terms. "Immediate" value is one of them, one which I slipped into the previous lesson. "Immediate" is a piece of jargon I'm not fond of, but it's the formal term meaning "use this actual number". In line 100, that means Load X with the number hex **0600**. The number sign preceding the value is used to indicate an immediate operand.

The rest of the listing should look fairly straightforward. The plus signs after X and Y mean automatically increment those registers by one. There are also ways of incrementing by two, or decrementing by one or two. Later for that.



But one thing might look peculiar, and that's the comma sitting in front of the X and Y in lines 120 and 130. To my eyes, that comma's a beautiful thing; it gives me computing power. Line 120 could have been written another way: **LDA 0,X+** . . . which means, Load A with the value in memory indexed by the X register plus an offset of zero. One more time. **LDA 0,X+**. Load A with the value in memory indexed by the X register *plus* an offset.

In this program, the offset value is an implied zero. It's implied by leaving it out. In effect, the A accumulator gets its value simply from the memory location indexed by the X register. If X is **$0600**, A loads its value from **$0600**. No problem.



But that offset can be an astoundingly powerful thing. Most kids have written letters to friends in code. They mix up the letters and ever so seriously send the message. Cryptogram puzzles work that way, too. Using the 6809's amazing indexed-offset technique, encoding — and decoding — that kind of message becomes a snap. I remember making off with a Scrabble set to write my cryptograms. I would sort out one alphabet of Scrabble tiles, and then write out the letters of the alphabet in order on a large sheet of paper. Then I'd shake up the letters and put them down on my paper, one at a time. A might be X, B would be L, C would turn into N, who knows. That would be my code. I would write my message and carefully code it, letter by letter.

Get a pencil and a large piece of paper. In one line across the paper, write the letters of the alphabet in a mixed-up order. When you've finished that, write, *in order*, the hex numbers **$00** to **$19** above those letters. The letters will be out of

```
* What is the name for a machine
instruction?

An opcode.

* What is the name for a value
or indication used by an
opcode?

An operand.

* Read the mnemonic LDX.

Load X register.

* Read the mnemonic LDX #$0600.

Load X register with the
immediate value hexadecimal
0600.

* What does immediate mean?

Use the actual value, the value
immediately following the
opcode.

* What symbol is used to
indicate an immediate operand?

The number sign or crosshatch
(#).

* What symbol is used to
indicate hexadecimal notation.

The dollar sign ($).

* Write the mnemonic for "load
the Y register with the
immediate value hexadecimal
1234".

LDY  #$1234

* Write the mnemonic for the
instruction "load the X register
with the immediate value 0"

LDX  #0     or
LDX  #0000  or
LDX  #$0000 or

* What does the comma indicate
in the mnemonic LDA  ,X ?

The comma indicates an offset.
```

* What is the offset in the mnemonic LDA ,X ? Why?

The offset is zero because it is not specified.

* What does the comma indicate in the mnemonic LDB $43,Y ?

The comma indicates an offset.

* What is the offset in the mnemonic LDB $43,Y ?

The offset is $43.

* Write the mnemonic for the instruction "load the A accumulator with memory indexed by X, with an offset of hexadecimal $9C".

LDA $9C,X

* What action does the mnemonic opcode LDX #$CCCC perform?

It loads the X register with the immediate value hexadecimal $CCCC.

* What action does the mnemonic opcode LDA $33,X perform?

It loads the A accumulator with the value found at memory indexed by X, with an offset of hexadecimal $33.

* You find these instructions:
LDX #$CCCC
LDA $33,X
From what memory location does A get its data?

$CCFF, that is, $CCCC offset by $33.

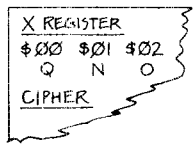
* What is the ASCII value for the letter A (in hex)?

Uppercase A is $41, lowercase a is $61

* What is the ASCII value for the letter Z (in hex)?

Uppercase Z is $5A, lowercase z is $6A.

order, but the hex numbers will be in order. Turn this tape back on when you're finished; turn the tape off now.



Now you've got 26 rearranged letters and 26 hex numbers in order. Above letter **$00** write "X Register". Below letter **$00** write "CIPHER". CIPHER is a convenience label that will identify the start of the coded alphabet. That's "X Register" above letter **$00** and the label "CIPHER" below letter **$00**.

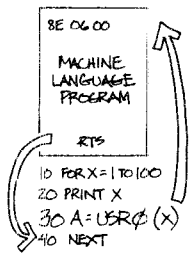
And now to the program. The idea here is to be able, given a value from somewhere, to extract the coded value from the table and provide it to the user.

ALPHA LETTER	ASCII CODE	CONSTANT OFFSET	TABLE POSITION
A	41	- 41	0
B	42	- 41	1
C	43	- 41	2
D	44	- 41	3
E	45	- 41	4
F	46	- 41	

Let's say the value is in ASCII, a normal state of affairs for these machines. Letter A is ASCII hex **41**, letter Z is hex **5A**. The question is how to get from ASCII values **$41** through **$5A** to the encrypted values in the table, which are numbered **$00** through **$19**. There's really no mystery or wonder to this part. If you subtract **$41** from **$41**, you get **$00**. Subtract **$41** from **$5A**, you get **$19**.

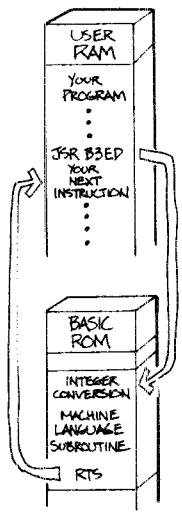
So the ASCII values come in from somewhere, you subtract **$41**, and the resulting number is the position of the encrypted value in the table. You extract the value from that position, and the encoding is done.

There's a program to write now, during which I'm going to introduce some new parts of the 6809 architecture. This would be a good time to take a break and review what's been done so far. When you've finished reviewing, open your Extended Color BASIC manual, and read pages 145, 146, and all except the last paragraph on page 147. Don't worry if you don't understand all of it; I'll explain later.



> Please read pages 145, 146 and 147 in the Extended Color BASIC manual. This is the beginning of the chapter called "Machine Language Routines".

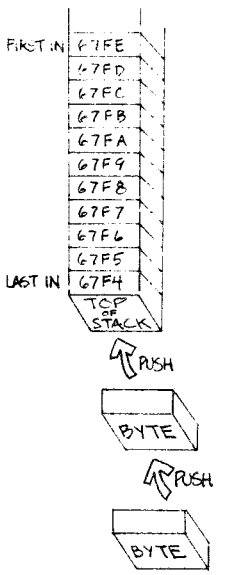
The program you have to create will accept an ASCII value, subtract a constant, and use the result to pluck a number from a table of encrypted letters.



You'll actually be creating a working program, so you need a jumping off place. BASIC is good. You can transfer a value from BASIC to machine language; it's part of the USR command. In your Extended Color BASIC book, the USR function was described. The "argument" they're talking about is the value transferred to a machine language program from BASIC, and that will be the ASCII value you are going to encrypt. Once control is given over to your machine language program from BASIC, *your* program must obtain that ASCII value.

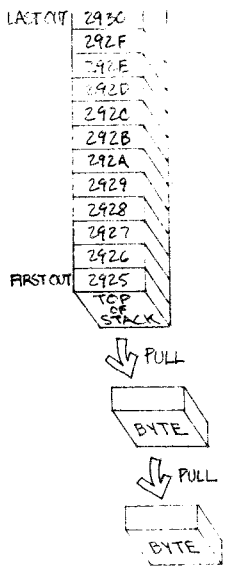
When USR is executed by BASIC, the first step is done for

you. The value is waiting in memory, and part of BASIC's own machine language commands are set up for your use. The Extended Color BASIC manual described this process of transferring your integer ASCII value by saying, "It's possible to force the argument to an integer by calling BASIC's INTCNV routine from the USR function (INTCNV = X'B3ED')." I'll tell you what that means. It means you can transfer an integer from BASIC to a machine language program by using a part of BASIC found at address **$B3ED**. *Your* program must consider the chunk of BASIC beginning at **$B3ED** to be its own subroutine.



Subroutines in machine language are almost identical in principle to the GOSUBs in BASIC, except that you have to know more about them. Primarily, you have to know about the stack. Return to your MC6809E data booklet, and look again at Figure 4 on page 5. Notice that below the X and Y registers are two registers marked User Stack Pointer and Hardware Stack Pointer.

The stack is one of the best- and worst-named registers in microprocessor programming. It's well named because it is, in fact, a stack full of bytes being temporarily stored. You put things on the stack in first-in, last-out order. That is, it's like that pile of magazines on your coffee table. The first magazine you stacked there is the last magazine that gets taken off the table because everything else is on top. Go look. I bet you didn't realize there was still a January 1975 *Reader's Digest* underneath all that.



Seriously, the stack is a register which points to a memory location. The address being pointed to changes as the stack grows or shrinks. But the stack is badly named because it works upside-down. It's what's known as a "push-down" stack. Every time I push a byte on the stack, the address decreases by one. It's like stacking those magazines on the ceiling. For the moment, just remember first-in, last-out.

The reason you have to know about the stack to use a subroutine is because it is on the stack where the 6809 processor puts the present address in its PC register — the program counter — when it jumps to a subroutine. It breaks the address into two bytes of data, pushes the two-byte address on the stack, and puts the address of the subroutine in the program counter. The next instruction, so far as the program counter knows, is now at the beginning of the subroutine! It goes along, executing instructions in the subroutine, until it comes across the command **RTS** (return from subroutine). The instruction decoder pulls that original two-byte address off the stack, reconstructs it, puts it in the program counter, and presto! you're back where you left off in the original program.

Some jargon now. This is known as a subroutine call, and its mnemonic is **JSR** — jump to subroutine. As I said, it works just like a BASIC GOSUB, and like BASIC, you can nest your subroutines — call one from inside another from inside another. But here's where the difference shows up. You don't have to keep track of much in BASIC — it

```
* How many letters are there in
the alphabet (in hex)?

There are $1A letters in the
alphabet.

* If A is considered letter
number $00, what is letter Z?

Letter number $19.

* If the X register points to a
memory location that contains a
special code for letter A
(letter number $00), write a
single mnemonic command to load
the A accumulator with the
special code for letter Z.

LDA $19,X

* How does BASIC transfer a
value to storage for use by a
machine-language program?

With the USR command.

* What is needed with the USR
command to transfer a value to
storage for use by a
machine-language program?

It needs an argument following
the command.

* If M is a BASIC variable, and
the value to be transferred is
149, write a USR command to
transfer a value to a
machine-language program.

M=USR(149)

* At what memory location does
BASIC's integer conversion
routine begin?

The integer conversion
subroutine starts at $B3ED.

* What does the mnemonic JSR
mean?

Jump to subroutine.

* What register does a jump to
subroutine require?

The stack.
```

Pushing and pulling

* Why does a jump to subroutine require the stack?

To store the current position of the program counter to use as a return address.

* What type of stack is found in the 6809 processor?

A push-down stack; or, a first-in last-out stack.

* What command places the program counter on the stack?

JSR, jump to subroutine.

* What command places the original address back in the program counter?

RTS, return from subroutine.

* What action does the command JSR $B3ED descibe?

Jump to subroutine at memory location $B3ED.

* What is the process of placing a value on the stack called?

Pushing.

* What is the process of taking a value off the stack called?

Pulling.

* What does the program counter (PC) keep track of?

The next instruction the processor is going to follow.

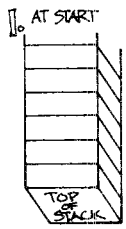
* At address $1000, a command is encounterd whose mnemonic is JSR $B3ED. Upon execution of JSR $B3ED, what value is pushed on the stack?

$1003.

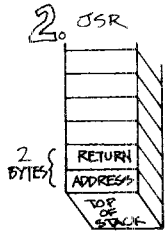
* How many bytes are pushed onto the stack when JSR $B3ED is executed?

Two.

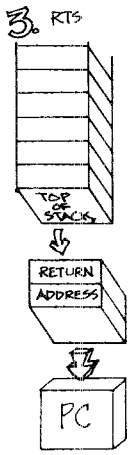
"cleans up" for you. But you've got to know where your machine language stack is, because it's also used to save information for later use.



Refer again to the Extended Color BASIC manual, on page 147, entitled "Returning to BASIC from a USR Function". It states, "The values of A, B, X and CC registers need not be preserved by the USR function." That implies that the value in the Y register is needed; how do you save it? By pushing it on the stack, that's how. Once the two bytes that make up the 16-bit Y register get pushed on the stack, you can then modify Y as you wish. Before returning to BASIC, pull Y from the stack, and off you go.



If you're ahead of me, then you're asking, "which stack?" The MC6809E data booklet indeed stated that there is both a User Stack and a Hardware Stack. Subroutine calls automatically use the Hardware Stack, so that's a certainty. For pushing and pulling various values, you might use either of the remaining stacks. But because of the complex software in the Color Computer, the User Stack is basically reserved. For the most part, stay away from it. The Hardware Stack is what's left.



Now the mnemonics. To push a value on the Hardware Stack, the mnemonic is "pushstack" — PSHS. The operand is the set of registers you wish to push. To push X, Y, and A, for example, you would pushstack X Y A — **PSHS X,Y,A.**

So where are you? You've got an encrypted ASCII alphabet in a table, you know you have to save the Y register for BASIC, you know that **$B3ED** is the address of the integer-conversion subroutine. Page 149 of the Extended Color BASIC manual tells you that **$B4F4** is the subroutine call that properly returns an integer value to BASIC. All that's left is to write the program. If you need it, now's the time to take a break and review.

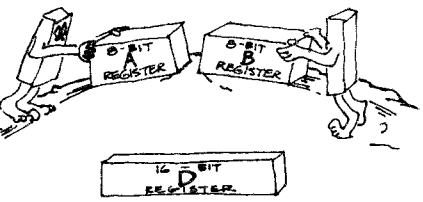
Now to the program; do it on paper first. The Y register must be saved, so pushstack Y — write **PSHS Y**. Now there's the matter of getting the value waiting in BASIC. Jump to the subroutine at **$B3ED** for that. Write **JSR $B3ED**. The manual tells you that the value from BASIC is returned in the D register. What's that? It's merely the name for both A and B 8-bit accumulators used as if they were a single 16-bit accumulator. Since the value is an ASCII character, it is only one byte in size, fitting into the B accumulator.

The encryption table has to be identified. Write Load X with immediate value CIPHER. Write "LDX" and across from it write "# CIPHER". The X register is pointing to the zeroeth entry in the encrypted ASCII table.

Remember that **$41** has to be subtracted from the ASCII value to get it into the range **$00** to **$19**. Subtract the immediate value of **$41** from the B register; that is, subtract from B immediate value **$41**. Write **SUBB #$41**.

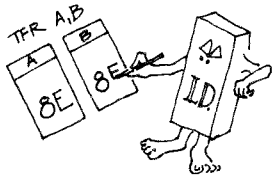
The magic is next. You know that the B register contains a value from **$00** to **$19**. You know that X is pointing to the zeroeth value in the encrypted table. All that's left of the hard work is to use that information to find the value you want from the table. That value is found at the address indexed by X, plus the offset value found in register B. Load A with value indexed by X offset by B. Write **LDA B,X**. You've got it.

The Extended BASIC manual says that to get the value back to BASIC, it has to be in the D register — remember that's A and B used as one register — and **$B4F4** has to be called. That means the value now in A has to be placed in B, since the B register is the least significant byte of the D register. There's a transfer instruction for that . . . transfer A to B. Write **TFR A,B**.



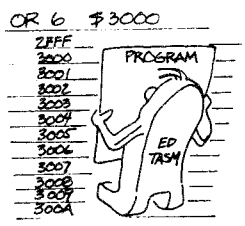
Now A and B contain the same value. You want A to be zero, so clear it. Write **CLRA**. It looks like most of the work is done, so call that routine that gives the value to BASIC. Write **JSR $B4F4**. Now get the Y register back (you do remember you saved the Y register, don't you). Pullstack Y. Write **PULS Y**. And finally, it's back to BASIC — return from subroutine. Write **RTS**.

There's a tape to load now. When you're done with that, take a break.



Program #9, an EDTASM+ program. Insert the EDTASM+ cartridge, and turn on the power to your computer. When the cursor appears, type L and press ENTER. The computer will search (S) and find (F). When the cursor reappears, display the program. Type P#:* and press ENTER. If the right-hand side of the program is not similar to the listing, or if an I/O error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix.

```
00100 CIPHER  EQU     $3000
00110         ORG     $3100
00120         PSHS    Y
00130         JSR     $B3ED
00140         LDX     #CIPHER
00150         SUBB    #$41
00160         LDA     B,X
00170         TFR     A,B
00180         CLRA
00190         JSR     $B4F4
00200         PULS    Y
00210         RTS
00220         END
```



Type P#:*, <repeat> and hit <ENTER>. There are just a few new things in this listing. Line 100 contains the notation **CIPHER EQU $3000**. This line tells the editor/assembler that the label CIPHER is to mean hex **3000**. So whenever it encounters the label CIPHER, the editor/assembler knows to work with the value **$3000**. This is called an "equate", and it makes life easier for you as a

* Using the previous example, upon a return from subroutine (RTS), what value is placed into the program counter (PC)?

$1003.

* Other than JSR, what instruction type places a value on the stack?

Push.

* How many stacks are there in the 6809?

Two.

* What are the names of the two 6809 stacks?

The user stack (U) and the hardware stack (S).

* Which stack do subroutines use automatically?

The hardware stack.

* What is the mnemonic for the command to place a value on the hardware stack?

Pushstack S, or PSHS.

* Write the mnemonic for pushing the X register on the hardware stack.

PSHS X

* Write the mnemonic for pushing the A accumulator on the hardware stack.

PSHS A

* Write the mnemonic for pushing both the A accumulator and X register on the hardware stack.

PSHS A,X

* What is the mnemonic for taking a value off the hardware stack?

Pullstack S, or PULS.

*** Write the mnemonic for taking the X register off the hardware stack.**

PULS X

*** Write the mnemonic for taking the A accumulator off the hardware stack.**

PULS A

*** Write the mnemonic for taking the B accumulator, X register and Y register off the hardware stack.**

PULS B,X,Y

*** If the value of the X register is $1234 and at address $1000 the program executes JSR $E3ED, what values would be found on the stack, from first in to last in?**

First in is $34, then $12, then $03, then $10.

*** Using the previous example, what would be the result after these two instructions:**
RTS
PULS Y

The main program would be returned to ($1003 back in the program counter) and Y would be $1234.

*** The previous example made Y equal to the value of X. What other instruction could have made Y equal to the value of X?**

Transfer X to Y (TFR X,Y)

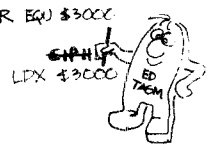
*** What does ORG mean?**

ORG means origin, the first memory location used in a mnemonic listing.

*** What does ORG $3F80 mean?**

It means the first memory location in a mnemonic listing is $3F80.

programmer. You can remember meaningful labels instead of heaps of numbers.



The other new item is in line 110, reading **ORG $3100**. This means that the origin, or first address, of your program will be memory location **$3100**.

Beyond that and the **END** statement in line 220, this program should look exactly like the one you wrote down. This is the source code for the encryption program — the mnemonic representation of the instructions you want the 6809E processor to follow.

Do a few things mechanically now; I want you to try the program, but I'm not ready to explain all about the editor/assembler. Some of that's for next time. Type A/IM/AO. I'll repeat that. A/IM/AO. Hit <ENTER>. A listing should be scrolling by, and your star prompt will return. The editor/assembler has just turned your mnemonic code into a group and 6809 instructions, and placed them in memory. Briefly, A means assemble the program; IM means assemble it into memory, and AO means absolute origin, that is, assemble the program exactly where your ORG statement says to do it.

Now Quit the editor/assembler. Type Q and hit <ENTER>. You will be in BASIC now, and I have another short program for you to load.

Program #10, a BASIC program. Turn on the power of your Extended Color BASIC computer. When the cursor appears, type CLOAD and press ENTER. The computer will search (S) and find (F). When the cursor reappears, LIST this program. If the program is not similar to the listing, or if an I/O error occurs, rewind to the start of the program and try again. For severe loading problems, see the Appendix.

```
10 DEFUSR0=&H3100
20 X=90:FORN=&H3000 TO &H3019:POKEN,X:X=X-1:NEXT
30 A$=INKEY$:IFA$("A" OR A$)"Z"THEN30
40 A=ASC(A$)
50 B=USR(A)
60 PRINTCHR$(B):
70 GOTO30
```

You've listed this program. Line 10 defines your USR program to be at hex **3100**, the origin you used. Line 20 places the letters of the alphabet in reverse order in memory starting at **$3000** — where the #CIPHER encryption table is supposed to be. Line 30 is an ordinary INKEY$ that picks off an uppercase character as you type it. Line 40 gets the ASCII value of the letter. So far, everything is BASIC you probably know, nothing special.

Finally, line 50 transfers the ASCII value to the machine language program and executes the program. When the machine language program is done, it returns to BASIC. Line 60 prints the ASCII character represented by the

value transferred back from the machine language program. Line 70 repeats the process.

RUN the program, and begin typing the alphabet. I'll be with you next time. Be sure to review this lesson before then.

* When using the editor/assembler, what does the A command mean?

A means assemble the mnemonic code into a group of 6809 instructions.

* When using the editor/assembler A command, what does /IM mean?

/IM means to assemble the mnemonic code into 6809 instructions, and place them in memory.

* When using the editor/assembler A (assemble) command with /IM (in memory), what does /AO mean?

/AO means to assemble the mnemonic code into 6809 instructions and place them in memory at the origin specified in the ORG line.

* The source listing says ORG $2400. You enter A/IM/AO. Where is the first byte of your source listing placed in memory?

At location $2400.

5.

You've been using mnemonics lately in creating machine language programs, and I think that's gotten away from the binary instructions themselves. It's these binary instructions which are doing the work; the mnemonics are how you and I remember what the instructions are and how they operate. For example, one of the instructions in the last session was to load the X register with the value labeled CIPHER. CIPHER in turn was address hex **3000**. Load X with an immediate value is in fact hex code **$8E**.

The purpose of the editor/assembler is to make programmers' lives easier by accepting understandable mnemonic statements like "Load X immediate CIPHER" and turning them into machine codes like hex **8E 30 00**. The mnemonics do make the program look long and complicated, but in fact, in spite of all the apparent typing, the entire program consists of 21 bytes!

I'd like you to load that encryption program again.

> Program #11, an EDTASM+ program. Insert the EDTASM+ cartridge, and turn on the power to your computer. When the cursor appears, type L and press ENTER. The computer will search (S) and find (F). When the cursor reappears, display the program. Type P#:* and press ENTER. If the right-hand side of the program is not similar to the listing, or if an I/O error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix.

```
            3000        00100 CIPHER  EQU     $3000
3100                    00110         ORG     $3100
3100 34     20          00120         PSHS    Y
3102 BD     B3ED        00130         JSR     $B3ED
3105 8E     3000        00140         LDX     #CIPHER
3108 C0     41          00150         SUBB    #$41
310A A6     85          00160         LDA     B,X
310C 1F     89          00170         TFR     A,B
310E 4F                 00180         CLRA
310F BD     B4F4        00190         JSR     $B4F4
3112 35     20          00200         PULS    Y
3114 39                 00210         RTS
            0000        00220         END
00000 TOTAL ERRORS
CIPHER  3000
```

Coming up in this lesson are the hows and whys of using the editor/assembler, and a reminder that its convenience features are just that -- conveniences. They are in no way a replacement for the awareness of what the machine language is actually doing.

* When a word like CIPHER appears in a mnemonic listing, what is it called?

A label.

* Is a label part of the program?

No, it is part of the source listing.

* Are the mnemonics the program?

No, they form the source listing.

* This is the hex code the program?

No, the hex code isn't the program either..

* Then if labels nor mnemonics nor hex code aren't the program, what is?

The binary machine instructions and data.

* If the label CIPHER is set to $3000, and the mnemonic LDX #CIPHER is assembled, what is the binary result?

Hex $8E 30 00, that is, 10001110 00110000 00000000.

* What does ORG mean?

Origin.

* What is the origin?

The first byte of an assembly listing.

* What is an organized group of labels, mnemonics, and operands called?

An assembly listing or the source code.

* What is the source code used to produce?

Object code.

* What is object code?

Binary instructions and/or data.

* How is object code produced from source code?

By assembling it.

* There are four columns in an EDTASM+ source code listing. What is in the first column?

The source reference line number.

* What is in the second column of an EDTASM+ source code listing?

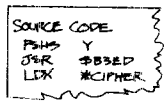
An optional label.

* What is in the third column of an EDTASM+ source code listing?

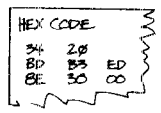
The opcode.

There's the program listing in front of you. Let me refresh your memory as to what this means. The label CIPHER was used to indicate a memory location **$3000**. The origin, that is the first instruction, of the program itself was set in memory at **$3100**. My choices here were arbitrary; and free memory could have been used. Since this program was to be used in conjunction with BASIC, the first action was to save the Y register on the stack, as recommended by the BASIC manual. Next, BASIC's integer-conversion subroutine was used to transfer the value from the BASIC USR function to your program; again, this information was recommended by the manual, a recommendation you have to trust.

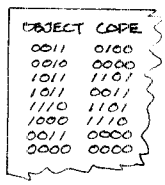
The X register was indexed to the first entry in a table of encrypted ASCII values. **$41** was subtracted from the B accumulator — recall that the B register contained the value after the integer conversion — to provide an offset of **$00** to **$19** to the encryption table. In line 160, the A accumulator loaded from memory indexed by X, with an offset of B, that encrypted ASCII value. In preparation for sending this value back to BASIC, it was transferred from A accumulator to B accumulator, and A accumulator was cleared to zero. Finally, the Y register was retrieved from the stack, and a return from subroutine landed the program back in BASIC.



I repeat that this is mnemonic code — code which serves as a kind of verbal reminder to you and I as programmers — but is not in itself something the 6809 processor can use. The 6809 can only understand simple binary instructions and data; the editor/assembler converts your mnemonic code into those binary instructions and data.

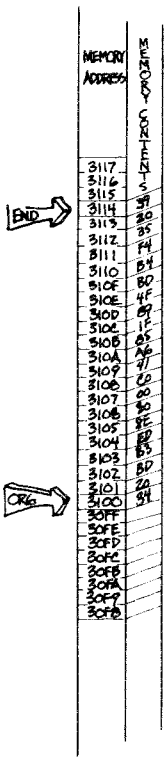


In this lesson, I want to guide you in using the editor/assembler, but first I would like you to see exactly what it's for. Type A, and hit <ENTER>. You'll see the "READY CASSETTE" message, meaning it's about to prepare an object code tape. "Object code" is the jargon for a set of binary instructions and data. Don't worry about inserting a tape now; just hit <ENTER> again. The tape recorder relay will click on, and after a short pause, the screen will scroll quickly by, filled with both your original source code and with additional hexadecimal numbers.



Reading the short, 32-character screen is tricky, so with all of these assembled programs, I've provided a printed listing for reference. Take a glance at the program in your documentation. It looks much like the original source code — in fact, it *includes* the entire source code — but there are several additions to it. All these additions are displayed in hexadecimal notation.

In the first column, the memory locations, that is the memory addresses to hold the program, are presented in hexadecimal. In this case, the program's first instruction begins at **$3100**, and the last instruction is found at **$3114**. The second and third columns contain the actual instructions and data that will be placed in memory for the 6809 processor to execute.

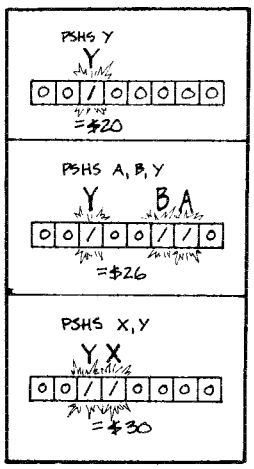


The second column contains the Opcode (that is, the operation code or instruction), and the third column contains the Operand (that is, the data the processor uses). I'll take each in order.

Opcodes first; follow down the column with me. The opcode to push a value on the hardware stack is **$34**. The opcode to make a subroutine call is **$BD**. **$8E** loads the X register with an immediate value, **$C0** subtracts an immediate value from the B accumulator, **$A6** loads the A accumulator in an indexed mode, **$1F** transfers a value between registers, and **$4F** clears the A accumulator to zero. Another subroutine call follows; that's **$BD**. The opcode to pull a value from the stack is **$35**, and a return from subroutine is **$39**.

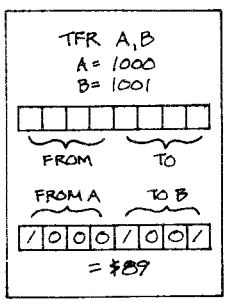
Each of these opcodes, after interpretation by the processor's internal instruction decoder, gives the 6809 information about what to do, what data is coming up next, and how many bytes long the operand will be. The operands themselves vary according to what the instruction demands. In lines 130, 140 and 190, for example, it's clear that the operands **$B3ED**, **$3000** and **$B4F4** are addresses, the first for a subroutine, the second for loading into the X register, and the last another subroutine. In line 150, the operand **$41** is the immediate value subtracted from the B accumulator.

Lines 120, 160, 170, and 200 are another matter. Here the operands are not immediate values, but rather informational data on how to complete the instruction. Look at line 120, for example; the mnemonic says "pushstack Y". As I've said, the opcode for pushstack is **$34**. How about that hex **20**?



Pull out your MC6809E data booklet, and turn to page 18. On page 18, find the heading PULU/PULS. There are two short tables under the heading marked "Pull Order, Push Order". You are looking at the order in which registers are placed on the stack, you're also looking at the individual binary digits within a byte.

The command you used was Push Y. Examine the table, and find the Y register. The Y register is third from the left, the position of bit 5. If you write a binary equivalent of this row of registers, where a binary one indicates which registers to push, then you would write **0010 0000**. That binary number is hex **20** . . . the precise operand assembled in line 120.



I don't want to browbeat you with bits and bytes, but it's extremely important to be aware, to keep in the back of your mind at all times, what these binary codes do. You don't need to memorize any of them; that's what your data booklet is for. But knowing how to interpret what you're seeing is key to effective programming and efficient debugging.

Let me give you just one more example of these binary operands. Keep your place on page 18 of the MC6809E

* What is in the fourth column of an EDTASM+ source code listing?

The operand, where required.

* The four columns in an EDTASM+ source code listing are...

The reference line number, the label, the opcode, and the operand.

* When an EDTASM+ source code listing is assembled, what information is added to the displayed listing?

The hexadecimal address and memory contents.

* How many extra columns of information are added when an EDTASM+ source code listing is assembled?

Three columns are added.

* What is in the first column of the assembled listing?

The address, in hexadecimal.

* What is in the second column of the assembled listing?

The opcode, in hexadecimal.

* What is in the third column of the assembled listing?

The operand, in hexadecimal.

* In an EDTASM+ source listing, how many columns are displayed?

Four.

* In an assembled EDTASM+ listing, how many columns are displayed?

Seven.

* What do the seven columns of an assembled EDTASM+ listing represent?

The address in hexadecimal; the opcode in hexadecimal; the operand in hexadecimal; the reference line number; an optional label; the opcode in mnemonics; the operand in mnemonics.

* What part of the assembled EDTASM+ listing is the machine language program?

No part of the assembled EDTASM+ listing is the machine language program.

* What is the machine language program?

It is the object code, or binary information.

* What does the A command instruct EDTASM+ to do?

To assemble the object code.

* Where is the final object code placed?

On the cassette tape.

* What does the command A/IM instruct EDTASM+ to do?

To assemble the object code into memory.

* What does the command A/IM/AO instruct EDTASM+ to do?

To assemble the object code into memory at the origin specified in the program listing.

* What is the assembler word for origin?

ORG.

* What does the mnemonic PSHS Y mean?

Push the Y register on the hardware stack.

data booklet, and look at line 170 in the program — the instruction is transfer A to B. The transfer opcode, as noted, is **$1F**. On page 18, under the heading TFR/EXG, you'll see combinations of four binary digits. Each combination represents a specific register. The "transfer from" register makes up the left-hand four digits of a byte; the "transfer to" register makes up the right-hand four digits. According to the chart, then, transfer from A to B should put a value of **1000** in the "from" position and **1001** in the "to" position, creating a complete binary word of **1000 1001**. **1000 1001**, you should expect by now, is hex **89** — the same value as the operand assembled in line 170.

Next in this lesson I will be guiding you through the entry and editing of source code using the editor/assembler EDTASM+. I recommend you take a break and review now, and when you are done with your break, turn to page 3 of the EDTASM+ manual, and read the Introduction.

Read and review the EDTASM+ introduction. The introduction is printed on the facing page; for more detailed information, continue with the EDTASM+ manual. Return to the tape when you have completed the reading.

Time to start fresh. If you've just come back from reading the EDTASM+ Introduction, your computer is probably up and ready to go. Even so, please turn the computer off, insert the editor/assembler EDTASM+ cartridge in the slot, pause, and turn it back on. The star prompt will come up shortly. I'm going to give you some guidance in entering, editing, and assembling your source and object code with the EDTASM+ program.

The first thing to remember is that EDTASM+ is a programmer's program. It doesn't have the fanciness and fussiness of BASIC, and it can't tell you if you've written a program that will work. Its job is exclusively to translate mnemonic source code into binary object code, and inform you if you've typed the source code incorrectly or made an error in labeling or numerical range, or if you have asked the processor to perform a function it's incapable of. (Another feature of the EDTASM+ program cartridge is ZBUG, but that's not for this time.)

To help you achieve your programming ends, the editor keystrokes are minimal and the editor's commands are few. If you are using an editor/assembler other than EDTASM+ (which you may remember I didn't recommend) these instructions will apply only in part; many of the specifics will be quite different. What all 6809 editor/assemblers have in common, however, is the mnemonic source code.

Time to start. Your most frequent editor commands will be Insert, Delete, Print, Number, and Edit. Just for reference

EDTASM+

The brain of the Color Computer is the 6809 Microprocessor. It is always operating in 6809 machine code, the only language it knows.

When you program in BASIC, a ROM program called the BASIC Interpreter "translates" each statement, one at a time, into 6809 machine code.

The Editor-Assembler + allows you to write a program in 6809 assembly language and assemble it into a single, efficient 6809 machine code program. This gives you two very powerful advantages:

- You are no longer limited to the commands in the BASIC language.
- Many steps that are necessary to interpret a BASIC statement into machine code will no longer be needed. Therefore, the programs you write with the Editor-Assembler + will run much faster, and probably use less memory.

This manual demonstrates how to use the Editor-Assembler +. It will not teach you how to program in assembly language. Radio Shack has an excellent book devoted to the subject. It's Catalog Number is 62-2077. You can purchase it through any Radio Shack store.

The Editor-Assembler + contains three systems:

- ***The Editor***, for writing and editing 6809 assembly language programs.
- ***The Assembler***, for assembling the programs into 6809 machine code.
- ***ZBUG***, for examining and debugging your machine code programs.

To use them, all you need is a Color Computer with 16K RAM and a tape recorder.

How You Will Use These Systems

1. First you'll *write the program* in assembly language, using mnemonics which the Assembler recognizes and which is fairly easy to use. This is done in the Editor and the resulting program listing is called TEXT.
2. Then you'll *assemble* the instructions of TEXT into machine code which the 6809 Microprocessor can recognize, but which looks like nonsense to most people. Thus, you'll create CODE consisting of op codes and data.
3. You'll use ZBUG to *test and debug* CODE until it's perfect. Then you'll store it on tape. Storing CODE is the final task of the Editor-Assembler +.
4. From BASIC, you'll *load* CODE (with CLOADM) *and run it*. You can either run it as a stand-alone program (with EXEC) or as a subroutine (with USR).

* What is the hexadecimal opcode for PSHS?

$34

* How is does the operand for opcode $34 (PSHS) identify which registers are to be pushed?

By the order of the binary digits in the operand.

* The order of the binary digits for the push operand is PC, S (or U), Y, X, DP, B, A, CC. What is the binary operand to push registers A, B, X and Y on the stack?

00110110.

* What is the hexadecimal value for binary 00110110?

$36

* What is the hexadecimal value for the opcode PSHS?

$34

* What is the complete hexadecimal instruction PSHS A,B,X,Y?

$34 36

* Once again, the order of binary digits for stack pushing is PC, S (or U), Y, X, DP, B, A, CC. What is the operand, in binary and hexadecimal, for PSHS X,B?

Binary 00010100, hexadecimal $14.

* What is the complete instruction, in binary and hexadecimal, for PSHS X,B?

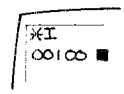
Binary 00110100 00010100, hexadecimal $34 14.

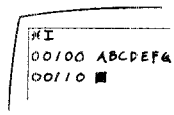
* What is another name for this kind of operand?

A postbyte.

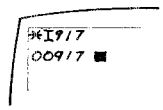
as you go along, I'll tell that you can get out of any EDTASM+ mode by hitting <BREAK>.

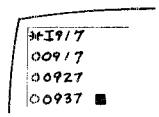
There is no requirement to manually number every line in EDTASM+, saving you considerable time and energy. Simply type and enter 'I'. The first available line number, 00100, is presented with the cursor ready for your information. You may now type anything you like on this line. Since renumbering and block search can be done, and since the editing commands are identical to BASIC's and already familiar to you, you might even want to use the editor as a low-grade word processor. For this lesson, though, the point is to develop 6809 mnemonic code. To practice, type something now . . . a few letters or numbers, whatever, and hit <ENTER>. The information in that line has been stored, and the next line, 00110, is ready for use. Type some more characters and hit <ENTER> again. Line 00120 is in place. At the start of a session, automatic line insert mode starts at 100 and advances in increments of ten lines. But you may change that any time. Tap <BREAK>.



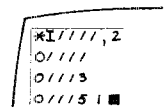


By typing and entering "I917", the editor will begin numbering lines at 917. Type and enter I917. The line 00917 will be presented together with the cursor. Hit <ENTER> a few times. Lines continue to be added in increments of 10, so you should be seeing 00927, 00937, 00947, etc. Tap <BREAK> again.



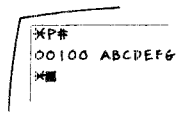


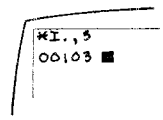
You can change the line increment as well as which lines you are inserting. Type and enter "I1111,2". Line 01111 will be displayed. Hit <ENTER> a few times, and notice that the line numbers do indeed increase by two at a time rather than 10 at a time . . . 01113, 01115, 01117, and so on.



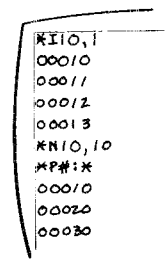
That's the essence of using the editor/assembler's automatic line numbering system.

To look at what you've done, you have to print the information on the screen. To avoid conflicts in the single-letter command system of EDTASM+, the letter "P" was chosen to print to the screen. In EDTASM+, the seemingly more logical "L" doesn't mean list; it means load from tape. So to print a line on the display, simply enter the letter P followed by the line number; leading zeros aren't important. For example, to display line 00110, just enter P110. The line will appear. Try that.





There are many convenience features in the editor/assembler, features which you will find reduces your programming time. To print the next 16 lines on the screen, for example, merely enter "P". Even better are the three symbols for first line, current line, and last line. First line is represented by a number sign (also called the crosshatch or pound symbol. I call it "pound" because it's easier for me to say than "crosshatch" and isn't as ambiguous as "number".). Use a period to indicate current line. The asterisk (the star) indicates the last line. Together with



those, the colon acts as the from-to delimiter, as in "P100:200".

So to print the first line of the program on the screen, just enter "P#". Print the whole program by entering "P#:*". Find your last line by entering "P*". Print the first three lines by entering "P#:120". Display from your current line to the end of the listing by entering "P.:*". With the symbols # for first line, . for current line, and * for last line, you've got complete control of your position within the program with the least amount of typing.

The insert mode uses these convenience features, too. Simply typing "I" requests the editor to insert a line, starting wherever you are now, at the increment you last used. "I.,3" will insert a numbered line at your present point, with an increment of 3 lines. "I#" will attempt to insert a line after the first one in your program, again using the last increment you specified.

Notice that, when you print your text on the display, there are numbered lines with no information. The editor is quite respectful of your requests, and, where you have indeed entered an unused line, it will let it stand. Unlike BASIC, re-entering a line number alone won't get rid of it. With EDTASM+, you must specifically delete unwanted lines with the D command.

Delete also uses the editor's set of convenience features. You can delete any line by entering D and the line number, such as D110. You can delete the first line using "D#", the last line using "D*", or the current line using "D." or just "D". To delete a group of lines, say 1111 to 1115, enter "D1111:1115". Try that. D1111:1115 <ENTER>. To delete the entire text so far, simply enter "D#:*". That's D#:*.

Now attempt to print a listing on your screen . . . enter "P." You'll get one of EDTASM's many full messages, built in to assist your programming without constant reference to the EDTASM+ manual. This message says, "BUFFER EMPTY". Since you have deleted the entire text by entering "D#:*", the editor is giving you the unequivocal confirmation that the text buffer in fact contains no lines.

Type "I10,1", and press <ENTER>. Line 10 will be presented. Type a few characters, and enter this line. Do the same for line 11, line 12, and line 13. Tap <BREAK>, and print the listing by entering "P#:*". Now insert a line between 11 and 12. Try "I11,1" <ENTER>. NO ROOM BETWEEN LINES, eh? Now try this: enter "N10,10". That's "N10,10". You're asking it to renumber, starting from line 10, in increments of 10 lines. Print the listing by entering "P#:*". You should see lines 10, 20, 30 and 40.

Now try entering "I10", as before. Still NO ROOM BETWEEN LINES? Don't forget that the last increment specified is the one the program will use . . . and that

```
* Does the TFR (transfer) opcode
have a postbyte?

Yes.

* Describe the TFR postbyte.

The TFR postbyte is divided in
half; the left (most
significant) half indicates
"from", the right (least
significant) half indicates
"to".

* How many columns are there in
an assembly source listing?

Four.

* What is found in the first
column?

The source reference line
number.

* What EDTASM+ command inserts
lines into the source listing?

The I command.

* How is line 999 inserted into
the source listing?

By entering I999

* What does I1000,5 mean?

Insert lines into the source
listing, beginning at line 1000
and continuing in increments of
5 lines.

* How do you insert lines,
starting with 500, in increments
of 50 lines?

I500,50

* What command displays source
lines on the screen?

The P command.

* How would you display source
line 40?

By entering P40
```

* How would you display the first source line?

By entering P#

* How would you display the last source line?

By entering P*

* How would you display sources lines 40 through 1000?

By entering P40:1000

* How would you display the entire source listing?

By entering P#:*

* What is the symbol for "current line"?

The period (.)

* How would you ask to edit the current line?

By entering E. (E period)

* How would you renumber the listing, with the renumbering beginning at line 1000 and proceeding in increments of 1 line?

N1000,1

* What are the symbols for first line, last line, and current line?

#, *, and . (pound, star and period)

* If your source listing were in increments of ten lines, how would you insert a line halfway between your current line and the next line?

By entering I.,5

increment was specified as 10 when you renumbered the listing. To insert lines between 10 and 20, how about entering "I10,2". There you have line 12, ready to go. Tap <BREAK> now.

The last of your most-used commands will be "E", the key letter for edit mode. E can be used only to edit a line at a time, but the convenience features # . and * are always available. Within the edit mode you have at your disposal all the editing features of Extended Color BASIC. These editing features are quite versatile, but I feel a little outside the scope of these lessons. There's lots more to be done with 6809 assembly language itself.

So here's my proposal. At the end of this lesson, review what has been done so far: binary and hex code, 6809 processor architecture, understanding mnemonics, and so forth. Then spend some time with those few EDTASM+ source programs that have been presented so far. Instead of loading them from tape, try typing them in; by the way, use the right arrow to tab between columns rather than using spaces between columns of source code. Also, turn to your Extended Color BASIC manual and your EDTASM+ manual, and get familiar with those editing features. You'll be using EDTASM+ for the duration of these tapes, and I won't be pausing as long when I describe commands. You'll need to know those editor commands, so put in the time learning its features *now* to make your work much easier later.