Learning the

6809

Micro Language Lab

Dennis Bathory Kitsz

Green Mountain Micro

Roxbury, Vermont

First Edition
Second Printing

The Micro Language Lab:

Learning the 6809

First Edition
First Printing
Printed in the United States of America.

ISBN 0—916015—00—9

Recording: Steve Lusk, Claire Manfredonia, JoAnn Trottier
Recording Supervision and Tape Editing: Dennis Bathory Kitsz
Illustrations: Jim (Doc) Holliday
Cover, Layout and Design: Dennis Kitsz
Typesetting: Northfield News; IJG Inc.
Additional Design and Preparation: M. J. Rufino Associates, Marie Lapre' Grabon
Sherlock Holmes: Peter Clarke
Dr. Watson: Kalvos Gesamte
Automobile: Steve's Honda
Marge: Claire Manfredonia
Cook: Steve Lusk
Mac: RB2—3
The Drivers: Kalvos Gesamte and JoAnn Trottier
Chocolate Cream Pie: Bev Fischer

Thanks to Jane and Ed Pincus, Chuck Trapp, Harv Pennington, Bruce Stuart, RB2—3, Jim and Ingrid Wilson, Tom Bentley, Mary Bocage, Michael Rufino, Matthew and Gabriel and Beth Ann Betit, Paul Wiener

Notebooks by Mid—America Plastics

Cataloging Information:
Kitsz, Dennis Bathory, 1949
Learning the 6809. (The Micro Language Lab). viii, 222 pp.
Roxbury, Vermont, Green Mountain Micro: 1983.

Preface

When IBM introduced its Personal Computer with grand gestures and flourishes, the reviewers and the public seemed overwhelmed, as if in the presence of royalty. The PC's 16-bit microprocessor was revered and its BASIC praised, while its operating flaws were forgiven. Everyone seemed to say, "Good show, IBM. Wish we'd thought of that!"

Tandy Corporation doesn't have that classy IBM image. When Radio Shack introduced its Color Computer, hardly anyone noticed. It looked for all the world like another toy, said the critics.

Maybe Radio Shack needs to work on its grand gestures and flourishes a little harder. That toylike Color Computer appeared more than a year before that IBM PC. So although the microcomputing press pointed to the PC as innovative for including line, circle, draw and paint commands, they had conveniently overlooked that these same BASIC commands were actually introduced a year earlier on the Color Computer. And while critics talked about 16-bit processing power in the IBM machine, they had conveniently overlooked that both the PC and the Color Computer contain powerful 16-bit "internal" -- but 8-bit "external" -- microprocessors.

As I said, it's an image problem. The Color Computer, at one-quarter or less the cost of IBM's pricey PC, is the computing bargain of the early 1980s. And the heart of the bargain lies in the heart of the computer: the 6809 processor.

The 6809 is the Maserati of the 6800 family. It's fast, sleek and powerful. Almost anything any processor can do, the 6809 can do better. Its software capability is almost unrivaled in the 8-bit world, and its hardware features are stable and easily applied. Combined with its cousins -- the 6883 address processor, the 6847 video processor, and the 6821 interface circuit -- the 6809 creates a simple yet versatile personal computer. The Color computer is actually a practical computer application suggested by Motorola, the 6809's manufacturer.

"Learning the 6809" was created to fill a knowledge gap. The 6800 family hasn't produced any real "pop" processors. The 6502 achieved its glamour in the Apple, the Z80 became known through its presence in so many different TRS-80 computers. The 6809 looks different. It works in powerful ways which are, unfortunately, alien to users of 6502, Z80 or IBM-PC-style 8088 computing.

Be prepared to work hard; this course isn't an information giveaway. If you want to find out how to copy Joe's Lumbergrunters game, forget it; the answer won't be here. But you will be able to answer the question yourself by applying the knowledge, tools and techniques I present. This isn't "Using the 6809 to Learn the Color Computer" -- it's "Learning the 6809", where the Color Computer is the practical example. When you finish this series of tapes, you'll have the tools to explore the programming limits of the Color Computer, you'll be prepared for programming other 6809-based machines, and you'll be ready for the programming concepts and principles of the 68000 family of full 16-bit processors.

Work hard. With concentrated listening, by working out each example and by answering every question, these 24 half-hour lessons should take you anywhere from 50 to 100 hours to complete. By then, you'll be speaking 6809. Work, enjoy, and good luck.

[illegible]

December 1983

Acknowledgments

It was midway through a long, bleak Vermont winter day spent with an incomprehensible microprocessor data book that I conceived of the Micro Language Lab. The data book made no sense to me. Engineers, I thought, don't speak English. No, I reconsidered, that's wrong. Engineers speak eloquently, but in an English far different from the rest of us. Just like musicians. And typographers. And artists. And priests.

A book was needed for 6809 users, and Color Computer owners in particular. I glanced at my library of programming books, looking desperately for ideas and inspiration. Nothing there. I couldn't think like Adam Osborne and I couldn't write like Bill Barden.

But talking was something fluid. Ideas that came to me easily when I was speaking would choke and gasp at my typing fingertips. Perhaps if I took microphone to hand, I could close my eyes and imagine a circle of anxious faces around me -- hanging on every word -- and the eloquence would begin...

The project got down to business at the same time Green Mountain Micro was established as my full-time occupation. I sat across from my old friend and business partner, RB2—3 (born with that name -- really!), and presented the idea. Sure, talk, great, he said, do it.

That was the easy part. The talking came quickly. But with me a musician and RB an artist, we found ourselves as babes in the business woods. We needed pretty notebooks, crates of cassettes, someone to print cassette labels and stick them on, a good and accurate typesetter, a nearby printer, recording and editing facilities, a duplicator, and a hundred sundries.

Everyone went to work. RB, our friends and new employees JoAnn and Steve, and my wife did the recording in my music studio. I edited the tapes onto the floor in a two-foot heap of gutteral stumbles and flubbering stutters. Graphics designers visiting from New York were ingloriously put to work on the layout. The typesetting was done very efficiently by computer connection to California, but on the trip back, the shiny (and expensive) new strips of typeset got lost -- twice! -- in the back rooms at Federal Express. People (specifically me) got sick, the printer went on Christmas vacation, and our New York visitors escaped in the dark of the night.

Meanwhile, advertisements placed three months ahead of time began to appear. Faithful customers had placed orders for the holidays. We worked round-the-clock, only to have the last few weeks tumble into an abyss of chaos and exhaustion. We blew our deadline. As I write this, the final pieces fit together. The result is Learning the 6809, what I consider my -- and Green Mountain Micro's -- finest work.

During the craziness of preparation, our combination home and office took on the look of a factory as dedicated people traipsed in and out, crossing paths at 3:30 a.m. in 25-below winter weather. Those deserving my sincerest thanks:
-- RB2-3, for going along with the Micro Language Lab idea and for leaving me alone and phone-free for a whole month.
-- Jim (the Doctor) Holliday, for completing three hundred illustrations in a record two weeks; and Lynda, for not holding those all-nighters against us.
-- Mary Bocage and Michael Rufino, who escaped in the night leaving it all under control; Marie Lapre Grabon for finding it under control.
-- Chuck Trapp, for controlling those typesetting codes for three straight weeks and through two lost shipments; Harv Pennington, for delivering on the promise; Bruce Stuart for remaining cool; and Paul Wiener for half-duplex.
-- Jim Wilson, Tom Bentley, and M. Dickey Drysdale, all of whose last-minute cooperation alleviated the typesetting-in-Vermont syndrome.
-- JoAnn Trottier and Steve Lusk, who realized too late the meaning of "going on salary".
-- and for things many and varied: Claire, Peter Clarke, Deb Marshall, Charlie Freiberg, Claire, N. Spike Maggio, Gerald and Susan D'Amico, Cornelius ("the burritos are in") Murray, Claire, Tom Hardy of Motorola, Greg Keilty, those first faithful 80 customers, and Claire.

Contents:

1.

Hello. I'm Dennis Kitsz, your guide through the subminiature world of assembly language programming for the 6809 microprocessor. As you move with me through these new software concepts, I believe you'll constantly have mixed emotions. You'll likely find it rewarding . . . frustrating . . . enlightening . . . tedious — as well as very fast and powerful.

You probably know Color BASIC or Extended Color BASIC. But please start off learning with a blank slate; clear BASIC from your mind. Except for a few early examples, BASIC won't help you to learn 6809 assembly language. And, if you haven't found out already, you'll be surprised to discover how slowly BASIC really does work for you. On the other hand, it *is* a language that spoils you, with many convenient features, error messages, and programming prompts. By contrast, assembly language will at first seem the height of tedious absurdity. "All that just to clear the screen?", you will ask.

Don't worry. The feeling is almost universal. I'll admit right here that the breakthrough in learning assembly language for me took almost a year. There was no one to guide me. And because I remember that sense of frustration, I want to guide you.

If you're a newcomer to 6809, but know other processors, be prepared for some major differences in concept and approach. These are different languages we'll be working with. So whether you're a seasoned programmer or discovering assembly language for the first time, don't rush through these tapes; work with each one. Try every program. I've organized each lesson carefully so I won't waste your time, but even so, every concept will be presented and reinforced; most demonstration programs are provided on tape to save you the typing. So turn off the TV or radio, send the kids to bed, unhook the telephone, and pack the spouse off to bowling or a movie. More than anything else, assembly language takes concentration, the elimination of distractions, and — occasionally — the ability to suspend time and reality. Let me say part of that

This is the programmed learning section of the Micro Language Lab. In this column you will find questions and answers about the accompanying text in the form of quick questions. Also, your regular exercises and self-tests appear in this column. To make best use of these questions, start at the top of the page, and use a card to reveal each question but to cover the answer. Try to answer the question, and immediately compare your answer to the answer in the book.

For full use of the Micro Language Lab, follow these steps for each lesson: First, listen to the cassette tapes and follow along. Second, read the text and attempt the accompanying questions as you go along. Third, start over and attempt the questions by themselves. Repeat the second and third steps until you can answer all the questions without reference to the text. Then you are ready for the next lesson.

It works like this:

* How many steps are involved in using the Micro Language Lab programmed learning?

Three steps are involved in the programmed learning.

Requirements

* What is the first of the three steps in the Micro Language Lab programmed learning?

The first step is to listen to the cassette tapes.

* What is the second of the three steps in the Micro Language Lab programmed learning?

Read the text and try the questions.

* What are the first two steps in the Micro Language Lab programmed learning?

1. Listen to the cassette tapes.
2. Read the text and try the question.

* What is the last of the three steps in the Micro Language Lab programmed learning?

The third step is to learn the answers to the questions without referring to the text.

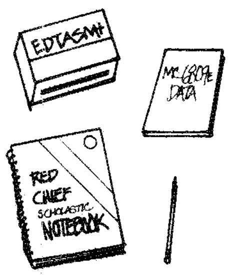
* What are the three steps in the Micro Language Lab programmed learning?

1. Listen to the cassette tapes.
2. Read the text and try the questions. 3. Learn the answers to all the questions.

So that's how it goes.

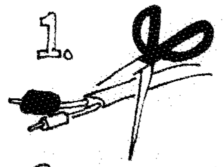
again. Assembly language takes concentration and the elimination of distractions.

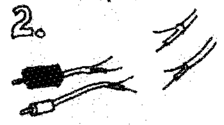
There are also some things you will need for this course. You can't get along without an Editor/Assembler, so please don't try. Get one. Radio Shack calls this program EDTASM+, and it's available in a ROMpack cartridge for all the Color Computers. It contains an Editor/Assembler system, which I'll help you learn to use, a rundown of the 6809 instructions, and other pertinent information. All the sample programs are compatible with EDTASM+.



You will also need a machine-language software monitor. That's part of the EDTASM+ cartridge, but if as you progress you feel you need more features, then there are several excellent commercial programs available.

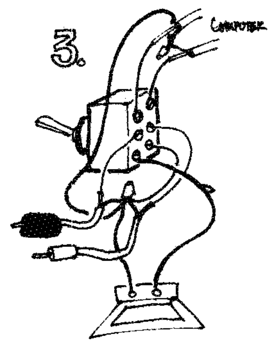
Blank cassettes are necessary only for saving original programs as you write them. You won't need blanks with this package to do any of the demonstration programs since everything is typed for you. But as you develop software, you may find that you like what you've done enough to keep it. For this you *will* need blank tapes.





Keep your Extended Color BASIC manual handy for reference, have paper and pencil ready, and take out the enclosed MC6809E data booklet and leave it nearby.

Finally, you will soon find that unplugging cables from your cassette player is no fun. Both my voice and all the programs are recorded together on these cassettes. Enclosed in this package are plans for a simple switch box so you can flip between listening to me and loading programs into your computer.

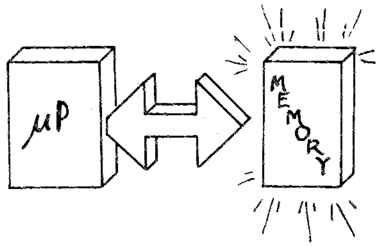


Support materials:

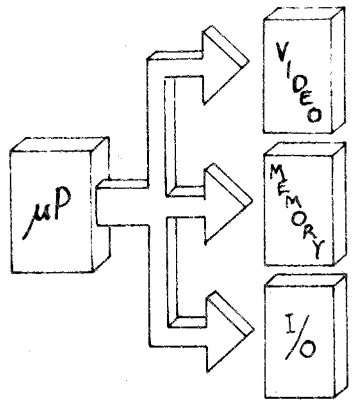
EDTASM+ and manual	RS Cat. No. 26-3250
Color Computer Technical Manual	RS Cat. No. 26-3193
Technical Manual Supplement	RS Parts No. 8749420
MC6809E data booklet (included)	Motorola DS9846-R1
MC6821 data booklet (included)	Motorola DS9435-R3
MC6847 data booklet (included)	Motorola DS9823
MC 6883 data booklet (included)	Motorola ADI-595R1

Now I want to tell you what you will be learning in this course. You will discover that assembly language is nothing like BASIC, but also that there are real advantages and disadvantages to using either one on the computer. You will learn binary and hexadecimal number systems, why they are needed at all, the ASCII codes, the job of the microprocessor, its architecture and timing, data flow, a little about how hardware relates to all of this, and lots of jargon. There will be lessons on memory maps, CPU control, input and output techniques, instruction sets, operation codes, instruction names, the inside and outside of the processor's world, and more jargon. Lots of demonstration programs will be provided, and in trying them you will learn how to use machine language monitors, editors, assemblers, and debugging techniques. Midway through the course, you will be learning all the different types of assembly language commands and their operation, how to use some subroutines already written for you in BASIC, the pitfalls of depending on that option, and more jargon. By the end of these tapes, you will be writing your own keyboard and screen subroutines, hopscotching data through memory, doing graphics and sound, and interfacing fast machine language with the simplicity of BASIC. And, of course, you'll be able to intimidate your friends with all the jargon you will use with such ease.

So now take some time to relax, clear your mind, and get set to begin learning 6809 assembly language programming. By the way, Claire is here to tell you exactly when to turn this tape on and off, when to load programs, and where to look in your booklet for your next instructions.



Let's get started. I've already said that the microprocessor's language is not BASIC. So what is it? Theoretically, that answer is simple. The microprocessor's language — the machine's language — is a set of binary signals which causes predictable electronic events to take place within a microprocessor and in relationship to its external memory, events which can be combined and expanded into control signals, mathematical calculations, video displays, and high-level languages like BASIC itself.



However, I'm not sure this definition is very a useful start. Let me try it from a different angle. Imagine your car is a computer. You unlock the door, open it, sit down, put on the seat belt, insert the key, start the ignition, release the brake, put the car in gear, let up on the clutch, step on the accelerator, turn the wheel, and off you go. That's BASIC.

Machine language takes you inside. You unlock the door by inserting a key whose ridges lift tumblers to specific heights, enabling a cylinder to turn inside a shell, releasing certain mechanical barriers. Open the door by pressing a button which engages some levers, slides and springs, allowing the door to be pulled out on hinges. The seat belt unrolls from a spring-loaded coil, perhaps turning off a small switch as it is pressed into a latch. Another key is for

* What is the first thing you will discover in this course?

That assembly language is nothing like BASIC.

* Name three other things you will learn in this course (there are several answers to this question)?

Number systems; architecture and timing; data flow ... or Memory maps; instruction sets; operation codes ... or Graphics; sound; jargon.

* Again, the first thing you will learn in this course is...

...that assembly language is nothing like BASIC.

* When you hear Claire's voice, she will tell you one of three things. What is the first one?

When to turn the tape on and off.

* Claire will tell you when to turn the taoe on and off. What is another thing she may tell you?

When to load programs.

* Claire will tell you when to turn the tape on and off and when to load programs. What else may she tell you?

Where to look in your book for your instructions.

* What is another name for the microprocessor's language?

Another name for the microprocessor's language is machine language.

* How is knowing BASIC like driving a car?

Because both are simple to use but cause complex operations inside a machine.

* What do you call the description of how the computer's designers have arranged its memory?

A memory map.

* How many characters of memory does the normal display screen use?

512 characters.

* At what memory location does the normal display screen begin on the Color Computer?

At memory location 1024.

* How many memory locations are there in the Color Computer?

There are 65,536 memory locations in the Color Computer.

* What is the arrangement of these memory locations called?

The memory map.

* Where does the normal display screen begin in the memory map?

At location 1024.

* Where does the normal display screen end in the memory map?

At location 1535.

* How many memory locations does the normal Color Computer display screen use?

The screen uses 512 locations.

* How many memory locations are there altogether in the Color Computer memory map?

There are 65,536 locations in the memory map.

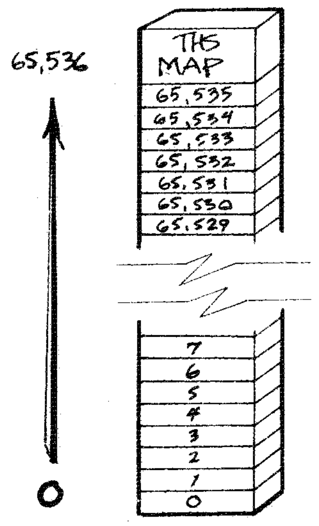
* What is the number of the first memory location?

It is number 0 (zero).

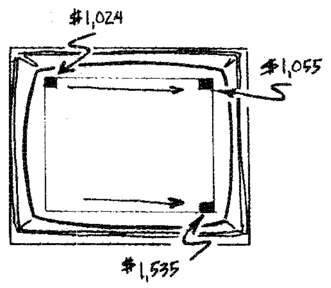
another set of tumblers which releases a clamp on the steering wheel and permits electrical current to flow through engine components. Turning the key further sends electricity to an electromagnet, pulling a starter motor into position, rotating the starter motor, spitting high voltage through rotors, wires and spark plugs in a very precise order, sucking gasoline and air into engine cavities, consequently igniting the gasoline and air mixture, pushing pistons which, through mechanical linkages, rotate the engine's crankshaft. The rotation also activates a generator which, combined with those explosions, causes a self-sustaining repetition. Electrical and monitoring circuits are activated. You release the key and prepare to put the car in gear.

By now you get the idea. Getting into a car and driving away is a simple task for a modern American. Yet the number of machine-level activities that take place in that short span is enormous. When you enter "PRINT 3 + 4" and BASIC responds "7", that simple action represents an equally astounding number of machine-level activities: checking the entire keyboard for your typing, displaying your typing in the correct screen position, interpreting your commands and checking them for correctness, calculating the results, displaying the results, and returning for your next input. That's a summary of the thousands of steps involved. Machine language is working for you at all times.

Where is the machine language? How do you get to it? And how does it work? Some folks tell me that the "dot on the screen" example is shopworn. Well, get ready. Here it is again. For me, an intellectual understanding of a concept is seldom as effective as seeing or hearing something concrete. Throughout this course, visual and sonic examples will be used frequently — so you know you've "done something". So, putting a dot on the screen is the place to start.



To put that dot on the screen, you have to know where the screen is. The "where" is what's known as the computer's memory map. This map is a description of how the computer's designers have arranged its memory. I'll talk a great deal about memory maps later in this course, but for the moment let me tell you that the normal Color Computer screen occupies a block of memory 512 characters long beginning at memory location 1024 and running through memory location 1535. That's where it lies in the overall map of 65,536 memory locations.



So when you ask BASIC to PRINT on the screen, evaluations are made to determine the exact screen location that is available, and the information is subsequently placed in screen memory for you to see and read. We can emulate this process. Turn your computer on, and when "OK" appears, type POKE 1024,110. (Repeat) Press ENTER. Your screen should show a black dot in the upper left hand corner — an ordinary period, actually. You could just as easily PRINT this from BASIC. But now try this. Type POKE 1024,46 (repeat), and press ENTER.

Now there's a black box with a white dot — a reverse-video period. There's nothing you can PRINT from BASIC to produce that, because it's one of BASIC's non-printable codes.

Simple as that seems, this example represents just one of the hundreds of capabilities that machine language offers. In fact, there are 32 characters BASIC doesn't let you see. Have a look in this next example.

Program #1, 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/0 error occurs, rewind to the start of the program and try again. For severe loading problems, see the Appendix.

```
10 CLS
20 PRINT"BASIC'S CHARACTER SET:"
30 FOR X = 0 TO 127
40 PRINT CHR$(X);
50 NEXT
60 PRINT:PRINT"THE WHOLE THING:"
70 FOR X = 0 TO 127
80 POKE 1216+X,X
90 NEXT
100 PRINT@448,"";
```

Run this program. You will see the 96 numbers, letters and symbols that BASIC can print. Below them you will see all 128 numbers, letters and symbols that your computer actually has available.

To summarize this program: BASIC prints its available characters, whereas the POKE statement manipulates memory to contain exactly what you wish.

The first advantage of machine language, then, will be to give you access to everything your computer has built into it, with no exceptions. Before I turn to another advantage, you should note now that the two sets of characters in the previous example are not displayed in the same order. I'll explain why later.

* Can you PRINT a reverse-video period on the screen using BASIC?

No, you can't PRINT a reverse-video period.

* What BASIC command do you use to display a reverse-video period?

POKE.

* What does POKE do?

POKE places a value directly in memory.

* How many characters can BASIC not display using PRINT?

32 characters cannot be displayed with PRINT.

* How many characters are available in the Color Computer?

128 characters are available.

* What command can display all 128 characters?

POKE.

* How does it display all 128 characters?

By directly manipulating display memory.

* What is the arrangement of memory locations called?

The memory map.

* Where does the normal Color Computer display screen start in this memory map?

At location 1024.

* What is the command for displaying value #111 at the first location in display memory?

POKE 1024,111

Program #2, 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 CLS
20 INPUT"CHARACTER";A$
30 PRINT"PRINTING..."
40 GOSUB 440
50 CLS : GOSUB 440 : TIMER = 0
60 FOR X = 1 TO 511
70 PRINT A$;
80 NEXT
90 A = TIMER : GOSUB 440
100 GOSUB 460
110 GOSUB 440 : CLS
120 PRINT"PRINTING STRINGS..."
130 GOSUB 440 : CLS : TIMER = 0
140 FOR X = 1 TO 15
150 PRINT STRING$(32,A$);
160 NEXT
170 PRINT STRING$(31,A$);
180 A = TIMER : GOSUB 440
190 GOSUB 460
200 GOSUB 440
210 CLS
220 PRINT"POKING CHARACTERS..."
230 A = ASC(A$)
240 GOSUB 440 : CLS : TIMER = 0
250 FOR X = 0 TO 511
260 POKE 1024+X,A
270 NEXT
280 A = TIMER : GOSUB 440
290 GOSUB460
300 GOSUB 440
310 CLS
320 PRINT"MACHINE LANGUAGE..."
330 DATA BD,B3,ED,8E,04,00,E7,80,8C,06,00,26,F9,39
340 FOR X = 16000 TO 16013
350 READ B$ : A = VAL ("&H"+B$)
360 POKE X,A
370 DEFUSR0=16000
380 NEXT : TIMER = 0
390 A = USR0(ASC(A$))
400 A = TIMER : GOSUB 440
410 GOSUB460
420 GOSUB 440
430 END
440 FOR N = 1 TO 500 : NEXT
450 RETURN
460 CLS :PRINT"TIMER READS"A
470 GOSUB440
480 RETURN
```

*** What is the purpose of program #2?**

To fill the screen with a display 512 identical characters.

*** What are the four ways this program fills the screen with characters?**

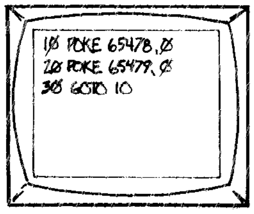
By PRINTing characters; by PRINTing strings; by POKEing values; by using machine language.

Welcome back. The program demonstrates the speed of 6809 assembly language. Its purpose is simply to fill the screen with 512 identical characters, which can be done in at least four ways: by printing 512 characters through BASIC, by printing strings of characters, by POKEing 512 characters from BASIC directly into screen memory, and by handing control over to a 6809 machine language program. RUN this program now.

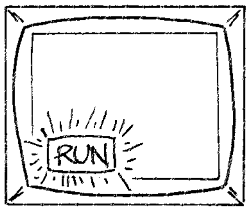
First, enter any uppercase letter from A to Z you wish displayed. Observe the BASIC printing technique. Notice the string printing method, which is quite fast. Now watch the BASIC POKEing technique. And finally, the machine language routine seems instantaneous.

Now there are three important things to notice. The first is the speed of the machine language program; don't miss that final display. Run the program again. This time, enter a number or punctuation mark as the character to be printed instead of a letter. Observe carefully as the printing and string printing finish that the LAST (512th) letter is missing. In BASIC, if you print in that 512th screen position, the screen automatically scrolls to the next line. But characters can be POKEd anywhere in memory, even in the last screen space. The machine language program is a fast way of doing that POKEing.

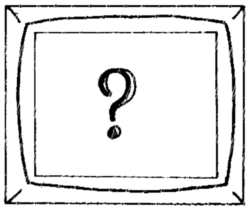
Yet there's something else. This time, the characters printed are not the same as those POKEd into memory or displayed by the machine language program. Recall the first program in this lesson — the characters weren't in the same order when printed and POKEd into memory. The reason is the hardware chosen to perform the video display. This hardware is limited to displaying only 64 characters — numbers, symbols, and uppercase letters. The Color Computer uses reverse (also called inverted) letters to represent lowercase. The BASIC software knows how to switch all these around to get the standard order — the order of ASCII, the American Standard Code for Information Interchange. This first, short machine language program doesn't do that. But it can be expanded. We'll return to that later.



The final lines of the BASIC program contain data statements and other commands which set up and execute a machine language program. Although you may examine these now, I'll hold back the detailed explanation of these for the moment.



So far, I've only played around with screen memory by putting some things on it. Now enter a three-line program; I'll read it to you. Line 10. POKE 65478,0. Line 20. POKE 65479,0. Line 30. GOTO 10. I'll repeat that; you can glance in the manual and check Program #3 to double-check.



```
10 POKE65478,0
20 POKE65479,0
30 GOTO10
```

RUN this program. What's that? It's delving into the heart of the computer, manipulating its control signals. It's video screen position information masquerading as computer memory. And that's the subject of the next lesson.

* What does ASCII mean?

American Standard Code for Information Interchange.

* How does the Color Computer represent lowercase letters?

Lowercase is represented by reverse video (white on black).

* Are the internal (hardware) Color Computer characters in ASCII order?

No.

* Does PRINT display the characters in ASCII order?

Yes.

* Does POKE display the characters in ASCII order?

No.

* Why does PRINT display the characters in ASCII order?

Because the BASIC software switches them.

* What BASIC command is used to show the internal order of the characters?

POKE.

* What does POKE do?

It places a value into memory.

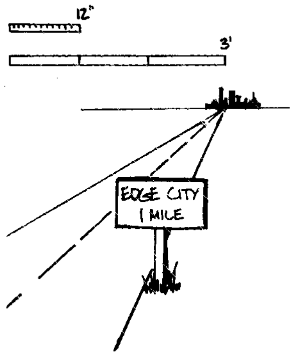
* What locations in the memory map does the normal Color Computer display screen use?

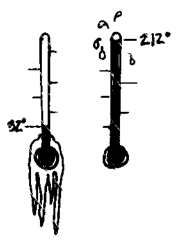
From locations 1024 to 1535.

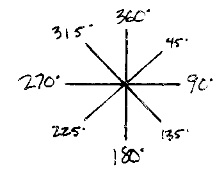
* Of the four methods in Program #2 — PRINTing characters, PRINTing strings, POKEing, and machine language — which is fastest?

Machine language is the fastest method.

2.







Welcome to the subject that strikes unreasoning terror in the heart of every programming novice — numbers and number systems. There are many opinions about computer number systems. Here's mine: if you don't learn them, you'll end up hacking your way through assembly language programming. You'll never feel comfortable or competent doing it.

That said, I'll start by telling you that I'm no mathematician. Numbers make me cringe. Yet binary and hexadecimal computer representation are really easy. Partly that's because I found that, when ordinary sheep failed me, counting backwards hexadecimal sheep jumping a fence put me to sleep before I reached zero. I'm not making it up.

Seriously, computer number systems have been made frightening by obscure use of language and knot-headed programmers. In truth, we live and live well in a world of non-decimal number systems. Here's a short list:

12 inches to a foot, 3 feet to a yard
5280 feet to a mile
32 degrees is freezing, 212 degrees is boiling
60 minutes in a hour, 24 hours in a day
7 days in a week, 52 weeks in a year
365 days in a year, but 360 degrees in a circle
3 teaspoons to a tablespoon, 4 quarts to a gallon
30 days hath September, April, June, and so forth

There are dozens, acres, ounces and hundreds of other examples. All are the daily measurements of our bread and butter, our life and time. All are irregular ways of numbering, but few confuse us. Chances are you can identify every one of these groups of numbers:

727, 737, and 747	98.6
33, 45 and 78	3.1416

You have finished the first lesson. The programmed learning section of that lesson was simple and repetitive; all of the programmed learning is somewhat repetitive, but as you go, the pace will begin to quicken. Also, the questions in this lesson will assume you know the material in the first lesson. Much of the groundwork in assembly language is rote learning, just like memorizing times tables, so keep up with the programmed learning questions.

* Different number systems are our heritage. How many cards in a poker deck? How many weeks in a year?

52.

* How many cards in a suit? How many weeks in a quarter?

4.

* How many spots in a card deck? How many days in a year?

365.

* How many face cards in a deck? How many months in a year?

12.

*** Is the decimal system used for computer operations?**

No.

*** Why aren't decimal numbers used for computer number systems?**

Because the activities the numbers represent would be clumsy or make little sense in decimal.

*** What number system is used for computers?**

The binary system.

*** How many numbers can be represented by the binary system?**

2 numbers.

*** What are the names of the two numbers represented by the binary system.**

The binary numbers are 0 and 1.

*** Name another pair of conditions that can be represented by the binary system.**

On and off.

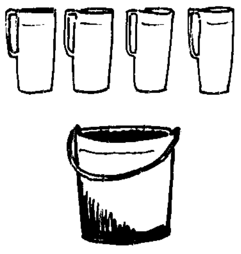
*** Name some other opposite pairs of conditions that might be represented by the binary system (there are many correct answers to this question).**

High and low; in and out; forward and backward; red and green; and so forth.

*** Was Dr. Watson a medical doctor?**

Yes. He was a general practitioner. This question in no way relates to the discussion of number systems.

What I guess came to mind were airplanes, records, normal body temperature, and pi. My point is that this is a conceptual issue. These are not numbers, they're representations of something useful in real life. And computer numbers are conceptual, too. The metric system is official in the U.S., but how much use does it get? Perhaps it too is a conceptual issue. I know how long a centimeter is, but can't convert from feet to meters. Same with a liter. Now that soft drinks come in liter bottles, I finally know what one is. Never could make a mathematical conversion of it, though. I can tell an acre, even though I don't know its actual measurement. In other words, once a number is represented by something in "real life", so to speak, I can make sense of it.

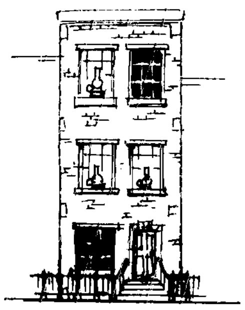


Basically that's what computer number systems are all about — they represent activities that are clumsy or make little sense when described by "regular" decimal numbers.

Keep that in mind. By now you're probably familiar with that old standard, the light-switch analogy. Computers, it's been said, operate using electronic switches that are either on or off, just like light switches. That's two conditions — the binary system, it's called. On or off.

Such a description is true as far as it goes, but it leaves out a lot. To cast a different light on the binary system, I've enlisted the help of two old friends, Sherlock Holmes and Doctor Watson, who will discover some clues in this slightly rewritten scene . . .

Watson: . . . but it's just someone turning the lamps on, Holmes. It's past dusk, after all.



Holmes: Ah, yes, Watson, but why would someone light a lamp and extinguish it so quickly? And move from room to room? Eh, Watson?

Watson: Perhaps they're looking for something they can't find.

Holmes: Or perhaps they're signaling someone. A cipher of some kind, I would say.

Watson: With lamps in five windows? Nonsense, Holmes!

Holmes: Just copy this down, Watson. I'll read starting from the uppermost window. Lamp on, lamp off, lamp on, lamp on. . .

Watson: Slow down, Holmes.

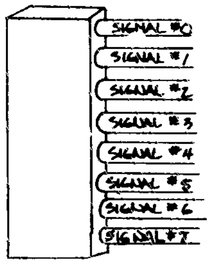
Holmes: Keep at it, Watson, they won't stop for your fingers. Lamp off. They're changing now. Again, from the uppermost. Off, on, off, off, on... I've got it, Watson! These are letters of the alphabet. Five windows create 32 combinations, enough for all the letters of the alphabet.

Watson: Amazing, Holmes!

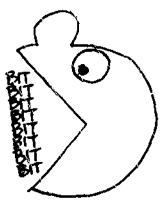
Holmes: You don't need to write down the lamps now, Watson. I think I can read the message. S – E – E M – E A – T . . . See me at the August Lion Tavern at 6 o'clock. That's it! We have just five minutes. Come on, Watson!. . . .

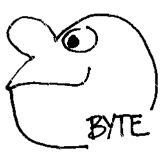
What Holmes discovered, of course, was that by using the most basic information — a simple pattern of lamps lit or extinguished — a complete message might be sent and received. Morse formalized that with his telegraph code. In this case, Holmes perceived quickly and correctly that with five lamps, 32 combinations were possible by rearranging the pattern of lamps lit and darkened.

You will find that computers are really quite simple-minded devices. You're dealing with nothing more than a vast but microscopic nest of electronic switches. There's no intelligence involved — just an impulse here, an impulse there, all moving very fast. For reasons that have more to do with manufacturing economy than anything else, the decision to use the on-off switch was chosen over something more familiar like a decimal type counter. Programming might have been much easier otherwise. But, cheap as it was to manufacture, the on-off idea limited each meaningful computer signal to those two conditions alone. For more conditions — larger numbers, that is — more on-off signals are needed. Groups of signals, all working simultaneously, like Holmes's five lamps.



Everything in computers began to take on the color of two choices, base 2, the binary system. Data was parceled out in base 2, and grouped in powers of 2. The first microprocessor device used four simultaneous signals for transmission of data. The 6809 uses eight signal lines. Newer, more sophisticated computers use 16 or more concurrent on-off signals.





You can probably guess I'm taking you easy into this. But stay with me. If you think back to Holmes's discovery, you'll remember that the operant concept was not the number, but rather the pattern of lamps. The patterns represented codes for letters — an inspired idea from the time Morse developed his telegraph code to the present day American Standard Code for Information Interchange (ASCII).

In computers, these are patterns of binary signals, thought of as binary numbers or binary digits. Binary digit is conveniently abbreviated "bit". So when the 6809 is called an 8-bit processor, that means that all its information is created from the combinations of eight binary digits. Here's the grabber: no matter what the information

* How many combinations did Holmes figure could be made from five lamps?

32 combinations.

* How many lamps would produce only 16 different patterns?

Four lamps.

* How many different combinations would Holmes have discovered if there were six lamps?

64 combinations.

* How many different combinations would Holmes have calculated from eight lamps?

256 combinations.

* Write down the powers of 2 from 2↑1 to 2↑8.

2, 4, 8, 16, 32, 64, 128, 256

* What number system is used in computers?

The binary system.

* How many different numbers can be represented by the binary system, and what are they?

2 numbers; they are 0 and 1.

* How many different combinations can be formed from eight on-or-off lamps? From eight one-or-zero binary digits?

Both answers are the same: 256 combinations.

* What does bit mean?

Bit means binary digit.

* How many binary digits (bits) are used by the 6809 processor to represent information?

Eight bits.

represents — letters, numbers, commands, display, sound, whatever — it is formed by some pattern of those eight binary digits, formed from those eight bits. The microprocessor, the computer's heart, can know the difference only by the context in which those digits are presented.

If that seems far-fetched, consider that there are only 26 letters in the alphabet, 10 numerical symbols, and a dozen or so punctuation marks. Those letters, in specific combinations and contexts, make up the half-million or so words in the English language. Those same letters, combined into words and melded through punctuation into sentences and paragraphs, can describe the entire known history of humanity with multiple levels of nuance, politics, or poetry. Quite a bit from a simple 26-letter code.

At last it's time to get down to specifics, and deal with those numerical symbols. The trick is for you to gain an appreciation of the computer number system that's used exclusively for clarity. It's called hexadecimal. Base 16. Don't run for the Maalox. Keep in mind that we're not talking about counting-type numbers here, but simply representations, symbolic abstractions.

First, there's a program to get up and running.

Program #4, 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/0 error occurs, rewind to the start of the program and try again. For severe loading problems, see the Appendix.

```
10 CLS:L=164
20 PRINT"CONVERSION:":PRINT"FROM DECIMAL TO BINARY"
30 FORB=1TO5
40 PRINT:INPUT"NUMBER Ø TO 255":X
50 IFX<ØORX>255THENPRINT"OUT OF RANGE":GOSUB290:GOTO40
60 GOSUB320
70 PRINTC;E;G;I;K;M;O;Q;
80 NEXT
90 PRINT:INPUT"<ENTER> TO CONTINUE";A$
100 CLS
110 PRINT:PRINT"DECIMAL AND BINARY DO NOT BEAR  A DISTINCT VISUA
L RELATIONSHIP:"
120 FORX=ØTO255
130 GOSUB320
140 PRINT@256.X;:PRINT@266,"<----DECIMAL"
150 PRINTC;E;G;I;K;M;O;Q;
160 NEXT
170 CLS
180 PRINT:PRINT"HEXADECIMAL AND BINARY SHOW A   CLEAR VISUAL REL
ATIONSHIP.  THE FOUR BINARY DIGITS CREATE 16    PATTERNS.  EACH
BINARY PATTERN  IS IDENTIFIED BY A UNIQUE       HEXADECIMAL SYMB
OL FROM Ø TO F."
190 FORX=ØTO255
200 X$=HEX$(X)
210 IFLEN(X$)=1THENX$="Ø"+X$
220 PRINT@260,"...."LEFT$(X$,1)".....   ...."RIGHT$(X$,1)"....."
230 GOSUB320
240 PRINT"    "C;E;G;I;K;M;O;Q
250 PRINT"     ..........  .........."
```

* How many different combinations can be formed from eight bits?

256 combinations.

* How many different combinations of binary digits can the 6809 processor produce from its 8 bits?

256 combinations.

* How does the 6809 processor distinguish letters, commands, display, sound and other purposes of the 256 combinations of 8 binary digits?

By the context in which those digits are presented.

* What number system is used in computers?

The binary number system is used in computers.

* What counting system is used for clarity in discussing computer numbers?

The hexadecimal counting system is used for clarity.

* How many numbers are represented by the hexadecimal counting system?

16 numbers are represented by the hexadecimal counting system.

```
260 NEXT
270 FORX=1TO300:NEXT
280 STOP
290 FORN=1TO1000:NEXT:RETURN
300 X=C*128+E*64+G*32+I*16+K*8+M*4+O*2+Q
310 RETURN
320 C=INT(X/128):D=C*128
330 E=INT((X-D)/64):F=E*64
340 G=INT((X-D-F)/32):H=G*32
350 I=INT((X-D-F-H)/16):J=I*16
360 K=INT((X-D-F-H-J)/8):L=K*8
370 M=INT(((X-D-F-H-J-L)/4):N=M*4
380 O=INT(((X-D-F-H-J-L-N)/2):P=O*2
390 Q=INT(X-D-F-H-J-L-N-P)
400 RETURN
```

I've been talking about symbols, relationships and legibility. I'm also talking about memorizing patterns for instant recognition. You're about to run a program which will show all 256 rearrangements of eight binary digits, represented as a string of ones and zeros. Run this program now. Enter a number from 0 to 255, and check out the binary equivalent printed below. Enter another number, and look. Enter three more numbers, and examine the binary equivalents. Chances are, what you see is not very useful. Hit <ENTER>. The decimal values from 0 to 255 will be displayed in order, together with their binary equivalents.

The decimal numbers count up nicely from 0 to 255, and the binary numbers also follow a regular pattern. That binary counting-up pattern probably isn't familiar to you yet, but there are lots of ways of understanding it. For example, as you watch, notice that the right-hand digit alternates quickly between 1 and 0. Its neighbor's alternation takes twice as long, and its neighbor's alternation in turn takes twice as long as that. There's that binary, base 2 system working again. The binary counting is useful to watch; try to get familiar with it.

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

When the decimal counting is finished, the program will show you a much easier system. Mentally break that eight-bit binary group into two halves. Remembering Sherlock Holmes' discovery, you can see that the four binary digits in each group can be rearranged 16 different ways. Instead of trying to recall lines of ones and zeros, though, each arrangement can be identified with a single, unique character. The arrangement **0000** can be identified as 0. **1** can be identified as 1. **0010** becomes 2, **0011** becomes 3, **0100** becomes 4, on up to **1001**, which is called 9. **1010** is labeled A, **1011** is labeled B, **1100** is labeled C, **1101** is D, **1110** is E, and **1111** is F.

As you watch the screen, you will notice that a separate symbol — a hexadecimal symbol — is used for each half of the 8-bit group. That gives you an easy-to-handle two-digit reference for each long binary number from **00000000** to **11111111**.

The advantage of this method is very real. By knowing a binary number, you can almost instantaneously know the hexadecimal equivalent. By knowing the hexadecimal

*** If you are working with eight binary digits, what is the binary equivalent of decimal number 1?**

00000001 is the 8-bit binary equivalent of the decimal number 1.

*** If you are working with eight binary digits, what is the binary equivalent of decimal number 255?**

11111111 is the 8-bit equivalent of decimal number 255.

*** What is the hexadecimal symbol for decimal number 1?**

1 is the hexadecimal symbol for decimal number 1.

*** What decimal number is represented by binary number 00001111?**

00001111 is the decimal number 15.

*** What hexadecimal number represents binary number 0000?**

Hexadecimal number 0 represents binary 0000.

*** What hexadecimal number represents binary number 1111?**

Hexadecimal number F represents binary 1111.

*** What hexadecimal number represents binary number 00001111?**

Hexadecimal number 0F represents binary 00001111.

*** Count the binary numbers from 0000 to 1111.**

0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111

Reading Hex

* Count the hexadecimal numbers from 0 to F.

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

* Count backwards in hexadecimal numbers from F to 0.

F, E, D, C, B, A, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0

* What binary number is represented by the hexadecimal number C?

Hexadecimal C is binary 1100.

* What is the shorthand word for hexadecimal?

The shorthand word for hexadecimal is hex.

* Count aloud quickly from hex 20 to hex 30.

Two-zero, two-one, two-two, two-three, two-four, two-five, two-six, two-seven, two-eight, two-nine, two-A, two-B, two-C, two-D, two-E, two-F, three-zero.

* What symbol is used to indicate a hex number?

The dollar sign.

* What is the hexadecimal number for binary 1100?

The hexadecimal number is $C.

* Count aloud quickly, backwards in hex from $FF to $E8.

FF, FE, FD, FC, FB, FA, F9, F8, F7, F6, F5, F4, F3, F2, F1, F0, EF, EE, ED, EC, EB, EA, E9, E8.

* What is the binary number for hexadecimal $AA?

Hexadecimal A is binary 1010, so hex $AA must be 10101010.

representation, you can get at the binary equivalent at any time. Remember, these microprocessors work in a binary world. Knowing that world is essential for you as the programmer to call the shots.

How can you learn the hexadecimal numbers? Memorize them. Just like the times tables in elementary school. Count sheep, forwards and backwards. Go back to this program and RUN170. Read the numbers aloud. By the way, hexadecimal numbers — you'll just call them hex after a while — are read a little different from decimal numbers. If there's a leading zero, for example, you hang onto it. Like this:

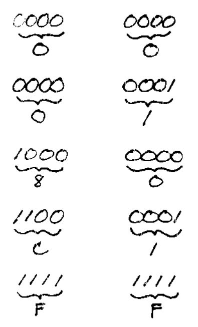
00 01 (not just "one") **02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10** (not "ten") **11** (not "eleven") **12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20** (not "twenty") and so on. Remember than when you get up to **9A 9B 9C 9D 9E 9F**, the next hex number is **A0**. It goes all the way up to **FF**.

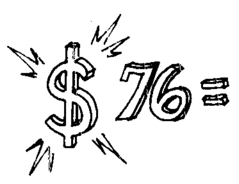
Another convention for hexadecimal numbers is their written form. The letters are always uppercase, and in order to distinguish hex from decimal, it's common practice to put a dollar sign in front of a hexadecimal number.

You should also try to learn your hex numbers backwards. Assembly language has certain kinds of program activities that move backwards, and being able to make an accurate backward count — **FF FE FD FC FB FA F9 F8** etc. — will ease this process.

As far as converting from decimal to hex and vice versa, you'll do it occasionally. Use a chart, a special calculator like the Texas Instruments programmer, or a formula. When you learn to use the editor/assembler/debugger programs, much of this conversion is done by the assembler itself. For the moment, learn to recognize the four-bit binary patterns and their hex equivalents. In fact, you might take a break from this tape right now to practice binary and hexadecimal patterns.

Hexadecimal numbers will be used for the remainder of this series. Please practice the hexadecimal numbers patterns and return to the tape when you can recognize the four-bit binary patterns and their hexadecimal equivalents.



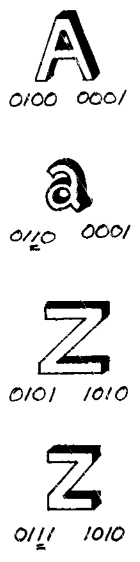


HEX 76 =
0111 0110

100
FF
FE
FD
FC
FB
FA
F9
F8
F7
F6
F5

Since this is a lesson about numbers and codes, I'd like to introduce another essential preliminary to diving into assembly language programming, the ASCII codes. ASCII — the American Standard Code for Information Interchange — is a set of 128 numerical codes to represent letters, numbers, symbols, punctuation, and special control functions.

I'll talk hex. Punctuation marks start at **$20**, numbers at **$30**. **$40** points to uppercase letters, **$60** starts lowercase. Simple? Only in hex. Ever try to convert from uppercase to lowercase in BASIC? It can be tricky. But in binary, it's a cinch. Grab paper and pencil.



Write down hexadecimal **41**, and across from it write its binary equivalent, **0100 0001**. This is the uppercase letter A. On the next line, write down hex **61**, and across from it the binary, **0110 0001**. This is lowercase a. Now write hex **5A**, and its binary, **0101 1010**; this is uppercase Z. Lowercase z is **7A**, binary **0111 1010**.

Sit back and look at these numbers. The hex numbers seem related enough, but the real clue lies in the binary. In referring to binary numbers, the rightmost digit is called bit zero. Find bit five in both upper and lowercase A; it's third from the left. Notice that bit five is the only digit that's different in upper and lower case. Same with letter Z. Bit five clearly distinguishes uppercase from lowercase. In decimal, upper and lowercase Z are 90 and 122 respectively. There's no visible relationship there. But bit five! Just one digit makes all the difference. ASCII looks illogical in decimal, not binary.

I'll talk more about ASCII codes, especially those from **$00** to **$1F** — the control codes that ring bells, backspace, line and form feed, carriage return, and perform special activities like clearing the screen. In the meantime, there's work for you to do.

For your assignment: learn to count in hexadecimal, explore all the ASCII codes in binary, and learn to read the ASCII bit table in the back of your documentation package. Review this lesson until you are familiar and comfortable with binary and hexadecimal. Please continue with these lessons only when you have reviewed the number systems thoroughly.

* What does ASCII stand for?

ASCII stands for American Standard Code for Information Interchange.

* Where are uppercase letters found in the ASCII code? Give the answer in hexadecimal.

The uppercase ASCII codes are $40 to $5F.

* Where are the lowercase letters found in the ASCII code? Give the answer in hexadecimal.

The lowercase ASCII codes are $60 to $7F.

* How are the rightmost and leftmost bits numbered in a group of eight binary digits?

The rightmost is bit 0, the leftmost is bit 7.

* What binary digit distinguishes uppercase from lowercase characters in the ASCII code?

Bit 5 distinguishes uppercase ASCII from lowercase ASCII.

* How is ASCII pronounced?

ASCII is pronounced ASSkey.

* The ASCII code for the uppercase letter E is hexadecimal $45. What are the binary and hexadecimal values for both uppercase and lowercase letter E?

Uppercase E is $45, binary 0100 0101. Lowercase E is $65, binary 0110 0101.

* What ASCII codes are located from hexadecimal $00 to $1F?

The machine control codes are found from $00 to $1F.

3.

Hello again. In this third lesson, we reach a critical point... the point of explaining the whys and wherefores of the 6809 microprocessor. You should have spent some serious time getting familiar and comfortable with binary and hexadecimal counting, as well as with the arrangement of ASCII characters. The workbook provides some exercises and self-tests; please complete them before continuing with this lesson. Especially if you're a first-timer to assembly language, that's very important.

There are many general ways in which microprocessors are described and defined: they are smart circuits, they are calculating devices, they are (as I've said) a microscopic nest of electronic switches. Microprocessors are all of these things and more. I'll use several terms interchangeably throughout these lessons — processor, microprocessor (or MPU), central processing unit (or CPU). In your Color Computer, these all mean the same thing: the 6809 microprocessor.

Inside all microprocessors, inside all MPUs, are a number of data holding stations called registers. More about the term register later, but at the heart of a microprocessor is a special calculator register, formally called an arithmetic logic unit, or ALU. The ALU holds one binary word — that is, a certain number of binary digits of information. I'm talking here about the 6809 processor. It accepts data in an eight-bit binary group, called by the tongue-in-cheek name "byte". The word size of the 6809's binary data is the byte — eight bits.

To describe it another way, the 6809 has eight wires connected to it for data. All eight wires become "live" simultaneously, conducting eight binary digits to the processor. This information is one byte.

So it's got this arithmetic logic unit, the ALU, which holds a byte of data. The ALU can then perform simple calculations with that byte of data. The calculations, which I'll get back to in detail shortly, are: addition, subtraction, and multiplication. Also, there are incrementing and

Things pick up speed now. There's lots of new information coming up, so make sure you've done all the exercises before ending this session.

* What does CPU stand for?

Central Processing Unit, the microprocessor.

* What is the Central Processing Unit (CPU) in the Color Computer?

The 6809 is the Color Computer's CPU.

* What holds the data inside the microprocessor?

Registers hold data inside the microprocessor.

* What does ALU mean and what does it do?

ALU means Arithmetic Logic Unit. The ALU performs calculations.

* How many bits of information does the ALU hold?

The ALU holds eight bits.

* What is the computer jargon for eight binary digits?

A byte is computer jargon for eight binary digits.

decrementing, which are essentially addition or subtraction by one. The ALU can perform logical operations such as AND, OR, NOT, and EXclusive OR. Finally, it can make a comparison with any other byte of data.

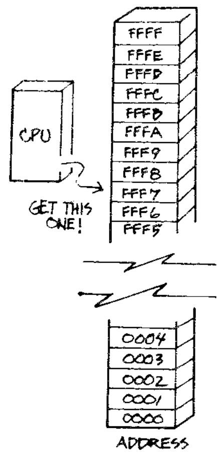
Most of the processor's hard work is done in the ALU. In fact, the 6809 is such an advanced processor that it contains two separate ALUs. Each one can add, subtract, increment, decrement, compare and perform logical operations. Together, they can be used to multiply.

The arithmetic logic units in most MPUs, in most processors, including the 6809, have several descriptive names. They are called accumulators. The ALU is also called an accumulator register. Finally, the 6809's own PAIR of accumulators are labeled A and B. So the arithmetic logic unit, the ALU, the accumulator, and the accumulator register effectively mean the same thing. In the 6809, they mean where the math is done — in A or B.

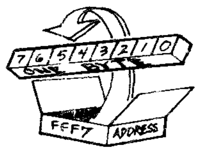
These A and B accumulators get the information they need by loading it. Loading: that's the term for obtaining data. The accumulators save the results by storing data. Load and store. Get data, save data.

I have to answer several questions at once now, because the actions that they represent are so intertwined. Here are the questions: How does an accumulator load or store data? Where does it load data from, and where does it store it?

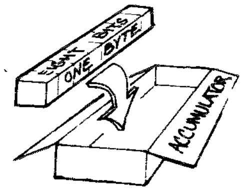
I'll start with the "where". The data the accumulator needs might be inside the microprocessor in another register. The term register is in fact quite general. The A and B registers of the 6809 are the arithmetic logic units, the accumulators. But there are other registers also capable of holding information, though these registers cannot by themselves do any mathematical calculations with the data. Their main purpose is to keep information handy for the accumulators.



But the most important place the accumulator obtains its information is from memory. Memory is a line of storage locations outside the 6809 processor itself. Each memory location can hold an eight-bit word, a byte.



I'll back off from that briefly to tell you *how* an accumulator loads or stores data. It follows the commands of an instruction decoder, that part of the processor which determines the actions the processor is to take. Now here's where my answers get intertwined. The instruction decoder gets its instructions from the same memory that stores data. In other words, when the instruction decoder gets a byte from memory that says "load something into the accumulator", the next byte in memory is that very "something". It all comes from the *same* line of memory.



Recall the last lesson. I said that the 256 possible

* What is a binary word?

A certain number of binary digits.

* What is the word size of the 6809 microprocessor?

The 6809's word size is eight bits.

* Again, what is the computer jargon for eight bits?

Eight bits make up a byte.

* Name three kinds of arithmetic the ALU performs.

Addition, subtraction, and multiplication.

* Name four kinds of logical functions the ALU performs.

AND, OR, NOT and Exclusive OR.

* Name the three other operations the ALU performs.

Incrementing, decrementing, and comparison.

* How many bytes can the ALU hold for doing its work?

The ALU holds one byte.

* Name all ten kinds of operations the ALU can perform on one byte.

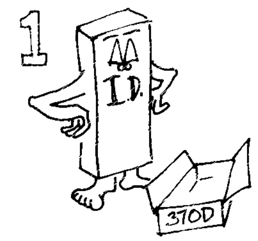
Addition, subtraction, multiplication, AND, OR, NOT, Exclusive OR, incrementing, decrementing, and comparison.

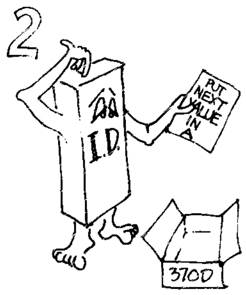
* What is the ALU called in the 6809 processor?

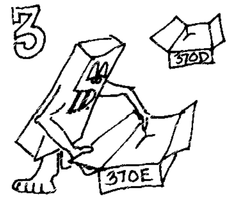
The ALU is called the accumulator.

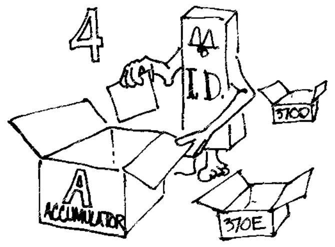
* How many accumulators are there in the 6809 processor?

The 6809 has two accumulators.









rearrangements of binary digits represent all the information the processor will ever need — instructions, numbers, ASCII characters, whatever. That's precisely true. I also said that it's the context that determines what the binary pattern means. Context is what assembly language programming is all about: ordering the bytes sos that they turn into a useful program.

So far you know that the processor, the MPU, gets both its instructions and data from memory. How does it distinguish them? That is, how does it understand their context?

To discover the answer, you must know that the memory locations are each uniquely numbered, starting from zero. These identification numbers are called addresses. How many memory addresses a given MPU has available are determined by the number of its address bits. In keeping with its total logical binary nature, the 6809 has 16 address bits. The total number of rearrangements of 16 binary digits, from **0000 0000 0000 0000** to **1111 1111 1111 1111**, is 65,536. It's what you call 64K (since a "K" in computer terms is 1,024).

Get a pencil and paper. Breaking them into groups of four digits, write down **0000 0000 0000 0000**. That's 16 zeros. Now, elsewhere on the paper, write down 16 ones. Also break those into groups of four: **1111 1111 1111 1111**. Above each group of four binary digits, write its equivalent hexadecimal symbol. For the 16 zeros, the hexadecimal value would be: dollar sign **0 0 0 0**. Don't forget that dollar sign; it identifies a hex number. Also write the hex value for 16 ones: dollar sign **F F F F**.

What you have just written is the address range — that is, the number of individual memory locations — available to the 6809 processor. **$0000** running through **$FFFF** are the addresses of the 6809 MPU.

---

* What are the names of the two accumulators in the 6809?

The two accumulators are called A and B.

* What are the ten kinds of operations the A and B accumulators can perform?

Addition, subtraction, multiplication, AND, OR, NOT, Exclusive OR, incrementing, decrementing, and comparison.

* What is the term for obtaining data?

Loading means obtaining data.

* What is the term for saving data?

Storing means saving data.

* What is the word size the 6809's accumulators can hold?

One byte, that is, eight bits.

* Aside from other registers, where do the A and B accumulators get their information?

The A and B accumulators get their information from memory.

* What is the word size of a memory location?

One byte, that is, eight bits.

* How many locations are in the Color Computer memory map?

65,536 memory locations.

* Describe the map size and word size of the 6809 processor's memory.

65,536 locations; each location is one byte in size.

* There are how many bytes in one "K"?

1024 bytes.

* There are how many "K" in 65,536 bytes?

64K.

* What is the number of the first and the last memory location in the Color Computer.

The first memory location is number 0; the last memory location is number 65,535.

* How many binary digits are needed to represent the range 0 to 65,535?

There are 65,536 possible combinations of 16 bits needed to represent the range 0 to 65,535.

* Write the number 0 in 16 binary digits.

The number zero in binary is 0000 0000 0000 0000.

* What is 0000 0000 0000 0000 in hexadecimal?

0000 0000 0000 0000 in hexadecimal is $0000.

* What is the number 65,535 in binary digits? Hint: it is the largest number that can be written using 16 bits.

65,535 in binary is 1111 1111 1111 1111.

* What is 1111 1111 1111 1111 in hexadecimal?

1111 1111 1111 1111 in hexadecimal is $FFFF.

* The 6809 microprocessor has a 64K memory map. How many bytes is 64K?

64K is 65,536 bytes (64K times 1,024 bytes per K)

You have just identified the 6809 processor's address range. Knowing now that the 6809's addresses run from **$0000** to **$FFFF**, you are ready to discover how the 6809 MPU distinguishes instructions it performs from the data it uses.

The 6809 goes through a fixed set of electronic actions whenever the power is turned on, or whenever the reset switch is pressed. The processor first does up its internal housekeeping. It requests the contents of memory at address **$FFFE**, and following that, it requests the contents of memory at address **$FFFF**.

Follow carefully here. The two bytes loaded from memory locations **$FFFE** and **$FFFF** are concatenated — that is, combined end-to-end. A byte is 8 bits; two bytes end-to-end are 16 bits. 16 bits happens to be the same size as the 6809 processor's address... something that didn't happen by chance. In fact, those two bytes are used as the address of the memory location containing the very first instruction the microprocessor will follow.

I'll repeat that. When the power is turned on, the 6809 fetches the two bytes stored at memory locations **$FFFE** and **$FFFF**. The processor concatenates them, producing a 16-bit value. That value is used as an address, and at that address is found the first instruction 6809 must execute.

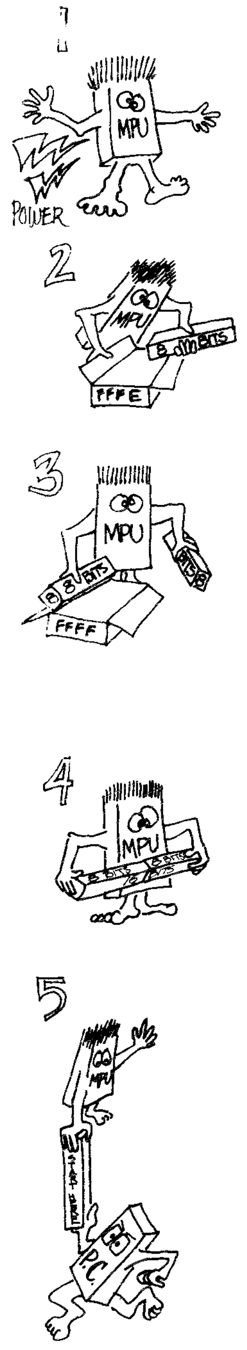
That address is put in a special 16-bit register called the program counter. From that point on, until the power is turned off, the program counter, called the PC, always keeps track of the next instruction the processor is going to follow. If the programmer has done a good job, the computer will begin executing the thousands of instructions that make up its language or operating system.

I think it's time for a summary.

I'm using microprocessor, MPU, processor, central processing unit, and CPU interchangeably. Inside the 6809 MPU are two arithmetic logic units, the ALUs, which each hold a single 8-bit word of data and perform simple calculations on that byte of data. The ALUs, also called accumulators, are identified as the A and B registers.

The registers load bytes of data from memory and store bytes of data in memory. There are 65,536 memory locations available to the 6809 MPU, and from them it gets both its instructions and data. The instruction decoder inside the MPU tells it what operations to perform in response to an instruction byte loaded from memory. The program counter register, the PC, keeps track of which instructions are next in line.

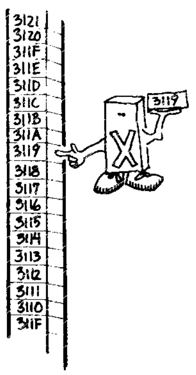
If you feel comfortable with this information, please continue with this tape. If any of it's shaky, start this lesson again; you might want to follow along in the text while reviewing the lesson.



What I'm discussing in this lesson is the 6809's architecture. That's the term for the logical organization of the processor. The water's about to get deeper, and I'm going to throw you in, so get ready to swim. Along with your documentation, there is a Motorola data booklet for the MC6809E processor. Find the booklet, and turn to page 5. Data booklets like these are meant for programming and hardware professionals, so much of it will initially appear incomprehensible. That fogginess is a trademark of data sheets.

We'll be concerned with the last two paragraphs on page 4, the first few on page 5, and most importantly Figure 5. Take a moment to locate those.

Look first at Figure 5. So far, you've found out about the program counter (PC) and the A and B accumulators. As you can see, there are actually several more registers.



The X and Y registers are effectively identical. They are called "index registers" because they act sort of like your index finger in a card file, pointing to a specific entry. Remember that registers are special data storage locations inside the processor. Each of these two index registers is 16 bits in size. Because X and Y are 16 bits, they can be used to point to a specific memory location, that is, to be indexed to that 16-bit address. Indexing is its most common function, but not its only use; here's an example of indexing.

Let's say there's a message to be displayed on the screen. I'll point my Y register to the video screen memory, and point my X register to the memory that contains the message. My program can then tell the A accumulator register to get the byte of data from the memory location indexed by X and put it in the memory location indexed by Y. Load A accumulator from memory indexed by X, store A accumulator to memory indexed by Y. The first letter of the message is then displayed. It's like telephone directory assistance. The operator indexes the number, the telephone transmits it, and you index it on your phone pad. If I increment both X and Y after displaying the first letter of the message — that is, if I add one to the present values of X and Y — they will be pointing to the next locations in memory. I can have the program repeat the process of loading from memory indexed by X and storing to memory indexed by Y. That would get the next letter of the message and display it in the next position on the screen. Load A from X-indexed memory, store A to Y-indexed memory.

I've got a program to do that.

* The 6809 microprocessor has a 64K memory map. What is its address range in decimal, in binary, and in hexadecimal.

The map runs from 0 to 65,535 in decimal, from 0000 0000 0000 0000 to 1111 1111 1111 1111 in binary, and from $0000 to $FFFF in hexadecimal.

* A byte of information is eight bits; an address is 16 bits. How many bytes are needed to describe an address.

Two bytes describe an address.

* What part of the processor determines what it must do?

The instruction decoder.

* Where does the instruction decoder get its instructions?

From memory.

* What memory locations does the processor use when the power is turned on?

It uses $FFFE and $FFFF when the power is turned on.

* What does the processor get from memory location $FFFE?

One byte.

* What does the processor get from memory location $FFFF?

One byte.

* The processor puts the bytes from memory locations $FFFE and $FFFF together. What is the process called, and what is the result in this case?

The process is called concatenation, and the result is a two-byte number.

* How many bits is two bytes?

16 bits.

Displaying a message

* How does the processor use the 16-bit number obtained by concatenating the contents of memory locations $FFFE and $FFFF?

It uses the 16-bit number as an address.

* What is the 16-bit number the address of?

The 16-bit number is the address of the first instruction the processor will execute.

* What part of the processor uses this instruction?

The instruction decoder.

* What keeps track of the instructions in the 6809 processor?

The program counter.

* What is the program counter and what is its shorthand name?

The program counter is a 16-bit register that contains the address of a computer instruction. Its shorthand name is PC.

* Name the index registers.

X and Y.

* What is the size of the X and Y index registers?

X and Y are each 16 bits in size.

* What is the most common function of the X and Y index registers?

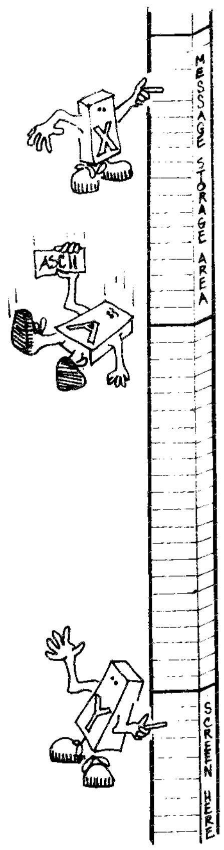
To index an address; that is, to hold the number of a memory location for reference.

* What is the starting address of the normal video display?

It starts at 1,024.

Program #5, 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/0 error occurs, rewind to the start of the program and try again. For severe loading problems, see the Appendix.

```
10 DATA 8E.06.00
20 DATA 10.8E.04.00
30 DATA A6.80
40 DATA A7.A0
50 DATA 8C.08.00
60 DATA 26.F7
70 DATA 39
110 REM LOAD X WITH $0600 MESSGE
120 REM LOAD Y WITH $0400 SCREEN
130 REM LOAD A FROM X-INDEXED,          AND INCREMENT X BY 1
140 REM STORE A TO Y-INDEXED,           AND INCREMENT Y BY 1
150 REM COMPARE IF X IS $0800
160 REM BRANCH BACK IF NOT
200 FOR N = 16000 TO 16016
210 READ A$
220 A=VAL("&H"+A$)
230 POKEN,A
240 NEXT
250 CLS
260 PRINT"THE MESSAGE YOU ARE READING WAS ORIGINALLY DISPLAYED B
Y PRINTINGIT NORMALLY USING BASIC.  IT CANBE RECALLED AT ANY TIM
E -- ONCE THE BASIC PROGRAM HAS BEEN RUN  -- BY TYPING "CHR$(34)
"EXEC"CHR$(34)"."
270 PRINT"THE BASIC PROGRAM PLACED INTO   MEMORY THE 6809 MACHIN
E LANGUAGEPROGRAM DESCRIBED IN LESSON 3.  IN THIS PROGRAM, INDEX
 REGISTERSX AND Y ARE USED TO TRANSFER A  GROUP OF BYTES (IN THI
S CASE    ORDINARY ASCII CHARACTERS) FROM ";
280 PRINT"ELSEWHERE IN MEMORY DIRECTLY TO THE SCREEN MEMORY."
290 PRINT" "STRING$(30,191);
300 FOR N = &H400 TO &H5FF
310 POKEN+&H200,PEEK(N)
320 NEXT
330 CLS
340 FORN=1TO1000:NEXT
350 EXEC16000
999 GOTO999
```



RUN this program. A message printed by BASIC will appear on the screen, the screen will be cleared, and the message will appear again, this time printed by the machine language program I've just described. Now I feel bound to prove that I'm not fooling you with some fancy BASIC manipulations. Once you have RUN the program the first time, hit <BREAK> and then delete it. Type NEW to clear out the program. Now, I say smugly, type EXEC — that's E-X-E-C — and hit <ENTER>. The message reappears, partly obliterated by an "OK".

To see the whole thing, enter these two lines. Line 10. EXEC. Line 20. GOTO 20. That's it. Line 10. EXEC. Line 20. GOTO 20. Now RUN that. There's the message.

Have some fun. Try changing the program. Hit <BREAK>. POKE 16001,128 <ENTER>. POKE 16012,130. <ENTER>. Then RUN. You can POKE

16001 with any number from 0 to 253. POKE 16012 with the previous number plus 2. For example, Hit <BREAK>. POKE 16001,0 <ENTER>. POKE 16012,2 <ENTER>. RUN again. Take a break here to load and RUN the next program. When you RUN it, notice that it expects you to input a hexadecimal number this time.

Program #6, 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/0 error occurs, rewind to the start of the program and try again. For severe loading problems, see the Appendix.

```
10 INPUT"MEMORY LOCATION (00 TO FE)";A$
20 A=VAL("&H"+A$):IFA>254ORA<0THEN10
30 POKE16001,A
40 IFA=254THENB=0ELSEB=A+2
50 POKE16012,B
60 EXEC
70 FORN=1TO1000:NEXT
80 CLS
90 GOTO10
```

You've RUN the program, and seen a number of curious screen displays. What you have done is simple. You redirected the X register, which was pointing to the message I stored in memory, to somewhere else in memory. You can see that it takes very few changes to spy anywhere into memory with even that little machine language program.

Return to Figure 5 in the MC6809E data booklet. At this point, I have introduced the A and B accumulators, the program counter PC, and the X and Y index registers. Again, if you feel you might need to review, this is the time to do it.

Turn your attention to Figure 5, and notice the bottom register marked CC — the condition codes. This special register gives the processor its limited intelligence. Also called the "flags", the condition code register contains bit-by-bit information about the processor's activities . . . what the processor does, and what the results indicate. In the beginning, the flags of most interest will be the Carry/Borrow Flag and the Zero Flag.

In this lesson's first program example, I had the A accumulator load a value from memory indexed by X, and store that value in memory indexed by Y. My program did that for exactly one screen full of information — 512 bytes. I should say that my program did that for exactly hex 200 bytes, which is an easier number to work with.

How did my program know to stop after **$200** bytes? Turn to your documentation book for this lesson, and follow

* That number was decimal since it didn't have a dollar sign in front of it. What is that starting address in hexadecimal?

1,204 in hexadecimal is $0400.

* If the X register is indexed to an ASCII character in memory and the Y register is indexed to the video display at $0400, how can the A accumulator get the message to the screen?

The A accumulator can load the value from memory indexed by X and store the value to memory indexed by Y.

* How does it do this?

It follows instructions.

* Where does it get the instructions?

It gets the instructions from memory.

* What is another name for the condition codes?

The flags.

* What information is held by the condition codes?

Bit-by-bit information about the processor's activities.

* What activities is the processor engaged in?

The instructions it is following.

* What keeps track of the instructions it is following?

The program counter, or PC.

* Now, where were we?

Talking about the condition codes, or flags.

* Oh yes. What does the instruction "compare" do?

It compares the contents of a register against another value.

* In what way does it compare?

It compares by performing a "ghost" subtraction.

* What does the ghost subtraction do?

It sets the condition codes (or flags) according to the results of the ghost subtraction.

* What are the results and the condition codes if the register's value is greater than the value being compared with?

If the register's value is greater than the value being compared with, the ghost subtraction causes no borrow and the result is not zero. The carry/borrow and zero flag is turned off.

* What are the results and condition codes if the register's value is the same as the value being compared with?

There is no borrow, but the result of the ghost subtraction is zero. The carry/borrow flag is off, but the zero flag turns on.

* Well, then, what if the register's value is less than the value being compared with?

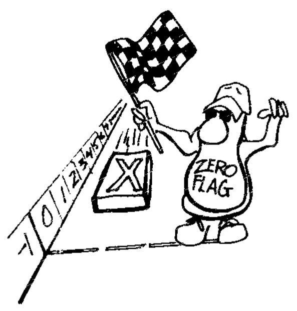
The result isn't zero, but the ghost subtraction demands a "borrow". The carry/borrow flag goes on, but the zero flag is off.

* So how do you use this information?

By learning the principles in this lesson very well before going on to the next lesson.

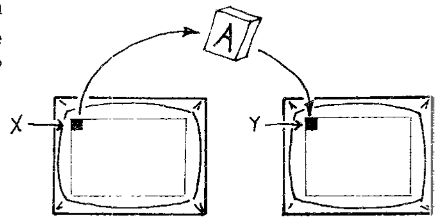
along with me as I describe a little more precisely how this program operates.

> **Step 1.** Load X register with the immediate value of **$0600**. This is the address where the message is stored in memory.
>
> **Step 2.** Load Y register with the immediate value of **$0400**. This is the address where the screen begins on the Color Computer.
>
> **Step 3.** Load A accumulator from memory indexed by X, and automatically increment the X register by one.
>
> **Step 4.** Store A accumulator to memory indexed by Y, and automatically increment the Y register by one.



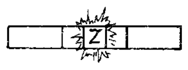
At the end of this step, there is a letter on the screen. X, having been incremented, now indexes the *second* character of the message, and Y, also having been incremented, indexes the *second* location on the screen. What I would like the program to do is somehow check to see if the job is finished. Here are the questions to consider:

When would the job be finished? When the screen is full. When is the screen full? If Y has been incremented, one step at a time, past the last screen position, and X has been incremented, one step at a time, past the last letter of the message. When is X past the last letter of the message? When it reaches **$0800**.



So the actions of Step 5 becomes clear.

> **Step 5.** Compare X register to the immediate value **$0800**.



"Compare" is a microprocessor instruction which does what might be called a "ghost" subtraction. The only purpose of the ghost subtraction's result is to discover if the value being compared *with* is higher, lower, or equal to the register being compared *to*. Both original values remain unchanged — no actual result has been produced — but the result of this comparison can be discovered by the 6809 reading the condition codes. Here's how it goes:

If the register's value is greater than the value being compared with, then the ghost subtraction results in a non-zero, positive number. Both the carry/borrow flag and the zero flag are reset — turned off, that is.

If the register's value is equal to the value being compared with, then the result of the ghost subtraction is zero, turning on the zero flag but leaving the carry/borrow flag off.

And finally, if the register's value is less than the value being compared with, then the ghost subtraction results in a negative (but non-zero) number. The carry/borrow flag goes on, the zero flag goes off.

So how does it fit here? At this point, I'm going to do something I like to avoid, and that's to explain assembly language using similar BASIC commands.

Program #7, 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 X = &H0600
20 Y = &H0400
30 A = PEEK(X)  : X = X + 1
40 POKE Y,A     : Y = Y + 1
50 IF X <> &H0800 THEN 30
60 END
```

This is a short program, and I'd like you to list it before you run it. In case you've never used BASIC's peculiar notation for hexadecimal, it's "ampersand H". Now in the 6809, X, Y and A registers are *not* variables. I'm using those names here just for visual effect. But follow this through. In line 10, X is **$0600**, the first memory location of the message. In line 20, Y is **$0400**, the first memory location of the screen. In line 30, A takes the value indexed by X — here I use PEEK to create the same effect — and X is incremented by one. In line 40, A stores its value at the location indexed by Y — here I use POKE to create that effect — and Y is incremented by one. In line 50, the compare is done. X is compared with **$0800**; if it isn't **$0800**, then the program isn't done, and it branches back to line 30.

RUN the program. It does, quite slowly, exactly what the machine language program did. To finish this lesson, load and examine the source code that follows on this tape.

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
```

Review:

* What does CPU mean; what CPU does the Color Computer use?

Central Processing Unit; the Color Computer uses a 6809 CPU.

* What are the terms for one binary digit and for eight binary digits?

The terms are bit and byte.

* What does ALU mean, and what kinds of arithmetic does the 6809's ALU perform?

ALU means Arithmetic Logic Unit, and the 6809's ALU performs addition, subtraction, multiplication, AND, OR, NOT, Exclusive OR, incrementing, decrementing, and comparison.

* How many ALUs does the 6809 have, and what are they called?

The 6809 has two ALUs called the A and B accumulators.

* Where do the accumulators get and save their information, and what are the terms for getting and saving data?

The accumulators the information from other registers and from memory; the process is loading and storing data.

* What is the address range of the 6809 CPU in hexadecimal?

The address range is $0000 to $FFFF.

* How does the processor get started, and what keeps track of its intructions?

By loading and concatenating the data at memory locations $FFFE and $FFFF, and using the result as the address of its first instruction. The program counter, or PC, keeps track of the instructions.

* What are the index registers, what do they hold, and what are they for?

The index registers are X and Y, they hold 16 bits each, and they are most often used to hold the address of a memory location.

* What are the condition codes?

The condition codes are bits that hold information about the processor's activities.

* Give another name for the condition codes, and name two of the codes.

Condition codes are also called flags; carry/borrow and zero are condition codes.

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.

* 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/0 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**

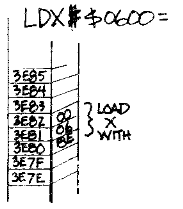
- Load A with the value from memory indexed by X, and increment X by one

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

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

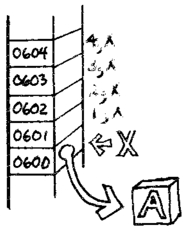
- Branch if the result of the previous computation was not zero, that is, if not equal, back to the instruction labeled LOOP.

- 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.



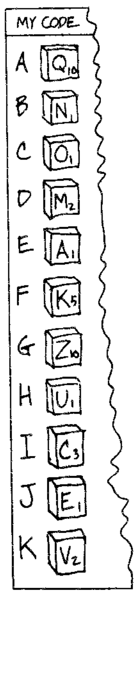
I've used some new terms. "Immediate" value is one of them, one which I slipped into the previous lesson. "Immediate" is a piece of jargon I'm not fond of, but it's the formal term meaning "use this actual number". In line 100, that means Load X with the number hex **0600**. The number sign preceding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

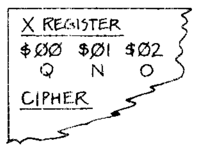
Uppercase A is $41, lowercase a is $61

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

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

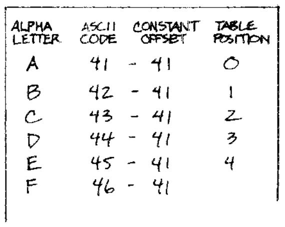
order, but the hex numbers will be in order. Turn this tape back on when you're finished; turn the tape off now.

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.

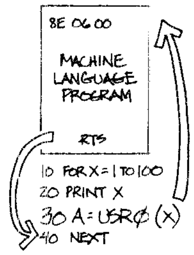
Let's say the value is in ASCII, a normal state of affairs for these machines. Letter A is ASCII hex **41**, letter Z is hex **5A**. The question is how to get from ASCII values **$41** through **$5A** to the encrypted values in the table, which are numbered **$00** through **$19**. There's really no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



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	

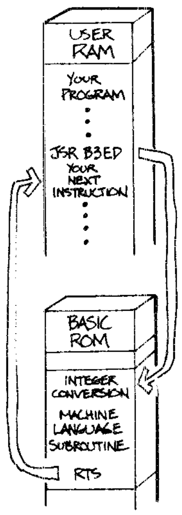
So the ASCII values come in from somewhere, you subtract **$41**, and the resulting number is the position of the encrypted value in the table. You extract the value from that position, and the encoding is done.

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.



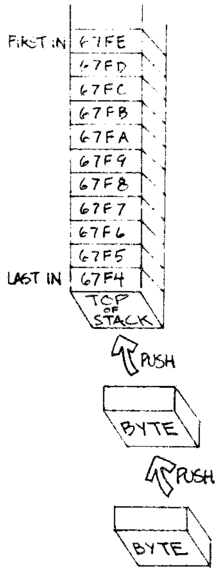
Please read pages 145, 146 and 147 in the Extended Color BASIC manual. This is the beginning of the chapter called "Machine Language Routines".

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.



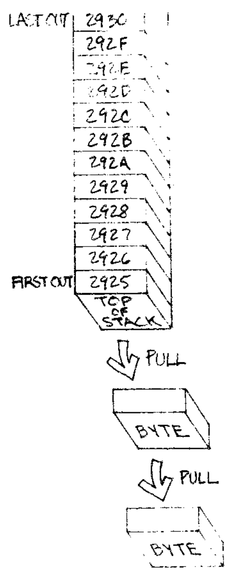
You'll actually be creating a working program, so you need a jumping off place. BASIC is good. You can transfer a value from BASIC to machine language; it's part of the USR command. In your Extended Color BASIC book, the USR function was described. The "argument" they're talking about is the value transferred 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.

* 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.

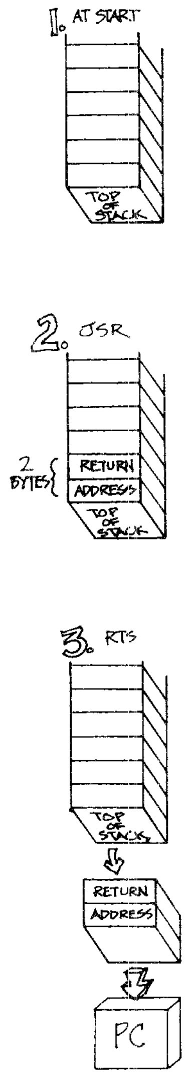
Now the mnemonics. To push a value on the Hardware Stack, the mnemonic is "pushstack" — PSHS. The operand is the set of registers you wish to push. To push X, Y, and A, for example, you would pushstack X Y A — **PSHS X,Y,A**.

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.

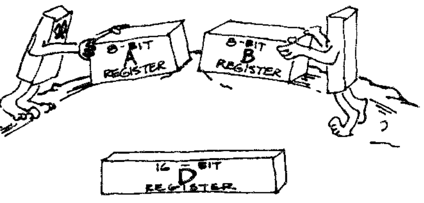
The encryption table has to be identified. Write Load X with immediate value CIPHER. Write "LDX" and across from it write "#CIPHER". The X register is pointing to the zeroeth entry in the encrypted ASCII table.

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**.



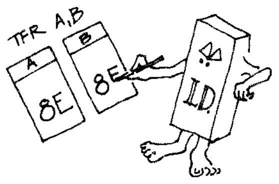
The magic is next. You know that the B register contains a value from **$00** to **$19**. You know that X is pointing to the zeroeth value in the encrypted table. All that's left of the hard work is to use that information to find the value you want from the table. That value is found at the address 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**.



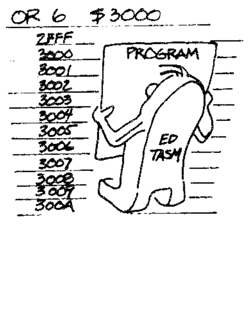
Now A and B contain the same value. You want A to be zero, so clear it. Write **CLRA**. It looks like most of the work is done, so call that routine that gives the value to BASIC. Write **JSR $B4F4**. Now get the Y register back (you do remember you saved the Y register, don't you). Pullstack Y. Write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

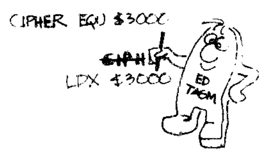
* Write the mnemonic for pushing both the A accumulator and X register on the hardware stack.

PSHS A,X

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

Pullstack S, or PULS.

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

* 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 $B3ED, 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 $3F00 mean?

It means the first memory location in a mnemonic listing is $3F00.

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/0 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?

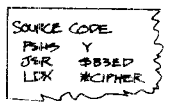
An optional label.

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

The opcode.

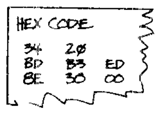
There's the program listing in front of you. Let me refresh your memory as to what this means. The label CIPHER was used to indicate a memory location **$3000**. The origin, that is the first instruction, of the program itself was set in memory at **$3100**. My choices here were arbitrary; and free memory 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.



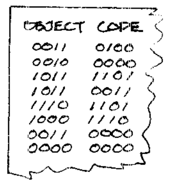
```
SOURCE CODE
 PSHS   Y
 JSR    $B3ED
 LDX    #CIPHER
```

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.



```
HEX CODE
 34   20
 BD   B3   ED
 8E   30   00
```

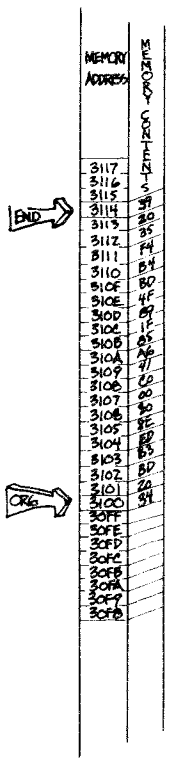
In this lesson, I want to guide you in using the editor/assembler, but first I would like you to see exactly what it's for. Type A, and hit <ENTER>. You'll see the "READY CASSETTE" message, meaning it's about to prepare an object code tape. "Object code" is the jargon for a set of binary instructions 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.



```
OBJECT CODE
 0011   0100
 0010   0000
 1011   1101
 1011   0011
 1110   1101
 1000   1110
 0011   0000
 0000   0000
```

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.

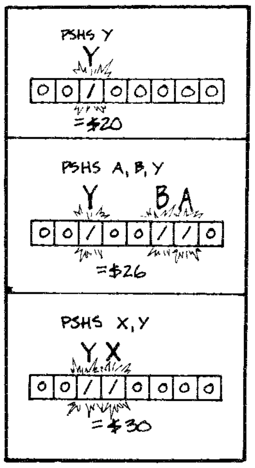
In the first column, the memory locations, that is the memory addresses to hold the program, are presented in hexadecimal. In this case, the program's first instruction begins at **$3100**, and the last instruction is found at **$3114**. The second and third columns contain the actual instructions and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

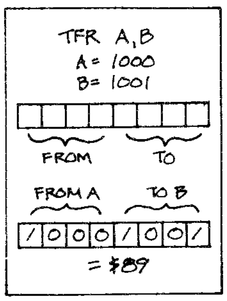
Each of these opcodes, after interpretation by the processor's internal instruction decoder, gives the 6809 information about what to do, what data is coming up next, and how many bytes long the operand will be. The operands themselves vary according to what the instruction demands. In lines 130, 140 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?

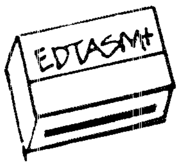
Push the Y register on the hardware stack.

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?

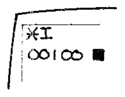
Binary 00110100 00010100, hexadecimal $34 14.

* What is another name for this kind of operand?

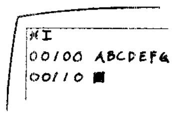
A postbyte.

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

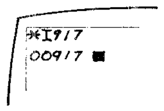
There is no requirement to manually number every line in EDTASM+, saving you considerable time and energy. Simply type and enter 'I'. The first available line number, 00100, is presented with the cursor ready for your information. You may now type anything you like on this line. Since renumbering and 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>.



```
*I
00100 ■
```

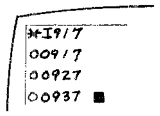


```
*I
00100 ABCDEFG
00110 ■
```



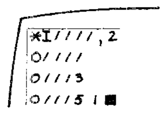
```
*I917
00917 ■
```

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.



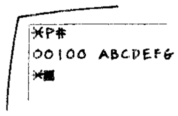
```
*I917
00917
00927
00937 ■
```

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.



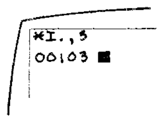
```
*I1111,2
01111
01113
01115 1 ■
```

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



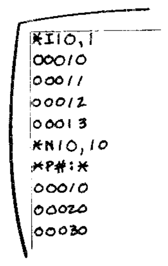
```
*P#
00100 ABCDEFG
*■
```

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.



```
*I.,5
00103 ■
```

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



```
*I10,1
00010
00011
00012
00013
*N10,10
*P#:*
00010
00020
00030
```

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.

6.

Hello again. Now that you have a firm grounding in using the editor/assembler, I've got to talk about some things that don't make me very happy. Those things make up the jargon of microprocessor programming. It's struck me that the major barrier to programming in assembly language is the terminology. The concepts themselves are simple — sometimes far too simple and endlessly tedious for fun, but simple nevertheless. But that simplicity also derives out of the arbitrariness of their origins.

I don't want to sound philosophical, but I've often been asked the question "why". Why "load" and "store" instead of something like "input data" and "output data"? Why a clumsy sounding word like "immediate"? How did the binary values get chosen for the instructions? The answers go back to the early days of computers and processors. In the same way that a "word of eight binary digits" became a "word of eight bits" and that in turn became known simply as a "byte", many of the terms involved in assembly programming are just arbitrary, and sometimes tongue-in-cheek, choices that stuck. Some were chosen because the alternatives are worse . . . "load immediate", for example. "Load absolute" implies a positive number so that's out; saying "load this number" or "load what's next" sound too silly for programming terms, even though a number sign actually precedes the operand and it is what's next.

The jargon can get overwhelming. If that weren't so, you probably wouldn't be listening to me now. It's not the programming that's hard; it's learning the language, from the descriptive terms through the programming actions. Yet I believe jargon is really essential to facilitating communication... so long as you *know* the jargon. A friend of mine once wrote that we're not intimidated by admitting, in pure, modern jargon, "I took a 747 non-stop"; we wouldn't think of saying "I flew inside a big silver bird who never paused to eat or drink."

There's truth in that comment; in the earlier lessons, some of you probably got tired hearing me say "American

This lesson begins the first of two lessons on the critical concept of addressing modes. The term sounds dry, the learning isn't especially fun, and the jargon is trying. Yet addressing modes give the 6809 processor its power. Before you begin, be sure you know the basic terminology presented in the previous lessons, and how to use EDTASM+.

* What does ASCII mean?

American Standard Code for Information Interchange.

* What is the term for an accumulator obtaining information from memory?

Loading.

* What is the term for an accumulator placing information in memory?

Storing.

* What is the term for one register placing information in another register?

Transferring.

* What is a word of eight binary digits?

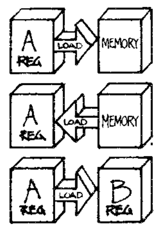
A byte.

Standard Code for Information Interchange". You knew I meant ASCII, I knew I meant ASCII, so why didn't I say so? I wanted you to know intuitively that this was a code for the interchange of information, not just letters.

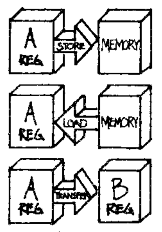
In a similar way, I was mystified by hockey terminology. Here were tens of thousands of people understanding the announcer's every phrase, understanding the motion of the puck as if it were their own heartbeats. I ate some popcorn, yelled a little, but mostly read the advertisements on the sideboards. The game began to take on multiple levels of excitement only when I began to understand its language.

There are also are those who consciously attempt to alter a language to simplify it, even to the point of creating new languages in the process. BASIC was one of the successes, Esperanto was one of the failures. The contemporary Russian alphabet was a success, Chicago school of spelling was a failure. I have an example relevant to this course. The creators of the Z80 thought "load" and "store" were really just directional variants of one concept, so they decided all such actions would be called "loads". That decision, while advantageous for learning the Z80 processor, stands in the way of someone being fluent on several microprocessors. It has made the Z80 dialect different from the 6809 dialect, where those variants were even further refined into "loads", "stores", and "transfers".

Z80 DIALECT



6809 DIALECT



I'm not stalling here, I'm just trying to prepare you for this lesson. The terms I am going to introduce all have specific meanings, and some are quite elegant summaries of complicated concepts. You already know one of them — the indexed addressing mode. There's a lot like that coming up, so take your time; don't rush. Review when you need to. You hired me to do this job, after all, and I'll patiently re-explain as often as you like.

The topic is addressing modes. That's how the processor obtains the data it needs to complete a given instruction. For this topic, I would like you to follow along with me in the documentation; these things are often easier to see than to say, especially when it comes to mnemonics. You'll also need to open your MC6809E data booklet to page 15, and have a marker on page 28.

While you're finding your place, and before actually discussing addressing modes, I'd like to recap the concept of addressing itself. The 6809 microprocessor has an 8-bit data bus and a 16-bit address bus. This means that it has 24 electrical connections to an external line of memory cells. A memory cell in this line is activated when it receives its particular 16-bit binary number from the processor on the address bus. Each memory cell is electrically connected in such a way that it — and only it — can respond to that binary address. When it responds, data is sent from or received by the 6809 along the 8-bit data bus. 6809 sends

* What is an addressing mode?

An addressing mode is how the machine language program gets its information.

* In the 6809, what is the size of the data bus?

The data bus is 8 bits wide.

* In the 6809, what is the size of the address bus?

The address bus is 16 bits wide.

* When does a memory cell appear "live"?

When it receives its particular 16-bit binary number from the processor.

* How is the 16-bit binary number sent by the processor?

By sending it on the address bus.

* How does the memory respond when it receives its address from the processor?

By sending or receiving data.

* How is data sent or received?

Along the data bus.

* What is the size of the 6809's data bus?

The 6809's data bus is 8 bits wide.

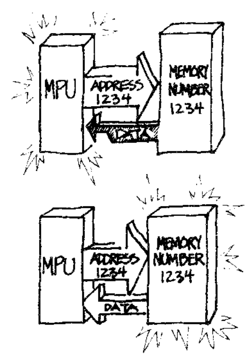
* What is an addressing mode?

An addressing mode is how the machine language program gets its information.

* Where does the processor get its data?

From memory.

the address, memory responds by sending or receiving the data.

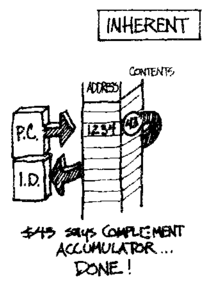


You don't need to know much about this electrical process; for programming purposes, you take it on faith that the machine's designers have organized the connections properly so that when your program wants information from memory location **$1234**, for example, memory location **$1234** will respond appropriately and provide your program with that information. Later you'll learn a little more about dealing with computer input and output, for which a touch of electronics will enter into the discussion.

As for addressing, you know now that the processor takes both its program and its data from memory, and stores its data in memory. Up to this point, I've presented concrete examples of specific memory uses — to store and execute the opcodes and operands of a program, and to store a table of data. I don't feel that learning through concrete example alone will broaden your programming abilities, so it's on to the discussion of the addressing modes. If at any point you get lost in the jargon or feel shaky about this, remember: AN ADDRESSING MODE IS HOW THE MACHINE-LANGUAGE PROGRAM GETS ITS INFORMATION.

Look at page 15 in the MC6809E data booklet. As noted, there are seven major categories of addressing modes in the 6809: inherent, register, immediate, extended, direct, indexed, and relative. The next two lessons will cover all seven modes; I'll save for later the three variants called extended indirect, indexed indirect, and program counter relative. Throughout this discussion, please remember that "opcode" means the machine-language instruction, and that "operand" means its data.

Inherent Addressing



Inherent addressing is the simplest mode. In this mode, all the information needed to complete the processor instruction is already present in the instruction itself. In other words, the address of the data needed to complete the instruction is inherent in the address of the instruction's opcode, which the processor's already got. You've used two of these inherent instructions up to this point: Clear A Accumulator (mnemonic **CLRA**, hex code **4F**) and Return from Subroutine (**RTS**, **$39**), both of which are inherent addressing. They have all they need to get the job done. Other examples of this mode are Multiply A Accumulator times B Accumulator (**MUL**, **$3D**). There's also Complement A Accumulator — that is, turn all zero bits to one, and all one bits to zero (mnemonic **COMA**, **$43**), and even No Operation (N-O-P or **NOP**, **$12**), which does nothing but waste time. If this last one sounds funny to you, you'll later discover how important it can be to waste time, since machine language actually moves too fast for some programming.

* Where does the processor get its program?

From memory.

* How does the processor distinguish program from data?

By the context.

* What is the term for how a machine language program gets its information?

An addressing mode.

* What is the term for a machine language instruction?

An opcode.

* What is the term for an opcode's data?

An operand.

* What addressing mode includes the information necessary to complete the instruction as part of the instruction itself?

Inherent addressing.

* Give examples of inherent addressing.

Any of the following will do (this isn't a complete list): CLRA, CLRB, RTS, MUL, COMA, COMB, NOP, ASLA, ASLB, ASRA, ASRB, DECA, DECB, INCA, INCB, LSLA, LSLB, LSRA, LSRB, NEGA, NEGB, ROLA, ROLB, RORA, RORB, TSTA, TSTB.

* What is inherent addressing?

Inherent addressing is an addressing mode in which the information needed to complete an instruction is part of the instruction itself.

* What is register addressing?

Register addressing is an addressing mode in which the information needed by the program is moved from one register to another.

* Give two examples of register addressing.

TFR and EXG. PSH and PUL can be considered register addressing.

* What addressing mode involves movement of data from register to register?

Register addressing.

* What addressing mode finds the data at the address immediately following the instruction itself?

Immediate addressing.

* Give examples of immediate addressing (make up operands for your examples).

Any of these will do: LDX #$3000, SUBB #$41, CMPX #$0800, LDA #$12, LDY #$1234, CMPY #$CCCC, etc.

* What is immediate addressing?

An addressing mode in which the data to be used is found at the address immediately following the instruction itself, in program order.

* What is extended addressing?

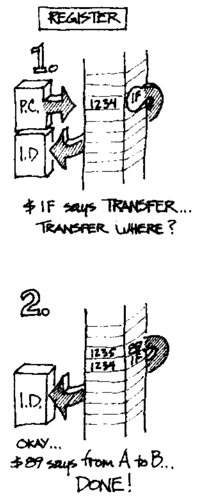
An addressing mode in which the two bytes following the opcode are the address of the data to be used to complete the instruction.

* In the instruction LDX $3456, where is the data?

The data is found at address $3456.

Register Addressing

The second mode is Register Addressing. In this case, the information needed by the program is transferred from one register to another. For example, the familiar Transfer Value from A Accumulator to B Accumulator (**TFR A,B**) is Register Addressing. This instruction is two bytes, the opcode meaning "transfer from register to register" (**$1F**) and the operand — called a "postbyte" — identifying which goes where (**$89** for transferring A to B). Another example of register addressing that you have used is Push Y and Pull Y (**$34 $20** and **$35 $20**). New examples include Exchange Registers (two bytes with an opcode of **$1E**), and all the other Push and Pull instructions (opcodes **$34** and **$35**, respectively).

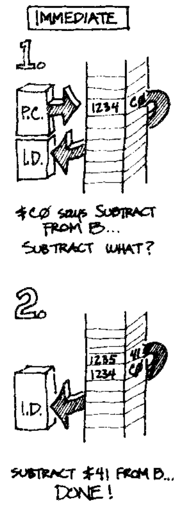

REGISTER
1.
P.C.
1234
1F
I.D
$1F says TRANSFER... TRANSFER WHERE?
2.
1235
1234
89
1F
I.D.
OKAY... $89 says from A to B.. DONE!

Don't be confused by the MC6809E data booklet; Register Addressing is easy. The data booklet first suggests that Register Addressing can be thought of as either distinct from or the same as Inherent Addressing. I leave that up to you, because the MC6809E data booklet can't make up its mind, either. The booklet clearly distinguishes between Register and Inherent Addressing on page 15, but calls them both "Inherent" on pages 28 and 29. To assist in the confusion, it even calls one group "Immediate" on page 31! I prefer to consider Register Addressing as distinct from Inherent Addressing. The opcode is all the information in the Inherent mode, but in Register Addressing, the data necessary to complete the instruction is described by the postbyte. If I've just confused you, then you may, as the judge says, disregard the previous remarks.

To recap: Inherent Addressing is a mode in which the address of the operand also addresses the data needed to complete the instruction, since the data is an inherent part of the instruction itself. Register Addressing is similar to Inherent addressing, and often includes a second byte known as a postbyte to furnish additional information needed to complete the instruction. Inherent and Register Addressing include Clearing, Incrementing, Decrementing and other internal single-register commands; Exchanging, Transfering and other register-to-register commands; Stack Pushes and Pulls; Subroutine Returns; and one-of-a-kind, specialized arithmetic functions such as Multiply, Sign Exchange, and Add-B-Register-to-X-Register.

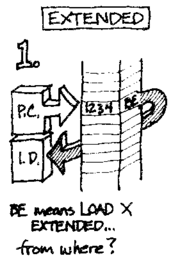
If you wish, review Inherent and Register Addressing in your documentation. For review, turn the tape off *now*.

Immediate Addressing

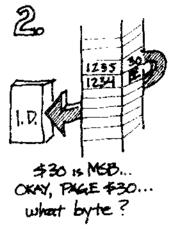

IMMEDIATE
1.
P.C.
1234
CØ
I.D.
$CØ says SUBTRACT FROM B... SUBTRACT WHAT?
2.
1235
1234
41
CØ
I.D.
SUBTRACT $41 FROM B... DONE!

Immediate Addressing is very transparent. The data to be used is found at the address immediately following the instruction itself, in program order. Among examples you have used already are Load X Register with value **$3000** (written **LDX #$3000**), and Subtract the value **$41** from B Accumulator (written **SUBB #$41**), and Compare X Register with **$0800** (written **CMPX #$0800**). Other examples include such logical instructions as AND A Accumulator with an immediate value, OR B Accumulator with an immediate value, Exclusive OR, and so forth; arithmetic such as ADD A Accumulator and SUBtract A Accumulator; and the now-familiar Load A, Load B, Load X, Load Y, etc., with an immediate value. The mnemonic notation for Immediate Addressing *always* includes the number sign in front of the operand, which tells the editor, "use this data!"

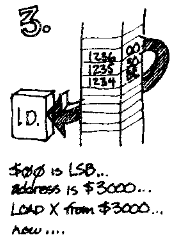
Extended Addressing



The word "Extended" implies reaching out, and Extended Addressing is just that. In Extended Addressing, the information following the opcode (that is, following the machine-language instruction itself) is *not* the data. What follows the opcode is the *address* in memory where the data can be found, rather than the actual data to be used. Here's an example. You have used **LDX #$3000**, which meant Load X with the immediate value **$3000**. In Extended Addressing, the notation is **LDX $3000**. Very similar, but with an entirely different meaning; glance at the documentation so you can see what I'm describing. **LDX #$3000** is immediate addressing; **LDX $3000** does not contain the number sign in front of the operand. That means that **$3000** is not the data, but is the address in memory where the processor will find the data to be loaded into X.



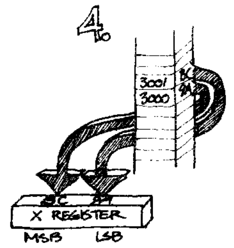
Did a question come to mind? How can the 16-bit X register load the 8-bit data at address **$3000**? Since the data at address **$3000** is only an 8-bit word, and since the X register requires 16 bits, the instruction decoder sees to it that the process is completed correctly. The information loaded into X is in fact all 16 bits. The first byte comes from the address specified by the operand (in this case **$3000**), and the second byte comes from the next address (in this case **$3001**), in order.



Extended addressing is used for both 8- and 16-bit registers. If the command were **LDA $3000**, then, the instruction decoder would make sure the 8-bit value at **$3000** was loaded into the 8-bit A Accumulator.

Here are two concrete examples:

- The instruction is **LDX $1234**. Address **$1234** contains **$AB**, and address **$1235** contains **$FF**. After executing the instruction **LDX $1234**, the X register will contain the value **$ABFF**.



- The instruction is **LDB $8888**. Address **$8888** contains **$10**. After executing the instruction **LDB $8888**, the B Accumulator will contain the value **$10**.

In all this, the 6809 processor's task is to be smart enough to place the information found at the specified memory location into the correct registers, making sure the number

* What kind of addressing mode is LDX $3456?

Extended addressing.

* In the instruction LDX #$3456, where is the data?

The data is immediately following the instruction; that is, the data is $3456.

* What kind of addressing mode is LDX #$3456?

Immediate addressing.

* What kind of addressing mode is LDA $1234?

Extended addressing.

* The B register contains $41; the A register contains $00; memory location $1111 contains $45. What are the contents of the A accumulator after each of the fllowing instructions are executed?

```
LDA  #$49
LDA  $1111
TFR  B,A
```

$49; $45; $41

* What addressing modes are LDA #$49, LDA $1111 and TFR B,A?

Immediate, extended and register addressing.

* What is an addressing mode?

How the machine language program gets its information.

* What ASCII characters are represented by $49, $45 and $41?

I, E and A

* What is direct addressing?

Direct addressing is an addressing mode where the direct page register and the value following the opcode are combined to form an address. At that address is found the data to complete the instruction.

* The DP register is set to $CC and the instruction LDA <$80. Where is the data?

At address $CC80.

* The DP register is set to $80 and the instruction is LDA <$CC. Where is the data?

At address $80CC.

* For each of the following examples, identify the addressing mode, and tell specifically where the data is found. Assume the direct page register is set to $A0.

* LDA #$41

Immediate; following the opcode LDA.

* LDX $3456

Extended; at addresses $3456 and $3457 (X needs two bytes).

* CLRA

Inherent; as part of the instruction.

* STA <$CC

Direct; at address $A0CC.

* TFR X,Y

Register; as described by the postbyte.

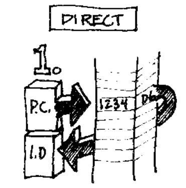
* CMPA $789A

Extended; at address $789A.

of bytes taken from sequential memory locations matches the size of the register requesting the data.

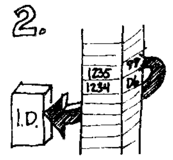
Direct Addressing

Direct Addressing obtains data for program use with great speed and memory economy. It depends on the organization of memory into pages. A "page" is a specific term in assembly language programming, meaning those 256 contiguous bytes of memory whose most-significant-byte is in common. For example, page **$00** contains the 256 addresses **$0000** to **$00FF**; page **$01** contains addresses **$0100** to **$01FF**; page **$FE** contains addresses **$FE00** to **$FEFF**. The 6809 and other 8-bit processors have a total 256 pages of 256 bytes.



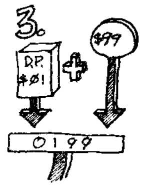
$D6 means LOAD B DIRECT... from where?

Return to the MC6809E data booklet, and turn to Figure 4 on page 5. That's the 6809 architecture you've been using. Up to this point, you have been introduced to all registers in the 6809 except one: the Direct Page register. Into the Direct Page register is transferred the most-significant byte of an address. In earlier processors, the direct page was fixed (usually to page **$00**), and consequently there was no Direct Page register. But the 6809 has this Direct Page register because its Direct Addressing can be done anywhere in memory.

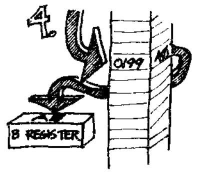


$99 is LSB...
LOAD FROM DIRECT PAGE BYTE $99...

So what's the point? First of all, each instruction using Direct Addressing takes one less byte of memory than Immediate or Extended Addressing. Since the most-significant byte is always ready for use in the Direct Page register, that byte need not be stored in program memory as part of the operand. Secondly, since Direct Addressing fetches one less byte from memory, the instruction can be completed faster.



The mnemonic notation for Direct Addressing uses the "less than" sign in front of the operand. For example, with the Direct Page set to **$AA**, the instruction **LDA<$80** would load the A accumulator with the value found at memory location **$AA80**. Beyond the economy of speed and memory, however, Direct Addressing is identical in principle to Extended Addressing: the desired data is not the operand itself, but at the memory location specified by the operand.



Examples

To review some examples of immediate, extended and direct addressing, follow me in your documentation booklet:

LDX #$1234 is immediate addressing, loading the value **$1234** into the X register.

LDX $1234 is extended addressing, loading the value found in memory at addresses **$1234** and **$1235** into the X register.

LDX <$34 is direct addressing; with the direct page set to **$12**, the value found at addresses **$1234** and **$1235** is loaded into the X register.

TFR Y,X is register addressing; if the value of the Y register is **$1234**, then the X register will be loaded with the value **$1234**.

LDB #$56 is immediate addressing, loading the value **$56** into the B Accumulator.

LDB $56 is extended addressing, loading the value found in memory at address **$0056** into the B Accumulator.

LDB <$56 is direct addressing; with the direct page set to **$00**, the value found at address **$0056** is loaded into the B Accumulator.

TFR A,B is register addressing; if the value of the A Accumulator is **$56**, then the B Accumulator will be loaded with the value **$56**.

CMPY #$789A is immediate addressing, comparing the value of the Y register with the actual value **$789A**.

CMPY $789A is extended addressing, comparing the value of the Y register with the value found in memory at locations **$789A** and **$789B**.

CMPY <$9A is direct addressing; with the Direct Page register set to **$78**, the values found at **$789A** and **$789B** are compared with the Y register.

CMPA #$BC is immediate addressing, comparing the value of the A Accumulator with the actual value **$BC**.

CMPA $BC is extended addressing, comparing the value of the A Accumulator with the value found in memory at **$00BC**.

CMPA <$BC is direct addressing; with the Direct Page register set to **$00**, the value found at **$00BC** is compared into the A Accumulator.

To review the major points: Addressing is the manner in which the program obtains the data it needs. An opcode is a machine language instruction. An operand is the information needed to complete an instruction.

The Inherent Addressing mode contains only an opcode. That opcode contains sufficient information to complete the instruction. Because there is no operand needed to provide additional data, the data is inherent in the address of the instruction.

The Register Addressing mode contains an opcode and usually a postbyte. The opcode tells the processor which kind of instruction will be executed, and the postbyte

* LDY #$CBA9

Immediate; the two bytes following the opcode LDY.

* STX <$

Direct: at address $A000 and $A001 (X is two bytes).

* COMB

Inherent: as part of the instruction itself.

* What is an addressing mode?

An addressing mode is how the machine language program gets its information.

* What is inherent addressing?

Inherent addressing is an addressing mode in which the information needed to complete an instruction is part of the instruction itself.

* What is register addressing?

Register addressing is an addressing mode in which the information needed by the program is moved from one register to another.

* What is immediate addressing?

An addressing mode in which the data to be used is found at the address immediately following the instruction itself, in program order.

* What is extended addressing?

An addressing mode in which the two bytes following the opcode are the address of the data to be used to complete the instruction.

Summary

* What is direct addressing?

Direct addressing is an addressing mode where the direct page register and the value following the opcode are combined to form an address. At that address is found the data to complete the instruction.

* What are the 6809's 16-bit registers?

The X and Y registers, the S and U stack pointers, and the PC (program counter). The D accumulator combines the A and B accumulators into a 16-bit register.

* What are the 6809's 8-bit registers?

The A and B accumulators, the CC (condition code) register, and the DP (direct page) register.

* Where does the processor get its data?

From memory.

* Where does the processor get its program?

From memory.

* How does the processor distinguish program from data?

By the context.

* What is the term for how a machine language program gets its information?

An addressing mode.

defines which registers will be used to complete the instruction.

The Immediate Addressing mode contains an opcode and one or two bytes of data. The opcode tells the processor which kind of instruction to execute, and the bytes of data are the specific information that is used by the processor to complete the instruction.

The Extended Addressing mode contains an opcode and two bytes of data. The opcode tells the processor which kind of instruction to execute, and the bytes of data are combined to create an address. At that address is found the data used by the processor to complete the instruction.

The Direct Addressing mode contains an opcode and one byte of data. The opcode tells the processor which kind of instruction to execute. The byte of data is used as the least-signficant-byte of an address, and the processor's internal Direct Page register is used as the most-significant byte. At the resulting adddress is found the data used by the processor to complete the instruction.

Please don't consider addressing modes just to be picky stuff. Virtually all the programming power of the 6809 processor comes from these addressing variants. I hope you will review this lesson several times until each of these five addressing modes begins to make sense.

7.

The topic is once again addressing modes, those ways in which the program gets the data it needs to complete a machine-language instruction.

I've described five modes so far: Inherent Addressing, an instruction which is essentially complete in itself; Register Addressing, where the opcode describes the instruction, and the postbyte indicates which registers are used; Immediate Addressing, where the necessary data immediately follows the opcode, within the program; Extended Addressing, in which the *two* bytes following the opcode are used to form the address where the data is located; and Direct Addressing, in which the *one* byte following the opcode is combined with the *one*-byte contents of the Direct Page register to form a memory address where the data can be found.

The remaining modes are Indexed and Relative Addressing, the topics of this lesson. As an aside, I know these two lessons are a little dry; I promise to do better soon, when you get back to hands-on programming.

Actually, you've already done Indexed Addressing. It's the most versatile way of getting data to your program, and it's quite easy to use. Any apparent complexity arises solely out of the incredible number of combinations you can make using this mode, each of which has its own jargon. The one unequivocal thing you *can* say about Indexed Addressing is that the operand in some way identifies the address at which the processor will locate the data it needs to complete the instruction. Don't forget during this that when I say something like "locate the data", I'm talking about loading, storing, comparing, adding, etc. — any machine language instruction that uses data to do its work.

In general, Indexed Addressing allows the processor to get data from memory by calculation. The memory location for that data is calculated by combining the value of a 16-bit register with an offset value. The offset can be either an actual numerical value or the value of an accumulator

You might be losing patience with these programmed learning sections. Keep up with them. Now they begin to take on more importance as the number of concepts you need to remember increases. Starting with the familiar...

* What is an addressing mode?

An addressing mode is how the machine language program gets its information.

* Name the addressing modes represented by these four instructions: CLRB, LDA #$99, LDX $05AA, STB <$33

Inherent; immediate; extended; direct.

* In inherent addressing, where is the data?

As part of the instruction.

* In immediate addressing, where is the data?

Following the opcode in memory.

* In extended addressing, where is the data?

At the address specified by the opcode.

* In direct addressing, where is the data?

At the address specified by the direct page concatenated with the information following the opcode.

* In all cases, where is the data?

In memory.

* In indexed addressing, data is found at an address in memory. What two things are necessary to locate the data?

A 16-bit register and an offset.

* What are the 16-bit registers in the 6809 processor?

X, Y, PC (program counter), S (hardware stack), and U (user stack).

* What are the three kinds of offsets used in indexed addressing?

Zero offset, constant offset, and register offset.

* Given a register and an offset, how are they used?

The value of the offset is added to the value of the register to calculate the address at which the data can be found.

* If the X register is $3000 and the A register is $41, where does the instruction LDB ,X find its data?

At address $3000.

* What kind of addressing is this?

Zero-offset indexed.

register. You've seen the usefulness of this method in that little code encryption program. The X register was set to the memory location at the start of the encryption table, and the offset added to pick your way through the table was in the B register.

These Indexed Addressing methods are called Zero-Offset Indexed, Constant-Offset Indexed, and Accumulator-Offset Indexed. More jargon. Zero-Offset Indexed means that what you see is what you get; the value in the register is the address of the data. Constant-Offset Indexed means that you're using a fixed constant — that is, a number other than zero — to add to the register's value in order to locate the data you need. Accumulator-Offset Indexed means that you can use the A, B, or combined D accumulator to give you in effect a variable offset. Add that variable offset to the register's value and you locate the data in memory.

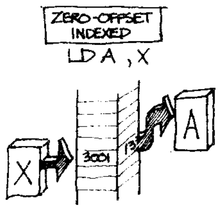
Indexed Addressing has other features. One of these is ostentatiously called Auto Increment/Decrement Indexed. It means that the register you're using to pinpoint a memory location may be incremented or decremented *as* the instruction is performed. As in the memory-to-screen message program you worked with earlier, this way of using Indexed Addressing makes transfer of information very quick and easy, requiring no additional steps to bump the register values along to the next byte in memory.

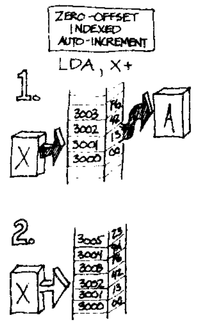
Although that program was used to transfer information just one byte at a time, in another situation you might want to use two-byte values. Therefore, the auto increment or decrement can be by either one byte as you've done, or by two bytes, further increasing the programming flexibility. For example, if you had stored a table of 16-bit integers, you would want to step through the table two bytes at a time to access its information.

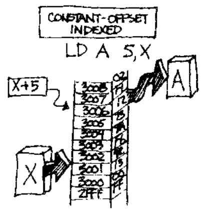
The Auto-Increment/Decrement Indexed mode has one quirk you have to keep in mind. When your memory pointer register is to be automatically incremented, that incrementing is done after the rest of the instruction is completed. But when a pointer register is decremented, that is done before the instruction is performed. Say that the value of the A Accumulator is to be stored at the memory location pointed to by Y. If an auto-increment is requested, A is first stored at Y, and then Y is incremented. However, if auto-decrement is desired, Y is first decremented, then A is stored at Y. This is a little awkward at first, but you'll find the programming makes sense to do that way. More on that later.

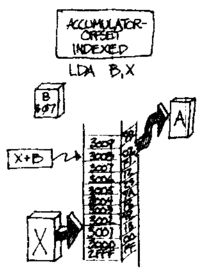
Now it's time to talk about mnemonics, which in this case will help make sense of Indexed Addressing. Please follow along with me in your documentation, and also have ready pages 16 and 17 of your MC6809E data booklet.

The format of the operand for Indexed Addressing is consistent. The offset is identified, followed by a comma,









and then the pointer register is named. I'm going to describe some variants on just one possibility, storing the A Accumulator at memory indexed by X:

Simply to store the A Accumulator at memory indexed by X, use the zero-offset indexed mode. It is written:

```
Mnemonic:               STA     ,X
Read:
                 Store A, zero-offset to X
Process:
1. Store A in memory location (  X  )
2. Change N and Z flags, reset V flag
3. Go on to next instruction
```

To store A at memory indexed by X plus an offset of **$10** bytes, use the constant-offset indexed mode. It is written:

```
Mnemonic:               STA     $1Ø,X
Read:
              Store A, constant offset $1Ø to X
Process:
1. Calculate X + $1Ø
2. Store A in memory location (X + $1Ø)
3. Change N and Z flags, reset V flag
4. Go on to next instruction
```

To store A at memory indexed by X, plus an offset of whatever value is in the B Accumulator, use the accumulator-offset indexed mode. It is written:

```
Mnemonic:               STA     B,X
Read:
             Store A, accumulator B offset to X
Process:
1. Calculate X + B
2. Store A in memory location (X + B)
3. Change N and Z flags, reset V flag
4. Go on to next intruction
```

Indexed examples

*** If the X register is $3ØØØ and the A register is $41, where does the instruction LDB $9C,X find its data?**

At address $3Ø9C.

*** What kind of addressing is this?**

Constant-offset indexed.

*** What is the constant in the previous example?**

$9C is the constant.

*** If the X register is $3ØØØ and the A register is $41, where does the instruction LDB A,X find its data?**

At address $3Ø41.

*** What kind of addressing is this?**

Accumulator-offset indexed.

*** What happens when LDA ,X is executed?**

The A accumulator is loaded with the value found in memory indexed by X.

*** What happens when LDA ,X+ is executed?**

The A accumulator is loaded with the value found in memory indexed by X, and then X is automatically incremented by one.

*** What addressing mode is this?**

Auto-increment/decrement indexed (specifically, auto-increment accumulator-offset indexed).

*** What happens when LDA ,-X is executed?**

The X register is decremented by one, and then the A accumulator is loaded with the value in memory indexed by the X register.

Indexed examples

* What addressing mode is this?

Auto-increment/decrement indexed (specifically, auto-decrement accumulator-offset indexed).

* What addressing modes are represented by these three instructions?

```
LDB   ,X
LDB   $19,X
LDB   A,X
```

Zero-offset indexed, constant-offset indexed, and accumulator-offset indexed.

* What addressing modes are represented by these three instructions?

```
LDA   ,X+
LDA   $19,X+
LDA   B,X+
```

Zero-offset auto-increment indexed, constant-offset auto-increment indexed, accumulator-offset auto-increment indexed.

* Read the following mnemonics:

* STA ,X

Store A, zero offset to X.

* STA $10,X

Store A, constant offset $10 to X.

* STA B,X

Store A, accumulator B offset to X.

* STA ,X+

Store A, zero offset to X, increment X by one.

* STA ,-X

Decrement X by one, store A, zero offset to X.

* STA $9AB,-X

Decrement X by one, store A, constant offset of $9AB to X.

To store A at memory indexed by X, and then to automatically increment X by one byte, use the zero-offset auto-increment/decrement indexed mode. It is written simply:

```
Mnemonic:              STA     ,X+
Read:
     Store A, zero offset to X,
     increment X by one byte
Process:
1. Store A in memory location (  X  )
2. Make X = X + 1
3. Change N and Z flags, reset V flag
4. Go on to next instruction
```

To store A at memory indexed by X, after automatically decrementing X by one byte, use the zero-offset auto-increment/decrement indexed mode. It is also simpler to write than to describe:

```
Mnemonic:              STA     ,-X
Read:
    Decrement X by one byte, store A,
    zero offset to X
Process:
1. Make X = X - 1
2. Store A in memory location (  X  )
3. Change N and Z flags, reset V flag
4. Go on to next instruction
```

To store A at memory indexed by X plus an offset of **$9AB** bytes, and following that to automatically increment X by one byte, use the constant-offset auto-increment/decrement indexed mode. It is written:

```
Mnemonic:              STA     $9AB,X+
Read:
Store A, $9AB constant offset to X,
 increment X by one byte
Process:
1. Calculate X + $9AB
2. Store A in memory location (X + $9AB)
3. Make X = X + 1
4. Change N and Z flags, reset V flag
5. Go on to next instruction
```

To store A at memory indexed by X plus an offset of **$9AB** bytes, after decrementing X by one byte, use the constant-offset auto-increment/decrement indexed mode. It is written:

```
Mnemonic:                STA     $9AB,-X
Read:
Decrement X by one byte, store A,
$9AB constant offset to X
Process:
1. Make X = X - 1
2. Calculate X + $9AB
3. Store A in memory location (X + $9AB)
4. Change N and Z flags, reset V flag
5. Go on to next instruction
```

To store A at memory indexed by X plus an offset of whatever value is in the B accumulator, and to automatically increment X by two bytes, use the accumulator-offset auto-increment/decrement mode. It is written:

```
Mnemonic:                STA     B,X++
Read:
Store A, accumulator B offset to X,
increment X by 2 bytes
Process:
1. Calculate X + B
2. Store A in memory location (X + B)
3. Make X = X + 2
4. Change N and Z flags, reset V flag
5. Go on to next instruction
```

To store A at memory indexed by X plus an offset of whatever value is in the B accumulator, after automatically decrementing X by two bytes, use the accumulator-offset auto-increment/decrement mode. It looks like this:

```
Mnemonic:                STA     B,--X
Read:
Decrement X by 2 bytes, store A,
accumulator B offset to X
Process:
1. Make X = X - 2
2. Calculate X + B
3. Store A in memory location (X + B)
4. Change N and Z flags, reset V flag
5. Go on to next instruction
```

---

* STA B,X++

Store A, accumulator B offset to X, increment X by two.

* STA B,--X

Decrement X by two bytes. store A, accumulator B offset to X.

* What addressing modes are represented by these five instructions:

```
CLRB
LDB  #$12
LDB  $1234
LDB  <$34
LDB  $12,X
```

Inherent, immediate, extended. direct, indexed (constant-offset indexed).

* BRA means branch always. What kind of addressing does BRA $FD indicate?

Relative addressing.

* Relative addressing is relative to what?

The program counter (PC).

* What does the program counter (PC) indicate?

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

* What is the relative position of the PC?

Since "relative" means relative to the position of the PC, then the PC is always relative position 00.

* What determines a number's sign (positive or negative) in binary?

The sign bit.

* Which bit is the sign bit?

The leftmost bit.

* When the leftmost bit is a zero, what is the number's sign?

Positive.

* When the leftmost bit is a one, what is the number's sign?

Negative.

* What is the binary equivalent of $C7?

$C7 is binary 11000111.

* Is $C7 positive or negative? Why?

Negative, because the leftmost bit (the sign bit) is a one.

* What is $7C in binary. Is $7C positive or negative? Why?

$7C is 01111100. It is positive, because the leftmost bit (the sign bit) is a zero.

* What is the relative position of the byte in memory directly preceding the PC?

Relative position -1, or $FF.

* What is the relative position of the byte in memory directly following the PC?

Relative position 01.

* Why does $FF mean -1?

Because the leftmost bit (the sign bit) is a one.

* What does BRA mean?

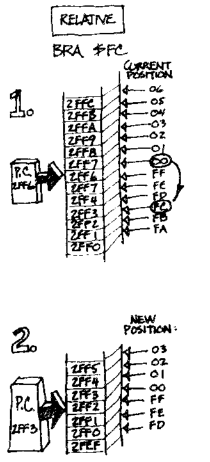
Branch always.

* The opcode for BRA is $20. When the instruction $20 FE is executed, what are the relative positions of opcode BRA and operand $FE?

Operand $FE is at relative position $FF (-1) and opcode BRA is at relative position $FE (-2).

As you can see, even storing the accumulator to memory indexed by X can be done a number of ways. A complete list would include six more variants that I haven't described; you'll have a chance to try these modes in your workbook. This is a good time to do that if you would like, or just to take a break and review.

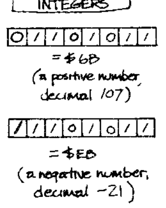
If you've been reviewing this lesson, you probably have an idea that indexed addressing is very flexible and not nearly so difficult as the jargon suggests. And, if you've had a glance at your MC6809E data booklet, then you know there's quite a bit more to the subtlety of indexed addressing. Even so, I would like to leave that topic for now and turn to Relative Addressing.



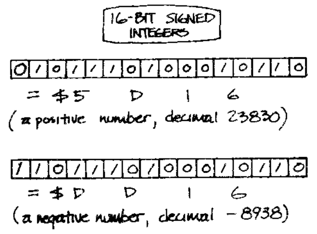
Relative Addressing is a good term, one of the best pieces of jargon you'll encounter. When Relative Addressing is employed, the data needed to complete an instruction is found at a location in memory *relative* to the present position of the Program Counter. Specifically, Relative Addressing is used to identify places in memory to which the program itself will branch.

To use Relative Addressing, though, you have to know about signs. I've not mentioned negative numbers in conjunction with binary or hexadecimal notation, and that's because the representation used is different from that in the decimal system. In the decimal system, of course, a negative 10 is simply written with a minus sign, -10. Computer binary numbers are called signed numbers, because the sign for positive or negative aspect is in fact a part of the number itself. That's simpler than it sounds. Where the sign of a number is unimportant, all the binary digits have the same meaning, as you've experienced so far. However, certain programming conditions — Relative Addressing is just one of them — need to know not only the length of a branch, but also which direction the branch goes. That is, how far will the program counter move, and will it move forward or backward, relative to the current position in the program?

To sign a number in binary, a unique procedure is used. If the most signficant bit — that is, the leftmost bit — of the number in question is zero, then the number is positive; if the most significant bit is one, then the number is considered negative. Remember, the sign bit is ignored except when it is needed.


8-BIT SIGNED INTEGERS
0 1 1 0 1 0 1 1
= $6B
(a positive number, decimal 107)
1 1 1 0 1 0 1 1
= $EB
(a negative number, decimal -21)

You *have* used a signed number in the programming you've done this far (in fact, a negative signed number), but you probably haven't noticed. Think back to the program which moved information from memory to the screen; there was an instruction that read "Branch if Not Equal" to a part of the program labeled "LOOP". At the time, I hustled you past that point, explaining only about the condition code register, how that branch would take place if the zero flag was not set, and that this was sort of like a BASIC GOTO. I didn't mention anything about the operand of that branch instruction.



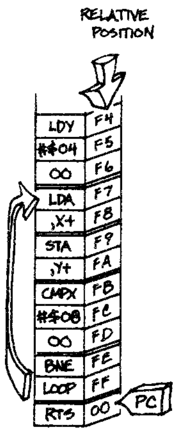
Turn to your documentation. That program is printed with this text; this time, though, the hex code appears with it.

```
4000                  00100          ORG     $4000
4000 8E    0800       00110          LDX     #$0800
4003 108E  0400       00120          LDY     #$0400
4007 A6    80         00130 LOOP     LDA     ,X+
4009 A7    A0         00140          STA     ,Y+
400B 8C    0800       00150          CMPX    #$0800
400E 26    F7         00160          BNE     LOOP
4010 39               00170          RTS
           0000       00180          END
00000 TOTAL ERRORS
LOOP     4007
```

It should look familiar. Incidentally, the load immediate instructions in lines 110 and 120, and the zero-offset auto-increment/decrement indexed instructions in lines 130 and 140 should be particularly understandable this time 'round. But my interest is line 160. There's that Branch if Not Equal to LOOP. Hex **$26** is the opcode for Branch if Not Equal. **$F7** is the operand. How does **$F7** describe a program branch?

The answer is to write it in binary. **$F7** translates into **1111 0111**. The most-significant bit, bit 7, is a *one*, meaning (for Relative Addressing purposes), this is a negative number. This is a *backwards* branch. Translated into a decimal number, this is –9. If you don't have a decimal/hex programmer's calculator, you can refer to the chart at the end of the documentation, or just count backwards . . . **$00** is 0. **$FF** is –1. **$FE** is –2. **$FD** is –3. **$FC** is –4. **$FB** is –5. **$FA** is –6. **$F9** is –7. **$F8** is –8. **$F7** is –9. There it is. –9.

The backwards branch is made from the Program Counter's present position. Recall that several lessons ago I said that the Program Counter points to the next instruction to be executed. Look at the listing again. The next instruction is in line 170, Return from Subroutine. The Program Counter is pointing to **RTS** when the Branch on Not Equal instruction is in progress. This is the starting point, relative position **$00**. You'll be counting backwards through the second and third columns, containing the hexadecimal opcodes and operands. Count backwards in the hex data with your finger. **$00** points to Return from Subroutine, hex code **$39**. Now start counting. **$FF**, **$FE** . . . that's the beginning of the Branch on Not Equal instruction. **$FD**, **$FC**, **$FB** . . . that puts you at the beginning of the Compare X opcode. **$FA**, **$F9** . . . that's the Store A command. **$F8**, **$F7** . . . and there it is, the beginning of the Load A instruction, right on the line with the label "LOOP".



Try it again, just to be certain. Start with the instruction Return from Subroutine as relative position **$00**, and count backwards through the bytes of data. **$FF**, **$FE**. **$FD**, **$FC**, **$FB**. **$FA**, **$F9**. **$F8**, **$F7**. The relative branch brings you back to the label "LOOP".

There's another way to do this, actually the way that the 6809 itself does it. The 6809 adds the relative branch operand to the address pointed to by the Program Counter.

* When $20 FE is executed, what happens to the program counter?

It is moved to relative position $FE, that is, -2.

* What is found at relative position $FE (-2)?

The opcode BRA.

* What is the complete instruction found at relative position $FE?

Branch always to relative position -2, BRA $FE, or $20 FE.

* Summarize what happens when the program encounters the instruction BRA $FE.

The program branches to relative position $FE, that is, back to the instruction BRA $FE. This is an endless loop.

* What is inherent addressing?

Inherent addressing is an addressing mode in which the information needed to complete an instruction is part of the instruction itself.

* What is register addressing?

An addressing mode in which the information needed by the program is moved from one register to another.

* What is immediate addressing?

An addressing mode in which the data to be used is found at the address immediately following the instruction itself, in program order.

* What is extended addressing?

An addressing mode in which the two bytes following the opcode are the address of the data to be used to complete the instruction.

* What is direct addressing?

An addressing mode where the direct page register and the value following the opcode are combined to form an address. At that address is found the data to complete the instruction.

* What is indexed addressing?

An addressing mode in which a 16-bit register and an offset are combined to produce a 16-bit result. The 16-bit result is used as an address; the data is found at that address.

* What is relative addressing?

An addressing mode where the operand is an offset relative to the current position of the program counter. Depending on the conditions of the relative instruction, the program will branch to this relative position.

* What is the term for how a machine language program gets its information?

An addressing mode.

If the relative branch is positive (bit 7 is zero), then that result becomes the address of the next instruction the processor will execute. However, if the relative branch value is negative, the 6809 decrements the most-signicant byte of the address, and uses that as the address of the next instruction. In this case, the Program Counter reads **$4010** and the relative branch is **$F7**.

```
        $4Ø1Ø
plus      $F7
       ------
is      $41Ø7
```

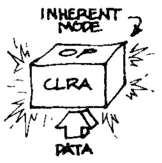
But **$F7** is negative, so the most signficant byte of the address (**$41**) is decremented to **$40**. The result is **$4007**. Glance at the listing. **$4007** is the address where you will find the label "LOOP".

The 6809 has two kinds of Relative Addressing — long and short. So far I've been describing short addressing. In short addressing, one byte is used to carry the program 127 addresses forward or 128 addresses backward. Long Relative Addressing uses two bytes, but the principle is the same. If the most-significant bit is zero, the long branch is positive; if the most-signficant bit is one, the long branch is negative. There are two major differences between the short and long branch. In the one-byte short branch, bit 7 is the most-significant bit; in the two-byte long branch, bit 15 is the most-significant bit. Also, the short branch can move only 127 addresses forward or 128 addresses backward; the long branch can move 32,767 addresses forward or 32,768 addresses backward in memory — that is, through the entire memory map of the computer. Long branches offer position independent programming. Remember the term "position independent"; I'll be talking quite a bit about that later.

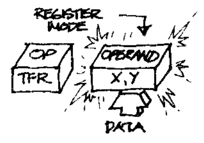
Relative Addressing, then, is unique in that the operand does not provide either an immediate value or a specific address to the processor. Rather, it provides a value which can be used to calculate a specific address in relationship to the present position in the program.

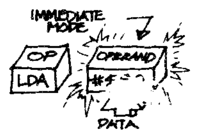
Time to summarize. There are seven major ways your program can obtain the information it needs. These are called the addressing modes.

1. The information can be implied by the instruction itself. This is Inherent Addressing. **CLRA** (Clear A Accumulator) is an example of Inherent Addressing.

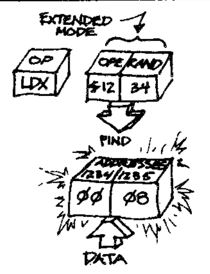


2. The information can deal with internal 6809 registers. This is Register Addressing. **TFR X,Y** (Transfer X Register to Y Register) is an example of Register Addressing.

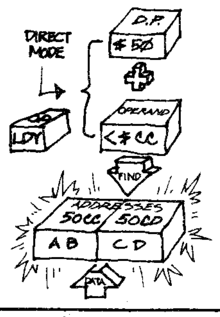




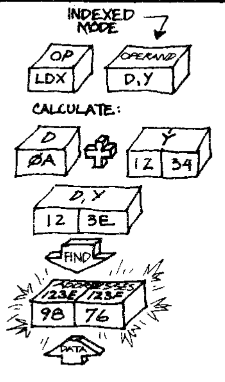
3. The information can be present immediately following the instruction itself. This is Immediate Addressing. **LDA #$80** (Load A Accumulator with the value **$80**) is an example of Immediate Addressing.



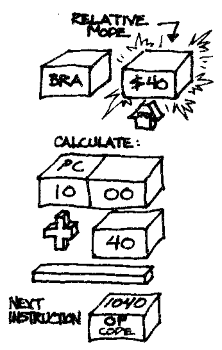
4. The information can take the form of a memory address where data can be found. This is Extended Addressing.
LDX $1234 (Load X Register with the information at Address **$1234**) is an example of Extended Addressing.



5. The information can take the form of the least-signficant half of a memory address. This can be combined with the value of the Direct Page register to locate the information in memory. This is Direct Addressing. If the Direct Page register is **$50**, then **LDY <$CC** (Load Y with the information at addresses **$50CC** and **$50CD**) is an example of Direct Addressing.



6. The information can take the form of a register value, which, together with an optional offset, identifies a memory address where data can be found. This is Indexed Addressing. **LDX D,Y** (Load X with the information at Address Y plus offset D) is an example of Indexed Addressing.



7. The information can take the form of a value to add to the Program Counter to determine a new position for the Program Counter. This is Relative Addressing. **BRA $40** (Branch Always to Program Counter plus **$40**) is an example of Relative Addressing.

Each of these modes is unique, and each contributes to the speed and economy of the 6809 processor. Please review this lesson and read pages 15 through 17 of your MC6809E data booklet. I haven't yet discussed what are called the Indirect Addressing Modes; if, when you read the data booklet, the Indirect modes make sense, then you're doing well indeed. If they're not clear to you, don't worry; that's for later. Once again, please review all the addressing modes before moving to the next lesson.

8.

The architecture of the 6809 processor has up to this point been described piecemeal. Now I'd like to summarize the 6809 processor's architecture, making the description a bit more formal. Please look once again at Figure 4 on page 5 of the MC6809E data booklet.

The PROGRAM COUNTER keeps the machine language program running in order. The Program Counter register contains the 16-bit address of the next instruction to be performed in the program sequence. The Program Counter can be changed directly by the programmer, by jumps and branches within the program, by subroutines, and by stack operations. The Program Counter is one of the POINTER registers.

The two ACCUMULATORS perform simple arithmetic. The A and B Accumulators are each one byte (8 bits) in size. For some operations, the two Accumulators are concatenated, creating a single, 16-bit Accumulator. When A and B are used together as one 16-bit Accumulator, they are collectively called the D Accumulator.

There are two INDEX registers, each 16 bits in size, which can be used to identify memory locations. Although by themselves they are very limited in capability, the Index Registers X and Y can be used, together with various calculated offsets, to load or store data anywhere in memory. To increase their flexibility, the X and Y registers can also be automatically incremented or decremented during the course of a machine language instruction. The Index Registers are also POINTER registers.

There are also two STACK POINTER registers, each 16 bits in size, and each with a different purpose. The User Stack Pointer, the U register, is only controlled by the programmer by pushing and pulling information. This program control allows information to be transferred easily between portions of a program. The Hardware Stack Pointer, the S register, is also used for pushing and pulling information, but is used automatically by the processor to save Program Counter address information during subroutine calls.

After two lessons of heavy abstract learning, you're back with some familiar concepts and practice. At the end of this lesson, you'll be a third of the way through the course — ready to jump into the programming details of the computer. So give this lesson lots of time, and practice each instruction until it's comfortable ... whether or not you know what it's good for!

* Name the 16-bit registers of the 6809.

X and Y, program counter PC, S and U stacks, and the D accumulator.

* Name the 8-bit registers of the 6809.

A and B accumulators, condition code register CC, and direct page register DP.

* What is the purpose of the program counter, PC?

It keeps the machine language program running in order.

* What value does the program counter hold?

The 16-bit address of the next instruction to be performed.

* What is the purpose of the A and B registers?

To perform simple arithmetic.

* What is the D register?

The concatenation of the 8-bit A and B registers into a single 16-bit register.

* What are the X and Y registers?

Index registers.

* How are index registers most often used?

To identify memory locations.

* What are the S and U registers?

The S register is the hardware stack pointer, and the U register is the user stack pointer.

* How are the S and U registers different?

The U register is reserved for pushing and pulling program information; the S register is used for pushing and pulling as well as for subroutine calls.

* What is the purpose of the condition code register?

The condition code register provides information about the most recent instruction executed by the processor.

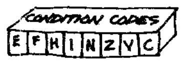
* What is another name for the condition code register?

The flags.

* What does the direct page register store?

The direct page register stores the most-significant half of an address.

The CONDITION CODE register, or flags, is an 8-bit register wherein each bit has a meaning and can be used to make simple judgments (such as greater than, less than, equal to, positive, negative, carry, borrow, etc.) within a program. The Condition Code Register is automatically modified by the results of machine language instructions, or can be changed directly by the programmer.

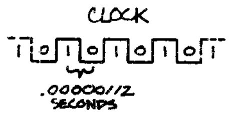


The DIRECT PAGE register, 8 bits wide, is given the most-significant byte of an address. During Direct Addressing, the Direct Page register provides this half of the address, and the program provides the least-significant half of the address. The result is a complete address which can be used to access data in memory.

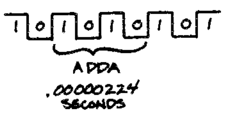
Please read pages 4 and 5, and the first portion of page 6, in the MC6809E data booklet. This section describes the architecture of the 6809 processor. Return to the tape when you have completed the reading.

I wanted you to read that to get a firm idea of the 6809's innards. The next step is getting a handle on some of the 6809's instructions, and for this I'll return to your computer and to a BASIC program. Turn back to your MC6809E data booklet, pages 30 and 31. These pages contain an alphabetical list of the 6809 processor instructions, and are chock full of information.

In the first column is the generalized mnemonic, such as ADD, DECrement, LoaD, etc. The second column shows the specific editor/assembler forms it can take, meaning how to indicate the registers or memory the instruction can use. The next block of information is entitled "Addressing Modes", and provides detailed information on each instruction in that mode, its specific opcode in hexadecimal, the number of bytes the instruction requires for completion, and the number of clock cycles needed for the process.



I haven't mentioned clock cycles before; they are vital to understand when your programming begins to get sophisticated. You've probably heard that the Color Computer runs at .89 MHz. Actually, the precise figure for the computer's speed is .894886 MHz, that is, 894,886 clock pulses per second. Any action taken by the 6809 processor is triggered by one clock pulse; at 894,886 clock pulses per second, that means that the Color Computer's 6809 can't do anything in a shorter time than .00000112 seconds. .00000112 seconds is 1.12 microseconds, slightly longer than a millionth of a second. Knowing this timing is important when writing programs that transfer information properly to the printer port, the RS-232, the cassette, the disk and other devices. Later, when you begin producing audio from your computer, knowing the clock cycles required for each 6809 instruction will be essential.



Back to the booklet, page 30. The description column, toward the right, gives in abbreviated notation the function of each machine language instruction. The symbols and abbreviations are explained at the bottom of the page; glance at the ADD instruction. You will discover that addition using the A Accumulator, mnemonic **ADDA**, is valid in four addressing modes. In the immediate mode, for example, you find that the hexadecimal opcode for this instruction is **8B**, that the complete instruction consists of 2 bytes, and that it takes 2 clock cycles (that is, 2.24 microseconds) to execute. The description column says that the result of A Accumulator plus a value from memory is transferred into the A Accumulator.

The last group of columns provides detailed information about the condition code register — how each flag is affected by the instruction. In the case of the **ADDA** instruction, all five condition code bits are affected (either set or reset) by the results of that command.

These are pretty dense pages. In order to simplify them a little, I've put together a program in BASIC. It's fairly long, so while it's loading, start to get familiar with pages 30 and 31. By the way, there are two program dumps on the tape, just to make certain you've got a good one.

Program #13, 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/0 error occurs, rewind to the start of the program and try again. For severe loading problems, see the Appendix.

```
1 CLS
2 PRINTSTRING$(32,45);
3 PRINTSTRING$(5,191)" INSTRUCTION EXAMPLES "STRING$(5,191);
4 PRINTSTRING$(32,45);
5 PRINT"(1)  ADD                    (ADD)
6 PRINT"(2)  AND            (LOGICAL AND)
7 PRINT"(3)  ASL/ASR (ARITHMETIC SHIFT)
8 PRINT"(4)  COM             (COMPLEMENT)
9 PRINT"(5)  DEC              (DECREMENT)
10 PRINT"(6)  EOR          (EXCLUSIVE OR)
11 PRINT"(7)  INC              (INCREMENT)
12 PRINT"(8)  LSL/LSR     (LOGICAL SHIFT)
13 PRINT"(9)  NEG               (NEGATIVE)
14 PRINT"(A)  OR              (LOGICAL OR)
15 PRINT"(B)  ROL/ROR             (ROTATE)
16 PRINT"(C)  SUB               (SUBTRACT)
17 PRINTCHR$(191)"  TOUCH 1 - C TO DEMONSTRATE  ";:POKE1535,191
18 A$=INKEY$:IFA$=""THEN18
19 A=ASC(A$):A=A-48:IFA<1 OR A>19 THEN18
20 ONA GOSUB23,37,50,76,86,97,112,122,138,18,18,18,18,18,18,18,1
49,163,192
21 RUN
22 GOTO22
23 CLS:NF=0:ZF=0:CF=0
24 PRINT"------> ADD TWO NUMBERS <------"
25 GOSUB225:IFQQ=1THEN23
26 INPUT"VALUE TO ADD FROM MEMORY OR FROMOTHER REGISTER (HEX)";A
2$:A$=A2$
27 Q=0:GOSUB210:IFQ=1THEN23
28 X=A:A2=A:GOSUB212:Q2$=Q$
29 X=A1+A2:A3=X
30 IFX>255THENX=X-256:CF=1:A3=X
31 IFX=0THENZF=1
```

*** How is the direct page register used?**

In the direct addressing mode, the register is used to create a complete address.

*** What is an addressing mode?**

How the machine language program gets its information.

*** What do you call the verbal description of a processor command?**

A mnemonic.

*** What is the proper name for a processor command?**

An opcode.

*** What is the clock speed of the Color Computer?**

.89 MHz (.894886 MHz or 894,886 pulses per second).

*** How long is one clock pulse on the Color Computer?**

Approximately .00000112 seconds or 1.12 microseconds. (More accurately, 1.11746 microseconds).

*** How long is a microsecond?**

One millionth of a second.

*** The MUL (multiply) instruction takes 11 clock cycles. How long is this on the Color Computer?**

11 times 1.11746 microseconds, or 12.29206 microseconds.

*** LDA immediate and LDB immediate each take 2 clock cycles. How long is each instruction on the Color Computer?**

2 times 1.11746 microseconds, or 2.23492 microseconds.

* STD extended takes 6 clock cycles. How long is this?

6 times 1.11746 microseconds, or 6.70476 microseconds.

* MULtiply is A times B, with the result in D. If a multiplication program consists of LDA and LDB immediate (each 2 clock cycles), MULtiply (11 clock cycles), and STD extended (6 clock cycles), how long is this?

(2+2+11+6) times 1.11746 microseconds, or 23.46666 microseconds).

* At 23.46666 microseconds per multiplication program, how many complete multiplication programs can the Color Computer do in one second?

The Color Computer can perform 42,613 multiplication programs per second.

* What is the purpose of the condition code register?

The condition code register provides information about the most recent instruction executed by the processor.

* In the following exercises, give the results of the instruction, where the result is found, and the effect on the three flags N, Z and C (condition codes negative, zero and carry). For example, the problem: A contains $41. Execute ADDA #$CC. The answer: A contains $0D. Carry flag set. Zero and negative flags reset.

* Problem: A contains $04. Execute ADDA #$FB.

Answer: A contains $FF. Negative flag set. Zero and carry flags reset.

```
32 GOSUB212:Q3$=Q$
33 PRINT
34 PRINTTAB(5)Q1$"     "A1$:PRINTTAB(5)Q2$"     "A2$:PRINTTAB(5)STRI
NG$(20,45):PRINTTAB(5)Q3$"     ";:IFA3<16THENPRINT"0"+HEX$(A3) ELS
EPRINTHEX$(A3)
35 GOSUB224
36 GOSUB222:RETURN
37 CLS:NF=0:ZF=0:CF=0
38 PRINT"--> LOGICAL AND  TWO NUMBERS <--";
39 GOSUB225:IFQQ=1THEN37
40 INPUT"VALUE TO AND FROM MEMORY OR FROMOTHER REGISTER (HEX)";A
2$:A$=A2$
41 Q=0:GOSUB210:IFQ=1THEN37
42 X=A:A2=A:GOSUB212:Q2$=Q$
43 X= A1 AND A2 : A3=X
44 IFX=0THENZF=1
45 GOSUB212:Q3$=Q$
46 PRINT
47 PRINTTAB(5)Q1$"     "A1$:PRINTTAB(5)Q2$"     "A2$:PRINTTAB(5)STRI
NG$(20,45):PRINTTAB(5)Q3$"     ";:IFA3<16THENPRINT"0"+HEX$(A3)ELSE
PRINTHEX$(A3)
48 GOSUB224
49 GOSUB222:RETURN
50 CLS:NF=0:ZF=0:CF=0
51 PRINT"ARITHMETIC SHIFT LEFT OR RIGHT":PRINT"TOUCH L OR R"
52 A$=INKEY$:IFA$="L"ORA$="l"THEN53ELSEIFA$="R"ORA$="r"THEN63ELS
E52
53 CLS:PRINT"---> ARITHMETIC SHIFT LEFT <---"
54 GOSUB225:IFQQ=1THEN53
55 X=A*2:A2=X
56 IFX>255THENX=X-256:CF=1:A2=X
57 IFX=0THENZF=1
58 GOSUB212:Q2$=Q$
59 PRINT
60 PRINTTAB(5)Q1$"     "A1$:PRINTTAB(5)"<---- SHIFT ----":PRINTTAB
(5)Q2$"     ";:IFA2<16THENPRINT"0"+HEX$(A2) ELSEPRINTHEX$(A2)
61 GOSUB224
62 GOSUB222:RETURN
63 CLS:PRINT"---> ARITHMETIC SHIFT RIGHT <---";
64 GOSUB225:IFQQ=1THEN63
65 IFA>127THENNF=1
66 X=FIX(A/2):IFX>63THENX=X OR128:A2=X:ELSEA2=X
67 IF(A/2)<>FIX(A/2)THENCF=1
68 IFX=0THENZF=1
69 GOSUB212:Q2$=Q$
70 PRINT
71 PRINTTAB(5)Q1$"     "A1$:PRINTTAB(5)"---- SHIFT ---->":PRINTTAB
(5)Q2$"     ";:IFA2<16THENPRINT"0"+HEX$(A2) ELSEPRINTHEX$(A2)
72 GOSUB224
73 IFNF=1 THEN PRINT:PRINT"NOTE BIT 7; SEE DATA BOOKLET.":GOTO75
74 IFCF=1 AND NF=0 THEN PRINT:PRINT"NOTE CARRY FLAG; SEE DATA BO
OK."
75 GOSUB222:RETURN
76 CLS:NF=0:ZF=0:CF=1
77 PRINT"----> COMPLEMENT A NUMBER <----"
78 GOSUB225:IFQQ=1THEN76
79 X=NOTA AND 255:A2=X:GOSUB212:Q2$=Q$
80 IFX=0THENZF=1
81 IFX>127THENNF=1
82 PRINT:PRINTTAB(5)Q1$"     "A1$:PRINTTAB(5)"** COMPLEMENT **":PR
INTTAB(5)Q2$"     ";:IFA2<16THENPRINT"0"+HEX$(A2) ELSEPRINTHEX$(A2
)
83 GOSUB224
84 PRINT:PRINT"NOTE CARRY FLAG; SEE DATA BOOK."
85 GOSUB222:RETURN
86 CLS:NF=0:ZF=0:CF=0
87 PRINT"----> DECREMENT A NUMBER <----"
88 GOSUB225:IFQQ=1THEN86
89 X=A-1:A2=X:IFX<0THENX=255:A2=X:NF=1
90 IFX=0THENZF=1
91 IFX>127THENNF=1
92 GOSUB212:Q2$=Q$
93 PRINT
94 PRINTTAB(5)Q1$"     "A1$:PRINTTAB(5)"** DECREMENT **":PRINTTAB(
5)Q2$"     ";:IFA2<16THENPRINT"0"+HEX$(A2) ELSEPRINTHEX$(A2)
95 GOSUB224
96 GOSUB222:RETURN
97 CLS:NF=0:ZF=0:CF=0
98 PRINT"LOGICAL EXCLUSIVE-OR TWO NUMBERS";
99 GOSUB225:IFQQ=1THEN97
100 INPUT"VALUE TO EXCLUSIVE-OR, TAKEN     FROM MEMORY OR FROM AN
OTHER     REGISTER";A2$:A$=A2$
101 QQ=0:GOSUB210:IFQQ=1THEN97
```

```
102 X=A:A2=A:GOSUB212:Q2$=Q$
103 X=(A1 AND NOT(A2)) OR (NOT(A1) AND A2):A3=X
104 IFX=0THENZF=1
105 IFX)127THENNF=1
106 GOSUB212:Q3$=Q$
107 PRINT
108 PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)Q2$"    "A2$:PRINTTAB(5)STR
ING$(20,45):PRINTTAB(5)Q3$"    ";:IFA3(16THENPRINT"0"+HEX$(A3) EL
SEPRINTHEX$(A3)
109 GOSUB224
110 GOSUB222:RETURN
111 RETURN
112 CLS:NF=0:ZF=0:CF=0
113 PRINT"----) INCREMENT A NUMBER (----"
114 GOSUB225:IFQQ=1THEN112
115 X=A+1:A2=X:IFX)255THENX=0:A2=X:ZF=1:NF=0
116 IFX)127THENNF=1
117 GOSUB212:Q2$=Q$
118 PRINT
119 PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)"** INCREMENT **":PRINTTAB
(5)Q2$"    ";:IFA2(16THENPRINT"0"+HEX$(A2) ELSEPRINTHEX$(A2)
120 GOSUB224
121 GOSUB222:RETURN
122 CLS:NF=0:ZF=0:CF=0
123 PRINT") LOGICAL SHIFT LEFT OR RIGHT ("
124 PRINT"TOUCH L OR R"
125 A$=INKEY$:IFA$="L"ORA$="l"THEN126ELSEIFA$="R"ORA$="r"THEN129
ELSE125
126 CLS:PRINT"----) LOGICAL SHIFT LEFT (----"
127 GOSUB225:IFQQ=1THEN126
128 GOTO55
129 CLS:PRINT"----) LOGICAL SHIFT RIGHT (----"
130 GOSUB225:IFQQ=1THEN129
131 X=FIX(A/2):A2=X:IFA/2()FIX(A/2)THENCF=1
132 IFX=0THENZF=1
133 GOSUB212:Q2$=Q$
134 PRINT
135 PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)"---- SHIFT ----)":PRINTTA
B(5)Q2$"    ";:IFA2(16THENPRINT"0"+HEX$(A2) ELSEPRINTHEX$(A2)
136 GOSUB224
137 GOSUB222:RETURN
138 CLS:NF=0:ZF=0:CF=0
139 PRINT"-------) NEGATE A NUMBER (-------"
140 GOSUB225:IFQQ=1THEN138
141 REM
142 REM
143 X=(NOTA AND 255)+1:A2=X:GOSUB212:Q2$=Q$
144 IFX=0THENZF=1:CF=1
145 IFX)127THENNF=1
146 PRINT:PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)"** NEGATIVE **":PRI
NTTAB(5)Q2$"    ";:IFA2(16THENPRINT"0"+HEX$(A2) ELSEPRINTHEX$(A2)
147 GOSUB224
148 GOSUB222:RETURN
149 CLS:NF=0:ZF=0:CF=0
150 PRINT"---) LOGICAL OR TWO NUMBERS (---";
151 GOSUB225:IFQQ=1THEN149
152 INPUT"VALUE TO OR FROM MEMORY OR FROM ANOTHER REGISTER (HEX)
";A2$:A$=A2$
153 QQ=0:GOSUB210:IFQQ=1THEN149
154 X=A:A2=A:GOSUB212:Q2$=Q$
155 X=A1 OR A2 : A3=X
156 IFX=0THENZF=1
157 IFX)127THENNF=1
158 GOSUB212:Q3$=Q$
159 PRINT
160 PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)Q2$"    "A2$:PRINTTAB(5)STR
ING$(20,45):PRINTTAB(5)Q3$"    ";:IFA3(16THENPRINT"0"+HEX$(A3) EL
SEPRINTHEX$(A3)
161 GOSUB224
162 GOSUB222:RETURN
163 CLS:NF=0:ZF=0:CF=0
164 PRINT"----) ROTATE LEFT OR RIGHT (----";
165 PRINT"TOUCH L OR R"
166 A$=INKEY$:IFA$="L"ORA$="l"THEN167ELSEIFA$="R"ORA$="r"THEN180
ELSE166
167 CLS:PRINT"STATE OF CARRY FLAG? (0 OR 1) ";
168 A$=INKEY$:IFA$="0" OR A$="1"THENPRINTA$:CF=VAL(A$):ELSE168
169 GOSUB225:IFQQ=1THEN167
170 X=A*2:A2=X
171 IFX(256THENX=X ORCF:A2=X:CF=0:GOTO173:ELSE172
172 X=X-256:X=X ORCF:CF=1:A2=X
173 IFX=0THENZF=1
174 IFX)127THENNF=1
```

* Problem: B contains $AA. Execute ANDB #$55.

Answer: B contains $00. Zero flag set. Negative flag reset. Carry flag unaffected.

* Problem: B contains $AA. Execute ANDB #$5F.

Answer: B contains $0A. Zero and negative flags reset. Carry flag unaffcted.

* Problem: A contains $FF. Execute ORA #$A5.

Answer: A contains $FF. Negative flag set. Zero flag reset. Carry flag unaffected.

* Problem: A contains $AA. Execute ORA #$55.

Answer: A contains $FF. Negative flag set. Zero flag reset. Carry flag unaffected.

* Problem: A contains $00. Execute ORA #$00.

Answer: A contains $00. Zero flag set. Negative flag reset. Carry flag unaffected.

* Problem: B contains $F0. Execute ORB #$0F.

Answer: B contains $FF. Negative flag set. Zero flag reset. Carry flag unaffected.

* Problem: B contains $FF. Execute COMB.

Answer: B contains $00. Zero flag set. Negative flag reset. Carry flag always set by COM instruction.

* Problem: A contains $AA. Execute COMA.

Answer: A contains $55. Zero and negative flags reset. Carry flag always set by COM instruction.

*** Problem: A contains $04. Execute ADDA #$FC.**

Answer: A contains $00. Zero and carry flags set. Negative flag reset.

*** Problem: A contains $04. Execute ADDA #$FD.**

Answer: A contains $01. Carry flag set. Negative and zero flags reset.

*** Problem: B contains $00. Execute SUBB #$01.**

Answer: B contains $7F. All flags reset.

*** Problem: B contains $81. Execute SUBB #$01.**

Answer: B contains $80. Negative flag set. Zero and carry flags reset.

*** Problem: B contains $80. Execute SUBB #$80.**

Answer: B contains $00. Zero flag set. Negative and carry flags reset.

*** Problem: B contains $80. Execute SUBB #$81.**

Answer: B contains $FF. Negative and carry flags set. Zero flag reset.

*** Problem: A contains $FF. Execute ANDA #$FF.**

Answer: A contains $FF. Negative flag set. Zero flag reset. Carry flag unaffected.

*** Problem: A contains $FF. Execute ANDA #$A5.**

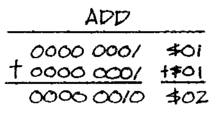
Answer: A contains $A5. Negative flag set. Zero flag reset. Carry flag unaffected.

```
175 GOSUB212:Q2$=Q$
176 PRINT
177 PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)"<--- ROTATE ---":PRINTTAB
(5)Q2$"    ";:IFA2<16THENPRINT"0"+HEX$(A2) ELSEPRINTHEX$(A2)
178 GOSUB224
179 GOSUB222:RETURN
180 CLS:PRINT"STATE OF CARRY FLAG? (0 OR 1) ";
181 A$=INKEY$:IFA$="0" OR A$="1"THENPRINTA$:CF=VAL(A$):ELSE181
182 GOSUB225:IFQQ=1THEN180
183 X=(FIX(A/2))OR(CF*128):A2=X
184 IFFIX(A/2)<>A/2THENCF=1ELSECF=0
185 IFX=0THENZF=1
186 IFX>127THENNF=1
187 GOSUB212:Q2$=Q$
188 PRINT
189 PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)"--- ROTATE --->":PRINTTAB
(5)Q2$"    ";:IFA2<16THENPRINT"0"+HEX$(A2) ELSEPRINTHEX$(A2)
190 GOSUB224
191 GOSUB222:RETURN
192 CLS:NF=0:ZF=0:CF=0
193 PRINT"----> SUBTRACT TWO NUMBERS <----";
194 GOSUB225:IFQQ=1THEN192
195 REM
196 REM
197 INPUT"VALUE TO SUBTRACT, TAKEN FROM   MEMORY OR OTHER REGIST
ER (HEX)";A2$:A$=A2$
198 QQ=0:GOSUB210:IFQQ=1THEN192
199 X=A:A2=A:GOSUB212:Q2$=Q$
200 X=A1-A2:A3=X
201 IFX<0THENCF=1:X=X+256:A3=X
202 IFX=0THENZF=1
203 IFX>127THENNF=1
204 GOSUB212:Q3$=Q$
205 PRINT
206 PRINTTAB(5)Q1$"    "A1$:PRINTTAB(5)Q2$"    "A2$:PRINTTAB(5)STR
ING$(20,45):PRINTTAB(5)Q3$"    ";:IFA3<16THENPRINT"0"+HEX$(A3) EL
SEPRINTHEX$(A3)
207 GOSUB224
208 GOSUB222:RETURN
209 FORN=1TO1000:NEXT:RETURN
210 A=VAL("&H"+A$):IFA<0 OR A>255 THEN PRINT"VALUE OUT OF RANGE"
:GOSUB209:QQ=1:RETURN
211 QQ=0:RETURN
212 C=INT(X/128):D=C*128
213 E=INT((X-D)/64):F=E*64
214 G=INT((X-D-F)/32):H=G*32
215 I=INT((X-D-F-H)/16):J=I*16
216 K=INT((X-D-F-H-J)/8):L=K*8
217 M=INT((X-D-F-H-J-L)/4):N=M*4
218 O=INT((X-D-F-H-J-L-N)/2):P=O*2
219 Q=INT(X-D-F-H-J-L-N-P)
220 Q$=STR$(C)+STR$(E)+STR$(G)+STR$(I)+STR$(K)+STR$(M)+STR$(O)+S
TR$(Q)
221 RETURN
222 PRINT"PRESS ENTER TO CONTINUE";
223 A$=INKEY$:IFA$<>CHR$(13)THEN223ELSERETURN
224 PRINT:PRINT"FLAGS:":PRINTTAB(7)"N  Z  C":PRINTTAB(6)C;ZF;CF:
PRINT:RETURN
225 INPUT"VALUE IN ACCUMULATOR (HEX)";A1$:A$=A1$
226 QQ=0:GOSUB210:IFQQ=1THENRETURN
227 X=A:A1=A:GOSUB212:Q1$=Q$
228 RETURN
```

RUN this program. On the screen are 12 common instructions selected from the total of 59 that the 6809 processor can execute. For your amusement, I've numbered them in hexadecimal.

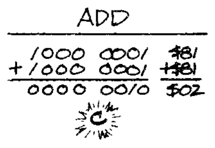
Some of the instructions will already be familiar, but I'd like you to get a detailed idea of how each one works, and what its results are. Here's how it goes. You will input all values in hexadecimal, but the display will be done in both hex and binary, so that the inner workings of each instruction will be evident. Although there are five flags, I've chosen only the most used ones (negative, zero, and carry) to display in these examples.

You can start with a familiar instruction, selection #1, ADD. Touch 1 on the keyboard.

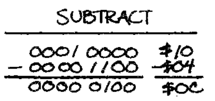


As you can see, for simplicity I'm making the assumption that the initial value will always be in an accumulator, and that all values will be 8 bits. Enter hex number **01** as the accumulator value. The second prompt appears. To the accumulator value will be added a value from memory or from another register in the processor. You'll add 1 to this. Type **$01** and hit <ENTER>.

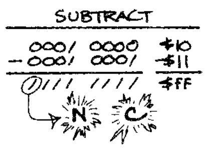
On your display are the two numbers being added, and the result, which is **$02**. All three are displayed in both binary and hexadecimal. The flags reveal three pieces of information: that the resulting number is not negative, it is not zero, and there was no carry generated by the calculation.



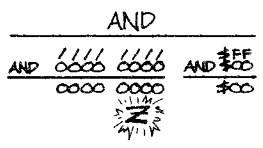
Hit <ENTER>, and touch 1 again. This time, enter hex **$81** into the accumulator. Add to this the value hex **$81**. The result is the same as before — **$02**. But the carry flag reveals something very important. It tells you that, although the apparent 8-bit result is **$02**, the addition actually produced a number larger than 8 bits.



Now some subtraction. Hit <ENTER>, and touch selection C. Enter **$10** into the accumulator. From this, subtract the value, say, **$04**. The result is **$0C**, a non-zero positive number, as the flags indicate. Hit <ENTER> and touch C again. Enter **$10** into the accumulator again, but this time subtract **$11**. The result is **$FF**. The flags tell the story. It is a negative number, and the carry/borrow flag shows that a borrow was required to complete it. That carry/borrow flag is vital to recognize.



Add and subtract are straightforward. Try each of them a few times at the end of this lesson. I'll go through the rest of the instructions in this group. When I'm done, you're on your own for a while. Let me walk over to the kitchen . . .



Hit <ENTER> to get back to the menu. You've tried ADD and SUBtract, so now touch 2 for logical AND. This is the first of the logical instructions (also called Boolean Algebra, but we'll forget *that* term). Logical AND requires both statements of a pair to be true for the result to be true. For example, this statement demonstrates logical AND: "If I break this plate AND Claire sees the broken plate, then she will scream at me". If either statement is not true — that is, if either I didn't break the plate, or if Claire didn't see the broken plate — then I'll get off. Here comes Claire now. <Breaking plate. "Look, you broke that plate! Arrrggh!!" etc.> Likewise, in binary arithmetic, both bits must be ones — that is, both bits must be true — for the result to be true. Enter **$FF** into the accumulator, and **$00** into memory. Each bit of the accumulator is ANDed with each corresponding bit in the operand. The results here are all zeros. The zero flag goes on.

*** Problem: A contains $EC. Execute COMA.**

Answer: A contains $13. Zero and negative flags reset. Carry flag always set by COM instruction.

*** Problem: A contains $47. Execute COMA.**

Answer: A contains $B8. Negative flag set. Zero flag reset. Carry flag always set by COM instruction.

*** Problem: B contains $0F. Execute COMB.**

Answer: B contains $F0. Negative flag set. Zero flag reset. Carry flag always set by COM instruction.

*** Problem: A contains $AA. Execute EORA #$00.**

Answer: A still contains $AA. Negative flag set. Zero flag reset. Carry flag not affected.

*** Problem: A contains $AA. Execute EORA #$AA.**

Answer: A contains $00. Zero flag set. Negative flag reset. Carry flag not affected.

*** Problem: A contains $AA. Execute EORA #$FF.**

Answer: A contains $55. Negative and zero flag reset. Carry flag not affected. Has effect of COMA except does not affect carry flag.

*** Problem: B contains $00. Execute EORB #$C8.**

Answer: B contains #$C8. Negative flag set. Zero flag reset. Carry flag not affected.

*** Problem:** A contains $0F. Execute ASLA.

Answer: A contains $1E. All flags reset.

*** Problem:** A contains $0F. Execute ASRA.

Answer: A contains $07. All flags reset.

*** Problem:** A contains $88. Execute ASLA.

Answer: A contains $10. Carry flag set as bit drops into "bucket". Negative and zero flags reset.

*** Problem:** A contains $88. Execute ASRA.

Answer: A contains $C4 (bit 7 duplicated at left). Negative flag set. Zero and carry flags reset.

*** Problem:** B contains $88. Carry flag is set. Execute ROLB.

Answer: B contains $11. Carry flag set. Zero and negative flags reset.

*** Problem:** B contains $88. Carry flag is set. Execute RORB.

Answer: B contains $C4. Negative flag is set. Carry and zero flags are reset.

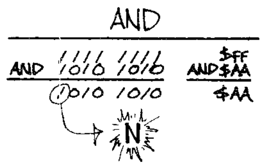
*** Problem:** A contains $02. Execute DECA.

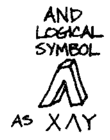
Answer: A contains $01. Negative and zero flags reset. Carry flag not affected.

*** Problem:** A contains $01. Execute DECA.

Answer: A contains $00. Zero flag set. Negative flag reset. Carry flag not affected.

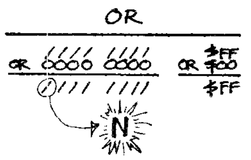
Hit <ENTER>, and touch 2. Again enter **$FF** into the accumulator. This time try **$AA** as the memory contents, and hit <ENTER>. Each bit of the pair is ANDed, and the result is **$AA**. The negative flag flips on.



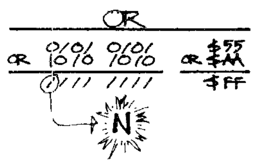


Contrast this with logical OR. Hit <ENTER>, and touch A. Logical OR states that if either or both of two conditions is true, then the result will be true. For example, this statement describes logical OR: "If I eat this pie OR I eat this ice cream, then I will be pleased." Binary numbers can't measure my level of pleasure, but they can report that <mouth full> I will be pleased if I eat either the pie or the ice cream, or if I eat both. Likewise, in binary arithmetic, if either number is a one — that is, if either number is true — the result will be true.

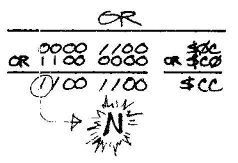
Enter **$FF** into the accumulator. Then enter **$00** as the operand. You can see two things: first, you find that since all bits in the accumulator are one, all bits in the result will be one, regardless of the operand; second, the negative flag flips on because bit 7 is a one. Hit <ENTER>, touch A, and put **$55** in the accumulator this time. As the operand, enter **$AA**. The numbers I chose here have alternating bits, just to demonstrate that neither byte need have bits in common — it is truly an either/or situation. Note the negative flag is on.



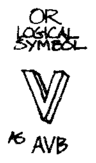
Just one more OR. Hit <ENTER>, and touch A. Put **$0C** in the accumulator, and **$C0** into the operand. In this example, you can see that where neither bit is true, zeros do result from the logical OR process. Again, you'll want to try examples of logical OR at the end of the lesson.



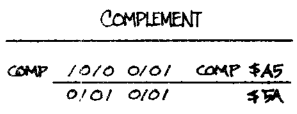
Move on to COMplement, selection 4. Hit <ENTER>, and touch 4. A number's complement is created by reversing all the binary digits in that number; it's the equivalent of a logical NOT. For example, enter **$A5**. All the zeros become ones, all the ones become zeros. The result after the complement is **$5A**. Hit <ENTER>, touch 4, and place **$FF** in the accumulator. The complement of **$FF** is **$00**. The zero flag flips on. Notice that in this instruction, the carry flag always turns on, regardless of the result, merely to indicate the completion of the instruction.



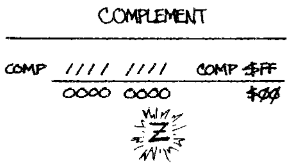
The logical Exclusive-OR instruction is next. This is a command used to "toggle" individual bits. You understand how logical AND, OR and NOT work. Just for review, two binary values ANDed together give a one result only if both values are one, as I mentioned above. Two binary values ORed together give a one result if either value is a one. Logical NOT simply flips one bits to zero, and zero bits to one, as in the COMplement statement you've already tried.



Logical Exclusive-OR gives a true result if either, but *not both*, of the premises are true. That's a little hard to analogize to real life, but since I'm still here in the kitchen, it might go something like this: "If I eat this full-course Chinese dinner Exclusive-OR if I eat this full-course Italian

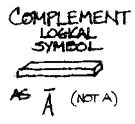


dinner, then I will be content." If I eat neither, I won't be content; if I eat both, I'll probably explode. Logical Exclusive-OR is the equivalent of the quantity (A and NOT B) ORed with the quantity (NOT A and B) . . . but that's not very revealing either.

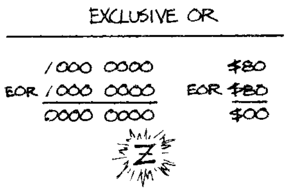


Try it this way: if two bits are different, the result will be one. If the bits are the same, the result will be zero. What makes this idea useful is that information can be "toggled" back and forth between numbers. Turn to the program for a visual example.

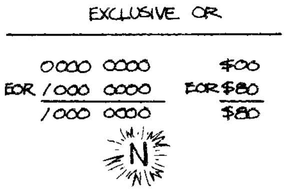
Hit <ENTER> and touch 6. You're going to toggle between, say, **$80** and **$00**. Enter **$80** into the accumulator. Pause here and think about hex **$80** and **$00**. In binary, **$80** is **1000 0000**, and **$00** is **0000 0000**. Only bit 7 is different here. You need to find a value that, when Ex-ORed with **1000 0000**, gives **0000 0000**. Recall how Exclusive-OR works: to get a zero result, the two bits being Ex-ORed must be the same. That suggests that **1000 0000** Ex-ORed with **1000 0000** should give an all-zero result. So the hex equivalent of **1000 0000** is what you want . . . and that's **$80**. Enter **$80**, and look at the binary display. Incidentally, the zero flag flipped on.



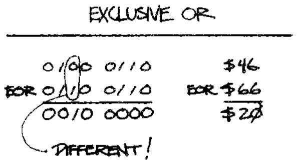
Hit <ENTER> and touch 6 again. This time, enter the result from the Ex-OR you just did. Enter **$00** into the accumulator. And enter **$80** as the operand. The result is **$80**. Here's why Exclusive-OR is called a toggle function. When value X is Ex-ORed with value Q, the result is value Y. When value Y is Ex-ORed with value Q, the result is value X. Under the Exclusive-OR function, value Q becomes a toggle, flipping back and forth between values X and Y.



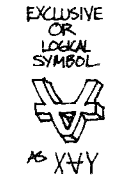
Remember the flashing "F" at the top of the screen when you load cassettes into the Color Computer? This alternates value **$46** with value **$66**. Hit <ENTER> and touch 6 again. Enter **$46** into the accumulator, and **$66** as the operand. The result should be **$20**. **$20** can then be used in a program as a toggling value. **$46** Exclusive OR **$20** is **$66**, **$66 EOR $20** is **$46**. Uppercase F becomes lowercase F, and vice versa. And the advantage to a toggling value is this: you don't have to know *which* state the original value is in to toggle it. That's ideal, because in this example, the tape-loading program doesn't have to keep track of which "F" it's displayed.



But enough of Exclusive-OR. You can try it at the end of this lesson.



Shifts and rotates are interesting commands. Essentially, they are binary multiplication or division by two. In the decimal system, a left shift is multiplication by ten, a right shift is division by ten. If that doesn't make immediate sense, consider the number 247. Shift it to the left and it becomes 2470; shift 247 to the right and it becomes 24.7 . . . multiplication and division by ten. The difference between types of binary shifts in the 6809 has to do with what happens to the bits on either end of the byte.



* Problem: A contains $00. Execute DECA.

Answer: A contains $FF. Negative flag is set. Zero flag is reset. Carry flag not affected.

* Problem: B contains $FE. Execute INCB.

Answer: B contains $FF. Negative flag is set. Zero flag is reset. Carry flag not affected.

* Problem: B contains $FF. Execute INCB.

Answer: B contains $00. Zero flag is set. Negative flag is reset. Carry flag not affected.

* Problem: B contains $00. Execute INCB.

Answer: B contains $01. Negative and zero flags are reset. Carry flag not affected.

* Problem: B contains $01. Execute NEGB.

Answer: B contains $FF. Negative and carry flags are set. Zero flag is reset.

* Problem: B contains $00. Execute NEGB.

Answer: B contains $00. Zero flag is set. Negative and carry flags are reset.

* Problem: B contains $80. Execute NEGB.

Answer: B contains $80. Negative and carry flags are set. Zero flag is reset.

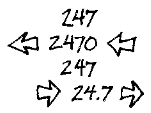
* Problem: A contains $AA. Execute NEGA.

Answer: A contains $56. All flags are reset.

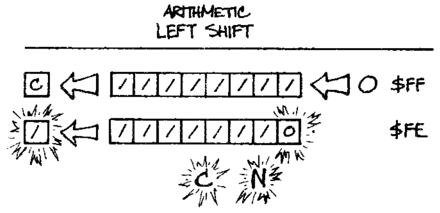
*** What is the purpose of the condition code register?**

The condition code register provides information about the most recent instruction executed by the processor.

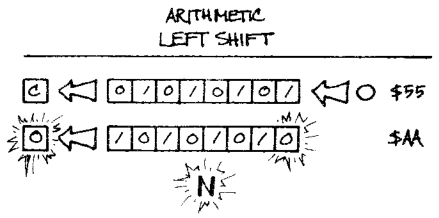
An arithmetic shift to the left puts a zero into the rightmost position; a similar shift to the right leaves a trail of the value of the leftmost bit. The bit that is shifted out the end of the byte falls into the carry flag; in a situation like this, the carry flag is sometimes called a "bit bucket". A logical shift left is identical to an arithmetic shift, but a logical shift right leaves a zero in the leftmost position. Again, the bit falling off the end drops into the carry flag. Finally, a rotate command is circular, as the bits move left or right through the carry flag. Try the arithmetic shift here.

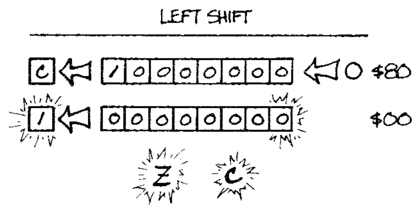


Hit <ENTER>, and touch 3. You've got a prompt for an arithmetic shift. Do the left shift first; touch L. Put a hex value **$FF** into the accumulator. The row of bits is shifted left, a zero follows from the right, and the leftmost bit ends up in the carry flag. Notice that since bit 7 is high, the negative flag also goes up.

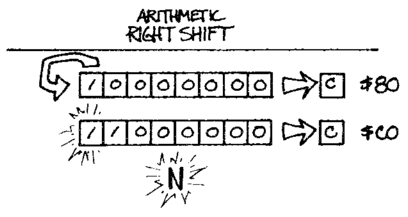


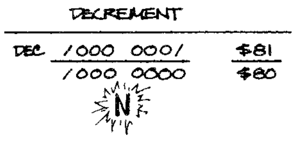
Hit <ENTER>, and touch 3. Touch L again. Put **$55** into the accumulator. Notice how the bits all move left. This number turns negative (becoming **$AA**), but the carry flag is zero. You can explore all those details later; try a right shift now.



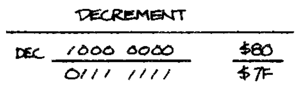


Hit <ENTER>, touch 3, and touch R. Put **$80** into the accumulator. This time observe bit 7, the leftmost bit. It begins to leave a trail of ones behind it; the value after shifting is **$C0**. Hit <ENTER>, touch 3, touch R, and enter **$C0**. The trail of ones continues to follow.

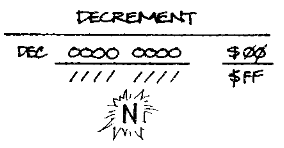


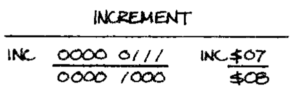


I'm going to skip doing the Logical Shifts and Rotates in this explanation; you can check out selection 8 and selection B on your own at the end of the lesson.

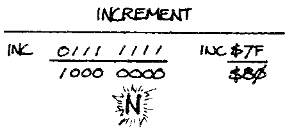


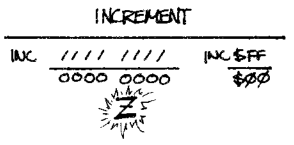
Move on to the next 6809 processor command. Hit <ENTER>, and touch 5. This is a decrement by one command. Enter **$81** into the accumulator. **$81** minus one is **$80**. The negative flag is on. Decrement it one more time; hit <ENTER>, touch 5, and put **$80** into the accumulator. The value becomes **$7F**; the negative flag is off. One more thing to notice with the decrement command. Hit <ENTER>, touch 5, and put **$00** into the accumulator. **$00** minus 1 is **$FF**. The negative flag flips on.



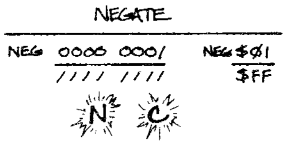


The opposite of the decrement is the increment, also a straightforward command. Hit <ENTER>, and touch 7. Enter **$07** into the accumulator. The value is incremented by one to **$08**. Not much there; all flags are off. Hit <ENTER>, and touch 7. This time put **$7F** into the accumulator. The value increments from **$7F** to **$80**. The negative flag flips on. Finally, hit <ENTER> and touch 7 again. Enter **$FF** into the accumulator. The number increments with the result being **$00**.

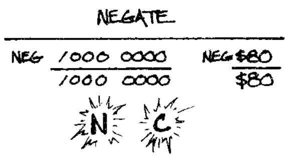




There's just one selection left, and that's NEGate, selection 9. Hit <ENTER>, and touch 9. Enter **$01** into the accumulator. The negative of **$01** is **$FF**. If you recall from an earlier lesson, you counted backwards from zero in one programming example, and it makes sense that one less than zero, –1 in decimal, would by **$FF** in 8-bit data.



Hit <ENTER> and touch 9. Put **$80** into the accumulator. The result is — **$80**! I'll leave you to check the flags and ponder that result.



Please review this lesson, spend some time with pages 30 and 31 of the MC6809E data booklet, and — most of all — keep using this program. Try every example; work the results out on paper, and see if you agree with the final display. Examine how the binary data works, how the instructions perform, and what the flags mean.

9.

Making things happen on your 6809-based Color Computer is the point of all this. I've created this series because your computer is a special machine — not just an isolated microprocessor, but an interrelated group of components capable of video, sound, storage and communication, with add-ons like joysticks and disks and printers. So while you're making your way through the intricacies of the 6809 itself, I'm also going to provide you with the information you need to use the whole computer.

That means I've got to talk about two things specific to the Color Computer: memory maps and smart components.

The memory map of your computer describes the way its 65,536 individual addresses are organized . . . what goes where. Simplicity is always important in laying out a memory map, and that holds true for the Color Computer. I've reproduced the Color Computer memory maps in the documentation so you can follow along.

There are a few special considerations in this machine, but the memory map I'll describe is what's set up when you turn the power on. Read/write memory — also known as random-access memory, or RAM — is located (talking in hexadecimal now) from address **$0000** to **$7FFF**. That's 32K of memory; if you have a 16K computer, your RAM ends at **$3FFF**. **$4000** to **$7FFF** remains unused until you fill it.

The BASIC language is made up of machine-language instructions and data, so it too occupies part of the memory map. BASIC is broken up into two halves, each half 8K long. From hex **$8000** to **$9FFF** you will find Extended Color BASIC, and from hex **$A000** to **$BFFF** you will find Color BASIC.

Starting at **$C000** is a blank space. As far as the processor is concerned, no memory is "blank" per se, but an off-the-shelf Color Computer doesn't have anything connected at **$C000**. However, when you plug in a ROMpack cartridge,

Practical application of your 6809 learning means knowing something about this particular 6809 environment. And that means knowing the Color Computer better. It's not the only 6809 machine there is, so you'll need to learn all new details if you purchase a Whatzit-99 or the CompuBlob.

* What do you call the description of how the computer's designers have arranged its memory?

A memory map.

* How many memory locations are there in the Color Computer?

65,536 locations.

* What is the address range of the Color Computer, in hex?

$0000 to $FFFF.

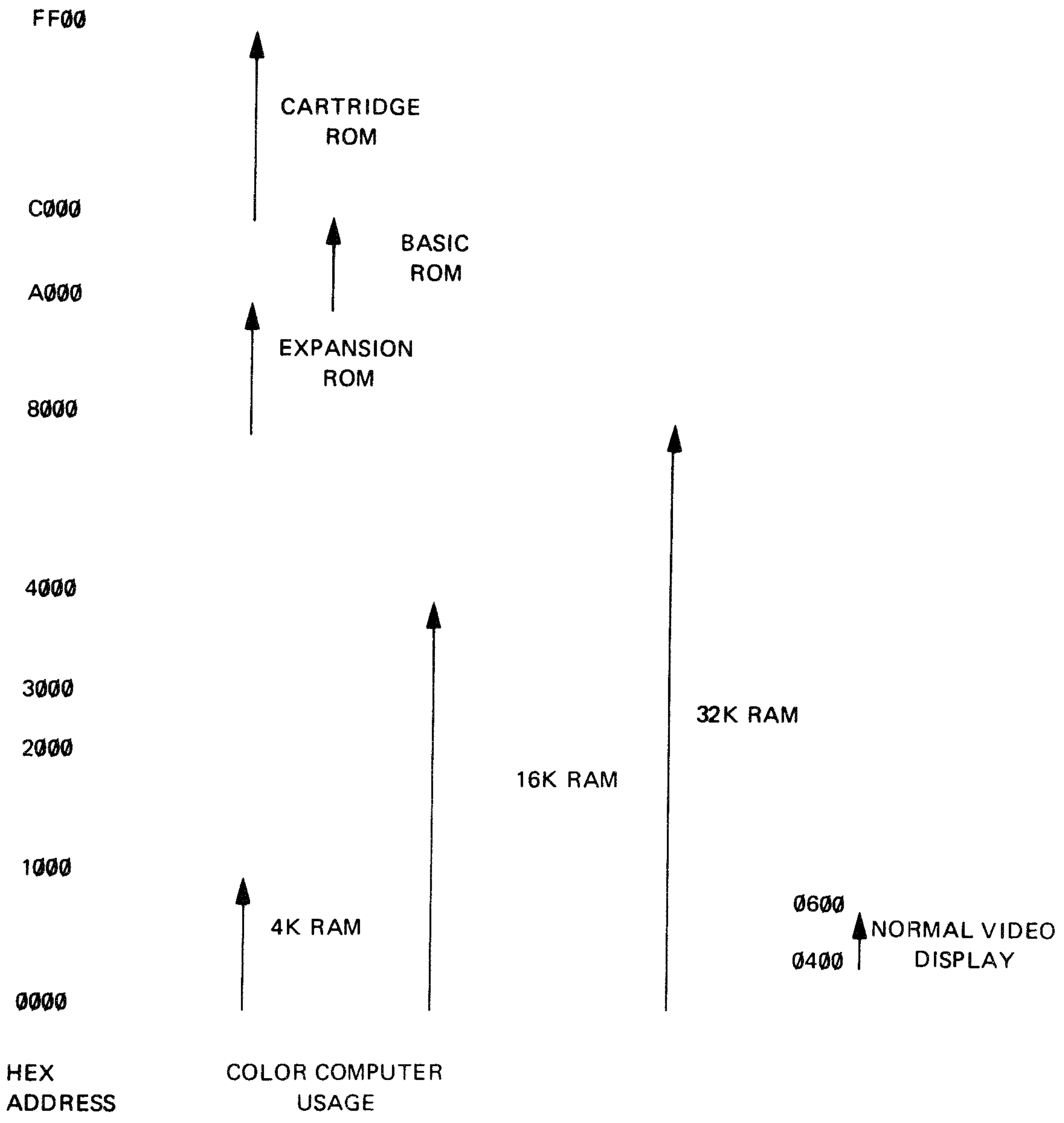
* How many "K" is the address range of the Color Computer?

64K.

* Where in the memory map is read-write (random-access) memory, or RAM, in the Color Computer on a 16K machine?

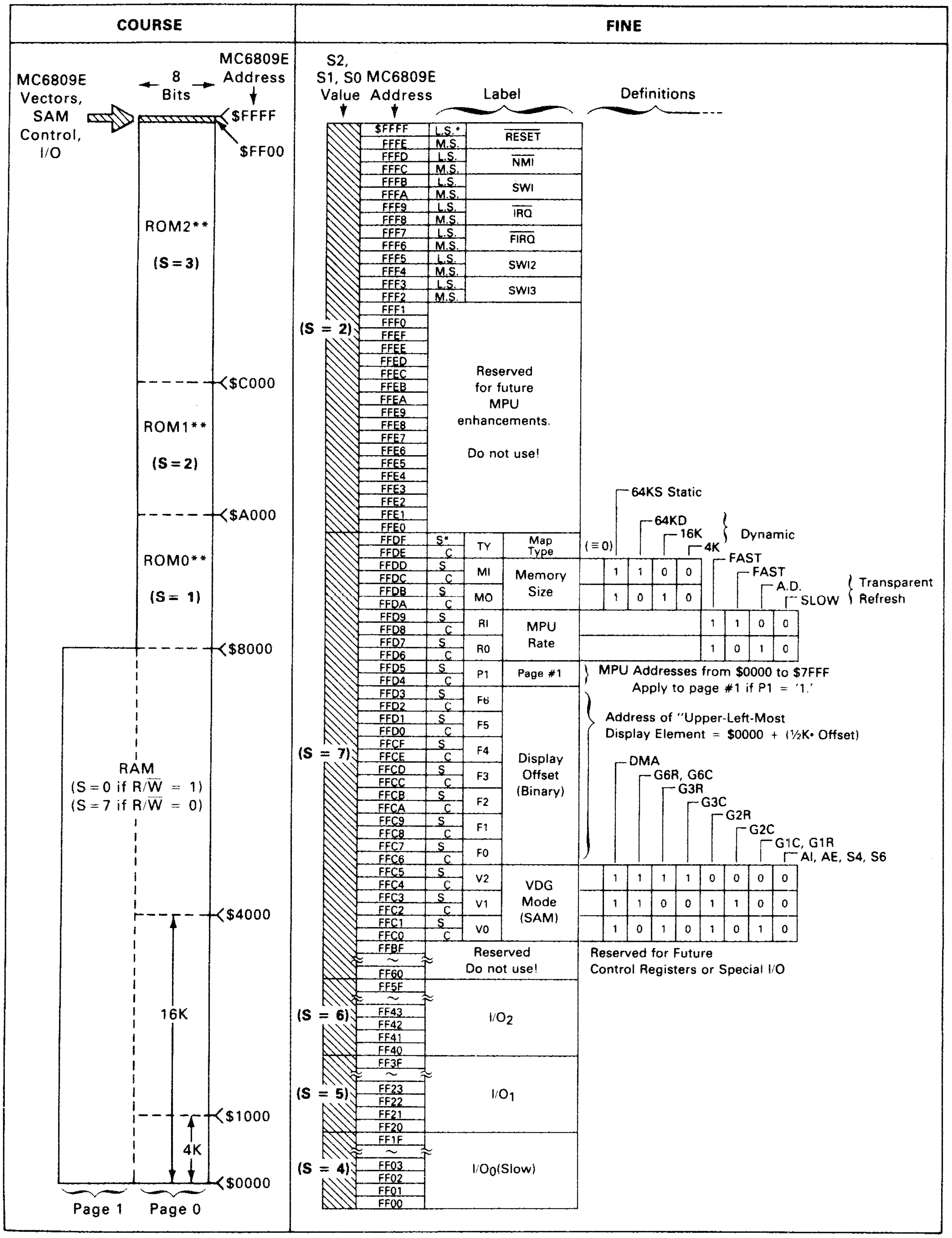
RAM is located from $0000 to $3FFF.

COLOR COMPUTER MEMORY MAP



FFFF OR BFFF	27	RESET VECTOR LSB
FFFE OR BFFE	A0	RESET VECTOR MSB
FFFD OR BFFD	09	NMI VECTOR LSB
FFFC OR BFFC	01	NMI VECTOR MSB
FFFB OR BFFB	06	SWI1 VECTOR LSB
FFFA OR BFFA	01	SWI1 VECTOR MSB
FFF9 OR BFF9	0C	IRQ VECTOR LSB
FFF8 OR BFF8	01	IRQ VECTOR MSB
FFF7 OR BFF7	0F	FIRQ VECTOR LSB
FFF6 OR BFF6	01	FIRQ VECTOR MSB
FFF5 OR BFF5	03	SWI2 VECTOR LSB
FFF4 OR BFF4	01	SWI2 VECTOR MSB
FFF3 OR BFF3	00	SWI3 VECTOR LSB
FFF2 OR BFF2	01	SWI3 VECTOR MSB

MEMORY MAP



*Note:
M.S. ≡ Most Significant S ≡ Set Bit
L.S. ≡ Least Significant C ≡ Clear Bit } (All bits are cleared when SAM is reset.)
S = Device Select value = 4 x S2 + 2 x S1 + 1 x S0

**May also be RAM

COLOR COMPUTER MEMORY MAP (cont'd)

FFØØ – FFØ3 PIA U8

FFØØ
- BIT Ø = KEYBOARD ROW 1 and right joystick switch
- BIT 1 = KEYBOARD ROW 2 and left joystick switch
- BIT 2 = KEYBOARD ROW 3
- BIT 3 = KEYBOARD ROW 4
- BIT 4 = KEYBOARD ROW 5
- BIT 5 = KEYBOARD ROW 6
- BIT 6 = KEYBOARD ROW 7
- BIT 7 = JOYSTICK COMPARISON INPUT

FFØ1
- BIT Ø, BIT 1: Control of the Horizontal sync clock (63.5 microseconds) Interrupt Input
  - BIT Ø: Ø=IRQ* to CPU Disabled; 1=IRQ* to CPU Enabled
  - BIT 1: Ø=Flag set on the falling edge of HS; 1=Flag set on the rising edge of HS
- BIT 2 = Normally 1: Ø=Changes FFØØ to the data direction register
- BIT 3 = SEL 1: LSB of the two analog MUX select lines
- BIT 4 = 1 Always
- BIT 5 = 1 Always
- BIT 6 Not Used
- BIT 7 = Horizontal sync interrupt flag

FFØ2
- BIT Ø= KEYBOARD COLUMN 1
- BIT 1= KEYBOARD COLUMN 2
- BIT 2= KEYBOARD COLUMN 3
- BIT 3= KEYBOARD COLUMN 4
- BIT 4= KEYBOARD COLUMN 5
- BIT 5= KEYBOARD COLUMN 6
- BIT 6= KEYBOARD COLUMN 7
- BIT 7= KEYBOARD COLUMN 8

FFØ3
- BIT Ø, BIT 1: Control of the field sync clock 16.667 Ms Interrupt Input
  - BIT Ø: Ø=IRQ* to CPU Disabled; 1=IRQ* to CPU Enabled
  - BIT 1: Ø= sets flag on falling edge FS; 1= sets flag on rising edge FS
- BIT 2 = NORMALLY 1: Ø= changes FF02 to the data direction register
- BIT 3 = SEL 2: MSB of the two analog MUX select lines
- BIT 4 = 1 Always
- BIT 5 = 1 Always
- BIT 6 Not Used
- BIT 7 = Field sync interrupt flag

COLOR COMPUTER MEMORY MAP (Cont'd)

FF2Ø – FF23 PIA U4

FF2Ø	BIT Ø = CASSETTE DATA INPUT		
	BIT 1 = RS-232 DATA OUTPUT		
	BIT 2 = 6 BIT D/A LSB		
	BIT 3 = 6 BIT D/A		
	BIT 4 = 6 BIT D/A		
	BIT 5 = 6 BIT D/A		
	BIT 6 = 6 BIT D/A		
	BIT 7 = 6 BIT D/A MSB		
FF21	BIT Ø	Control of the CD RS-232 status Input	Ø = FIRQ* to CPU Disabled 1 = FIRQ* to CPU Enabled
	BIT 1		Ø = set flag on falling edge CD 1 = set flag on rising edge CD
	BIT 2 = Normally 1:	Ø = changes FF2Ø to the data direction register	
	BIT 3 = Cassette Motor Control:	Ø = OFF	1 = ON
	BIT 4 = 1 Always		
	BIT 5 = 1 Always		
	BIT 6 Not Used		
	BIT 7 = CD Interrupt Flag		
FF22	BIT Ø = RS-232 DATA INPUT		
	BIT 1 = SINGLE BIT SOUND OUTPUT		
	BIT 2 = RAM SIZE INPUT	LOW = 4K	HIGH = 16K
	BIT 3 = VDG CONTROL OUTPUT		CSS
	BIT 4 = VDG CONTROL OUTPUT		GMØ & $\overline{\text{INT}}$/EXT
	BIT 5 = VDG CONTROL OUTPUT		GM1
	BIT 6 = VDG CONTROL OUTPUT		GM2
	BIT 7 = VDG CONTROL OUTPUT		$\overline{\text{A}}$/G
FF23	BIT Ø	Control of the Cartridge Interrupt Input	Ø = FIRQ* to CPU Disabled 1 = FIRQ* to CPU Enabled
	BIT 1		Ø = sets flag on falling edge CART* 1 = sets flag on rising edge CART*
	BIT 2 = Normally 1:	Ø = changes FF22 to the data direction register	
	BIT 3 = Six BIT Sound Enable		
	BIT 4 = 1 Always		
	BIT 5 = 1 Always		
	BIT 6 = Not Used		
	BIT 7 = Cartridge Interrupt Flag		

the addresses from **$C000** to **$FEFF** are decoded for use by the ROMpack. Notice I said **$C000** to **$FEFF**.

There is a block of memory from **$FF00** to **$FFFF** that is very special. In the back of your documentation, find the data booklet entitled MC6883, and turn to page 17. Here is a table marked Memory Map Type #0. Look at the left half, marked "course" (meaning a course breakdown of the memory map). You can see the layout of the address blocks I've described so far, and at the very top, a small block called "MC6809E vectors, SAM control, I/O". A blowup of this tiny block is shown on the right side of the figure, marked "fine".

Before looking at the detailed map, I want to tell you about the SAM. You may have heard this term before; I was mystified the first time I encountered it. You're holding the SAM's data booklet now. SAM means "Synchonous Address Multiplexer", a mouthful that breaks down to three simple concepts. It's synchronous because it is completely synchronized with the operation of the 6809 processor itself, as well as with the video display, memory, and so forth. It deals with addresses, its main task. And it is a multiplexer because it is the traffic cop, sending the proper addresses to the correct memory blocks. If that doesn't interest you, then let me say that, all because of the SAM, your Color Computer is a 96K computer.

On to the map. Start from the bottom of the "fine" map. You'll see three blocks from **$FF00** to **$FF5F** marked I/O, meaning input/output. At these addresses — and more on this later — are found the keyboard, joystick inputs, cassette input and output, printer input and output, cassette motor control, various high-resolution color modes, and other computer control information. Also, the plug-in disk pack and different peripheral devices use these input/output addresses. That's a lot to know about, but the many capabilities of the Color Computer are found in these input/output blocks.

Next up on the map is a group of addresses ($FF60 to **$FFBF**) which are not defined yet by the manufacturer of the Synchronous Address Multiplexer, the SAM.

Up from there at address **$FFC0** begin a unique series of SAM registers. There was a standard joke among memory engineers that they'd developed the read/write memory — where information could be stored and retrieved — and the read-only memory, where information was permanently fixed and could only be retrieved — but hadn't developed the write-only memory, where information could be stored but couldn't be retrieved. Well, the SAM's got it. Actually, these memory locations are called write-only registers, and their job is to perform computer control functions. Your program keeps track of what's been done, since these are infrequently accessed items. Interestingly, what data you store in these registers is completely irrelevant . . . all that matters is that you store *something* there.

* What is the range of RAM on a 32K machine?

RAM is located from $0000 to $7FFF.

* The Color Computer's operating language is located in what kind of memory?

Read-only memory, or ROM.

* The Color Computer's operating language is in two linked parts. What are they called?

Color BASIC and Extended Color BASIC.

* Where is Color BASIC in the memory map?

From $A000 to $BFFF.

* Where is Extended Color BASIC in the memory map?

From $8000 to $9FFF.

* What is located from $C000 to $FEFF on an off-the-shelf the Color Computer?

Nothing; the space is reserved.

* What is the space from $C000 to $FEFF reserved for?

For plug-in cartridge ROM, also called ROMpacks or program cartridges.

* What is located in the memory map from $FF00 to $FFFF?

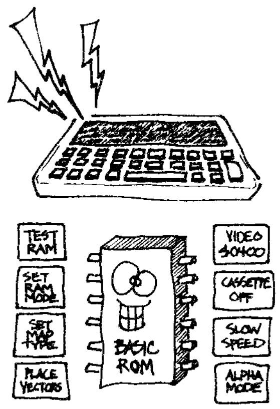
MC6809E vectors, SAM control and I/O.

* What is the SAM?

The Synchronous Address Multiplexer.

* What does I/O mean?

I/O means input/output.



Included in the write-only registers are six addresses to set and reset the eight graphics display modes; 14 addresses to define the area of memory to be displayed on the screen; and 12 addresses to define which 32K block or RAM will be used in a 96K machine, what processor speed will be used, how much memory is available, and which memory map arrangement will be used.

All of these registers are set up by Color BASIC when the power is turned on, but you *can* change them at any time. I've got a little BASIC program to play around with the video graphics modes. Get it loaded, and then I'll tell you about it.

Program #14, 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/0 error occurs, rewind to the start of the program and try again. For severe loading problems, see the Appendix.

```
1  REM * USING ALL VIDEO MODES
2  REM * PORT $FF22 SELECTS VIDEO
3  FORX=8TO128STEP8
4  POKE&HFF22,(X OR 4)
5  REM * ADDRESSES TO CLEAR MODE
6  C1=&HFFC0:C2=&HFFC2:C3=&HFFC4
7  REM * ADDRESSES TO SET MODE
8  S1=&HFFC1:S2=&HFFC3:S3=&HFFC5
9  REM  ***********************
10  REM * BEGIN CHANGING MODES *
11  REM ***********************
12  POKEC1,0:POKEC2,0:POKEC3,0
13  GOSUB30
14  POKES1,0:POKEC2,0:POKEC3,0
15  GOSUB30
16  POKEC1,0:POKES2,0:POKEC3,0
17  GOSUB30
18  POKES1,0:POKES2,0:POKEC3,0
19  GOSUB30
20  POKEC1,0:POKEC2,0:POKES3,0
21  GOSUB30
22  POKES1,0:POKEC2,0:POKES3,0
23  GOSUB30
24  POKEC1,0:POKES2,0:POKES3,0
25  GOSUB30
26  POKES1,0:POKES2,0:POKES3,0
27  GOSUB30
28  NEXT
29  END
30  FORN=1TO300:NEXT
31  RETURN
```

LIST lines 1 to 4. Line 2 says "Port **$FF22** selects video". What's port **$FF22**? This is another bit of jargon. The electronic circuits which allow the 6809 processor in the Color Computer to use its keyboard, cassette, video, etc., are called "peripheral interface adaptors". There are two peripheral interface adaptors, or PIAs, built into the

* What are the I/O (input/output) addreses?

$FF00 to $FF5F.

* Name some of the input/output devices located at the I/O addresses from $FF00 to $FF5F.

Keyboard, joystick inputs, cassette input and output, printer input and output, cassette motor control, sound output, high-resolution color mode control, and other computer control information.

* What does SAM mean?

Synchronous Address Multiplexer.

* The SAM contains memory locations reserved for control; what kind of registers are these?

Write-only registers.

* Name some of the purposes of these write-only registers.

To set or reset eight graphics display modes; to define the area of memory to be displayed on the screen; to define which 32K block of RAM will be used in a 96K machine; to determine what processor speed will be used; to indicate how much memory is available; to specify which memory map arrangement is to be used.

* What does SAM mean?

Synchronous Address Multiplexer.

* What does PIA mean?

Peripheral Interface Adaptor.

* What is the proper term for "setting up" a computer device.

Configuring.

computer, and each is given four memory addresses. The first PIA, for example, uses addresses **$FF00** through **$FF03**. These addresses — and I won't spend a lot of time on this right now — have two functions. **$FF01** and **$FF03** — the odd-numbered registers — are called "control registers", and are used for setting up (the word for that is "configuring") the PIAs. The even-numbered addresses **$FF00** and **$FF02** open to the outside world. They are called "ports".

What this means is that ports **$FF00** and **$FF02** of the first PIA are configured by addresses **$FF01** and **$FF03**. They are configured as input or output. That way the processor can receive or send information to the outside world when it executes machine-language instructions which load or store data at those memory addresses.

In this example, the processor can address the second PIA at **$FF20**, **$FF21**, **$FF22**, and **$FF23**. The PIA configuration using **$FF21** and **$FF23** has already been done at power-up, so that's not your concern for the moment. What you need to know is this: in address **$FF22** are the video graphics modes. One of the two color sets is selected by bit 3; graphics mode zero is turned on or off by bit 4; graphics mode one is turned on or off by bit 5; graphics mode two is turned on or off by bit 6; and the alphanumeric or graphic choice is made by bit 7. So each of the most-signficant 6 bits of address **$FF22** has a different purpose in setting up the video display.

Unless you've spent a lot of time cracking your brains over your BASIC manuals, I probably just dropped another bucket of unknowns in your lap — the graphics modes. It turns out that the Color Computer is a chain of "smart" circuits — the 6809E processor connects to the 6883 synchronous address multiplexer which in turn connects to the 6847 video display generator and the 6821 peripheral interface adaptors. Forget all those numbers. Just dig out your old COLOR BASIC manual — that's the COLOR BASIC manual, not the Extended Color one — and turn to page 256. RUN the program you've got in your computer now, and while it's running, read pages 256 through 266. If you've been spoiled by the Extended Color BASIC graphics modes, then you probably forgot all about these pages in that old Color BASIC manual. So dig in now.

* What is the term for memory addresses that open to the outside world?

Ports.

* How many ports does the Color Computer have?

Four.

* What are the Color Computer port addresses?

$FF00, $FF02, $FF20 and $FF22.

* What is the term for "setting up" a computer device?

Configuring.

* What four PIA addresses configure the four port addresses?

$FF01, $FF03, $FF21 and $FF23.

* What are the port addresses configured as?

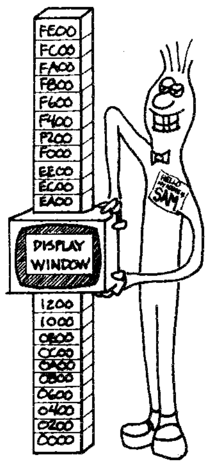
Input or output.

* How does the processor send or receive information (input or output information) with respect to the outside world?

By loading or storing data at the port memory addresses.

* At port $FF22, what is the purpose of bit 3?

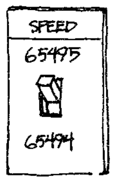
To choose one of two color sets.



By now I expect that the use of decimal numbers in the Color BASIC manual obscures rather than illuminates how all this works. You'd probably like to take a break, but don't do it yet. While this information is still fresh, I'd like you to RUN once again the program in the computer.

What you see when you run the program are all the possible combinations of alphanumeric and graphic modes that can be created by the combination of the synchronous address multiplexer (that is, the SAM) and the video display generator (that is, the VDG). I've already mentioned about port **$FF22** in the memory map. Just to review, bits 3 through 7 of that byte can be used to select one of two color sets; turn graphic modes one, two and three on or off; and select between alphanumerics and graphics.

The choice of bits you turn on or off at port **$FF22** can then be combined with the SAM's video registers to offer additional possibilities for display. To get at them, though, you have to understand how the SAM's peculiar "write-only" registers work. You still have that BASIC program in place. LIST lines 5 through 8. I've defined six variables here. C1, C2 and C3 mean clear 1, clear 2 and clear 3, and are defined as the three even-numbered addresses **$FFC0**, **$FFC2** and **$FFC4**. S1, S2 and S3 mean set 1, set 2, and set 3, and are defined as the three odd-numbered addresses **$FFC1**, **$FFC3** and **$FFC5**. It turns out that writing to an *address*, no matter what the data stored, either sets or resets a condition within the SAM.



Some of you may have used the high-speed mode on your Color Computer, sometimes called the Vitamin Q poke. You probably wrote it, POKE65495,0 and to get normal speed, POKE 65494,0. When you did that POKE, you were actually executing a Store Accumulator to memory location **$FFD7** for high speed and **$FFD6** for normal speed.

Flip to the SAM data booklet (the booklet marked MC6883), and return to page 17. Locate addresses **$FFC0** through **$FFC5**. These are the video display modes, the VDG modes. At the right of the addresses, the mode combinations are shown in binary. To turn on any of these modes, the binary data has to be expressed as a trio of addresses — either the clear address (the even ones) or the set addresses (the odd ones).

Likewise, locate addresses **$FFD6** and **$FFD7**. They are part of a group of addresses that affect speed of the computer. At power up, your computer is in the "slow" mode. By writing to **$FFD7**, you set the "A.D.", or address dependent, mode. In that mode, your BASIC ROM zipped along at double speed, and your RAM just stayed the way it was. Had you poked **$FFD9**, you would have gone into the "fast RAM" mode, losing both the video display and the refresh your memory needs to keep its information.

You don't need the BASIC program now, so <BREAK> out of it if it's still running. I want to show you what happens when you use the "fast RAM" mode at address **$FFD9**.

* What is the purpose of bits 4 through 6?

To select among the graphics modes.

* What is the purpose of bit 7?

To select either alphanumerics or graphics.

* What does PIA mean?

Peripheral Interface Adaptor.

* What is the term for memory addresses which open to the outside world?

Ports.

* What does SAM mean?

Synchronous Address Multiplexer.

* What sets or resets a condition within the SAM?

Writing to a SAM address (register).

* What sets or resets video display modes?

Writing to the SAM video display addresses (registers).

* What are the SAM video display registers?

$FFC0 through $FFC5.

* What changes the computer's processing speed?

Writing to the SAM clock rate addresses (registers).

* What are the SAM clock rate registers?

$FFD6 through $FFD9.

Display offset

* What is the normal speed of the Color Computer ?

.89 MHz (894,886 pulses per second).

* Where is the normal video display screen on the Color Computer (in decimal and hex)?

At 1024 ($0400 hex).

* What does VDG mean?

Video Display Generator.

* What determines the screen being displayed?

The SAM display offset addresses (registers).

* What are the display offset registers?

$FFC6 through $FFD3.

* How many bits of the 16-bit address are selected by the display offset registers?

Seven.

* How many combinations of 7 bits are possible?

128.

* How many display screens are possible by using the SAM's display offset addressing technique?

128.

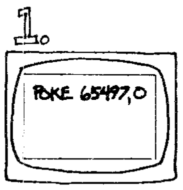
* How do you create a display offset address?

By writing to the SAM display offset registers.

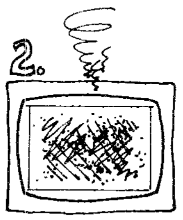
* How do you create the offset address 0000000?

By writing to all the even-numbered SAM display offset addresses (registers).

$FFD9 is 65497 decimal. So type POKE 65497,0 and hit <ENTER>. POKE 65497,0.



Screen freaked out, right? Hit your Reset button on the back right to get back your screen. Whether or not the program is still intact depends, for technical reasons, on whether you have a 16K, 32K or 64K machine.



There's some more to find out about the SAM, so I have another program for you.

Program #15, 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 and try again. For severe loading problems, see the Appendix.

```
1 CLS
2 PRINT" REDIRECTING THE VIDEO DISPLAY":PRINT
3 C0=&HFFC6:C1=&HFFC8:C2=&HFFCA:C3=&HFFCC:C4=&HFFCE:C5=&HFFD0:C6
=&HFFD2
4 S0=&HFFC7:S1=&HFFC9:S2=&HFFCB:S3=&HFFCD:S4=&HFFCF:S5=&HFFD1:S6
=&HFFD3
5 INPUT"THE NORMAL SCREEN IS LOCATED AT $0400 TO $05FF.  THE SAM
 ALLOWS THE SCREEN TO POINT TO ANY PLACEIN MEMORY.  THERE ARE 12
8 SCREENSIN ALL.  ENTER A NUMBER FROM 0 TO 127 TO DISPLAY A SCRE
EN";A$
6 A=VAL(A$):IFA<0ORA>127THENCLS:GOTO5
7 B6=FIX(A/64)
8 B5=FIX((A-(B6*64))/32)
9 B4=FIX((A-(B6*64)-(B5*32))/16)
10 B3=FIX((A-(B6*64)-(B5*32)-(B4*16))/8)
11 B2=FIX((A-(B6*64)-(B5*32)-(B4*16)-(B3*8))/4)
12 B1=FIX((A-(B6*64)-(B5*32)-(B4*16)-(B3*8)-(B2*4))/2)
13 B0=FIX(A-(B6*64)-(B5*32)-(B4*16)-(B3*8)-(B2*4)-(B1*2))
14 IFB0=0THENPOKEC0,0ELSEPOKES0,0
15 IFB1=0THENPOKEC1,0ELSEPOKES1,0
16 IFB2=0THENPOKEC2,0ELSEPOKES2,0
17 IFB3=0THENPOKEC3,0ELSEPOKES3,0
18 IFB4=0THENPOKEC4,0ELSEPOKES4,0
19 IFB5=0THENPOKEC5,0ELSEPOKES5,0
20 IFB6=0THENPOKEC6,0ELSEPOKES6,0
21 FORN=1TO2000:NEXT
22 GOTO1
```

The object of this program is to manipulate the SAM "display offset" registers. This nifty technique makes it possible to display 128 entirely different screens of information, each 512 (hex **$200**) bytes long.

RUN this program, and enter 2 in response to the prompt. There is a pause, and the cursor is back. Of the 128 possible screens, the one you normally look at the screen #2. Now enter 0. Aha. A screen full of garbage and wiggly characters appears before you. Try that again; enter 0. Screen #0 is what you see, and screen #0 reveals pages **$00** and **$01** of your memory. Remember the Direct Page register? The Color Computer's BASIC sets the DP register to **$00**, meaning what you're seeing is all the down-and-dirty work BASIC does to count, calculate, delay, and so on.

Now I'll show you what's happening there. Turn once again to page 17 of the SAM data booklet, where the detailed memory map is shown. Addresses **$FFC6** to **$FFD3** are called a display offset value, and a strange formula is given, reading "Address of upper-left-most display element = **$0000** + (1/2K * offset)". Obscurity won't triumph, I'll tell you. What this means is that you can display any area of memory directly on the screen, in even 512-byte blocks.

Addresses **$FFC6** to **$FFD3** are those write-only SAM registers again, used here to create the most-significant 7 bits of an address. Writing to the even-numbered registers starting with **$FFC6** clears bits to zero; writing to the odd-numbered registers sets bits to one. So if you store information in all the even-numbered registers, you create the binary number **0000 000** . . . 7 bits long. If you store information in all the even-numbered registers except **$FFC8**, but store information in the odd-numbered register **$FFC9**, and you create the binary number **0000 010**. Those are the most significant seven bits of addresses **0000 0100 0000 0000** through **0000 0101 1111 1111**. Those binary addresses translate into **$0400** to **$05FF** — the address of the normal video screen.

That's all I have for you this time. I would like you to LIST this program, and get an idea of how to manipulate the addresses. Take a break, play with the program, and then come back for the next session; you'll be translating these concepts into an assembly-language subroutine.

To review: the Color Computer is more than a smart 6809 processor, and so effective programming on this machine requires knowing the rest of the smart devices inside it. These devices include a video display generator (VDG) to provide alphanumeric and graphic displays in several colors; a synchronous address multiplexer (SAM) to coordinate and synchronize events involving input/output, display, and memory addressing; and two peripheral interface adapters (PIAs) to provide input and output for keyboard, cassette, printer, video, sound, and other computer control functions.

These smart devices all have control signals which are connected into the memory map and given specific addresses. By storing information at these addresses, your programs can have control of all the computer's functions.

Please review this lesson, and familiarize yourself with the programming aspects presented in the data booklets for the MC6883 SAM, the MC6847 VDG, and the MC6821 PIA.

After you've finished trying out and examining this program, there's one more at the end of the lesson. Load, LIST and RUN it. It should give you some ideas.

* What are the even-numbered display offset registers?

$FFC6, $FFC8, $FFCA, $FFCC, $FFCE, $FFD0 and $FFD2.

* How do you create the display offset address 1111111?

By writing to all the odd-numbered SAM display offset registers.

* What are the odd-numbered SAM display offset registers?

$FFC7, $FFC9, $FFCB, $FFCD, $FFCF, $FFD1 and $FFD3.

* How do you create the diplay offset address 0110110?

By writing to a combination of odd and even addresses: $FFC6, $FFC9, $FFCB, $FFCC, $FFCF, $FFD1 and $FFD2.

* What is the address of the first byte displayed on the screen with the offset address 0110110?

The first byte (the upper-left-most byte) displayed is $6C00.

* What does VDG mean?

Video Display Generator.

* What does PIA mean?

Peripheral Interface Adaptor.

* What does SAM mean?

Synchronous Address Multiplexer.

* What is located in the lower half of the Color Computer's memory map (from $0000 to $7FFF)?

Read/write memory (random-access memory), or RAM.

Program #16

* What is located from $8000 to $9FFF?

Extended Color BASIC in read-only memory (ROM).

* What is located from $A000 to $BFFF?

Color BASIC in read-only memory (ROM).

* What is located from $C000 to $FEFF?

Nothing unless a cartridge read-only memory (ROM) pack is plugged in.

* What is located from $FF00 to $FFFF?

MC6809E vectors, SAM control, and I/O.

* What do you call the description of how the computer's designers have arranged its memory?

The memory map.

Program #16, 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/0 error occurs, rewind to the start of the program and try again. For severe loading problems, see the Appendix.

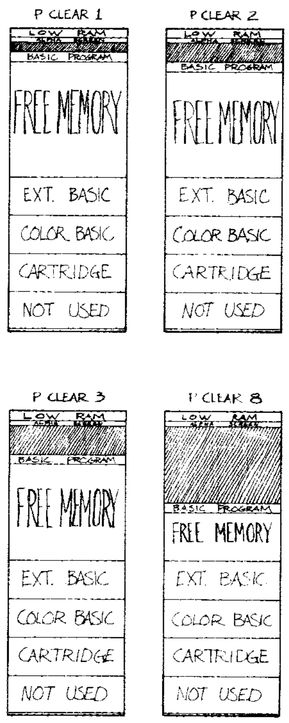
```
1 CLS:CLEAR200.16292:PCLEAR4:X=&H0400
2 GOSUB43:GOSUB86:GOSUB99
3 GOSUB55:GOSUB86:GOSUB99
4 GOSUB65:GOSUB86:GOSUB94:GOSUB99
5 GOSUB76:GOSUB86:GOSUB94:GOSUB99
6 GOSUB86:GOSUB99
7 GOSUB94:GOSUB99
8 GOSUB97:GOSUB99
9 DATA B7,FF,C7,B7,FF,C9,B7,FF,CA,B7,FF,CC,39
10 DATA B7,FF,C6,B7,FF,C8,B7,FF,CB,B7,FF,CC,39
11 DATA B7,FF,C7,B7,FF,C8,B7,FF,CB,B7,FF,CC,39
12 DATA B7,FF,C6,B7,FF,C9,B7,FF,CB,B7,FF,CC,39
13 DATA B7,FF,C7,B7,FF,C9,B7,FF,CB,B7,FF,CC,39
14 DATA B7,FF,C6,B7,FF,C8,B7,FF,CA,B7,FF,CD,39
15 DATA B7,FF,C7,B7,FF,C8,B7,FF,CA,B7,FF,CD,39
16 FORX=16293 TO 16383:READA$:A=VAL("&H"+A$):POKEX,A:NEXT
17 DEFUSR1=16293
18 DEFUSR2=16306
19 DEFUSR3=16319
20 DEFUSR4=16332
21 DEFUSR5=16345
22 DEFUSR6=16358
23 DEFUSR7=16371
24 FORA=1TO40
25 GOSUB100:GOSUB108
26 GOSUB101:GOSUB108
27 GOSUB102:GOSUB108
28 GOSUB103:GOSUB108
29 NEXT
30 FORA=1TO20
31 GOSUB103:GOSUB108
32 GOSUB104:GOSUB108
33 NEXT
34 FORA=1TO20
35 GOSUB104:GOSUB108
36 GOSUB105:GOSUB108
37 NEXT
38 FORA=1TO20
39 GOSUB105:GOSUB108
40 GOSUB106:GOSUB108
41 NEXT
42 GOTO24
43 REM
44 PRINT@0,"*   *   *   *   *   *   *   *    ";
45 PRINTSTRING$(31,32)"*";
46 PRINT:PRINT:PRINT"*"
47 PRINTSTRING$(31,32)"*";
48 PRINT:PRINT:PRINT"*"
49 PRINTSTRING$(31,32)"*";
50 PRINT:PRINT:PRINT"*"
51 PRINTSTRING$(31,32)"*";
52 PRINTSTRING$(32,32);
53 PRINT" *   *   *   *   *   *   *   * ";
54 RETURN
55 PRINT@0," *   *   *   *   *   *   *   *"
56 PRINT:PRINTSTRING$(31,32)"*";
57 PRINT"*":PRINT:PRINT
58 PRINTSTRING$(31,32)"*";
59 PRINT"*":PRINT:PRINT
60 PRINTSTRING$(31,32)"*";
```

```
61 PRINT"*":PRINT:PRINT
62 PRINTSTRING$(31,32)"*";
63 PRINT"*   *   *   *   *   *   *   *  ";
64 RETURN
65 PRINT@0," *   *   *   *   *   *   *   *"
66 PRINT:PRINT"*"
67 PRINTSTRING$(31,32)"*";
68 PRINT:PRINT:PRINT"*"
69 PRINTSTRING$(31,32)"*";
70 PRINT:PRINT:PRINT"*"
71 PRINTSTRING$(31,32)"*";
72 PRINT:PRINT:PRINT"*"
73 PRINT"   *   *   *   *   *   *   *   ";
74 POKE1535,106
75 RETURN
76 PRINT@0,"   *   *   *   *   *   *   *   *";
77 PRINT"*":PRINT:PRINT
78 PRINTSTRING$(31,32)"*";
79 PRINT"*":PRINT:PRINT
80 PRINTSTRING$(31,32)"*";
81 PRINT"*":PRINT:PRINT
82 PRINTSTRING$(31,32)"*";
83 PRINT"*":PRINT
84 PRINT"  *   *   *   *   *   *   *   *";:POKE1535,96
85 RETURN
86 PRINT@68,"the message can be made";
87 PRINT@132," TO FLICKER AND FLASH";
88 PRINT@196,STRING$(23,191);
89 PRINT@260,"       GREEN <<";
90 PRINT@292,"       MOUNTAIN";
91 PRINT@324,"       >> MICRO";
92 PRINT@388,STRING$(23,191);
93 RETURN
94 PRINT@196,STRING$(23,207);
95 PRINT@388,STRING$(23,207);
96 RETURN
97 PRINT@132," TO flicker AND flash";
98 RETURN
99 X=X+&H200:Y=&H0400:FORQ=X TO X+512:POKEQ,PEEK(Y):Y=Y+1:NEXT:R
ETURN
100 M=USR1(0):RETURN
101 M=USR2(0):RETURN
102 M=USR3(0):RETURN
103 M=USR4(0):RETURN
104 M=USR5(0):RETURN
105 M=USR6(0):RETURN
106 M=USR7(0):RETURN
107 FORN=1TO500:NEXT:RETURN
108 FORN=1TO4:NEXT:RETURN
```

10.

Welcome back. I hope you've had a little fun with the final program in the last session. If you took the time to contrast the listing of that program with the previous one, you may have noticed a group of hexadecimal numbers and a series of USR routines in place of the BASIC POKEs. Remember that the synchronous address multiplexer — the SAM — uses write-only registers that are located in the upper area of memory. Fourteen of those addresses are used to set or reset the individual binary digits of a 7-bit video display address.

Turn back to the last program listing. PCLEAR4 in the first line is intended to release memory for Extended BASIC's high-resolution graphics. What it actually does is move the BASIC program itself in memory, freeing a large block memory space between **$0600** and the start of the BASIC program. The way I've arranged the screens is first to print them on the screen, meaning they appear in memory at **$0400** to **$05FF**, the usual address of screen memory when you turn the computer on. The info is printed on the screen by seven subroutines, and then, byte by byte, POKEd into memory at **$0600**, **$0800**, **$0A00**, etc., in blocks of 512 bytes.



The screens are then prepared. All that remains is to redirect the video display by changing the video address in the SAM. My earlier program POKEd the changes in place, but the changes happen too slowly in BASIC. The results are illegible, with unwanted screens flickering by between POKEs. So I've set up some simple machine-language subroutines, which you can see in raw form in lines 9 through 15.

I'd like you to read these. Turn to your MC6809E data booklet, and open to pages 28 and 29. The first hexadecimal byte in the program is **$B7**. Look through the data booklet's numerical listing, and you find that **B7** corresponds to STA, or Store A Accumulator, in the extended addressing mode. The extended addressing mode, as you know, means that the two bytes following the opcode form an address where the data is loaded from or

Coming into this lesson with concepts securely in your mind, you'll be solving a problem by structuring and programming a useful piece of software. Review comes first, then you'll get right into it.

*** What does the BASIC POKE statement do?**

It directly manipulates memory.

*** What is the purpose of BASIC's PCLEAR statement?**

To release memory for high-resolution graphics.

*** What controls the video display address?**

SAM registers.

*** Where is the video screen located in the normal Color Computer?**

At $0400.

*** How is the address determined?**

By writing to the SAM display offset registers.

* What is extended addressing?

An addressing mode where the two bytes following the opcode form an address where the data can be found.

* What addressing mode is utilized by STA $FFC7?

Extended addressing.

* Remember that it's the act of storing — not the information stored — into the SAM registers that determines the result. With that in mind, what action is taken by:
STA $FFC7
STA $FFC9
STA $FFCA
STA $FFCC
STA $FFCE
STA $FFD0
STA $FFD2

The video display offset address 0000011 is selected.

* What memory address is this?

$0600.

* The hex opcode for store A accumulator extended is $B7. What does $B7 06 00 indicate?

Store A accumulator at memory address $0600 (STA $0600).

* What is the clock speed of the Color Computer?

.89 MHz (894,886 clock cycles or pulses per second).

* How long is one clock cycle?

1.11746 microseconds (millionths of a second).

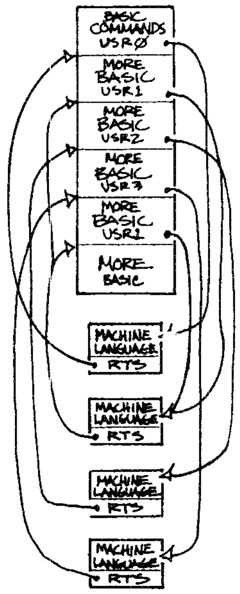
* How many clock cycles does a STA extended command take (the information is in the data booklet).

5 clock cycles.

stored. The next two hex numbers in the program are **$FF** and **$C7**. Your SAM data booklet will tell you that **$FFC7** is the address to set the least-significant bit of the video display address.

Follow the remaining hex bytes in the listing. You'll see **B7 FF C9**, meaning Store A Accumulator at **$FFC9**; **B7 FF CA**, Store A Accumulator at **$FFCA**; **B7 FF CC**, Store A Accumulator at **$FFCC**; and **39**. Check **$39** in the numerical instruction list on page 28 of the MC6809E data booklet. It's an opcode that will become very familiar — it is **RTS**, Return from Subroutine.

So the first group of bytes in line 9 of the BASIC program store the A Accumulator at **$FFC7**, **$FFC9**, **$FFCA** and **$FFCC**. A check of the SAM registers will show that these actions will place the binary value **0011** in bits 9, 10, 11 and 12 of the video address. Bits 13, 14, and 15 (the most signficant bits) are all zero, because that's where they were established when the computer was turned on. The full result of this short subroutine, then is to create the video address **0000 0110 0000 0000**. I'll translate that for you. It's address **$0600**, the address of the first screen the BASIC program POKEd into memory. By analyzing each of lines 9 through 15, you will see that the video display addresses created are **$0600**, **$0800**, **$0A00**, and so forth.



These seven short machine-language subroutines, then, are a quick version of the BASIC POKEs that were used to redirect the screen in the previous program. The speed here, however, is too fast to see. How fast is it? Glad I asked that. Flip to page 31 in the MC6809E data booklet, and look up the mnemonic **STA**. Under the heading "Extended", you'll find the opcode **$B7**. The next column tells you that a Store A Accumulator Extended takes five clock cycles. There are four Store A Accumulator instructions in each video display switching subroutine, meaning a total of 20 clock cycles. The **RTS** (Return from Subroutine) takes 5 clock cycles. The whole subroutine takes 25 clock cycles. At your Color Computer clock rate of 894,886 clock cycles per second, that means the subroutine is finished with its work in .00002794 seconds — 30 millionths of a second, about the time it takes the electron beam to sweep halfway across the TV screen.

I want to close a knowledge gap now. Obviously I've been talking about machine language subroutines in this BASIC program. BASIC puts those subroutines into memory in a very clumsy way. Look at the program listing. In lines 9 through 15 are a series of BASIC DATA statements in which the hexadecimal numbers are treated as strings. In line 16, I have variable X select the memory area to be used; in this case it's 16293 to 16383, hexadecimal addresses **$3FA5** to **$3FFF**.

The next step has the hexadecimal byte masquerading as a two-character ASCII string read as variable A$. BASIC identifies hexadecimal by the symbol "&H", so "&H" is

LET A$= "C3"
LET B=
""&H""+A$
(B$ BECOMES &HC3)
VAL(B$) is VAL(&HC3)
VAL(B$)=
195

USRØ - USR9
at
POWER UP

USRØ	$B44A
USR1	$B44A
USR2	$B44A
USR3	$B44A
USR4	$B44A
USR5	$B44A
USR6	$B44A
USR7	$B44A
USR8	$B44A
USR9	$B44A

DEFUSR1=
16293

USR Ø	$B44A
USR 1	$3FA5
USR 2	$B44A
USR 3	$B44A
USR 4	$B44A
USR 5	$B44A
USR 6	$B44A
USR 7	$B44A
USR 8	$B44A
USR 9	$B44A

DEFUSR 2=
16306

USRØ	$B44A
USR 1	$3FA5
USR 2	$3FB2
USR 3	$B44A
USR4	$B44A
USR 5	$B44A
USR 6	$B44A
USR 7	$B44A
USR 8	$B44A
USR 9	$B44A

concatenated with each two-character ASCII string. In this way, BASIC can be tricked into taking the value of the string, and that value can then be POKEd into memory. All that happens in line 16. Seven machine-language subroutine entry points are established in lines 17 through 23. Extended Color BASIC allows ten entry points altogether named USR0 through USR9; this program defines USR1 through USR7 for the seven screens to be displayed. Finally, lines 24 through 41 execute these subroutines in a fancy series of FOR-NEXT loops, and delay appropriately. By changing the order of the loops, you can make the seven messages flicker and flash in a variety of ways.

Here's a recap: Seven 512-byte screens are created in the memory below the BASIC program, allocated by PCLEAR4. These screens are displayed by machine-language subroutines that switch the video display registers in the SAM. I hope this hybrid BASIC / machine-language program gives you some ideas for effective but simple program displays.

As for the knowledge gap, the technique for creating short machine-language programs and POKEing them into memory via BASIC is something you can use often. Write the program, either byte-by-byte or using an editor/assembler. Take the hexadecimal opcodes and operands in the order they will appear in memory, and put the values into a bunch of BASIC DATA statements. Read each value, convert it to a number BASIC can use, and POKE it into memory. By using the DEFUSR command, define where your program will begin execution. From that point on, it only takes a USR command to execute your machine-language program. Review the program you've just run until you understand how that's done.

Before I leave this program, please load the mnemonic source code that follows.

Program #17, 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.

```
3FA5                  00100         ORG      $3FA5
3FA5 B7      FFC7     00110 SCRN1   STA      $FFC7
3FA8 B7      FFC9     00120         STA      $FFC9
3FAB B7      FFCA     00130         STA      $FFCA
3FAE B7      FFCC     00140         STA      $FFCC
3FB1 39               00150         RTS
3FB2 B7      FFC6     00160 SCRN2   STA      $FFC6
3FB5 B7      FFC8     00170         STA      $FFC8
3FB8 B7      FFCB     00180         STA      $FFCB
3FBB B7      FFCC     00190         STA      $FFCC
3FBE 39               00200         RTS
3FBF B7      FFC7     00210 SCRN3   STA      $FFC7
3FC2 B7      FFC8     00220         STA      $FFC8
```

* How long is that?

5 times 1.11746, or 5.5873 microseconds.

* How many STA extendeds is that per second?

1000000 microseconds divided by 5.5873 per STA extended instruction, or roughly 179,000 per second.

* BASIC can perform roughly 68 POKEs per second. How much faster is the machine language equivalent of STA extended?

179,000 divided by 68, or about 2,632 times faster.

* What is the standard symbol for hexadecimal?

The dollar sign ($).

* What is the BASIC symbol for hexadecimal?

The symbol ampsersand plus the letter H (&H).

* What command is used for a BASIC machine language entry point?

USR.

* BASIC needs to know the starting point of a machine language program. How does it get it?

With the DEFUSR command.

* What does DEFUSR3=&H3FBF mean?

It means that the entry point (execution address) for USR routine number 3 is at location $3FBF.

* Write a statement that informs BASIC that machine language program #7 begins at $3FF3.

DEFUSR7=&H3FF3.

* What is BASIC's representation of hexadecimal?

Ampersand plus H (&H).

* If variable C$ is A9, write a statement to make C equal to the hexadecimal value of C$.

C = VAL("&H"+C$)

* What is hand assembly?

Figuring the hex (binary) code byte by byte from the mnemonic (source) code.

* Hand assemble STA $FFC7 into hex, and then binary, code.

STA $FFC7 becomes $B7 FF C7, which becomes 10110111 11111111 11000111.

* What addressing mode is this?

Extended addressing.

* How many bits represent an address?

16 bits.

* How many hexadecimal characters is this?

4 hex characters.

* How many bits represent the memory contents at an address (the data)?

8 bits.

* How many hex characters is this?

2 hex characters.

* What is the value 00010010 in hexadecimal?

$12

* What are the ASCII values for "1" and "2"?

$31 and $32.

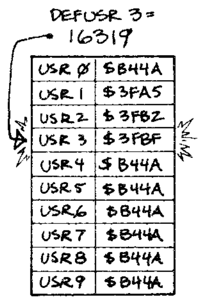
```
3FC5 B7    FFCB      00230          STA      $FFCB
3FC8 B7    FFCC      00240          STA      $FFCC
3FCB 39              00250          RTS
3FCC B7    FFC6      00260 SCRN4    STA      $FFC6
3FCF B7    FFC9      00270          STA      $FFC9
3FD2 B7    FFCB      00280          STA      $FFCB
3FD5 B7    FFCC      00290          STA      $FFCC
3FD8 39              00300          RTS
3FD9 B7    FFC7      00310 SCRN5    STA      $FFC7
3FDC B7    FFC9      00320          STA      $FFC9
3FDF B7    FFCB      00330          STA      $FFCB
3FE2 B7    FFCC      00340          STA      $FFCC
3FE5 39              00350          RTS
3FE6 B7    FFC6      00360 SCRN6    STA      $FFC6
3FE9 B7    FFC8      00370          STA      $FFC8
3FEC B7    FFCA      00380          STA      $FFCA
3FEF B7    FFCD      00390          STA      $FFCD
3FF2 39              00400          RTS
3FF3 B7    FFC7      00410 SCRN7    STA      $FFC7
3FF6 B7    FFC8      00420          STA      $FFC8
3FF9 B7    FFCA      00430          STA      $FFCA
3FFC B7    FFCD      00440          STA      $FFCD
3FFF 39              00450          RTS
           0000      00460          END
00000 TOTAL ERRORS
SCRN1    3FA5
SCRN2    3FB2
SCRN3    3FBF
SCRN4    3FCC
SCRN5    3FD9
SCRN6    3FE6
SCRN7    3FF3
```

Type A/NO and hit <ENTER>. Lines of information scroll by. The incredible thing about this mnemonic source code — and most mnemonic source code — is that it looks so massive. Here are 36 lines of typing, 7 labels, 8 columns wide, practically filling a page. And yet all this resolves into a mere 91 bytes of actual program, little more than a third of what a BASIC program line can hold.

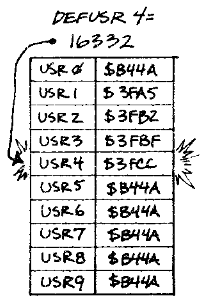
Since I knew precisely what I wanted, and since this program was so short and consistent, I actually figured out the hex code byte by byte using the MC6809E data booklet. Later I typed this source code for you. But in doing the hand programming, I had to keep track of where each subroutine began. The nice part about an editor/assembler is that whatever you have in mind can be typed and examined easily, even if it seems long. The editor/assembler picks up typing errors, whereas hand assembling each byte can be a highly error-prone procedure. Plus, by liberally scattering labels in the code, critical addresses can be identified; in fact, the assembler provides a complete display of all labels at the end of the assembled listing. Which teaches you more? My vote is for hand assembly. I'll help you with some of that.

For hand assembly you'll need paper and pencil, plus your MC6809E data booklet open to pages 30 and 31. The problem will turn away from flashy video displays for awhile; here it is:

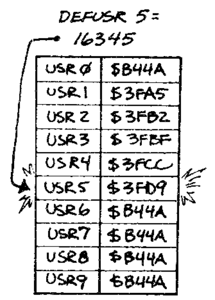
Given an address transferred from a BASIC program, create a display which will present eight lines of information. The first line will contain the address and eight hexadecimal bytes of memory contents separated by spaces. If the address is **$2000**, for example, the display


DEFUSR 3=
16319

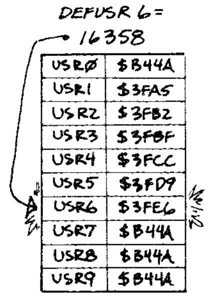
USR Ø	$B44A
USR 1	$3FA5
USR 2	$3FB2
USR 3	$3FBF
USR 4	$B44A
USR 5	$B44A
USR 6	$B44A
USR 7	$B44A
USR 8	$B44A
USR 9	$B44A


DEFUSR 4=
16332

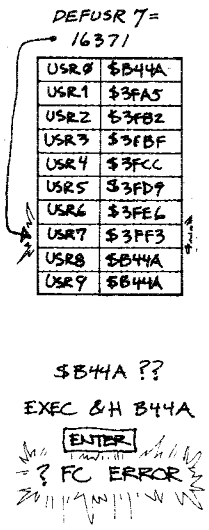
USR Ø	$B44A
USR 1	$3FA5
USR 2	$3FB2
USR 3	$3FBF
USR 4	$3FCC
USR 5	$B44A
USR 6	$B44A
USR 7	$B44A
USR 8	$B44A
USR 9	$B44A


DEFUSR 5=
16345

USR Ø	$B44A
USR 1	$3FA5
USR 2	$3FB2
USR 3	$3FBF
USR 4	$3FCC
USR 5	$3FD9
USR 6	$B44A
USR 7	$B44A
USR 8	$B44A
USR 9	$B44A


DEFUSR 6=
16358

USR Ø	$B44A
USR 1	$3FA5
USR 2	$3FB2
USR 3	$3FBF
USR 4	$3FCC
USR 5	$3FD9
USR 6	$3FE6
USR 7	$B44A
USR 8	$B44A
USR 9	$B44A



DEFUSR 7=
16371

USRØ	$B44A
USR1	$3FA5
USR2	$3FB2
USR3	$3FBF
USR4	$3FCC
USR5	$3FD9
USR6	$3FE6
USR7	$3FF3
USR8	$B44A
USR9	$B44A

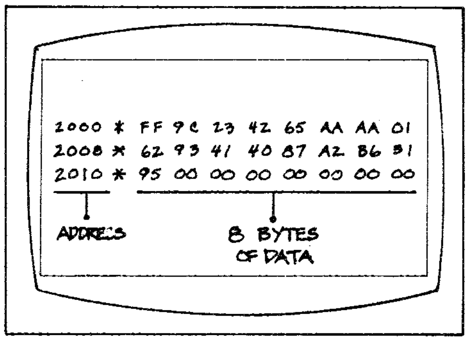
$B44A ??

EXEC &H B44A

ENTER

? FC ERROR

should print **$2000** followed by the data found in memory locations **$2000**, **$2001**, **$2002**, etc., up to **$2007**. In the next line, the address **$2008** would be displayed, together with the memory data found at **$2008** through **$200F**. And on down for a total of eight lines. Ready?



```
2000 * FF 9C 23 42 65 AA AA 01
2008 * 62 93 41 40 87 A2 B6 31
2010 * 95 00 00 00 00 00 00 00
```

Know how to tell if you're ready? Think about Session 8, where I presented a dozen machine language instructions and showed how they worked, including how flags were affected. If that's not clear and reasonably fresh in your mind, review it now. When those instructions make sense to you, you're ready to move on.

The problem at hand is to transfer an integer from BASIC which represents an address in memory you'd like to examine. That examination will display 8 lines, each line containing one address and 8 consecutive bytes of memory data. In all, 64 bytes of data will be displayed. First, conceptualize the problem. Information in integer form is to be transferred to the machine-language program. That part is easy; the USR function is used, with the target address being the operand in parentheses. You've already used the integer-conversion routine from the BASIC ROM in order to retrieve a value from BASIC for your machine-language program's use, so that's easy.

Once you've got the integer value in your own program, two things need to be done. First, it has to be treated as displayable information. The address must be converted to four ASCII characters for presentation as a hexadecimal display. Second, the integer has to be treated as the address itself in order to retrieve the memory information for display.

How about an integer-to-ASCII conversion routine, then? You'll want to break it down into simple modules, if possible. Start by looking for modularity, small consistent units that you can program. What you know you have are 16 binary digits which you want to represent on the screen as four ASCII characters in hexadecimal notation. There's a clue there. 16 binary digits. Four ASCII characters. You already know that a single hexadecimal number represents four binary digits. The solution lies in that knowledge: treat each four-bit group as an identical task. A single subroutine.

* What is the value of 11001101 in hexadecimal?

$CD

* What are the ASCII values for "C" and "D"?

$43 and $44.

* What is the value 10001110 in hexadecimal?

$8E

* What are the ASCII values for "8" and "E"?

$38 and $45.

* An address is $A0D7. What are the four ASCII values (A, 0, D and 7)?

$41, $30, $44 and $37.

* What is the ASCII value for a space?

$20.

* To display the address $A0D7, a space, and the contents of $A0D7 (which is $8E), what ASCII values must be used?

$41 30 44 37 20 38 45

* Where are these ASCII values placed?

In display memory.

* Where is display memory on the normal Color Computer?

From $0400 to $05FF.

* How many bytes is the value $8E?

One byte, $8E.

* How many bytes are the ASCII values needed to represent the value $8E?

Two bytes, $38 and $45.

* How many bytes is the address $A0D7?

Two bytes, $A0 and $D7.

* How many bytes are the ASCII values needed to represent the value $A0D7?

Four bytes, $41, $30, $44 and $37.

* What are the ASCII values for the characters "0" through "9"?

$30 through $39.

* What are the ASCII values for the characters "A" through "F"?

$41 through $46.

* What is the number $8E in binary?

$8E in binary is 1000 1110.

* In the number $8E, which bits represent the number 8?

The leftmost four bits.

* In the number $8E, which bits represent the number E?

The rightmost four bits.

* What are the leftmost and rightmost four bits of $8E?

1000 and 1110

* What are the binary values for 8 and E?

0000 1000 and 0000 1110.

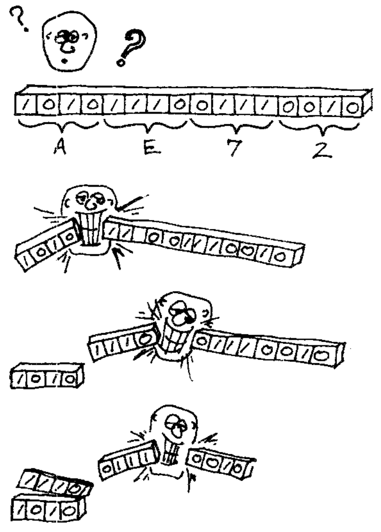
* What are the binary values for ASCII "8" and ASCII "E"?

0011 1000 and 0100 0101.

* What is the difference between binary 8 and ASCII "8"?

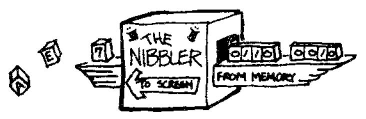
Binary 8 is 0000 1000 and ASCII "8" is 0011 1000; the difference is 0011 0000, or $30.

That line of thinking brings you one step closer to a modular approach. Each time you have four bits in hand, you can call the subroutine that creates an ASCII character from them. Now you need only sketch out that subroutine. Recall a few sessions ago how, in order to access a table of encrypted codes, a constant value had to be subtracted from the ASCII characters to obtain numbers starting from zero. In this case, you have a complementary situation. You have four binary digits equivalent to the hexadecimal numbers **0** through **F**. In order to produce ASCII characters, then, it's necessary to *add* a constant value. To display the number zero as the *character* 0 with the ASCII value of hex **$30**, you would add hex **$30**. To display the number one as the character 1 with the ASCII value **$31**, again you would add **$30**. You would do that right up through number nine which is displayed as the character 9, ASCII value **$39**. The constant you add is **$30**.

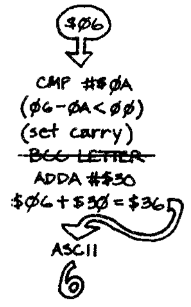


So far so good. But when you get to number A, you're in a little trouble. Binary **1010** is number **A**. Character A is ASCII value hex **$41**. The constant you must add to number A to get character A is hex **$37**. It's consistent from A through F — add **$37** to the value and you get the ASCII character.

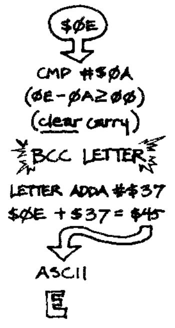
How do you reconcile the two different constants? The answer is simple: you don't. You find out whether the value is **0** through **9** or **A** through **F**, and add the constant **$30** or **$37** accordingly.



That looks like enough information for a subroutine. The "entry condition", as it's called, is a group of four binary digits. That four-bit number is checked to see whether it is greater or less than 9. If it's greater than 9, you add the constant **$37**; if it's 9 or less, you add the constant **$30**. The result is an ASCII character which, when displayed, represents the hexadecimal numerical value. The ASCII character is the subroutine's "exit condition". The nice part about a subroutine like this is its versatility — not only can it be used to display the digits of an address, it's just as good for displaying the bytes of memory data.



Mnemonically speaking, that would operate like this. The A Accumulator enters with the four-bit number. It's compared immediate with **$0A**. If the number is greater than nine, the carry/borrow flag would not be set. The program would Branch on Carry Clear to an instruction to add **$37** and then return from subroutine; otherwise it would add **$30** and return from subroutine. The A Accumulator enters with the number and exits with the ASCII character. Pretty slick.



It would look like this, assuming the A Accumulator holds the four-bit number:

```
CONVRT  CMPA    #$ØA
        BCC     LETTER
        ADDA    #$3Ø
        RTS
LETTER  ADDA    #$37
        RTS
```

Now there's the task of breaking the 16-bit address into four 4-bit groups. Half of that's done already, since the 16-bit address is split into two 8-bit bytes. Creating this subroutine from there demands just a little convoluted thinking.

You have 8 bits. You only want to use four bits at a time, and these four bits have to be in the least-significant positions. In other words, if the number is **$3C**, you want to convert the four bits **0011** into a **3**, and the four bits **1100** into a **C**. The least-signficant four bits of the byte are just about ready to use. All that remains is to temporarily get rid of the most-significant four bits. The term is "mask" the bits, meaning create a mask so that only the bits you need show through.

The mask here is AND. Recall how the AND instruction works. Both conditions must be a one for the result to be a one. To mask out the four leftmost bits of the byte, then, you would AND each of those four bits with zero. To mask IN the four rightmost bits you would AND each of those four bits to one. I'll repeat that a different way. If the leftmost four bits are ANDed with zero, no matter what those bits are, the result of the ANDing will be zero. If the rightmost four bits and ANDed with one, no matter what those bits are, they will effectively remain the same.

$3C=
00111100
AND 00001111
00001100
=$ØC

Scratch it out on paper and look at it. Use the example **$3C** that I just mentioned. Write down the binary equivalent: **0011 1100**. Underneath it, write down the mask: **0000 1111**. Now use the AND function:

```
Ø AND Ø is Ø
Ø AND Ø is Ø
1 AND Ø is Ø
1 AND Ø is Ø
```

That's the leftmost four bits. Now the rightmost:

```
1 AND 1 is 1
1 AND 1 is 1
Ø AND 1 is Ø
Ø AND 1 is Ø
```

There are the rightmost four bits. The mask to use here is **$0F**. To recap: to retrieve the least-signficant four bits of a byte, use the mask **$0F**.

You can pause here to review that section if you like.

The next task is to retrieve the leftmost four bits. If logic holds, then you can again use a mask. Since the bits you want are to the left, then the mask **1111 0000** should suffice. That's **$F0**; it will result in the four leftmost bits being masked *in*, and the four rightmost bits being masked *out*.

There's a problem, though. Although it masks in the bits you want, they're not in the correct place. You need them on the *right* side of the byte to represent the 4-bit numbers **$0** through **$F**. You have to get those bits from left to right.

*** What is the difference between binary E and ASCII "E"?**

Binary E is 0000 1110 and ASCII "E" is 0100 0101; the difference is 0100 0000, or $37.

*** What is the constant difference between binary values 0 through 9 and ASCII values "0" through "9"?**

The constant difference is $30.

*** What is the constant difference between binary values A through F and ASCII values "A" through "F"?**

The constant difference is $37.

*** What logical function states: both of two conditions must be true for the result to be true?**

The AND function.

*** How are the rightmost four bits retrieved from the number $0E (0000 1110)?**

By masking the leftmost four bits.

*** What mask is used?**

AND 00001111.

*** If A contains $0E, what mnemonic command is used to retrieve the rightmost four bits?**

ANDA #$0F (AND A accumulator immediate with $0F, binary 00001111).

*** What constant is added to $0E to produce the ASCII character "E"?**

$37.

*** How are the leftmost four bits retrieved from the number $8E (1000 1110)?**

By shifting the bits right four times.

*** When $8E is shifted right once, what is the result (in hex and binary)?**

0100 0111 ($47).

*** When $8E is shifted right twice, three times, and four times, what are the results (in hex and binary)?**

0010 0011 ($23), 0001 0001 ($11) and 0000 1000 ($08).

*** What constant is added to $08 to produce the ASCII character "8"?**

$30.

*** What is necessary to convert the least significant half of a byte to a 4-bit number?**

Masking with $0F.

*** What is necessary to convert the most significant half of a byte to a 4-bit number?**

Rotating right four times.

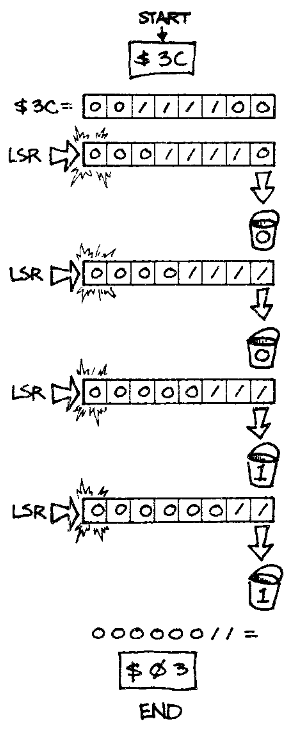
*** What is necessary to convert a 4-bit binary number to a hexadecimal ASCII character?**

The addition of a constant.

Recall the various rotate and shift commands from an earlier session. You'll need to refer to your MC6809E data sheet to choose the particular rotate or shift you want; open to pages 30 and 31.

You know that you need to move these bits to the right. Your choices are ASR (arithmetic shift right), LSR (logical shift right), and ROR (rotate right). Look at each one. ASR reproduces the leftmost bit each time you shift, so this doesn't look very good. If you shifted first and masked second, it would work. How about LSR? It shifts right and brings zeros in from the left as it shifts. That one looks good. Finally, ROR swings the bits 'round from the other side of the byte, so you would need to mask the results afterward.

The logical shift right (LSR) looks the best. In fact, it looks excellent. Since the bits shifted out the right side end up in the bit bucket, and zeros come in from the left, you don't even have to bother masking this before you use it. The process of shifting it right gives you not only the four bits you need, but eliminates those you don't want.



Here's a summary of these two program segments: the byte is to be displayed as two hexadecimal ASCII characters. The leftmost four bits are obtained by logically shifting the byte right four times. The rightmost four bits are obtained by masking the original byte with **$0F**. All that remains is to make sure the original value is saved before modifying it. Push A Accumulator will take care of saving the byte, and Pull A Accumulator will get it back when it's needed. In terms of mnemonics, and assuming the value to be displayed is in the A Accumulator, the complete routine would look like this:

```
BYTBIT  PSHS   A       Push A Accumulator onto stack
        LSRA           Logical Shift Right A Accumulator
        LSRA           Logical Shift Right A Accumulator
        LSRA           Logical Shift Right A Accumulator
        LSRA           Logical Shift Right A Accumulator
        JSR    CONVRT  Jump to ASCII conversion subroutine
        JSR    DISPLY  Jump to screen display subroutine
        PULS   A       Pull A Accumulator from stack
        ANDA   #$ØF    AND A Accumulator immediate with $ØF
        JSR    CONVRT  Jump to ASCII conversion subroutine
        JSR    DISPLY  Jump to screen display subroutine
```

At this point, two major portions of the problem have been solved: the 8-bit byte has been converted to two 4-bit numbers, and those 4-bit numbers have been converted to ASCII characters. The screen display routine has yet to be done. I'll leave you with these considerations: your program has to know where to start the screen display in memory, that is, it has to be initialized. The current screen display position has to be updated so that the next character displayed will appear in the next available position.

Review this lesson, and consider those problems for next time.

11.

The topic is hand assembly. Last time I started you working on a program to display memory locations and their contents. At the end of the session, you had produced two pieces of that program: the byte-to-nybble conversion routine (a nybble is four bits), and the hexadecimal-to-ASCII conversion routine. The byte-to-nybble conversion was made up of two steps. To move the most-significant nybble into the righthand portion of the byte, the byte was logically shifted right four times. To obtain the least-significant nybble, a mask of **$0F** was ANDed with the value of the byte.

The problem I posed at the end of the session was this one: create a single-character display subroutine that, when called, places a character in the correct location on the screen and updates the program to point to the next available screen location.

To help solve this, I hope you thought back to the message-display program you created in the third session. There wasn't much to that display routine, and there isn't much to this one either. At the beginning of this program, then, you would initialize the first screen location, perhaps in the Y register. Each Color Computer screen line is 32 characters long — that's hex **$20**. So to start on the fourth line of the screen, you would load the Y register with the immediate value of **$0480** at the start of the program:

```
LDY     #$Ø48Ø
```

is the mnemonic. If the ASCII value to be displayed is the A Accumulator, and the Y register points to the current location on the screen, then you would store the A Accumulator in memory — display memory, that is — indexed by Y. To update that location, choose the auto-increment/decrement zero-offset indexed mode. You remember that mouthful. That's Store A Accumulator at memory indexed simply by Y, auto-increment Y by one, and then return from subroutine. Label it DISPLY:

```
DISPLY  STA  ,Y+
        RTS
```

Hand assembly really hasn't gotten underway yet. At this point, the program is still being structured and converted into mnemonic source code. So far, a complete byte-to-ASCII conversion system has been developed. What's to come is a display routine, plus a kind of executive structure.

* What is the location of the normal display screen on the Color Computer?

$0400 to $05FF.

* Each line of the display is 32 characters long. What line starts at $0480?

If $0400 is the start of the first line, then $0480 is the start of the fifth line.

* If the Y register points to screen location $0480 and the A register contains the ASCII value, what mnemonic instruction would place the ASCII value on the screen?

STA ,Y

* What mnemonic instruction would place the ASCII value on the screen, and automatically move the Y pointer register to the next screen position?

STA ,Y+

* Write two instructions that, given the conditions just used, create a complete ASCII display and screen update routine.

```
STA  ,Y+
RTS
```

* What does STA ,Y+ mean?

Store A accumulator to memory indexed by the Y register, with no offset, and automatically increment Y.

* Given that A contains $2A and B contains $20, what do the following four instructions do?
STB ,Y+
STA ,Y+
STA ,Y+
STB ,Y+

The four instructions display space, star, star, space.

* What does JSR $B3ED identify on the Color Computer?

An integer conversion subroutine in the BASIC ROM.

* What are the results of JSR $B3ED?

A 16-bit signed integer is found in the D register.

* What does integer mean?

A number without a fractional (or decimal) part; a whole number.

* What does "signed integer" mean?

It means the number is positive or negative.

* How is the sign indicated?

By the leftmost bit; 0 is positive, 1 is negative.

* In the display program, how is the sign information used?

It isn't. The number is treated as a 16-bit unsigned integer.

* In the program, the instruction STA ($0001 appears. What addressing mode is this?

Direct addressing.

* In the program, the instruction LDA #$2A appears. What addressing mode is this?

Immediate addressing.

* In the program, the instruction JSR $B3ED appears. What addressing mode is this?

Extended addressing.

* In the program, the instruction BNE LLOOP appears. What does BNE LLOOP mean?

It means Branch Not Equal to the instruction labeled in the source listing "LLOOP".

That should do the trick. A short, sweet 3-byte subroutine that illustrates the power of the 6809 processor.

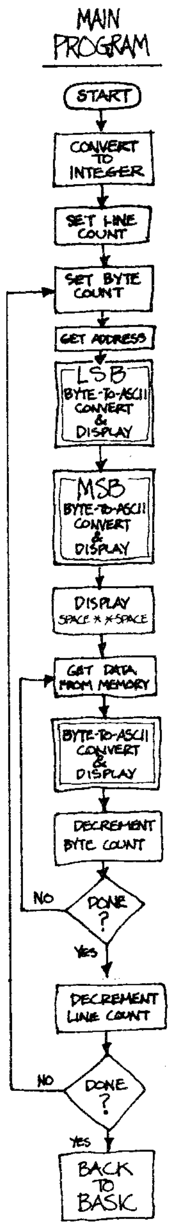
That seems to cover the necessary subroutines — conversion and display. What's left to create is a kind of executive program which accepts the address from BASIC, searches for the memory data, and calls the subroutines you've just created. This executive's job would be to call for the value from BASIC, initialize the screen parameters, do the screen line and screen character counting, call the convert and display subroutines, and return to BASIC when all is done.

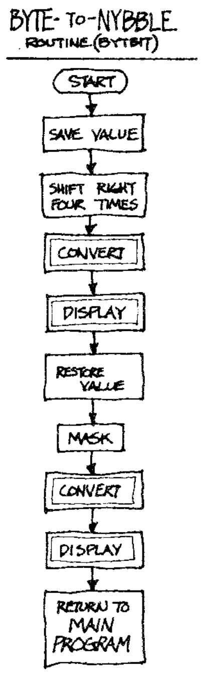
The sequence as I see it comes out to 15 steps:

1. Get the target address from BASIC
2. Initialize the screen starting position
3. Initialize the line and character counts — 8 lines, memory bytes per line
4. Convert and display the most-significant byte of the memory address
5. Convert and display the least-significant byte of the memory address
6. Display a space as a separator
7. Display two stars or other separators
8. Display another space as another separator
9. Get the memory contents of the address
10. Convert and display that memory byte
11. Display another space as a divider
12. Increment the target address
13. Loop for 7 more memory bytes, for a total of 8
14. Loop for 7 more lines of address, for a total of 8
15. And finally, return to BASIC

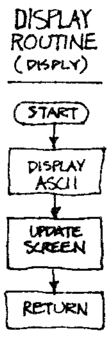
I've prepared a program that follows these steps; open to your documentation and follow along. The program is in mnemonics, which you will be hand-assembling. I'll explain each line briefly; those which you haven't already written should fall into place.

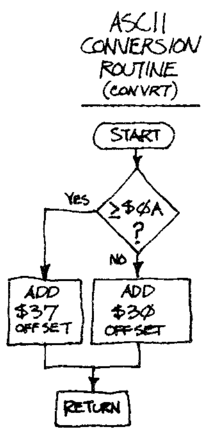
```
---------------------------------------------------------------
        JSR   $B3ED      BASIC INTEGER-CONVERT ROUTINE
        LDY   #$0480     FIRST SCREEN LOCATION TO USE
        TFR   D,X        GIVE INT-CONV RESULT TO X REG
        LDA   #8         PUT 8 LINES INTO ACCUMULATOR
        STA   <0001      LINE COUNT INTO DIR. PAGE 01
...............................................................
LLOOP   LDA   #8         PUT 8 BYTES INTO ACCUMULATOR
        STA   <0000      BYTE COUNT INTO DIR. PAGE 00
        TFR   X,D        INT-CONV RESULT BACK TO D REG
        JSR   BYTBIT     BYTE-TQ-ASCII CONV. & DISPLAY
        TFR   B,A        MOST SIGN. BYTE INTO A ACCUM.
        JSR   BYTBIT     BYTE-TO-ASCII CONV. & DISPLAY
        LDA   #$2A       PUT ASCII FOR "*" INTO A ACC.
        LDB   #$20       ASCII FOR SPACE INTO B ACCUM.
        STB   ,Y+        DISPLAY SPACE, GET NEXT POSN.
        STA   ,Y+        DISPLAY STAR, GET NEXT POSN.
        STA   ,Y+        DISPLAY STAR, GET NEXT POSN.
        STB   ,Y+        DISPLAY SPACE, GET NEXT POSN.
```





```
BLOOP   LDA    ,X+        GET MEMORY CONTENTS X-INDEXED
        JSR    BYTBIT     BYTE-TO-ASCII CONV. & DISPLAY
        STB    ,Y+        DISPLAY SPACE, GET NEXT POSN.
        DEC    <ØØØ1      DECREMENT NUMBER OF BYTES
        BNE    BLOOP      REPEAT UNTIL ALL 8 DISPLAYED
        DEC    <ØØØØ      DEC. NUMBER OF DISPLAY LINES
        BNE    LLOOP      REPEAT UNTIL ALL 8 DISPLAYED
        RTS               BACK TO BASIC WHEN ALL DONE
------------------------------------------------------------
BYTBIT  PSHS   A          SAVE BYTE STORED IN A ACCUM.
        LSRA              SHIFT TO RIGHT ONE BIT ....
        LSRA              .... AND SHIFT ONE MORE ....
        LSRA              .... AND SHIFT ONE MORE ....
        LSRA              .... TIL 4 BITS ARE AT RIGHT
        JSR    CONVRT     NYBBLE-TO-ASCII CONVERSION
        JSR    DISPLY     DISPLAY ASCII CHAR. & UPDATE
        PULS   A          RECOVER ORIGINAL BYTE STORED
        ANDA   #$ØF       MASK IN RIGHT-HAND NYBBLE
        JSR    CONVRT     NYBBLE-TO-ASCII CONVERSION
        JSR    DISPLY     DISPLAY ASCII CHAR. & UPDATE
        RTS               TWO CHARS. CONV'D & DIPLAYED
------------------------------------------------------------
CONVRT  CMPA   #$ØA       COMPARE NYBBLE AGAINST $ØA
        BCC    LETTER     IF CARRY CLEAR, A ACC. >= $ØA
        ADDA   #$3Ø       ELSE IS A NUMBER, SO ADD $3Ø
        RTS               CONVERSION COMPLETE; RETURN
        ....................................................
LETTER  ADDA   #$37       IT IS A LETTER, SO ADD $37
        RTS               CONVERSION COMPLETE; RETURN
------------------------------------------------------------
DISPLY  STA    ,Y+        DISPLAY ASCII, UPDATE SCREEN
        RTS               DIPLAYED & UPDATED; RETURN
------------------------------------------------------------
```





Now comes the time-consuming part. I want you to translate each one of these mnemonics into the binary opcodes and operands the computer will need to execute the program. I'm confident this program works — there are some anomalies, but you'll discover them soon enough — so open your MC6809E data booklet to pages 30 through 33.

Assume that the program will be stored in memory beginning at **$3F00**. Since some of you have 16K machines whose uppermost RAM address is **$3FFF**, this gives you 256 bytes of room for the program. I can tell you now that this program will occupy less than 100 bytes, and with some experience you'll be able to scope out program lengths like this one. One other assumption to make is the address of the Direct Page, which is **$00**; that information is provided in your EDTASM+ manual, in the memory map appendix, which also informs you that direct page addresses **$00** through **$7F** are free for your use.

For the hand assembly, you'll need several sheets of lined notebook paper, with the addresses **$3F00** through **$3F60** in a column down the left side. This is a good time to take a break for a review, and also to get the paper ready.

* What addressing mode is BNE LLOOP?

Relative addressing.

* In the program, the instruction STB ,Y+ appears. What addressing mode is this?

Indexed addressing (specifically, zero-offset indexed).

* In the program, the instruction LSRA appears. What addressing mode is this?

Inherent addressing.

* What is hand assembly?

Figuring the hex (binary) code byte by byte from the mnemonic (source) code.

* The following inherent instructions appear in the program. Hand assemble each:

* Hand assemble LSRA.

$44

* Hand assemble RTS.

$39

* The following immediate instructions appear in the program. Hand assemble each one:

* Hand assemble LDY #$0400.

$10 8E 04 00

* Hand assemble LDA #$00.

$86 00

* Hand assemble LDB #$20.

$C6 20

* Hand assemble ANDA #$0F.

$84 0F

* Hand assemble ADDA #$30.

$8B 30

* The direct instruction STA ($0001 appears in the program. Hand assemble it.

$97 01

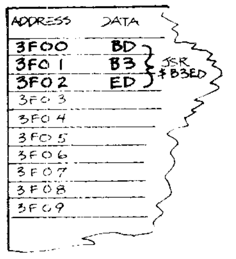
* The following register instructions appear in the program. Hand assemble each one:

* Hand assemble TFR D,X.

$1F 01

```
* Hand assemble TFR B,A.

$1F 98

* Hand assemble PSHS A.

$34 02

* Hand assemble PULS A.

$35 02

* The    following    indexed
instructions  appear  in  the
program.   Hand  assemble  each
one:

* Hand assemble STB ,Y+

$E7 A0

* Hand assemble STA ,Y+

$A7 A0

* Hand assemble LDA ,X+

$A6 80

* The   following   immediate
instructions  do  not  appear in
the program.  Hand assemble each
one.

* Hand assemble ADDD #$C3C3

$C3 C3 C3

* Hand assemble ANDCC #$AF

$1C AF

* Hand assemble CMPX #$05FF

$8C 05 FF

* Hand assemble CMPA #$FF

$81 FF

* Hand assemble EORA #$20

$88 20

* Hand assemble LDD #$BBAA

$CC BB AA

* Hand assemble ORB #$AC

$CA AC

* Hand assemble SUBA #$02

$80 02

* The    following    extended
instructions  do  not  appear in
the program.  Hand assemble each
one.

* Hand assemble ADDA $1000

$BB 10 00
```

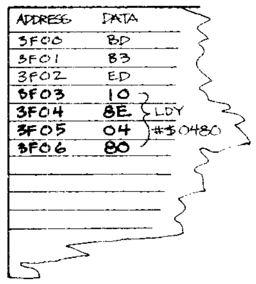
You should have your notebook paper ready, and your MC6809E data booklet open to page 30.



ADDRESS	DATA	
3F00	BD	JSR $B3ED
3F01	B3	
3F02	ED	
3F03		
3F04		
3F05		
3F06		
3F07		
3F08		
3F09		

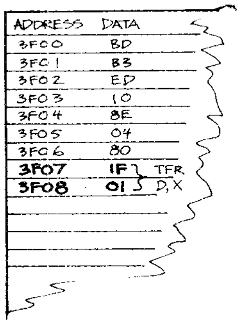
Start with the first instruction, JSR **$B3ED**. Find JSR on page 30. This is an extended addressing mode; the opcode you should find is **$BD**. On your paper, next to address **$3F00**, write **$BD**. At address **$3F01**, write the first byte of the operand, which is **$B3**. At address **$3F02**, write the second byte, **$ED**. You have hand-assembled the first instruction, **JSR $B3ED**, into three binary bytes, **$BD B3 ED**.

Your pencil should be poised above address **$3F03**, ready to assemble the instruction LDY immediate #$0480. Find mnemonic LD on page 30, and follow in the second column until you find LDY. This is one of a limited number of two-byte opcodes, and its hex representation is **$10 8E**. The 6809 is a newcomer, based on the 6800 microprocessor. Opcodes like LDY are additions to the original 6800 instructions; where there's no room to fit an opcode in the binary instruction set, certain bytes are set aside as doorways into further instructions. The hex codes **$10** and **$11** serve that purpose; later on, check page 29 for a list of these.



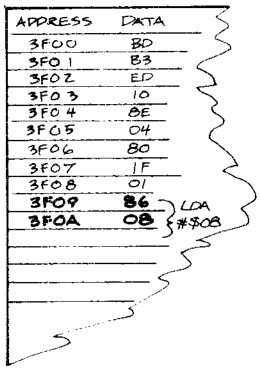
ADDRESS	DATA	
3F00	BD	
3F01	B3	
3F02	ED	
3F03	10	LDY
3F04	8E	
3F05	04	#$0480
3F06	80	

Back to the program. The opcode for LDY, then, is **$10 8E**. So across from address **$3F03**, write **$10, and across from address $3F04**, write **$8E**. Since this is an immediate instruction, the next two bytes are the operand. Next to addresses **$3F05** and **$3F06**, write the bytes **$04** and **$80**, respectively. You have now assembled the second program command.



ADDRESS	DATA	
3F00	BD	
3F01	B3	
3F02	ED	
3F03	10	
3F04	8E	
3F05	04	
3F06	80	
3F07	1F	TFR
3F08	01	D,X

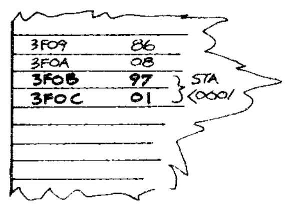
Those two were easy. The next instruction is **TFR D,X** (transfer D to X), which you can find on page 31. You'll find this in the immediate column, although that's stretching the point. The opcode is **$1F**, so write that next to address **$3F07**. The operand is D,X. Turn to page 34, where you'll find a block labeled "Transfer/Exchange Post Byte". This byte is divided into two four-bit blocks, that is, into two nybbles. The left-hand nybble is the source register, and the right-hand nybble is the destination register. The binary information below names the registers. Your program is transferring D to X. The source register is D, the destination register is X. Checking the table, you find that D is value **0000** and X is value **0001**. The combined byte is therefore **0000 0001**, or hex **$01**. Across from memory location **$3F08**, write **$01**. The opcode and operand for **TFR D,X** assemble to **$1F 01**.



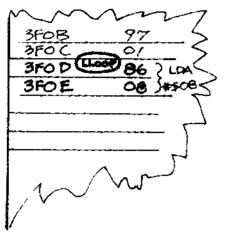
ADDRESS	DATA	
3F00	BD	
3F01	B3	
3F02	ED	
3F03	10	
3F04	8E	
3F05	04	
3F06	80	
3F07	1F	
3F08	01	
3F09	86	LDA
3F0A	08	#$08

Next. LDA immediate with 8. Back on page 30, under the LD instruction, you can find LDA. Since this is an immediate instruction, the opcode is **$86**. Next to address **$3F09**, write **$86**. The instruction is immediate, so the data is 8. Write **$08** across from address **$3F0A**. Things are moving now.

The instruction is STA Direct Page <0001. STA is found on page 31 under the instruction ST. This is a direct



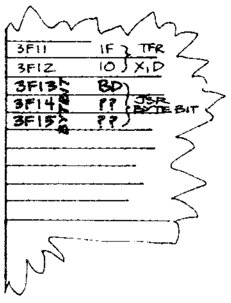
addressing mode, so the operand under the direct heading is **$97**. Write **$97** across from address **$3F0B**. In a direct instruction, the page is known, so only the least-significant byte is used as the operand. The address is **$0001** on page **$00**, so the least-significant byte is **$01**. That's the operand; write **$01** next to address **$3F0C**.



The next two instructions are virtually identical. LDA immediate 8 is again **$86 08**. Write **$86** next to **$3F0D**, and **$08** next to **$3F0E**. STA Direct Page <0000 is also very similar, assembling to **$97 00**. Write **$97** next to **$3F0F**, and write **$00** next to **$3F10**. The only thing to keep in mind is the label LLOOP, an abbreviation for Line Loop. Your program needs to come back to that address **$3F0D** each time it has to display a new line, so mark that label down on the bottom of the last page of your papers: write LLOOP, and across from it write the address **$3F0D**.

You're only 16 bytes into the program. I've already told you it will run nearly 100 bytes, so you're probably beginning to conclude that this assembly language stuff isn't for you. Hang on! The editor/assembler will do this all for you in seconds, but I'm convinced it won't do you any good to assemble everything by machine. There are two advantages to hand assembly: first, by the time you've hand assembled a program, you know it intimately. Second, if you're ever in a bind and need a quick diagnostic program, POKEing values into place may be the only solution. You have to be able to assemble a program from the data booklet, or you're wasting your time learning about this powerful 6809 processor.

Back to work. Transfer X to D — **TFR X,D**. The opcode you've used. Next to address **$3F11** write **$1F**, the transfer opcode. This time the source register is X and the destination register is D. If you've forgotten, turn to page 34. X register is binary **0001**, D register is binary **0000**. The composite byte made from these two nybbles is **0001 0000**, or hexadecimal **$10**. That's the operand. Next to address **$3F12**, write **$10**.



The next instruction is **JSR BYTBIT**. You've used the opcode for Jump to Subroutine (JSR) — that's **$BD**. Write **$BD** next to address **$3F13**. But how do you deal with the operand? You know it's an extended operand, which means it's two bytes. The subroutine BYTBIT is within the program you're writing, but you don't know its address yet. What you do now is leave two blank spaces at addresses **$3F14** and **$3F15**. You'll fill them in later when you know what they are. There are two pass-throughs to any assembly process, and this is the first pass.

The next free address is **$3F16**. The command is transfer, **$1F**. Write that next to **$3F16**. The transfer is from B to A. Again, turn to page 34. The source register is B, binary nybble **1001**; the destination register is A, binary nybble **1000**. The combined byte is **1001 1000**, or hex **$98**. Next to address **$3F17**, write **$98**.

```
* Hand assemble CMPB $FFFF

$F1 FF FF

* Hand assemble EORB $0001

$F8 00 01

* Hand assemble JMP $B3ED

$7E B3 ED

* Hand assemble LDX $7FFF

$BE 7F FF

* Hand assemble LDY $7FFF

$10 BE 7F FF

* Hand assemble LSR $0100

$74 01 00

* Hand assemble STD $0DDC

$FD 0D DC
```

* The following inherent instructions do not appear in the program. Hand assemble each one.

```
* Hand assemble ASRA

$47

* Hand assemble CLRB

$5F

* Hand assemble COMA

$43

* Hand assemble INCB

$5C

* Hand assemble LSLB

$58

* Hand assemble NEGA

$40

* Hand assemble RORA

$46

* Hand assemble RTS

$39
```

* The following register instructions do not appear in the program. Hand assemble each one.

```
* Hand assemble PULS A,CC,X,Y

$35 35
```

* Hand assemble PSHS A,B,X,Y,CC,U,DP,PC

$36 FF

* Hand assemble TFR DP,B

$1F B9

* The following indexed instructions do not appear in the program. Hand assemble each one.

* Hand assemble CMPA ,Y

$A1 A4

* Hand assemble CMPA ,Y+

$A1 A0

* Hand assemble CMPA 5,Y

$A1 25

* Hand assemble CMPA $7F,Y

$A1 A8 7F

* Hand assemble CMPA $1234,Y

$A1 A9 12 34

* What does CMPA ,Y+ mean?

Compare A accumulator to memory indexed by the Y register, with no offset, and automatically increment Y.

* What is hand assembly?

Figuring the hex (binary) code byte by byte from the mnemonic (source) code.

Another JSR to BYTBIT is next. Write the opcode for JSR, hex **$BD**, next to address **$3F18**, and leave blank spaces at **$3F19** and **$3F1A**. Again, when you find out where the subroutine BYTBIT is, you'll fill those in.

A LDA immediate is next. That instruction's been used before; the opcode is **$86**, the operand here is an immediate value, **$2A**. Write **$86** and **2A** next to addresses **$3F1B** and **$3F1C**, respectively.

LDB is a similar opcode to LDA. You'll find it right below; LDB immediate is **$C6**. Write **$C6** next to address **$3F1D**, and write its immediate operand, **$20**, next to address **$3F1E**.

On to **STB ,Y+**. Find the ST instructioon on page 31, and locate STB in the indexed addressing mode. The opcode is **$E7**. Next to address **$3F1F**, write **$E7**. In the column labeled "number of bytes", it says "2+", meaning this instruction requires a total of 2 or more bytes to complete. You have to determine how many and what they mean. Hand-assembling indexed addressing is the trickiest, but zero-offset indexed isn't bad. That's what you have here.

Turn to page 33. Find the table entitled "Indexed Addressing Postbyte Register Bit Assignments". This one byte contains a bucketful of information. It identifies the register, what kind of addressing mode is used with that register, and whether the addressing is non-indirect or indirect. I haven't talked about indirect addressing, so don't worry about that yet. In the right-hand column of this table is a description of each addressing mode; "EA" means effective address, that is, the address the instruction will calculate and use. The mode used in this instruction is auto-increment, zero-offset. That's the second mode down. The definition of "RR" is shown below the table. Your instruction uses the Y register, so RR is 01. Plug 01 into the binary digits shown, and the resulting number is **10100000**. The postbyte for the Y register in zero-offset indexed, auto-increment mode is hex **$A0**. There's your operand. Next to address **$3F20**, write **$A0**.

Between now and the next session, use your MC6809E data booklet to complete the rest of the program. If the process is still unclear, review the session up to this point. Don't cheat on me, now. When you can do this hand assembly without your hand held by me, then you're ready to go on. Talk to you then.

12.

Hello again. I hope you have been successful in your hand assembly of the remainder of the program. Here's a summary of what you should have been doing. . .

The next three instructions are easy. STA indexed is **$A7**. Write **$A7** next to address **$3F21**. The operand is zero-offset indexed; auto-increment Y register is the same as before. Across from address **$3F22**, write **$AO**. The following instruction is the same, **$A7 AO**. Write **$A7 AO** next to **$3F23** and **$3F24**, respectively. Finally, **STB ,Y+** comes around again. You know that's **$E7 AO**, so write **$E7 AO** next to **$3F25** and **$3F26** in turn.

Since you can use the table on page 33, the next instruction should strike no fear. It's **LDA ,X+**. Load A indexed, from page 30, is **$A6**. Write **$A6** next to address **$3F27**. Now glance at the chart on page 33. This is still auto-increment indexed, which is the second line of the table. The register is X, meaning the value for "RR" is 00. Plug 00 into the blank, and the binary byte becomes **1000 0000**. That's hex **$80**, and that's your operand. Next to address **$3F28**, write **$80**. And be sure to note the label BLOOP here at address **$3F27**. You've got to get back there later.

There's nothing really new in the rest of the main program, just tedious hand assembly. The next instruction is a **JSR**. That's hex code **$BD**. Write **$BD** next to address **$3F29**, and leave the next two addresses blank. Still don't know where the subroutine will be.

STB ,Y+ is next, and you can steal that information from earlier. STB indexed is **$E7**; write that at address **$3F2C**. Auto-increment zero-offset indexed Y is **$AO**; write that at address **$3F2D**.

The decrement instruction is next. Find that on page 30. This is decrement a direct page memory location you're dealing with, opcode **$0A**. Next to **$3F2E** write **$0A**. The location to decrement is **$00**, so that's your operand. Write **$00** next to address **$3F2F**.

Hand assembly is tiresome and troublesome. But it teaches you, giving you a level of intimacy with the machine that you can't achieve with mnemonics alone. If this kind of detail bothers you, consider that understanding someone else's program -- without recourse to commented source code -- can only be achieved by disassembling and examining the binary information. Knowing it both ways is your key to programming versatility.

* What is hand assembly?

Figuring the hex (binary) code byte by byte from the mnemonic (source) code.

* What are the bytes in an indexed instruction?

The opcode, the postbyte, and additional bytes of operand if necessary.

* Hand assemble LDA ,X+

$A6 80

* What is LDA ,X+ in binary?

10100110 10000000

* Hand assemble LDA $1234,X

$A6 89 12 34

* What indexed addressing mode is LDA $1234,X?

16-bit constant-offset indexed.

* If the label BLOOP is found at address $3F27, hand assemble this instruction, found at address $3F30: BNE BLOOP

$26 F5

* What addressing mode is this?

Relative addressing.

* Relative addressing is relative to what?

The program counter (PC).

* In the assembly of BNE BLOOP ($26 F5), what does the value $F5 signify?

An offset relative to the program counter.

* What is the offset in binary?

In binary, 11110101

* What is the offset in decimal?

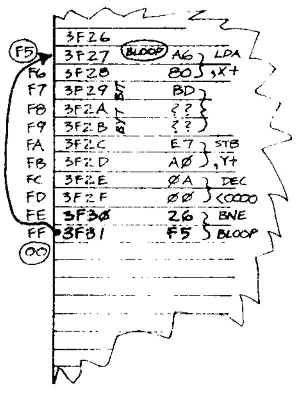
In decimal, -11.

* What makes $F5 negative?

The fact that in $F5 (11110101), the leftmost bit is a one.

* The following exercises are hand disassembly, that is, the translation from hexadecimal (or binary) code into mnemonic code. This is done with unknown programs for purposes of examining the operation of the program. Use the chart in the MC6809E data booklet on pages 28 and 29 for help. Disassemble, describe and give the mnemonic for each of the following groups of bytes.

Finally a branch instruction. Branch on Not Equal can be found on page 32. Find the table at the bottom right labeled “Simple Conditional Branches”. Under “false”, second mnemonic down, is BNE. The opcode shown is **$26**. So next to address **$3F30**, write **$26**. At this point in the program, the Program Counter is pointing to the next instruction in line after this one . . . meaning the Program Counter is pointing to address **$3F32**. Now locate your label BLOOP. This is where the branch is going. If **$3F32** is relative position **00**, count backwards to the address BLOOP, which is **$3F27**. **FF, FE; FD, FC; FB, FA; F9, F8, F7; F6, F5.** **$F5** is the position of BLOOP relative to the Program Counter. That makes **$F5** your operand for the relative branch BNE. Next to address **$3F31**, then, write **$F5**.

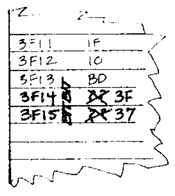


Decrement direct page you know already. The opcode is **$0A**, and should be written next to address **$3F32**. The operand for direct page **$00**, least significant byte **$01**, is **$01**. Next to address **$3F33**, write **$01**.

Another relative branch follows. This is BNE again, opcode **$26**. Write that down next to address **$3F34**. Now comes the counting backwards from the Program Counter, which is pointing to **$3F36**. You’ve got to get all the way back to LLOOP at address **$3F0D**. If you subtract it instead of counting backwards, you’ll get the value **$D7**. I won’t put you through it this time. Just write your relative branch operand **$D7** at address **$3F35**.

All that remains of the main program now is the return from subroutine. Find that on page 31 if you need to. It’s opcode **$39**. Next to address **$3F36**, write **$39**. The main program is complete. Only the subroutines remain; the subroutine BYTBIT is coming up next, and its address is **$3F37**.

The subroutine BYTBIT begins at **$3F37**, meaning your three blank operands earlier in the program were filled with that address. The first action of the subroutine was to push the A Accumulator on the stack. **$34** is the opcode, and using the push/pull order chart, you found that **$02** is the operand. Four logical shift right A accumulator commands followed; each of these is **$44**.



Two more subroutine calls follow, **$BD** being the opcode for jump to subroutine. The addresses, which you had to calculate on your second assembly pass, are respectively **$3F4E** and **$3F58**.

Pull accumulator is **$35 02**, the operand calculated in the same manner as for the push command. And A immediate with **$0F** is represented **$84 0F**.

A familiar pair of subroutine calls follows — **$BD 3F 4E** and **$BD 3F 58** — and the convert and display subroutine finishes with the return from subroutine, **$39**.

The short CONVRT subroutine compares A immediate with **$0A** — that’s **$81 0A**. It branches on carry clear (or

BHS . . . branch on high or same, meaning greater or equal) to the label LETTER. You calculated that relative branch to be **$03**, giving an instruction of **$24 03**. Add A immediate with **$30** is **$8B 30**, and return from subroutine is again **$39**.

At the label LETTER, add A immediate with **$37** is **$8B 37**, followed by **RTS, $39**.

Finally, the short display and update routine is made up of **STA ,Y+** . . . store A at Y, zero-offset, auto-increment. That pattern is familiar enough to copy the information from earlier in the program — **$A7 A0**. And, at last, the final return from subroutine, **$39**.

Your program should run from address **$3F00** to **$3F5A**, a total of 91 bytes. Look in your documentation, and see if your hand-assembled hexadecimal code agrees with mine:

```
3F00 ** BD B3 ED 10 8E 04 80 1F
3F08 ** 01 86 08 97 01 86 08 97
3F10 ** 00 1F 10 BD 3F 37 1F 98
3F18 ** BD 3F 37 86 2A C6 20 E7
3F20 ** A0 A7 A0 A7 A0 E7 A0 A6
3F28 ** 80 BD 3F 37 E7 A0 0A 00
3F30 ** 26 F5 0A 01 26 D7 39 34
3F38 ** 02 44 44 44 44 BD 3F 4E
3F40 ** BD 3F 58 35 02 84 0F BD
3F48 ** 3F 4E BD 3F 58 39 81 0A
3F50 ** 24 03 8B 30 39 8B 37 39
3F58 ** A7 A0 39
```

Time to get it running. I've got this batch of hexadecimal code prepared for you as a series of BASIC DATA statements.

Program #18, 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 DATA BD,B3,ED,10,8E,04,80,1F
15 DATA 01,86,08,97,01,86,08,97
20 DATA 00,1F,10,BD,3F,37,1F,98
25 DATA BD,3F,37,86,2A,C6,20,E7
30 DATA A0,A7,A0,A7,A0,E7,A0,A6
35 DATA 80,BD,3F,37,E7,A0,0A,00
40 DATA 26,F5,0A,01,26,D7,39,34
45 DATA 02,44,44,44,44,BD,3F,4E
50 DATA BD,3F,58,35,02,84,0F,BD
55 DATA 3F,4E,BD,3F,58,39,81,0A
60 DATA 24,03,8B,30,39,8B,37,39
65 DATA A7,A0,39
70 FORX=&H3F00 TO &H3F5A
75 READA$:A=VAL("&H"+A$)
80 POKEX,A:NEXT:DEFUSR0=&H3F00
85 CLS:PRINT"TEST ADDRESS 3F00:"
90 M=USR0(&H3F00)
```

*** Disassemble $BD B3 ED**

$BD is jump to subroutine, extended addressing mode; therefore, $BD B3 ED is JSR $B3ED.

*** Disassemble $86 6A**

$86 is load A accumulator immediate; therefore, $86 6A is LDA #$6A.

*** Disassemble $44**

$44 is an inherent instruction, logical shift right A accumulator: LSRA.

*** Disassemble $35 02**

$35 is pull from the hardware stack; $02 is binary 00000010, indicating the A accumulator. Therefore, the instruction is PULS A.

*** Disassemble $1F 01**

$1F is transfer from register to register; 01 is binary 0000 0001. The transfer-from register is D (0000) and the transfer-to register is X (0001). Therefore the instruction is TFR D,X.

*** Disassemble $10 8E 04 80**

$10 8E is a two-byte opcode for load Y register immediate; the Y register is 16 bits, so $04 80 is the 16-bit operand. Therefore, the instruction is LDY #$0480.

*** Disassemble $81 0A**

$81 is compare A register immediate; therefore the instruction is CMPA #$0A.

*** Disassemble $0A 01**

$0A is the opcode for decrement memory direct; therefore the instruction is DEC ($NN01, where NN is the direct page register.

* Disassemble $86 6A C6 60

$86 is load A accumulator immediate, so $86 6A is LDA #$6A. That means $C6 60 must be another instruction. $C6 is load B accumulator immediate, so $C6 60 is LDB #$60. You can't fool me.

* Disassemble $A7 A0

$A7 is store A accumulator indexed; A0 is binary 10100000. Referring to the chart, the only postbyte that ends in 0000 is ,R+. 1RR0 applied to 1010 makes RR=01. 01 is the Y register. Therefore, $A7 A0 is STA ,Y+.

* Disassemble $A6 80

$A6 is load A accumulator indexed. $80 is 10000000. This is again ,R+ indexed mode, with 1RR0 = 1000, RR = 00 = X register. Therefore, $A6 80 is LDA ,X+.

* What does &H mean in BASIC?

Hexadecimal.

* If A$="BD", what is the value of A after this statement: A=VAL("&H"+A$)?

A equals decimal 189 (hex $BD).

* What does DEFUSR0=&H3F00 mean?

Define the BASIC user entry point number 0 to be at $3F00.

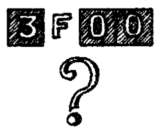
* How many characters (letters, numbers and symbols) does the Color Computer display using PRINT?

96.

* How many characters is the Color Computer capable of displaying using POKE?

128.

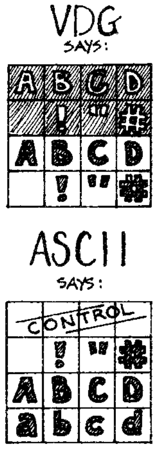
So there it is. A reasonably painful first hand-assembly, resolved into a mere 12 lines of BASIC DATA statements, POKEd in place and used as a subroutine via the USR command. In this test program, the address transferred for display is **$3F00** — so you can look at the machine language program itself. Will it work? No, it won't. That is, not exactly as you expect. RUN the program.



Well, you *are* looking at your own hand-assembly, but something's amiss. The letters are okay, but the numbers are shown in reverse video.

A peculiarity like this is one of my reasons for preparing these lessons with the Color Computer in mind. If you've been using your Color Computer for a while, you know that upper case characters, plus numbers and symbols, are presented normally, but that lowercase characters are represented by reverse video. What you're running into here is the video display generator, the VDG. There's a software shuffle done by BASIC to accept your ASCII information and translate it into VDG codes.

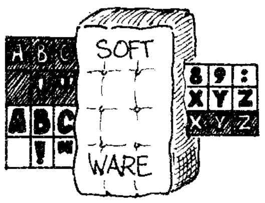
It's bit time again. The video display generator contains only 64 letters, numbers and symbols, all standard uppercase characters. No lowercase or control characters were included in the design and manufacture of this part. To display any character in this set, then, only bits 0 through 5 in a byte are used. However, bits 6 and 7 are connected to the VDG. Bit 7 turns on the low-resolution color graphics, which BASIC calls CHR$(128) through CHR$(255) — hexadecimal **$80** through **$FF**. Bit 6 is the tricky one. It is used to turn on the inverse-video mode for the alphanumeric characters. When bit 6 is a one, normal characters are seen; when bit 6 is a zero, reverse characters are displayed.



Think back to my example of POKEing vs. PRINTing the screen, way back in the first session. PRINTing the characters resulted in their appearance in normal ASCII order — control characters from **$00** to **$1F** were not displayed, **$20** through **$3F** were numbers and symbols, **$40** through **$5F** was uppercase, and **$60** through **$7F** was reverse-video-style lowercase.

But POKEing the values to the screen resulted in something different. Values **$00** through **$1F** revealed reverse-video-style lowercase, **$20** through **$3F** displayed a not-before-seen group of reverse-video numbers and symbols, **$40** through **$5F** showed the uppercase characters in their proper ASCII position, and **$60** through **$7F** displayed the normal set of numbers and symbols.

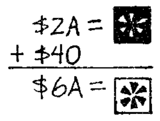
The reasons should begin to come clear. If bit 7 is zero, then alphanumerics are displayed instead of graphics. If bit 6 is zero, all characters are displayed in reverse video mode. In other words, the hardware of the Color Computer understands that all characters from **00 00 0000** to **00 11 1111** — that is, from hex **$00** to **$3F** — are reverse characters. Conversely, if bit 6 is one, all characters are displayed in normal video mode. The Color Computer hardware then understands that all characters from **01 00 0000** to **01 11 1111** — that is, from hex **$40** to **$7F** — are normal characters.



The BASIC language works with ASCII, so this hardware business is a pain. BASIC is forced to translate ASCII to hardware and hardware to ASCII every time it does a screen display! So whenever you write software in machine language, you will also have to provide some sort of translation. Here's a summary:

If you want:	You have to use:
ASCII $ØØ to $1F, control functions	Control software without display.
ASCII $2Ø to $3F, numbers and symbols	Hardware $6Ø to $7F
ASCII $4Ø to $5F, normal uppercase	Hardware $4Ø to $5F (no change).
ASCII $6Ø to $7F, normal lowercase	Hardware $ØØ to $1F

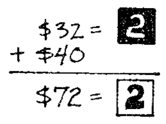
In normal display (such as BASIC), hardware values **$20** to **$3F** are not used; these are the reverse numbers and symbols. The program you just created, in attempting to use legitimate ASCII values, used the hardware values for reverse characters. That accounts for the funky screen display.



$2A = 0010 1010
$6A = 0110 1010

Now you have enough information to get out of that dilemma. Turn back to your hand-assembled listing and locate the spots where a display character is established. You'll find address **$3F1C** is supposed to be a star, hex **$2A**. Glance at your documentation where the summary I just gave you is printed. If you want to display **$2A**, then, you actually need to use the hardware value **$6A**. Put that in place. Type POKE &H3F1C,&H6A and hit <ENTER>. That's POKE &H3F1C,&H6A <ENTER>. That should give you a proper star; try it. Type GOTO85 and hit <ENTER>.

The stars are okay now. The spaces are next. A space is **$20**, which means the hardware requires a **$60**. In your hand assembly, you'll find that space at address **$3F1E**. Change it now. Type POKE &H3F1E,&H60 and hit <ENTER>. That's POKE &H3F1E,&H60 <ENTER>. The spaces should be cleared up. Type GOTO85 and hit <ENTER>.



$32 = 0011 0010
$72 = 0111 0010

Only the reverse numbers remain to cure. This happened in the ASCII conversion subroutine that began at address **$3F4E**. Find that subroutine. At address **$3F52**, an offset of **$30** was added to convert from the number 0 through 9 to ASCII CHARACTERS "0" through "9". Hex values for these are **$30** through **$39**, meaning the hardware needs **$70** through **$79** to present the numbers correctly. So the

*** What does VDG mean?**

Video display generator.

*** How many unique characters is the VDG capable of displaying?**

64.

*** Why can the VDG display 64 characters, whereas the Color Computer can display 128?**

Because the Color Computer displays 64 normal characters and 64 reverse-video characters.

*** What do the ASCII codes from $00 to $1F represent?**

Control codes (carriage return, backspace, tab, etc.)

*** What do the ASCII codes from $20 to $3F represent?**

Numbers, symbols and punctuation.

*** What do the ASCII codes from $40 to $5F represent?**

Uppercase (capital) letters.

*** What do the ASCII codes from $60 to $7F represent?**

Lowercase (small) letters.

*** What do VDG codes $00 to $1F represent?**

Lowercase (reverse) letters.

*** What do VDG codes $20 to $3F represent?**

Reverse-video numbers, symbols and punctuation.

*** What do VDG codes $40 to $5F represent?**

Uppercase letters.

offset at address **$3F53** has to be changed from a proper ASCII **$30** to the hardware's demand of **$70**. Do it. POKE &H3F53,&H70 and hit <ENTER>. That's POKE &H3F53,&H70 <ENTER>. That should cure the numbers. Type GOTO85 and hit <ENTER>.

That did it. The address and data display is complete. That video hardware shuffle is a little tricky, so if it's not clear to you at this point, please review from the start of this session. You can break now. Otherwise, I have a program for you to load.

*** What do VDG codes $60 to $7F represent?**

Numbers, symbols and punctuation.

*** To create the display "A0D7 ** 8E" in ASCII, what ten bytes would be used?**

$41 30 44 37 20 2A 2A 20 38 45

*** To create the display "A0D7 ** 8E" in VDG terms, what ten bytes would be used?**

$41 70 44 77 60 6A 6A 60 78 45

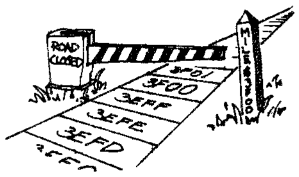
Program #19, an EDTASM+ program. Insert the EDTASM+ cartridge, and turn on the power to your computer. When the cursor appears, type L and press ENTER. The computer will search (S) and find (F). When the cursor reappears, display the program. Type P#:* and press ENTER. If the right-hand side of the program 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.

```
                B3ED       00100 INTCNV    EQU       $B3ED
                0000       00110 BYTES     EQU       $0000
                0001       00120 LINES     EQU       $0001
                           00130 *
3F00                       00140           ORG       $3F00
                           00150 *
3F00 BD         B3ED       00160           JSR       INTCNV
3F03 108E       0480       00170           LDY       #$0480
3F07 1F         01         00180           TFR       D,X
3F09 86         08         00190           LDA       #8
3F0B 97         01         00200           STA       <LINES
3F0D 86         08         00210 LLOOP     LDA       #8
3F0F 97         00         00220           STA       <BYTES
3F11 1F         10         00230           TFR       X,D
3F13 BD         3F37       00240           JSR       BYTBIT
3F16 1F         98         00250           TFR       B,A
3F18 BD         3F37       00260           JSR       BYTBIT
3F1B 86         6A         00270           LDA       #$6A
3F1D C6         60         00280           LDB       #$60
3F1F E7         A0         00290           STB       ,Y+
3F21 A7         A0         00300           STA       ,Y+
3F23 A7         A0         00310           STA       ,Y+
3F25 E7         A0         00320           STB       ,Y+
3F27 A6         80         00330 BLOOP     LDA       ,X+
3F29 BD         3F37       00340           JSR       BYTBIT
3F2C E7         A0         00350           STB       ,Y+
3F2E 0A         00         00360           DEC       <BYTES
3F30 26         F5         00370           BNE       BLOOP
3F32 0A         01         00380           DEC       <LINES
3F34 26         D7         00390           BNE       LLOOP
3F36 39                    00400           RTS
                           00410 *
3F37 34         02         00420 BYTBIT    PSHS      A
3F39 44                    00430           LSRA
3F3A 44                    00440           LSRA
3F3B 44                    00450           LSRA
3F3C 44                    00460           LSRA
3F3D BD         3F4E       00470           JSR       CONVRT
3F40 BD         3F58       00480           JSR       DISPLY
3F43 35         02         00490           PULS      A
3F45 84         0F         00500           ANDA      #$0F
3F47 BD         3F4E       00510           JSR       CONVRT
3F4A BD         3F58       00520           JSR       DISPLY
3F4D 39                    00530           RTS
```

```
                       00540 *
3F4E 81     0A         00550 CONVRT   CMPA     #$0A
3F50 24     03         00560          BCC      LETTER
3F52 8B     70         00570          ADDA     #$70
3F54 39                00580          RTS
3F55 8B     37         00590 LETTER   ADDA     #$37
3F57 39                00600          RTS
3F58 A7     A0         00620 DISPLY   STA      ,Y+
3F5A 39                00630          RTS
            0000       00640          END
00000 TOTAL ERRORS
BLOOP    3F27
BYTBIT   3F37
BYTES    0000
CONVRT   3F4E
DISPLY   3F58
INTCNV   B3ED
LETTER   3F55
LINES    0001
LLOOP    3F0D
```

Here's the complete program you just created. You have the entire mnemonic listing available, which the assembler can convert to machine language very quickly.

You'll assemble this, go right into BASIC, and load the next program on the tape. Here's how it goes. Type A/IM/AO and hit <ENTER>. The listing will scroll by, and the program will be assembled at **$3F00**. When the star prompt and cursor return, quit the editor/assembler: Type Q <ENTER>. In a few seconds, the Extended Color BASIC message will appear. You know the program is at **$3F00**, so protect memory.



If you've never protected memory before, the purpose is to tell BASIC that a certain area is off-limits. BASIC will make no attempt to use protected memory, except through PEEK, POKE andDM statements. You can refer to your BASIC manual for details. Type CLEAR 200,&H3F00 and hit <ENTER>. That's CLEAR 200,&H3F00. Now you can load the next program.

> Program #20, 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/0 error occurs, rewind to the start of the program and try again. For sever loading problems, see the Appendix.

```
10 DEFUSR0=&H3F00
20 CLS
30 PRINT@0,"":PRINT@0,"";
40 INPUT"ADDRESS";A$
50 A=VAL("&H"+A$)
60 IFA>32767THENA=A-65536
70 M=USR(A)
80 GOTO30
```

* What is the process of translating mnemonic (source) code into binary (object) code called?

Assembly.

* What is the process of translating binary (object) code into mnemonic (source) code called?

Disassembly.

* What is the term for binary digit?

Bit.

* What is the term for eight binary digits (bits)?

A byte.

* What is the term for four binary digits?

A nybble (also spelled nibble).

* What number system represents binary digits?

The binary system.

* What number system organizes the binary numbers into convenient size?

The hexadecimal system.

Summary: Architecture

* What does ASCII mean?

The American Standard Code for Information Interchange.

* What are the ASCII codes?

Control codes; numbers, symbols, and punctuation; and upper and lowercase letters.

* What is the term for a computer instruction?

An opcode.

* What is the term for an opcode's data?

An operand.

* What is the term for the design and purpose of a processor?

Its architecture.

* What is the architecture of the 6809 processor?

Accumulators A and B; index registers X and Y; stack pointers S and U; direct page register DP; condition code register CC; program counter PC; 65,536 bytes of memory.

* What is an addressing mode?

The way in which a machine language program gets its information.

* What are the 6809's addressing modes?

Inherent, register, immediate, extended, direct, and indexed.

If you LIST this program, you can see that it is simply a little BASIC input routine which subsequently calls your machine-language subroutine. Since the value transferred is an integer, the range is –32768 to +32767. To get at addresses higher than 32767 (that's hex **$7FFF**), there has to be a conversion. Line 60 is a little trick that's good to know. So assuming the assembly went well, then your program should reside at **$3F00** right now. RUN the program.

Enter any address from **$0000** to **$FFFF**. There's memory, 64 bytes of it. Play around with this program for a little bit, and then come back to this tape for a summary.

Please examine a number of areas of memory using this BASIC program. When you are confident of the significance and application of this process, return to the tape.

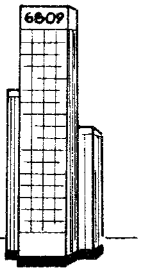
This course is halfway now. If you're a newcomer to assembly language, you're probably just a little overwhelmed. The jargon and the concepts take *do* some time to settle. On the other hand, if you're an experienced programmer here just to learn the specifics of the 6809, then I know you're itching to get on with it. In either case, I would like you to stay with me for a summary of the main points from these past twelve sessions.

Assembly language is not BASIC, but forms a perfect companion to BASIC, as you have seen with the programs so far. It is capable of easy access to functions not easily available via BASIC, such as reverse numbers and symbols, and is fast and flexible. From assembly language is built the language of BASIC itself.

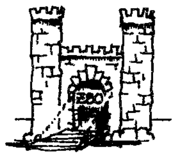
Assembly language represents computer instructions. Computer instructions are actually electronic signal patterns best represented by the binary number system. Binary numbers are difficult to recognize, so binary patterns are visually organized by using a single symbol for each group of four bits. This is the purpose of the hexadecimal numbering system **1** through **9** and **A** through **F**. Logical arrangements and patterns of binary digits were used in the creation of the American Standard Code for Information Interchange, ASCII.

A processor can interpret binary patterns as either instructions or data. The instruction pattern is known as the opcode, and the data pattern is called the operand. All binary patterns in a program are found in memory, and it is only the order and context which differentiate opcodes from operands.

Each processor has a unique design and purpose. This design and purpose is known as the processor's architecture. The architecture of the 6809 is particularly strong in the way it accesses information. The architecture of the 6809 consists of a program counter PC, two arithmetic-performing accumulators A and B, two index registers X and Y, two stack pointers S and U, a condition code register CC (also called the flags), and a direct page register DP.

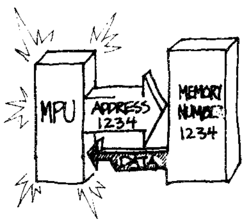


To manipulate these registers through a program, the binary code must be presented to the instruction decoder. Because binary and hexadecimal representations are machine-level instructions with little value for the human programmer, instruction names are used to ease the programming process. They are called mnemonics. A program written in mnemonics cannot be executed; only the machine code the mnemonics represent can be executed. The process of translating mnemonics into machine code is called assembly. The program that permits editing mnemonic code, also called source code, is an editor. The program that translates this source code into machine code, also called object code, is an assembler. Usually these are combined in a single program known as an editor/assember; this series uses the program EDTASM+ as its editor/assembler.



A machine language program must access information. The way it finds this information in memory is called an addressing mode. There are several major addressing modes in the 6809 processor. Inherent addressing has the data implied as part of the opcode itself. Register addressing has the data available in one of the 6809 registers. Immediate addressing presents the data in the program memory immediately following the opcode. Extended addressing presents a complete 16-bit memory address at which the data can be found. Direct addressing presents an 8-bit address which is combined with the DP register to locate the data in memory.

Indexed addressing uses registers and offsets to calculate the address in memory at which data can be found. This mode is complex and flexible, with automatic incrementing and decrementing of registers as the instruction is executed. Relative addressing presents a value which directs the program to a position in memory relative to the current position of the program counter.



Information is accessed by processor instructions. Among these are store and load, which save and retrieve information in memory; arithmetic instructions such as add, subtract, decrement, increment, and negate; logical instructions such as AND, OR, complement, and Exclusive-OR; bit shift and rotation instructions; jumps and branches to other program locations or subroutines.

Programs and information are stored in memory. The organization of memory is called a memory map, which can contain read-write memory, read-only memory, special-purpose memory registers, and input/output ports.

Summary: Addressing

* What do RAM, ROM, CPU, SAM, PIA, and VDG mean?

RAM means read/write memory, also known as random-access memory; ROM means read-only memory; CPU means central processing unit; SAM means synchronous address multiplexer; PIA means peripheral interface adapter; and VDG means video display generator.

* What is the process of translating mnemonic (source) code into binary (object) code called?

Assembly.

* What programming tool performs this task?

An assembler.

* How many topics must you know to continue this course?

Six topics.

* What is the first topic you need to know to continue this course?

How to use the MC6809E data booklet.

* What is the second topic you need to know?

How to enter and edit programs using EDTASM+.

* What is the third topic you need to know?

How to count in binary and hexadecimal.

* What is the fourth necessary topic?

How to create BASIC programs which POKE machine language into memory.

Summary: Special devices

*** What is the fifth item you need to know?**

All the 6809 instructions presented up to the 12th session.

*** What is the final topic?**

The addressing modes presented up to the 12th session.

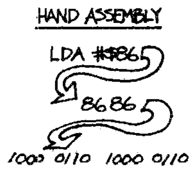
*** How many questions have you answered so far in this course?**

895. Bet you didn't know that.

The Color Computer has a specific memory map and several hardware devices. Read-write memory, or RAM, is located in the bottom two quarters of the memory map; the BASIC language in read-only-memory (or ROM) occupies the third quarter; most of the upper quarter is occupied by cartridge ROM when it is plugged in.

The top 256 bytes of memory have a special purpose, and are used by Peripheral Interface Adaptors (the PIAs) as input/output ports for the keyboard, cassette, printer, video display, and other reserved purposes. In the Color Computer, the most sophisticated of these functions is performed by the Synchronous Address Multiplexer (the SAM), which controls the memory circuitry, the processor speed, and the Video Display Generator (the VDG). By using a combination of the PIAs and the SAM, the VDG can be placed into several modes of alphanumerics and low- and high-resolution color graphics, and can be made to display any area of memory. Machine-language programs most easily control these devices.

Machine-language programs for control, display or any type of programming can be assembled using a tool such as the editor/assembler, or can be assembled by hand using a list of commands and their respective binary codes. Hand assembly is tedious, but is valuable for learning to create compact and efficient programs, and for understanding the specific actions taken by the processor. Confident hand-assembly can reveal peculiarities in a computer, such as the alphanumeric display method in the Color Computer.



So that's a very fast trip through the past twelve sessions. I recommend that you take a breather now and review these lessons, because I plan to pick up the pace from here on. Things you must know to continue are: how to use the MC6809E data booklet; how to enter and edit programs using EDTASM+; how to count in binary and hexadecimal; and how to create BASIC programs which POKE machine-language information into memory. You must also know the 6809 instructions that have been presented so far, and all the addressing modes which I've explained.

I won't have time to summarize all of this again, so if you think you need to, please review now. I can't emphasize enough the need to review, because I can tell you from experience that if you get in to this too deeply and your background is not secure, the new information will muddy the old information so badly it will all become useless.

Now that I've issued my dire warnings, I hope you will continue this series. I'll be presenting graphics and sound software soon, and giving you pointers on making your programs short, be quick, and run bug-free. Speak to you next time.

13.

Hello and welcome back to the final half of "Learning the 6809." The pace will quicken somewhat, so I hope you've given yourself a solid foundation in the essentials of machine language programming that I presented in the first half of this course.

The topic this session is timing: that's the careful organization of computer instructions to perform tasks at a known speed. Unlike mechanical timers or ordinary clocks, the computer operations you can be certain of actually simplify this task. You are certain of the clock speed, that is, the number of fixed pulses per second by which the processor completes its instructions. And, you can identify the specific number of those clock cycles each instruction requires, since this is consistent... and the full information is provided with the data booklet.

You may not be as impressed as I am with this concept. But consider that all the real-world interfacing of the computer depends on some sort of timing. Here are just a few of those interfacing tasks:

1. Communication with a printer is timed. A printer connected to the Color Computer expects precisely 600 binary digits per second.

2. Cassette input and output is astoundingly precise. Not only is the timing of the binary digits critical, but the shape of the sound's wave recorded on the tape is important. Care in these timings overcomes the inherently poor quality of portable cassette recorders.

3. Keyboard input even uses timing. As the metal contacts of the keys close, a little electromechanical bouncing takes place. This bounce must be timed through so as not to produce unwanted double or triple characters.

4. BASIC sound commands need frequency

The very speed that makes machine language a programming delight also makes it difficult when dealing with a real world operating in human terms. You start wishing for BASIC after a few hours of meticulously timed program actions. But you'll never be able to create sound or games with real punch and clarity from BASIC, so machine-language bit twiddling is the solution. Onward!

* What is the clock speed of the Color Computer?

.89 MHz (894,886 clock pulses per second).

* If a printer expects information at 600 binary digits per second, how many clock pulses is that?

Approximately 1,492 clock pulses.

* If a given computer activity had to take place 100 times per second, how many clock cycles would that be?

Approximately 89,489 clock cycles.

* At 1,000 activities per second?

Approximately 8,949 clock cycles.

* At 10,000 activities per second?

Approximately 895 clock cycles.

* What does Hz mean?

Hz means Hertz, or cycles (pulses) per second.

* What does MHz mean?

MHz means megaHertz, or million cycles per second.

* Which of the following require consideration of timing: cassette input and output; serial printer output; keyboard input; sound output.

All require consideration of timing.

* Why does cassette input and output require timing?

Because the data must be recorded and received at a known rate.

* Why does serial printer output require timing?

Because a serial printer must receive data at a known rate.

* Why does keyboard input require timing?

Because mechanical contact bounce must be ignored (timed through).

* Why does sound output require timing?

Because sound is made up of specific frequencies, and frequencies are inherently time-based.

information in order to produce proper musical pitches.

These four examples are only the most obvious. Subtle kinds of timing permeate machine language programming.

I'd like to start with the simplest kind of timing, the delay loop. No doubt you've used FOR-NEXT loops in BASIC to time such things as screen presentations and Inkey$ input. Another interesting use of delay loops is for simplified communications timing . . . in the example I've got for you, it's used for sending fast and accurate Morse Code. Now Morse Code might be a little bit of an anachronism in this computer era, but it's interesting and I think quite a lot of fun.

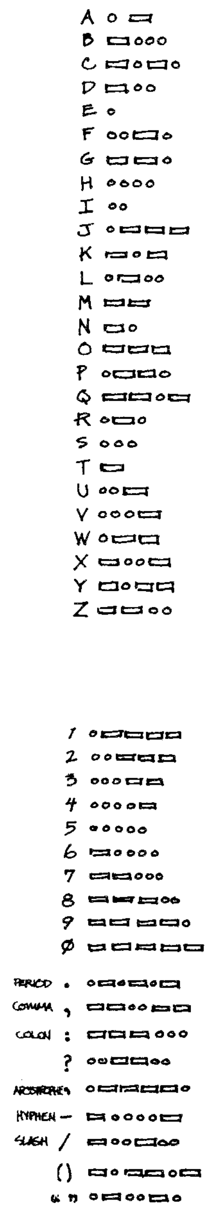
First, conceptualize the problem and establish some parameters. Morse code is that pattern of long and short beeps that has been used for over a century to communicate across telegraph wires and via radio. In this example, the code might be sent from the keyboard, or it might be sent from a prepared, edited message. Also, you've got to establish the speed of code transmission and choose the pitch of the beep. Finally, the character set to be used must be selected (that is, the whole set or just the alphabetic characters).

Let's take the last first, and say that the entire 6-bit ASCII character set should be used. Those are numbers and uppercase letters. Let's set the beep at a clear 1,000 Hz — 1,000 cycles or vibrations per second. And finally, establish the transmission speed at about 10 words per minute. Before actually programming these last two items, keep in mind that it might be wise to make both the beep frequency and the transmission speed flexible, so they can be changed by the operator to match the circumstances.

Now to the concept. It seems to break down into a few simple steps coupled with a some crucial subroutines. It looks like this. A message is found somewhere in memory. The code for each character is located in a table of Morse codes. After the code is identified, it is used in conjunction with two or three subroutines to produce beeps and silences of the proper timing.

Now I don't know very much about Morse Code, but from what I'm told, it consists of short and long beeps known as "dits" and "dahs." A "dah" is roughly three times as long as a "dit," and all beeps are separated by "dit"-length silences. Letters are separated by "dah"-length silences, and words, when separated at all, are separated by about two "dahs."

Before I get too far ahead, let me play for you a little bit of professional Morse Code . . .



What you just heard was the message "Hello how are you." It's a series of pure, regular beeping tones and silences. You might think that Extended Color BASIC has a perfectly adequate group of SOUND and PLAY commands, tailored to this kind of task. Unfortunately, they won't do for a number of reasons. First, the beep length is a fixed multiple of the shortest length. Morse Code speeds often fall in between these fixed lengths. Next, the BASIC programming is very clumsy, using a long array, substantive error-checking, and various loops. But worst of all is the slight but distinguishable "gargling" in the sound, an adulteration of the pure tone with pops and burbles. At first — and especially if you are listening on an inexpensive television — that impurity may be obscured by the limited TV sound. But if you listen through a separate amplifier hooked to the cassette output, the unevenness of the sound becomes distinct. Think about those things as you load and run the following BASIC program.

Program #21, a BASIC program. Turn on the power of your Extended Color BASIC computer. Whe 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/0 error occurs. rewind to the start of the program and try again. For severe loading problems. see the Appendix.

```
1 CLEAR500:DIMA$(60)
2 FORX=32 TO 90
3 READA$(X-32)
4 NEXT
5 CLS
6 PRINT"TYPE YOUR MESSAGE"
7 PRINT"(MAXIMUM 255 CHARACTERS)"
8 LINEINPUTB$
9 PRINT:PRINT"SENDING MESSAGE..."
10 FORX=1TO LEN(B$)
11 A=ASC(MID$(B$,X,1))
12 A=A-32:PRINTMID$(B$,X,1);
13 C$=A$(A)
14 IFC$="SP"THEN22
15 PRINTC$" ";
16 FORY=1TOLEN(C$)
17 Q$=MID$(C$,Y,1)
18 IFQ$="."THENSOUND240,1
19 IFQ$="-"THENSOUND240,3
20 FORZ=1TO50:NEXT
21 NEXT
22 FORZ=1TO100:NEXT
23 NEXT
24 GOTO5
25 DATASP,SP,.-..-.,SP,SP,SP,SP,.----.,-.--.-,-.--.-,SP,SP,--..-
-,-....-,.-.-.-,-..-.
26 DATA-----,.----,..---,...--,....-,.....,-....,--...,---..,--
--.,---...,SP,SP,SP,SP,..--..
27 DATASP,.-,-...,-.-.,-..,.,..-.,--.,....,..,.---,-.-,.-..,--,-
.,---
28 DATA.--.,--.-,.-.,...,-,..-,...-,.--,-..-,-.--,--..
```

You've just heard a BASIC solution to the problem of transmitting Morse Code. For the simplest of purposes, this kind of code transmission might be adequate. But we can do far better in machine language. The timing and

*** Is timing required for Morse Code?**

Yes.

*** Name the timing considerations needed for Morse Code.**

The length of the "dit", the length of the "dah", the length of silences, and the frequency of the beep.

*** How long is a "dah" with respect to a "dit"?**

Three times as long.

*** What commands produce sound in BASIC?**

SOUND and PLAY.

*** How long is the shortest BASIC beep (using SOUND X,1)?**

Approximately 1/14 of a second. (You weren't told that in the text).

*** If a loop contains two load A immediates, two store A extendeds, and one branch to make a complete loop, how many clock cycles are required for this loop?**

2 times 2 cycles, plus 2 times 5 cycles, plus 2 cycles ... a total of 16 cycles.

*** At 894,886 clock cycles per second, how many loops is this?**

894,886 divided by 16, or 55,930 cycles.

* What is the main disadvantage of producing sound using BASIC?

The "gargling" or unevenness of the sound.

* What causes the "gargling" of the sound?

An interrupt.

* Three things happen when an interrupt occurs. What are they?

The microprocessor finishes its current instruction, saves important information, and follows programming instructions in reponse to the interrupt.

* What is the process of acting on an interrupt called?

Servicing the interrupt.

* What causes an interrupt?

When an external signal line changes from one to zero.

* What three things happen when an interrupt occurs?

The microprocessor finishes its current instruction, saves important information, and follows programming instructions in reponse to the interrupt.

* Can more than one interrupt occur?

Yes.

* Which interrupt gets taken care of?

The one with higher priority.

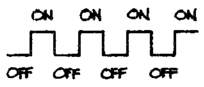
* What is the process of taking care of the interrupt called?

Servicing the interrupt.

control of the sound can be intimately precise, and that annoying gargle will disappear.

What, then, do you suppose causes that gargling sound? It seems to be a regularly recurring group of little hiccups as the tone proceeds. In fact, those hiccups are the time it takes the computer to briefly abandon the sound program in progress and perform other tasks. It is responding to an interrupt.

Practically all microprocessors are provided with electrical connections known as interrupt lines. When these interrupt lines are made to change from one to zero by some external happening, the microprocessor finishes its current instruction, saves important information, and follows special programming instructions in response to that interrupt.



It's like one of those drive-up fast-food places. We take you now to Sam's Roadside Kitchen in Roadside, New Jersey, where the sign reads "Honk for Sam's roadside drive-up service noon to 6 only. Other times honk at your own risk"..

Marge the Waitress: <indoors, talking to cook> One fries, two BLTs, three chili dogs . . . <honk> Alright, alright. <back to cook> . . . and one onion rings. Get those ready. There's a guy out honkin' that thing like Little Richard. <going outdoors> Yeah, what'll you have?

Car one: Three burgers, two fries, a shake.

Marge: Ya want bunny burgers or buddy burgers?

Car one: One bunny burger, two buddy burgers.

Marge: <indoors again>. One bunny, two buddies, fries. Where's my order? <to counter> Anything else, Joe? How 'bout you, Mac?

Mac: Yeah, gimme another dog, will ya Marge? With onions an' cheese, too.

Marge: <to cook> Cheese dog onions.

Kitchen: Orders up.

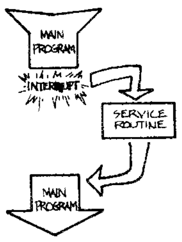
Marge: <to cook> Hey where's my steak? And what about . . . <honking> . . . the chili dog. Damn. Gotta get that. <outside again, honking continues> Yeah, yeah, whaddaya want?

Car two: Gimme three bunnies and... <honking from third car>

Marge: <to third car> Hey fella I'm busy. Sit on it till I get to ya. <back to car> Three bunnies. What else, and make it quick.

Car 2: How about filet mignon and truffles and leeks vinaigrette. . .

In this example, the restaurant was the computer, and Marge its microprocessor. The cook and customers served as program and storage memory. The car horn was the interrupt. Recall how Marge finished only the immediate task, and then went out to take care of the drive-up customer. In computer terms, that process is called "servicing the interrupt." When servicing an interrupt, the computer saves program counter and registers so that all information is intact when it returns from the interrupt service routine to finish its previous task. Interestingly, Marge chose to put the third honking customer on hold while she finished with the second. In that case, the interrupt in progress had a higher priority. Had the third car been an old favorite customer, however, Marge might have serviced that interrupt first, leaving the current task incomplete until the interrupt service routine was done.



Finally, Sam's sign said that drive-up service was from noon to six only. Other times, honking customers would be ignored. That process is known as masking an interrupt. They'll be much more talk about interrupts and how they are used later; right now, we only want to know how to get them out of the Morse Code beep. To do that, you have a little reading ahead.

Please read the information on interrupts in the MC6809E data booklet. The condition codes are described at the top of page 6, the vectors are shown in Table 1 on page 6, and an explanation is presented in the first column on page 9. Return to the tape when you have completed the reading.

The most important thing you've discovered about interrupts, at least with respect to the Morse Code program, is that the interrupts can be turned off — masked, that is — by using the condition code register. Bits 4 and 6 are responsible for interrupts; there must be some way to use logical functions AND and OR with the condition code register to mask bits 4 and 6 in or out at will. There is.

Turn to the MC6809E data booklet, pages 30 and 31. You'll find that ANDCC is a special-purpose instruction available only in the immediate addressing mode; so is ORCC on the next page. You might want to check me on paper for what follows. If you wish to set bits 4 and 6 to one — that is, turn the interrupts off — you would OR the byte with binary **0101 0000**. Bits 0 through 3, 5 and would remain unaffected. To turn the interrupts on you need to set bits 4 and 6 to zero; to do that, you would AND the condition code register byte with **10101111**. In either case, the original condition codes for carry, negative, zero, half-

* Can interrupts be ignored?

Yes.

* What permits the processor to ignore an interrupt?

Masking the interrupt.

* Is there an interrupt taking place when BASIC is running?

Yes.

* What is one effect of the interrupt?

A "gargling" in the SOUND command.

* What causes the gargling?

The time taken to service the interrupt; the interrupt service routine.

* Can sound be produced without "gargling"?

Yes.

* How can sound be produced without gargling?

By producing it in machine language.

* How can machine language stop the gargling?

By turning off the interrupt.

* What is the proper term for turning off an interrupt?

Masking the interrupt.

* What determines whether an interrupt is masked or enabled?

The condition code register.

* What part of the condition code register determines whether an interrupt is masked or enabled?

Bits 4 and 6.

* What masks an interrupt?

Setting its condition code bit to a one.

* What commands can be used to affect the condition code register directly?

ANDCC and ORCC, both immediate instructions.

* What condition code bits determine whether interrupts are masked or enabled?

Bits 4 and 6.

* What command specifically masks out (turns off) both interrupts?

ORCC #$50 (binary 01010000).

* What command specifically enables (turns on) both interrupts?

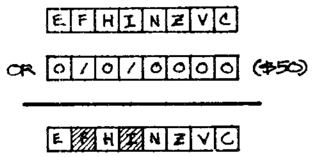
ANDCC #$AF (binary 10101111).

* Three things happen when an interrupt occurs. What are they?

The microprocessor finishes its current instruction, saves important information, and follows programming instructions in reponse to the interrupt.

* For purposes of clarity and simplicity, this session assumes that Morse Code is a maximum of 5 beeps long. For letters and numbers this is true, but punctuation requires 6 beeps. These exceptions will be handled by modifications to the program in the next session. Five beeps are assumed to demonstrate the dramatic simplicity of code translation. Since code translation (ASCII to IBM's EBCDIC, Baudot to ASCII, printer translations, etc.) is an important part of machine-language programming, learning to do it the simplest way is important.

carry, overflow and "entire flag" would be preserved, but bits 4 and 6, the IRQ and fast IRQ interrupts, would be changed. To set interrupts on, then, AND with hex **$AF**; to turn interrupts off, OR with hex **$50**. Much more later.

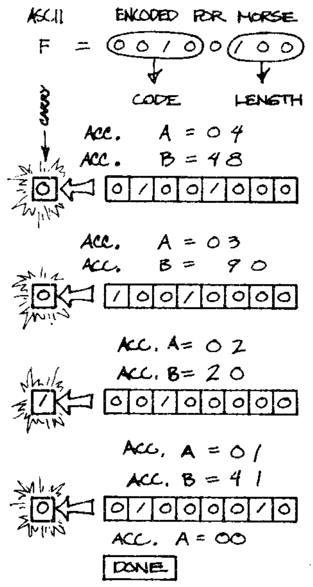


Discussing interrupts has taken this lesson well out of its way. The topic was timing, and specifically, the timing necessary to produce pulses of sound in a known order, with a known pitch, and at a known speed. I'll turn back to that now.

Morse Code was a brilliant invention. It provided a compact method of transmitting letters. It was fast, because the most-used letters contained less beeps than the least-used letters. It accommodated all physical talents because trained operators could send and receive at high speed, whereas novices could still be understood at only a few words per minute. The compactness of Morse Code, however, increases the programming difficulty for us. The letter E, a single dit, contrasts with the letter Z, dah dah dit dit.

For this program, the cross-listing of ASCII codes and Morse Code has to provide two kinds of information: the pattern of dits and dahs, and also the total number of beeps in the letter. The longest character has five beeps, which could be stored as five bits in a byte . . . dits could be represented by zeros and dahs by ones. The remaining three bits could be used to indicate the length of the Morse character. One byte might do the job.

The next question is how to arrange those bits within the byte. The dit-dah pattern could go on either side of the byte, as could the number of beeps. But one arrangement makes special sense. Recall that in an earlier lesson, a binary-to-ASCII conversion was performed. It was always necessary to make sure the nybble was to the right side of the byte to be in the proper form. That's the case here, too. By keeping the rightmost three bits reserved for the length of the Morse Code, the only work you need do is mask the leftmost five bits to retrieve the original number.



Follow me in the book for this. The letter S is dit-dit-dit. According to my suggestion, dit-dit-dit becomes binary **000**. The length is three beeps, so the length is binary **011**. Place the beeps at the left and the length at the right and you've got the composite byte **000 00 011**. By contrast, the letter O is dah-dah-dah. It translates into **111** for the code, and again to **011** for the length. The composite code is **111 00 011**.

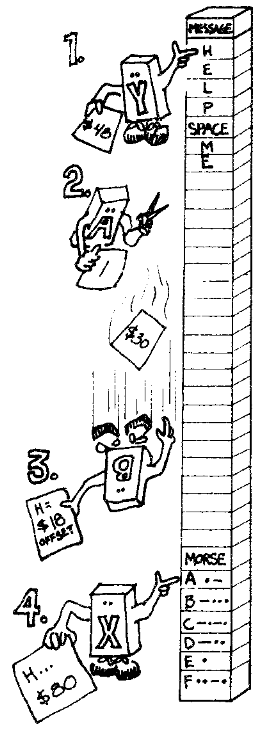
But even better is what you can do with the encoded beep information at the left of the byte. By rotating the byte to the left, the beep bits drop into the carry flag in head-first order. Dit-dah-dah-dit, represented by **0110**, rotates left and falls into the carry flag in the precise order **0-1-1-0**, or dit-dah-dah-dit. By using the carry flag as a condition for program branching, the process is assured. The program can branch-on-no-carry to a "dit"-length beep, and branch-on-carry to a "dah" length beep.

In actual program form, the message SOS might be encoded like this:

ASCII Letter	Morse Code	ASCII Code	Morse Binary	Morse Hex
S	...	$53	ØØØØØØ11	$Ø3
O	---	$4F	111ØØØ11	$E3
S	...	$53	ØØØØØØ11	$Ø3

I'd like you to take a break now and draw up a chart of all the ASCII characters and their Morse Code equivalents, as shown in the sample above. This will give you an intimate sense for the way in which this code is being assembled.

On a sheet of paper, list the characters in ASCII order, their Morse Code equivalents, their ASCII codes, their Morse binary encoding, and their encoded Morse hexadecimal representation. When you have completed the sheet, return to the tape.



Now you've got a complete cross-reference table in hand, and you understand the general workings of the program you've got to create. Let me review the structure so far.

1. Pluck an ASCII value from the message.

2. Find the encoded Morse equivalent in the table.

3. Use the length information in the rightmost three bits as a counter.

4. Shift the leftmost five bits into the carry flag.

5. Transmit dits or dahs based on the carry flag, and for the number of beeps held by the counter.

6. Pick up the next letter and continue.

This looks like a reasonable structure; it should resolve into this simplified program (follow me in the book):

```
START LDX   MORSE   * Encoded Morse in memory
      LDY   TEXT    * ASCII message in memory
AGAIN LDB   ,Y+     * Get ASCII, point to next
      SUBB  OFFSET  * Strip ASCII offset
      LDA   B,X     * B+X = Morse table position
      TFR   A,B     * Save encoded Morse
      ANDA  $Ø7     * Keep the code length
```

* What do ASCII codes $20 through $3F represent?

Numbers, symbols and punctuation.

* What do ASCII codes $40 through $5F represent?

Uppercase letters.

* The following questions deal with the specific program being created in this session.

* In the structure chosen for this example, where is the Morse Code length information stored?

In the rightmost three bits of a byte.

* Where is the actual Morse Code pattern stored?

In the leftmost five bits of a byte.

* How is the length information retrieved?

By masking the byte with binary 00000111.

* If the byte is in the A accumulator, what instruction is used to mask in the right three bits?

ANDA #$07

* "Dit" and "dah" are represented by what information?

"Dit" is zero, "dah" is one.

* Where is the "dit" and "dah" information stored?

In the leftmost five bits of a byte.

* How is the information retrieved?

By rotating the bits leftward into the carry flag.

* What is the advantage to having the code length on the right side of the byte?

It only needs to be masked, not shifted, to become the correct value.

* What is the advantage to having the code beeps on the left side of the byte?

There are two advantages: they are in place to be rotated left into the carry flag, and they are in the correct order from first to last as they are rotated into the carry flag.

* The letter S is dit-dit-dit. What is its length in binary?

Its length is three beeps, 011 binary.

* The letter S is dit-dit-dit. What is the binary equivalent of its beeps?

The binary equivalent of the beeps is 000.

* What is the complete encoded byte for letter S, pattern dit-dit-dit?

Pattern 000 plus two unused bits (00) plus the length 011. The result is 00000011.

* What is the hex equivalent of binary 00000011?

Binary 00000011 is hex $03.

* Name the timing considerations needed for Morse Code.

The length of the "dit", the length of the "dah", the length of silences, and the frequency of the beep.

```
NEWBIT ROLB          * Drop into carry
       BCS    JUMP1  * On C=1, do dah, do dah
       JSR    DIT    * On C=Ø, go do the dit
       BRA    NEXT   * .. and go past
JUMP1  JSR    DAH    * Here's the dah to play
NEXT   DECA          * Length = Length-1
       BNE    NEWBIT * Next bit if Length <> Ø
       JMP    AGAIN  * Back for next character
```

There are a few things missing from this structure. As shown, there are no spaces obvious between letters or words. Even if silences were included in the beep routines, there wouldn't be any break between streams of beeps. So silence must be added. And then there's the question of what to do when the message is finished. In my example, the transmission continues right through memory. It's got to be made to stop. To achieve a pause, I've selected a yet-to-be-written subroutine called QUIET. As for the message end, the greater-than and less-than characters aren't present in the Morse Code system. I've decided to use the greater than sign to indicate "end of message," and the less-than sign to mean "repeat message." With that in mind, I've got a program for you to load.

Program #22, 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.

List this program screen by screen using the P command. You'll find that I put the message at **$3000**, the Morse Code table at **$2F00**, and the program itself at **$2E00**. Examine this program carefully, and see if its compactness makes sense. Also, check your handwritten Morse table against mine. All that's left to write are the dit, dah, and silence subroutines. Till next time.

14.

I hope that you've had good luck creating the program to take an ASCII message and translate it into a series of subroutine calls, calls that would, once the final beeping routines are created, transmit Morse Code.

Just to review, you'll remember that the structure of the program was set up to read an ASCII message, character by character. It would then locate an encoded version of the Morse Code from an in-memory table, and use that information to produce a pattern of dits and dahs. The program you've created up to this point should look something like the one I have for you next.

Program #23, 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/0 error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix.

Let me take you through my program. I know that without real person-to-person interaction the things I've done and the things you've done won't match. You might feel like the work you have just finished is somewhat in vain. Not true. This is really the first program I've left you alone to structure, and it's invaluable that you contrast the two.

Last session I told you that some characters which didn't exist in the traditional Morse Code set might be ideal for using as end-of-message markers, to tell your program that the message was complete, and should either be ended or repeated. I chose to use the greater-than sign for end of message, and the less than sign for a continuous repeat of the message. Recall that the message itself would be stored beginning at **$3000**. As for the program, I suggested you put that at **$2E00**, leaving plenty of room for the program

Keep in mind that as this program is developed, translating codes and producing sound is the object. Whether it is Morse Code or any code isn't critical, and whether it's a beep or an entire musical tune isn't the point. You are finding out how to manipulate sound and make translations. Also, remember that the final program is going to make adjustments for that Morse Code punctuation which is six beeps long; these are exceptions which don't affect the heart of the program and its inherent simplicity.

* Name the timing considerations needed for Morse Code.

The length of the "dit", the length of the "dah", the length of silences, and the frequency of the beep.

* The letter Y is dah-dit-dah-dah. What is its length in binary?

Its length is four, binary 100.

* The letter Y is dah-dit-dah-dah. What is its binary beep pattern?

Its beep pattern is 1011.

* What is the complete encoded value for Y, pattern dah-dit-dah-dah?

Beep pattern 1011 plus one unused bit (0) plus a length of 100. The result is 10110100.

* What is 10110100 in hexadecimal?

10110100 is hexadecimal $B4.

* If the A accumulator contains $B4, and you execute ANDA #$07, what is the result?

The result is $04.

* If the B accumulator contains $B4, and you execute ROLB four times, what is the condition of the carry flag after each ROLB?

C=1, C=0, C=1, C=1.

* How long is a "dah" with respect to a "dit"?

Three times as long.

* Three things happen when an interrupt occurs. What are they?

The microprocessor finishes it current instruction, saves important information, and follows programming instructions in reponse to the interrupt.

* What is the process of acting on an interrupt called?

Servicing the interrupt.

* What is the proper term for turning off an interrupt?

Masking the interrupt.

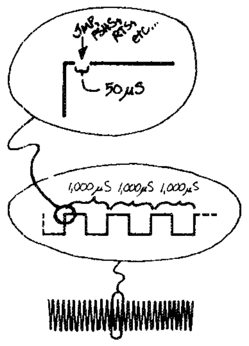
* What condition code bits determine whether interrupts are masked or enabled?

Bits 4 and 6.

and the Morse Code lookup table. I put the table at $2F00.

The program itself turns out to be surprisingly simple. There are three options for producing dits and dahs shown in my listing. The first option best represents the actual, expected circumstances — a dah is three times the length of a dit. Therefore, two separate subroutines, one dit and one dah, are created for this purpose. The other solutions might not be immediately obvious.

To understand the second option, consider that the real world we're dealing with here is lots slower than the computer world. In this case, the computer operations involved in determining the difference between dits and dahs, and the time required to call the dit subroutine, are minuscule. In fact, these operations are nearly inaudible in the course of a real-world beep. A jump to subroutine (JSR) and a return form subroutine (RTS) take only 13 clock cycles, and the two likely PUSH/PULL combinations used to save information before performing the beep subroutine itself add 28 more cycles. The total clock cycles, 41, demand under 50 microseconds for completion. Recall that I suggested a beep frequency of 1,000 Hz. That frequency means that each pulse that makes up this beep frequency is 1,000 microseconds — so these additional jumps and returns add only about 5 percent to the time it takes to create one single pulse of the beep. So you can see that in a case where the computer is much faster than a real-world event, alternative approaches such as this can simplify the actual machine code you must write.

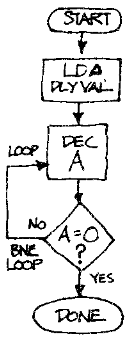


The third method is a favorite of programmers because it allows a subroutine to be an all-purpose building block. In this method, a value is given to the accumulator, a value which indicates a dit or a dah. The beeping routine then uses this value to calculate the overall loop length of dit or dah. In a different and more precise way, this subroutine performs s similarly efficient function to the previous one.

More on all of these when the actual beep-creation routines are assembled.

First, I'd like to turn to the problem of creating the beep itself. What is a beep? A beep is a tone, or a pitch — a rapid, consistent and regular fluctuation of air molecules. This isn't a lesson on acoustics, so I'll make it short. A rapid, consistent and regular compression and decompression of the air is perceived as a tone or pitch. A loudspeaker which is pushed forward and back rapidly, consistently and regularly will compress and decompress the air in a similar way. Electrical impulses which alternate between two voltage levels can create the necessary speaker motion. A computer program can provide those impulses.

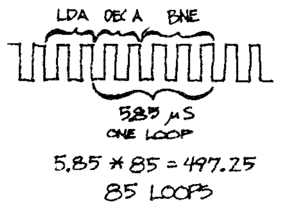
So that reveals the structure of the computer program, and also gets us — the long way 'round — to the question of time delays. Alternating between a one and a zero is a simple task, something you've done already. The task of the beeping program is to alternate between one and zero at a predictable rate — in this case, 1,000 Hz, or 1,000 alternations per second.



A simple delay loop in machine language might look like this (and you can follow along with me in the book):

```
            LDA     DLYVAL
LOOP        DECA
            BNE     LOOP
```

The A accumulator contains the delay value. The BNE instruction loops back until A equals zero. This simple delay allows a maximum loop of 256 iterations — the largest 8-bit number the A accumulator can hold. According to the MC6809E data book, LDA immediate requires 2 clock cycles to complete, DECA takes 2 clock cycles, and BNE needs 3 cycles. The goal of the delay is 500 microseconds total. LDA only happens once, so that leaves about 497 microseconds to go. The DECA/BNE combination of 5 clock cycles is 5.85 microseconds on the Color Computer, meaning a total of 85 loops (497.25 microseconds) does the trick. The value for label DELAY, then, is 85 decimal. We won't fix this in concrete yet, though, because certain bits and pieces of the program that might add extra delay to the process haven't been written yet. But it's working delay information for now.



You might have picked up on my delay of 500 microseconds. If the beep is 1,000 Hz, then a complete pulse is 1000 microseconds. A complete pulse. That means one pulse up for air compression, and one pulse down for air decompression . . . a total of 1,000 microseconds, 500 microseconds for the up pulse, 500 microseconds for the down pulse.

Now how about the length of the beep? Eventually that's going to vary, too, but for the moment, let's make a dit one-fifth of a second, and a dah three-fifths of a second. Since the beep is 1,000 Hz, or 1,000 pulses per second, then one-fifth of a second is just 200 pulses. So the program begins to look like this (again, follow me in the book):

```
           LDB     2ØØ    * LENGTH
OUTLP      LDA     85     * FREQUENCY
INLP1      DECA           * DONE YET?
           BNE     INLP1  * WAIT
           LDA     85     * FREQUENCY
INLP2      DECA           * DONE YET?
           BNE     INLP2  * WAIT
           DECB           * BEEP END?
           BNE     OUTLP  * MORE
```

There's still no actual beeping going on here because I haven't described how to do it. For this you need to recall the detailed memory map presented several lessons

* What command specifically masks out (turns off) both interrupts?

ORCC #$50 (binary 01010000).

* What command specifically enables (turns on) both interrupts?

ANDCC #$AF (binary 10101111).

* What is the clock speed of the Color Computer?

.89 MHz (894,886 clock pulses per second).

* What does Hz mean?

Hz means Hertz, or cycles (pulses) per second.

* What does MHz mean?

MHz means megaHertz, or million cycles per second.

* Why does sound output require timing?

Because sound is made up of specific frequencies, and frequencies are inherently time-based.

* How long is the shortest BASIC beep (using SOUND X,1)?

Approximately 1/14 of a second.

* If a loop contains two load A immediates, two store A extendeds, and one branch to make a complete loop, how many clock cycles are required for this loop?

2 times 2 cycles, plus 2 times 5 cycles, plus 3 cycles ... a total of 17 cycles.

* At 894,886 clock cycles per second, how many loops is this?

894,886 divided by 17, or 52,640 cycles.

* At 894,996 clock cycles per second, how many clock cycles are available to produce a 1,000-Hz beep?

894,996 divided by 1,000, or about 895 clock cycles.

* If a beep frequency is 1,000 Hz, how long is each "pulse" of sound.

Each pulse is 1/1000th of a second (1 millisecond or 1,000 microseconds).

* How long is one Color Computer clock cycle?

Approximately 1.11746 microseconds.

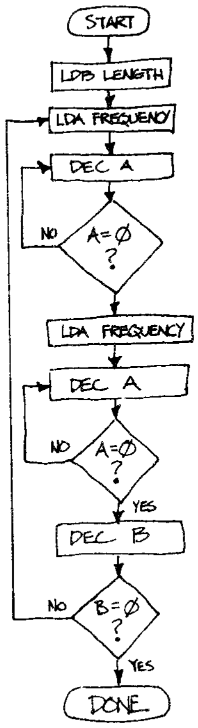
* How many clock cycles pass by in 1,000 microseconds?

Approximately 895.

earlier. You can turn back to that later; for the moment I'll tell you that "single bit sound," that is, sound produced by pulsing on and off one bit in a memory location, is found at address **$FF22**. Alternations of one and zero made at bit one of location **$FF22** will be heard on the television speaker or the cassette output.

So once again you discover that the subroutine becomes surprisingly simple in its final form. It's coming up next. After you've got the program loaded, take some time to look at it, then come back to the tape.

Program #24, 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/0 error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix.



```
2E00                00100           ORG     $2E00
                    00110 *
2E00 1A    50       00120 BEGIN     ORCC    #$50     * TURN OFF BOTH INTERRUPTS
                    00130 *
2E02 86    32       00140           LDA     #$32     * 0011 0010 SETS DD REGISTER
2E04 B7    FF23     00150           STA     $FF23    * ACCESS DATA DIR. REGISTER
2E07 86    FA       00160           LDA     #$FA     * 1111 1010 SETS S.B. OUTPUT
2E09 B7    FF22     00170           STA     $FF22    * TURN ON SINGLE BIT SOUND
2E0C 86    36       00180           LDA     #$36     * 0011 0110 SETS PD I/O
2E0E B7    FF23     00190           STA     $FF23    * RESTORE I/O CONFIGURATION
                    00200 *
2E11 C6    64       00210 BEEP      LDB     #100     * GET OUTSIDE BEEPWAVE VALUE
2E13 86    28       00220 OUTLP     LDA     #40      * GET INSIDE BEEPWAVE LENGTH
2E15 4A             00230 INLP1     DECA             * DECREMENT INSIDE DELAY
2E16 26    FD       00240           BNE     INLP1    * AND WAIT TO TOTAL LOOPS
2E18 86    02       00250           LDA     #$02     * GET HIGH PART OF BEEPWAVE
2E1A BA    FF22     00260           ORA     $FF22    * OR WITH PORT OUTPUT STATUS
2E1D B7    FF22     00270           STA     $FF22    * AND OUTPUT THE RESULT
2E20 86    28       00280           LDA     #40      * GET ANOTHER DELAY VALUE
2E22 4A             00290 INLP2     DECA             * AND COUNT DOWN THRU DELAY
2E23 26    FD       00300           BNE     INLP2    * WAIT THROUGH TOTAL LOOPS
2E25 86    FD       00310           LDA     #$FD     * SET LOW PART OF BEEPWAVE
2E27 B4    FF22     00320           ANDA    $FF22    * AND WITH CURRENT STATUS
2E2A B7    FF22     00330           STA     $FF22    * AND OUTPUT LOW BEEPWAVE
2E2D 5A             00340           DECB             * DECREMENT NUMBER OF WAVES
2E2E 26    E3       00350           BNE     OUTLP    * AND GO BACK TILL ALL DONE
2E30 39             00360           RTS              * AND BACK TO PROGRAM
           2E00     00370           END     BEGIN
00000 TOTAL ERRORS
BEEP    2E11
BEGIN   2E00
INLP1   2E15
INLP2   2E22
OUTLP   2E13
```

* What is a beep?

A tone; a pitch; a rapid, consistent and regular fluctuation of air molecules.

* What electrical device produces a beep?

A loudspeaker.

As it stands now, this program should be set up to provide a single beep. First, assemble this program in memory, at the origin shown. Type A/IM/AO, and hit ENTER. That's A/IM/AO. Now quit the editor/assembler. Type Q and hit enter.

You'll be in BASIC. To produce that beep, you have to remember the origin of the program. Type three lines:

```
10 EXEC&H2E00
20 FORX=1TO20:NEXT
30 GOTO10
```

Turn up the television volume, and RUN this program. You should hear a continuous series of beeps. As you listen, though, you won't hear that old familiar gargle in the sound. The interrupts have been turned off; the sound is pure. The routine works. What's left is to combine the beep routine, the Morse Code table lookup routine, and all the rest into a complete, usable program.

We'll be going on to complete the full program next, so it would be a good time to take a break if you wish to review. After that, load the program.

Program #25, 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.

* What causes the loudspeaker to produce a beep?

Electrical impulses which alternate voltage levels.

* Can a computer program provide electrical impulses?

Yes.

* How are the electrical impulses represented?

By binary ones and zeros.

* Two LDA immediates, two STA extendeds, and BNE make a 16-cycle loop; it might LDA #0 and #1 alternately and STA to an address to make the speaker produce sound. If 1,000 microseconds is 895 clock cycles, but the loop is only 16 cycles, what else is needed?

A delay.

```
3E00                00100           ORG     $3E00
                    00110 *
3E00 1A   50        00120 BEGIN     ORCC    #$50     * TURN OFF BOTH INTERRUPTS
                    00130 *
3E02 86   32        00140           LDA     #$32     * 0011 0010 SETS DD REGISTER
3E04 B7   FF23      00150           STA     $FF23    * ACCESS DATA DIR. REGISTER
3E07 86   FA        00160           LDA     #$FA     * 1111 1010 SETS S.B. OUTPUT
3E09 B7   FF22      00170           STA     $FF22    * TURN ON SINGLE BIT SOUND
3E0C 86   36        00180           LDA     #$36     * 0011 0110 SETS PD I/O
3E0E B7   FF23      00190           STA     $FF23    * RESTORE I/O CONFIGURATION
                    00200 *
3E11 8E   3E99      00210 START     LDX     #MORSE   * POINT TO MORSE CODE TABLE
3E14 108E 3ED4      00220           LDY     #TEXT    * POINT TO MESSAGE IN MEMORY
3E18 E6   A0        00230 AGAIN     LDB     ,Y+      * GET VALUE FROM THE TEXT
3E1A C1   3C        00240           CMPB    #$3C     * CHECK IF LESS-THAN SIGN
3E1C 27   F3        00250           BEQ     START    * IF "<", THEN REPEAT MESSAGE
3E1E C1   3E        00260           CMPB    #$3E     * CHECK IF GREATER-THAN SIGN
3E20 27   43        00270           BEQ     OUT      * IF ">", RETURN TO BASIC
3E22 C0   20        00280           SUBB    #$20     * STRIP OFFSET (TABLE $20-$5B)
3E24 A6   85        00290           LDA     B,X      * ENCODED MORSE FROM TABLE
3E26 27   2C        00300           BEQ     SPACE    * IF 00, THEN SEND A SILENCE
3E28 34   02        00310           PSHS    A        * SAVE ENCODED MORSE VALUE
3E2A C1   10        00320           CMPB    #$10     * CHECK IF PUNCTUATION
3E2C 23   08        00330           BLS     EXCEPT   * IF SO, GO TO EXCEPTION
3E2E C1   1A        00340           CMPB    #$1A     * CHECK IF A COLON
3E30 27   04        00350           BEQ     EXCEPT   * IF SO, GO TO EXCEPTION
3E32 C1   1F        00360           CMPB    #$1F     * CHECK IF A QUESTION MARK
3E34 26   05        00370           BNE     NORMAL   * IF NOT, PROCEED NORMALLY
3E36 84   03        00380 EXCEPT    ANDA    #$03     * EXCEPTION USES RIGHT 2 BITS
3E38 48             00390           ASLA             * LEFT SHIFT CREATES NUMBER 6
3E39 20   02        00400           BRA     GETVAL   * NOW READY FOR THE ACTION
3E3B 84   07        00410 NORMAL    ANDA    #$07     * NORMAL USES RIGHT 3 BITS
3E3D 35   04        00420 GETVAL    PULS    B        * RESTORE ENCODED MORSE VALUE
3E3F 59             00430 NEWBIT    ROLB             * ROTATE BIT TO CARRY FLAG
3E40 25   04        00440           BCS     JUMP1    * IF SET, THEN JUMP TO DAH
3E42 8D   1D        00450           BSR     DIT      * OTHERWISE SEND A DIT
3E44 20   02        00460           BRA     NEXT     * AND GO TO THE NEXT BIT
3E46 8D   15        00470 JUMP1     BSR     DAH      * THEN SEND THE DAH
3E48 4A             00480 NEXT      DECA             * SEE IF DONE WITH ALL BEEPS
3E49 26   F4        00490           BNE     NEWBIT   * IF NOT, THEN GET ANOTHER
3E4B C6   04        00500           LDB     #$04     * OTHERWISE GET TIMING VALUE
3E4D 8D   3B        00510 LETRLP    BSR     QUIET    * AND CALL INTER-LETTER PAUSE
3E4F 5A             00520           DECB             * DECREMENT PAUSE COUNTER
3E50 26   FB        00530           BNE     LETRLP   * AND LOOP BACK TILL DONE
```

```
3E52 20    C4       00540          BRA     AGAIN   * THEN GO BACK FOR MORE TEXT
                    00550 *
3E54 C6    08       00560 SPACE    LDB     #$08    * 8 SILENCES FOR A SPACE
3E56 8D    32       00570 SPCLP    BSR     QUIET   * AND GO SEND THE SILENCE
3E58 5A             00580          DECB            * DECREMENT THE SILENCE COUNT
3E59 26    FB       00590          BNE     SPCLP   * AND LOOP BACK TILL DONE
3E5B 20    BB       00600          BRA     AGAIN   * AND GO GET NEXT TEXT
                    00610 *
3E5D 8D    07       00620 DAH      BSR     BEEP    * PERFORM FIRST 1/3 BEEP
3E5F 8D    05       00630          BSR     BEEP    * PERFORM SECOND 1/3 BEEP
3E61 8D    03       00640 DIT      BSR     BEEP    * PERFORM DIT OR LAST 1/3
3E63 8D    25       00650          BSR     QUIET   * AND PUT IN A SILENCE
3E65 39             00660 OUT      RTS             * BACK TO PROGRAM (OR BASIC)
                    00670 *
3E66 34    06       00680 BEEP     PSHS    A,B     * SAVE COUNT AND MORSE CODE
3E68 C6    64       00690          LDB     #100    * GET OUTSIDE BEEPWAVE VALUE
3E6A 86    28       00700 OUTLP    LDA     #40     * GET INSIDE BEEPWAVE LENGTH
3E6C 4A             00710 INLP1    DECA            * DECREMENT INSIDE DELAY
3E6D 26    FD       00720          BNE     INLP1   * AND WAIT TO TOTAL LOOPS
3E6F 86    02       00730          LDA     #$02    * GET HIGH PART OF BEEPWAVE
3E71 BA    FF22     00740          ORA     $FF22   * OR WITH PORT OUTPUT STATUS
3E74 B7    FF22     00750          STA     $FF22   * AND OUTPUT THE RESULT
3E77 86    28       00760          LDA     #40     * GET ANOTHER DELAY VALUE
3E79 4A             00770 INLP2    DECA            * AND COUNT DOWN THRU DELAY
3E7A 26    FD       00780          BNE     INLP2   * WAIT THROUGH TOTAL LOOPS
3E7C 86    FD       00790          LDA     #$FD    * SET LOW PART OF BEEPWAVE
3E7E B4    FF22     00800          ANDA    $FF22   * AND WITH CURRENT STATUS
3E81 B7    FF22     00810          STA     $FF22   * AND OUTPUT LOW BEEPWAVE
3E84 5A             00820          DECB            * DECREMENT NUMBER OF WAVES
3E85 26    E3       00830          BNE     OUTLP   * AND GO BACK TILL ALL DONE
3E87 35    06       00840          PULS    A,B     * RESTORE COUNTER AND MORSE
3E89 39             00850          RTS             * AND BACK TO PROGRAM
                    00860 *
3E8A 34    06       00870 QUIET    PSHS    A,B     * SAVE COUNTER AND MORSE
3E8C C6    64       00880          LDB     #100    * GET OUTSIDE DELAY VALUE
3E8E 86    64       00890 QLP1     LDA     #100    * GET INSIDE DELAY VALUE
3E90 4A             00900 QLP2     DECA            * COUNT DOWN THRU INNER LOOP
3E91 26    FD       00910          BNE     QLP2    * AND WAIT FOR THE COUNT
3E93 5A             00920          DECB            * COUNT DOWN THRU OUTER LOOP
3E94 26    F8       00930          BNE     QLP1    * AND WAIT FOR THE COUNT
3E96 35    06       00940          PULS    A,B     * RESTORE COUNTER AND MORSE
3E98 39             00950          RTS             * BACK TO MAIN PROGRAM
                    00960 *
3E99       00       00970 MORSE    FCB     $00     * SPACE
3E9A       00       00980          FCB     $00     * ! = SPACE
3E9B       4B       00990          FCB     $4B     * " (.-..-.) (010010 11) **
3E9C       00       01000          FCB     $00     * # = SPACE
3E9D       00       01010          FCB     $00     * $ = SPACE
3E9E       00       01020          FCB     $00     * % = SPACE
3E9F       00       01030          FCB     $00     * & = SPACE
3EA0       7B       01040          FCB     $7B     * ' (.----.) (011110 11) **
3EA1       B7       01050          FCB     $B7     * ( (-.--.-) (101101 11) **
3EA2       B7       01060          FCB     $B7     * ) (-.--.-) (101101 11) **
3EA3       00       01070          FCB     $00     * * = SPACE
3EA4       00       01080          FCB     $00     * + = SPACE
3EA5       CF       01090          FCB     $CF     * , (--..--) (110011 11) **
3EA6       87       01100          FCB     $87     * - (-....-) (100001 11) **
3EA7       57       01110          FCB     $57     * . (.-.-.-) (010101 11) **
3EA8       93       01120          FCB     $93     * / (-..-..) (100100 11) **
3EA9       FD       01130          FCB     $FD     * 0 (-----) (11111 101)
3EAA       7D       01140          FCB     $7D     * 1 (.----) (01111 101)
3EAB       3D       01150          FCB     $3D     * 2 (..---) (00111 101)
3EAC       1D       01160          FCB     $1D     * 3 (...--) (00011 101)
3EAD       0D       01170          FCB     $0D     * 4 (....-) (00001 101)
3EAE       05       01180          FCB     $05     * 5 (.....) (00000 101)
3EAF       85       01190          FCB     $85     * 6 (-....) (10000 101)
3EB0       C5       01200          FCB     $C5     * 7 (--...) (11000 101)
3EB1       E5       01210          FCB     $E5     * 8 (---..) (11100 101)
3EB2       F5       01220          FCB     $F5     * 9 (----.) (11110 101)
3EB3       E3       01230          FCB     $E3     * : (---...) (111000 11) **
3EB4       00       01240          FCB     $00     * ; = SPACE
3EB5       00       01250          FCB     $00     * < = SPACE
3EB6       00       01260          FCB     $00     * = = SPACE
3EB7       00       01270          FCB     $00     * > = SPACE
3EB8       33       01280          FCB     $33     * ? (..--..) (001100 11) **
3EB9       00       01290          FCB     $00     * @ = SPACE
3EBA       41       01300          FCB     $41     * A (.-) (01 000 001)
3EBB       84       01310          FCB     $84     * B (-...) (1000 0 100)
3EBC       A4       01320          FCB     $A4     * C (-.-.) (1010 0 100)
3EBD       83       01330          FCB     $83     * D (-..) (100 00 011)
3EBE       01       01340          FCB     $01     * E (.) (0 0000 001)
3EBF       24       01350          FCB     $24     * F (..-.) (0010 0 100)
3EC0       C3       01360          FCB     $C3     * G (--.) (110 00 011)
```

```
3EC1      04        01370          FCB      $04      * H (....) (0000 0 100)
3EC2      02        01380          FCB      $02      * I (..) (00 000 010)
3EC3      74        01390          FCB      $74      * J (.---) (0111 0 100)
3EC4      A3        01400          FCB      $A3      * K (-.-) (101 00 011)
3EC5      44        01410          FCB      $44      * L (.-..) (0100 0 100)
3EC6      C2        01420          FCB      $C2      * M (--) (11 000 010)
3EC7      82        01430          FCB      $82      * N (-.) (10 000 010)
3EC8      E3        01440          FCB      $E3      * O (---) (111 00 011)
3EC9      64        01450          FCB      $64      * P (.--.) (0110 0 100)
3ECA      D4        01460          FCB      $D4      * Q (--.-) (1101 0 100)
3ECB      43        01470          FCB      $43      * R (.-.) (010 00 011)
3ECC      03        01480          FCB      $03      * S (...) (000 00 011)
3ECD      81        01490          FCB      $81      * T (-) (1 0000 001)
3ECE      23        01500          FCB      $23      * U (..-) (001 00 011)
3ECF      14        01510          FCB      $14      * V (...-) (0001 0 100)
3ED0      63        01520          FCB      $63      * W (.--) (011 00 011)
3ED1      94        01530          FCB      $94      * X (-..-) (1001 0 100)
3ED2      B4        01540          FCB      $B4      * Y (-.--) (1011 0 100)
3ED3      C4        01550          FCB      $C4      * Z (--..) (1100 0 100)
                    01560 * NOTE:
                    01570 * THE ITEMS MARKED WITH A DOUBLE ASTERISK (**) ARE
                    01580 * PROCESSED BY THE EXCEPTION BEEPING ROUTINE.
                    01590 *
3ED4      59        01600 TEXT     FCC      /YOU ARE LISTENING TO THE /
3EED      4D        01610          FCC      /MICRO LANGUAGE LAB, PRESENTED
3F0B      42        01620          FCC      /BY GREEN MOUNTAIN MICRO.)/
                    01630 *
          3E00      01640          END      BEGIN
00000 TOTAL ERRORS
AGAIN    3E18
BEEP     3E66
BEGIN    3E00
DAH      3E5D
DIT      3E61
EXCEPT   3E36
GETVAL   3E3D
INLP1    3E6C
INLP2    3E79
JUMP1    3E46
LETRLP   3E4D
MORSE    3E99
NEWBIT   3E3F
NEXT     3E48
NORMAL   3E3B
OUT      3E65
OUTLP    3E6A
QLP1     3E8E
QLP2     3E90
QUIET    3E8A
SPACE    3E54
SPCLP    3E56
START    3E11
TEXT     3ED4
```

* If the A accumulator contains a delay value, what is the simplest possible delay procedure?

Decrement A accumulator, and branch on not equal back to decrement A accumulator. When A reaches zero, the loop ends.

* How long is this delay loop (excluding the original load A accumulator)?

2 cycles (DECA) plus 3 cycles (BNE) is 5 cycles, that is 5 times 1.11746 or 5.5873 microseconds. (The less precise value of 5 times 1.15 gives 5.75 microseconds).

By this point the program should be no surprise. Display it using the P command. You'll notice little new; the program has been structured precisely along the lines of the original description. There are just two additions: a comparison is made to find out if the ASCII value in the message represents a greater-than sign or a less-than sign. On greater than, the transmission is completed, the program is ended and returned to BASIC; on less than, the message is repeated.

The other addition to the program is the comparison for **$20**, the ASCII value for a space. There is no space in Morse Code, but for purposes of clarity in transmission, a little extra time is traditionally inserted between words. If a space is found in the ASCII text, then, the QUIET routine

* How many loops would be required for a delay of 500 microseconds?

500 microseconds divided by 5.5873, or approximately 89 (not including the remainder of the program; 85 is used as a test value in the example).

* Why is a delay of 500 microseconds used instead of 1,000 microseconds?

Because the sound alternates between 1 and 0, half the time on and half the time off. 500 microseconds delay is needed for each half.

* What are the Color Computer port addresses?

$FF00, $FF02, $FF20 and $FF22.

* At port $FF22, what is the purpose of bit one?

It is used for sound output.

* What action causes sound?

Alternations of one and zero made at bit one of port $FF22.

* Where is the sound heard?

On the television speaker or through the cassette output.

* What is the term for "setting up" a computer device?

Configuring.

* What PIA address configures port $FF22?

Address $FF23.

* What does PIA mean?

Peripheral Interface Adapter.

is called to insert a slight pause in the transmission.

Interestingly, this program represents quite closely the actual human process from which is was derived. A person reads a message, recalls the code, and transmits it with a series of accurate, trained muscular movements. Programming is not always such a parallel to real life; enjoy it this time.

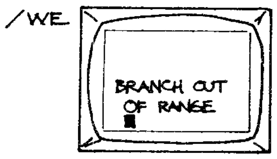
I'd like to take the remaining time to review editor/assembler assembly commands. I haven't presented these before, but you have undoubtedly come across them as you worked with EDTASM+ while completing this program.

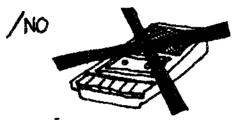
EDTASM's command to assemble your mnemonics into machine language is "A". The "A" command has a number of options called "switches". If you enter simply the letter "A", the assembler will display a scrolling assembled listing, will provide a table of all labels you've used in the source code, note any errors you've generated, and — assuming you have a tape recorder connected and ready — will prepare a cassette containing the final binary object code. The object code tape will be called "NONAME", and will load under BASIC's CLOADM command.

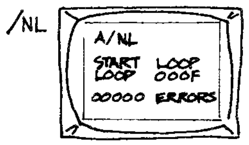
There are many other options that this one. The format of the "A" command is A, space, filename, switch. For example, to assemble and save the resulting machine-language program to tape under the name "DISPLAY", you would type A DISPLAY <ENTER>.

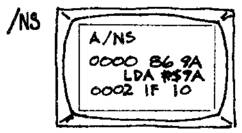
The switches are two-letter command options separated by slashes. These are:

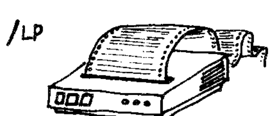
/WE	Wait for errors. The display stops if you have made an error in an opcode, an operand, a range, a typo, etc. The assembler will display a descriptive error message.
/NO	No object code tape is created. I use this switch until I have eliminated all errors picked up by the assembler.
/NL	No listing is displayed. Especially during correction of minor errors, or when you only want to see a list of labels, you can turn off the long listing.
/NS	No symbol table is displayed. The symbol table is the proper name for a list of all the labels used in the program, together with the addresses at which they appear.
/LP	Line printer command. Everything that is displayed on screen is also sent to the Color Computer's serial printer.
/IM	In-memory assembly. This is an excellent debugging tool. The program is not only assembled into binary code, it is also placed directly in memory, ready to run.
/AO	Absolute origin. If you do an in-memory assembly, you can let the machine assemble the program at its predetermined location,

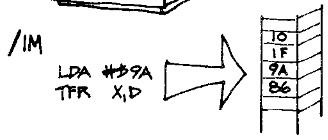


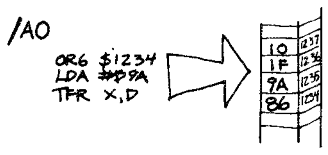






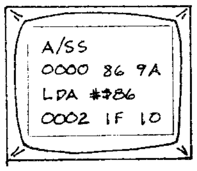






	or at the memory location you specified in your ORG statement. Absolute origin uses your ORG.
/MO	Manual origin. This permits you to move the source code, tables, and so forth, that EDTASM needs to work with, so that your own program doesn't conflict with it.
/SS	This is the short screen option. Finding your way through an assembling program with the Color Computer's 32-character screen can be messy. The short screen places the assembled hex address, opcodes and operand on a line by themselves, with the mnemonics out of the way on the next line.

/ss



For details on all these commands, of course, read pages 13 through 16 of your EDTASM+ manual. For the moment, I want you to try the In-Memory assembly option, at your own origin, for the Morse Code program and for as many other programs as you would like to try up to this point. Next time, a new topic.

* Name the timing considerations necessary to produce a beep.

The length of the beep and the frequency of the beep.

* The following questions refer to EDTASM+ assembly (A) commands.

* What does A/NO produce?

An assembled listing and symbol table.

* What does A produce?

An assembled listing, symbol table, and object code sent to the cassette.

* What does A/NL/NS/NO produce?

Only a report of errors at the end of the assembly. No listing appears on the screen.

* What does A/NL/NO produce?

A listing of the symbol table.

* What is the symbol table?

A listing of all labels used in a program, together with the addresses at which they appear.

* What does A/NO/LP produce?

A complete assembled listing send to the printer.

* What does A/IM/AO produce?

A screen listing of the assembled program and symbol table, as well as an object code placed in memory at the origin specified in the source code.

* What does A/SS produce?

A screen listing in "short screen" format, where lines are broken up for easier reading; a symbol table and object code to cassette are also produced.

* What does A/WE produce?

A listing, symbol table, and object code; it also stops at any line in which an error occurs.

* What does A mean?

Assemble.

* What is assembly?

The translation of source (mnemonic) code into binary (object) code.

15.

Hands-on programming takes a back seat for this lesson as the topic once again returns to addressing modes. For this session, you'll want to have your MC6809E data booklet out again. Turn to pages 15 through 17. On pages 15 through 17, where you've previously learned about inherent, immediate, extended, direct, relative, and indexed addressing, you'll also find information about additional applications of those modes.

These remaining addressing modes are called indirect addressing. "Indirect" is an excellent description of this concept, because the operand is not the data (as in immediate addressing), nor is the operand the address of the data (as in extended addressing). No, in the case of indirect addressing, the operand is an address which points to an address where the data can be found. Once again. The operand is an address. That address points to another address. In turn, that address points to the data.

This is easier to understand through example than description. In fact, I've already introduced indirect addressing, but not by name. Recall how I described the power-up of the 6809 microprocessor. When the power is turned on, I said, the processor immediately identifies addresses **$FFFE** and **$FFFF**. It concatenates the 8-bit data found at these addresses, producing a 16-bit number. That 16-bit number becomes the address of the first instruction the processor is to follow. That's indirect addressing.

I know this method, properly called "indexed indirect", sounds like a clumsy and roundabout way of getting information. It's not clumsy, but it is roundabout, and that roundaboutness is its precise advantage. Let's say you've got a super-high-speed action game in the writing, and you need to make moves based on keyboard input. We'll talk about keyboard input itself later, but imagine for the moment that the numbers 0 to 9 are crucial in your game. Say each number causes an entirely different game action, such as shooting balls or using flippers in some sort of arcade pinball. You could, of course, check the value of each number, and if it fits, jump off to a routine. It might

One of the most important differences among the dozen or so popular microprocessors is their respective architectures. The 6809's architecture is created to facilitate finding data, and its myriad addressing modes are key to finding data. Indexed addressing is part of your programming library already; indirect addressing is coming up, together with information on handling and manipulating high-resolution graphics.

* Where does the 6809's instruction decoder get its instructions?

From memory.

* What memory locations does the processor use when the power is turned on?

It uses $FFFE and $FFFF when the power is turned on.

* What does the processor get from memory location $FFFE?

The most significant byte of a 16-bit number.

* What does the processor get from memory location $FFFF?

The least significant byte of a 16-bit number.

Indexed indirect

* What does the processor do with the two bytes from $FFFE and $FFFF?

It concatenates them.

* What is the result of concatenating the bytes from $FFFE and $FFFF?

A 16-bit number.

* What does the 16-bit number represent?

The address of the first instruction the processor will execute.

* The processor obtains the two bytes at $FFFE and $FFFF, concatenates them, and uses the 16-bit result as an address. What addressing mode is this?

Indirect addressing.

* What addressing mode is LDX #$1234?

Immediate addressing.

* What addressing mode is LDX $1234?

Extended addressing.

* What addressing mode is LDX ($1234)?

Indirect addressing.

* In the immediate addressing mode as represented by LDX #$1234, where is the data?

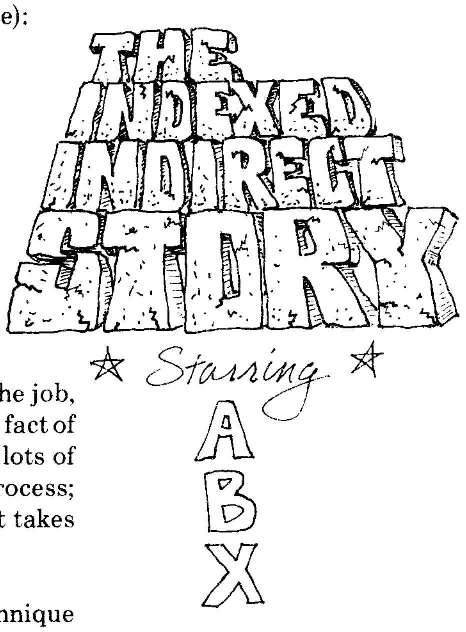
Immediately following the opcode, that is, the data is $1234.

* In the extended addressing mode as represented by LDX $1234, where is the data?

At address $1234 and $1235.

look something like this (follow me in the book here):

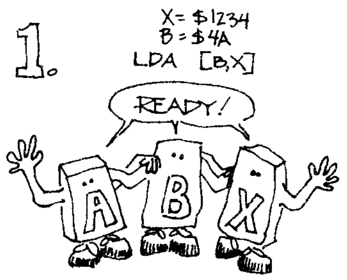
```
        CMPA    $3Ø       * 2 bytes
        BNE     NEXT1     * 2 bytes
        JMP     FLIP      * 3 bytes
NEXT1   CMPA    $31
        BNE     NEXT2
        JMP     FLAP
NEXT2   CMPA    $32
        BNE     NEXT3
        JMP     FLOP
```

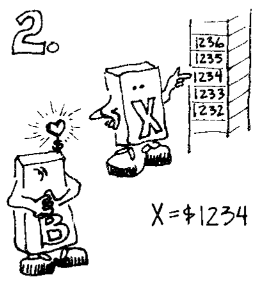


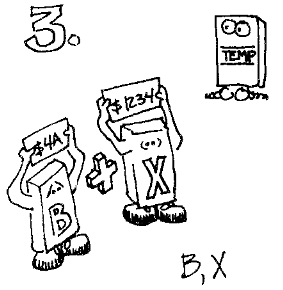
. . . and so on. This kind of programming would do the job, and naturally it would be quite fast due to the simple fact of its being written in machine language. But it grabs lots of memory, and, if timing is critical, it is an uneven process; that is, getting to the last possible choice in the list takes more machine time than getting to the first choice.

There's an entirely different and very powerful technique available with indexed indirect. Consider this. You can load an index register X or Y with the zeroeth element of a table of subroutine addresses, subtract an ASCII offset from your accumulator, double A, and simply jump to the address indexed indirectly by X plus A. Look in the book:

```
        LDX     TABLE   * Addresses
        SUBA    $3Ø     * Strip ASCII
        ROLA            * Double A
        JMP     A,X     * Indexed indirect

TABLE   FCB     $1234   * Subroutine #Ø
        FCB     $1366   * Subroutine #1
        FCB     $1A9C   * Subroutine #2
        FCB     $2ØEF   * etc......
```

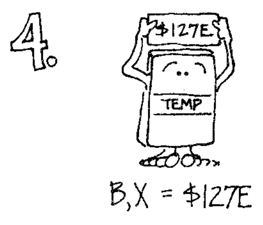


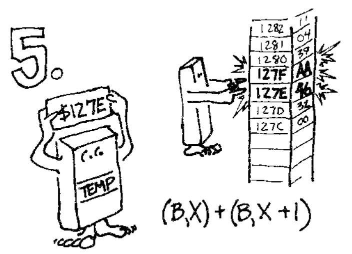


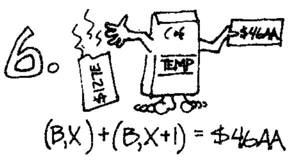


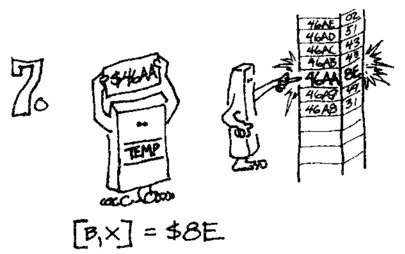
The A accumulator is rotated left (that is, doubled) because it takes two bytes to create an address. The indexing process needs to skip every two bytes. Observe that the original compare-branch-or-jump routine takes 70 bytes for ten choices. This indexed indirect routine has the advantage of being more regular and much faster, yet it takes only 29 bytes. For long or fast programs, the savings in time and memory can be significant, and for timed programs, the regularity can be meaningful. Let's put it to work.

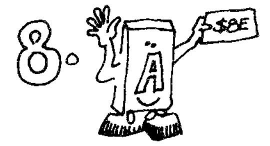
The program will be the Game of Life, a nifty set of rules by which theoretical populations of cells are born, live, and die. The rules are simple. First, this mythical population lives in a regular, two-dimensional grid. On this grid, which can be imagined simply as intersecting horizontal and vertical lines, any given cell position is surrounded by 8 other cell positions. Three "live" cells will give birth in any cell they surround; two or three surrounding cells will keep that cell alive. If the neighborhood population grows over three cells, or falls under two cells, the cell dies.

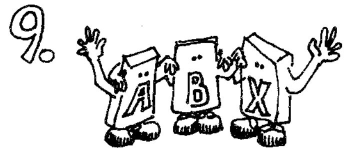
4.


5.


6.


7.


8.


9.


The
~End~

These simple rules can cause an incredible number of predictable population patterns to arise. Civilizations grow and shrink, rise and fall. Some stabilize in tiny colonies, or rise to great empires. On a video screen, these changes can transform a random population into an astoundngly regular ebb and flow. It becomes hypnotic.

You don't play the Game of Life. Once you have created an initial population, it plays itself until the populations have stabilized in life or death.

And, with such a simple set of rules, it becomes a perfect computer application. The set of rules by which any applicaton is completed is called an algorithm. Let me reiterate the algorithm for the Game of Life.

1. Where three cells surround an empty position, cell birth takes place.

2. Where two or three cells surround a live cell, life goes on.

3. When the surrounding population drops below two, a cell dies.

4. When the surrounding population rises above three, a cell dies.

I'd first like you to see this in slow motion. I've got a BASIC program for you.

> Problem #26, 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 PMODE0:PCLEAR1:DIMA(65,33)
20 FORX=1024TO1535
30 POKEX,127+RND(17):NEXT
40 FORX=1TO62:FORY=1TO30
50 IFPOINT(X,Y)<>0THEN60ELSE140
60 A(X-1,Y-1)=A(X-1,Y-1)+1
70 A(X,Y-1)=A(X,Y-1)+1
80 A(X+1,Y-1)=A(X+1,Y-1)+1
90 A(X-1,Y)=A(X-1,Y)+1
100 A(X+1,Y)=A(X+1,Y)+1
110 A(X-1,Y+1)=A(X-1,Y+1)+1
120 A(X,Y+1)=A(X,Y+1)+1
130 A(X+1,Y+1)=A(X+1,Y+1)+1
140 NEXT:NEXT
150 FORX=0TO63:FORY=0TO31
160 IFA(X,Y)<2THENRESET(X,Y):GOTO190
170 IFA(X,Y)=3THENSET(X,Y,1):GOTO190
180 IFA(X,Y)>3THENRESET(X,Y)
190 NEXT:NEXT
200 FORX=0TO65:FORY=0TO33
210 A(X,Y)=0:NEXT:NEXT:GOTO40
```

* In the indirect addressing mode as represented by LDX ($1234), where is the data?

At the address determined by concatenating the bytes found at $1234 and $1235.

* If addresses $1234 and $1235 contain $A0 and $00 respectively, and if addresses $A000 and $A001 contain $7F and $F0 respectively, what does the X register contain after LDX #$1234?

X contains $1234.

* If addresses $1234 and $1235 contain $A0 and $00 respectively, and if addresses $A000 and $A001 contain $7F and $F0 respectively, what does the X register contain after LDX $1234?

X contains $A000.

* If addresses $1234 and $1235 contain $A0 and $00 respectively, and if addresses $A000 and $A001 contain $7F and $F0 respectively, what does the X register contain after LDX ($1234)?

X contains $7FF0.

* What specific addressing mode is LDX $1234?

Extended addressing.

* What specific addressing mode is LDX ($1234)?

Extended indirect addressing.

* What specific addressing mode is LDA ,Y?

Zero-offset indexed.

* What specific addressing mode is LDA (,Y)?

Zero-offset indexed indirect addressing.

* There are basic rules to the Game of Life. What is the general term for a set of rules?

An algorithm.

* There are four parts to the Game of Life algorithm, each involving a cell position on a grid. How many potential neighbors are there in a regular, two-dimensional square grid?

Eight neighbors.

* What do three neighbors produce in a dormant cell?

A cell "birth".

* What do two or three neighbors produce in a live cell?

No change.

* What do less than two neighbors produce?

A cell "death".

* What do more than three neighbors produce?

A cell "death".

* What is the general term for a set of rules?

An algorithm.

* What does VDG mean?

Video Display Generator.

* The VDG performs what functions?

The display of alphanumeric characters, and the display of several resolutions of color and monochrome graphics.

* What does SAM mean?

Synchronous Address Multiplexer.

RUN this program. As you watch, a random population is generated in low-resolution graphics. This is the starting population, the garden of Eden, if you will. Once the population has been established, the Game of Life begins. As you watch, I'll tell you that the Game of Life is now a traditional computer problem, originally invented and proposed by British mathematician John Conway. His proposal delighted computer people at the time, and continues to be fascinating as more detailed color screens and more capable computers are developed.

The populations you are watching develop slowly, since BASIC must make a large number of simple comparisons and calculations for which it is ill-suited. Doing such calculations by hand can take hours per generation. Yet the simple-mindedness of machine language finds this a fertile area.

The process of moving from generation to generation is made up of one overall task: check the "neighborhood" of cells, so to speak, and than maintain the status quo, give birth to cells, or kill cells. There are many ways of dealing with that task, however. You might check by neighborhood, or by cell, or look for the presence of any population in an area. Statistically, the Game of Life more often results in a lesser number of live cells — at least after the Garden of Eden has been created and the generational growth has begun.

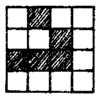
My old friend and teacher Phil approached the algorithm from this point of view. It complicated the programming slightly, but sped along the real time required to move from generation to generation. That's the approach I'm going to use for this example, so I'd like you to keep it in mind. As you progress with 6809 assembly language, you might like to give the Game of Life a try using other approaches.

To begin with the Game of Life on the Color Computer, you have to know how to establish the degree of screen resolution you wish to use, and how that mode is manipulated. This is especially important when using the 6847 video display generator because each graphics mode has a different number of colors and a different manner of dealing with how the bits in a memory byte are reflected on the screen. In your notebook, find the MC6847 video display generator data sheet, and turn to page 19.

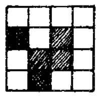
These modes should already be familiar to you from an earlier lesson, but you should review them now. Also, you'll want to look in the MC6883 data sheet to Appendix A on pages 20 through 22. Because of the way the SAM and the VDG are connected, these special modes are also available. Take time now to read the information on graphic display modes.

Turn to the MC6847 video display generator (VDG) data booklet and read the information on page 19. Also, open the MC6883 synchronous address multiplexer (SAM) data booklet and read Appendix A, pages 20 through 22. Return to the tape when you have completed the reading.

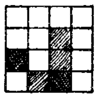
I'd like to select detailed, regular and square picture elements. The highest resolution mode offers individual pixels, but I'd also like color so that different generations and empty cells are shown in different colors — empty cells in black, perhaps, births in yellow and established cells in blue. The mode labeled CG3 offers a 2x2 pixel in four colors. I'll use it.


GARDEN OF EDEN

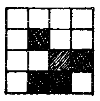
Now, rules in hand and video mode selected, I can structure the neighborhood counting process. To create this screen, 3072 bytes are required to produce 12,288 screen points, at a resolution of 128 by 96. In the screen memory, combinations of bit pairs determine the color, and four bit pairs fill a byte.


GENERATION 1

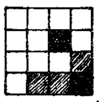
By now, you should be able to establish mode CG3 and select the screen memory using the upper memory address and the SAM registers. To accomplish this, remember to refer to both the MC6847 VDG data booklet and the memory map in the MC6883 SAM data booklet. I won't take time for that here; you'll be able to double-check your results against the program listing in the next lesson.


GENERATION 2

Among the other things to establish is the color set — that is, which set of four possible colors to display. The sets are green, yellow, blue and red for set #0; buff, cyan, magenta, and orange make up set #1. The choice for color set is specified in the Color Computer memory map as bit 3 of output port address **$FF22**.


GENERATION 3

Some arbitrary decisions must be made. I've selected addresses **$0000** through **$0BFF** for the video display; that address has to be presented to the SAM. Recall that the SAM contains write-only registers which are set or reset to produce the 7-bit upper portion of the display address. Review the MC6883 data sheet if you need to refresh that information.


GENERATION 4

And finally, interrupts must be turned off to speed the execution of the program. Again, the details of all these setup routines will be shown in the final program in the next lesson; you should attempt to do them in the meantime.

Let me give you a summary now of the pre-program setup:

1. Interrupts must be disabled.

* On the Color Computer, there are two considerations necessary to establish VDG modes and colors. What are they?

Port $FF22 and the SAM video registers.

* What bit of port $FF22 selects the color set?

Bit 3.

* What are the SAM memory addresses called?

Write-only registers.

* Name the addresses of the SAM write-only registers that control the video display offset.

Addresses $FFC6 to $FFD3.

* What is the video display offset address the address of?

The upper-left-most picture element shown on the video display screen.

* What is the term for "picture element"

Pixel.

* In the most detailed mode, what is the pixel size of the video screen (pixels wide by pixels high)?

256 pixels wide by 192 pixels high.

* What is the pixel size in mode CG3 (color graphics 3)?

2 pixels by 2 pixels.

* What is the size of the screen in mode CG3 (width by height)?

128 wide by 96 high.

2. One of two possible color sets must be selected. This program will use set #1 for greatest definition.

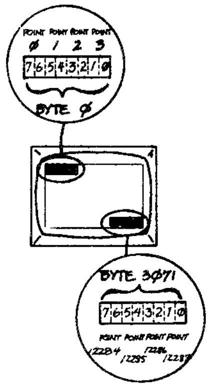
3. The display screen memory must be defined. Screen memory will run 3,072 bytes from **$0000** to **$0BFF**.

4. The color graphics modes must be established. Color Graphics 3 will be used, binary mode 100, to achieve a screen resolution of 12,288 points in four colors.

The final setup information is actually the Garden of Eden population itself. Because there's memory garbage and other information present upon powering up into BASIC or EDTASM+, you'll be able to use that residual material as the Garden of Eden. Creation of random numbers is a subject for later in this course; so until then, Life begins in the garbage pile of memory.

I'll be speaking often about scratchpad memory. Also called a work area, scratchpad memory acts as temporary storage for calculated information on the way to a final result. For example, long division on a microprocessor is a fairly complicated task, and there aren't enough registers inside the 6809 processor to complete it. All the temporary quotients, remainders, and so forth, are stored in a working arithmetic area. When you call for the answer to a complicated mathematical formula in BASIC, the working area required can be thirty or forty bytes, in addition to temporary stack storage of the results from within each and every set of parentheses.

In this version of the Game of Life, 3072 bytes of display memory represent 12,288 screen display points. In other words, each screen display point requires two bits of a byte. These quarter bytes provide for economical use of memory, but are more time-consuming to handle in a program because they have to be shifted left or right, or masked, or whatever, to retrieve their information. Point #0 on the screen is byte zero, bits 7 and 6; point #2 is byte zero, bits 5 and 4; point #12287 on the screen is byte #3071, bits 1 and 0. The relationship isn't difficult, but program handling can be.



Exclusively for reasons of speed, then, I chose to set up a scratchpad memory 12,288 bytes long — one byte for each point on the screen. Although it's wasteful of memory, it's very speedy because my "neighbor" information is immediately accessible in raw form. Since there are from 0 to 8 neighbors for every point, I could have used nybbles, but I chose to use the whole byte to avoid the time required for rotating and masking.

I've also made an arbitrary decision to choose **$1000** to **$3FFF** for these 12,288 bytes of scratchpad memory. That's hex **$3000** bytes. So the screen display runs from **$0000** to **$0BFF** and the scratchpad runs from **$1000** to

Scratchpad memory

* How many different points are displayed on the screen in mode CG3?

12,288 points.

* How many bytes are required for the screen in mode CG3?

3,072 bytes.

* How many colors are available in mode CG3?

Four colors.

* What are the four colors of VDG color set #0?

Green, yellow, blue and red.

* What are the four colors of VDG color set #1?

Buff, cyan, magenta and orange.

* What bit of what port address selects the color set to be displayed?

Bit 3 of port $FF22.

* In the Game of Life algorithm, what causes cell birth in a dormant cell?

Three immediate neighbors.

* In the Game of Life algorithm, what causes cell death?

Either less than two or more than three immediate neighbors.

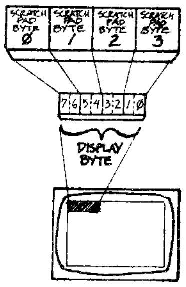
* In the Game of Life algorithm, what causes no change to a living cell?

Either two or three immediate neighbors.

* What is another name for a work area of memory?

A scratchpad.

$3FFF. All the memory that's left for the program itself is **$0C00** to **$0FFF**. I'll put the program at **$0C80**.



So I'd like you to begin by writing a program beginning at **$0C80** to perform the setup, and to clear scratchpad memory to zero. Once again, the setup is in four steps: disable interrupts, select color set #1, point display memory to begin at **$0000**, and choose color graphics mode 3. Scratchpad memory runs from **$1000** to **$3FFF** and must be filled with zeroes.

Before I leave you with this project, I'd like to suggest that although there is a simple way to fill memory, there is a faster but less obvious one. The simple way is to load an accumulator with the value needed to fill memory, to point the X or Y register to the start of that memory, and to store-and-increment your way through.

The less obvious method is to use the 6809's fast and powerful stack instructions. Re-examine the stack instructions in the MC6809E data book, including their opcodes and speed, and — without looking ahead to my solution in the next lesson — work out both ways of filling up that scratchpad memory.

* In display mode CG3, how many display positions are represented by one byte?

Four display positions are represented by one byte.

* How many bits of a byte are necessary for each display position in mode CG3?

Two bits are necessary for each display position.

* Why two bits?

Because mode CG3 displays four colors, and all combinations of two bits are necessary to display four colors.

* What addressing mode is LDX #$1234?

Immediate addressing.

* What addressing mode is LDX $1234?

Extended addressing.

* What addressing mode is LDX ($1234)?

Indirect addressing.

* What specific addressing mode is LDX $1234?

Extended addressing.

* What specific addressing mode is LDX ($1234)?

Extended indirect addressing.

* What specific addressing mode is LDA ,Y?

Zero-offset indexed.

* What specific addressing mode is LDA (,Y)?

Zero-offset indexed indirect addressing.

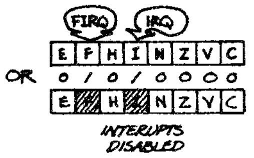
16.

Although we're still pretty far from its actual use in this program, I want to remind you that the current topic is indexed indirect addressing. Indexed indirect is the mode where the operand is the address of a memory location, and the contents of the pair of sequential 8-bit memory locations make up an address which is the eventual location of the data. For example, say register X points to memory location **$3000**. Say that memory location **$3000** contains byte **$AB** and memory location **$3001** contains byte **$99**. Now say finally that memory location **$AB99** contains byte **$FF**. Load A accumulator zero-offset indexed indirect to X would result in A containing **$FF**. I've got some illustrations of that concept in the book, and the program we're writing should help understand the usefulness of the technique.

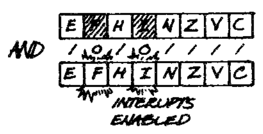
By this session you should have prepared the setup information and the scratchpad memory clearing. Your setup should look something like mine. To disable interrupts, you would . . .

```
            ORCC    #$5Ø
```

. . . which sets bits 4 and 6 of the condition code register.



To choose color set #1, you need to set bit 3 of port address **$FF22**. Furthermore, bits 4, 5 and 6 are the graphics mode selection bits, and bit 7 is the alphanumeric/graphic selector. If you hadn't taken a look at all these control bits, then now's the time. Turn to page 15 of your MC6883 data booklet (page 15 of the SAM data booklet).



There are 16 different display modes presented on this page, all but one available in two color sets. That gives you 31 choices. This wide selection is only available in computers where the 6847 video display generator and the 6883 SAM are used together; both are smart circuits, and so they interact in complex and versatile ways. The mode I've selected for the Game of Life is full color graphics 3. If you follow down on the chart, you'll see full graphics 3C, and the required bit conditions. The detailed memory map shows these bits; I'll remind you that the MC6847 modes

Indirect addressing is coming along. There's still color graphics mode details to deal with, and some ideas in quickly moving data from place to place. Clock cycles will come into play in the evaluation of speed -- since we all want graphics and speed to go hand in hand. There are also reviews of former information to cover, so that this programming doesn't move along too fast.

* What addressing mode is LDX $1234?

Extended addressing.

* What addressing mode is LDX ($1234)?

Indirect addressing.

* What specific addressing mode is LDX $1234?

Extended addressing.

* What specific addressing mode is LDX ($1234)?

Extended indirect addressing.

(the first five columns) are, respectively, bits 7, 6, 5, 4 and 3.

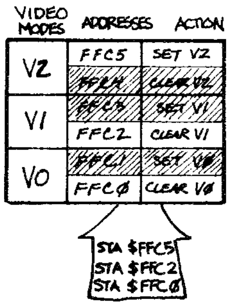
So to achieve mode G3C, bit 7 must be high, bit 6 is high, bits 5 and 4 are low, and bit 3 is up to you. Bit 3 is high for color set #1. So the left five bits of the binary number created for port **$FF22** is **11001**. The rightmost three digits are powered up to 111 on a 16K machine. So the complete binary number to select full color graphics mode 3, color set 1, is **11001111**, or hex **$CF**. The instructions are simplicity itself . . .

```
LDA     #$CF
STA     $FF22
```

. . . easily selecting the proper modes.

But in the process, don't forget the SAM. It has to be properly programmed as well. According to the chart you've been looking at, mode G3C requires that SAM bits V2, V1 and V0 be programmed binary 100. Turn the page in the SAM data booklet, and look at the map on page 17. Addresses **$FFC0** through **$FFC5** control the SAM modes. To set mode G3C, then, set bit V2 and clear bits V1 and V0. That means, remembering the SAM's write-only register technique, write to addresses **$FFC5**, **$FFC2**, and **$FFC0**. So you store any value to **$FFC5**, **$FFC2** and **$FFC0** . . .

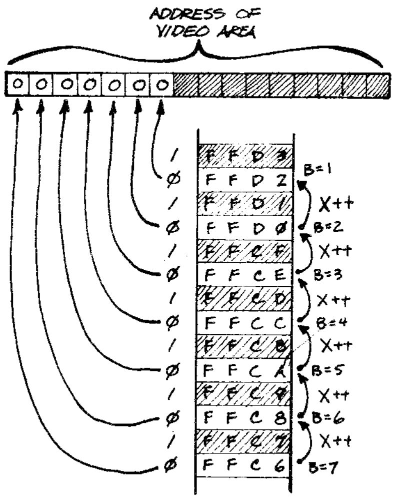
```
STA $FFC5
STA $FFC2
STA $FFCØ
```



The final setup information is to choose the display memory starting address, which I've selected to be found at **$0000**. The display offset information is provided by the SAM, so that setup information must be written to the SAM. That is done — again recalling one of early lessons — by writing to addresses **$FFC6** through **$FFD2**. To establish starting address **$0000** means that binary values **0000 000** must be put into the seven SAM address offset positions. To place a zero in the SAM — that is, to clear a bit — you write to an even-numbered address. To place a one in the SAM — to set a bit — you write to an odd-numbered address. As you've seen, the display memory calls for binary **0000 000**, so that calls for writing to addresses **$FFC6**, **$FFC8**, **$FFCA**, **$FFCC**, **$FFCE**, **$FFD0** and **$FFD2**.

The most straighforward way of doing this might be to store an accumulator at each location . . . **STA $FFC6**, **STA $FFC8**, **STA $FFCA**, etc. Since all even addresses are being set, I've chosen this case-specific solution. . .

```
          LDB  #$Ø7
          LDX  #$FFC6
VIDEO     STA  ,X++
          DECB
          BNE  VIDEO
```



SETTING THE VIDEO OFFSET

* What specific addressing mode is LDA ,Y?

Zero-offset indexed.

* What specific addressing mode is LDA (,Y)?

Zero-offset indexed indirect addressing.

* What is necessary to choose VDG color set #1 on the Color Computer?

Setting bit 3 of port address $FF22 chooses color set #1.

* Which bits of port $FF22 select the graphics modes?

Bits 4, 5 and 6 select the graphics modes.

* Which bit of port $FF22 selects between alphanumerics and graphics?

Bit 7 of port $FF22 selects alphanumerics or graphics.

* Name my three cats.

Aida, Mehitabel and Beulah.

* There are basic rules to the Game of Life. What is the general term for a set of rules?

An algorithm.

* There are four parts to the Game of Life algorithm, each involving a cell position on a grid. How many potential neighbors are there in a regular, two-dimensional square grid?

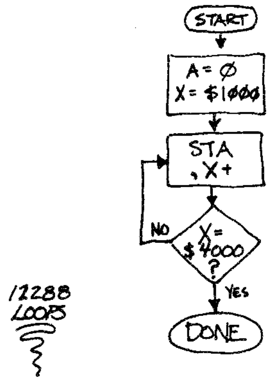
Eight neighbors.

* What do three neighbors produce in a dormant cell?

A cell "birth".

. . . which writes to every other even-numbered address for a total of seven. The B register does the counting, and the X register points to the first SAM video address. For contrast, have a look at the general-purpose example shown at the bottom of page 16 in the SAM data booklet.

This completes the pre-program setup; take a break now and review this process, especially if your solutions are substantially different from mine.

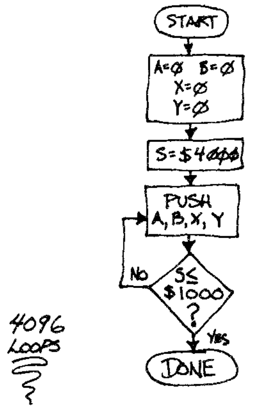


Last time I suggested you have a look at stack instructions and see how they might be used to fill memory. First, here's a standard method of doing a memory fill from **$1000** to **$3FFF** (you can follow along in the book):

```
        CLRA           * Set A to Ø
        LDX   #$1ØØØ   * Point X to ;b;$1ØØØ
LOOP    STA   ,X+      * Store zero
        CMPX  #$4ØØØ   * See if finished
        BNE   LOOP     * Go back til done
```

The main time-consuming part of this routine consists of the last three instructions, requiring 6, 4 and 3 cycles respectively. The total of 13 cycles is repeated 12,288 times, for a total of 159,744 cycles. At 894,886 clock cycles per second, this operation takes a considerable 178.5 milliseconds . . . nearly one-fifth of a second. For fast action games, that isn't.

Consider this solution instead (follow me in the book):



```
        CLRA            * Set A to Ø
        CLRB            * Set B to Ø
        TFR   D,X       * Set X to Ø
        TFR   D,Y       * Set Y to Ø
        LDS   #$4ØØØ    * Point S to top
LOOP    PSHS  A,B,X,Y   * Push 6 bytes
        CMPS  #$1ØØØ    * See if bottom
        BPL   LOOP      * Back until lower
```

After clearing A, B, X and Y to zero, the S stack is pointed to the top of the memory area to be cleared. Remember that the stack pushes down from the top. A, B, X and Y are then pushed on the stack using one instruction. The stack is compared immediate with **$1000**, the bottom of scratchpad memory, and if the result is plus (if S is greater than or equal to **$1000**), the routine is repeated.

The number of clock cycles required for the PSHS instruction is 5 plus 1 additional for each byte pushed. Six bytes are pushed in total, meaning that PSHS A,B,X,Y takes 11 cycles to complete. So the heart of this memory fill routine requires 11, 5 and 3 cycles . . . a total of 19. 19 cycles is longer than the 13 needed for the previous example. But remember that in this case, *six* bytes are pushed at once.

* What do two or three neighbors produce in a live cell?

No change.

* What do less than two neighbors produce?

A cell "death".

* What do more than three neighbors produce?

A cell "death".

* What addresses control the SAM video modes?

Addresses $FFCØ through $FFC5.

* What addresses control the SAM video display offset address?

Addresses $FFC6 through $FFD3.

* What is the video display offset address the address of?

The upper-left-most picture element shown on the video display screen.

* What is the term for "picture element"

Pixel.

* In the most detailed mode, what is the pixel size of the video screen (pixels wide by pixels high)?

256 pixels wide by 192 pixels high.

* What is the pixel size in mode CG3 (color graphics 3)?

2 pixels by 2 pixels.

* What is the size of the screen in mode CG3 (width by height)?

128 wide by 96 high.

Joe's neighborhood

* How many different points are displayed on the screen in mode CG3?

12,288 points.

* How many bytes are required for the screen in mode CG3?

3,072 bytes.

* How many colors are available in mode CG3?

Four colors.

* What are the four colors of VDG color set #0?

Green, yellow, blue and red.

* What are the four colors of VDG color set #1?

Buff, cyan, magenta and orange.

* The push and pull S stack commands require an operand, plus what additional information?

A postbyte.

* What information is contained in the postbyte?

Bits indicating which registers are to be pushed or pulled.

* How many registers can be pushed or pulled?

Eight registers.

* What are the eight registers which can be pushed or pulled?

PC, U, Y, X, DP, B, A and CC

* How many bytes are involved in pushing all eight registers?

12 bytes (2 each for PC, U, Y and X, 1 each for DP, B, A and CC).

12,288 divided by 6 is 2,048 . . . there are only 2,048 repetitions of this routine. 2,048 times 19 is 38,912 clock cycles; again, at 894,886 clock cycles per second, this instruction completes in only 43.5 milliseconds. That's slightly less than one-quarter the time of the previous method, just one-twentieth of a second.

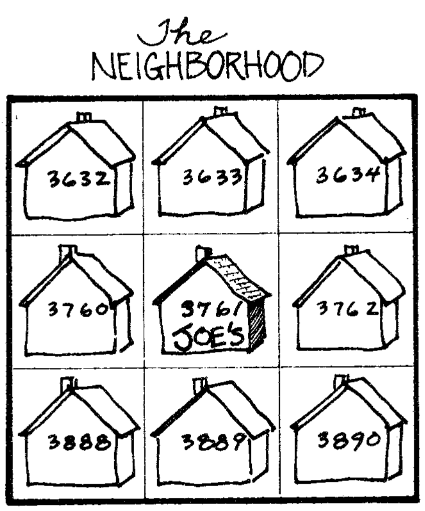
So where you need to fill blocks of memory very quickly, the push-stack method is ideal. Don't forget to save the stack pointer if you need to, and also to replace the stack pointer when you're done with it. In this program, I put the stack pointer at **$0DBF** when I'm finished with it:

```
LDS   #$ØDBF
```

If you would like to look through these examples, this is a good time to stop and do that.

At this point, the Garden of Eden is populated, video display setup is complete, interrupts are disabled, scratchpad memory is cleared, and the stack is in position. It's time to evaluate the Garden of Eden for the population of its neighborhoods. I'll summarize the Hooper technique.

If Joe lives in a house on this regular memory grid, then he's potentially got a neighbor to the northwest, north, and northeast; to the west and east; and to the southwest, south, and southeast. Eight neighbors in all. The screen grid in this graphics mode is 128 by 96, 128 houses across by 96 houses down. If Joe lives in house #3761, then he's got potential neighbors in houses 3761 minus 129 (that's northwest), 3761 minus 128 (that's north), 3761 minus 127 (that's northeast). There's a house to the west at 3761 minus 1, and a house to the east at 3761 plus 1. Finally, there are houses to the southwest at 3761 plus 127, to the south at 3761 plus 128, and to the southeast at 3761 plus 129.



I'll convert those to hex. The screen is hex **$80** by **$60**, so Joe's got neighbors at −**$81**, −**$80**, −**$7F**, −**$01**, +**$01**, +**$7F**, +**$80**, and +**$81**. If the Y index register points to Joe, then the neighbor offsets would be:

-81	-80	-7F
-1	0	+1
+7F	+80	+81

```
           -$81,Y
           -$8Ø,Y
           -$7F,Y
             -1,Y
              1,Y
            $7F,Y
            $8Ø,Y
and         $81,Y
```

The process, then, is really a kind of inverse of this. If those eight are Joe's potential neighbors, then Joe is the neighbor of those eight. So instead of going to every cell and evaluating all eight neighbors, you can go to every cell and see if it is alive (that's the key). If it's alive, you increment the neighbor count; if not, you move along to the next.

So instead of making 12,288 checks of 8 neighbors, you make 12,288 checks for life. So only if a cell is live does the action become:

```
INC    -$81,Y
INC    -$8Ø,Y
INC    -$7F,Y
INC      -1,Y
INC       1,Y
INC     $7F,Y
INC     $8Ø,Y
INC     $81,Y
```

This is the heart of the neighborhood scratchpad routine. But think back to the actual display screen. Only 3,072 bytes are used to display the 12,288 cells. Somehow you've got to break these into quarter bytes very quickly, and evaluate them. You might choose a mask-and-shift strategy, where each pair of bits is shifted left, then masked and evaluated. You might choose a mask-and-compare strategy, where four separate routines are used to evalute the four separate quarter-bytes. Both methods would work, but in addition to masking or shifting, each technique would require saving and restoring the original value, testing, and branching.

Ø	Ø	DORMANT
Ø	/	NEWBORN
/	Ø	"DEVIANT"
/	/	MATURE

The method I've selected takes advantage of the rotate instruction, which rotates the bits of a byte around in a circle — but through the carry flag. You can take a look at the MC6809E data booklet to see exactly how the ROL and ROR instructions work. The advantage here is that, by carefully selecting how I represent live and dormant cells, I can rotate bits of the display byte through the carry flag and use the carry to branch to the proper routine. Rather than spend time explaining the concept, I'll take you right to the routine itself. Look in your book. As you examine the program excerpt, keep in mind that I've defined **00** as a dormant cell, **01** as a newborn cell, and **11** as a mature cell

* How many clock cycles does a push or pull operand use?

5 cycles.

* How many additional clock cycles are required for each register pushed or pulled?

1 cycle for each register pushed or pulled.

* How many cycles are required to execute the instruction PSHS PC,U,Y,X,DP,B,A,CC ?

17 cycles are required.

* How long is one Color Computer clock cycle?

1.11746 microseconds.

* How long does the instruction PSHS PC,U,Y,X,DP,B,A,CC take on the Color Computer?

About 19 microseconds (18.99 microseconds).

* How long would it take to fill 6,144 bytes of memory using PSHS PC,U,Y,X,DP,B,A,CC at 18.99 microseconds per instruction?

9723 microseconds (.ØØ9723 seconds), or about 1/1ØØ of a second.

* How many clock cycles is PSHS A,B,X,Y?

5 plus 6, or 11.

* How long is PSHS A,B,X,Y?

11 times 1.11746, or 12.3 microseconds.

* How long would filling 6,144 bytes of memory take using PSHS A,B,X,Y?

19446 microseconds (.Ø19446 seconds), or about 1/5Ø of a second).

Rotate to test

* It is theoretically possible to fill memory merely by executing a long series of PSHS A,B,X,Y. However, a comparison and branch would be required, resulting in a sequence like this:

```
LOOP   PSHS A,B,X,Y
       CMPS #$2000
       BPL  LOOP
```

How long would one iteration of this sequence take?

11 cycles plus 5 cycles plus 3 cycles, or 19 cycles total; 19 cycles times 1.11746 microseconds is 21.23 microseconds.

* How long would it take the above sequence to fill 6,144 bytes?

(21.23 microseconds per loop) times (6,144 bytes divided by 6 bytes per loop) is about 21,739 microseconds, about .02 seconds.

* The Game of Life uses 12,288 bytes. How long would it take to fill 12,288 bytes using this kind of sequence?

About .043 seconds.

* If the S stack pointer is set to $1000 and the instruction PSHS A,B,X,Y is executed, where will the S stack pointer be at the conclusion of the instruction?

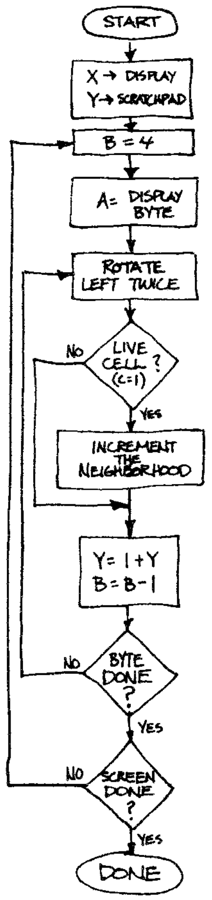
At $1000 minus 6, or $0FFA.

* Why $0FFA instead of $1006?

Because the stacks move downward in memory; they are push-down stacks.

(that is, past the first generation).

```
        LDX    #$0000    * Point X to display
        LDY    #$1000    * Point Y to scratchpad
NXTCEL  LDB    #4        * Count four quarter bytes
        LDA    ,X+       * Get video display byte
        PSHS   CC        * Save carry flag info
QUARTR  PULS   CC        * Restore carry flag info
        ROLA             * Rotate A through carry
        ROLA             * And rotate A again
        PSHS   CC        * Save carry (part of byte)
        BCC    NEXTQ     * If C=0, then cell not live
        INC    -$81,Y    * Northwest neighbor
        INC    -$80,Y    * Northern neighbor
        INC    -$7F,Y    * Northeast neighbor
        INC    -$01,Y    * Western neighbor
        INC    +$01,Y    * Eastern neighbor
        INC    +$7F,Y    * Southwest neighbor
        INC    +$80,Y    * Southern neighbor
        INC    +$81,Y    * Southeast neighbor
NEXTQ   LEAY   1,Y       * Get next scratchpad position
        DECB             * Count down quarter bytes
        BNE    QUARTR    * Get next quarter byte
        PULS   CC        * Else restore stack info
        CMPX   #$0C00    * See if at end of display
        BNE    NXTCEL    * Get next value
```



X and Y are pointed to display and scratchpad, respectively. The B accumulator serves as a quarter-byte counter. The A accumulator holds the byte from display to be evaluated.

Now here's the trick. The value in A is rotated left twice, through the carry flag. That leaves the rightmost bit of the display pair sitting in the carry flag, and the leftmost bit of the pair sitting in the bit **0** position of the accumulator. If the carry is clear, the cell is either dormant (**00**) or defined as illegal (**10**); in either case, it is not a neighbor, so the routine moves down to the label NEXTQ. If the carry is set, then the cell is either a newborn (**01**) or a mature cell (**11**), and the eight neighborhood incrementing instructions are completed. The label NEXTQ follows. More about the instruction "Load effective address" later; what you see effectively increments Y by one. The quarter-byte counter is decremented, and the rotate-and-branch routine is repeated until four quarter bytes have been done. The **CMPX #$0C00** tests for the end of the 3,072 byte display area. That's it. At the end of 3,072 groups of four quarter-byte tests, 12,288 bytes of scratchpad memory will be filled with neighborhood information.

To understand this process more intimately, take some time to draw a small grid of display points excerpted from the screen (say 16 by 16), a corresponding page of memory bytes (it would be 4 by 16), and a chart of scratchpad memory. Put some random cells in place on the display grid, then determine the display bytes. Finally, evaluate the results in scratchpad memory. In the next lesson, you'll do the actual neighborhood checking and updating.

17.

Hello again. I hope you aren't impatient with this step-by-step approach. In this lesson, you'll finally be getting to the application of indexed indirect addressing, and be completing the Game of Life. The result will be surprisingly short — under 240 bytes — and quite fast.

We left off having performed all the setups: disabling interrupts, selecting color graphics mode 3 with color set 1 (12,288 points in buff, cyan, magenta, orange), video display address **$0000**. 12,288 bytes of scratchpad memory has been cleared and filled with neighborhood information, that is, values 0 to 8.

Once the neighborhood values have been determined, that information is used to give birth to a cell, to allow a cell to become dormant, or to leave the cell unchanged. As with all programming, there are many ways to make this happen. And, as always, the most obvious solution isn't necessarily the fastest or the most efficient. The obvious solution is something like this . . .

```
LDA     ,Y
BEQ     DEATH
DECA
BEQ     DEATH
DECA
BEQ     NO CHANGE
DECA
BEQ     BIRTH
DECA
```

. . . and so on. Another technique — and a fast one — would have started by filling the scratchpad memory with **$FE** instead of **$00**. In this circumstance, zero or one neighbors would result in the scratchpad value being left with **$FE** or **$FF**. Two neighbors would produce **$00** in the scratchpad, three neighbors would yield **$01**, and more than three neighbors would produce **$02**. A much quicker method, the final routine would look like this . . .

Creating longer programs like this one can be time-consuming "up front", but care taken at this stage will assure a partly functioning — if not perfect — result when you type EXEC. No, this Game of Life didn't work for me the first time. But the screen showed proper modes and colors, and there was generation-to-generation motion. It wasn't right, but there was enough to begin serious debugging. If you spend your first few hours creating structure, then outlining modules, and finally stringing the pieces together, chances are your program will begin to show evidence of life from your first EXEC.

* What instructions are used to turn interrupts on and off?

ORCC and ANDCC.

* Why does ORCC #$50 turn off interrupts?

Because setting bits 4 and 6 of the condition code register turns off interrupts; #$50 is 01010000, so ORCC #$50 sets bits 4 and 6 without altering the other six bits.

* What addressing mode is JMP $3456?

Extended addressing.

* What addressing mode is JMP A,X?

Indexed addressing.

* What addressing mode is JMP (A,X)?

Indexed indirect addressing.

* X points to $1234. A is set to 4. The memory locations $1230 through $123F contain $01 02 45 47 93 96 A2 01 11 F5 36 92 19 5E 22 00. Where is the program counter after JMP $1234?

PC is at $1234.

* X points to $1234. A is set to 4. The memory locations $1230 through $123F contain $01 02 45 47 93 96 A2 01 11 F5 36 92 19 5E 22 00. Where is the program counter after JMP ($1234)?

PC is at $9396.

* X points to $1234. A is set to 4. The memory locations $1230 through $123F contain $01 02 45 47 93 96 A2 01 11 F5 36 92 19 5E 22 00. Where is the program counter after JMP A,X?

PC is at $1238.

* X points to $1234. A is set to 4. The memory locations $1230 through $123F contain $01 02 45 47 93 96 A2 01 11 F5 36 92 19 5E 22 00. Where is the program counter after JMP (A,X)?

PC is at $11F5.

```
        LDA    ,Y        * Get value from scratchpad
        BRN    DEATH     * If negative ($FE or ;b;$FF), then death
        BEQ    NO CHANGE* If zero ($00), then no change
        DECA             * Decrement A to set flags
        BEQ    BIRTH     * If zero ($01 minus 1), then birth
        BRA    DEATH     * Otherwise is ;b;$02 or greater
```

There's a lot in that short routine, and it's very fast. In fact, in this situation, it's a tossup in speed to the one I've chosen. Depending on the value in the scratchpad, it can take three, six, 11 or 14 clock cycles to complete; my sample method always takes 12 cycles. My guess is that in a "mature" civilization, the former method would be faster. But since this is a lesson on indirect indexed addressing, then indirect indexed addressing it is.

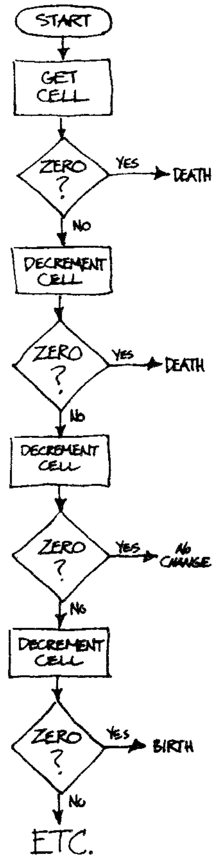
Indexed indirect looks like this . . .

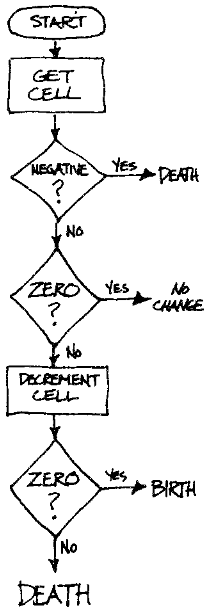
```
        ASLA             * Double A (two-byte offset)
        LDX    #TABLE    * Point to table of addresses
        JMP    [A,X] ·
     * Go to routine at X+A
TABLE   FDB    DEATH
        FDB    DEATH
        FDB    NO CHANGE
        FDB    BIRTH
        FDB    DEATH
```

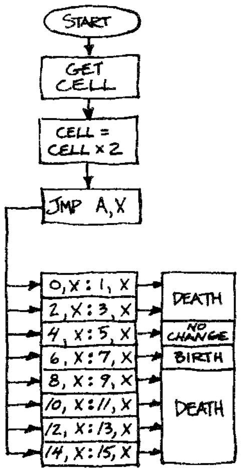
. . . etc. The A register is shifted left to double its value; this is true because an address is two bytes long and therefore requires a two-byte offset. The X register is pointed to the zeroeth entry in the table of jump addresses. The command JMP [A,X] causes the sum of A+X to be calculated, the data at addresses A+X and A+X+1 to be retrieved and concatenated, and the result to be given to the Program Counter. This routine is simple — more transparent than the earlier one — and demonstrates regularity and consistency. Take some time to review this routine and contrast it with the previous one. You'll note that where things might get complicated (for example, if there were ten or twenty choices instead of merely eight), the former routine gets serpentine and sluggish, whereas the indirect indexed jump is a fast and streamlined 6-byte jewel.

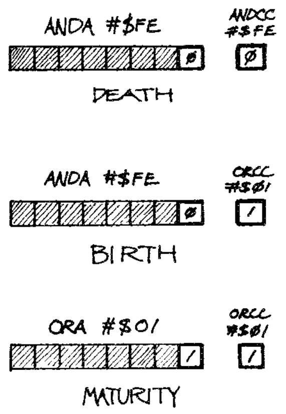
By the way. You see that I've used the notation "FDB" in the short program excerpt above. This is an assembler "pseudo-op", an instruction for the assembler to use the information you've provided and place the equivalent binary data in memory. The pseudo-op FCB places a single byte in the program; FDB places two bytes; and FCC places an ASCII string. Refer to the EDTASM+ manual, page 35, for details on how to use these.

I would like you to take a break here and examine the way this indexed indirect mode is used.









The scratchpad is being evaluated, so all that's left to write is the set of death, birth, and no-change routines. To force a cell into dormancy, both bits are set to zero; the resulting color is buff, the same color as the background. Recalling that the display byte has been rotated through the carry flag, the routine looks like this . . .

```
ANDA    #$FE     * Set to ØX
ANDCC   #$FE     * Set to ØØ
BRA     EXIT     * Go out
```

. . . and you can leave it to a general-purpose exit routine to complete the rotation and testing.

The no-change routine is slightly more complicated because it isn't really no change. As you recall, I wanted to add some visual variety by having newborn cells displayed in a different color from mature cells. Newborns are color **01** (cyan) and matures are 11 (orange), so "no change" for these means changing newborns to matures, and leaving matures as is. On the other hand, dormant cells are left dormant, and illegal cells present in the Garden of Eden are made dormant. Dormants are buff (**00**) and illegals are magenta (**10**), so "no change" for these means changing illegals to dormants, and leaving dormants as is. Here's how it looks:

```
          BCS     HIGH    * Go if C = 1
                          * C = Ø
          ANDA    #$FE    * Set to ØØ
          BRA     EXIT    * Go out
HIGH                      * C = 1
          ORA     #$Ø1    * Set to 11
          BRA     EXIT    * Go out
```

That leaves only the birth routine, which, if a cell is already alive, can be considered a "no change" routine. It is slightly more complex than the previous routines because dormant cells must be changed to newborns (**00** to **01**); illegal cells must be changed to newborns (**10** to **01**); newborns from the previous generation must be changed to oldsters (**01** to **11**); and oldsters are left unchanged (**11** to **11**). Putting it in chart form helps; look in the book:

Birth Routine

Present cell:	Changes to:
ØØ (buff) (dormant)	Ø1 (cyan) (newborn)
1Ø (magenta) (illegal)	Ø1 (cyan) (newborn)
Ø1 (cyan) (newborn)	11 (orange) (mature)
11 (orange) (mature)	11 (orange) (mature)

The carry flag is again the determining factor. If the carry flag is clear (zero), a newborn is created; if the carry flag is set, an oldster is created (or maintained). Here's how that looks . . .

* X points to $1234. A is set to 4. The memory locations $123Ø through $123F contain $Ø1 Ø2 45 47 93 96 A2 Ø1 11 F5 36 92 19 5E 22 ØØ. Where is the program counter after JMP $Ø9,X?

PC is at $123D.

* X points to $1234. A is set to 4. The memory locations $123Ø through $123F contain $Ø1 Ø2 45 47 93 96 A2 Ø1 11 F5 36 92 19 5E 22 ØØ. Where is the program counter after JMP ($Ø9,X)?

PC is at $5E22.

* X points to $1234. A is set to 4. The memory locations $123Ø through $123F contain $Ø1 Ø2 45 47 93 96 A2 Ø1 11 F5 36 92 19 5E 22 ØØ. Where is the program counter after JMP -2,X?

PC is at $1232.

* X points to $1234. A is set to 4. The memory locations $123Ø through $123F contain $Ø1 Ø2 45 47 93 96 A2 Ø1 11 F5 36 92 19 5E 22 ØØ. Where is the program counter after JMP (-2,X)?

PC is at $4547.

* X points to $1234. A is set to 4. The memory locations $123Ø through $123F contain $Ø1 Ø2 45 47 93 96 A2 Ø1 11 F5 36 92 19 5E 22 ØØ. Where is the program counter after JMP ,X?

PC is at $1234.

* X points to $1234. A is set to 4. The memory locations $123Ø through $123F contain $Ø1 Ø2 45 47 93 96 A2 Ø1 11 F5 36 92 19 5E 22 ØØ. Where is the program counter after JMP (,X)?

PC is at $9396.

* What is the term for an instruction for the assembler to use the information you've provided to place binary data in memory?

A pseudo-op.

* What kind of information does the pseudo-op FCB place in memory?

One byte.

* What kind of information does the pseudo-op FDB place in memory?

Two bytes.

* What kind of information does the pseudo-op FCC place in memory?

An ASCII string of characters.

* Hand assemble the following:

```
LDX   $1234
JMP   $AØD7
FCB   $A6
FDB   $ØØF3
```

```
$BE 12 34 7E AØ D7 A6 ØØ F3
```

* What is another name for a work area of memory?

A scratchpad.

* What is the scratchpad used for in this Game of Life?

To store neighborhood information.

* In this Game of Life, what bit pair represents a dormant cell?

Bit pair ØØ.

* In this Game of Life, what bit pair represents a newborn cell?

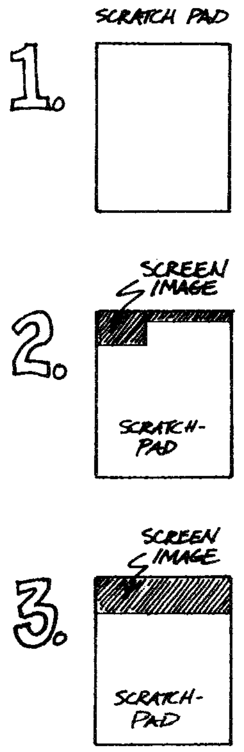
Bit pair Ø1.

* In this Game of Life, what bit pair represents a mature cell?

Bit pair 11.

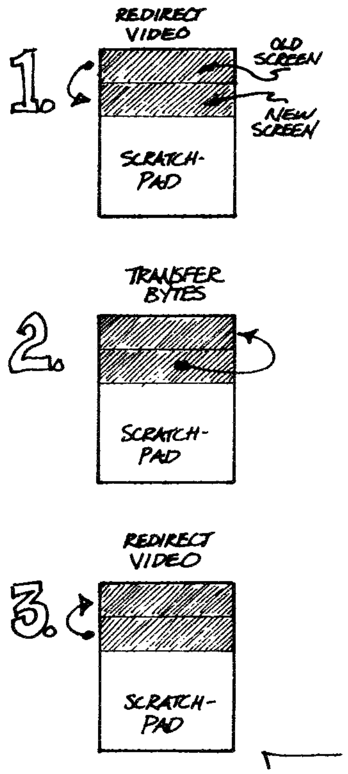
```
         BCC   LOW     * Pass if C = Ø
                       * C = 1
         ORA   #$Ø1    * Set to 11
         BRA   EXIT    * Go out
LOW                    * C = Ø
         ANDA  #$FE    * Set to ØØ
         ORCC  #$Ø1    * Set to Ø1
         BRA   EXIT    * Go out
```

So there you have the heart of it. There's some work to do right at the end. Consider this: if you store the display byte directly back on the screen, the new generation will swim down over the previous generation. Since one of the premises of the Game of Life is that all generational changes take place simultaneously, this swimming effect should be avoided. It can be avoided by filling a second area of memory and switching screens. But with 3,072 bytes required for display, 12,288 bytes required for the scratchpad, and about 230 bytes for program and stack, that leaves less than 700 bytes for a second screen. So what to do?



My solution lies in using that scratchpad for two purposes. Think of it this way. Each four-cell display byte is represented by four bytes of scratchpad memory. Once four scratchpad bytes have been used to determine the new display byte, they are no longer needed. After eight scratchpad bytes are evaluated, two display bytes have been produced. After all 12,288 scratchpad bytes have been used, 3,072 display bytes have been produced.

In that pattern lies the opportunity. The new display screen can be placed in scratchpad memory, because the using up of the scratchpad memory always outpaces by a ratio of four to one the production of display memory bytes. When the new screen has been produced, the video offset address in the SAM can be switched to that new screen in scratchpad memory.



Now since scratchpad memory has to be used again for the next generation, that screen has to be ushered out of that area of memory. Once the video has been redirected from **$0000** to **$1000**, the contents beginning at **$1000** can be transferred to memory beginning at **$0000**. Then, when the original display memory is filled with the new generation, the video offset address can be switched back. The evaluation of the generation, production of the new generation, screen switching, and display memory transfers are entirely invisible. Here's how the code looks . .

```
       STA   $FFCD    * Switch to screen at $1ØØØ
       LDX   #$1ØØØ   * Point X to new screen
       LDY   #$ØØØØ   * Point Y to old screen
TRNSFR LDA   ,X+      * Get value from new screen
       STA   ,Y+      * Store value to former screen
       CMPX   #$1CØØ  * See if screen is finished
       BNE    TRNSFR  * Go back to finish screen
       STA    $FFCC   * Redirect video to $ØØØØ
```

And there you have it. All that's really left to do is to put all the pieces together, keep track of where and how the stack is used, and organize the automatic repeating process to keep the generations going. The entire commented program listing follows on this tape, and is also printed in the book. You can load and examine this listing, and then run the object code which is also on the tape. After you're done reviewing the listing and trying the program, I'll summarize the programming concepts and ideas from these past three sessions.

* In this Game of Life, what is bit pair 10?

An "illegal" or "deviant" cell.

* Why is bit pair 10 an illegal or deviant cell?

Because it is present only in the Garden of Eden, but is not found in future generations.

* Why are bit pairs used?

Two provide a four-color display.

* What video mode is this?

Video mode CG3.

Program #27A, 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 it an I/0 error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix.

```
                   00100 ******************************************************
                   00110 *                                                    *
                   00120 *                 THE GAME OF LIFE                   *
                   00130 *                                                    *
                   00140 *   BASED ON THE PASTIME DEVELOPED BY JOHN CONWAY    *
                   00150 *                                                    *
                   00160 * COLOR COMPUTER VERSION 1.0 BY DENNIS BATHORY KITSZ *
                   00170 *                                                    *
                   00180 ******************************************************
                   00190 *
0C80               00200         ORG     $0C80
                   00210 *
                   00220 * DISABLE INTERRUPTS BY MASKING I AND F BITS IN CCR  *
                   00230 *
0C80 1A    50      00240 START   ORCC    #$50     * DISABLE INTERRUPTS
                   00250 *
                   00260 * COLOR SET 0 CHOICES ARE GREEN, YELLOW, BLUE, RED
                   00270 * COLOR SET 1 CHOICES ARE BUFF, CYAN, MAGENTA, ORANGE
                   00280 * PORT ADDRESS $FF22 CONTROLS COLOR SETS, OTHER INFO
                   00290 *
0C82 86    CF      00300         LDA     #$CF     * VALUE FOR COLOR SET
0C84 B7    FF22    00310         STA     $FF22    * CHOOSE COLOR SET
                   00320 *
                   00330 * THE DISPLAY SCREEN MEMORY IS SELECTED TO RUN FROM
                   00340 * $0000 TO $0BF0 (3072 BYTES), USING COLOR GRAPHICS
                   00350 * MODE 3.  THE FOLLOWING ROUTINE RESETS EVEN ADDRESSES
                   00360 * $FFC6 THROUGH $FFD2 IN THE SAM, SELECTING THE VIDEO.
                   00370 *
0C87 C6    07      00380         LDB     #$07     * VIDEO DISPLAY ADDRESSES
0C89 8E    FFC6    00390         LDX     #$FFC6   * FIRST SAM VIDEO ADDRESS
0C8C A7    81      00400 VIDEO   STA     ,X++     * DO EVERY OTHER ADDRESS
0C8E 5A            00410         DECB             * DONE WITH SETUP YET?
0C8F 26    FB      00420         BNE     VIDEO    * DO NEXT DISPLAY ADDRESS
                   00430 *
                   00440 * THERE ARE THREE GRAPHICS MODES TO BE SELECTED TO
                   00450 * ACHIEVE COLOR GRAPHICS MODE 3.  SAM ADDRESSES $FFC0
                   00460 * THROUGH $FFC5 SET UP COLOR GRAPHICS MODES.
                   00470 *
0C91 B7    FFC5    00480         STA     $FFC5    * SET GRAPHIC MODE 2
0C94 B7    FFC2    00490         STA     $FFC2    * RESET GRAPHIC MODE 1
0C97 B7    FFC0    00500         STA     $FFC0    * RESET GRAPHIC MODE 0
                   00510 *
                   00520 * THE FOLLOWING ROUTINE CLEARS A 12K AREA OF MEMORY
                   00530 * FOR USE AS A SCRATCHPAD WORK AREA WHEN EVALUATING
                   00540 * THE PRESENT GENERATION OF CELLS.  THE METHOD CHOSEN
                   00550 * HERE TO CLEAR MEMORY IS VERY FAST.  INSTEAD OF A
                   00560 * "STA ,X+" STYLE OF MEMORY FILLING, SIX BYTES (THE
                   00570 * TWO ACCUMULATORS PLUS THE X AND Y REGISTERS) ARE
                   00580 * CLEARED TO ZERO.  THE STACK IS POINTED TO THE TOP
                   00590 * OF THE MEMORY TO BE CLEARED, AND THE SIX BYTES ARE
                   00600 * PUSHED ON THE STACK UNTIL THE MEMORY AREA IS FULL.
                   00610 *
```

```
0C9A 4F                AGAIN   CLRA              * SET ACCUMULATOR A = 0
0C9B 5F                        CLRB              * SET ACCUMULATOR B = 0
0C9C 1F   02                   TFR     D,Y       * SET REGISTER Y = 0
0C9E 1F   01                   TFR     D,X       * SET REGISTER X = 0
0CA0 10CE 4000                 LDS     #$4000    * STACK TO TOP OF SCRATCHPAD
0CA4 34   36           NEXT1   PSHS    A,B,X,Y   * PUSH 6 BYTES ON THE STACK
0CA6 118C 1000                 CMPS    #$1000    * IS STACK UNDER #$1000 YET?
0CAA 2A   F8                   BPL     NEXT1     * KEEP GOING IF NOT THERE
                       *
                       * THE STACK IS THEN SET OUT OF THE WAY OF THE
                       * "NEIGHBORHOOD" INCREMENTING ROUTINE WHICH FOLLOWS.
                       *
0CAC 10CE 0DBF                 LDS     #$0DBF
                       *
                       * THE "NEIGHBORHOOD" INCREMENTING ROUTINE USES THE
                       * HOOPER METHOD.  IN MOST CONCEPTUALIZATIONS OF THE
                       * GAME OF LIFE, EACH CELL POSITION IS CHECKED FOR
                       * THE NUMBER OF NEIGHBORS WHICH SURROUND IT.  IT TURNS
                       * OUT THAT, AFTER THE FIRST GENERATION (THE GARDEN OF
                       * EDEN GENERATION), THERE ARE ALWAYS LESS LIVE CELLS
                       * THAN DORMANT ONES.  SO INSTEAD OF CHECKING FOR THE
                       * NEIGHBORS OF EACH CELL, IT IS FASTER TO CHECK EACH
                       * CELL TO DETERMINE WHOSE NEIGHBOR IT IS.  WHEN ALL
                       * CELLS HAVE BEEN CHECKED, A COUNT OF NEIGHBORS HAS
                       * BEEN CREATED.
                       *
0CB0 8E   0000                 LDX     #$0000    * POINT X TO DISPLAY
0CB3 108E 1000                 LDY     #$1000    * POINT Y TO SCRATCHPAD
                       *
0CB7 C6   04           NXTCEL  LDB     #$04      * COUNT FOUR QUARTER BYTES
0CB9 A6   80                   LDA     ,X+       * GET VIDEO DISPLAY BYTE
0CBB 49                QUARTR  ROLA              * ROTATE A THROUGH CARRY
0CBC 49                        ROLA              * ROTATE A THROUGH CARRY
0CBD 24   19                   BCC     NEXTQ     * IF C=0, THEN CELL NOT LIVE
                       *
                       * HERE IS THE NEIGHBORHOOD:
                       *       --------------------
                       *       / -81 / -80 / -7F /
                       *       --------------------
                       *       / -01 / JOE / +01 /
                       *       --------------------
                       *       / +7F / +80 / +81 /
                       *       --------------------
                       *
0CBF 6C   A9 FF7F              INC     -$81,Y    * UPPER LEFT NEIGHBOR
0CC3 6C   A8 80                INC     -$80,Y    * UPPER NEIGHBOR
0CC6 6C   A8 81                INC     -$7F,Y    * UPPER RIGHT NEIGHBOR
0CC9 6C   3F                   INC     -$01,Y    * LEFT NEIGHBOR
0CCB 6C   21                   INC     $01,Y     * RIGHT NEIGHBOR
0CCD 6C   A8 7F                INC     $7F,Y     * LOWER LEFT NEIGHBOR
0CD0 6C   A9 0080              INC     $80,Y     * LOWER NEIGHBOR
0CD4 6C   A9 0081              INC     $81,Y     * LOWER RIGHT NEIGHBOR
                       *
0CD8 31   21           NEXTQ   LEAY    1,Y       * GET NEXT SCRATCHPAD POSITION
0CDA 5A                        DECB              * COUNT DOWN BY QUARTER BYTES
0CDB 26   DE                   BNE     QUARTR    * GET NEXT QUARTER BYTE
0CDD 8C   0C00                 CMPX    #$0C00    * SEE IF END OF DISPLAY
0CE0 26   D5                   BNE     NXTCEL    * ELSE GET NEXT VALUE
                       *
                       * ONCE THE NEIGHBORHOODS HAVE BEEN DETERMINED, THE
                       * INFORMATION IS USED TO GIVE BIRTH TO A CELL, ALLOW
                       * A CELL TO DIE, OR LEAVE THE CELL UNCHANGED.  A TEST
                       * FOR 0 OR 1; 2; 3; 4 OR MORE NEIGHBORS COULD BE DONE
                       * BY USING THE SCRATCHPAD INFORMATION.  IN THIS CASE,
                       * THE INFORMATION (0 THROUGH 8 NEIGHBORS) IN THE
                       * SCRATCHPAD IS USED AS AN OFFSET TO A TABLE OF
                       * ADDRESSES.  THE X REGISTER POINTS TO THE ZEROETH
                       * ENTRY IN THE TABLE, AND THE A REGISTER PROVIDES THE
                       * OFFSET.  X+A IS THE ADDRESS OF THE DEATH, BIRTH AND
                       * NO CHANGE ROUTINES.
                       *
0CE2 8E   0000                 LDX     #$0000    * POINT X TO VIDEO DISPLAY
0CE5 108E 1000                 LDY     #$1000    * POINT Y TO SCRATCHPAD
                       *
0CE9 C6   04           CIRCLE  LDB     #$04      * COUNT FOUR QUARTER BYTES
0CEB A6   84                   LDA     ,X        * GET VIDEO DISPLAY BYTE
0CED 34   10                   PSHS    X         * STASH X REGISTER FOR LATER
0CEF 34   03                   PSHS    A,CC      * STASH VIDEO, CARRY INFO
0CF1 35   03           HERE    PULS    A,CC      * RESTORE VIDEO, CARRY INFO
0CF3 49                        ROLA              * ROTATE THROUGH CARRY FLAG
0CF4 49                        ROLA              * ROTATE THROUGH CARRY FLAG
0CF5 34   03                   PSHS    A,CC      * RE-SAVE ROTATING A, CARRY
0CF7 A6   A0                   LDA     ,Y+       * GET VALUE FROM SCRATCHPAD
```

```
0CF9 48                            ASLA               * DOUBLE IT (2-BYTE OFFSET)
0CFA 8E      0CFF                  LDX     #ZAP       * GET START OF TABLE
0CFD 6E      96                    JMP     ↑A,X←      * ADD OFFSET & JUMP TO ROUTINE
                             *
0CFF         0D11            ZAP   FDB     DEATH      * ROUTINE NEIGHBORHOOD = 0
0D01         0D11                  FDB     DEATH      * ROUTINE NEIGHBORHOOD = 1
0D03         0D19                  FDB     NOCHNG     * ROUTINE NEIGHBORHOOD = 2
0D05         0D25                  FDB     BIRTH      * ROUTINE NEIGHBORHOOD = 3
0D07         0D11                  FDB     DEATH      * ROUTINE NEIGHBORHOOD = 4
0D09         0D11                  FDB     DEATH      * ROUTINE NEIGHBORHOOD = 5
0D0B         0D11                  FDB     DEATH      * ROUTINE NEIGHBORHOOD = 6
0D0D         0D11                  FDB     DEATH      * ROUTINE NEIGHBORHOOD = 7
0D0F         0D11                  FDB     DEATH      * ROUTINE NEIGHBORHOOD = 8
                             *
                             * THE DEATH ROUTINE MUST CREATE COLOR VALUE 00 ON THE
                             * COLOR GRAPHICS DISPLAY SCREEN.  HALF OF THIS VALUE
                             * IS PRESENTLY IN THE A ACCUMULATOR, AND THE OTHER
                             * HALF IS IN THE CARRY FLAG.  BOTH ARE SET TO ZERO
                             * IN THIS ROUTINE.
                             *
0D11 35      03              DEATH  PULS   A,CC       * SAVE DISPLAY, ROTATING BIT
0D13 84      FE                     ANDA   #$FE       * MASK OUT BIT ZERO
0D15 1C      FE                     ANDCC  #$FE       * MASK OUT CARRY BIT
0D17 20      18                     BRA    OUT        * GO OUT TO ROTATE & DISPLAY
                             *
                             * THE NO-CHANGE ROUTINE IS NOT PRECISELY THAT IN THIS
                             * CASE.  COLORS IN THIS GRAPHICS MODE ARE BUFF, CYAN,
                             * MAGENTA AND ORANGE, AS REPRESENTED BY PATTERNS 00,
                             * 01, 10 AND 11.  IN THIS PROGRAM, DORMANT CELLS ARE
                             * SHOWN IN BUFF (00), NEWBORN CELLS IN CYAN (01), AND
                             * CELLS OLDER THAN ONE GENERATION AS ORANGE (11).  THE
                             * VALUE 10 (MAGENTA) IS DEFINED AS ILLEGAL.  HOWEVER,
                             * SHOULD IT OCCUR IN THE "GARDEN OF EDEN", IT MUST BE
                             * CHANGED TO 00.  THIS ROUTINE MAKES THE CHANGE.
                             *
0D19 35      03              NOCHNG PULS   A,CC       * RESTORE ROTATED DISPLAY INFO
0D1B 25      04                     BCS    HIGH       * IF SET, MEANS BIT ZERO = 1
0D1D 84      FE                     ANDA   #$FE       * MAKE 10 OR 00 BECOME 00
0D1F 20      10                     BRA    OUT        * AND GO OUT, STORE & DISPLAY
0D21 8A      01              HIGH   ORA    #$01       * MAKE 01 OR 11 BECOME 11
0D23 20      0C                     BRA    OUT        * AND GO OUT, STORE & DISPLAY
                             *
                             * THE BIRTH ROUTINE IS ALSO A "NO CHANGE" ROUTINE IF
                             * THE NUMBER OF NEIGHBORS IS PRECISELY THREE AND A
                             * LIVE CELL ALREADY EXISTS.  VALUES FOR NEWBORNS ARE
                             * GIVEN AS 01 (CYAN), BUT ALREADY EXISTING CELLS MUST
                             * BE CHANGED TO OLDER CELLS IN ORANGE (11).  ALSO,
                             * ANY ILLEGALS (10, MAGENTA) MUST BE CHANGED.
                             *
0D25 35      03              BIRTH  PULS   A,CC       * GET ROTATED DISPLAY VALUE
0D27 24      04                     BCC    LOW        * GO IF CARRY = 0 (00 OR 10)
0D29 8A      01                     ORA    #$01       * IF C=1, MAKE 11 = OLDSTER
0D2B 20      04                     BRA    OUT        * GO OUT, STORE AND DISPLAY
0D2D 84      FE              LOW    ANDA   #$FE       * C = 0; MAKE 00 OR 01 BE 00
0D2F 1A      01                     ORCC   #$01       * THEN MAKE VALUE BECOME 01
                             *
                             * THE "OUT" ROUTINE IS AN ORDERLY EXIT, TESTING FOR
                             * THE ROTATED POSITION OF A (THE QUARTER-BYTE COUNT),
                             * DOING THE FINAL (NINTH) ROTATE TO GET THE BYTE
                             * BACK IN POSITION IF NECESSARY, STORING THE FINAL
                             * BYTE IN A TEMPORARY SCREEN, AND BRANCHING BACK IF
                             * THE ENTIRE 3,072 BYTE BLOCK (12,288 CELLS) HAS NOT
                             * BEEN DONE.
                             *
0D31 34      03              OUT    PSHS   A,CC       * STASH ROTATING BIT, VIDEO
0D33 5A                             DECB              * TEST FOR NEXT QUARTER BYTE
0D34 26      BB                     BNE    HERE       * IF NOT DONE, GET NEXT QUARTER
0D36 35      03                     PULS   A,CC       * RESTORE ROTATING BIT, VIDEO
0D38 35      10                     PULS   X          * RECOVER STASHED X REGISTER
0D3A 49                             ROLA              * ROTATE TO RESTORE POSITION
0D3B A7      89 1000                STA    $1000,X    * AND STORE BACK INTO DISPLAY
0D3F 30      01                     LEAX   1,X        * GET NEXT POSITION IN PLACE
0D41 8C      0C00                   CMPX   #$0C00     * SEE IF END OF DISPLAY YET
0D44 26      A3                     BNE    CIRCLE     * IF NOT, BACK FOR NEXT BYTE
                             *
                             * THE FOLLOWING ROUTINE REDIRECTS THE SCREEN TO $1000,
                             * WHERE THE NEW GENERATION HAS BEEN CREATED.  IT THEN
                             * COPIES THAT INFORMATION INTO THE SCREEN STARTING AT
                             * $0000, AND SWITCHES SCREENS.  THIS WORK-AND-SWITCH
                             * PROCESS PREVENTS THE NEW GENERATION FROM SWIMMING
                             * DOWNWARD OVER THE PREVIOUS GENERATION AS YOU WATCH.
                             *
0D46 B7      FFCD                   STA    $FFCD      * SWITCH TO SCREEN AT $1000
```

```
                   02280 *
0D49 8E   1000     02290          LDX    #$1000  * POINT X TO NEW SCREEN
0D4C 108E 0000     02300          LDY    #$0000  * POINT Y TO OLD SCREEN
0D50 A6   80       02310 XFER     LDA    ,X+     * GET VALUE FROM NEW SCREEN
0D52 A7   A0       02320          STA    ,Y+     * TRANSFER VALUE TO OLD SCREEN
0D54 8C   1C00     02330          CMPX   #$1C00  * SEE IF SCREEN IS FINISHED
0D57 26   F7       02340          BNE    XFER    * GO BACK TO FINISH SCREEN
                   02350 *
0D59 B7   FFCC     02360          STA    $FFCC   * REDIRECT VIDEO TO $0000
                   02370 *
0D5C 7E   0C9A     02380          JMP    AGAIN   * AND REPEAT THE WHOLE PROCESS
                   02390 *
          0C80     02400          END    START
00000 TOTAL ERRORS
AGAIN   0C9A
BIRTH   0D25
CIRCLE  0CE9
DEATH   0D11
HERE    0CF1
HIGH    0D21
LOW     0D2D
NEXT1   0CA4
NEXTQ   0CD8
NOCHNG  0D19
NXTCEL  0CB7
OUT     0D31
QUARTR  0CBB
START   0C80
VIDEO   0C8C
XFER    0D50
ZAP     0CFF
```

* In this video display, if the A accumulator contains 11110100, what cells are present?

Mature, mature, newborn and dormant.

* If the A accumulator contains 10110000, what cells are present?

Illegal, mature, dormant and dormant.

* What generation is this? Why?

The Garden of Eden, because illegal cells cannot occur in subsequent generations to the Garden of Eden.

* A contains 10110000 in the Garden of Eden. If the algorithm says all cells remain unchanged -- in terms of this Game of Life -- what will the A accumulator contain in the next generation? Why?

A will contain 01110000 because illegals are changed to newborns after the Garden of Eden.

Program #27B, an object code program. Turn on the power to 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, type EXEC and press ENTER. The program will execute automatically. If an I/0 error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix.

Watching the Game of Life is a fascinating experience. A lot has been written about this pastime, and versions in three dimensions and many colors have been developed.

My intention with these three lessons was not only to introduce the concept of indirect indexed addressing, but also to demonstrate with an apparently complicated example the idea of compartmentalized or modular programming. The modules were designed for speed, but with little modification they could be used as complete subroutines . . . those that select color sets, video display memory, and graphics modes and the one to fill memory are complete. The Life routines consist of the evaluation block and the more complex regeneration section. I've drawn arrows in the program listing to show the clear program flow. I'll also tell you that this program wasn't an off-the-cuff creation; it was in fact revised nearly 20 times before it was ready for you to see. Not that it didn't work until 20 tries, but rather that I used more instructions than I needed to do some of the work. In looking for economies of speed, I was able to rethink the details of each routine. You'll do that too as you attempt larger-scale programs.

Here's a summary of the concepts that you have seen:

1. You should be able to establish video modes by referring to the SAM and VDG setup charts and the Color Computer memory map.

2. You should be able to set A, B, X and Y registers and then use the push stack instructions for fast memory filling or clearing.

3. You shold be able to set the stack pointer to a specific place in memory using the LDS instruction.

4. You should understand how to use the rotate instruction to rotate part of a byte into the carry flag, and then employ that information for program branching.

5. You should be able to use a fixed pointer plus a variable offset to select an address from a table of addresses, and then access the information at the resulting address. In other words, you should understand indirect indexed addressing.

6. You should be able to directly manipulate the condition codes (in this example, the carry flags and interrupt masks) using ANDCC and ORCC instructions.

7. You should understand the whys and hows of switching video display modes to hide manipulation of memory.

8. You should have read about pseudo-ops in the EDTASM+ manual, and be able to use ORG, EQU, END, FCB, FCC, FDB and RMB. In summary, these are:

ORG defines the first byte of the program.

EQU identifies the value of a label.

END concludes the assembly process.

FCB identifies a specific byte to be placed in memory.

FDB identifies a spedific two-byte word to be placed in memory.

FCC identifies an ASCII string to be placed in memory.

RMB tells the assembler to reserve — that is, not to use — a block of memory.

I hope all these concepts are clear to you. If you don't feel completely confident, please review. Review the written text for specific items, and review all three lessons if you don't think you could create a complete body of assembly

* If A is 01110000, what are the values of A in binary and hexadecimal, and the value of the carry flag, when ROLA is executed four times? (Assume the carry flag is zero to start).

```
Start A = 01110000, 70; C=0
      A = 11000000, C0; C=1
      A = 00000011, 03; C=1
      A = 00001110, 0E; C=0
      A = 00111000, 38; C=0
```

* The scratchpad memory in this Game of Life is used for two purposes; what are they?

1. To store the neighborhood count during evaluation.
2. To build a new screen containing the next generation.

* What is an algorithm?

A general term for a set of rules.

* What is a pixel?

A picture element.

* What does VDG mean?

Video Display Generator.

* How many pixels does the Color Computer's VDG provide?

256 horizontal by 192 vertical pixels.

* How many different points are displayed on the screen in the most detailed graphics mode (mode RG6)?

49,152 points.

* How many bytes are required for the most detailed graphics mode (mode RG6)?

6,144 bytes.

Summary

* Why can 6,144 bytes display 49,152 points?

Because one byte represents eight display points.

* What addressing mode is JMP $3456?

Extended addressing.

* What addressing mode is JMP A,X?

Indexed addressing.

* What addressing mode is JMP (A,X)?

Indexed indirect addressing.

* What does the instruction LDS #$1000 do?

It loads the hardware stack pointer (S) with the value $1000.

* What does the instruction ORCC #$50 do?

It turns off the interrupts.

* What kind of instruction is FDB $A0D7?

It is an assembler pseudo-op.

* What does FDB $A0D7 do?

It tells the assembler to place the two-byte word $A0D7 in memory.

* What is an addressing mode?

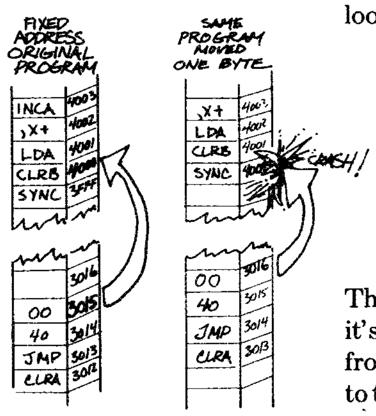
The way in which a machine language instruction gets its information.

code to solve a similar programming problem. These three lessons have offered approach, conceptualization, decision-making, and programming technique. These three lessons — in fact, the past *five* lessons — are the gateway to the rest of this course. I urge you to understand them well. Till next time.

18.

Have you ever typed in a long assembly language program listing from a magazine, accepting on faith that it would work on your Color Computer? And then finding out that your XYZ disk system or your Apex memory dewormer was already using that area of memory? Within certain limitations, that inflexible approach to memory use isn't necessary any more. Utility programs — especially those in semi-permanent installations such as the XYZ disk or Apex dewormer — should be able to be moved to other areas of memory and still perform their advertised functions. Until the introduction of the 6809, microprocessors couldn't offer this as a standard feature... a feature known as Position Independent Programming. Your Color Computer can do it. Position Independent Programming is the topic of this session.

To understand position independence, you have to understand the limitations of position dependence. Have a look at the program in the book; the mnemonics read:



```
1000  8E    1234          LDX  #$1234
1003  108E  5678          LDY  #$5678
1007  B6    FF20   LOOP   LDA  $FF20
100A  27    03      ↑     BEQ  LATER ─┐
100C  7E    1007    └──── JMP  LOOP   │
100F  7F    0001   LATER  CLR  $0001 ◄┘
```

There's nothing especially useful about this program, but it's good enough code. The A accumulator is being loaded from what looks like an input port address, and branching to the label LATER if the loaded value is zero. If it's not, the program jumps back to the position marked LOOP.

But what if you needed to move this program from address **$1000** to, for example, address **$2000**? Well, if you were the programmer, you would simply load the source code into EDTASM+ and re-assemble it at the new origin. But if you had purchased the program and you didn't know its structure or contents, but nevertheless needed to move the binary code from **$1000** to **$2000**, something unhappy

I sigh at the prospect of having to disassemble, examine and relocate some assembly language applications programs — spreadsheets are one example — faced with their enormous size and complexity. This usually happens when I want to tiptoe around some special printer or video driver I've created. With 6809 programs I've had the chance to be pleasantly surprised, since some not only can be located easily in other areas of memory, they automatically relocate themselves to respect memory limits and other configurations you've set ahead of time. Machine language programs which run independent of their position in memory is the exciting goal of this session.

* What is an addressing mode?

The way a machine language program gets its information.

* What addressing mode is JMP $1234?

Extended addressing.

* What addressing mode is BRA LOOP?

Relative addressing.

* Relative addressing is relative to what?

The program counter (PC).

* How does BRA $FE differ from JMP $3456 if both instructions begin at address $3456?

They differ in that BRA is 2 bytes and relative addressing, whereas JMP is 3 bytes and extended addressing.

* How is BRA $FE similar to JMP $3456 if both instructions begin at address $3456?

Both are endless loops.

* Is BRA $FE an endless loop if it appears at address $3455?

Yes.

* Is JMP $3456 an endless loop if it appears at address $3455?

No.

* What happens to JMP $3456 if it is moved to address $3455?

The desired opcode JMP ($7E) is now at $3455. The program counter points to address $3456 where it finds $34 56 instead of $7E. $34 56 isn't an instruction, but the processor thinks it is, executing $34 56 -- PSHS U,X,A,B. Crash!

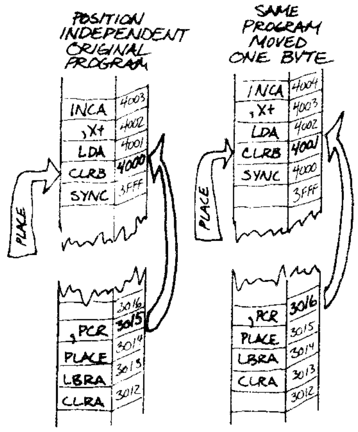
* What do mnemonics BEQ and BNE mean?

Branch if equal to and branch if not equal to.

* What do mnemonics BCC and BCS mean?

Branch on carry clear and branch on carry set.

would occur. Everything in the program would seem perfect until it reached that jump to label LOOP. As far as the binary code is concerned, that jump is specifically to address **$1007**. **$1007** is an absolute, fixed address; with the program now residing at **$2000**, trouble would be on the way. By contrast, the program branch to label LATER is relative addressing . . . the branch is measured from the current position of the program counter. Do you see that? JMP goes to a known, numbered, fixed memory location; BEQ moves to a new position relative to wherever the program is now.



Now, I did use JMP in this example when I could easily have used branch always, BRA. But what if the jump were to an address 5,000 addresses away? An ordinary branch can't move that far, since it's limited to relative movement between +127 and –128. And what about subroutines? The opcode JSR also requires a fixed address. And then there's always the problem of loading X and Y registers with the locations of important tables of information found within the limits of the program. How can these memory locations be identified if not by their fixed locations? Those are the frustrating questions of position independence: how to avoid specifying a fixed, numerical address anywhere in the program.

Well, you can probably guess that I wouldn't be asking those rhetorical questions if I didn't already have an answer. And you're right. The 6809 commands JMP and JSR can be cashed in for the 6809's flexible Branch and Long Branch commands. Not only can you execute long branches to any relative position throughout all of memory, but you can perform long branches to subroutines in any relative position throughout memory. And those load immediate instructions can be cashed in for what's known as "program counter relative" indexing.

The price you pay for these relative branches or indexings is an additional clock cycle or two, plus a slightly different process of thinking. Everything can become relative to the program counter, not just short and long branches, but even loads and stores. Loads and stores can make use of the special ",PCR" version of the indexed addressing mode.

Before I get carried away with the excitement of generalities, I want you to do a little reading. Open your MC6809E data book, turn to page 17, and read the section headed "Program Counter Relative." Also read page 18, the heading "LEAX/LÈAY/LEAU/LEAS." Finally, turn to page 32 and read the summary of the 6809's short and long branch instructions.

Turn to the MC6809E data book, page 17, and read the section headed "Program Counter Relative." Also turn to page 18, and read the section headed "LEAX/LEAY/LEAU/LEAS." Finally, turn to page 32 and read the summary of the 6809's short and long branch instructions. Return to the tape when you have completed the reading.

LDX #$1000
LEAX 1,X
$\boxed{\text{CALCULATE } 1+X}$
X BECOMES $1001
LDA ,X
LOADS A
with contents of
$1001

LDY #$4000
LEAY $9AA,Y
$\boxed{\text{CALCULATE } 9AA+Y}$
Y BECOMES $49AA
LDB ,Y
LOADS B
with contents of
$49AA

Let me start with the LEA instructions, which are easier to use than to describe; you can be looking at page 18 as I talk. LEA (Load Effective Address) is really no mystery, it's just a highly jargonized name for an old, familiar concept. Here's how LEA came clear to me: There exist no unique increment or decrement instructions for the 16-bit X or Y registers in the 6809. Considering how often I wanted to move these registers forward or back in memory, I thought this might be a serious deficiency in the 6809's capability. Sure, you know that there are automatic increment and decrement modes, but these require loading or storing information to get them to work. So I spent some time cracking my brains over LEAX and LEAY.

I discovered that Increment X is actually **LEAX 1,X**... that is, make X become X with an offset of 1. Decrement X, then, must be **LEAX -1,X**. It seemed clumsy then, but not now. Maybe these are a little less easy to think of or use than a straightforward increment or decrement, but they are many times more flexible. If **LEAX 1,X** makes X become X+1, then **LEAX 2,X** makes X become X+2. You're no longer limited to simple increments or decrements. **LEAX -40,X** makes X equal X–40. **LEAY 12345,Y** makes Y equal Y+12345. That was the key. I began to understand that the clumsy phrase "load effective address" was a jargon-filled way of saying the same thing that "LET" says in BASIC. Whereas BASIC would say LET Y = Y+150, the 6809 assembly language says **LEAY 150,Y** . . . load Y with the effective address 150+Y.

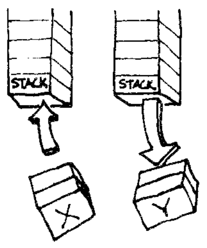
"INCREMENT X"
THINK
LEAX 1,X

"DECREMENT X"
THINK
LEAX -1,X

But there's more. Not only can X=X+10 by writing **LEAX 10,X**, but X can equal Y+10 by writing **LEAX 10,Y**. . . or Y can equal S–50 by writing **LEAY -50,S** . . . or U can equal X by writing **LEAU 0,X**. In fact, depending on your requirements, the 6809 processor offers three different ways of making one 16-bit register equal another: you've got **TFR X,Y**. Then there's **PSHS X** followed by **PULS Y**. And then you can **LEAX 0,Y**.

TFR X,Y

PSHS X
PULS Y



Here's more about Load Effective Address. You can use the A, B or combination D accumulators as variable offsets. For example, X can be made equal to A plus X by writing **LEAX A,X**.

But by far the most versatile and powerful application of the LEA instructions is in the writing of position independent programs. In the programs I've presented so far, I've always loaded the X or Y registers with specific values. For example, in the Life program that was

* What is the branching range of BRA (and other branch instructions)?

PC-128 to PC+127 (PC-$80 to PC+$7F).

* What does LBRA mean?

Long branch always.

* What is the branching range of LBRA (and other long branch instructions)?

* PC-32768 to PC+32767 (PC-$8000 to PC+$7FFF).

* What addressing mode is BEQ LOOP?

Relative addressing.

* What addressing mode is LBEQ LOOP?

Relative addressing.

* What does LEA mean?

LEA means Load Effective Address.

* What is the effect of LEAX 1,X?

X becomes X+1.

* What is the effect of LEAX $45,X?

X becomes X+$45.

* What is the effect of LEAX 1,Y?

X becomes Y+1.

* What is the effect of LEAX -5,Y?

X becomes Y-5.

* What is the effect of LEAY 12345,Y?

Y becomes Y+12345 (Y+$3039).

* If A is \$32 and X is \$1000, what is the effect of LEAX A,X?

X becomes X+A, that is, X becomes \$1032.

* If X = \$1000, give the value of Y after:
TFR X,Y

Y becomes \$1000.

* If X = \$1000, give the value of Y after:
PSHS X
PULS Y

Y becomes \$1000.

* If X = \$1000, give the value of Y after:
LEAY 0,X

Y becomes \$1000.

* If X = \$1010, give the value of Y after:
LEAY -16,Y

Y becomes \$1000.

* What does LEA mean?

LEA means Load Effective Address.

* What does ",PCR" mean?

",PCR" means program counter relative mode.

* If the instruction LDX #ARITH1 is found at address \$1000, and label ARITH1 points to \$2000, what is X after the instruction is executed?

X points to \$2000.

* If the instruction LDX ARITH1,PCR is found at address \$1000, and label ARITH1 points to \$2000, what is X after the instruction is executed?

X points to \$2000.

completed in the last session, you remember that the X register was pointed to a table of information by loading the X register with the actual address of the table. I wrote **LDX #TABLE**. But there's another way, a position independent way.

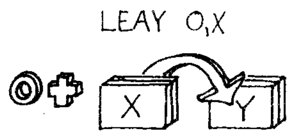
I might instead have written **LEAX TABLE,PCR**. That's **LEAX TABLE,PCR**. And that says "Load X with the effective address calculated from the distance between the present position of the program counter and the address of the table." In other words, I know the distance from here to where I'm going. By giving that distance to the 6809, it can calculate the resulting address, and give that result to the X register.

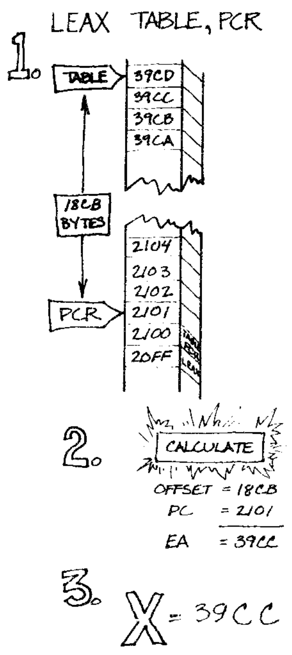
No longer are you constrained to a fixed address. Instead of demanding to know, "where is it?", the 6809 need only ask "how far is it from here?". I'll get back to Load Effective Address; in the meantime, just remember that when you see LEAX, think LET X. You see **LEAY 10,Y** and you think LET Y be 10 plus Y. Purists might want my head for that, but I'll risk it. When you see LEA, think LET.

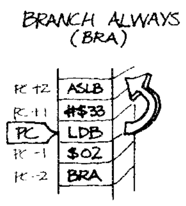
Among the other position-independent commands are the branches. You've been using the branches since early on in this course, but I've never given them any formal time. I'll make up for that now.

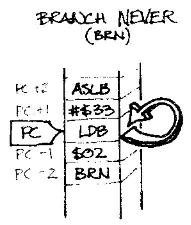
Like the program counter relative instructions, the branches are also based on "how far from here?" rather than "where?". In all, there are 62 variations of relative branches, depending on how you think of them. Turn to page 32 of the MC6809E data book. You'll see the branch instructions in four groups: simple, simple conditional, signed conditional, and unsigned conditional. Some overlap, serving dual purposes. I'm going to describe the short branches, but keep in mind that the long branches are identical in principle and application. The only difference is that the short branches reach a span of 256 bytes, and the long branches reach a span of 65,536 bytes.

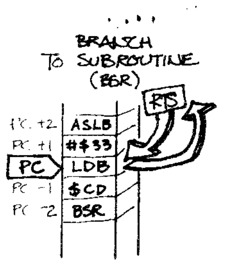
Simple branches are just that. When the instruction decoder finds a simple, it follows the command, calculates the new address, and hands it to the program counter. These three are branch always (BRA), branch never (BRN) and branch to subroutine (BSR). Two of these make sense; but what about "branch never"? "Branch never" is one of those delightful bizarrities of computer logic. "Branch never" exists as a default of the processor's architecture. All branches have what are called true and false versions; branch always is the true version, so "branch never" is the false version. Branch always makes the branch, very much like the command JMP. "Branch never" continues with the main program flow. But keep it in mind; it's surprisingly useful. Should you be doing critical timing where every machine byte and clock cycle counts, remember that no operation (NOP) uses one byte and 2 cycles; "branch never" has the effect of a NOP, but it uses two bytes and 3 cycles; and long "branch never" also has the effect of a NOP, but it uses 4 bytes and 5 cycles.



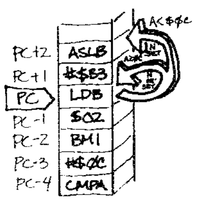




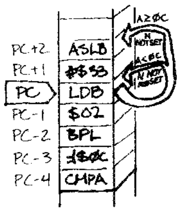




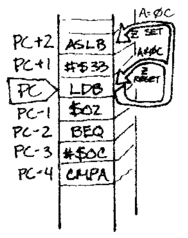
BRANCH ON MINUS (BMI)



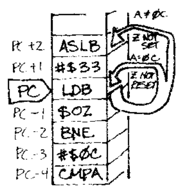
BRANCH ON PLUS (BPL)

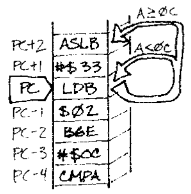


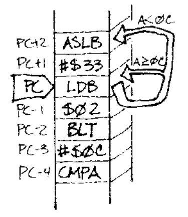
BRANCH ON EQUAL (BEQ)



BRANCH ON NOT EQUAL (BNE)







Enough of the simple branches; on to the simple conditional branches. These are changes of program flow conceived of as direct responses to the condition codes.

1. Branch on minus and branch on plus are in response to the state of the negative (N) flag.

2. Branch on equal and branch on not equal are in response to the state of the zero (Z) flag.

3. Branch on overflow set and branch on overflow clear respond to the state of the overflow (V) flag.

4. Finally, branch on carry set and branch on carry clear respond to the state of the carry (C) flag.

Those eight conditional branches should make sense to you, since you've used most of them in programming already.

The signed and unsigned conditional branches take account of not only the flags but also the type of arithmetic being used, in order to produce a composite result and make a branching decision. The signed conditional branches assume that you are using signed arithmetic, that is, where you are thinking in terms of positive and negative, so that the most significant bit is important to the calculation. There are three types of signed conditional branch, arranged five ways:

1. Branch on greater than (BGT), and its opposite, branch on less than or equal to (BLE). Remember that in signed arithmetic, **$01** is greater than **$FE**, that is, 1 is greater than −1.

2. The complementary instructions to the previous ones are branch on greater than or equal to (BGE) and branch on less than (BLT).

3. Signed branches also make use of the familiar branch on equal (BEQ) and branch on not equal (BNE).

4 and 5. The final two pairs of branches are identical to the first to pairs, but are conceived in reverse. At the end of this lesson, take the time to examine the four tables at the bottom of page 32 of the data booklet, and try to clarify how the pair "branch on greater than"/ "branch on less than or equal to" is different in conception from "branch on less than or equal to"/"branch on greater than".

The remaining branch types are the unsigned conditional branches. These are effectively identical to the previous

* What does BSR mean?

BSR means Branch to subroutine.

* What do mnemonics BGT, BGE, BLT and BLE mean?

Branch on greater than, branch on greater than or equal to, branch on less than, and branch on less than or equal to.

* What do mnemonics BRA and BRN mean?

Branch always and branch never.

* In unsigned arithmetic, which is the higher number, $7F or $00?

$7F is a higher number than $00.

* In unsigned arithmetic, which is the higher number, $AA or $55?

$AA is a higher number than $55.

* In signed arithmetic, which is the greater number, $AA or $55?

$55 (being positive) is greater than $AA (being negative).

* In signed arithmetic, which is the greater number, $FF or $00?

$00 is greater than $FF (-1).

* What specific kind of instruction is BGT (branch on greater than)?

BGT is a signed conditional branch.

* What specific kind of instruction is BHS (branch on higher than or same as)?

BHS is an unsigned conditional branch.

* If A contains $FF and is compared to memory containing $00, would the branch BGT be taken or not? Why?

It would not be taken because $FF (decimal -1) is less than $00, and BGT is a signed conditional branch.

* If A contains $FF and is compared to memory containing $00, would the branch BHS be taken or not? Why?

The branch would be taken because $FF (decimal 255) is higher than $00, and BHS is an unsigned conditional branch.

* What addressing mode are BHS and BGT?

Relative addressing.

* What addressing mode is JMP $1234?

Extended addressing.

* What addressing mode is JMP ($1234)?

Extended indirect addressing.

* What does the mnemonic LBLO mean?

Long branch if lower than.

* What addressing mode is this?

Relative addressing.

* What is the branching range of BLO?

The range is -128 ($80) to +127 ($7F) relative to the program counter.

* What is the branching range of LBLO?

The range is -32768 ($8000) to +32767 ($7FFF), relative to the program counter.

ones, but negativeness or positiveness do not affect the result. These branches are:

1. Branch on higher than (BHI), and its opposite, branch on lower than or same as (BLS). In unsigned arithmetic, **$FE** is greater than **$01**, that is, 254 is greater than 1.

2. Branch on higher than or same as (BHS), and its opposite, branch on lower than (BLO).

3. The familiar branch on equal (BEQ) and branch on not equal (BNE) are also part of the unsigned set of branches.

4 and 5. Finally, there are the inverse pairs of the first sets of conditions. Again, examine these tables at the end of the lesson.

So how do these all fit together? How do you choose among simple conditional, signed conditional, and unsigned conditional branches? Here's how:

- If you're using the flags directly, such as with rotations, yes/no comparisons, etc., use the simple conditional branches. If you're thinking about the condition codes per se, then you want to use simple conditional.

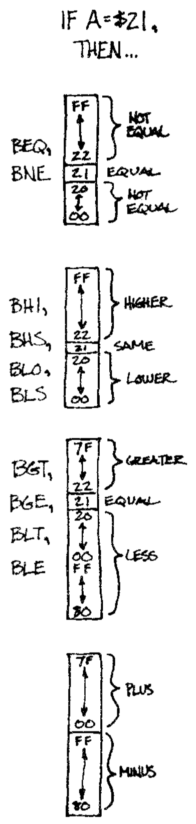
- If you're doing arithmetic, such as creating mathematical subroutines, or if you're using numbers transferred from BASIC, use signed conditional branches. Real numbers are positive and negative, so use signed conditional branches when doing that kind of math.

- If you're making a series of value comparisons or checking table entries, then use the unsigned conditional branches. These are similar to the simple conditional branches, except they allow you a little more flexibility or programming compactness.

Some experimenting will make the choices clear. I've got a program I think you'll like. Get your computer on and up in Extended BASIC. When you're ready, type and enter this BASIC line; follow in your book:

```
PCLEAR8:PMODE4,1:PCLS:PMODE4,5:PCLS:CLOADM:EXEC
```

Your computer will be ready and searching for an object code program. It's coming up.



Program #28, an object code program. Turn on the power to 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, type EXEC and press ENTER. The program will execute automatically. If an I/0 error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix.

BASIC started by clearing an area of graphics memory, so what you should be seeing is a clean high-resolution graphics screen with a narrow, random-looking band of colors walking down the screen from top to bottom. At the same time, a continuous tone is coming from the loudspeaker. The tone hiccups each time the colored band moves down the screen.

Before you sigh "so what" to yourself, let me tell you what you're looking at. The band of random color isn't random at all. It's the program. The program itself is being displayed as if it were screen information. That shouldn't be a surprise, since memory is memory so far as the microprocessor is concerned. But it can be disconcerting to actually snoop into the program's private memory lair.

Now to my point. This band of color is MOVING. The program is producing a tone, then moving itself, erasing its trail, and re-executing in a new position in memory. Eventually, the loudspeaker will let out a strangled squawk and probably return an "OK" to your screen, as the moving program crashes into the un-writable BASIC ROM.

This is a completely position-independent program. When you're ready, you can load the assembly source code and have a look. I'll be back for the next lesson and a complete walk-through of this program, and a re-explanation and summary of the process of position-independent code. Enjoy this one.

Program #29, 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/0 error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix.

```
1000                00100         ORG     $1000
                    00110 *
          FF20      00120 SPORT   EQU     $FF20
          00AA      00130 DIFFER  EQU     LAST-FIRST
                    00140 *
                    00150 * DISABLE THE INTERRUPTS
1000 1A   50        00160 FIRST   ORCC    #$50
                    00170 *
                    00180 * OPEN THE SOUND LATCH
1002 86   3C        00190         LDA     #$3C
1004 B7   FF23      00200         STA     $FF23
                    00210 *
```

* How many groups of branches are there?

There are four groups of branches.

* What are the four kinds of branches?

Simple branches, simple conditional branches, unsigned conditional branches, and signed conditional branches.

* What is a position independent program?

A program designed to run correctly no matter where it is located in memory.

```
                       00220 * SELECT VIDEO ADDRESS
1007 C6    07          00230         LDB     #$07
1009 8E    FFC6        00240         LDX     #$FFC6
100C A7    81          00250 VIDEO   STA     ,X++
100E 5A                00260         DECB
100F 26    FB          00270         BNE     VIDEO
1011 B7    FFCD        00280         STA     $FFCD
                       00290 *
                       00300 * SELECT GRAPHICS MODE
1014 B7    FFC5        00310         STA     $FFC5
1017 B7    FFC3        00320         STA     $FFC3
101A B7    FFC0        00330         STA     $FFC0
                       00340 *
                       00350 * SELECT COLOR SET, MODE
101D 86    C7          00360         LDA     #$C7
101F B7    FF22        00370         STA     $FF22
                       00380 *
                       00390 * ERASE PREVIOUS PROGRAM
1022 C6    AA          00400 ERASE   LDB     #DIFFER
1024 30    8C D9       00410         LEAX    FIRST,PCR
1027 30    89 FF56     00420         LEAX    -DIFFER,X
102B 4F                00430         CLRA
102C A7    80          00440 KLEEN   STA     ,X+
102E 5A                00450         DECB
102F 26    FB          00460         BNE     KLEEN
                       00470 *
                       00480 * BEEP FOR ALL TO HEAR
1031 8D    12          00490         BSR     BEEP
                       00500 *
                       00510 * TRANSFER PROGRAM AHEAD
1033 C6    AA          00520         LDB     #DIFFER
1035 30    8C C8       00530         LEAX    FIRST,PCR
1038 31    8D 006E     00540         LEAY    LAST,PCR
103C A6    80          00550 LOOP    LDA     ,X+
103E A7    A0          00560         STA     ,Y+
1040 5A                00570         DECB
1041 26    F9          00580         BNE     LOOP
                       00590 *
                       00600 * AND GO TO MOVED PROGRAM
1043 20    65          00610         BRA     LAST
                       00620 *
1045 86    FF          00630 BEEP    LDA     #$FF
1047 34    02          00640 REBEEP  PSHS    A
1049 86    3E          00650         LDA     #$3E
104B 30    8D 001C     00660         LEAX    WAVES,PCR
104F E6    86          00670 WAVER   LDB     A,X
1051 58                00680         ASLB
1052 58                00690         ASLB
1053 F7    FF20        00700         STB     SPORT
1056 8D    09          00710         BSR     DELAY
1058 4A                00720         DECA
1059 26    F4          00730         BNE     WAVER
105B 35    02          00740         PULS    A
105D 4A                00750         DECA
105E 26    E7          00760         BNE     REBEEP
1060 39                00770         RTS
                       00780 *
1061 34    02          00790 DELAY   PSHS    A
1063 86    06          00800         LDA     #$06
1065 4A                00810 DLOOP   DECA
1066 26    FD          00820         BNE     DLOOP
1068 35    02          00830         PULS    A
106A 39                00840         RTS
                       00850 *
106B       1F1C        00860 WAVES   FDB     $1F1C
106D       1916        00870         FDB     $1916
106F       1310        00880         FDB     $1310
1071       0D0B        00890         FDB     $0D0B
1073       0806        00900         FDB     $0806
1075       0403        00910         FDB     $0403
1077       0201        00920         FDB     $0201
1079       0000        00930         FDB     $0000
107B       0000        00940         FDB     $0000
107D       0001        00950         FDB     $0001
107F       0204        00960         FDB     $0204
1081       0608        00970         FDB     $0608
1083       0A0C        00980         FDB     $0A0C
1085       0F12        00990         FDB     $0F12
1087       1417        01000         FDB     $1417
1089       1B1E        01010         FDB     $1B1E
108B       2124        01020         FDB     $2124
108D       272A        01030         FDB     $272A
108F       2D30        01040         FDB     $2D30
```

```
1091        3235      01050           FDB      $3235
1093        3739      01060           FDB      $3739
1095        3A3C      01070           FDB      $3A3C
1097        3D3E      01080           FDB      $3D3E
1099        3E3E      01090           FDB      $3E3E
109B        3E3E      01100           FDB      $3E3E
109D        3D3C      01110           FDB      $3D3C
109F        3B39      01120           FDB      $3B39
10A1        3735      01130           FDB      $3735
10A3        3330      01140           FDB      $3330
10A5        2E2B      01150           FDB      $2E2B
10A7        2825      01160           FDB      $2825
10A9        22        01170           FCB      $22
                      01180 *
            10AA      01190 LAST      EQU      *
                      01200 *
            1000      01210           END      FIRST
00000 TOTAL ERRORS
```

19.

Welcome back. During this session I want to review the concept of position independent programming, and to take you through the self-moving, position-independent program from the end of the last lesson. Get that source code loaded again.

Program #29, 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/0 error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix.

```
1000                    00100          ORG      $1000
                        00110 *
              FF20      00120 SPORT    EQU      $FF20
              00AA      00130 DIFFER   EQU      LAST-FIRST
                        00140 *
                        00150 * DISABLE THE INTERRUPTS
1000 1A       50        00160 FIRST    ORCC     #$50
                        00170 *
                        00180 * OPEN THE SOUND LATCH
1002 86       3C        00190          LDA      #$3C
1004 B7       FF23      00200          STA      $FF23
                        00210 *
                        00220 * SELECT VIDEO ADDRESS
1007 C6       07        00230          LDB      #$07
1009 8E       FFC6      00240          LDX      #$FFC6
100C A7       81        00250 VIDEO    STA      ,X++
100E 5A                 00260          DECB
100F 26       FB        00270          BNE      VIDEO
1011 B7       FFCD      00280          STA      $FFCD
                        00290 *
                        00300 * SELECT GRAPHICS MODE
1014 D7       FFC5      00310          STA      $FFC5
1017 B7       FFC3      00320          STA      $FFC3
101A B7       FFC0      00330          STA      $FFC0
                        00340 *
                        00350 * SELECT COLOR SET, MODE
101D 86       C7        00360          LDA      #$C7
101F B7       FF22      00370          STA      $FF22
                        00380 *
                        00390 * ERASE PREVIOUS PROGRAM
1022 C6       AA        00400 ERASE    LDB      #DIFFER
1024 30       8C D9     00410          LEAX     FIRST,PCR
1027 30       89 FF56   00420          LEAX     -DIFFER,X
102B 4F                 00430          CLRA
```

The position-independent program really isn't all just tricks and gimmicks. Its real purpose is to make the machine code "transportable". BASIC is transportable; you don't need to load it to a specific memory location. You just load and run. High-level languages have to work that way, but machine language had a hard time ... until the 6809.

* What is a position independent program?

A program designed to run correctly no matter where it is located in memory.

Program #29 reprise

```
102C A7    80         00440 KLEEN    STA      ,X+
102E 5A               00450          DECB
102F 26    FB         00460          BNE      KLEEN
                      00470 *
                      00480 * BEEP FOR ALL TO HEAR
1031 8D    12         00490          BSR      BEEP
                      00500 *
                      00510 * TRANSFER PROGRAM AHEAD
1033 C6    AA         00520          LDB      #DIFFER
1035 30    8C C8      00530          LEAX     FIRST,PCR
1038 31    8D 006E    00540          LEAY     LAST,PCR
103C A6    80         00550 LOOP     LDA      ,X+
103E A7    A0         00560          STA      ,Y+
1040 5A               00570          DECB
1041 26    F9         00580          BNE      LOOP
                      00590 *
                      00600 * AND GO TO MOVED PROGRAM
1043 20    65         00610          BRA      LAST
                      00620 *
1045 86    FF         00630 BEEP     LDA      #$FF
1047 34    02         00640 REBEEP   PSHS     A
1049 86    3E         00650          LDA      #$3E
104B 30    8D 001C    00660          LEAX     WAVES,PCR
104F E6    86         00670 WAVER    LDB      A,X
1051 58               00680          ASLB
1052 58               00690          ASLB
1053 F7    FF20       00700          STB      SPORT
1056 8D    09         00710          BSR      DELAY
1058 4A               00720          DECA
1059 26    F4         00730          BNE      WAVER
105B 35    02         00740          PULS     A
105D 4A               00750          DECA
105E 26    E7         00760          BNE      REBEEP
1060 39               00770          RTS
                      00780 *
1061 34    02         00790 DELAY    PSHS     A
1063 86    06         00800          LDA      #$06
1065 4A               00810 DLOOP    DECA
1066 26    FD         00820          BNE      DLOOP
1068 35    02         00830          PULS     A
106A 39               00840          RTS
                      00850 *
106B       1F1C       00860 WAVES    FDB      $1F1C
106D       1916       00870          FDB      $1916
106F       1310       00880          FDB      $1310
1071       0D0B       00890          FDB      $0D0B
1073       0806       00900          FDB      $0806
1075       0403       00910          FDB      $0403
1077       0201       00920          FDB      $0201
1079       0000       00930          FDB      $0000
107B       0000       00940          FDB      $0000
107D       0001       00950          FDB      $0001
107F       0204       00960          FDB      $0204
1081       0608       00970          FDB      $0608
1083       0A0C       00980          FDB      $0A0C
1085       0F12       00990          FDB      $0F12
1087       1417       01000          FDB      $1417
1089       1B1E       01010          FDB      $1B1E
108B       2124       01020          FDB      $2124
108D       272A       01030          FDB      $272A
108F       2D30       01040          FDB      $2D30
1091       3235       01050          FDB      $3235
1093       3739       01060          FDB      $3739
1095       3A3C       01070          FDB      $3A3C
1097       3D3E       01080          FDB      $3D3E
1099       3E3E       01090          FDB      $3E3E
109B       3E3E       01100          FDB      $3E3E
109D       3D3C       01110          FDB      $3D3C
109F       3B39       01120          FDB      $3B39
10A1       3735       01130          FDB      $3735
10A3       3330       01140          FDB      $3330
10A5       2E2B       01150          FDB      $2E2B
10A7       2825       01160          FDB      $2825
10A9       22         01170          FCB      $22
                      01180 *
           10AA       01190 LAST     EQU      *
                      01200 *
           1000       01210          END      FIRST
00000 TOTAL ERRORS
```

```
BEEP    1045
DELAY   1061
DIFFER  00AA
DLOOP   1065
ERASE   1022
FIRST   1000
KLEEN   102C
LAST    10AA
LOOP    103C
REBEEP  1047
SPORT   FF20
VIDEO   100C
WAVER   104F
WAVES   106B
```

The opening lines of the source code should look familiar to you. Interrupts are disabled to keep the tone pure; the sound latch is opened (recall that process from the Morse Code routine); the video address **$1000** is selected via the SAM registers; high-resolution color graphics, color set, and detail level are selected through an address port. Up to that point, everything is as it has been.

The real differences begin with the routine labeled ERASE. The value identified as DIFFER has been calculated by the assembler from my labels LAST minus FIRST. The first byte of the program I labeled FIRST, and one byte after the last byte I labeled LAST. At the start of the assembly listing, I have the assembler calculate LAST minus FIRST . . . which is, of course, the length of the entire program. So accumulator B is loaded with the length of the program.

There follow two significant instructions . . .

```
        LEAX    FIRST,PCR
        LEAX    -DIFFER,X
```

$\boxed{\text{SOURCE}}$

FIRST EQU $1000
DIFFER EQU LAST-FIRST
LEAX FIRST, PCR

$\boxed{\text{ASSEMBLY}}$

LOCATE
↳ LEAX -$D9, PCR

$\boxed{\text{EXECUTION}}$

CALCULATE
↳ X = PC + (-$D9)

LEAX - DIFFER, X

$\boxed{\text{ASSEMBLY}}$

LOCATE
↳ LEAX -$AA, X

$\boxed{\text{EXECUTION}}$

CALCULATE
↳ X = X-$AA

$\boxed{\text{RESULT}}$

X = $ØF56

LEAX FIRST,PCR requests that the assembler compute the distance from the program counter to the label FIRST, and make the resultant address available for use by the X register. In other words, after **LEAX FIRST,PCR**, the X register points to the beginning of the program. Then comes the instruction **LEAX -DIFFER,X**. That command instructs the processor to let X equal the present X value minus the value DIFFER. So the effect of those two instructions is to point the X register to a place in memory one program length before the program. Let me go through that one more time. **LEAX FIRST,PCR** is a program-counter relative instruction that calculates the distance between the current position of the program counter and the label FIRST, and assigns the resultant address to register X. Using this technique, X ends up pointing to the start of the program, without ever knowing what absolute address that start actually is until now. After that,
LEAX -DIFFER,X provides the X register with the effective address X offset by -DIFFER. Let X equal X minus DIFFER. X now points to a location in memory DIFFER places back from its previous position, still without ever knowing the absolute address beforehand. Again: **LEAX FIRST,PCR**. Let X point to the address FIRST places from the program counter. **LEAX -DIFFER,X**. Let X point to the address -DIFFER places away from its previous position. No specific addresses involved . . . position independent . . . program-counter relative.

* How many groups of branches are there?

There are four groups of branches.

* What are the four kinds of branches?

Simple branches, simple conditional branches, unsigned conditional branches, and signed conditional branches.

* What is the branching range of the branch instructions?

The range is -128 ($80) to +127 ($7F) relative to the program counter.

* What is the branching range of the long branch instructions?

The range is -32768 ($8000) to +32767 ($7FFF), relative to the program counter.

* What addressing mode are all the branches, both long and short?

Relative addressing.

* Relative addressing is relative to what?

The program counter.

* What does ",PCR" mean?

Program counter relative.

* What does LEA mean?

LEA means Load Effective Address.

* What is the effect of LEAX 1,X?

X becomes X+1.

* What is the effect of LEAX $45,X?

X becomes X+$45.

* What is the effect of LEAX 1,Y?

X becomes Y+1.

* What is the effect of LEAX -5,Y?

X becomes Y-5.

* If A is $32 and X is $1000, what is the effect of LEAX A,X?

X becomes X+A, that is, X becomes $1032.

* What is the effect of LEAX 1,X?

X becomes X+1.

* What is the effect of LEAX -1,X?

X becomes X-1.

* The 6809 processor provides an INCA command. What is the equivalent of INCX, a fictitious command?

LEAX 1,X

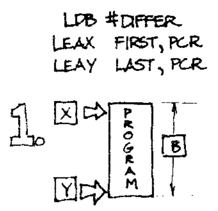
* The 6809 provides a DECA command. What is the equivalent of DECX, a fictitious command?

LEAX -1,X

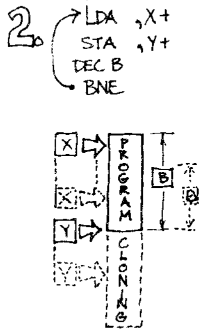
* If the first byte of a program is labeled START, what is the effect of LEAX START,PCR if the program is ORGed at $1000?

X becomes $1000.

The next four instructions fill up the memory area from –DIFFER,X to FIRST with zeroes; the B register contains DIFFER, the total number of bytes in the program. That is, a block of memory as long as the program from –DIFFER,X to FIRST will be cleared to zero.



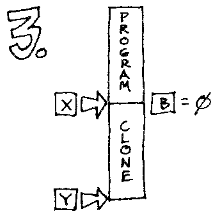
Following those contortions is a relative branch to the subroutine BEEP. I'll get back to BEEP in a minute.



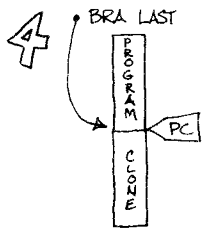
After the branch to and back from BEEP, the B register is once more loaded with the program's length. Following that . . .

```
        LEAX    FIRST,PCR
        LEAY    LAST,PCR
```

Again using the program counter relative technique, the X register is pointed to the beginning of the program, and the Y register is pointed to the byte after the last byte in the program. By means of a standard load-and-store loop — which should be tiresomely familiar by now — the information pointed to by X is transferred to memory pointed to by Y, and both memory pointers are incremented by one. The loop continues until B is decremented to zero. In other words, a copy of the program is made immediately following the end of itself.



The final instruction is the grabber. The program is told to execute a branch to the label LAST. The LAST has become the FIRST. The program, having just been copied, is born again and seemingly begins anew in a fresh area of memory. It once again sets up the video and sound parameters — a redundant act I included for effect. At this point, the reason for the ERASE routine presented earlier should become clear. ERASE causes the previous program to be cleared out of memory — the program hides its own trail as it beeps and copies itself.



EXECUTE THE CLONE !!

So what you see is a screen full of memory, and revealed on that screen you are watching is a program that beeps, duplicates itself in a new location, branches into its new self, and eradicates its old self.

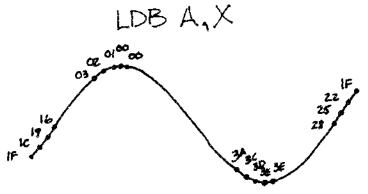
Chances are you wouldn't ever need to write a program like this. But you might want to write something like the BEEP subroutine, a routine that you can stuff anywhere you like in memory. Have a look at it.

Part of its structure should be familiar. The A register is set up as the length of the beep, and there are values being sent out the sound port to the television speaker. But there's something new. **LEAX WAVES,PCR** (again using program-counter relative addressing) points the X register to a table labeled WAVES. So what's this table?

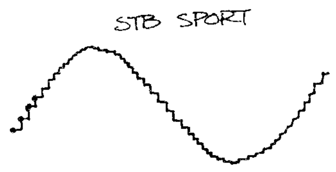
It might look at first like a table of addresses. It isn't. It's a 63-byte reference table . . . these are bytes, not addresses. I just wanted to save myself some typing by compressing them the way you see them. So you can read this table as a group of 63 bytes: **$1F $1C $19 $16 $13 $10 $0D**, etc. Trans'ated back into the form in which I created them, they read like this:

```
.ØØØ3Ø73
.Ø995276
.1983681
.2952265
.3891352
.4791557
.5643887
.6439825
```

. . . and so forth. It's actually a table of mathematical sines, made positive and multiplied by a constant so that the table falls into the range of positive integers 0 to 63. The reason I've done this is because the Color Computer contains a 6-bit digital-to-analog converter, a circuit which converts a 6-bit binary number into an equivalent voltage. That voltage can be used for a variety of purposes, including the production of sound.



I described this briefly when you were exploring the Morse Code examples. This time you'll be putting it to use. Move back now to the BEEP routine itself. Notice that beginning with the third instruction, the BEEP program loads the A accumulator with **$3E**, points the X register to that table, and then loads the value found at X indexed by A into the B accumulator. The value is shifted to the left (from the low 6 bits to the high 6 bits, where the computer's digital-to-analog converter output happens to be wired). That value is then stored at SPORT, the sound output address in the computer. A brief delay is made, then the next element in the table is acquired and output to the sound port, until all 63 elements have been used up. The routine then loops until 255 repetitions of the table have been output.



The sine wave is the simplest of all musical sounds. By creating a series of numerical values which outline a sine-shaped wave and subsequently putting those values through the computer's 6-bit converter, an equivalent sound wave is produced through the loudspeaker. It sounds like the sine wave it represents.

Take a break now, and make some changes in the subroutine. You can assemble and use the BEEP subroutine separately, if you like. If you use it separately, remember to turn off interrupts by using **ORCC #$50**, and also to turn on the sound latch by storing **$3C** at memory address **$FF23**. I'd like you to play around with the length of the beep (found at line 630 being loaded into the A register), with the frequency of the beep (found in the delay loop at line 800), and with the quality of the sound (by changing the values in the wavetable beginning at line 860). When you're comfortable with how these routines work, thoroughly review both this lesson and the previous one. I'll be back with a summary of position independent programming, and then I'll finish up this session by introducing the remaining 6809E instructions.

* If the first byte of a program is labeled START, what is the effect of LEAX START,PCR if the program is ORGed at $1234?

X becomes $1234.

* If the first byte of a program is labeled START, what is the effect of LEAX START,PCR if the program is ORGed at $AAAA?

X becomes $AAAA.

* What addressing mode is LEAX WAVES,PCR?

Program-counter relative.

* What is a pseudo-op?

An instruction to the assembler.

* What pseudo-op places a single byte in memory?

FCB.

* What pseudo-op places two consecutive bytes in memory?

FDB.

* What pseudo-op places an ASCII string of characters in memory?

FCC.

* Does the Color Computer have a digital-to-analog converter?

Yes.

* A digital-to-analog converter converts what to what?

A binary number to an equivalent voltage.

* At what memory location is the Color Computer's digital-to-analog converter found?

At location $FF22.

* How many bits can be sent to the Color Computer's digital-to-analog converter?

6 bits.

* What is the range (in binary, hex and decimal) of the Color Computer's digital-to-analog converter?

Binary 000000 to 111111; hexadecimal $00 to $3F; decimal 0 to 63.

* The Color Computer's digital-to-analog converter ranges from 0 to 5 volts, divided into 64 steps. Zero output is 0/64ths, full output is 64/64ths; that is, it has a step size or resolution of 1/64th of the output. If 000000 is sent to the digital-to-analog converter, what is the output?

000000 is 0/64ths, or 0 volts.

* If 111111 is sent to the digital-to-analog converter, what is the output?

111111 is 63/64ths, or 4.921875 volts.

* If 101010 is sent to the digital-to-analog converter, what is the output?

101010 is 42/64ths, or 3.28125 volts.

* If all the values from 000000 to 111111 and back to 000000 are sent to the digital-to-analog converter, what will a graph of the final voltage output look like?

A triangle.

* If the Color Computer's digital-to-analog converter were 7 bits instead of six, what would be the step size (the resolution)?

1/128th of the output.

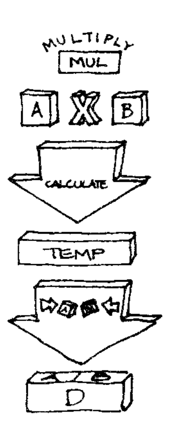
Experiment with the length, pitch and sound quality of the beep in this program. The length of the beep is loaded into the A register in line 630 of Program 29. The frequency of the beep is found in the delay loop in line 800. The wavetable begins at line 860. When you are confident you understand the application of these features, return to the tape.

Position independent programming, then, is the creation of machine language in a way that allows the final assembled binary program to execute anywhere in memory. This quality of position independence is achieved by making all memory pointers, program branches and subroutines relative to the position of the program counter. In that way, the processor never needs to know "where", but only needs to know "how far from here".

Among the commands used with position independent programming are the three dozen variants of the branch (with its 256-byte range) and the long branch (with its 65,536-byte range). Branches come in simple form, where they are always obeyed; in simple conditional form, where their actions depend on the state of specific condition codes; in unsigned conditional form for "higher" and "lower" judgments; and in signed conditonal form for "greater than" and "less than" judgments in with positive and negative arithmetic.

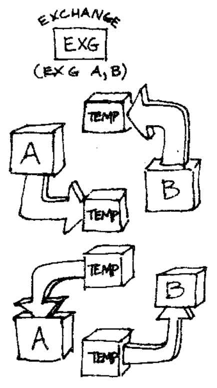
The other commands to achieve position independence are the LEA, or load effective address, group. When used with in program-counter relative form, 16-bit registers can be pointed to any location in memory by virtue of that location's position relative to the current position of the program counter. It's almost mandatory to use an editor/assembler and labels to do this. For the experience, you might try hand-assembling a few LEAX instructions in the program-counter-relative mode.

The advantages of position independence are obvious; the disadvantages are a slight increase in the amount of programming code required, and a loss in execution speed. For fast action games and high speed — where position independence is hardly necessary anyway — compact, address-specific programming is adequate and desirable. For utility programs, mathematical subroutines, and other semi-permanent programs (especially those which will be used with other machine-language software), position independence is virtually required.

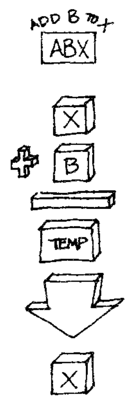


Only a few commands remain in the 6809 instruction set. Some you've come across, and some are brand new to this course. One you've seen is multiply, **MUL**. When **MUL** is executed, the contents of the A accumulator is multiplied by the contents of the B accumulator, and the result is placed in the combined D accumulator. This is an unsigned multiply, meaning the full 8 by 8 bit multiplication is completed without reference to it being positive or negative. Positive integers are assumed for this multiplication. Although **MUL** takes 11 machine cycles (it is the longest 6809 instruction), it saves the several steps required by other processors, where multiplication is done by many succeeding steps of shifting and adding.

Another you've already seen is no operation, mnemonic **NOP**. The **NOP** has several uses, most frequently as a time-waster for sound, input/output, communication, or other timing loops. The **NOP** takes two cycles to execute, during which no other aspect of the procesors's operation is affected.

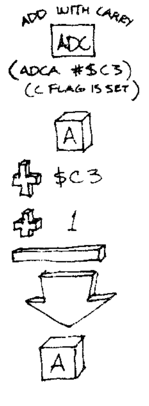


Another instruction which you haven't specifically used, but is in a familiar family, is exchange, EXG. Like the transfer (TFR) command, EXG uses an opcode and a postbyte to describe the registers needed. TFR replicates the value in the source register into the destination register. EXG swaps the values in the two registers. EXG is useful for organizing A and B registers properly in the 16-bit D register; for placing information into the more flexible X register; for temporarily swapping stacks; and so forth.



Since I just mentioned the X register as being more flexible, I'll present the command **ABX**. **ABX** instructs the processor to add the value of the B register to the X register. This inherent instruction is very fast, and acts as a kind of fixed increment for X. If X has to move through a high resolution graphics screen hex **$80** bytes at a time, for example, it would be most efficient to set B to **$80** and execute **ABX**. Especially inside a loop, **ABX** would bump the X pointer down to the next graphics screen line in a short time.

Two complementary instructions are add with carry (ADC) and subtract with borrow (SBC). These are standard add and subtract commands, except that the carry/borrow flag is made a part of the computation. I'll talk more about ADC and SBC when I get to the representation of numbers in a later lesson.



TST and BIT are related quick testing instructions. BIT causes the processor to AND the value of an accumulator with a memory location. Certain flags are affected, but the original contents of both accumulator and memory remain unchanged. BIT is particularly useful for locating numbers or ASCII strings in memory, since the value in the accumulator isn't affected as it moves and tests byte after byte.

TST is similar to BIT, but is oriented toward signed numbers. TST tests the value of the operand — which can be a memory location or either accumulator — and sets the negative and zero flags according to what it finds. Signed conditional branches (BGT, BLE, BGE, BLT, BEQ and BNE) are usually placed after the TST.

* If the Color Computer's digital-to-analog converter were 8 bits instead of six, what would be the step size (the resolution)?

1/256th of the output.

* What is the step size (the resolution) of the Color Computer's digital-to-analog converter?

1/64th of the output.

* What is the highest resolution of this table of sine values for the Color Computer's digital-to-analog converter?

1/64th of the sine wave shape.

* The following questions refer to the remaining 6809 instructions introduced in Lesson 19.

* What is the action of MUL?

The contents of the A accumulator is multiplied by the contents of the B accumulator, and the result is placed in the D accumulator.

* Is the result of MUL signed or unsigned?

Unsigned.

* If A contains $08 and B contains $C2, what is the result of MUL?

D contains $0610.

* If A contains $55 and B contains $AA, what is the result of MUL?

D contains $3872.

* If A contains $FF and B contains $FF, what is the result of MUL?

D contains $FE01.

* What is the result after NOP?

No change to any registers or memory locations; no operation takes place.

* If A contains $08 and B contains $C2, what is the result of EXG A,B?

A contains $C2 and B contains $08.

* If X contains $FFEE and Y contains $01CD, what is the result of EXG X,Y?

X contains $01CD and Y contains $FFEE.

* If X contains $01CD and B contains $33, what is the result of ABX?

X contains $0200.

* If X contains $FFFF and B contains $08, what is the result of ABX?

X contains $0007.

* If A contains $10 and the carry flag is set, what is the result of ADCA #$10?

$10+$10+C = $21

* If B contains $01, what is the result of SEX?

D contains $0001.

* If B contains $FF, what is the result of SEX?

D contains $FFFF.

* If B contains $80, what is the result of SEX?

D contains $FF80.

* A contains $43 and ADDA $99 is executed. What is the result after DAA?

A contains $42 and the carry flag is set.

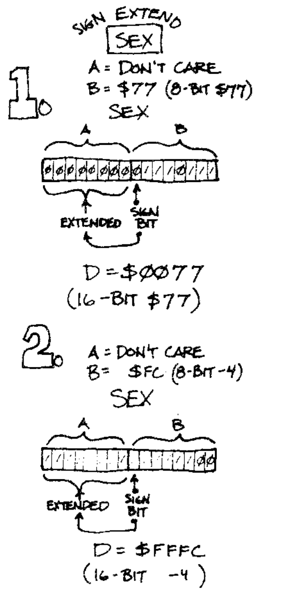
The next instruction also has to do with signed arithmetic. Called sign extend (**SEX**), it results in the sign of the B accumulator being extended into the A accumulator for a complete, signed 16-bit number in the D register. In other words, if B is a positive number, A will become **$00**. If B is **$77**, for example, after **SEX**, the D register will be **$0077**. On the other hand, if B is a negative number, A will become **$FF**. That is, if B is **$FC** (–4 decimal in 8-bit signed arithmetic), a negative number, its sign is extended so that the resulting D register is **$FFFC** — still –4 decimal in 16-bit arithmetic. If that isn't clear, count backwards, first in 8 bits and then in 16 bits. Starting with **$00**, **$FF** is –1, **$FE** is –2, **$FD** is –3, **$FC** is –4. Now start with **$0000**, a 16-bit number. **$FFFF** is –1, **$FFFE** is –2, **$FFFD** is –3, **$FFFC is –4. Sign extend, mnemonic SEX**, sees to it that an 8-bit signed value is properly transformed into a 16-bit signed value.

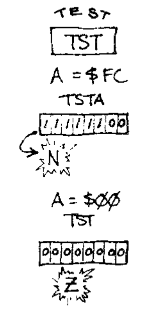
All that's left is **DAA**, the decimal addition adjustment. Microprocessors are working in binary, base 2, and that operation is represented by hexidecimal, base 16. As you've discovered, none of this fits very well with base 10, the decimal system. Some processors contain a decimal mode of operation, where adjustments are made automatically after every computation to compensate for the base 10 system. In other words, no number larger than binary **1001** is allowed in a nybble.

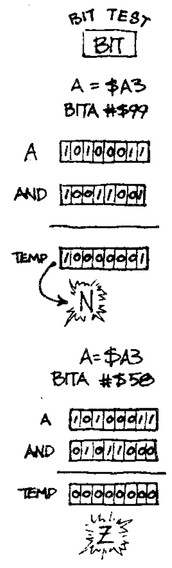
Sadly, decimal mode is is one of the few desirable features not found in the 6809 processor. In its place is the instruction decimal addition adjust, or **DAA**. When executed after and ADD or ADC, the values in the accumulator are converted from true binary mode to a decimal version called binary-coded-decimal, or BCD. The nybbles of the byte are adjusted, and the carry flag set if necessary, to turn the binary result into BCD.

For example, if I were to **LDA #$77** and then **ADDA #$77** (note both these are binary-coded-decimal numbers), the binary result would be hex **#$EE**. Although I want these to be decimal representations, the processor treats them as if they were binary. If I follow those commands with **DAA**, however, a series of tests and corrections are made. **$54** is left in the accumulator and the carry flag is set. That's the number 154 in BCD, the sum of 77 BCD plus 77 BCD. Review the summary of **DAA** on page 43 of your EDTASM+ manual; there will be more on this later.

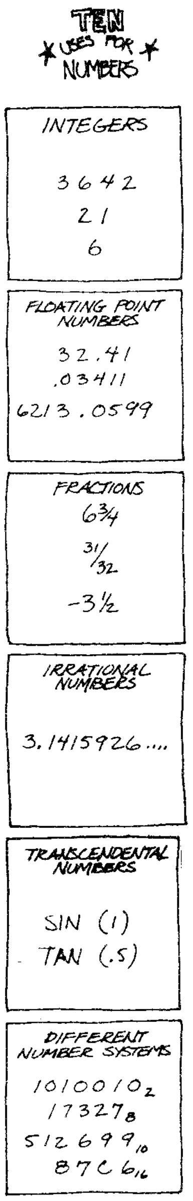
By the way, it's especially with an operation such as **DAA** that the command ADC comes into play. The carry generated by **DAA** in the previous example has to be taken into consideration when doing arithmetic with larger numbers. Keep that in mind, as I'll be covering that in Representation of Numbers, the next lesson.







20.



What is a number? I've been wanting to ask you that question at just about every session, but I think now's the time for it. What is a number?

No matter what comes to mind in response to the question, it's probably right, and that means the computer has to deal with it. Somehow, the binary data has got to be arranged to handle all those conceptions. Numbers might include . . .

- integers both positive and negative.
- floating-point decimal numbers
- fractions
- irrational numbers and transcendental functions
- different number systems
- identification or code numbers
- scales or scientific ranges
- money
- very large or very small numbers

Some of these — like floating-point numbers and money — are just slight conceptual variations. Others — like transcendental numbers and different number base systems — are strikingly dissimilar.

Learning the details of handling all these different numbers in assembly language would require a separate course, so I'm going to limit the discussion to simple numbers. Once you've got this session down, you'll be ready for all the rest. You already understand positive and negative integers, so I mean to go one step further — to floating point numbers and how they are represented in binary notation.

Signed and unsigned integers have been the limit for the calculation and conversion examples so far. Numbers are mind-bogglingly more than that, and binary format has to be forced to handle them all. Floating-point notation — that is, representation of decimal numbers — is far and away the most obscure topic in assembly language. Even to the experienced, it comes only with irritation.

* What is an integer?

A whole number with no fractional part.

* How does the 6809 represent signed integers?

By using the most significant bit (the leftmost bit) of a number.

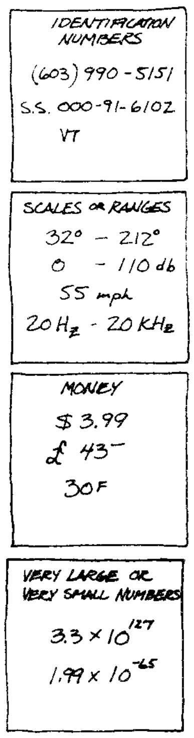
* What is the sign for positive and negative?

A zero in the most significant bit is positive; a one in the most significant bit is negative.

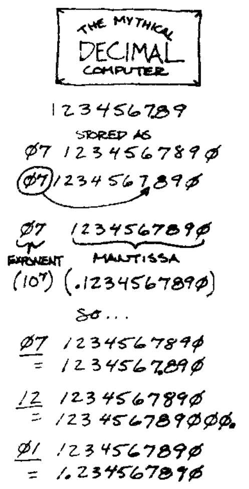
Accuracy and range

"Floating point" is jargon for numbers in complete form — positive or negative numbers, with integer and fractional portions. All numbers in the Color Computer's BASIC are stored as floating point numbers, whether they look like integers or not. The number 10, for example, is actually thought of as 10.0000000 with the computer's internal hexadecimal representation **$84 20 00 00 00**. One million is thought of as 1000000.00, with the internal hexadecimal representation of **$94 74 24 00 00**. 0.1 becomes 0.100000000 and is represented by hexadecimal **$7D 4C CC CC CD**, and one-millionth is 0.0000010 and is stored as hexadecimal **$6D 06 37 BD 06**.

Don't expect these hexadecimal patterns to make any sense as I read them to you. They are, in fact, five-byte groupings capable of representing any number from –170,141,173,000,000,000,000,000,000,000,000,000,000 (negative 170 trillion, 141 billion, 173 million billion billion billion) to +170,141,173,000,000,000,000,000,000,000,000,000,000. The Color BASIC language can handle these with nine significant digits of accuracy — that is, only the first nine digits are used for the actual computations. This is excellent accuracy (far better than my old 4-digit slide rule), but not always enough for the modern age of high technology, with its measurements of astronomical vastness or molecular smallness. By understanding how floating point numbers are represented, it is possible to extend the accuracy of numbers to as many digits as you need. No matter how fast the machine's speed, handling such large numbers will take time; but handling large and small numbers will be possible — even via BASIC.



Now to what those numbers mean. The principle is, once again, disarmingly simple. Let me start the explanation as if you were using a decimal computer instead of a binary one. Take the decimal number 1234567.89. Now say this decimal computer you own has a precision of 10 significant digits. The number is really 1234567.890 for your computer. And of course this decimal computer doesn't have a decimal point inside the number — it can only store information on where the decimal point is. It won't actually put one there except for display.



So the number is 1234567.89, meaning the decimal point is between the seventh and eight positions. So by storing 7 followed by 1234567890, you can say that the number stored in your special decimal computer is 1234567 point 89, with the trailing zero dropped. Simply by changing your descriptive information you can change the number's power of 10. By storing 12 followed by 1234567890, you automatically know that the number you want is 123,456,789,000. By storing 1 plus 1234567890, the number becomes 1.23456789.

There's no difference in the Color Computer's representation of numbers from the description of this imaginary decimal computer. The five bytes used to describe a floating point number on the Color Computer are in binary. That's the key. To represent one million as

* Show $8FC2 in binary; is it positive or negative? Why?

$8FC2 is 1000 1111 1100 0010, and it is negative because the most significant bit is a one.

* What is a floating-point number?

A number with a fractional part.

* Can 326 be a floating-point number? Why?

Yes, because the fractional part is zero (.0000000....)

* What is the accuracy of the Color Computer?

9 significant digits.

* What is a significant digit?

The part of the actual number used in storage or computation.

* What are the signficant digits of 123,456,789,876,543,210 on the Color Computer?

The most significant digits are 123456789.

* How would 123,456,789,876,-543,210 be displayed on the Color Computer?

It would be displayed 1.2345679E+17.

* What does the E mean in 1.2345679E+17?

E means exponent, that is, the power of 10 by which the number is multiplied; in other words, 1.2345679 times 10 to the 17th.

* What is 10 to the 17th (10↑17)?

100,000,000,000,000,000

$94 74 24 00 00 is to store it with one descriptive byte telling where the point is, plus a string of binary digits. Here's a case where hexadecimal is pretty useless. Binary is the only solution to seeing it.

The exponent byte comes first, which is a power of 2 — essentially a description of where the decimal point goes. In this case, I'm going to coin a term . . . I think this should be called a binaral point, since this is binary notation. **$80** is the central value around which the binaral point swings. From **$81** to **$FF** represents from 1 to 127 places to the left of the binaral point — numbers greater than one; from **$01** to **$7F** represents from 127 to 1 places to the right of the point.

Back to the number one million, stored as **$94 74 24 00 00**. **$80** is the pivot point, so **$94** minus **$80** is **$14**. That means that this number has hex **$14** — decimal 20 — digits to the left of the binaral point. I'll write the remainder of this in binary:

```
0111 0100 0010 0100 0000 0000 0000 0000
```

The leftmost bit of these 32 digits is used as the sign bit; as usual, 0 is the positive sign and 1 is the negative sign. In numerical terms (exactly why is difficult to explain but will become clear with experience), this bit is assumed to be a 1 for calculation purposes. That is, since any number's got to have some digit to multiply by other than zero, at least one 1 will appear . . . and that's the case no matter whether the number is positive or negative. So whether the sign bit is 1 or 0, this bit is included in the calculation as if it were a 1. Turning back to the string of binary digits, it becomes (please follow along in the book now):

```
1111 0100 0010 0100 0000 0000 0000 0000
```

Since the point is after the 20th position (hex **$14**), count over from the left. The left, for one of the rare times in computer terms, is called the **first** rather than the zeroeth position. Putting the point in place makes the number read:

```
11110100001001000000.000000000000
```

Now you do one of two things. The first option is to sum the powers of two to calculate the result, starting from just left of the point. Zero times 2↑0 plus 0 times 2↑1 plus 0 times 2↑2, keeping the sum as you move on up to 1 times 2↑19. That's actually the sum of 2↑19 + 2↑18 + 2↑17 + 2↑16 + 2↑14 + 2↑9 + 2↑6, which is 1,000,000.

Or as an alternative you can break the binary into four-bit groups, again starting from the immediate left of the point, and convert those to hexadecimal: it becomes **$F4240**. According to my hexadecimal calculator, **$F4240** is, indeed, one million in decimal.

I'm going to take you through a few of these for practice. Let me hand you just any five-byte group that comes to mind as I put this lesson together. I'll keep it positive and large until

* What is 2 to the zeroeth (2↑0)?

1 (any number to the zero power is 1).

* What is 2 to the 1st (2↑1)?

2

* What is 2 to the 2nd (2↑2)?

4

* What is 2 to the 3rd (2↑3)?

8

* What is 2 to the 16th (2↑16)?

65536

* What is 2↑0 plus 2↑1 plus 2↑2 plus 2↑3?

1+2+4+8, or 15.

* What is 2↑0 plus 2↑1 plus 2↑2 ... up to 2↑15?

1+2+4 ... +32768, or 65535.

* In floating-point binary notation, what is the first byte?

The exponent byte.

* If the exponent byte is $99, what does it indicate?

It indicates 2 to the power $19 (hex) or 2↑25 (decimal).

* If the exponent byte is $81, what does it indicate?

It indicates 2 to the power $01, or 2.

* What are the four bytes following the exponent byte called?

The mantissa.

* If the four bytes are $5A 00 00 00, what is the binary mantissa?

The binary mantissa is 0101 1010 0000 0000 0000 0000 0000 0000.

* Which bit is the sign bit?

The leftmost bit is the sign bit.

* If the mantissa is $5A 00 00 00, what is its sign?

The sign of $5A 00 00 00 is positive (the leftmost bit is a zero).

* What is a "normalized" mantissa?

A mantissa in which the leftmost bit has been set to a one after its sign is known.

* If the (normalized) mantissa is 1101 1010 0000 0000 0000 0000 0000 0000 and the exponent byte is $88, place the point.

1101 1010. 0000 0000 0000 0000 0000 0000

* What is this number?

2↑1 + 2↑3 + 2↑4 + 2↑6 + 2↑7 = 218

* If the (normalized) mantissa is 1101 1010 0000 0000 0000 0000 0000 0000 and the exponent byte is $91, place the point.

1101 1010 0000 0000 0.000 0000 0000 0000

* What is this number?

2↑10 + 2↑12 + 2↑13 + 2↑15 + 2↑16 = 111,616

you get the hang of it. Let's say the bytes you see are **$9F 66 7D 80 1F**. Write those bytes down. **$9F 66 7D 80 1F**. The leftmost byte is the power-of-two exponent, you recall, revolving around the **$80** pivot point. **$9F** minus **$80** is **$1F**, so you know that this number is **$1F** (that is, decimal 31) digits long. The digits themselves are **66 7D 80 1F**, which, when translated into binary, become:

```
ØllØ ØllØ Ølll llØl lØØØ ØØØØ ØØØl llll
```

You can follow along in the book or write those down. The leftmost bit is the sign bit; it's zero, so this is a positive number. Now you can replace the zero with the "normalized" one for calculation purposes. Here's the number:

```
lllØ ØllØ Ølll llØl lØØØ ØØØØ ØØØl llll
```

The binaral point is after the **$1F**th digit . . . that's the 31st digit. So the binary number now is:

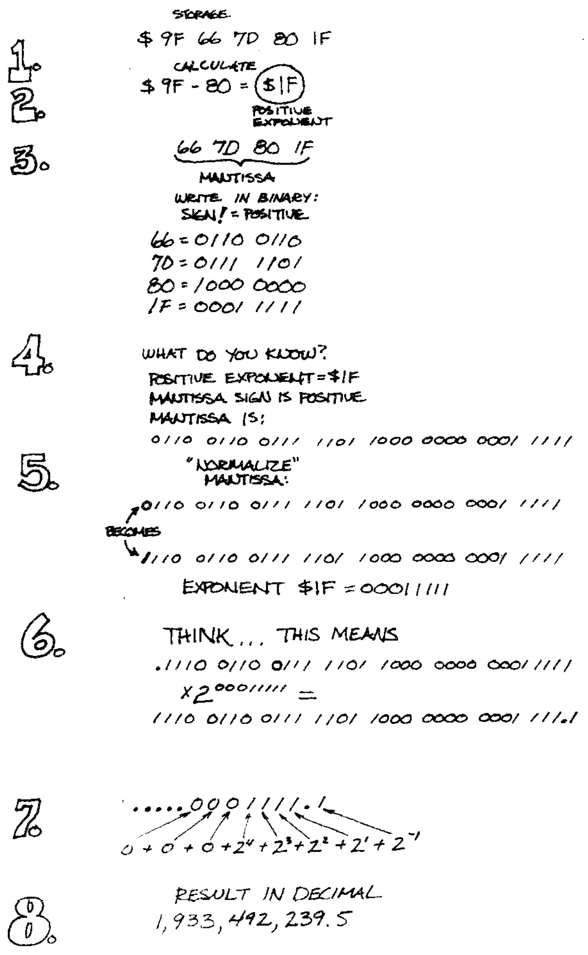
```
lllØØllØØlllllØllØØØØØØØØØØllll.l
```

I'll do the sum with the powers of two method. 2↑0 + 2↑1 + 2↑2 + 2↑3 + 2↑14 + 2↑15 + 2↑17 + 2↑18 + 2↑19 + 2↑20 + 2↑21 + 2↑24 + 2↑25 + 2↑28 + 2↑29 + 2↑30 works out to 1,933,492,239. Remember, you start from the immediate left of the point and sum up the powers of two. The result once again is 1,933,492,239.

But what about that .1 at the end of the binary string? What is that and how do you use it?

In decimal, the numbers to the right of the decimal point represent negative power of 10, or, if you like 1/10ths, 1/100ths, 1/1000ths, 1/10000ths, etc. In binary, the numbers to the right of the point represent — you guessed it — negative powers of two. one-halves, one-quarters, one-eighths, 1/16ths, 1/32nds, 1/64ths, etc. So that ".1" at the end represents 1/2, or in decimal, 0.5. The resulting number should therefore be 1,933,492,239.5 on the Color Computer.

That's both right and wrong. A few minutes ago I said that the Color Computer BASIC's accuracy is only nine digits. That's a choice made mostly for reasons of speed and consistency. If you write X = 1933492239.5 and enter it on the computer, your PRINT X will reveal 1.9334922 4E+09. That's BASIC's scientific notation for 1.93349224 times 10 the 9th. In other words, not 1,933,492,239.5, but rather 1,933,492,240. Only nine significant digits are used, so part of the number gets rounded off and abbreviated.



Another random example. Stay with me. **$7C 91 32 2F 00**. Write it down to work with. **$7C 91 32 2F 00**. **$7C** in the first position is the exponent byte, but this time it's less than the **$80** pivot. This is a number with values all to the right of the decimal point. That is, a number less than 1. **$7C** is –4 binary positions, so you know the binary number begins with .0000 .

The hexadecimal for the rest of the number is **$91 32 2F 00**, which is, in binary:

```
1001 0001 0011 0010 0010 1111 0000 0000
```

The one in the leftmost position means this time we've got a negative number. Now that you know that, you'll also remember that it is "normalized" to one for purposes of calculation. So the result is

```
0001 0001 0011 0010 0010 1111 0000 0000.
```

With .0000 in front of it, it becomes:

```
.00001001000100110010001011110000000000
```

Normalized notation

* If the mantissa is 1101 1010 0000 0000 0000 0000 0000 0000 and the exponent byte is $99, place the point.

1101 1010 0000 0000 0000 0000 0.000 0000

* What is this number?

2↑18 + 2↑20 + 2↑21 + 2↑23 + 2↑24 = 28,573,696.

* If the (normalized) mantissa is 1101 1010 0000 0000 0000 0000 0000 0000 and the exponent byte is $A0, place the point.

1101 1010 0000 0000 0000 0000 0000 0000.

* What is this number?

2↑25 + 2↑27 + 2↑28 + 2↑30 + 2↑31 = 3,657,433,088

* What are the two parts of floating-point representation called?

The exponent and the mantissa.

* What is 1 divided by 2↑1?

1/(2↑1) is 1/2.

* What is 1 divided by 2↑2?

1/(2↑2) is 1/4.

* What is 1 divided by 2↑16?

1/(2↑16) is 1/65536.

* If the (normalized) mantissa is 1110 0000 0000 0000 0000 0000 0000 0000 and the exponent byte is $82, place the point.

11.10 0000 0000 0000 0000 0000 0000 0000

* What is this number?

2↑1 + 2↑0 + 1/(2↑1) = 3 1/2 = 3.5

* If the normalized mantissa is 1111 1111 1111 1111 1111 1111 1111 1111 and the exponent byte is $90, place the point.

1111 1111 1111 1111. 1111 1111 1111 1111

* What is this number?

2↑0 + 2↑1 + 2↑2 + 2↑3 + 2↑4 + 2↑5 + 2↑6 + 2↑7 + 2↑8 + 2↑9 + 2↑10 + 2↑11 + 2↑12 + 2↑13 + 2↑14 + 2↑15 = 65,535. 1/(2↑1) + 1/(2↑2) + 1/(2↑3) + 1/(2↑4) + 1/(2↑5) + 1/(2↑6) + 1/(2↑7) + 1/(2↑8) + 1/(2↑9) + 1/(2↑10) + 1/(2↑11) + 1/(2↑12) + 1/(2↑13) + 1/(2↑14) + 1/(2↑16) = .99998474121094. Therefore, the number is 65535.99998474121094.

* What would this number be in the Color Computer's most significant digits?

65535.99998 is rounded off (up) and becomes 65536.

* What are the two parts of floating-point representation called?

The exponent and the mantissa.

* What information does BASIC's VARPTR provide?

The address of a BASIC variable.

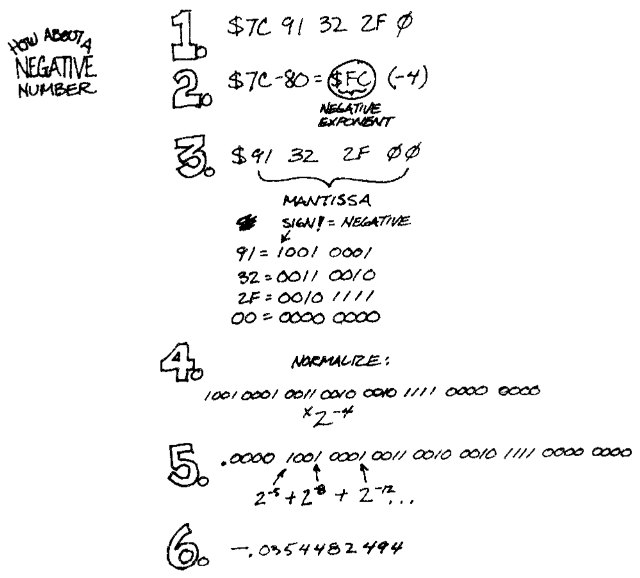
* In the case of floating-point variable N, what is found at VARPTR(N)-2 through VARPTR(N)+4?

VARPTR(N)-2 and VARPTR(N)-1 are the name of the variable; VARPTR(N) is the exponent; and VARPTR(N)+1 through VARPTR(N)+4 comprise the mantissa.

* What does VARPTR mean?

Variable pointer.

These are now fractional powers of two. You must sum 1/2 plus 1/4 plus 1/8, and so on. You can think of this calculation as: zero times 1/2↑1plus 0 times 1/2↑2, etc. In this case that's 1/2↑5 + 1/2↑8 + 1/2↑12 + 1/2↑15 + 1/2↑16 + 1/2↑19 + 1/2↑23 + 1/2↑25 + 1/2↑26 + 1/2↑27 + 1/2↑28 . . . that is, –.035448249429465 . Your Color Computer would report that as the slightly less precise –.0354482494. The number stored in the computer as **$7C 91 32 2F 00** becomes –.0354482494.



While we're at it, let me quickly bring back one of the examples I gave at the start . . . I said that 0.1 was represented as **$7D 4C CC CC CD**. Have a look. **$7D** is a right-of-the-point prefix of .000 and the number **4C CC CC CD** translates to binary . . .

```
0100 1100 1100 1100 1100 1100 1100 1101
```

It's positive; the normalized one changes the number to:

```
1100 1100 1100 1100 1100 1100 1100 1101
```

Together with the prefix, it reads:

```
.00011001100110011001100110011001101
```

Calling that out in powers of two, it's 1/2↑4 + 1/2↑5 + 1/2↑8 + 1/2↑9 + 1/2↑12 + 1/2↑13 + 1/2↑16 + 1/2↑17 + 1/2↑20 + 1/2↑21 + 1/2↑24 + 1/2↑25 + 1/2↑28 + 1/2↑29 + 1/2↑32 + 1/2↑33 + 1/2↑35 . . . and that calculates to .10000000000582. Notice that residual .00000000000582 tacked on to the end of the number. That's the tiny binary error that you've probably experienced creeping into BASIC calculations.

More on that error and other floating point concepts after a break. You might be weary of all these calculations. They'll get both easier and unnecessary later. For the moment, I would like you to review the lesson up to this point, book in hand, calculating along with me on paper or on your hand calculator. There's a lot to this floating point math, and understanding how to push around those bits is vital if you wish to work with numbers on your computer.

Please review the concept of floating point numbers. When you are confident of the theory of floating point binary notation, return to the tape.

Here's a summary of floating point numbers. As stored in BASIC's variable table, they consist of seven descriptive bytes. The first two bytes are the variable name; the last five represent the number itself. The first byte of the group of five is the binary exponent, from 2↑-127 to 2↑+127. The next four bytes are the mantissa, that is, the number itself, expressed in binary digits. The leftmost bit of the 32-bit group is normalized to a one for purposes of calculation, but as stored it represents the sign of the number.

This complex-sounding process provides, in five bytes, the ability to store decimal numbers across the range +/- 1.70141176E+38 (2↑126.9999999) to +/- 5.87747201E-39 (2↑-126.9999999), with nine digits of accuracy.

There are two ways to access BASIC variables from machine language. One way is via the USR command, and the other is by accessing BASIC's variable table. The variable table storage is the one I've been describing. There is another slightly different kind of storage when the variable is transferred via USR. It's described in the Extended BASIC manual on pages 147 and 149, under the heading "USR Function Arguments". You've already read part of this, but now it should make more sense; take a few minutes to re-read that now, and pay special attention to the description of the "Floating Point Accumulator".

Open the Extended Color BASIC manual and re-read pages 147 through 149, headed "USR Function Arguments," concentrating on the new information on page 147. Read this thoroughly, as it now applies to your understanding of binary floating point representation. Return to the tape when you have completed the reading.

You haven't been told the whole story in that reading. You should know that the contents of VARPTR(X)-2 and VARPTR(X)-1 are the variable's name. VARPTR is an excellent function, one that machine language programs can use extensively.

* What is an integer?

A whole number with no fractional part.

* What is a floating-point number?

A number with a fractional part.

* What are the two parts of floating-point representation called?

The exponent and the mantissa.

* The exponent and the mantissa are in what number system?

The exponent and the mantissa are in binary (base 2).

Just for a taste of the use of VARPTR, type and enter X=1. Simply X=1, and enter. PRINT X will display the number 1. Do it to be certain; PRINT X. Now POKE VARPTR(X),&HFF. That's POKE VARPTR(X),&HFF. PRINT X. The result will be 8.50705918E+37. VARPTR(X) is the exponent of the number, which that POKE with **$FF** has raised to an enormous power. *Now* POKE VARPTR(X),1. That's POKE VARPTR(X),1. The result is just the opposite: PRINT X will reveal the amazingly small value 2.93873588E–39.

Now get things in range. POKE VARPTR(X),&H88. That's POKE VARPTR(X),&H88. Now PRINT X. You've got 128. Now mess around with the rest of the number. POKE VARPTR(X)+3,1. POKE VARPTR(X)+3,1. And then PRINT X. Now it's 128.000015. How about POKE VARPTR(X)+2,&HAA. It's 128.664078. VARPTR(X) is the power-of-two exponent, VARPTR(X)+1 through VARPTR(X)+4 are the binary digits of the mantissa. POKE around with the 5-byte descriptor at the end of this lesson; it should give you additional perspective on how those numbers are stored. As a real exercise, POKE VARPTR(X) through VARPTR(X)+4 with random numbers, and see if, by knowing the result of PRINT X, you can determine those four numbers. With the description I've given you, you should be able to do it.

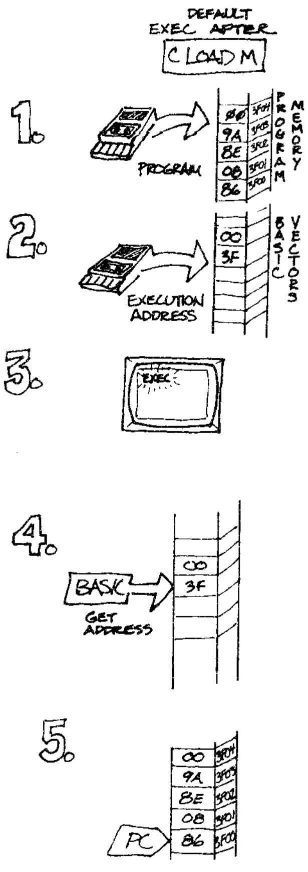
One more time for that description: five bytes, power-of-two exponent first, 32 binary digits next, with the leftmost the sign bit, but considered to be a 1 for purposes of calculation. When you've got a good handle on the floating point representation, you're ready for the next session. Give it a try, enjoy it, and I'll talk to you next time.

Program #30, 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.

```
1 CLEAR500:DIMA(5),A$(5),B$(16):A=0:B=0:FORI=0TO15:READB$(I):NEXT
2 CLS:INPUT"NUMBER TO DISPLAY";A:PRINT:B=VARPTR(A):FORI=1TO5:A(I)=PEEK(B+I-1):NEXT
3 FORI=1TO5:A$(I)=HEX$(A(I)):IFA(I)(16THENA$(I)="0"+A$(I)
4 NEXT:PRINT"   FLOATING POINT STORAGE OF":PRINTTAB(9)A
5 PRINT:PRINTTAB(6)"$ ";:FORI=1TO5:PRINTA$(I)" ";:NEXT:PRINT
6 PRINT:PRINT"   BINARY REPRESENTATION OF":PRINTTAB(9)A:PRINT
7 IFA=0THENC$=STRING$(32,"0"):GOTO14
8 G=VAL("&H"+LEFT$(A$(2),1)):C=8OR G:G=G AND8:A$(2)=HEX$(C)+RIGHT$(A$(2),1)
9 FORI=2TO5:C=VAL("&H"+LEFT$(A$(I),1)):D=VAL("&H"+RIGHT$(A$(I),1))
10 C$=C$+B$(C)+B$(D):NEXT:E=A(1)-128:IFE(0THEN11ELSE12
11 E=-E:C$="."+STRING$(E,"0")+C$:GOTO13
12 IFE(LEN(C$)THENC$=LEFT$(C$,E)+"."+RIGHT$(C$,LEN(C$)-E) ELSEC$=C$+"("+STRING$(E,"0")+".)"
13 IFG=8THENS$="(-)"
14 C$=S$+C$:PRINTC$:PRINT:PRINT"  TOUCH (ENTER) TO CONTINUE"
15 DATA0000,0001,0010,0011,0100,0101,0110,0111,1000,1001,1010,1011,1100,1101,1110,1111
16 A$=INKEY$:IFA$=""THEN16ELSERUN
```

21.

You've arrived. This is the time for putting what you know about assembly language together with what you know about BASIC. I'll get right to it by reviewing how BASIC hands over control to a machine language program: through the USR function and via the EXEC call.



EXEC is used for execution after a load from cassette, but EXEC also is the simplest, and probably the best choice for including in a BASIC driver program for fast program speed. EXEC is a direct transfer of control to the machine language subroutine, where no attempt is made to pass along variable information. EXEC 12288, for example, departs from the BASIC program in progress and begins machine language execution at address 12288 (hex **$3000**). The Morse Code program, where the start and end of the message are fixed, and the Game of Life, which simply begins, need no more information than an execution address. In the Game of Life, EXEC 3200 provided the starting address (3200 is **$0C80**) and told BASIC to relinquish control to the machine language program. That was it.

EXEC stores the address you provide in a memory location accessible to BASIC, so that the next time you use EXEC, you need not specify an address; it will automatically use the last one until you change it. Also, the entire BASIC re-entry information is stored, so — unless you've messed around things in BASIC's direct page — all will be intact when your machine language program reaches its final RTS (return from subroutine).

That does bring up the question of BASIC's direct page, and in fact, where you can store the machine language programs you will be writing. The back of your BASIC and EDTASM+ manuals gives you some information about how your computer's operating parameters are organized; turn to page 63 of the EDTASM+ book. Pay careful attention to the opening information on page 63. The manual states that when BASIC is in use, the direct page register is pointed to page00; it notes that BASIC requires certain portions of this page, as well as quite a bit of page

Unless you have high hopes for creating full-scale commercial software, chances are you'll be using BASIC as your home base. A BASIC driver can be simple or complex, performing straight-forward program executions, transferring variables back and forth, and creating graphics frameworks using BASIC's powerful drawing commands. No matter what your goal, knowing the relationship between BASIC and assembly language can speed your programming.

* The simplest command to execute a machine language program from BASIC is what?

EXEC.

* What information does EXEC require?

The starting address of the machine language program.

* EXEC&HA0D7 means what? (Try it!)

Begin executing a machine language program beginning at address $A0D7.

01. Other portions here and there are marked, "can be used for machine-code programs."

Look at number 6 in the third column. Entitled "User Memory," this is described as "Total space for user machine-language routines. No space is reserved for this on start-up, but this can be reset by the CLEAR statement."

Now what about all this? How can you be confident that the program you place in memory will stay there? And that the program will be accessed as expected? How is memory protected from BASIC? What does "protected" mean? And how is your program protected from other machine-language programs?

Frankly, the answers to those questions depend on how you plan to use your software. If you're going to use only your own software, and use your software with BASIC as you see it described in these manuals, then you're safe. But . . .

. . . if you plan to use commercial software, such as special printer drivers or communications programs or math routines or whatever, you run the risk of having your program conflict with that program.

. . . if you plan to use a disk system, especially OS-9, the memory mapping of these devices may alter the area you plan to use.

. . . if you intend to write software to sell, you must expect that memory conflicts will arise with both other commercial software and the user's software — somewhat the converse of what I said earlier.

There are options. I'm not offering any business advice if you plan to sell your software, but I can recommend that you make your commercial machine-language programs position independent. Use the guidelines and approach as presented in session 19. If your program must be position dependent for reasons of speed or memory economy, then provide with it a relocator — a companion machine language routine that will automatically rework the program to fit in another area of memory. Beyond those recommendations, you're on your own as a software businessperson.

For most important programming, I'll stand by my position-independence or relocation recommendations. Let me tell you how to make position independence work for you on the Color Computer — after you've already written the position-intependent software, of course. Here's how:

In your assembly listing, place the origin at **$0000**. You can do that by specifically typing ORG **$0000** for reference, or by leaving out the ORG statement. The assembler assumes **$0000** if you don't specify otherwise. Save the source program to tape, and also assemble and save the object code.

* What does "&H" mean in BASIC?

&H means hexadecimal.

* How does a machine language program get back to BASIC?

By executing RTS (return from subroutine).

* When BASIC is operating, where is the 6809's direct page register?

The direct page register is set to $00.

* What is a position independent program?

A program designed to run correctly no matter where it is located in memory.

* Which of these is a position independent command: LBRA TRAK1 or JMP TRAK1?

LBRA TRAK1 is position independent.

* What addressing mode is LBRA TRAK1?

Relative addressing.

* Which of these is a position independent command: LDX #TABLE or LEAX TABLE,PCR?

LEAX TABLE,PCR is position independent.

* What addressing mode is LEAX TABLE,PCR?

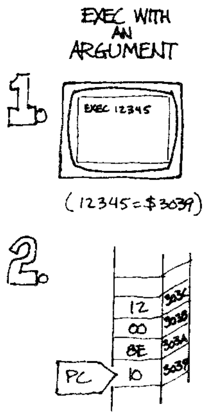
Relative addressing (specifically, program counter relative).

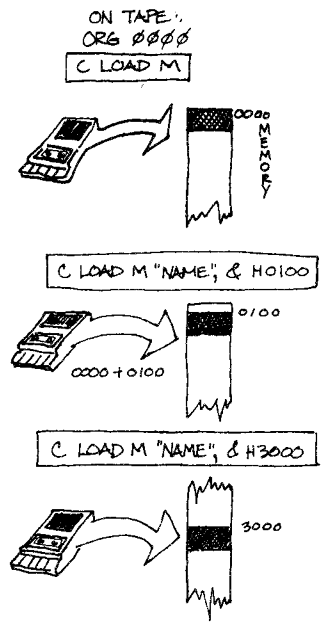
* What does LEAX -1,X mean?

Let X become X-1.

* What does LEAX $45,Y mean?

Let X become Y+$45.





When you want to load and run this machine language program, you use the command CLOADM. If the program's name is "Blurb", for example, you would normally type
CLOADM"BLURB". With position-independent programs, you need something more. You need an offset address, an address which is added to the CLOADMed address to produce a resultant location in memory where the program is going to be stored. For example, if the program is to load into memory beginning at 12288 (hex **$3000**), you would type CLOADM"BLURB",12288 or CLOADM"BLURB",&H3000. The program will add up your origin ($0000) and the offset address ($3000), and begin loading the program into that area of memory. Since your origin is **$0000**, then the offset address turns out to be the same as the loading address. And since the program is position-independent, it can start running as soon as you type EXEC.

I've got a very brief object program coming up. All this program does is load a short ASCII message into memory. The source program goes like this (you can glance in the book):

```
ØØ1ØØ          FCC   /THE MESSAGE IS HERE/
ØØ11Ø          END
```

After I assemble this, I will have nothing but a group of 19 bytes on the resulting tape. The lack of a specified origin means that the assembler will place these 19 bytes beginning at address **0000**. The program coming up next is dumped to tape ten times; its name is "TEXT". So if you want to see this message, what you want to type is CLOADM"TEXT",1024. This will load the message to the first space on the video screen. Then try any location on the screen and see where the subsequent nine messages come into view. Remember that the normal screen is mapped from **$0400** to **$0600** (1024 to 1535 decimal), so to see the whole message, your offset addresses should be in that range. So try these ten messages; I'll be back with a description of what memory is free and how to use it.

Programs #31A to 31J, object code programs. Turn on the power to your Extended Color BASIC computer. When the cursor appears, type CLOADM and press ENTER. The computer will search (S) and find (F). When the cursor reappears, type EXEC and press ENTER. The program will execute automatically. If an I/O error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix. For additional loading of programs 31B through 31J, refer to the text.

What you've just done is load blocks of binary information into memory. Since the binary information was saved to tape with a loading address of **$0000**, the offset addresses you specified in the CLOADM command became the actual loading addresses for the binary data.

* What cassette command loads BASIC programs?

CLOAD.

* What cassette command loads machine language programs directly into memory?

CLOADM.

* What does ORG mean?

Origin, that is, the first byte of a source listing.

* If ORG is left out of an assembly listing, where does the assembler begin assembly (the default ORG)?

At address $ØØØØ.

* If a machine language program named TESTER were ORGed at $ØØØØ, what BASIC command would load that program?

CLOADM or CLOADM"TESTER".

* If a machine language program named TESTER were ORGed at $ØØØØ, what BASIC command would load that program starting at $2ØØØ?

CLOADM"TESTER",&H2ØØØ or CLOADM"TESTER",8192

* What is the value appended to the CLOADM command called?

An offset address.

* Where is the normal video screen in the Color Computer memory map?

At locations 1Ø24 through 1535 ($Ø4ØØ through $Ø5FF).

* Where do the high-resolution video screens begin on the Color Computer?

At location 1536 ($Ø6ØØ).

* How much space does PCLEAR1 reserve?

1536 bytes ($0600 bytes).

* How much space is reserved by PCLEAR1 through PCLEAR8?

1536 through 12,288 bytes.

* If high-resolution graphics will not be used by BASIC, can machine language programs be stored in the area reserved by PCLEAR1?

Yes.

* How much space is reserved by PCLEAR1, and what are the addresses?

1536 bytes are reserved from $0600 to $0C00.

* What is the purpose of CLEAR?

To reset all BASIC variables.

* What is the purpose of CLEAR N, where N is a number?

To reset all BASIC variables, and to reserve N bytes for BASIC string manipulations.

* What is the purpose of CLEAR N,X where N and X are numbers?

To reset all BASIC variables, to reserve N bytes for BASIC string manipulations, and to protect memory from BASIC beginning at address X.

* What effect does CLEAR200,16384 have?

It resets BASIC variables, sets aside 200 bytes for BASIC string manipulations, and makes 16384 ($4000) the start of protected memory.

* What is protected memory?

Memory that is not available for BASIC's use.

Such position-independent programs can, naturally, be moved as often as you wish. The next question, therefore, is where do you put the programs?

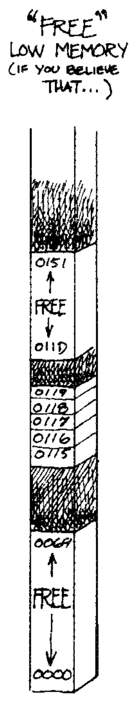
There are four places in your computer's read/write memory for convenient storage of machine language programs. You may store this binary data in low memory as provided in the memory map; in high memory protected from BASIC; in high-resolution graphics memory; and inside a BASIC program line. Each in turn now.

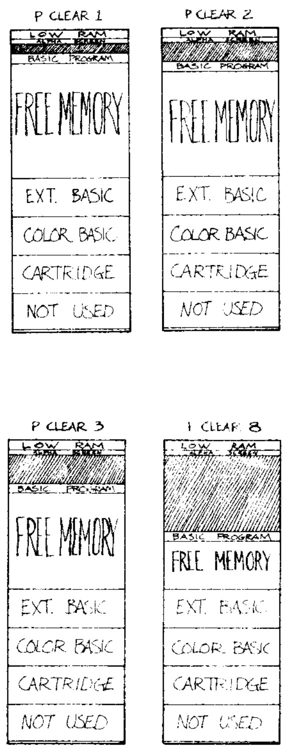
Storing a program in low memory is not safe from **$0000** to **$0069**. These are 106 bytes called "free", and there is also so-called free memory at **$0115** to **$0119** (five bytes), and a block of 53 bytes from **$011D** to **$0151**. None of this is safe for program storage; don't use it. The EDTASM book's phrase "can be used by machine language programs" means that you can store data here while your machine language program is underway. When you return to BASIC (especially if your machine language program is an integral part of a running BASIC program), the information BASIC needs in low memory is likely to be altered. However, with one of the excellent detailed Color Computer memory maps that have been published, you can learn how and when low memory is used by BASIC. So for now your rule of thumb about low memory is: don't.

There are three remaining options, and, with only a few reservations, all of these options are good ones. They are high memory, high-resolution graphics memory, and memory inside a BASIC program.

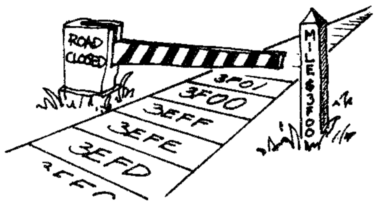
Storing programs in graphics memory is easy and reasonably safe. PCLEAR is BASIC's way of reserving high-resolution graphics memory, and PCLEAR1 is the smallest amount of graphics reserved memory allowed. PCLEAR1 allows 1,536 bytes of memory from **$0600** to **$0BFF** to be used for storing a machine language program. There are two major caveats to this process. Most obvious is the fact that you can't use high resolution graphics if you choose this method. Since this memory is intended to be a graphics screen, using any graphics command risks damaging the stored program. The other warning is the POKE often used on memory location 25, where the PCLEAR number of graphics pages are stored. Since PCLEAR0 is not allowed, a POKE to that location has been popularly used to free up some extra memory. But by doing that, you wipe out the graphics screen and the machine-language program along with it. So, in summary, use graphics memory for your program only if you can be sure BASIC will not be using high-resolution graphics commands.

The most popular mode of storing binary code is by placing it in protected high memory. BASIC is specifically set up to allow this use, and I consider it wisest to follow those recommendations when all other considerations seem equal.





Protecting high memory is an easy task. The BASIC CLEAR statement is used for this. The CLEAR statement performs three functions: CLEAR alone resets all BASIC variables and arrays; CLEAR followed by one number (as in CLEAR200) sets aside space for BASIC's string functions; and finally, CLEAR followed by two numbers (as in CLEAR200,14000) sets aside string space as well as creates a boundary beyond which BASIC may not trespass.



In the case of CLEAR200,14000, memory locations 14000 (that's hex **$36B0**) and above are not used by BASIC. It's as if your computer only had 14,000 bytes of memory instead of 16,384 for a 16K machine or 32,768 for a 32K machine. The only commands that will affect this memory are POKE (which can change any RAM location or output address) and CLOADM (which will attempt to load its data to the specified address whether memory is present there or not).

Here's how it works. If you have a 400-byte machine language program which you want to store in the high memory of a 16K computer, you first determine if the program is position independent. If you wrote the program, then you'll know for sure; otherwise, read the documentation for help. Assuming you know that the program will load and run in those top 400 bytes, you then subtract 400 from the highest memory location in the 16K computer. 16383 minus 400 is 15983. Then you'll need to determine if whatever BASIC program you're about to run will need more or less string space; the space allocated at power-up is 200 bytes. With that in mind, construct the CLEAR statement in the form CLEAR string space comma memory barrier. To protect 400 bytes and have the normal amount of string space, you would enter CLEAR200,15983.

Memory is protected and you are ready to CLOADM your machine language program or other binary data. For more information on CLEAR, use your BASIC manuals.

The final method of placing a binary program into memory is called the in-string or string-packing method. This technique was first popularized on the classic TRS-80 Model I, and remains a favorite for short, position-independent programs. Keep in mind that this isn't a universal technique; it expects certain features found only in Microsoft's dialect of BASIC.

To understand string-packing, you have to do a little rethinking about BASIC itself. The BASIC you're most familiar with is a programming language. You don't often think of a BASIC program as anything but what it represents. But a BASIC program is something different from what it represents... it is something that fills memory. And something that fills memory is binary information. A machine-language program is binary information. If both are binary information, can they coexist in a single listing?

* Can a machine language program be stored in this protected memory?

Yes.

* Once a machine language program is in place, what commands are used to execute the program?

EXEC or USR.

* In the Color Computer, what essential computer "matter" does a BASIC program consist of?

Binary information.

* What does a machine language program consist of?

Binary information.

* Microsoft created Color and Extended Color BASIC. What technique of storing a machine language program can be used with Microsoft BASIC?

String-packing.

* What is string-packing?

Placing a machine language program inside a BASIC string, within the BASIC program itself.

* What information does BASIC's VARPTR provide?

The address of a BASIC variable.

* What three pieces of information does VARPTR provide about a string?

The variable name, the length of the string, and the address of the first byte in memory where the string is located.

* What is the longest line that can be typed in BASIC?

240 characters.

POKEing a string

* If a string variable name takes two bytes, an equal sign takes one byte, and the quotation marks take two bytes, how many bytes are left for the string itself?

235 bytes.

* What is the longest machine language program that can be stored in a BASIC string?

235 bytes.

* What three BASIC commands are essential for string packing?

VARPTR, PEEK and POKE.

* Why is VARPTR necessary for string packing?

VARPTR is needed to locate the address of the string's vital information in memory.

* Why is PEEK necessary for string packing?

After VARPTR provides the address of the string information, PEEK is used to determine the length and address of the string itself.

* Why is POKE necessary for string packing?

The instructions and data that make up the machine language program must be POKEd into memory where the string currently resides.

* Why are commands POKEd in place?

Because all 256 possible combinations from $00 to $FF cannot be typed from the keyboard.

* What two BASIC commands are used to run (access) a machine language program?

EXEC and USR.

Don't think about the source code now. Think about the binary code. And consider that BASIC has at least two situations in which it does not tamper with or interpret information as part of a program. In other words, there are two situations in which BASIC doesn't mess with what you type: after a remark (REM) statement and inside the quotation marks of a string variable.

One of these two situations is of special value. Recall the last session when I talked about floating-point arithmetic. I mentioned a BASIC command called variable pointer, or VARPTR. The command VARPTR points not only to floating-point numbers, it points to any variable. So type, enter and run this one-line program . . .

```
10 A$ = "THESE ARE THIRTY-ONE CHARACTERS"
```

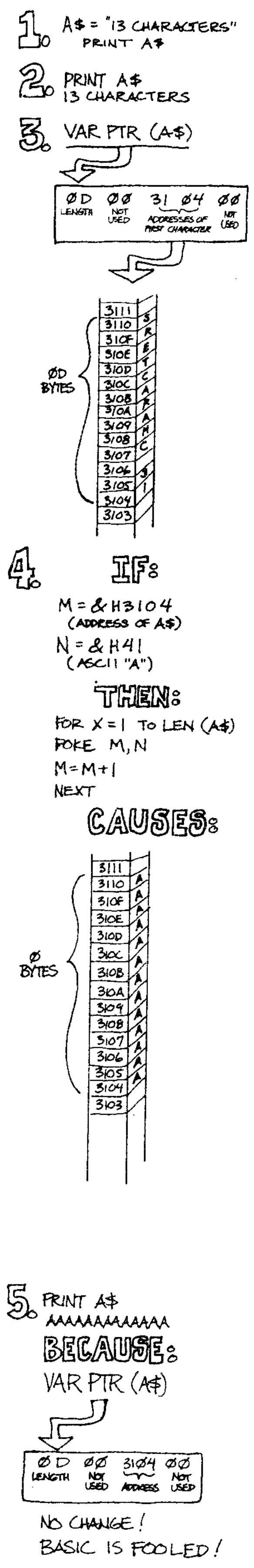
. . . you can then ask for VARPTR(A$). PRINT VARPTR(A$). The computer will report 7726. At memory location 7726 is information about A$, five bytes of it. In your reading of the Extended Color BASIC manual for the last session, you may recall that the first byte is the length of the string, and the third and fourth bytes are the address of the first character in the string.

There's the clue. The third and fourth bytes of this descriptive information are the address of the first character of the string. If you create a string of the correct length, and if you know where the string is in memory, and if you are confident that BASIC won't mess with the strings as they appear in program lines (it won't), and if — at last — you don't plan to use that variable for any other purpose within the program . . . then you can safely store a machine-language program within those quotation marks.

I wish you could just type such a program right into the string. You can't, of course, because you might need any one of the 256 possible bytes for your machine program, but only about 96 are typable from the keyboard. So you have to go in the back door.

The key to the back door is right there in the mailbox. You use the variable pointer VARPTR to find out where A$ sits in memory, you take a listing of your program in hexadecimal bytes, and you POKE them, one at a time, into the place occupied by A$. You could POKE bytes one at a time by hand, but there are easier ways. I have a program to show you the details; list it, but don't run it until I'm back with you.

Program #32, 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 A=0:B=0:C=0:D=0:N=0:Q=0:X=0
20 B$=""
30 A$ = "14 CHARACTERS!"
40 A = VARPTR(A$)
50 B = PEEK(A+2)
60 C = PEEK(A+3)
70 D = B*&H100 + C
80 DEFUSR0 = D
90 FOR X = 1 TO LEN(A$)
100 READ B$
110 POKE D,VAL("&H"+B$)
120 D = D+1
130 NEXT X
140 DATA BD,B3,ED,8E,04,00,E7
150 DATA 80,8C,06,00,26,F9,39
160 STOP
170 INPUT Q
180 N = USR0(Q)
190 FOR X = 1 TO 500 : NEXT
200 GOTO170
```

LIST this program. Notice that A$ is 14 characters long. And notice that in lines 140 through 150 are a group of what look like hexadecimal numbers, presented as DATA statements, and following them is a short routine to read, convert and POKE them in place. The program, which you can hand-disassemble (that is, convert from hex to source code), simply fills the screen with any character you input. Look at these statements:

```
A = VARPTR(A$)
B = PEEK(A+2)
C = PEEK(A+3)
D = B * &H100 + C
DEFUSR0 = D
```

There are ten USR calls allowed by BASIC, USR0 through USR9, meaning you can have up to ten different machine language programs. DEFUSR identifies for BASIC the starting address of the machine-language program. In this case, the program is stored in A$, so variable A finds VARPTR(A$), and variables B and C obtain the two address bytes where A$ can be found. Since BASIC's workings are in decimal, you can't just dip in and pull out a 16-bit address; you've got to combine the most-significant byte with the least-significant byte to get a result. In case you hadn't thought of it this way before, you'll notice that in hexadecimal, the most-significant byte is always **$100** times the least-significant byte. So the resulting address is hex **$100** times B, plus C.

All of this could be combined into the complicated looking formula DEFUSR0 = &H100 * (PEEK(VARPTR(A$)+2) + PEEK(VARPTR(A$)+3). Now matter how you write it, it defines where the machine language program starts.

Now RUN the program; it will BREAK in 160. LIST the program, and have a look at A$. It looks longer now (it isn't) and seems to be garbage. Type and enter PRINT A$. A peculiar but different result.

* If a machine language program is at $3000, use EXEC to access it.

EXEC&H3000.

* What must be done before a machine language program can be accessed with USR?

The entry point must be defined.

* What BASIC command is used to define the USR entry point?

DEFUSR.

* How many USR entry points does Extended Color BASIC offer? What are they?

Ten entry points, USR0 through USR9.

* What is the advantage of USR in certain situations?

USR can transfer information to the machine language program.

* If the machine language program begins at $3000, define the USR0 entry point.

DEFUSR0=&H3000

* If the information to be transferred to the program beginning at $3000 is 12345, give the USR0 command to access the program and transfer the information.

```
PRINT USR0(12345) or N = USR0(12345)
```

* Give the formula to define a USR9 entry point to a machine language program stored in N$.

```
DEFUSR9 = &H100 * ( PEEK(VARPTR(N$)+2 + PEEK(VARPTR(N$)+3 )
```

* In string packing, what values should be avoided if possible? Why?

$00, because it is BASIC's end-of-line marker; and $22, because it is equivalent to an ASCII quotation mark.

* How does a machine language program get back to BASIC?

By executing RTS (return from subroutine).

* What is a position independent program?

A program designed to run correctly no matter where it is located in memory.

Recall many lessons past when I said that a single 8-bit word of memory had to serve many purposes. Now you see them. You have a machine language program stored inside A$'s quotation marks. When you print it, it looks like graphics and ASCII characters. When you list it, it looks like BASIC commands. What is it? It's still your machine language program, but the PRINT routine doesn't know that; the BASIC PRINT routine thinks it's a string to print. The BASIC LIST routine thinks you somehow stuffed commands inside the quotation marks. You can ponder that on your own; I'm getting back to the program.

You've already run this program, so just type and enter CONT for continue. The prompt asks you for a value from 0 to 255. Enter a value. The screen fills with your character and returns to BASIC; nothing new here. The machine language program is at work. Try more. Each time, the screen fills almost instantly.

Now tap BREAK, and LIST this program again. Notice in line 170 that the program inputs variable Q, and in line 180 the command N = USR0(Q) is encountered. The variable Q is passed to the machine-language program, which converts it and uses it to fill the screen.

There you have it: VARPTR used to find a string in a BASIC program, the machine-language program packed into that string, DEFUSR set to point to the start of the program packed into the string, and USR commanded to execute the program. One of the slickest methods ever devised.

Just a few warnings. First, be careful not to use the variable over again in the program. It won't erase where the machine language program is, nor its contents, but at some point either the BASIC or the machine language might end up misinformed about where things are.

Second, save the program before running it, and once you've run the lines containing A$ and all the POKEs, don't run past them again. Here's why: there are two hexadecimal values that can't appear in the string. One of them is **$22** and the other is **$00**. **$00** is used as BASIC's end-of-line pointer, so when it sees **$00**, it thinks it's reached the end of a BASIC line. Again, the program might end up misdirected. The value **$22** happens to be the ASCII value for a quotation mark, and more than two quotation marks will cause a dreaded syntax error.

The value **$00** is the opcode negate direct; you won't use that much. But **$22** is, unfortunately, PSHS, and that one's almost unavoidable. Have a look at this program, and when you're done, I have one more. Study them both before the next session.

Program #33, 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/0 error occurs, rewind to the start of the program and try again. For severe loading problems, see the Appendix.

```
1 A=0:X=0:B=0:C=0
2 A$="THIRTY-FIVE CHARACTERS ARE NEEDED!!"
3 A=VARPTR(A$):C=256*(PEEK(A+2))+PEEK(A+3)
4 FORX=1TOLEN(A$):READB$:B=VAL("&H"+B$)
5 POKEX+C-1,B:NEXT:DEFUSR0=C
6 CLS:PRINT:PRINT"YOUR BORDER CHARACTER"
7 INPUT"ENTER A NUMBER FROM 0 TO 255";A
8 IFA<0ORA>255THEN7:ELSEM=USR0(A)
9 FORN=1TO1000:NEXT:GOTO6
10 DATABD,B3,ED,8E,04,00,86,21,E7,80,4A,26,FB,30,1F,86,0E,30
11 DATA88,1F,E7,80,E7,84,4A,26,F6,86,20,E7,80,4A,26,FB,39
```

22.

What was I saying? Oh yes. Interrupts. Let me take you back to Sam's Kitchen in Roadside, New Jersey, where you can honk for drive up service from noon to 6. Have another listen to Marge at work . . .

Marge: One fries, two BLTs, three chili dogs . . . <honk> Alright, alright . . . and one onion rings. Get those ready. There's a guy out there honkin' that thing like Little Richard. <outdoors> Yeah, what'll you have?

Car one: Three burgers, two fries, a shake.

Marge: Ya want bunny burgers or buddy burgers?

Car one: One bunny burger, two buddy burgers.

Marge: <indoors> One bunny, two buddies, fries. Where's my order? <at counter> Anything else, Joe? How 'bout you, Mac?

Mac: Yeah, gimme another dog, will ya Marge? With onions an' cheese, too.

Marge: Cheese dog onions.

Kitchen: Orders up.

Marge: Hey where's my steak? And what about . . . <honking> . . . the chili dog. Damn. Gotta get that. yeah, yeah, whaddaya want?

Car two: Gimme three bunnies and . . . <honking from third car>

Marge: <to third car> Hey fell I'm busy. Sit on it till I get to ya. <back to car> Three bunnies. What else, and make it quick.

What was I saying? Oh yes. Interrupts. Having been to Sam's Kitchen twice, you should have an idea that interrupts are crucial to special kinds of programming. But what kind of program would demand such fancy footwork? If the programming is so tricky, why bother?

* Three things happen when an interrupt occurs. What are they?

The microprocessor finishes its current instruction, saves important information, and follows programming instructions in response to the interrupt.

* What is the process of acting on an interrupt called?

Servicing the interrupt.

* What causes an interrupt?

When an external signal line changes from one to zero.

* Can more than one interrupt occur?

Yes.

* Which interrupt gets taken care of first?

The one with higher priority.

Car two: How about filet mignon and truffles and leeks vinaigrette . . .

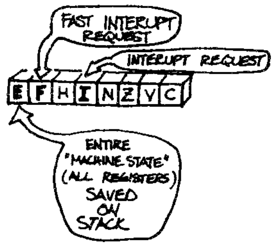
The restaurant is the computer, and Marge is the microprocessor. The cook and customers are program and storage memory. The car horn was the interrupt. Marge finished was she was doing, serviced the interrupt, and returned to finish her previous task. When two interrupts occurred, car two had a higher priority. Finally, the drive-up interrupt was masked out except from noon to six.

The 6809E processor has one power-up reset signal, three hardware and three software interrupts, plus two unique instructions to synchronize itself with hardware interrupts. All of these 6809 interrupts are possible on the Color Computer, and some are already in use by BASIC.

The RESET control is used when the power is turned on to the computer, or when the reset switch is pressed on the back of the machine. It is a separate electrical connection to the 6809 processor, and the RESET cannot be masked by software; it is always accepted.

The most important of the interrupts — that is, the interrupt with the highest priority — is the NMI, or non-maskable interrupt. It is a separate electrical connection to the processor and, like RESET, it cannot be turned off by software. It always commands the attention of the processor.

Of next highest priority is the fast interrupt request, or FIRQ. The FIRQ can be turned off in software by setting bit 6 (the F bit) of the condition code register. **ORCC #$40** can be used to set this bit, turning off the interrupt; **ANDCC #$BF** can be used to clear bit 6 to turn on the interrupt. When the FIRQ comes along, the condition code register and program counter are put on the stack, and the interrupt service routine is begun. The FIRQ is fast because it leaves the remainder of the register stacking up to the interrupt service routine. If a register is not used, it won't need to be put on the stack. I'll talk about the requirements for speed later on.



The interrupt with the lowest priority is called simply the interrupt request, or IRQ. When a zero appears on this electrical connection to the CPU, all the registers — what's known as the entire machine state — are saved on the stack. This interrupt is turned off in software by setting bit 4 (the I bit) of the condition codes, and turned on by clearing bit 4. **ORCC #$10** turns it off; **ANDCC #$EF** turns it on.

ORCC #$50 turns off both interrupts; **ANDCC #$AF** turns on both interrupts.

You'll remember that I described indirect addressing by explaining how the computer obtained its first instruction after the power was turned on. The processor went to addresses **$FFFE** and **$FFFF**, concatenated the contents, and used that as the address of the first instruction. There

* Can interrupts be ignored?

Yes.

* What permits the processor to ignore an interrupt?

Masking the interrupt.

* What determines whether an interrupt is masked or enabled?

The condition code register.

* What part of the condition code register determines whether an interrupt is masked or enabled?

Bits 4 and 6.

* What masks an interrupt?

Setting its condition code bit to a one.

* What commands can be used to affect the condition code register directly?

ANDCC and ORCC, both immediate instructions.

* What command specifically masks out (turns off) both interrupts?

ORCC #$50 (binary 01010000).

* What command specifically enables (turns on) both interrupts?

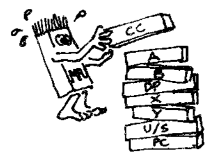
ANDCC #$AF (binary 10101111).

* Three things happen when an interrupt occurs. What are they?

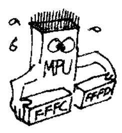
The microprocessor finishes its current instruction, saves important information, and follows programming instructions in response to the interrupt.

are in fact seven such address pairs, called "vectors". Power-on reset plus each of the six interrupts has its own vector from **$FFF2** to **$FFFF**.

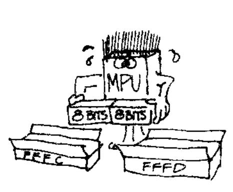
Here's how these vectors look in the Color Computer:



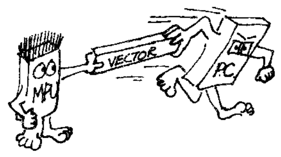
FUNCTION	VECTORS	ADDRESS	CONTENTS
RESET	FFFE+FFFF	AØ27	< BOOT >
NMI	FFFC+FFFD	Ø1Ø9	--------
SWI1	FFFA+FFFB	Ø1Ø6	--------
IRQ	FFF8+FFF9	Ø1ØC	JMP 894C
FIRQ	FFF6+FFF7	Ø1ØF	JMP AØF6
SWI2	FFF4+FFF5	Ø1Ø3	--------
SWI3	FFF2+FFF3	Ø1ØØ	--------



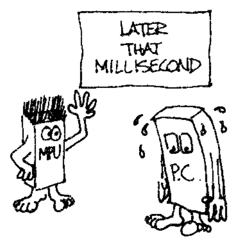
The power-up RESET goes right to address **$A027**, a location in Color BASIC which establishes all the important parameters of the language.



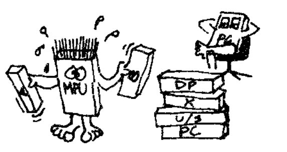
NMI is not used by Color BASIC or Extended Color BASIC, but three unfilled bytes in low RAM are reserved for future use. The future use is provided because the NMI is wired to connection #4 on the computer's cartridge slot.

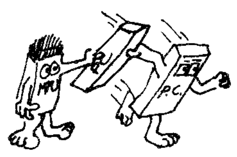


Software interrupts SWI1, SWI2 and SWI3 are also left undefined with three unfilled bytes at their vector locations; they are used by debugging programs such as ZBUG, part of your EDTASM+ cartridge. Yes, we will talk about debugging . . . next time. On to the other interrupts.



FIRQ, the fast interrupt, is hooked to one of the peripheral interface adaptors, connecting to both the PIA's interrupt output lines. The input to the PIA's interrupt control signals are two: the carrier detection (CD) line of the RS-232 communications interface, and the cartridge-in-place (CART) connection, #8 on the computer's cartridge connector. This interrupt serves a dual purpose. When FIRQ occurs, the vector concatenated from addresses **$FFF6** and **$FFF7** point to address **$010F**; at address **010F** is the instruction **JMP $A0F6**, a location in the Color BASIC ROM.





The slower interrupt IRQ is connected to the second peripheral interface adaptor, also to both of its interrupt outputs. The interrupt control inputs of this PIA are connected to the horizontal synchronization (HS) and field or vertical synchronization (FS) outputs of the video display generator. Again, this interrupt serves a dual purpose. When IRQ takes place, the address in vectors **FFF8** and **FFF9** are concatenated to produce address **$010C**. At **$010C** is found the instruction **JMP $894C**, an address in the Extended Color BASIC ROM.

* Is there an interrupt that cannot be masked (turned off)?

Yes.

* What interrupt cannot be masked?

The non-maskable interrupt, or NMI.

* What interrupt has the highest priority?

The NMI.

* What interrupt has the second highest priority?

The fast interrupt request, or FIRQ.

* What bit of the condition code register masks or enables the FIRQ?

Bit 6 masks or enables the FIRQ.

* What information is saved when the FIRQ occurs?

The condition code register and program counter are saved on the stack.

* What is the lowest priority interrupt?

The interrupt request, or IRQ.

* What bit of the condition code register masks or enables the IRQ?

Bit 4 masks or enables the IRQ.

* What information is saved when the IRQ occurs?

All the registers are saved on the stack.

* What is the process of acting on an interrupt called?

Servicing the interrupt.

* How does the program counter find where to go to service the interrupt?

From a vector, or address, in the last 16 bytes of memory.

* What purpose does NMI serve on the Color Computer?

None; it is not used.

* What purpose does FIRQ serve on the Color Computer?

It is used for the RS-232 communications carrier detection line, and for the cartridge-in-place connection on the cartridge connector.

* What purpose does the IRQ serve on the Color Computer?

It is connected to horizontal and vertical synchronization signals from the video display generator.

* What are the terms for vertical and horizontal synchronization with respect to the Color Computer.

Field sync (FS) and horizontal sync (HS).

* How often does the field sync (FS) signal occur?

60 times per second.

* How often does the horizontal sync (HS) signal occur?

15,720 times per second.

* What port address determines which interrupt is fed through to the 6809 processor?

Port address $FF03.

* What condition code bit masks or enables the IRQ?

Bit 4 masks or enables the IRQ.

In all these cases, the addresses in low RAM can be changed or filled in, redirecting the interrupts to any location in memory. You'll be using those addresses.

Now I've given you a formal description of the vectors and the hookup, but I expect it doesn't mean a whole lot to you at this point. I'm going to continue with a detailed description of how everything fits together into a neat package, but first I want you to read the technical information.

Read the MC6809E data booklet page 9 (NMI, FIRQ, IRQ); read the MC6821 data booklet page 7 (peripheral interface lines) and page 8 (internal controls), and Figure 18, page 10; read the MC6847 data booklet page 13 (Field Sync and Horizontal Sync). If you have the Color Computer Technical Reference Manual, read Section III (Theory of Operation). Return to the tape when you have completed the reading.

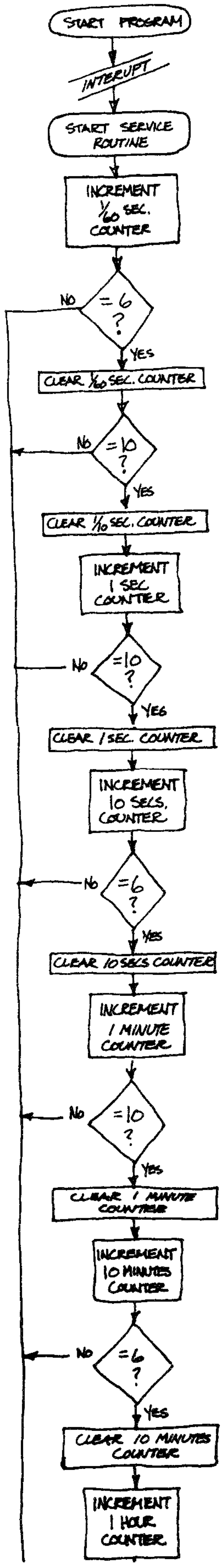
Read the MC6809 data booklet page 9 (NMI, FIRQ, IRQ); read the MC6821 data booklet page 7 (peripheral interface lines) and page 8 (internal controls), and Figure 18, page 10; read the MC6847 data booklet page 13 (Field Sync and Horizontal Sync). If you have the Color Computer Technical Reference Manual, read Section III (Theory of Operation).

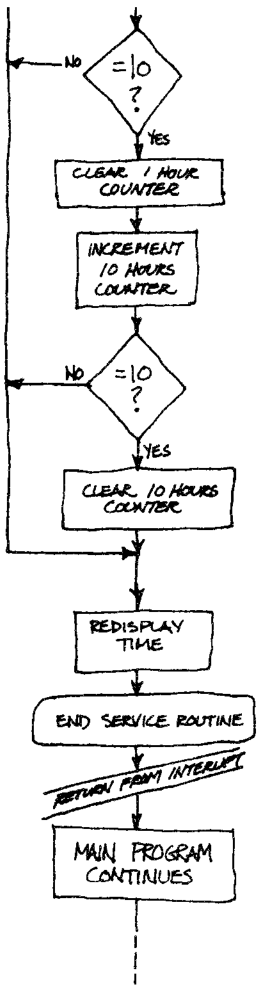
Now putting it together. By correctly writing data to the PIAs, you can make it possible for the computer to detect an RS-232 carrier, to detect the presence of a plug-in cartridge, or to synchronize your programs to the video display either horizontally or vertically. All you need to add is software.

I've got two demonstrations of this. The first is a continuous on-screen software clock; the second, an example of synchronizing the video display with programming changes to the screen.

I'm going to put a clock in the upper right corner of the video screen. It will be there no matter what else is displayed on the screen, whether you're listing, entering or editing a line, or running a BASIC program. It will even keep running with certain machine language programs that don't turn off interrupts or change the vectors. I think I'd like it to count in tenths of a second up to 99 hours, 59 minutes, 59.9 seconds.

You've read the data booklets, so maybe you're ahead of me. Remember the video display generator's field sync (FS) signal, which is used for interrupting the processor. The video display generator's field sync signal occurs at precisely 60 times each second. By enabling the interrupt (bit 0 of port address **$FF03**), I can get an interrupt to occur 60 times each second. If I keep track of those ticks and





update my screen with a new time every six interrupts, then I've got a tenth-of-a-second clock. From a tenth-of-a-second clock I can create a full real-time software clock.

Here's the structure of the setup and interrupt service routine:

1. Set aside some memory for the clock; it might be an image of the actual display (such as 12:59:02.2).

2. Enable the 60-per-second interrupts.

3. On an interrupt, increment the sixtieth-of-a-second counter. If the sixtieth-of-a-second counter passes 5, increment the tenth-of-a-second counter, and clear the sixtieth-of-a-second counter to 0. If the tenth-of-a-second counter passes 9, increment the one-second counter and clear the tenth-of-a-second counter to 0. If the one-second counter passes 9, increment the ten-second counter and clear the one-second counter to 0. If the ten-second counter passes 5, increment the one-minute counter and clear the ten-second counter to 0. If the one-minute counter passes 9, increment the ten-minute counter and clear the one-minute counter to 0. If the ten-minute counter passes 5, increment the one-hour counter and clear the ten-minute counter to 0. If the one-hour counter passes 9, increment the ten-hour counter and clear the one-hour counter to 0. If the ten-hour counter passes 9 clear it to 0.

4. Display the new time to the screen; the re-display will take place every sixtieth of a second, appearing as a continuous display.

5. Clear the interrupt status at port **$FF02**.

6. Return from the interrupt.

The setup process has to clear the way for the interrupts without getting interrupted in the middle of things. So all interrupts go off right at the start; the address of your own routine is placed into the RAM vector; the proper interrupt signal (in this case, the 60-per-second FS) is enabled; interrupts are re-enabled; and the setup routine returns to BASIC. Earlier in the book I presented a map of the computer's input/output port bits. Bit 0 of control port **$FF03** provides for the FS signal to be latched as an interrupt. So the whole routine might look like this:

```
ORCC   #$5Ø      * Turn off interrupts
LDX    #$START   * Service routine start
STX    $Ø1ØD     * Store after "JMP" in vector
LDA    #$37      * Value to enable FS
STA    $FFØ3     * Enable FS through PIA
ANDCC  #$EF      * Re-enable IRQ interrupt
RTS              * Back to BASIC
```

That's the setup. The interrupt service routine itself is really quite simple; get the whole thing loaded into EDTASM+, and then come back for a walk-through.

* What instruction masks the IRQ?

ORCC #$10 masks the IRQ.

* What instruction enables the IRQ?

ANDCC #$EF enables the IRQ.

* What instruction returns to the program in progress after an interrupt has been serviced?

Return from interrupt, RTI.

* When IRQ occurs, where does the program counter obtain the address of the interrupt service routine?

From a vector in high memory.

* What is the IRQ vector found?

The IRQ vector is found at $FFF8 and $FFF9.

* On the Color Computer, where does the IRQ vector point?

The IRQ vector points to address $010C.

* Where is $010C in the Color Computer memory map?

In RAM, on page $01.

* In the Color Computer running with BASIC, the service routine shown in this example ends with JMP $894C. Where is $894C in the Color Computer memory map?

$894C is in the Color BASIC ROM.

* Why does this service routine end with JMP $894C instead of RTI?

Because the interrupt still has to be used by BASIC for the cursor flash and the TIMER command.

Program #34

Program #34, 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.

```
3F00                00100           ORG     $3F00
                    00110 *
3F00 1A    50       00120 INTOFF    ORCC    #$50     * TURN INTERRUPTS OFF
3F02 8E    3F10     00130           LDX     #START   * POINT X TO SERVICE ROUTINE
3F05 BF    010D     00140           STX     $010D    * STORE ROUTINE TO IRQ VECTOR
3F08 86    37       00150           LDA     #$37     * VALUE 00110111 FOR MASKING
3F0A B7    FF03     00160           STA     $FF03    * TURN ON VERTICAL SYNC
3F0D 1C    EF       00170           ANDCC   #$EF     * TURN INTERRUPTS ON
3F0F 39             00180           RTS              * AND BACK TO BASIC "OK"
                    00190 *
3F10 8E    3F77     00200 START     LDX     #IMAGE+10        * POINT X TO 1/10 SEC.
3F13 C6    30       00210           LDB     #$30     * B BECOMES ASCII OFFSET
3F15 6C    84       00220           INC     ,X       * INCREMENT 1/10 SECONDS
3F17 A6    84       00230           LDA     ,X       * GET 1/10 SECONDS VALUE
3F19 81    36       00240           CMPA    #$36     * IS 6/10 SECONDS COUNTED?
3F1B 2D    2C       00250           BLT     OUT      * IF NOT 6/10 SECONDS, OUT
3F1D 8D    40       00260           BSR     DEC1     * ELSE BAC UP 1 MEM. LOCATION
3F1F 81    3A       00270           CMPA    #$3A     * IS IT 1 SECOND YET?
3F21 2D    26       00280           BLT     OUT      * IF NOT 1 SECOND, OUT
3F23 8D    41       00290           BSR     DEC2     * ELSE BACK UP 2 MEM. LOCNS.
3F25 81    3A       00300           CMPA    #$3A     * IS IT 10 SECONDS YET?
3F27 2D    20       00310           BLT     OUT      * IF NOT 10 SECONDS, OUT
3F29 8D    34       00320           BSR     DEC1     * BACK UP 1 MEM. LOCATION
3F2B 81    36       00330           CMPA    #$36     * IS IT 60 SECONDS YET?
3F2D 2D    1A       00340           BLT     OUT      * IF NOT 60 SECONDS, OUT
3F2F 8D    35       00350           BSR     DEC2     * ELSE BACK UP 2 MEM. LOCNS.
3F31 81    3A       00360           CMPA    #$3A     * IS IT 10 MINUTES YET?
3F33 2D    14       00370           BLT     OUT      * IF NOT 10 MINUTES, OUT
3F35 8D    28       00380           BSR     DEC1     * ELSE BACK UP 1 MEM. LOCATION
3F37 81    36       00390           CMPA    #$36     * IS IT 60 MINUTES YET?
3F39 2D    0E       00400           BLT     OUT      * IF NOT 60 MINUTES, OUT
3F3B 8D    29       00410           BSR     DEC2     * ELSE BACK UP 2 MEM. LOCNS.
3F3D 81    3A       00420           CMPA    #$3A     * IS IT 10 HOURS YET?
3F3F 2D    08       00430           BLT     OUT      * IF NOT 10 HOURS, OUT
3F41 8D    1C       00440           BSR     DEC1     * ELSE BACK UP 1 MEM. LOCATION
3F43 81    3A       00450           CMPA    #$3A     * IS IT 100 HOURS YET?
3F45 2D    02       00460           BLT     OUT      * IF NOT 100 HOURS, OUT
3F47 E7    84       00470           STB     ,X       * PLACE $30 (ASCII ZERO)
                    00480 *
3F49 108E  0416     00490 OUT       LDY     #$0416   * POINT TO RIGHT SCREEN
3F4D 8E    3F6D     00500           LDX     #IMAGE   * POINT X TO CLOCK IMAGE
3F50 C6    0A       00510           LDB     #$0A     * COUNT 10 SCREEN POSITIONS
3F52 A6    80       00520 LOOP      LDA     ,X+      * GET CHARACTER FROM CLOCK
3F54 A7    A0       00530           STA     ,Y+      * AND PLACE IT ON THE SCREEN
3F56 5A             00540           DECB             * DONE WITH IMAGE YET?
3F57 26    F9       00550           BNE     LOOP     * IF NOT, THEN GET NEXT CHAR.
                    00560           *
3F59 B6    FF02     00570           LDA     $FF02    * CLEAR VERT. SYNC LATCH
3F5C 7E    894C     00580           JMP     $894C    * AND TO BASIC TO DO RTI
                    00590 *
3F5F E7    84       00600 DEC1      STB     ,X       * PLACE $30 (ASCII ZERO)
3F61 6C    82       00610           INC     ,-X      * BACK UP ONE MEM. LOCATION
3F63 A6    84       00620           LDA     ,X       * GET VALUE FROM IMAGE
3F65 39             00630           RTS              * BACK TO MAIN PROGRAM
                    00640 *
3F66 E7    84       00650 DEC2      STB     ,X       * PLACE $30 (ASCII ZERO)
3F68 6C    83       00660           INC     ,--X     * BACK UP TWO MEM. LOCATIONS
3F6A A6    84       00670           LDA     ,X       * GET VALUE FROM IMAGE
3F6C 39             00680           RTS              * BACK TO MAIN PROGRAM
                    00690 *
3F6D       30       00700 IMAGE     FCC     /00:00:00.00/
           30
           3A
           30
           30
           3A
           30
```

```
            30
            2E
            30
            30
                        00710 *
            3F00        00720           END         INTOFF
00000 TOTAL ERRORS
DEC1      3F5F
DEC2      3F66
IMAGE     3F6D
INTOFF    3F00
LOOP      3F52
OUT       3F49
START     3F10
```

The opening is the 16-byte setup routine, turning off interrupts, redirecting the interrupt vector to my interrupt service routine, passing through the 60-per-second interrupt, turning on interrupts, and returning to BASIC.

The service routine itself is a strung-out series of increments and comparisons. The sixtieth-of-a-second clock image in memory is incremented and tested for **$36** (the ASCII value for the character 6). If it's less than six, out it goes; otherwise, it begins a down-the-line test. Notice in the DEC1 and DEC2 routines the use of an indexed pre-decrement command; right along you've been seeing the post-increment commands such as **LDA ,X+**, but this is the first time the pre-decrement has turned up. Since this routine is bumping backwards in memory (from sixtieths of a second up to tens of hours), a decrement is needed.

Check the sequence in the subroutine:

```
STB     ,X
INC     ,-X
LDA     ,X
```

The value in B (an ASCII zero) is stored in memory pointed to by X. The X pointer is decremented and then its contents are incremented. Two things of complementary character are here — the pointer is first decremented, then its contents are incremented. And finally, the A accumulator is loaded with the contents of the memory location now pointed to by X.

After all the increments, tests and updates are complete, the memory image of the time is transferred to the screen. In line 490, Y points to location **$0416** on the screen, and X points to the updated clock. A short loop transfers the information.

Finally, the command **LDA $FF02** resets the latched interrupt from the PIA. In your reading of the MC6821 data booklet, page 8, this was mentioned. I'll read that paragraph. "The four interrupt flag bits are set by active transitions of signals on the four interrupt and peripheral control lines when those lines are programmed to be inputs. These bits cannot be set directly from the MPU data bus and are reset indirectly by a read peripheral data operation on the appropriate section." In other words, flags go up inside the PIA when an interrupt takes place; by reading from the PIA, the flag goes down. **LDA $FF02** reads from the PIA and turns off the interrupt flag.

*** What happened to the RTI needed at the end of an interrupt? Where is it?**

The RTI is found in the BASIC ROM after it finishes with the cursor flash and timer update.

*** When using the MC6821 PIA to cause the interrupt, what is also necessary at the end of the service routine?**

The PIA's interrupt latch must be reset.

*** What would happen if the latch were not reset?**

No further interrupts would pass through the PIA to the processor.

*** What two addresses are used by the PIA that handles the IRQ?**

Addresses $FF02 and $FF03.

*** What command resets the interrupt latch?**

Any command that reads from port address $FF02, such as LDA $FF02.

*** What does IRQ mean?**

IRQ means interrupt request.

*** What does PIA mean?**

PIA means Peripheral Interface Adapter.

*** What do FS and HS mean?**

FS means Field Sync and HS means Horizontal Sync.

* What does VDG mean?

VDG means Video Display Generator.

* What does A/IM/AO mean?

Assemble into memory at the absolute origin specified in the source listing.

* Three things happen when an interrupt occurs. What are they?

The microprocessor finishes its current instruction, saves important information, and follows programming instructions in response to the interrupt.

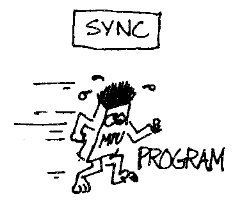
The last instruction (**JMP $894C**) might not make sense to you. You probably expected a return from interrupt instruction (RTI). Let me explain. You'll recall that the interrupt vector for IRQ goes to address **$0110** in low RAM for its instruction. At that location is found the instruction **JMP $894C**. In order for this time display program to work properly with BASIC, it must chain itself to BASIC's vectors. That vector and the subsequent JMP $894C controls the cursor flashing, among other things. So it's go to be there. By replacing **JMP $894C** with the **JMP $3F10** that gets the time display routine going, the program has intercepted a vital part of BASIC's operating system. To keep the link from IRQ vector **$0110** to ROM location **$894C**, this program intercepts **$0110**, patches itself in place, and finishes by jumping to **$894C**. The chain is complete; the time is displayed and BASIC has its cursor. BASIC finishes by executing the return from interrupt (RTI).

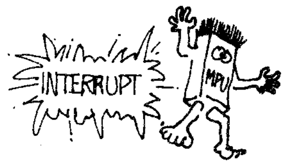
I think it seems simple enough. Give it a try. Assemble this program in memory at the correct origin. Type A/IM/AO and hit enter. The program will assemble into memory. When it's finished and the cursor has returned, type and enter Q. You will quit EDTASM+ and return to BASIC. Protect memory now; this program resides from **$3F00** to **$3F77**, so protect from **$3F00** on up. Type and enter CLEAR200,&H3F00. That's CLEAR200,&H3F00.

Ready? Type and enter EXEC&H3F00.
There's the clock, ticking away in the upper-right-hand corner of the screen. You can enter, edit and list and run BASIC programs. Try a few short programs, and see how it looks to have the clock in the corner.

When you're done with that, try one more test. Create a BASIC program and CSAVE it to tape. I don't care what kind of program it is, and you don't really even need to have the tape running. I just want you to CSAVE something, and keep an eye on the screen. Before the next session, figure out what you see and why it must happen that way. Have fun.

23.





In this lesson, I'm going to turn to video display synchronization achieved with interrupts. But please keep something in mind as you review these past two lessons. This may be the Color Computer you're using, but it's the 6809 processor you're learning to program. Although every 6809 processor is made with these interrupt capabilities and signals, those interrupt signals might be wired in a completely different way on another type of computer. alternative internal wiring might also mean that the vectors in memory would be changed and that the timing of the interrupts would be more or less frequent. Chances are — except for the method used to turn interrupts on and off, which is a function of the 6809's condition code register — everything would be handled differently. Since you're learning the 6809 on the Color Computer, I know these programs will work for you. But if you change computer systems, you'll have to apply the principles but not necessarily the actualities of these interrupt sessions.

That said, it's on to video synchronization. There are only two unique instructions left to talk about on the 6809. These are **SYNC** and **CWAI**.

SYNC and **CWAI** are similar instructions: both cause the 6809 to stop processing — that is, cease to follow program instructions — and wait for an interrupt to occur. **SYNC** (for synchronize) simply turns the processing off, to the point of making it electronically invisible to the rest of the computer components. **SYNC** is especially useful when connecting multiple computers to the same memory; you can't do that with the Color Computer because all the necessary connections aren't there, but **SYNC** makes it possible for some other 6809 computers to work as multiple processor systems.

Like **SYNC**, **CWAI** also causes the processor to stop, but not immediately. **CWAI** (meaning clear condition code bits immediate and wait for interrupt) first places all the registers on the stack and then sets the E flag; the E flag tells the processor that the entire machine state has been

Dealing with interrupts is no more complicated than any assembly programming. The only hitches are getting to the interrupt service routine and back from it without any errors, and, where timing is absolutely critical, getting it over with before it's time for more interrupts. The 60-per-second interrupt in the last lesson's clock program was leisure time at its most relaxing compared with the program in this lesson!

* What is the process of acting on an interrupt called?

Servicing the interrupt.

* How does the program counter find where to go to service the interrupt?

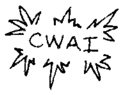
From a vector, or address, in the last 16 bytes of memory.

* What purpose does the IRQ serve on the Color Computer?

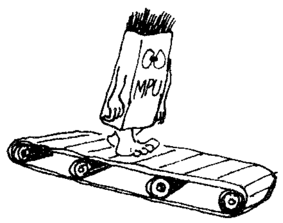
It is connected to horizontal and vertical synchronization signals from the video display generator.

saved on the stack. The **CWAI** instruction also keeps the processor active with respect to the outside world; there is no "invisibility" with **CWAI**.

The effective similarity between **SYNC** and **CWAI**, then, is that they both stop the processor's operations and wait for an interrupt to occur. The effective difference is that **SYNC** just stops the operation, whereas **CWAI** also presets the condition codes and saves all the registers.

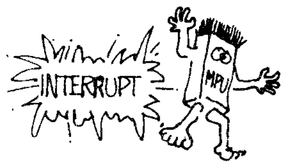


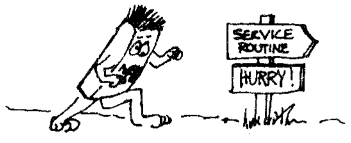
I'll be using **SYNC** for these demonstrations. You might be wondering why stopping the processing with SYNC would be preferable to the straightforward use of an interrupt as I showed you in the last session. With **SYNC**, you can complete all the programming work you need for a change of video contents, then enter **SYNC** mode and wait for further instruction. The amount of time you've got for the program and the timing of the interrupts becomes more important as you write the program, but lets the program work more effectively.



Let me turn back to the peripheral interface adaptors, the PIAs, and their control registers. Addresses **$FF01** and **$FF03** have the important information:

Bit	Function
Ø	Ø = disable interrupt, 1 = enable interrupt request to processor.
1	Ø = falling transition, 1 = rising transition sets IRQA/B1 output.
2	Ø = data direction register, 1 = control register; established at power-up.
3	One of a pair of binary select signals for control of the analog multiplexer (see technical manual for details).
4,5	Establishes CA2/CB2 as output controlled by bit 3 -- always 1 on Color Computer.
6	Interrupt flag when CA2/CB2 is an input; not used on the Color Computer.
7	Interrupt flag from CA1/CB1 -- vertical or horizontal TV synchronization.





Now that you know, what do you do with it? I've got to get technical on you. This is one of those times when hardware meets software, and in order to program what you need, you've got to understand what's going on.

The television screen display isn't a fixed image of some kind, but rather the result of a single, constantly moving electron beam aimed from the back and sweeping across

Port bits

* What are the terms for vertical and horizontal synchronization with respect to the Color Computer?

Field sync (FS) and horizontal sync (HS).

* How often does the field sync (FS) signal occur?

6Ø times per second.

* How often does the horizontal sync (HS) signal occur?

15,72Ø times per second.

* What port address determines which interrupt is fed through to the 68Ø9 processor?

Port address $FFØ3.

* What condition code bit masks or enables the IRQ?

Bit 4 masks or enables the IRQ.

* What instruction masks the IRQ?

ORCC #$1Ø masks the IRQ.

* What instruction enables the IRQ?

ANDCC #$EF enables the IRQ.

* What instruction returns to the program in progress after an interrupt has been serviced?

Return from interrupt, RTI.

* What is the IRQ vector found?

The IRQ vector is found at $FFF8 and $FFF9.

* On the Color Computer, where does the IRQ vector point?

The IRQ vector points to address $Ø1ØC.

the front of a glass tube. As the beam sweeps by, rare-earth elements known as phosphors are excited by the beam and glow blue, green or red.

By depending on the mixing of the primary colors of blue, green or red (technically called cyan, green and magenta), and also on our eyes' persistence — that is, the ability to retain an image for a small fraction of a second — a complete, multi-colored picture seems to be formed.

If you look at the front of the picture tube with a magnifying glass, you can see the separate colors. By moving your hand quickly in front of the screen, you can see the image "break up" as your hand's outline is strobed by the changing screen image produced by that moving electron beam.

There's only one electron beam, and it's moving fast. It sweeps across the screen, changing color and brightness as it goes, then turns off, sweeps back, turns on, and draws the next line. It draws 262 lines altogether, all the while keeping those lines separated by moving slowly down the screen; one screen full of lines is called a "field". At television speed, "slowly" is only a comparative term, because the beam goes from top to bottom of the screen 60 times each second. On the Color Computer, that's 15,720 lines drawn every second.

What keeps all this happening at the correct time and keeps the beam at the correct place on the screen is known as synchronization. The electrical signal that tells the beam when to start each line across is called horizontal synchronization, or horizontal sync. The signal that tells the beam when to get to the top of the screen and start the next field is called vertical synchronization, or vertical sync. Although it would be simpler to call these horizontal sync and vertical sync, I'm not going to do that. I want to avoid confusing these sync signals with the 6809 processor command **SYNC**.

The MC6847 video display generator, the VDG, creates horizontal and vertical synchronization, and also another signal called field synchronization. Field synchronization is the time between the end of the active display (the very bottom right of the green block that makes up the display screen) and the top of the screen (25 lines before the start of the green block).

For a complete look at all this, open your MC6847 video display generator data booklet, and turn to page 11. On page 11 of the MC6847 data booklet, you can see the relationship between the blank areas and the active display area. Take a few minutes to examine Figures 13 and 14.

---

* Where is $010C in the Color Computer memory map?

In RAM, on page $01.

* When using the MC6821 PIA to cause the interrupt, what is also necessary at the end of the service routine?

The PIA's interrupt latch must be reset.

* What two addresses are used by the PIA that handles the IRQ?

Addresses $FF02 and $FF03.

* What command resets the interrupt latch?

Any command that reads from port address $FF02, such as LDA $FF02.

* What actions does the SYNC instruction cause?

It causes the processor to stop processing instructions and wait for an interrupt to occur.

* What actions does the CWAI instruction cause?

It ANDs the condition code bits with a value, places all the registers on the stack, sets the E flag, stops further processing and waits for an interrupt.

* How are the software actions of SYNC and CWAI alike?

Both stop further processing and wait for an interrupt.

* How are the software actions SYNC and CWAI different?

CWAI (Clear and Wait for Interrupt) performs logical and stack operations, whereas SYNC (Synchronize with Interrupt) does not.

* How are the hardware actions of SYNC and CWAI different?

CWAI keeps the processor active with respect to the outside world (to the other circuits); SYNC makes it electronically invisible (called a tri-state condition).

* How many horizontal lines does the electron beam draw on the video display screen?

262 horizontal lines are drawn on the screen.

* What is one complete group of 262 lines called?

One group of 262 lines comprises a field.

* What is the "green block" in the center of the video screen?

The "green block" is the active display area.

* How many horizontal electron beam lines comprise the active display area?

192 horizontal lines make up the active display area.

* How many fields of 262 lines are drawn each second?

60 fields are drawn each second.

* How many lines are drawn each second?

262 lines times 60 fields, or 15,720 lines are drawn each second.

* What controls the horizontal lines and vertical fields?

The Video Display Generator, the VDG.

Turn to page 11 in the MC6847 video display generator (VDG) data booklet and examine Figures 13 and 14, which present the active display area of the Color Computer. Familiarize yourself with the number of horizontal lines and their arrangement. Return to the tape when you have completed the reading.

Don't bite your lip; this is all going to fit together very shortly. When you know about field synchronization and horizontal synchronization, you know two important things. The first thing you know is the time when your processor is free to make its calculations, scan the keyboard, and so forth. That time falls between the end of the active display area and the top of the screen. And that time starts when field synchronization (FS) goes from one to zero, and that time ends when FS goes from zero to one. The 6809 processor can find out when FS changes.

The second thing you know is when the beam starts at the left of the screen and when it ends at the right. It starts when horizontal synchronization (HS) goes from one to zero and ends when HS goes from zero to one. The time when HS is off the screen very short, however (about one CPU clock cycle), so in effect, the important time is the start of the HS period, when HS goes from one to zero. The 6809 processor can find out when HS changes.

So here's an outline of the features as they relate to software.

1. FS goes from high to low. You're out of the screen and free to calculate and perform other operations.

2. FS goes from low to high. You've got to start paying attention to screen lines.

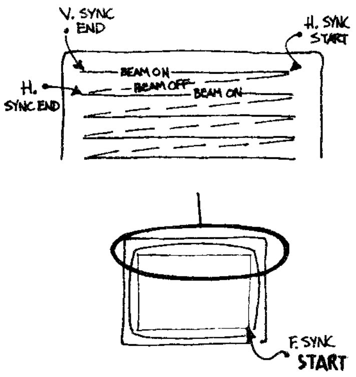
3. HS goes from high to low. The screen has started.

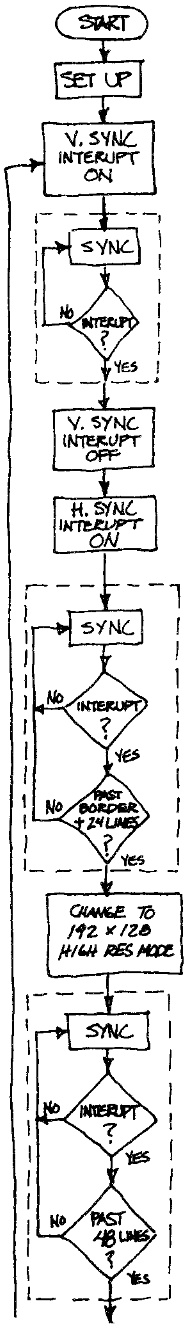
4. Count 38 HS pulses and you're in the display area.

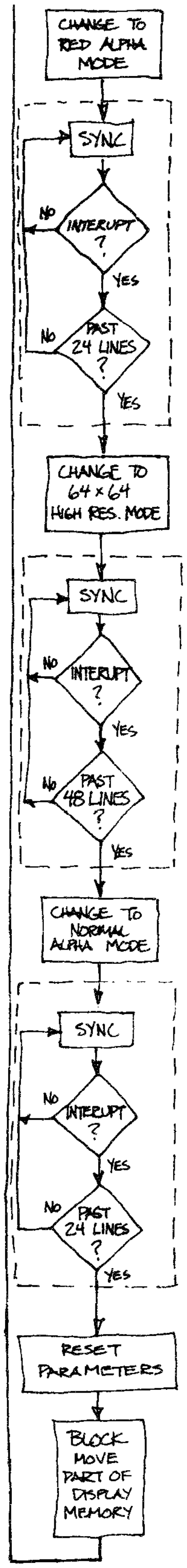
5. 192 HS pulses make up one active screen.

6. Repeat it all 60 times and you've got one full second of programming.

Now it's getting closer. Feed through the vertical or field synchronization to the processor's interrupt, and execute the **SYNC** command. When it occurs, execute a vertical synchronization service routine. That routine should turn off that feed-through and turn on the horizontal synchronization feed-through. Create another interrupt service routine for the horizontal synchronization. Begin







counting until you reach the top of the active display area. Then you can change the display and count screen lines in short programming bursts, ending each with **SYNC**. When you have counted 192 lines, the screen display area is completed. You can turn off the horizontal feed-through, turn back on the vertical synchronization feed-through, return to the main loop for your calculations and more sophisticated programming. When that's done, you can execute the **SYNC** command and wait for the process to start all over.

A practical example is the only way of understanding what this is good for and how to use it. Before that, though, please review this lesson so far, reread the control register information in the MC6821 peripheral interface adapator data booklet, and re-examine the screen outline on page 11 in the MC6847 data booklet.

Review this lesson. After reviewing, read the control register information in the MC6821 data booklet, pages 7 and 8. Also continue to become familiar with the screen outlines on page 11 of the MC6847 VDG data booklet. Return to the tape when you have completed the reading.

The practical example I've got is about as impractical as they come in some respects. It shows a bunch of random colors and shapes on the screen, together with alphanumerics. There are standard letters and characters (black on green), high resolution color graphics, more characters (black on red), medium resolution color graphics, and more characters. The trick is that all of them are displayed on the same screen at the same time.

Getting a mix of high-resolution graphics and standard alphanumerics on the screen at the same time is a simple function of synchronizing and counting. If you synchronize to the vertical synchronization pulse, you know where the screen starts. If you synchronize to the horizontal synchronization pulse, you know where each of the 192 screen lines is. If you are familiar with your graphics modes, then you know what character is where on what line.

All that's left is the implementation. My example presents two rows of alphanumeric characters, a 192 by 48 block of high resolution color graphics, two more rows of alpha characters (but in red instead of green), a 64 by 16 block of medium resolution color graphics, and three rows of alpha characters. I haven't filled memory with anything in particular, so it's just random junk. But the junk'll be moving. Load the source code. I'll take you through it, and do some explaining.

* What is FS (Field Synchronization) on the VDG?

The time between the end of the active display area and the top of the screen.

* When does FS go from high to low (one to zero)?

When the electron beam leaves the active display area.

* When does FS go from low to high (zero to one)?

When the electron beam reaches the top of the screen.

* When does HS go from one to zero?

When the electron beam begins drawing a line on the screen.

* When does HS go from zero to one?

When the electon beam finishes drawing a line on the screen.

* According to the MC6847 data booklet, how many HS pulses occur before the "green block" -- the active display area -- begins?

38 HS pulses occur before the active display area begins.

* How many HS pulses occur during active display (within the "green block")?

192 HS pulses occur within the active display.

* According to the MC6847 data booklet, how many HS pulses occur after the active display area ends?

32 HS pulses occur after the active display area ends.

Program #35, an EDTASM+ program. Insert the EDTASM+ cartridge, and turn on the power to your computer. When the cursor appears, typle 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.

```
          ØØØC     ØØ1ØØ ROW     EQU     12
          ØØ23     ØØ11Ø BORDER  EQU     35
          Ø42Ø     ØØ12Ø HIRES   EQU     $Ø42Ø
          Ø8ØØ     ØØ13Ø VIDTOP  EQU     $Ø8ØØ
          FFØØ     ØØ14Ø CLEARH  EQU     $FFØØ
          FFØ2     ØØ15Ø CLEARV  EQU     $FFØ2
          FFØ1     ØØ16Ø HSPORT  EQU     $FFØ1
          FFØ3     ØØ17Ø VSPORT  EQU     $FFØ3
          Ø1ØD     ØØ18Ø VECTOR  EQU     $Ø1ØD
          FFCØ     ØØ19Ø VIDCLØ  EQU     $FFCØ
          FFC1     ØØ2ØØ VIDSTØ  EQU     $FFC1
          FFC2     ØØ21Ø VIDCL1  EQU     $FFC2
          FFC3     ØØ22Ø VIDST1  EQU     $FFC3
          FFC4     ØØ23Ø VIDCL2  EQU     $FFC4
          FFC5     ØØ24Ø VIDST2  EQU     $FFC5
          FF22     ØØ25Ø VIDPRT  EQU     $FF22
                   ØØ26Ø *
3FØØ               ØØ27Ø         ORG     $3FØØ
                   ØØ28Ø *
                   ØØ29Ø * GET & SAVE BASIC VECTOR
                   ØØ3ØØ * PLACE THIS VECTOR
3FØØ 1A   5Ø       ØØ31Ø BEGIN   ORCC    #$5Ø
3FØ2 BE   Ø1ØD     ØØ32Ø         LDX     VECTOR
3FØ5 BF   3FC8     ØØ33Ø         STX     STOREV
3FØ8 8E   3F7D     ØØ34Ø         LDX     #INTER
3FØB BF   Ø1ØD     ØØ35Ø         STX     VECTOR
                   ØØ36Ø *
                   ØØ37Ø * INTERRUPTS OFF.
                   ØØ38Ø * HORIZONTAL SYNC OFF.
                   ØØ39Ø * VERTICAL SYNC ON.
                   ØØ4ØØ * SELECT ALPHA MODE.
                   ØØ41Ø * INTERRUPTS ON.
                   ØØ42Ø * WAIT FOR VERTICAL SYNC.
3FØE 86   36       ØØ43Ø STAR    LDA     #$36
3F1Ø B7   FFØ1     ØØ44Ø         STA     HSPORT
3F13 4C            ØØ45Ø         INCA
3F14 B7   FFØ3     ØØ46Ø         STA     VSPORT
3F17 B7   FFC4     ØØ47Ø         STA     VIDCL2
3F1A B7   FFC2     ØØ48Ø         STA     VIDCL1
3F1D B7   FFCØ     ØØ49Ø         STA     VIDCLØ
3F2Ø 8E   3F7F     ØØ5ØØ         LDX     #SCREEN
3F23 1C   EF       ØØ51Ø         ANDCC   #$EF
3F25 13            ØØ52Ø         SYNC
                   ØØ53Ø *
                   ØØ54Ø * WAIT FOR HORIZ. SYNC.
                   ØØ55Ø * COUNT BORDR + 24 LINES.
                   ØØ56Ø * CHANGE TO 128X192 COLOR
3F26 8E   3F94     ØØ57Ø         LDX     #LINE
3F29 C6   3B       ØØ58Ø         LDB     #BORDER+2*ROW
3F2B 1C   EF       ØØ59Ø         ANDCC   #$EF
3F2D 13            ØØ6ØØ LOOP1   SYNC
3F2E 5A            ØØ61Ø         DECB
3F2F 26   FC       ØØ62Ø         BNE     LOOP1
3F31 86   EF       ØØ63Ø         LDA     #$EF
3F33 B7   FF22     ØØ64Ø         STA     VIDPRT
3F36 B7   FFC5     ØØ65Ø         STA     VIDST2
3F39 B7   FFC3     ØØ66Ø         STA     VIDST1
                   ØØ67Ø *
                   ØØ68Ø * WAIT FOR HORIZ. SYNC.
                   ØØ69Ø * COUNT 48 LINES.
                   ØØ7ØØ * CHANGE TO ALPHA MODE.
3F3C C6   3Ø       ØØ71Ø         LDB     #4*ROW
3F3E 13            ØØ72Ø LOOP2   SYNC
3F3F 5A            ØØ73Ø         DECB
3F4Ø 26   FC       ØØ74Ø         BNE     LOOP2
3F42 86   ØF       ØØ75Ø         LDA     #$ØF
3F44 B7   FF22     ØØ76Ø         STA     VIDPRT
3F47 B7   FFC4     ØØ77Ø         STA     VIDCL2
3F4A B7   FFC2     ØØ78Ø         STA     VIDCL1
```

```
                   *
                   * WAIT FOR HORIZ. SYNC.
                   * COUNT 24 LINES.
                   * CHANGE TO 64X64 COLOR.
3F4D C6   18                LDB     #2*ROW
3F4F 13            LOOP3    SYNC
3F50 5A                     DECB
3F51 26   FC                BNE     LOOP3
3F53 86   8F                LDA     #$8F
3F55 B7   FF22              STA     VIDPRT
3F58 B7   FFC4              STA     VIDCL2
3F5B B7   FFC2              STA     VIDCL1
3F5E B7   FFC1              STA     VIDST0
                   *
                   * WAIT FOR HORIZ. SYNC.
                   * COUNT 48 LINES.
                   * CHANGE TO ALPHA MODE.
3F61 C6   30                LDB     #4*ROW
3F63 13            LOOP4    SYNC
3F64 5A                     DECB
3F65 26   FC                BNE     LOOP4
3F67 86   07                LDA     #$07
3F69 B7   FF22              STA     VIDPRT
3F6C B7   FFC0              STA     VIDCL0
                   *
                   * WAIT FOR HORIZ. SYNC.
                   * COUNT 48 LINES.
3F6F C6   30                LDB     #4*ROW
3F71 13            LOOP5    SYNC
3F72 5A                     DECB
3F73 26   FC                BNE     LOOP5
                   *
                   * INTERRUPTS OFF.
                   * DO BYTE FINAGLE STUFF.
                   * START IT ALL AGAIN.
3F75 1A   50       STOP     ORCC    #$50
3F77 BD   3F98              JSR     INCREM
3F7A 7E   3F0E              JMP     STAR
                   *
                   * SUBROUTINES FOLLOW.
                   * JUMP OFFSET INDEXED.
                   * X POINTS TO ROUTINE.
3F7D 6E   84       INTER    JMP     ,X
                   *
                   * CLEAR FIELD SYNC LATCH.
                   * SELECT ALPHA MODE.
                   * TURN VERTICAL SYNC OFF.
                   * TURN HORIZ. SYNC ON.
                   * CLEAR HOR. SYNC LATCH.
                   * BACK TO MAIN PROGRAM.
3F7F B6   FF02     SCREEN   LDA     CLEARV
3F82 86   07                LDA     #$07
3F84 B7   FF22              STA     VIDPRT
3F87 86   36                LDA     #$36
3F89 B7   FF03              STA     VSPORT
3F8C 4C                     INCA
3F8D B7   FF01              STA     HSPORT
3F90 B6   FF00              LDA     CLEARH
3F93 3B                     RTI
                   *
                   * CLEAR HOR. SYNC LATCH.
                   * BACK TO MAIN PROGRAM.
3F94 B6   FF00     LINE     LDA     CLEARH
3F97 3B                     RTI
                   *
                   * BYTE-FINAGLE ROUTINE.
                   * BLOCK MOVES $44 BYTES
                   *  AT A TIME, CONTINUING
                   *  UNTIL #VIDTOP IS
                   *  REACHED.
                   * RESETS STORAGE AND
                   *  START LOCATIONS,
                   *  INCREMENTS Y TO NEXT
                   *  BLOCK MOVE POINT.
3F98 BE   3FC2     INCREM   LDX     XSTORE
3F9B 10BE 3FC4              LDY     YSTORE
3F9F C6   44                LDB     #$44
3FA1 A6   A0       FILLUP   LDA     ,Y+
3FA3 A7   80                STA     ,X+
3FA5 5A                     DECB
3FA6 26   F9                BNE     FILLUP
3FA8 8C   0800              CMPX    #VIDTOP
3FAB 2D   0D                BLT     VIDMOR
```

```
3FAD 8E    0420      01620              LDX       #HIRES
3FB0 10BE  3FC6      01630              LDY       YHOLD
3FB4 31    21        01640              LEAY      1,Y
3FB6 10BF  3FC6      01650              STY       YHOLD
3FBA 10BF  3FC4      01660 VIDMOR       STY       YSTORE
3FBE BF    3FC2      01670              STX       XSTORE
3FC1 39              01680              RTS
                     01690 *
3FC2       0600      01700 XSTORE       FDB       $0600
3FC4       0000      01710 YSTORE       FDB       $0000
3FC6       0000      01720 YHOLD        FDB       $0000
3FC8                 01730 STOREV       RMB       02
                     01740 *
           3FCA      01750 ZZZZZZ       EQU       *
                     01760 *
           3F00      01770              END       BEGIN
00000 TOTAL ERRORS
BEGIN    3F00
BORDER   0023
CLEARH   FF00
CLEARV   FF02
FILLUP   3FA1
HIRES    0420
HSPORT   FF01
INCREM   3F98
INTER    3F7D
LINE     3F94
LOOP1    3F2D
LOOP2    3F3E
LOOP3    3F4F
LOOP4    3F63
LOOP5    3F71
ROW      000C
SCREEN   3F7F
STAR     3F0E
STOP     3F75
STOREV   3FC8
VECTOR   010D
VIDCL0   FFC0
VIDCL1   FFC2
VIDCL2   FFC4
VIDMOR   3FBA
VIDPRT   FF22
VIDST0   FFC1
VIDST1   FFC3
VIDST2   FFC5
VIDTOP   0800
VSPORT   FF03
XSTORE   3FC2
YHOLD    3FC6
YSTORE   3FC4
ZZZZZZ   3FCA
```

I've prepared this source listing to make full use of labels. Print the first screenful of lines; start with me at the top.

* What happens at the end of the active display area?

FS goes from high to low (one to zero).

Internally, the MC6847 video display generator counts to 12, which is the number of horizontal lines that make up a single alpha character position; so I label 12 as ROW. The upper border is defined by the 6847, so I label that BORDER. I'll be moving some display bytes around for effect; these moving display bytes will start at memory labeled HIRES and end at memory labeled VIDTOP.

* What PIA address handles the FS interrupt?

Port address $FF03.

* What PIA address resets the FS interrupt?

Reading port address $FF02.

* What PIA address handles the HS interrupt?

Port address $FF01.

The remaining are labels of key function addresses in upper memory; some you've seen before. As you have read in the MC6821 data booklet, the horizontal synchronization interrupt is cleared by reading **$FF00** and the vertical synchronization interrupt is cleared by reading **$FF02**; they are labeled CLEARH and CLEARV. The actual synchronization interrupts are fed through to the 6809's IRQ line by writing enabling information to ports **$FF01** and **$FF03**, here labeled HSPORT and VSPORT.

The IRQ vector from high memory finds its commands as the operand of the JMP at **$010C**, so **$010D** is labeled VECTOR.

There are six SAM addresses that control the video modes. The odd addresses clear the mode bit to zero, the even addresses set the bit to one. You know that. So mode bits 0, 1 and 2 are labeled VIDCL0 and VIDST0, VIDCL1 and VIDST1, VIDCL2 and VIDST2. Finally, the port address for the remaining video controls is found at **$FF22**; it's labeled VIDPRT.

Now display the last few lines of the program; begin at line 1500. P1500:*. Labels XSTORE, YSTORE and STOREV are two-byte groups set aside for temporary storage of video positions between vertical synchronization pulses.

So now you know the pack of labels I've got here. I've tried not to clutter this listing with lots of comments, so follow with me now. The first block of code turns off all interrupts which may have been enabled, and replaces the IRQ vector at **$010D** in RAM with my interrupt service routine. In the next block, horizontal synchronization interrupts are turned off, vertical synchronization interrupts are turned on, and alphanumeric video mode is selected.

The X register is loaded with a pointer to the vertical synchronization service routine, interrupts are enabled, and the processor enters **SYNC** mode. It now waits for the vertical synchronization pulse to force an interrupt. When the vertical synchronization interrupt occurs, the interrupt service routine is entered.

This routine finds the proper service by performing a zero-offset indexed jump based on the contents of the X register. Since X was pointed to the routine labeled SCREEN, this routine is performed. The SCREEN service routine clears the vertical synchronization latch, selects alpha mode, turns off the vertical synchronization feed-through, turns of the horizontal synchronization feed-through, clears the horizontal synchronization latch, and returns from the interrupt. It returns with everything set up for being interrupted by the horizontal synchronization pulse.

In other words, when the program starts, everything sets up and waits for the SCREEN service routine, which identifies the top of the screen and sets things up for the 262 horizontal interrupts.

The return from interrupt brings things back in the program to where the X register is pointed to the LINE service routine, the B register is set up to count through the screen border lines and 24 displayed lines. Remember I'm talking about electron beam lines here, not the usual lines of text. Interrupts are enabled, and the **SYNC** wait is on.

* What PIA address resets the HS interrupt?

Reading port address $FF00.

* What two items control the VDG modes?

Port $FF22 and the SAM control the various VDG modes.

* What is the general term for setting up the PIA or the VDG?

Configuring.

* After configuring the PIA for interrupts and the VDG for modes, the address of the interrupt service routine is put in place. How is that address accessed?

Through the IRQ vector in high memory.

* Where is the IRQ vector in high memory, and where does it point on the Color Computer?

The IRQ vector is at $FFF8 and $FFF9, and points to $010C in RAM.

* What addressing mode is this?

Indirect addressing.

* What does IRQ mean?

Interrupt request.

* How often does the horizontal interrupt HS occur?

15,720 times per second.

* According to the MC6847 data booklet, about how long is this?

It is approximately 63.5 microseconds.

The LINE service routine, arrived at through the zero-offset-indexed jump, merely clears the horizontal interrupt and returns. The B register is decremented, and if the selected number of electron beam lines is not yet counted through, **SYNC** is entered again. When the count is finished, the video mode is changed, the row counter recharged with a new value, and the **SYNC** state re-established.

There are five of these horizontal **SYNC** loops, each changing the video mode after a specific number of horizontal lines have been completed.

After the top border plus 192 horizontal lines, the active display area is complete and interrupts are disabled by the program. A short byte-move subroutine is called — you can put anything you like here — which bumps some display bytes around in the high resolution area. It lets you know something is happening. After the return from that byte-finagling subroutine, the process of vertical and horizontal synchronization starts again.

There are some important things to know. First of all, the horizontal interrupt occurs about ever 63.5 microseconds. That means you've got just about 57 clock cycles to perform your horizontal interrupt service routine. LOOP3 is the longest — I'll leave the calculations to you — but it makes it.

The other critical timing depends on the value of B (**$44** in my example) used to count bytes moved between vertical synchronization interrupts. In this case, **$44** is the highest number of moves I could fit between pulses.

Now keep in mind that this is a relatively crude demonstration of the possibilities of video manipulation. If you're interested in creating fast games or using powerful graphics capabilities, this method should give you as much power as any of the famous commercial game machines.

Now try it. Assemble this in memory by typing A/IM/AO/NL/NS. Assemble in memory at the absolute origin with no listing and no symbol table displayed. That's A/IM/AO/NL/NS. In a few seconds, the prompt and cursor will reture. Quit the editor/assembler by typing and entering Q. When the BASIC sign on message appears, you're ready for the demo. Type and enter EXEC&H3F00. That's where it all starts. EXEC&H3F00.

There's your mixed-mode display with moving parts. Study the listing and review this lesson; next time the trials and tribulations of debugging, hints and ideas, and a summary of what you have been learning. Till then.

* How many 6809 clock cycles is this on the Color Computer?

63.5 divided by 1.11746 is under 57 clock cycles.

* What actions does the SYNC instruction cause?

It causes the processor to stop processing instructions and wait for an interrupt to occur.

* What actions does the CWAI instruction cause?

It ANDs the conditon code bits with a value, places all the registers on the stack, sets the E flag, stops further processing and waits for an interrupt.

* How are the software actions of SYNC and CWAI alike?

Both stop further processing and wait for an interrupt.

* How are the software actions SYNC and CWAI different?

CWAI (Clear and Wait for Interrupt) performs logical and stack operations, whereas SYNC (Synchronize with Interrupt) does not.

* How many horizontal lines does the electron beam draw on the video display screen?

262 horizontal lines are drawn on the screen.

* What is one complete group of 262 lines called?

One group of 262 lines comprises a field.

* What is the "green block" in the center of the video screen?

The "green block" is the active display area.

24.

Welcome back. Up to this point, you've been walking an unfamiliar but well-lit path through assembly language. When this road ends, though, you'll be staring ahead into a kind of wilderness. If you know the natural signs, the footprints in the snow, how to feed and shelter yourself, then you'll survive to create your own paths. This course has been your outdoor survival training.

But that country isn't like this city, so you'll need not only the kit of tools — the editor/assembler, the data booklets, and the knowledge — but you'll also need something to cut a path in the underbrush so you can see through the woods and ahead to your destination.

That tool is a debugger. Sometimes called a machine-language monitor, the debugger is a program which displays memory contents, takes memory contents apart and translates them into mnemonics, does calculations and even steps through programs an instruction at a time.

The debugger is the "plus" in EDTASM+. This debugger is called ZBUG; get it ready now. Turn off your computer, insert the EDTASM+ cartridge, and turn the computer back on. The usual star prompt and flashing EDTASM cursor will appear. Type Z and press ENTER. The star prompt has changed to a crosshatch. You are in the ZBUG monitor. Now type E and press ENTER. Your star prompt returns and you are back in EDTASM.

Start with a program; you'll be doing the typing in this final lesson. The program is shown in the book. Enter it with the usual EDTASM insert-line mode (I), and assemble it to memory at the origin shown (A/IM/AO):

```
        ORG     $3FØØ
VIDEO   EQU     $Ø48Ø
COUNT   EQU     $ØØØØ
START   LDX     #VIDEO
        LDB     #COUNT
        LDA     #$FF
LOOP    INCA
```

As I come to the end of this course, it feels to me like a great novel should be ending, with its stereotypical sunsets, tears or flourishes. Rather than that, it's just some debugging and summaries. Maybe later for my Great American Novel; for now, you finish learning the 6809.

* What is another name for a debugging program?

A machine language monitor.

* What is the name of the machine language monitor that is part of EDTASM+?

ZBUG is the debugger.

* What is a breakpoint?

A stopping place in a machine language program inserted for debugging purposes.

* What is used as a breakpoint in ZBUG?

The software interrupt SWI.

```
        STA     ,X+
        DECB
        BNE     LOOP
        SWI
        END     START
```

When it's assembled, enter ZBUG by typing Z and pressing ENTER. Have a look at the assembly; your origin was **$3F00**, so type **3F00** followed by a slash. **3F00/** reveals **LDX #VIDEO**. When you do an in-memory assembly, ZBUG references your labels. Start pressing the down arrow. The program is being shown to you command by command, with each labeled as in the program.

Type BREAK. Again type **3F00**, but this time follow it with a comma. Type a few more commas, and continue tapping the comma quickly. Watch the screen carefully as you scroll through the commands. Reverse-video characters begin to appear and scroll up the screen. Eventually these change to normal characters, and finally to graphics characters. The program is executing step by step; the instructions . . .

```
LOOP  INCA
      STA  ,X+
      DECB
      BNE  LOOP
```

. . . are passing by and actually performing their functions. Keep pressing the comma. It takes four taps of the comma to produce one character, so you can see the repetitive nature of this program.

The character of the program is already familiar to you. It's nothing more than a memory fill starting at **$0480** and continuing for 256 loop repetitions.

Now watch it work at full speed; go to the start. Type G3F00 and tap ENTER. G3F00 ENTER. The screen gets blasted instantaneously with 256 characters, and ZBUG prints "8 BRK @ LOOP+6". There's the software interrupt command at work. Don't remember it? Tap E and return to EDTASM, and print the source code (P#:*) on the screen.

Right before the END statement is SWI, the software interrupt. This is what ZBUG uses for its breakpoints. More about breakpoints in a minute; back to ZBUG. Type and enter Z.

You've seen the labels in this listing. Now look at the actual hex values. Type H and hit ENTER. Now type **3F00/** and examine the display. Instead of symbolic notation using the labels (it used to read **LDX #VIDEO**) you now see **LDX #480**. Oh yes. The default notation in ZBUG is hexadecimal.

Type S and hit ENTER. Now 3F00/ once again reveals **LDX #VIDEO**. Another of these. Type B and ENTER. 3F00/ now shows **8E**, the hexadecimal value at memory

* What happens when a program encounters a software interrupt?

All the registers are saved and the program counter obtains the SWI vector from high memory.

* What is another name for "all the registers"?

The machine state.

* When an interrupt saves the machine state, what flag does it set?

The E, or entire state, flag.

* What is another name for a machine language monitor?

A debugger.

* The following questions summarize the concepts you should have learned from this course.

* How are machine language impulses represented?

By ones are zeros.

* What numbers system consists only of ones and zeros?

The binary system.

* What is the abbreviation for binary digit, and what is a group of four and a group of eight binary digits called?

A binary digit is a bit; four binary digits is a nybble (nibble); eight binary digits is a byte.

* What number system is used in programming for the convenient representation of binary numbers?

The hexadecimal number system.

location **$3F00**. Hitting the down arrow reveals labeled locations with hexadecimal values. And finally, one more to try. Type and enter A. Tap the down arrow and you see ASCII characters.

There's some preliminary work with ZBUG. Now you have reading to do. Chapters 2, 5 and 6 of the EDTASM+ manual have a complete description of the features of ZBUG. Read all the chapters and try the examples presented in the manual. Pay particular attention to the use of breakpoints — stopping places in the program — and the ways you can examine and change both memory and register contents. This powerful debugger will make finding those program glitches and endless loops lots easier. When you're done with the reading, come back for some suggestions on using ZBUG, and for a final summary of this course.

Using ZBUG is time consuming, but worthwhile. You might get tired of going through a long delay loop, though. In a case like this, use the register examination mode to change the loop value so it's almost done. Then you can continue execution, and the loop will complete.

One type of program that is almost impossible to debug in this manner is the interrupt-driven program. Enabling and disabling interrupts can be done, but when it comes to their actual execution, ZBUG will hang up, waiting for the interrupts which will never come. So for this kind of program, try your debugging by changing interrupts to subroutines in key places, saving and restoring the entire machine state (all the registers), and simulating the interrupts.

With the ability to use multiple number systems, to provide automatic calculations, to single-step your programs, and to display memory and registers, ZBUG is your most important tool — other than your own careful thinking and programming — in completing working, speedy and efficient assembly language programs. At the end of this lesson, use ZBUG to examine and execute each of the assembly language programs in this course.

In this course you have learned that assembly language is a representation of machine language, a carefully organized pattern of electronic impulses. These electronic impulses directly manipulate the actions of the microprocessor, and are therefore extremely fast and can be organized to perform any function which the computer's hardware permits. As patterns of electronic impulses, this kind of programming is distinctly different from high-level languages such as BASIC, languages which are in themselves constructed from large-scale patterns of machine commands.

Machine language consists of electronic impulses which are best expressed as one and zero conditions. The binary

* What does ASCII mean? What is it used for?

ASCII means American Standard Code for Information Interchange; ASCII is a binary pattern of control codes and characters used for computer communication and display.

* What is the overall organization of a processor called?

The architecture.

* Describe the architecture of the 6809 processor.

The 6809 consists of an Arithmetic Logic Unit and an Instruction Decoder; a program counter PC, accumulators A and B, index registers X and Y, stack pointers S and U, direct page register DP, and condition code register CC; A and B can be combined into accumulator D.

* What are processor commands called? What is the data used by the commands called?

Processor commands are operation codes, or opcodes; the data used by the commands are the operands.

* What are the verbal descriptions of processor commands called? What is a program listing containing these descriptions called?

Verbal descriptions are mnemonics, and a program listing containing mnemonics is called source code.

* What does an assembler do?

An assembler translates source code into object, or binary, code.

* What is an addressing mode?

An addressing mode is the way a machine language program gets the information it needs to complete an instruction.

* What are the 6809's addressing modes?

Inherent, register, immediate, extended, direct, indexed and relative.

* Describe inherent addressing.

The mode in which the opcode contains all the information the processor needs to complete the instruction.

* Describe register addressing.

The mode in which a postbyte describes the registers which are used to complete the instruction.

* Describe immediate addressing.

The mode in which the information to complete the instruction immediately follows the opcode.

* Describe extended addressing.

The mode in which the information is found at the address given after the opcode.

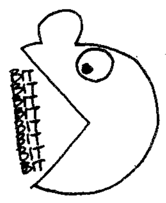
* Describe direct addressing.

The mode in which the information is found at the address calculated from the direct page register and the value following the opcode.

* Describe indexed addressing.

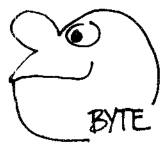
The mode in which the information is found at the address calculated from a fixed or variable offset and an index register.

system is a representation of ones and zeros, so the binary system counts in powers of two. The binary digits (the bits) are organized in groups of eight. These eight-bit groups are called bytes, and the byte is the word size for the 6809 processor.



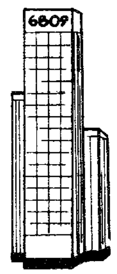
6809 words can stand for commands, data, characters, and can be used for counting and distances. When 6809 words are used as characters, those words are patterned in accordance with the American Standard Code for Information Interchange (ASCII).

All microprocessors have an overall organization known as architecture. The architecture of the 6809 encompasses its internal architecture, plus the ability to address 65,536 bytes of external memory. The internal architecture includes an arithmetic logic unit (ALU), an instruction decoder (ID), a 16-bit program counter (PC), two 8-bit accumulators (A and B), two 16-bit index registers (X and Y), two 16-bit stack points (S and U), an 8-bit direct page register (DP), and an 8-bit condition code register holding the flags (CC). The two 8-bit accumulators A and B can be combined to produce the 16-bit accumulator D.

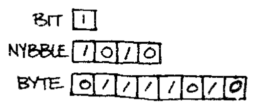


$ 5A = ASCII Z
$ 5A = command DEC B

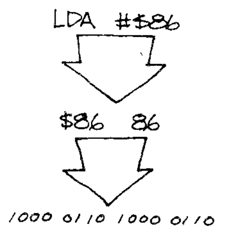
Commands to the 6809 processor are electronic impulses, represented by binary digits, and organized as bytes. The binary bytes are themselves thought of as two 4-bit groups, each of which is represented in hexadecimal notation. Hexadecimal notation, also called hex, counts from **0** through **F** and best expresses the character of 4-bit group. The 4-bit half of a byte is sometimes called a nybble.



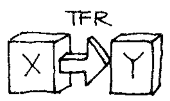
The hexadecimal notation represents the binary patterns, but the commands themselves are further abstracted into verbal descriptions. The verbal descriptions are called mnemonics, and the mnemonics are used for the construction of source code. Source code is a readable, quasi-verbal description of the processor actions that make up a complete program.

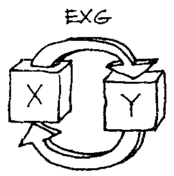


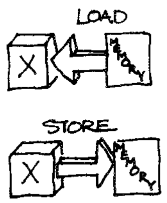
Source code is made up of mnemonics for binary machine commands, called opcodes, and the necessary information to complete the command, called the operand. Opcodes and operands — together with labels, origins, ends, byte descriptions, comments, and other information — make up the complete source listing. The source listing is entered and edited using an assembler, and translated from its source form to machine language by an assembler. The assembler takes the source code and produces from it the machine language, called object code.



The most common machine instructions move information inside the processor, move information from the processor to memory, and from the memory to the processor. These are transfers, exchanges, stores and loads. The processor manipulates this information through arithmetic and logical functions. The arithmetic includes addition, subtraction, multiplication, incrementing and decrementing. The logic includes AND, OR, Complement, Negation,







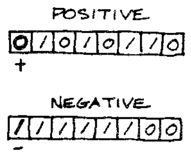
Exclusive-OR. Other processor manipulations of data include shifting or rotating bits left or right, testing for bits, comparison with other data, clearing to zero, and special functions for decimal addition and positive and negative arithmetic.

The processor obtains its information by providing the address of the data in external memory. The processor can determine the address it needs in a variety of simple and complex ways. These techniques are called addressing modes.

INHERENT
CLRA
REGISTER
TFR X,Y
IMMEDIATE
LDX #$0400
EXTENDED
LDY $1234
DIRECT
LDX <$33
INDEXED
LDB $41,X
INDIRECT
LDA [$19,Y]

Among the addressing modes in the 6809 processor are inherent, register, immediate, extended, direct and indexed. The inherent mode contains all the information the processor needs to complete an instruction. The register mode specifies information which informs the processor what internal registers to use. The immediate mode provides the processor with a value to use directly. The extended mode gives the processor an address at which it can find the information it needs. The direct mode combines the special direct page register with information to locate the data in memory. The indexed mode calculates a result from register information and fixed or variable offsets, and uses the results of that calculation to find the data in memory. Automatic incrementing or decrementing of certain registers can be specified in the indexed addressing mode. The relative mode instructs the processor to find information in relation to its current position in memory.

One of the features of the 6809 processor which speeds its operation and makes access of data simpler is the indexed indirect addressing mode. This mode applies to most of the previous indexing modes, and permits the processor to access information through a second level. The data is found at the address specified by the data found at an address determined by the processor from the instruction of the operand. This doesn't lend itself to a summary, so refer to lessons 15, 16 and 17 for more.



Great program structure is achieved using the indexed indirect addressing mode. By using an index relative to the current position of the program counter, complete program position independence within memory can be achieved.

The information actually received by the processor through all these adddressing modes is simply one byte at a time, but that byte can have many purposes. It can be a simple number; it can be positive or negative (that is, be signed); it can represent a character, or it can be part of a memory address.

The memory addresses themselves are (from the processor's viewpoint) identical. However, their arrangement within the Color Computer is somewhat different and quite specific. Because of the synchronous address multiplexer (the SAM), the memory addresses (known as the memory map) are organized for special

* Describe relative addressing.

The mode in which the information is found relative to the position of the program counter.

* What are the levels of addressing?

Non-indirect and indirect.

* What does SAM mean?

Synchronous Address Multiplexer.

* What is found from $0000 to $7FFF in the Color Computer memory map?

RAM (read-write memory).

* What is found from $8000 to $9FFF in the map?

The Extended Color BASIC ROM (read-only memory).

* What is found from $A000 to $BFFF in the map?

The Color BASIC ROM.

* What is found from $C000 to $FEFF in the map?

Cartridge ROM, when plugged in.

* What is found from $FF00 to $FFFF in the map?

Vectors and SAM registers, control, ports, video graphics display, processor speed, video addresses, and other functions.

* What is assembly?

The process of converting source code (mnemonics) into object (binary) code.

* What is disassembly?

The process of translating binary code into a source (mnemonic) listing.

Course summary

* What does VDG mean, and what is its purpose on the Color Computer?

VDG means Video Display Generator; it is used for alphanumeric, semigraphic, and high-resolution graphic and color display.

* What does Hz mean? What does it mean when it is said that the Color Computer has a .89 MHz clock?

Hz means Hertz, clock pulses per second; a .89 MHz clock means a master set of pulses occurring approximately 890,000 times per second.

* What is a position independent program? What addressing mode is essential to position independent programming?

A machine language program designed to run correctly no matter where it is located in memory is position independent. Relative addressing is necessary for position independent programming.

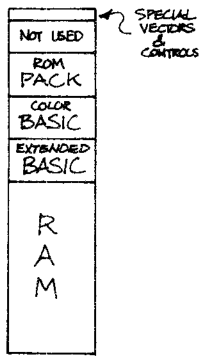
* What is an integer?

A number, positive or negative, which contains no fractional or decimal part.

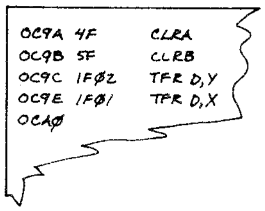
* What is a floating point number?

A number, positive or negative, which contains a fractional or decimal part.

purposes. From the start of memory to address **$7FFF** is reserved for read-write memory, or RAM; the next four blocks of memory (starting at **$8000**, **$A000**, **$C000** and **$E000**) are reserved for read-only memory, or ROM, and in the Color Computer are used for Extended BASIC, Color BASIC, and cartridge ROM. The last block is unused in the Color Computer. RAM may be substituted for ROM under certain conditions.



The last 256 bytes of memory are reserved for vectors and control, ports, video graphics display, processor speed, video addresses, and other functions. By writing information to the SAM, these functions can be turned on or off. Among the most important functions designed into the Color Computer are: control of the cassette and printer output; selection of 16 different low-, high-, and medium-resolution color graphics modes; RS-232 communications input and output; keyboard input; input from joysticks or other analog devices; control over the processor's clock speed; output of sound; determination of available memory and selection of the type of memory arrangement; control of and storage of vectors for interrupts.



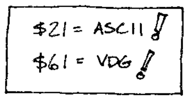
```
OC9A 4F     CLRA
OC9B 5F     CLRB
OC9C 1FØ2   TFR D,Y
OC9E 1FØ1   TFR D,X
OCAØ
```

Source code is normally assembled using an editor/assembler package, but hand assembly can be performed. For hand assembly, a list of opcodes and their respective hexadecimal equivalents is necessary. Also, it's essential to have a description of how each opcode works, the flags it affects, and how its operands are constructed and used.

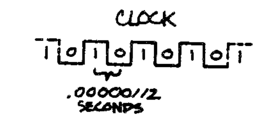
Assembly, whether by hand or using an assembler, is a two-pass process. During the first pass, the opcodes are assembled and put in place, and during the second pass the operands are created, calculated or otherwise determined directly from the operand information in the source listing, or from the labels used in the listing.

During the assembly process, the automatic assembler detects and reports errors. Hand assembly will reveal those opcodes or operands which are not permitted according to the information provided in the processor's data booklet.

Even correct source code can produce incorrect results, depending on the hardware configuration of the computer. In the case of the Color Computer, the most obvious conflict is with the standard ASCII codes and the video display generator, which uses a different arrangement of the four groups of 32 characters. These hardware conflicts are resolved by the programmer through debugging in combination with a careful reading of the software aspects of the hardware documentation.


$21 = ASCII !
$61 = VDG !

During hand assembly, the timing of instructions may be extremely critical. Especially during sound or communications processes, the timing of each instruction must be calculated. This timing is based on the computer's master clock frequency, which is specified as Hertz (Hz) or clock cycles per second. All the timing information is provided as part of the processor's data booklet. Some



timing is consistent with every occurrence of an instruction, other timing depends on the character of the operand.

The goal of position independent programming — that is, programs that will load and execute in any area of memory — can be achieved with the 6809 processor. Position independence is achieved using program-counter relative instructions (,PCR instructions), load-effective-address commands (LEA), long and short subroutine branches, and long and short program counter branches (simple, simple conditional, plus signed and unsigned conditional). By structuring the program around modular subroutines, both clarity and position independence can result.

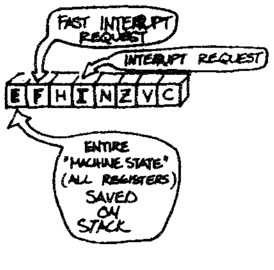
Among the less clear aspects of programming is the handling of floating-point numbers, that is those numbers consisting of both an integer and fractional part. The representation in the Color Computer is as a power of two exponent plus a four-byte mantissa. This achieves an overall accuracy of ten digits and an overall range of ten to the plus-or-minus 38th power.

10110111.1-
0101010...

10110111.1-
0101010...

Using these numbers, and using BASIC at all, requires an understanding of its handling of free memory, how it loads machine-language programs, and the accessing of machine language programs via EXEC and USR. BASIC's USR command permits direct transfer of numerical or text information to a machine-language subroutine. BASIC's VARPTR command permits access to BASIC variables for use by a machine-language subroutine, and also provides a unique method of packing a machine-language program into a BASIC string variable in a program line.

EXEC ADDRESS
USR (ARGUMENT)



The 6809 processor was created with interrupts in mind. Interrupts are hardware signals which cause the processor to set aside its current program and perform an interrupt service routine. Interrupts are use to provide accurate and program-independent timing and control functions. Hardware interrupts IRQ, FIRQ and NMI are used on the Color Computer; software interrupts SWI, SWI2 and SWI3 are used in ZBUG and in other kinds of program debugging, and for fast operating system subroutine calls on other kinds of computers.

Interrupts may be used for very fast timing, such as for synchronization with the video display. Video signals are used for interrupts on the Color Computer, and can be used as ordinary interrupts or in combination with the **SYNC** or **CWAI** commands for complete synchronization with the monitor picture.

The process of creating complete assembly language programs involves thinking the application through, creating a structure, writing modular subroutines, linking together the individual pieces, and debugging the whole.

Your Micro Language Lab course in Learning the 6809 is over, but your facility in programming has just begun. Now that you've reached this point, many earlier programs will

* What BASIC commands are used for accessing machine language programs? What does each mean?

EXEC, USR, DEFUSR, VARPTR, POKE and CLOADM. EXEC means execute a machine language program at the given entry point (starting address). USR means execute a machine language program, and transfer a variable from BASIC to it. DEFUSR defines a machine language program entry point (starting address). VARPTR means variable pointer, and is used to determine the position of a BASIC variable in memory. It can be used for packing machine language programs into BASIC string variables. POKE places a byte directly into memory. CLOADM loads binary information directly into memory.

* What are the 6809 interrupts?

Hardware interrupts NMI, FIRQ, IRQ and software interrupts SWI, SWI2, and SWI3.

* What happens when an interrupt occurs?

The processor completes its current instruction, saves important machine information, and services the interrupt.

* What commands stop processor operation and wait for an interrupt?

SYNC and CWAI.

* Your course in learning the 6809 is now complete. I welcome your reaction, especially to this programmed learning section. Please send your comments to me, Dennis Kitsz, Green Mountain Micro, Roxbury, Vermont 05669.

begin to make more sense. Please review this course lesson by lesson, continue to use the question-and-answer programmed text in the margins, and try each of the example programs. The ability to program the 6809 — and all its smart cousins — is now yours.

I'm your programming guide, Dennis Kitsz. Good bye.

When You See It In Memory, What Is It?

This chart is a cross-reference of all Color Computer codes from $00 to $FF. The codes are presented in binary, hexadecimal and decimal, followed by their ASCII equivalents. Both the 6809 procesor command mnemonic and BASIC "tokens" are given. The 10+ and 11+ commands are 6809 processor commands which use the value $10 or $11 as a prefix to other commands. For example, $10 21 is the opcode for long branch never (LBRN). Likewise, there are BASIC commands which take the prefix $FF. For example, token $82 is REM, whereas $FF 82 represents ABS.

BINARY	HEX	DECIMAL	ASCII	COMMAND	10+COMMAND	11+COMMAND	BASIC COMMAND	FF+COMMAND
0000 0000	00	0	NUL	NEG	—	—	—	SGN
0000 0001	01	1	SOH	—	—	—	—	INT
0000 0010	02	2	STX	—	—	—	—	ABS
0000 0011	03	3	ETX	COM	—	—	—	USR
0000 0100	04	4	EOT	LSR	—	—	—	RND
0000 0101	05	5	ENQ	—	—	—	—	SIN
0000 0110	06	6	ACK	ROR	—	—	—	PEEK
0000 0111	07	7	BEL	ASR	—	—	—	LEN
0000 1000	08	8	BS	ASL,LSL	—	—	—	STR$
0000 1001	09	9	HT	ROL	—	—	—	VAL
0000 1010	0A	10	LF	DEC	—	—	—	ASC
0000 1011	0B	11	VT	—	—	—	—	CHR$
0000 1100	0C	12	FF	INC	—	—	—	EOF
0000 1101	0D	13	CR	TST	—	—	—	JOYSTK
0000 1110	0E	14	SO	JMP	—	—	—	LEFT$
0000 1111	0F	15	SI	CLR	—	—	—	RIGHT$
0001 0000	10	16	DLE	(SEE 10+)	—	—	—	MID$
0001 0001	11	17	DC1	(SEE 11+)	—	—	—	POINT
0001 0010	12	18	DC2	NOP	—	—	—	INKEY$
0001 0011	13	19	DC3	SYNC	—	—	—	MEM
0001 0100	14	20	DC4	—	—	—	—	ATN
0001 0101	15	21	NAK	—	—	—	—	COS
0001 0110	16	22	SYN	LBRA	—	—	—	TAN
0001 0111	17	23	ETB	LBSR	—	—	—	EXP
0001 1000	18	24	CAN	—	—	—	—	FIX
0001 1001	19	25	EM	DAA	—	—	—	LOG
0001 1010	1A	26	SUB	ORCC	—	—	—	POS
0001 1011	1B	27	ESC	—	—	—	—	SQR
0001 1100	1C	28	FS	ANDCC	—	—	—	HEX$
0001 1101	1D	29	GS	SEX	—	—	—	VARPTR
0001 1110	1E	30	RS	EXG	—	—	—	INSTR
0001 1111	1F	31	US	TFR	—	—	—	TIMER

0010 0000	20	32	SPACE	BRA	—	—	—	PPOINT
0010 0001	21	33	!	BRN	LBRN	—	—	STRING$
0010 0010	22	34	"	BHI	LBHI	—	—	—
0010 0011	23	35	#	BLS	LBLS	—	—	—
0010 0100	24	36	$	BHS,BCC	LBHS,LBCC	—	—	—
0010 0101	25	37	%	BLO,BCS	LBLO,BLCS	—	—	—
0010 0110	26	38	&	BNE	LBNE	—	—	—
0010 0111	27	39	'	BEQ	LBEQ	—	—	—
0010 1000	28	40	(	BVC	LBVC	—	—	—
0010 1001	29	41	)	BVS	LBVS	—	—	—
0010 1010	2A	42	*	BPL	LBPL	—	—	—
0010 1011	2B	43	+	BMI	LBMI	—	—	—
0010 1100	2C	44	,	BGE	LBGE	—	—	—
0010 1101	2D	45	-	BLT	LBLT	—	—	—
0010 1110	2E	46	.	BGT	LBGT	—	—	—
0010 1111	2F	47	/	BLE	LBLE	—	—	—
0011 0000	30	48	0	LEAX	—	—	—	—
0011 0001	31	49	1	LEAY	—	—	—	—
0011 0010	32	50	2	LEAS	—	—	—	—
0011 0011	33	51	3	LEAU	—	—	—	—
0011 0100	34	52	4	PSHS	—	—	—	—
0011 0101	35	53	5	PULS	—	—	—	—
0011 0110	36	54	6	PSHU	—	—	—	—
0011 0111	37	55	7	PULU	—	—	—	—
0011 1000	38	56	8	—	—	—	—	—
0011 1001	39	57	9	RTS	—	—	—	—
0011 1010	3A	58	:	ABX	—	—	—	—
0011 1011	3B	59	;	RTI	—	—	—	—
0011 1100	3C	60	<	CWAI	—	—	—	—
0011 1101	3D	61	=	MUL	—	—	—	—
0011 1110	3E	62	>	—	—	—	—	—
0011 1111	3F	63	?	SWI	SWI2	SWI3	—	—
0100 0000	40	64	@	NEGA	—	—	—	—
0100 0001	41	65	A	—	—	—	—	—
0100 0010	42	66	B	—	—	—	—	—
0100 0011	43	67	C	COMA	—	—	—	—
0100 0100	44	68	D	LSRA	—	—	—	—
0100 0101	45	69	E	—	—	—	—	—
0100 0110	46	70	F	RORA	—	—	—	—
0100 0111	47	71	G	ASRA	—	—	—	—
0100 1000	48	72	H	ALSA,LSLA	—	—	—	—
0100 1001	49	73	I	ROLA	—	—	—	—
0100 1010	4A	74	J	DECA	—	—	—	—
0100 1011	4B	75	K	—	—	—	—	—
0100 1100	4C	76	L	INCA	—	—	—	—
0100 1101	4D	77	M	TSTA	—	—	—	—
0100 1110	4E	78	N	—	—	—	—	—
0100 1111	4F	79	O	CLRA	—	—	—	—

0101 0000	50	80	P	NEGB	—	—	—	—
0101 0001	51	81	Q	—	—	—	—	—
0101 0010	52	82	R	—	—	—	—	—
0101 0011	53	83	S	COMB	—	—	—	—
0101 0100	54	84	T	LSRB	—	—	—	—
0101 0101	55	85	U	—	—	—	—	—
0101 0110	56	86	V	RORB	—	—	—	—
0101 0111	57	87	W	ASRB	—	—	—	—
0101 1000	58	88	X	ASLB,LSLB	—	—	—	—
0101 1001	59	89	Y	ROLB	—	—	—	—
0101 1010	5A	90	Z	DECB	—	—	—	—
0101 1011	5B	91	L.BKT.	—	—	—	—	—
0101 1100	5C	92	SLANT	INCB	—	—	—	—
0101 1101	5D	93	R.BKT.	TSTB	—	—	—	—
0101 1110	5E	94	CARAT	—	—	—	—	—
0101 1111	5F	95	L.ARR.	CLRB	—	—	—	—
0110 0000	60	96	@	NEG	—	—	—	—
0110 0001	61	97	a	—	—	—	—	—
0110 0010	62	98	b	—	—	—	—	—
0110 0011	63	99	c	COM	—	—	—	—
0110 0100	64	100	d	LSR	—	—	—	—
0110 0101	65	101	e	—	—	—	—	—
0110 0110	66	102	f	ROR	—	—	—	—
0110 0111	67	103	g	ASR	—	—	—	—
0110 1000	68	104	h	ASL,LSL	—	—	—	—
0110 1001	69	105	i	ROL	—	—	—	—
0110 1010	6A	106	j	DEC	—	—	—	—
0110 1011	6B	107	k	—	—	—	—	—
0110 1100	6C	108	l	INC	—	—	—	—
0110 1101	6D	109	m	TST	—	—	—	—
0110 1110	6E	110	n	JMP	—	—	—	—
0110 1111	6F	111	o	CLR	—	—	—	—
0111 0000	70	112	p	NEG	—	—	—	—
0111 0001	71	113	q	—	—	—	—	—
0111 0010	72	114	r	—	—	—	—	—
0111 0011	73	115	s	COM	—	—	—	—
0111 0100	74	116	t	LSR	—	—	—	—
0111 0101	75	117	u	—	—	—	—	—
0111 0110	76	118	v	ROR	—	—	—	—
0111 0111	77	119	w	ASR	—	—	—	—
0111 1000	78	120	x	ASL,LSL	—	—	—	—
0111 1001	79	121	y	ROL	—	—	—	—
0111 1010	7A	122	z	DEC	—	—	—	—
0111 1011	7B	123	L.BRCE.	—	—	—	—	—
0111 1100	7C	124	SEP.	INC	—	—	—	—
0111 1101	7D	125	R.BRCE.	TST	—	—	—	—
0111 1110	7E	126	WAVE	JMP	—	—	—	—
0111 1111	7F	127	DELETE	CLR	—	—	—	—

1000 0000	80	128		SUBA	—	—	FOR	SGN
1000 0001	81	129	▗	CMPA	—	—	GO	INT
1000 0010	82	130	▖	SBCA	—	—	REM	ABS
1000 0011	83	131	▄	SUBD	CMPD	CMPU	'	USR
1000 0100	84	132	▝	ANDA	—	—	ELSE	RND
1000 0101	85	133	▐	BITA	—	—	IF	SIN
1000 0110	86	134	▞	LDA	—	—	DATA	PEEK
1000 0111	87	135	▟	—	—	—	PRINT	LEN
1000 1000	88	136	▘	EORA	—	—	ON	STR$
1000 1001	89	137	▚	ADCA	—	—	INPUT	VAL
1000 1010	8A	138	▌	ORA	—	—	END	ASC
1000 1011	8B	139	▙	ADDA	—	—	NEXT	CHR$
1000 1100	8C	140	▀	CMPX	CMPY	CMPS	DIM	EOF
1000 1101	8D	141	▜	BSR	—	—	READ	JOYSTK
1000 1110	8E	142	▛	LDX	LDY	—	RUN	LEFT$
1000 1111	8F	143	█	—	—	—	RESTORE	RIGHT$
1001 0000	90	144		SUBA	—	—	RETURN	MID$
1001 0001	91	145	▗	CMPA	—	—	STOP	POINT
1001 0010	92	146	▖	SBCA	—	—	POKE	INKEY$
1001 0011	93	147	▄	SUBD	CMPD	CMPU	CONT	MEM
1001 0100	94	148	▝	ANDA	—	—	LIST	ATN
1001 0101	95	149	▐	BITA	—	—	CLEAR	COS
1001 0110	96	150	▞	LDA	—	—	NEW	TAN
1001 0111	97	151	▟	STA	—	—	CLOAD	EXP
1001 1000	98	152	▘	EORA	—	—	CSAVE	FIX
1001 1001	99	153	▚	ADCA	—	—	OPEN	LOG
1001 1010	9A	154	▌	ORA	—	—	CLOSE	POS
1001 1011	9B	155	▙	ADDA	—	—	LLIST	SQR
1001 1100	9C	156	▀	CMPX	CMPY	CMPS	SET	HEX$
1001 1101	9D	157	▜	JSR	—	—	RESET	VARPTR
1001 1110	9E	158	▛	LDX	LDY	—	CLS	INSTR
1001 1111	9F	159	█	STX	STY	—	MOTOR	TIMER
1010 0000	A0	160		SUBA	—	—	SOUND	PPOINT
1010 0001	A1	161	▗	CMPA	—	—	AUDIO	STRING$
1010 0010	A2	162	▖	SBCA	—	—	EXEC	—
1010 0011	A3	163	▄	SUBD	CMPD	CMPU	SKIPF	—
1010 0100	A4	164	▝	ANDA	—	—	TAB(	—
1010 0101	A5	165	▐	BITA	—	—	TO	—
1010 0110	A6	166	▞	LDA	—	—	SUB	—
1010 0111	A7	167	▟	STA	—	—	THEN	—
1010 1000	A8	168	▘	EORA	—	—	NOT	—
1010 1001	A9	169	▚	ADCA	—	—	STEP	—
1010 1010	AA	170	▌	ORA	—	—	OFF	—
1010 1011	AB	171	▙	ADDA	—	—	+	—
1010 1100	AC	172	▀	CMPX	CMPY	CMPS	-	—
1010 1101	AD	173	▜	JSR	—	—	*	—
1010 1110	AE	174	▛	LDX	LDY	—	/	—
1010 1111	AF	175	█	STX	STY	—		—

1011 0000	B0	176		SUBA	—	—	AND	—
1011 0001	B1	177	▗	CMPA	—	—	OR	—
1011 0010	B2	178	▖	SBCA	—	—	)	—
1011 0011	B3	179	▄	SUBD	CMPD	CMPU	=	—
1011 0100	B4	180	▝	ANDA	—	—	(	—
1011 0101	B5	181	▐	BITA	—	—	DEL	—
1011 0110	B6	182	▞	LDA	—	—	EDIT	—
1011 0111	B7	183	▟	STA	—	—	TRON	—
1011 1000	B8	184	▘	EORA	—	—	TROFF	—
1011 1001	B9	185	▚	ADCA	—	—	DEF	—
1011 1010	BA	186	▌	ORA	—	—	LET	—
1011 1011	BB	187	▙	ADDA	—	—	LINE	—
1011 1100	BC	188	▀	CMPX	CMPY	CMPS	PCLS	—
1011 1101	BD	189	▜	JSR	—	—	PSET	—
1011 1110	BE	190	▛	LDX	LDY	—	PRESET	—
1011 1111	BF	191	█	STX	STY	—	SCREEN	—
1100 0000	C0	192		SUBB	—	—	PCLEAR	—
1100 0001	C1	193	▗	CMPB	—	—	COLOR	—
1100 0010	C2	194	▖	SBCB	—	—	CIRCLE	—
1100 0011	C3	195	▄	ADDD	—	—	PAINT	—
1100 0100	C4	196	▝	ANDB	—	—	GET	—
1100 0101	C5	197	▐	BITB	—	—	PUT	—
1100 0110	C6	198	▞	LDB	—	—	DRAW	—
1100 0111	C7	199	▟	—	—	—	PCOPY	—
1100 1000	C8	200	▘	EORB	—	—	PMODE	—
1100 1001	C9	201	▚	ADCB	—	—	PLAY	—
1100 1010	CA	202	▌	ORB	—	—	DLOAD	—
1100 1011	CB	203	▙	ADDB	—	—	RENUM	—
1100 1100	CC	204	▀	LDD	—	—	FN	—
1100 1101	CD	205	▜	—	—	—	USING	—
1100 1110	CE	206	▛	LDU	LDS	—	—	—
1100 1111	CF	207	█	—	—	—	—	—
1101 0000	D0	208		SUBB	—	—	—	—
1101 0001	D1	209	▗	CMPB	—	—	—	—
1101 0010	D2	210	▖	SBCB	—	—	—	—
1101 0011	D3	211	▄	ADDD	—	—	—	—
1101 0100	D4	212	▝	ANDB	—	—	—	—
1101 0101	D5	213	▐	BITB	—	—	—	—
1101 0110	D6	214	▞	LDB	—	—	—	—
1101 0111	D7	215	▟	STB	—	—	—	—
1101 1000	D8	216	▘	EORB	—	—	—	—
1101 1001	D9	217	▚	ADCB	—	—	—	—
1101 1010	DA	218	▌	ORB	—	—	—	—
1101 1011	DB	219	▙	ADDB	—	—	—	—
1101 1100	DC	220	▀	LDD	—	—	—	—
1101 1101	DD	221	▜	STD	—	—	—	—
1101 1110	DE	222	▛	LDU	LDS	—	—	—
1101 1111	DF	223	█	STU	STS	—	—	—

1110 0000	E0	224		SUBB	—	—	—	—
1110 0001	E1	225	▘	CMPB	—	—	—	—
1110 0010	E2	226	▝	SBCB	—	—	—	—
1110 0011	E3	227	▀	ADDD	—	—	—	—
1110 0100	E4	228	▖	ANDB	—	—	—	—
1110 0101	E5	229	▌	BITB	—	—	—	—
1110 0110	E6	230	▞	LDB	—	—	—	—
1110 0111	E7	231	▛	STB	—	—	—	—
1110 1000	E8	232	▗	EORB	—	—	—	—
1110 1001	E9	233	▚	ADCB	—	—	—	—
1110 1010	EA	234	▐	ORB	—	—	—	—
1110 1011	EB	235	▜	ADDB	—	—	—	—
1110 1100	EC	236	▄	LDD	—	—	—	—
1110 1101	ED	237	▙	STD	—	—	—	—
1110 1110	EE	238	▟	LDU	LDS	—	—	—
1110 1111	EF	239	█	STU	STS	—	—	—
1111 0000	F0	240		SUBB	—	—	—	—
1111 0001	F1	241	▘	CMPB	—	—	—	—
1111 0010	F2	242	▝	SBCB	—	—	—	—
1111 0011	F3	243	▀	ADDD	—	—	—	—
1111 0100	F4	244	▖	ANDB	—	—	—	—
1111 0101	F5	245	▌	BITB	—	—	—	—
1111 0110	F6	246	▞	LDB	—	—	—	—
1111 0111	F7	247	▛	STB	—	—	—	—
1111 1000	F8	248	▗	EORB	—	—	—	—
1111 1001	F9	249	▚	ADCB	—	—	—	—
1111 1010	FA	250	▐	ORB	—	—	—	—
1111 1011	FB	251	▜	ADDB	—	—	—	—
1111 1100	FC	252	▄	LDD	—	—	—	—
1111 1101	FD	253	▙	STD	—	—	—	—
1111 1110	FE	254	▟	LDU	LDS	—	—	—
1111 1111	FF	255	█	STU	STS	—	(SEE FF+)	—

Cassette Loading Problems

The Micro Language Lab tapes contain both audio and programs. Until you get accustomed to the voice-data sequence, you may experience some loading problems.

1. Be sure to have the volume adjusted properly for program loading. Our cassettes load very well with the CTR-80A volume set between 6 and 7, although this may be too loud for listening to the audio.

2. These are 60-minute cassettes and should be treated as good music tapes. Your tape recorder must be clean and demagnetized. Obtain cleaning solution and demagnetizers from a Radio Shack or other hi-fi store.

3. Don't miss the beginning of the program. When you hear "turn the tape off now", than means now! The program begins within 5 seconds.

4. Should you have continued loading problems with one program or one tape, you may exchange the defective tape at no charge. Should you have loading problems with several tapes, suspect your tape player. We use good tape and excellent mastering and duplication equipment to assure quality.

5. If you find the audio-data combination cumbersome, Green Mountain Micro can offer you a separate tape containing all the Micro Language Lab programs. Call or write for price and availability.