Introduction to Vectrex Programming

by Christopher L. Tumber, copyright 1998 all rights reserved.

Contents
========

Introduction
============

This is my attempt at an introductory tutorial on programming the Vectrex and the 6809. It should hopefully put you over the "hump" of just getting started. However, this should not be considered a comprehensive text of Vectrex programming. My own knowledge of the Vectrex is hardly expert and there are plenty of other resources out there.

The most important piece of advice I can give someone learning assembly language for the first time is to go slowly and be sure you understand everything before you move on. There are a lot of very fundamental concepts which will be completely new to you, even if you have programming experience in high level languages (like C, BASIC, PASCAL) and it is vitally important that you have a firm grasp of the basics or what comes later will just be gibberish to you.

Hexadecimal Notation
====================

Learning and getting comfortable with hexadecimal notation is perhaps the single most important step to learning assembly language programming. What's more, it has been my experience in teaching people assembly that it is one of the most difficult subjects to learn.

This is because counting and numbers were one of the first things we ever learned. What's more, we all learned them by wrote when we were 4 or 5 years old and the concepts behind numbers were never explained to us. Instead, we've been using numbers over and over again every day of our lives without ever really thinking about what numbers are or why we use them the way we do.

In order to understand hexadecimal, you have to understand what numbers really mean and why we use the numbers we do.

To begin with, you have to realise that the numbers 1 or 2 or 5 or 397 are arbitrary symbols and don't in and of themselves mean ANYTHING.

Sometime thousands of years ago a couple guys were sitting around and needed a way of counting. They had a bunch of sheep or a heard or cattle and needed to keep track of how many they had. If they had enough for breeding. If they had enough to feed themselves throughout the winter. That kind of thing.
So, they came up with some symbols. They called the symbol 1 "One" and the symbol 2 "Two" and so on but that decision was completely arbitrary. They could just as easily decided to call the symbol 5 "One" and the symbol @ "Two".

What's more, the decision that the symbol 1 or "One" would mean "A single unit of something" and 2 or "Two" would mean "A single unit PLUS another unit of something" is also completely arbitrary. 1 or "One" could have just as easily meant "A single unit PLUS another unit PLUS another unit". Or the symbol 5 could have mean't "I don't have any of these at all" (what we call ZERO).

It's the same with words, really. Why does the word "Apple" mean "A round red fruit that grows on trees"? Because, sometime hundreds or thousands of years ago someone decided "Apple" would be a good name for that fruit and we've been using it ever since. They could just as easily decided "Orange" or "Mustard" was a good name and we'd be using that instead. Yup, my mom bakes a great mustard pie....

The only reason the word "Apple" or "One" or the symbol 1 means anything is because we all agree on the meaning. If you ask someone who only speaks French for an "Apple" he's not going to know what you are talking about. Not because the word "Apple" doesn't have any meaning anymore, but because his language doesn't agree that "A round red fruit that grows on trees" is called an "Apple" but instead he calls it a "Pomme".

Similarily, we have all agreed that the numbers 0,1,2,3,4,5,6,7,8,9 all have a certain meaning.

However, sometime ago in the days of the Roman Empire we would have been counting I,II,III,IV,V,VI,VII,VII,IX,X. (Most people have seen Roman Numerals before, if not, they still appear often on clocks and watches...)

Both systems of counting refer to the same thing. Both are just as valid.

We use our system primarily because we have ten fingers. This system of counting is known as Base 10 or Decimal because there are ten different symbols used (0 through 9).

When we count in decimal, we start with 0 and go 1,2,3,4,5,6,7,8 up to 9. However, when we reach the number TEN we have a problem. We don't have a single, simple symbol for the number TEN. Instead, we have to start to RE-USE symbols. So, we use two symbols instead of just one and come up with: 10.

Why?

The most obvious answer is "Because my first grade teacher said so" but that doesn't really address what's going on here. We counted from 0 to 9 and then ran out of symbols. So, we had to come up with something to do because we need to be able to count A LOT higher than just 9.

So, what we do when we get "Stuck" at 9 is instead of using just one symbol as we do to represent 0 through 9 we use TWO symbols side by side to represent the numbers 10 through 99. This is important! We get around the fact that we REALLY only can count from 0 to 9 by adding a new column of symbols, by using two symbols instead of just one. So, instead of having a different symbol for every possible number there is (billions or trillions of symbols) we only have TEN symbols and we just re-use these TEN by symbols by putting more and more of them next to each other. So we can make a number like 145 or 32354654.

What's more, the symbol and position of the symbol is important. The first of two symbols is the "Tens column". This is the number of TEN items we have, or, it's the number of times we got "Stuck" and nine and had to carry over to the tens column because we ran out of symbols. The second symbol is the "Ones columns" or the number of single items, just as when we were counting from 0 to 9.

Using this system, we can express ANY number using just the symbols 0 through 9. So the symbols 324 means we counted up from 0 to 9 and had to carry over to 10. Then, we counted up from 10 to 19 and AGAIN had to carry over to get 20. We repeated this process all the way up to 99. At 99, we were again out of symbols so we need to use THREE symbols now and carried over to 100. We keep up this process of counting up and carrying over until we reached 324.

As I mentioned, we use this system primarily because we have 10 fingers. Well, what if we DIDN'T have 10 fingers? What if we only had 8 (Octal) or what if we had 16 (Hexadecimal)...

You can see where this is going....

Well, a computer actually only has two fingers. Or to be more precise, a computer has switches. Switches are either on or off. All of a computer's operations, including counting, are just turning switches on or off.
So, a computer can only really count from 0 to 1. Unlike us, where we can count from 0 to 9. However, just as we can "fudge" it and count higher than 9 by using more than one symbol (324) so too a computer can count higher than 1 by using more symbols.

However, just as we get "Stuck" when we reach 9 and have to start over at 0 with another symbol 1 in front, a computer has to start over after it hits 1.

So a computer counts:	When we count:
0	0
1	1
10	2
11	3
100	4
101	5
110	6

This is called "Binary" or "Binary Notation" and it is the way all digital computers actually count. Counting in binary, we can't use the symbols 2 through 9 (We just can't, okay, that's the rule!) we can only use 0 and 1. So, every time we hit 1 it's just like hitting 9 in decimal and we have to carry and add another symbol if we want to count any higher; going from 1 to 10 or 11 to 100. When a computer counts from 11 to 100 it's just the same carry over as when we count 9 then 10 except the computer runs out of symbols earlier so it has to carry and add a symbol at "two" instead of at "ten".

So, if all computers actually count in Binary, why is Hexadecimal so important?

Well, it turns out that Binary and Hexadecimal happen to fit together really nicely.

As you may know, a single Binary digit is called one "Bit" so 0 is a bit or 1 is a bit and 11 is two bits. 111 is three bits and so on.

Well, 8 "Bits" make a "Byte". The term "Byte" you've certainly heard before, it's one of the most common computer terms. Since a byte is 8 bits, a byte is: 10001000 or 10101010 or 11111111 or any other such combination of 8 0's and 1's.

A slightly less know term is a "Nibble", a nibble is 4 bits or half a byte. A Nibble is 1000 or 1111 or 0101 or...

Let’s consider all the numbers that can be represented by a Nibble:

Binary	Decimal	Hexadecimal
0000	0 (Zero)	0
0001	1 (One)	1
0010	2 (Two)	2
0011	3 (Three)	3
0100	4 (Four)	4
0101	5 (Five)	5
0110	6 (Six)	6
0111	7 (Seven)	7
1000	8 (Eight)	8
1001	9 (Nine)	9
1010	10 (Ten)	A
1011	11 (Eleven)	B
1100	12 (Twelve)	C
1101	13 (Thirteen)	D
1110	14 (Fourteen)	E
1111	15 (Fifteen)	F

In the case of 0000 or 0001 we use the leading zeros to show that the number is four bits (a nibble or half a byte). In the same way 009 or 09 is the same as 9 (in Decimal or Base 10), we just normally drop the leading zeros because we don't need them. However, in Binary (and Hexadecimal) Notation we tend to keep them in there to show how much space the number takes up (a nibble, a byte or even two bytes).

As you can see, I've also included the equivalent Hexadecimal Notation.

In Hexadecimal, instead of using 10 digits (symbols) as in Decimal or using 2 digits as in Binary we use 16 digits (symbols).

Of course, we only actually HAVE the digits 0 through 9 to use. But, we need 16. So, we borrow the extra 6 symbols we need from the alphabet. We use the letters A through F.

This is a big sticking point for some reason, this using letters as numbers. A lot of people have trouble getting their brain around it. Remember earlier we decided that numbers and their meanings are completely arbitrary. 1 doesn't REALLY mean anything except that we say so. 5 doesn't REALLY mean anything either, we just say it does. Well, now, we're going to say that the letter A means "One more than 9" and the letter B means "One more than A" and so one.

We could have used ANY symbols. We could have made up new symbols. We use the letters A through F because they're a lot easier to remember than some weird symbol just made up for that purpose. There's also no other occaision where you'll see numbers mixed up with letters like 2A5F.

Except Hexadecimal (and some postal codes and license plates....!)

So there's not really any room for confusion.

Now, if you refer back to that listing of all the values that can be represented by a nibble, you'll see that all nibble values can be represented by a single Hexadecimal symbol from 0 to F.

Pretty convenient, eh?

If order to represent those numbers in decimal, we have to use 0 to 15. (Two digits, where we can get away with only one in hexadecimal. Plus, decimal may be 1 digit or it may be 2. Hexadecimal is ALWAYS 1 digit. This is actually important as where computers are concerned standardisation is a REALLY good thing!)

In fact, if you take a Byte:

Binary	Decimal	Hexadecimal
00000000	0	00
00000001	1	01
.		
.		
.		
10001001	137	89
10001010	138	90
.		
.		
.		
11111110	254	FE
11111111	255	FF

I'm not going to list all the possible values in a Byte because that would be 256 lines long. However, as you can see any Byte value of 8 bits can be expressed with 2 Hexadecimal digits. It takes up to 3 decimal digits.

The only reason we use Hexadecimal is because it translates back and forth to Binary so nice and neatly. Trust me, it's A LOT easier to deal with Hexadecimal than Binary!

In order to avoid confusion, it is often necessary to be able to tell when we're talking about a decimal number or a hexadecimal. The most common method is to precede hexadecimal numbers with a $. (There are others but I'm not going to go into them here. I'll only use these and if/when you come upon the others you'll know.)

So we would write 10 or $0A. These are really the same number.

While working on an 8-bit machine like the Vectrex you will often see 1 and 2 byte hexadecimal values.

A 1 byte hexadecimal value looks like $2F.

A 2 byte hexadecimal value looks like $C32F.

The difference is similar to the difference between 10 and 1000, a matter of scale. In most computers, values tend to be divided up neatly into Bytes. You CAN work with just 1 bit, or with a nibble or any part thereof but most of the time you're working with 1 byte or 2 byte values. (You can also work with larger values, but probably won't be doing so very often on the Vectrex).

1 byte values can represent $00 to $FF (0 to 255) and 2 byte values can represent $0000 to $FFFF (0 to 65535). Again, it's a matter of convention because these values fit neatly into bytes.

If you need to translate hexadecimal values back to decimal, it is very easy to do so.

Remember, is decimal a number like 324 really means:

```
    3 X 100   (3 times 100)      (100 = 10^2)
  + 2 X 10    (Plus 2 times 10)  (10  = 10^1)
    4         (Plus 4 times 1)   (1   = 10^0)
   -------
    324
```

(The symbol ^ means "To the power of")

To convert a hexadecimal number back to decimal, you do the same thing, EXCEPT each leading digit is a factor of 16 instead of a factor of 10.

So, $C32F is

```
    12 X 4096  ($C=12 -> 12 times 4096)     (4096 = 16^3)
+    3 X 256   ($3=3  -> Plus 3 times 256)  (256  = 16^2)
     2 X 16    ($2=2  -> Plus 2 times 16)   (16   = 16^1)
    15         ($F=15 -> Plus 15 times 1)   (1    = 16^0)
   -----------
    49967
```

So $C32F = 49967

You could do all your assembly programming in decimal, however it isn't nearly as neat and tidy as hexadecimal simply because of the way hexadecimal translates back to binary. In other words, hexadecimal is a kind of shorthand. A compromise between binary, which is what the computer actually uses but is quite awkward and unwieldy for us humans and decimal which is what we humans normal use but is awkward for computers.

Your assembler should be able to deal with both hexadecimal and decimal (and binary for that matter) values equally. It's generally up to you to decide which is most convenient for you to use in any given situation. You'll tend to find that you refer to memory locations in hexadecimal but some data and variables in decimal. It's really a matter of convenience, when telling the computer what to do it's generally easier to "speak it's language" and use hexadecimal.

You'll also find hexadecimal used all over the place by other programmers so it's REALLY important to learn if you're going to look at anyone else's code. You could conceivably write all your programs using only decimal notation but it'd look really odd and you probably get beat up and called a geek at recess...

Signed or Two's Compliment
==========================

The previous description of Hexadecimal Notation has one problem.

How do you express -80 in Hexadecimal?

With a lot of CPU's you're on your own if you want to use negative numbers and have to come up with your own way of dealing with negatives.
The 6809 however uses Two's Compliment integers and has the capability of dealing with negative numbers built in.

Two's Compliment is kind of a weird format (and you thought hex was strange!) where negative numbers are the bit-wise inverse of the positive number +1.

```
So,      1 = 0001
but     -1 = 1111 (1110 + 1 =1111)

and    127 = 0111
but   -127 = 1001 (1000 + 1 =1001)
```

The bottom line is that in 1 byte you can use -128 to 127 and in 2 bytes you can express -32768 to 32767 and if you're going to use positive and negative values (and you will!) you should probably use decimal instead of hexadecimal.

Memory
======

First of all, a computer's memory is NOTHING like a person's memory.

As noted before, a computer really only sees everything as a series of 0's and 1's. We abstract these 0's and 1's out to be hexadecimal numbers which then mean something to us. To the computer, they're just a bunch of on and off switches.

Picture a computer's memory like a school hallway filled with lockers. Each locker is one byte of memory and can hold one thing. Each locker is also individually numbered (in hexadecimal) so we can identify it. (ie: "What's in locker $2333??")

For our purposes, there are two types of memory, RAM and ROM.

If a locker is ROM (Read Only Memory) we can look into the locker but we can't EVER change the contents.

If a locker is RAM (Random Access Memory) we can look in the locker AND we can change the contents at any time. We can store anything we want in there.

So, if your PC has 8 Megs RAM, it has 8 million lockers that can be used to store things (A meg is a million).

There are two basic kinds of things we can have in each locker, a program or data. Our program is simply a series of instructions which tell the Vectrex what to do. Just like you tell your kid brother what to do you have to tell the Vectrex exactly what to do. This is our program.

Our lockers can also contain data. Data is a little different than program in that it isn't exactly instructions for the Vectrex, rather, it is information our program will use in some way (ie: Lines of text, musical notes in a song).

To the Vectrex, a locker is a locker, it doesn't matter what's stored there. All that matters is what we tell the Vectrex to do with that locker. The locker could contain parts of our program or it could contain some data.
We generally tell the Vectrex to execute program instructions, however, it will be more than happy to try and execute data as if it were instructions or manipulate our program as if it was data. It's up to us to make sure we tell the Vectrex to go to the right lockers when we want it to do something or to go to the right locker to get some data.

Again, each of these lockers is 1 byte. So each locker actually just contains a number. If the locker contains program code, then that number represents an instruction for the computer. If the number is $01 it tells the Vectrex to do one thing, if it is $55 it tells the Vectrex to do something completely different.

If the contents of the locker is data, then it could be the circumference of a circle or the distance from the Earth to the Moon. It's up to us to decide. To the Vectrex, it's all just numbers. We just tell the Vectrex "Do the instructions in locker $1000" or "Get us the data in $1000".

In the high-school named "Vectrex High", locker numbers $0000 to $7FFF contain our program (game). These lockers are ROM. They're burned onto the cartridge and may never be changed. $0000 to $7FFF is 32768 lockers or 32K. Therefore, our maximum program size is 32K (without doing any REALLY fancy stuff that's well beyond the scope of this text...) We decide the contents of these ROM lockers when we write our program. When the Vectrex starts up, it (essentially) goes to the first locker, at $0000 and looks for the first instruction. It then proceeds sequentially through the lockers ($0000 then $0001 then $0002 then...) until or unless the instructions in those lockers tell it to do otherwise.

Locker numbers $C800 to $CFFF are _supposed_ to be RAM that we can use, however for some reason we can only ACTUALLY use $C800 to $CBFF (If you ever discover why, please let us all know!!). What's more, the Vectrex itself actually uses lockers $C800 to $C87F for it's own storage purposes. So we really only can use lockers $C880 to $CBFF to store our gym shorts. These lockers are empty when you first turn on the Vectrex. However, throughout the course of our program, we can put stuff in here. Stuff like scores, spaceship positions, etc.

Locker numbers $E000 to $FFFF are also ROM. However, THESE lockers were filled by the designers of the Vectrex. These lockers contain the BIOS. The BIOS is a set of program instructions built into the Vectrex that we can use for our own programs. A handy bunch of instructions that do all kinds of things like draw vectors and make sounds.

Obviously, there are a whole bunch of lockers I haven't mentioned, but we don't need to go into those. Really!

These are the basic lockers you need to be able to get into.

The Instruction Set
===================

The Vectrex (or to be more specific the 6809) has a number of built in instructions. Just like you can tell your kid brother to wash the dishes for you but you can't get him to drive you to the mall because he's only 8, the Vectrex can only do so many things. This is the instruction set. You WILL need a list of the Vectrex's instruction set. I won't provide it here, there are a couple on my WWW site and you can get a copy from Motorola themselves (this is a REALLY good idea, btw! See my web site for instructions on how).

A single line of instruction for the Vectrex looks like this:

```
$1000     deca

  ^        ^
  |        |
  |         ---------- The instruction
  |
   ------------------- The locker number
```

It doesn't really matter what this actually does, I just want you to get the idea of the format.

The first number, $1000 is the memory location where this instruction resides. This is the locker number where the Vectrex will find this instruction. The term "deca" is an instruction, it tells the Vectrex to do something.

Most instructions will, at first look incomprehensible to you. In most cases they are abbreviations which is why they don't mean anything to you right now. There are maybe a dozen instructions you will use all the time, and you'll quickly come to recognise them.

As I mentioned previously, the contents of all lockers (memory) is actually just a number. So why did I just tell you the contents of locker $1000 is "deca"?

Well, I lied!

$1000 does just contain a number. However, if you were going to try to memorise all instructions by their actual number you would have a REALLY hard time as there are hundreds of them. Instead, we use a mnemonic like "deca" because it's much easier to remember and understand.

The purpose of an ASSEMBLER is to take our mnemonics like "deca" and translate them to their real, numeric code that the Vectrex actually sees (in this case "deca" becomes $4A). Meanwhile, a DISASSEMBLER will convert the numeric code back to mnemonic code that we can understand more easily.

In general, each of the various instructions provided do a very small task. Unlike a high level language like C or BASIC where the instructions can perform relatively complex tasks, assembly language instructions tend to be very simple and it is by doing several instructions in sequence that we accomplish large jobs.

Once the Vectrex has finished with our instruction, it will move onto the next locker and look there for another instruction. For example:

```
$1000    deca
$1001    tsta
```

That's generally the way it works. The Vectrex just goes sequentially through all our lockers executing whatever instructions it finds there.

Of course, it's a little more complex than that.

Some instructions tell the Vectrex to go to a different locker to find the next instruction.

Some instructions are a little more complicated and actually take up to four lockers to hold.

Something like:

```
$1000     sta $C880
$1003     deca
$1004     tsta
```

The Assembler
=============

Our list of instructions for the Vectrex is know as Source Code. All this is is a normal text file with all your instruction for the Vectrex in it. You can write your source code in any text editor as long as it will save in standard ASCII text.

An Assembler is a program that translates our source code into the actual machine language that the Vectrex can read.

In my case, I use the AS9 assembler which adds an intermediate stage in that it doesn't translate into machine code directly but instead into hexadecimal. The program hex2bin is then used to convert this to the true machine code.

To take my source code, run it through the assembler AND run the resulting program on the Vectrex emulator, I enter the following commands at the DOS prompt:

```
as9_new omega.as9
hex2bin omega.s19
vectrex omega.b00
```

The file omega.as9 is my source code. The file omega.b00 is the resulting Vectrex ready game file like BERZERK.GAM or BEDLAM.GAM for the emulator.

Labels and Constants
====================

If you look at some source code, you'll notice that every line DOES NOT start with a locker number.

If fact most if not all lines don't have locker numbers at all.

This is because the ASSEMBLER does this for us automatically. Let's say we wrote a program that was 100 lines long. Having to type in every memory address (locker number) for each program line would be very tedious. Also, because some instructions take up more than 1 locker it would be very easy to make a mistake and wind up with instructions in the wrong locker.

Instead, we just tell the ASSEMBLER what locker to START putting our program into. After that, the ASSEMBLER just goes sequentially down the line of lockers adding more instructions according to our source code.

We tell the ASSEMBLER where to start with a line like:

```
        ORG     $0000
```

This line, which you'll find near the beginning of every Vectrex program tells the ASSEMBLER "Start putting program code in memory location $0000".

If you were programming on a different machine, you might start at a different locker. However, on the Vectrex you ALWAYS start with locker $0000.

So, instead of our code looking like this:

```
$1000     sta $C880
$1003     deca
$1004     tsta
```

It's just going to look like this:

```
          sta $C880
          deca
          tsta
```

But, what if for some reason we need to find the lockers with "sta $C880" in it? Before, we could say it's in $1000. We always knew where it was. But, because the ASSEMBLER is doing the counting for us we don't really know exactly which locker it's going to put this instruction in. What if we move this piece of code around? What if we put something in the locker in front of it, so it all gets moved down a locker?

That's what a label is for.

If we need to be able to find this locker later, we attach a label to it like this:

```
kill_ship:
          sta $C880
          deca
          tsta
```

The characters "kill_ship" is the label, using the colon is what indicates that this is a label and isn't considered part of the label itself.

This label acts sort of like a name tag stuck onto the locker. We don't have to refer to the actual locker number anymore, we can just refer to the name tag (label). When the ASSEMBLER actually translates our source code into machine code, it subtitutes the actual locker numbers for any labels.

The big benefit of using labels is that they are a lot easier to remember AND they are a lot more meaningful than just a locker number. If at some point in the future, you need to find the above code you will remember that it's kill_ship, remembering that it was at $1000 would be a lot more difficult. Also, if someone else comes along and reads your code, kill_ship gives you a pretty good idea of what that code does. $1000 could mean anything.

Taking this one step further, there are often a lot of numbers or values you use over and over again. Often these will be declared as constants at the begining of a progrma. If you look at any Vectrex source code you will generally see a list of such constants. Constants look like:

```
user_RAM        EQU     $C880
```

The line above means that whenever the ASSEMBLER sees "user_RAM" later it interprets it as $C880. As shown in the Memory section, $C880 is the start of RAM available for use by the programmer.

So, by adding this line to the start of our program, whenever we need to look in the first RAM locker we use, we can say "user_RAM" instead of $C880.

After adding this line to the start of our program, we could change:

```
kill_ship:
          sta $C880
          deca
          tsta
```

To instead read:

```
user_RAM        EQU     $C880
```

```
kill_ship:
          sta user_RAM
          deca
          tsta
```

These two snippets of code mean the very same thing. However, using labels and constants has made it much easier to read, and much easier for us to remember where things go.

In addition, constants MUST all be placed at the very begining of your source code, BEFORE any actual instructions. This makes constants even more convenient because if you ever forget them, you can just go to the start of code and look them up.

Registers
=========

Lockers (memory) can hold either program or data.

If lockers contain data, there's generally not much you can do with it while it's sitting in the locker. You have to take it out to use it.

To do this, you take data out of a locker and put it in your pocket. Once in your pocket, you can fool around with it. Change it, manipulate it, and eventually if you want to, put it back in the locker.

Of course, you have several pockets. A couple in your pants, your jacket and of course your shirt pocket with it's pocket protector! So, you can have a few things in your pockets without having to put them back into a locker while you work with what's in another pocket.

The Vectrex's pocket's are called REGISTERS.

The registers are called (rather unimaginatively) A,B,D,X,Y,U,S,PC,DP and CC.

Some of these pockets actually have pretty specific uses.. You only put your wallet in your back pocket or your breast pocket for example. But some are very general purpose; your jacket pockets might contain your keys or change or...

The A and B registers are 8-bit (1 byte) registers which are the most often used for general purpose data manipulation. However, the A and B registers are a little weird. They're like how some jeans have a little "change pocket" inside the front right pocket. You know, anything in that little pocket, it's not in the same pocket as your front right pocket. But, if you think about it it sort of is...

A is the right hand pocket. B is the little change pocket. The D register is these two pockets combined! The D register is a 16-bit (2 byte) register that is actually a combination of A and B.

So, if the A register has a value of $A0 and the B register has a value of $FF then the D register has a value of $A0FF. If you change the contents of the D register, you ALSO change A and B. If you change A, then you change the "top" part of D. If you change B then you change the "bottom" part of D.

Generally you are either using A and/or B for their 8-bit storage OR you're using D for it's 16-bit storage at any given time. It's sort of "cheating" by giving you a couple extra registers without using up any more space because A and B "overlap" with D.

Most of your data manipulation type work will be done using A, B or D.

The X and Y registers are two 16-bit registers. They're usually used as pointers in Indexed Mode Instructions (see below).

U and S are both Stack Pointers (more on that later).

PC is the program counter. This is the number of the current locker being checked to see what instruction to execute. This register can be manipulated by a variety of instructions to cause the program to progress other than sequentially (for example, if you want to execute some code over and over again).

DP is a special register for use with Direct Addressing. More on that later as well.

CC is the Condition Code Register. It tells you the results of some types of instructions the Vectrex just did (ie: If the result was a Zero, or Negative, or if there's a Carry or Borrow (for addition/subtraction)).

As an example. Let's say locker number $C880 contains the number of ships a player has left in the game. Let's say the player has just qualified for a bonus ship by scoring a bunch of points.

What we do, is we go to locker $C880. We see what value is stored in that locker (The player could have 1 ship left or 5 or anything...). We take this value and put it in A. We then add 1 to A. We put whatever's in A back into locker $C880. We then go on to do whatever else needs to be done (play some special "Bonus Man" music, continue with the game, etc...)

On the other hand. Let's say the player has just been killed.

Well, then we'd go to locker $C880. Put the contents of that locker into A. Now, we subtract 1 from A. Next, we check if A is equal to zero. If it is, we go do whatever we need to do to end the game ("Got you humanoid!"). If it's not 0, then we put the new value in A into $C880 and continue with the game.

Somewhere else in the program, we would have a some instructions that look into $C880 and then print that many ships remaining onto the screen.

Seems like a lot of work, eh?

Well, it is! Nobody said this was easy!

That's why most sane people use C or BASIC or PASCAL....

Actually, it's not really that difficult. The above isn't really the best example. It actually IS possible to do some VERY SIMPLE things to data while it is still in a locker. So doing something like increasing/decreasing the number of ships left would actually be simpler. However, for any complex operations you do have to move data out of the locker (memory) and into your pockets (registers).

Some Instructions

We've finally come to the point where you're going to have to start to learn some of the 6809's instruction set into to go any further. So, we'll start with some of the more common and more simple instructions.

The first is LD.

LD is an abbreviation for LoaD. LD simply puts a value into one of the Vectrex's pockets (registers). LD may be used with any of several registers:

LDA - Load a value into the A register
LDB - Load a value into the B register
LDD - Load a value into the D register
LDS - Load a value into the S register
LDU - Load a value into the U register
LDX - Load a value into the X register
LDY - Load a value into the Y register

To go back to our previous examples, suppose we were starting a game and we wanted to put a value of 3 into A because that's how many ships a player starts out with. The instruction to do this would be:

```
        LDA #3
```

Pretty simple, eh?

That sticks a 3 right into the A register. Anything that happened to be in A already is overwritten and lost so you want to be sure that whenever you put something into a register (or memory location for that matter) that you're done with whatever was in there before.

On the other hand, if you wanted to to take whatever's in memory location (locker) number $C880 the instruction would be:

```
        LDA $C880
```

That takes a copy of whatever is in $C880 and copies it into register A. $C880 is itself not changed and keeps whatever was there. In effect, taking something from a locker (memory) and putting it into your pocket (register) only copies and doesn't actually move it.

Another ultra-common instruction is ST.

ST is an abbreviation for STore.

STore takes a value in a register and copies that value into a memory location. ST can be used with the A, B, D, S, U, X and Y registers.

To store the contents of register A back into $C880 the instruction is:

```
        STA $C880
```

Pretty simple, right?

Okay, two more basic instrcutions and then we're done for now. These are INC and DEC. Both INC and DEC may be used with register A or B or directly on memory. Remember I said a few instructions could be used directly on memory? Well, INC and DEC are two of those.

INC is and abbreviation for INCrement and DEC is an abbreviation for DECrement.

The instruction:

```
               DECA
```

Will reduce the value of register A by one.

The instruction:

```
              INCA
```

Will increase the value of register of A by one.

So, summing it all up, if we wanted to take the contents of $C880, put it in register A, subtract 1 and then put the result back into $C880 the instructions would be:

```
ships_left     EQU   $C880
```

[There'd be a whole bunch of other code in here]

```
kill_ship:
          LDA ships_left   ;Get number of ships remaining
          DECA             ;Subtract 1
          STA ships_left   ;Put result back
```

(We could have simply used the instruction DEC $C880 or DEC ships_left. However, we've done it this way for a reason. So that we can do other things while the number of ships left is in A. Like check if the game is over because the player's out of ships. We'll tackle this later.)

This introduces something else new. Commenting.

Any time the ASSEMBLER sees a semi-colon (;), it ignores everything to the right of that semi-colon and treats it as a comment. You can use comments to remind yourself what sections of code do, particularly complicated sections whose purpose might not be obvious. Also use comments to help other people trying to read and decipher your code.

Addressing Modes

```
================
```

The 6809 offers 5 different addressing modes.

The addressing modes Inherent, Immediate, Extended, Direct, Indexed
and they look like this:

```
DECA             - Inherent
LDA #$00         - Immediate
LDA $C880        - Extended
LDA $80          - Direct
LDA #$80,X       - Indexed
LDA B,X          - Indexed
LDA ,X++         - Indexed
LDA [$C880]      - Indexed
```

Addressing modes are slightly different variations on each instructions. Each instruction may be able to use 1 or more addressing modes. No instruction can use all 5 modes. The simplest modes are Inherent, Immediate and Extended.

Addressing modes are just the different ways an instruction can handle data. For example, you go to the library to do research on the big science fair project. To find out about volcanos, you can go to the encyclopedias. You can look up the books specifically on volcanos. You can search the web for pages on volcanos. You can use the microfiche to look up newspaper and magazine stories on volcanos.

Any way you do it, it's all a very similar result. You're looking up information on volcanos.

Addressing modes are like those alternate sources of information. They're alternate targets for the assembly language instruction.

Immediate mode means that the target of the instruction is an actual value, and that value will follow the instruction immediately. So,

```
LDA #$00
```

Puts the actual value #$00 (Zero) into register A. If you were to pry open the chip and examine the A register right after executing this instruction, you would find it contains the number #$00.

Extended mode means the target of the instruction is a MEMORY LOCATION, and the memory location follows the instruction. So,

```
LDA $C880
```

Copies whatever was in memory location (locker number) $C880 into register A. If there was a #$00 in $C880, then A will now also contain #$00. If there was a #$FF in $C880, then A will now also contain #$FF.

Inherent mode stands alone and unlike Immediate or Extended takes no further arguments like #$00 or $C880. Instead, the effect of the instruction is completely self-contained. These are really the simplest of instructions.

```
DECA
```

Means decrement (decrease) the A register by 1. So, if the A register previously contained a #$01 (One) it would now contain a #$00 (Zero). As you can see, the effect of the instruction is self-evident. You don't need any further information whether a memory address or actual value. It's all right there.

Direct addressing is actually very similar to Extended. Picture if you will some code which looks like this:

```
LDA $C880
STA $C881
LDA $C882
STA $C883
LDA $C884
STA $C885
LDA $C886
STA $C887
LDA $C888
STA $C889
```

Basically, all this does is move some values around. From $C880 to $C881, from $C882 to $C883. This is just an example, it doesn't really do anything. But it's similar to a lot of stuff you might actually want to do.

As you noticed, this code is actually VERY repetitive. In particular, $C8 is used over and over again. As you've probably noticed, computers are really good at doing repetitive tasks, and at making it easier for you to tell it to do repetitive tasks.

Direct addressing is just such a short-cut.

With Direct addressing, we could instead do this:

```
LDA #$C8
TFR A,DP
LDA $80
STA $81
LDA $82
STA $83
LDA $84
STA $85
LDA $86
STA $87
LDA $88
STA $89
```

Now, I've thrown a bit of a curve-ball at you here. You've never seen TFR before. Well, I'll explain.

Direct mode make uses of another register, the DP register. The DP register is a 1 byte register whose sole function is Direct mode addressing (You could possibly use it as an extra data register like A or B but that would probably be more trouble than it's worth).

When the 6809 encounters a Direct mode instruction, it consults the DP register and using the value in DP, acts like an Extended mode instruction, but with that DP value as the first two digits in the address.

So, if DP is equal to #$C8 then:

```
LDA $00
```

is the same as:

```
LDA #$C800
```

See, shortcut! Saves you typing, and saves 1 byte of memory in the program by not having to have #$C8 there. It's also faster. This Direct mode addressing can be very handy because you are often referring to memory locations that have the same prefix. For example, I mentioned that $C800 to $CBFF are free RAM that you can use. Well, if you only use $C800 to $C8FF, you can use Direct mode in all of you instructions involving RAM.

As for the TFR instruction. Well, unlike the A, B, D, X, Y, U and S registers you CAN'T do a LoaD instruction directly into DP (You just can't, don't you argue with me young man!). So, instead, what we do is:

```
LDA #$C8   ;Puts #$C8 into regiuster A
TFR A,DP   ;TransFeRs the contents of A into DP
```

So we've just done and end-around this limitation and put the value #$C8 into DP in a round-about way... First into A, then into DP. Think Double play ball: Shortstop to Second to First. You can't skip Second base or you miss the out...

Indexed addressing modes are a little more complicated and a bit involved. So we'll hold off looking at them for a while. You're better off getting yourself familiar with some of the other more basic instructions and concepts involved in 6809 and Vectrex programming.

Program Control
===============

Normally, your program will proceed sequentially from first instruction to last. So:

```
LDA $C880
STA $C881
LDA $C882
STA $C883
```

Starts with the LDA $C880, then does STA $C881, then LDA $C882 and finally STA $C883.

Often that's not enough. Sometimes you want to execute certain portions of code only under certain conditions. Sometimes you want to re-use code. The 6809 offers a variety of instructions to alter the flow of the program.

The instructions to do this are:

BCC
BCS
BEQ
BGE
BGT
BHI
BHS
BLE
BLO
BLS
BLT
BMI
BNE
BPL
BRA
BRN
BSR
BVC
BVS
JMP
JSR

Yes, that's a lot of them. But you don't really need to know all of them right now, we'll stick with the basics.

JMP (JuMP) and BRA (BRanch Always) are unconditional jumps.

For example:

```
  LDA $C880
  STA $C881
  JMP another_section
.
.        ;Bunch of code in here
.
another_section:
.
.        ;More code
.
```

When the program hits:

```
JMP another_section
```

The next instruction that will be executed is at the label another_section:

The difference between JMP and BRA is that JMP can jump to ANY place in the program. BRA on the other hand is limited, it can only go a maximum of 127 bytes ahead or behind. This makes it a shorter instruction than JMP, JMP takes 3 bytes while BRA only takes 2. So, if possible you generally want to use BRA but that's often not practical.

JSR (Jump to SubRoutine) and BSR (Branch to SubRoutine) call subroutines. Subroutines are sections of code which you want to re-use. Some sections of code can be made to serve a very general purpose and can be re-used in a variety of places. For example, something simple like adding two numbers.

You don't neccessarily want to re-do that same code over and over every time you want to add two numbers. So, you use a subroutine:

```
  LDA $C880
  STA $C881
  JSR another_section
  LDA $C884
  STA $C885
.
.        ;Bunch of code in here
.
another_section:
  LDA $C882
  STA $C883
  RTS
.
.        ;More code
.
```

As before, when the program gets to:

```
  JSR another_section
```

It jumps immediately to:

```
another_section:
```

However, in this case things are a little different. The program will later come to:

```
  RTS
```

All RTS means is ReTurn from Subroutine. As soon as the program hits one of these, it goes back to where it hit the last JSR and resumes just after that JSR. So in other words, the above code has the same effect as just:

```
  LDA $C880
  STA $C881
  LDA $C882
  STA $C883
  LDA $C884
  STA $C885
```

Just as with JMP and BRA, BSR is very similar to JSR. Except JSR can go anywhere in the program, while BSR is limited to 127 bytes ahead or behind.

The other branching commands are all conditional. They only do the jumping if a certain condition has been met. As I previously mentioned, one of the 6809 registers is the Condition Code (CC) register. This register keeps track of the results of previous instructions. It's composed of several pieces (actually bits) one or more of which are checked by the conditional branches to determine their effect.

These bits are: Half Carry (H), Negative (N), Zero (Z), Overflow (V) and Carry (C).

If an operation results in a Zero, then the Zero bit will be set. If an operation results in a Negative, the the Negative bit will be set. And so on.

For example, BEQ (Branch on Equal) only branches if there was a Zero result. Suppose we write some code which is executed in a video game when the player gets shot or collides with something. We would do something like this, a subroutine called kill_ship:

```
ships_left   EQU  $C880   ;Set up a variable which keeps track of remaining
.                         ;ships
.
.
.
kill_ship:
  DEC ships_left    ;Decrement the number of ships left by 1
  BEQ game_over     ;Check if that was the last ship, and branch if it was
.
.                   ;Code which restarts level, etc
.
.
game_over:
.                   ;Code which ends the game
.
.
```

We only want to go to the game_over section of code if that was the last ship. Otherwise, we just continue with the game. You will use these kinds of conditional branches all over the place. Virtually every time you want your program to check if something has happened... If the player has crashed into a wall, i a player has been shot, if a player has moved the joystick, if the player has pressed fire, if the player's shot has hit anything, if the player deserves an extra ships, etc, etc, etc.

BNE (Branch Not Equal) is the exact opposite of BEQ. This instruction branches only if the Zero bit had NOT been set.

The other branch instructions are all fairly simple. The MC6809E Microprocessor Programming Manual from Motorola gives a description of them all, as will most other reference materials. So I'll leave it to you to discover them, basically there's a branch for ever bit in the Condition Code (CC) register and then some which test 2 bits.

However, I will mention a couple things when dealing with branching instructions.

First, all of the branching instruction beginning with the letter B (That is, everything except JMP and JSR) are short jump instructions and are limited to jumping 127 bytes forward of back. However, you CAN turn these all into long jump instructions by placing the letter L in front of the instruction name.

So, BRA is a short jump instruction and LBRA is the same instruction, but will handle long jumps. The difference is that LBRA takes an extra byte.

The other thing to keep in mind is that some assemblers treat Relative and Absolute jumps differently and can give you weird results if you mix them.

The BRA, BEQ, BNE, BSR (all of the starting with B) jumps and their L prefixed equivalents (LBRA, LBEQ, LBNE, LBSR) are all Relative jumps. JMP and JSR are absolute jumps.

What this means is that:

```
BRA some_place
```

When it's assembled, actually tells the 6809 to "Jump ahead/behind a certain number of bytes" the number of bytes to jump is determined at the time of assembly by how far away the label "some_place" is. So, if some_place is 20 bytes ahead in the code, then this instruction is actually translated to machine code meaning "Jump ahead 20 bytes".

On the other hand:

```
JMP some_place
```

Is translated to mean "Jump ahead/behind to a certain address". So, if some_place is memory address $2000 then this instruction is translated to "Jump to $2000".

This is a subtle but important difference. It's important because some assemblers, like the AS9 assembler I use don't let you mix label destinations for these two kinds of jumps. So, if you have code that looks like this:

```
.  ;Some code here
.
  BNE some_place
.
.  ;Some code here
.
.
  JMP some_place
.
.  ;Some code here
.
some_place:
```

You will get a "phasing error" even though it looks perfectly reasonable! Instead, you need to do it like this:

```
.  ;Some code here
.
  BNE some_place
.
.  ;Some code here
.
.
  BRA some_place
.
.  ;Some code here
.
some_place:
```

Believe you me, it took a long, long time and a lot of frustration before I realised what this problem was! The bottom line is don't mix relative and absolute jumps to the same destination!

See, I've already just saved you a pile of headaches!

Loops

As mentioned, one of the most important functions of computers is their ability to do repetitive tasks and to do them over and over and over again without ever getting tired.

That's really what looping is really all about. Let's say you wanted the computer to do something really simple over and over again. Let's say we want it to do this:

```
LDA $C880
STA $C881
```

Maybe 5 times. Why would we ever want the computer to do this ten times? Well, honestly we probably wouldn't but I want to keep this example simple because it's not what we are doing over and over that's important but HOW we are going to do it over and over. We could just as easily say we want to do 100 lines of code 5 times, but, that 100 lines would just be distracting and possibly confusion so we'll stick to something really simple like:

```
LDA $C880
STA $C881
```

And do it 5 times. Well, the simplest and most obvious way is to just do this:

```
LDA $C880
STA $C881
LDA $C880
STA $C881
LDA $C880
STA $C881
LDA $C880
STA $C881
LDA $C880
STA $C881
```

That will work fine and in fact there are times when you will want to do something similar and do it just this way (because this is actually a little faster than the other way I'm about to show you). But usually only if you are doing something fairly simple and not very many times.

But what if I told you to do this 50 times? Or what if we were working on that complicated 100 lines of code instead of just 2 simple lines? Suddenly, this because a HUGE piece of code. Hundred and hundreds of lines!

Of course, there's a better way... If there wasn't, you wouldn't be reading this section!

What we do is a loop:

```
loop_variable     EQU   $C880   ;Create a variable for looping


  LDA #$05            ;Initialise our loop
  STA loop_variable   ;variable with a value of 5

loop_start:           ;This is the label at the start of the loop
  LDA $C880           ;These are the instructions we want to repeat 5 times
  STA $C881           ;These are the instructions we want to repeat 5 times
  DEC loop_variable   ;Subtract the loop_variable by 1
  BNE loop_start      ;If the loop variable is not Zero, jump to loop_start
```

And that's it. This code will execute our:

```
  LDA $C880
  STA $C881
```

Instructions 5 times. In essence, what we are doing is creating a variable called loop_variable. The 6809 uses this variable to keep track of how many times it has executed the loop. Just the same as if you were keeping track of how many times you were going to do something over and over by counting as you go.

As we've discussed several times before,

```
loop_variable     EQU   $C880   ;Create a variable for looping
```

Creates a label pointing to memory address $C880. We will later use this memeory address as our variable. The first two lines of actual code:

```
  LDA #$05            ;Initialise our loop
  STA loop_variable   ;variable with a value of 5
```

Put a value of 5 into this memory location. First we LoaD #$05 into register A and then we STore register A into loop_variable. loop_variable of course points to $C880. We go through register A because you can't put a value directly into a memory location we have to go first through one of the registers like A, B, D, X or Y... Just like we did when we wanted to set register DP back when we were talking about Direct addressing.

Next we have the label which is the start of the loop:

```
loop_start:           ;This is the label at the start of the loop
```

This is the place in the code we want to come back to when we are going to execute the next bit over and over again. The code that we are going to do over and over again being:

```
  LDA $C880           ;These are the instructions we want to repeat 5 times
  STA $C881           ;These are the instructions we want to repeat 5 times
```

Which are just the instructions from before that we want to execute a total of 5 times.

Now things start to get interesting. First we,

```
  DEC loop_variable   ;Subtract the loop_variable by 1
```

This decreases the value in loop_variable by 1. So the first time through, loop_variable will have a value of 5 (We set it to five just on the previous page there). When we execute this instruction, loop_variable is reduced to 4.

Next, we:

```
  BNE loop_start      ;If the loop variable is not Zero, jump to loop_start
```

This checks to see if the Zero bit of the CC register has been set. The previous instruct, the DEC will set the Zero bit if the result of the decrement was a Zero. Otherwise it will clear it.

The first time through the loop, loop_variable will have been decreased from 5 to 4. That's not zero, so the Zero bit is cleared. This means that the BNE instruction causes the program to jump to loop_start.

And we go through it again, back to:

```
loop_start:           ;This is the label at the start of the loop
  LDA $C880           ;These are the instructions we want to repeat 5 times
  STA $C881           ;These are the instructions we want to repeat 5 times
```

And so we execute our instructions a second time. And then a third, a fourth and a fifth time.

However, eventually we will get to the:

```
  DEC loop_variable   ;Subtract the loop_variable by 1
```

Instruction when loop_variable is 1 (Having previously been decreased from 5 to 4, then 4 to 3, then 3 to 2, then 2 to 1), so this time when we decreased loop_counter the result is 0. This sets the Zero flag. So when we get to

```
  BNE loop_start      ;If the loop variable is not Zero, jump to loop_start
```

The Zero flag IS set! So this instruction does nothing, it does NOT branch us back to loop_start. Instead, it just continues on to the next instruction and the rest of our program. The loop is complete.

The thing about looping is that it's very compact and very easy to a something A LOT of times. For example, if we wanted to make this loop execute 100 times instead of 5, all we'd have to do is:

```
loop_variable     EQU   $C880   ;Create a variable for looping


  LDA #100            ;Initialise our loop
  STA loop_variable   ;variable with a value of 100

loop_start:           ;This is the label at the start of the loop
  LDA $C880           ;These are the instructions we want to repeat 5 times
  STA $C881           ;These are the instructions we want to repeat 5 times
  DEC loop_variable   ;Subtract the loop_variable by 1
  BNE loop_start      ;If the loop variable is not Zero, jump to loop_start
```

And poof, the code will happily execute itself 100 times!

Pretty simple, eh? I hope you realise you just learned one of the most important concepts in computer programming...

Basic Math

Some Bios Routines

The Program Skeleton

This is about the simplest, most basic program you can write for the Vectrex which will actually assemble and run:

```
        ORG     $0000

; Magic Init Block

        FCB     $67,$20
        FCC     "GCE XXXX"
        FCB     $80
        FDB     music
        FDB     $f850
        FDB     $30b8
        FCC     "SIMPLE"
        FCB     $80,$0
start:
        bra start
music:
        FDB     $fee8
        FDB     $feb6
        FCB     $0,$80
        FCB     $0,$80
```

In fact, it's so simple it doesn't really do anything at all. But it will work and it shows you all the parts which MUST be included in a Vectrex program in order for it to work.

First is the line:

```
        ORG     $0000
```

This line tells the assembler to start assembling code at memory location $0000. For the Vectrex, ALL code begins at $0000. However, on many other computers code can start in a variety of places. Or, you may want to assemble your code in a bunch of little pieces and then join them together later. As the assembler is multi-purpose it includes the option to begin assembly anywhere, but for our purposes with the Vectrex we will ALWAYS use:

```
        ORG     $0000
```

Next is the "Magic Init Block"

```
; Magic Init Block
```

```
        FCB     $67,$20
        FCC     "GCE XXXX"
        FCB     $80
        FDB     music
        FDB     $f850
        FDB     $30b8
        FCC     "SIMPLE"
        FCB     $80,$0
```

This actually contains all the information the Vectrex needs to compose the screen you see at startup. "GCE XXXX" is the copyright info. "SIMPLE" is the name of the program/game. music is the starting address of the music data that is played when the game first starts up. In this case, I've set it to some music data included below, but there are also several pieces of music built into the BIOS that you can use. Eventually, you may want to write your own music.

The actual program is:

```
start:
        bra start
```

Which just tells the 6809 to endlessly loop and do nothing. Like I said, a very simple program.

Finally is the music data:

```
music:
        FDB     $fee8
        FDB     $feb6
        FCB     $0,$80
        FCB     $0,$80
```

I won't go into the format for music here, but this code just tells the Vectrex not to play any music.

And that's it. That's a Vectrex program, though granted it doesn't do much of anything.

Your First Program
==================

So let's do something.

This is a simple program which draw a line. You can actually clip this out and assemble it and run it on the Vectrex emulator:

```
waitrecal               EQU     $f192
move_pen7f_to_d         EQU     $f2fc
intensity_to_A          EQU     $f2ab
draw_to_d               EQU     $f3df

        ORG     $0000
```

```
; Magic Init Block

        FCB     $67,$20
        FCC     "GCE XXXX"
        FCB     $80
        FDB     music
        FDB     $f850
        FDB     $30b8
        FCC     "LINE"
        FCB     $80,$0
start:

;Draws a big line from the middle of screen to right edge.
draw_line:
        jsr   waitrecal                 ;Reset the CRT
        lda   #00                       ;Get y
        ldb   #00                       ;Get x
        jsr   move_pen7f_to_d           ;go to (x,y)
        lda   #$7f                      ;Get the Intensity
        jsr   intensity_to_A            ;Set intensity
        lda   #00                       ;Get y
        ldb   #127                      ;Get x
        jsr   draw_to_d                 ;draw a line to (x,y)
        bra   draw_line


music:
        FDB     $fee8
        FDB     $feb6
        FCB     $0,$80
        FCB     $0,$80
```

The first bit of the program should come as no surprise to you. It's simply several labels describing the BIOS routines we will be using. This is a simple program, so there are no variables or anything, just these labels:

```
waitrecal               EQU     $f192
move_pen7f_to_d         EQU     $f2fc
intensity_to_A          EQU     $f2ab
draw_to_d               EQU     $f3df
```

Next comes the "Magic Init Block" which I won't reproduce here and then the program begins:

```
draw_line:
        jsr   waitrecal                 ;Reset the CRT
```

We call waitrecal first because it resets everything, moves the pen back to (0,0) and waits until the timer clears itself so our timing is right.

Next, we move the pen:

```
        lda   #00                       ;Get y
        ldb   #00                        ;Get x
        jsr   move_pen7f_to_d            ;go to (x,y)
```

In this case, we don't really move the pen anywhere, but you still have to call a move_pen routine before you draw or it won't work. If you want to get ambitious, try changing the #$00 values and you'll see the line start in different places on the screen.

Next, we set the brightness of the lines to maximum:

```
        lda   #$7f                       ;Get the Intensity
        jsr   intensity_to_A             ;Set intensity
```

Again, if you want to get ambitious, try using #$3f or #$1f instead of #$7f for the brightness.

Finally, having set all this up, we draw the line:

```
        lda   #00                        ;Get y
        ldb   #127                       ;Get x
        jsr   draw_to_d                  ;draw a line to (x,y)
```

This draws a horizontal line of length 127. Again, try different values instead of #00 or #127 and you'll draw lines of different lengths.

Of course, since this is a Vectrex, we have to go back to the start and keep redrawinf the screen constantly or it will fade and we do this with a jump back to the start of the program:

```
        bra   draw_line
```

And finally the music description which just tells the Vectrex not to play any music.

Now, play around with this, move the line around, make the line longer and shorter. And whatever you do, don't worry about hurting anything! Even if you screw it up, it would be next to impossible for you to damage anything.
If the assembler gives you an error check that you didn't change the wrong thing. If the emulator does something wierd, well it does something wierd. No bid deal there!

You'd be surprised how often I've had the emulator do something really wierd and unexpected on me because I messed something up!

Don't ever be afraid to make a mistake! You'll never learn unless you actually try, unless you actually mess around with and change code.

Indexed Addressing and Tables
=============================

Indexed is the most complicated addressing mode, but also the most powerful. It is used to determine an Address by combing or referencing other values and is very useful in manipulating tables or large chunks of data.

For example, lets say you have a table of values. Maybe the heights of the highest volcanos in the world. The table starts at memory address $C880.

If you want to look up the height of the first volcano, you would simply:

```
LDA $C880
```

If you wanted the height of the fifth volcano, you would:

```
LDA $C884
```

Which is pretty straightforward. Unfortunately, it's also very limited. Suppose you don't know which volcano you want to reference? Instead, you want whoever's using your program to pick which volcano's height to get?

Well, you could do it the hard way, and have a the person input which volcano and then have a separate section of code for each possible selection and execute it depending upon what he selected. So you would have a section which did a LDA $C880 and one which did a $C884. But what if you have 100 volcanoes? That's suddenly an awful lot of code.

So instead, you use Indexing. Instead, you'd do this:

```
volcano   EQU  $C900
.
.
LDA #$05
STA volcano
.
.
LDX #$C880      ;Set register X to #$C880
LDA volcano,X   ;LoaD register A with the value stored in the memory location
                ;which is the sum of the value in register X and the value
                ;in memory location volcano
```

So, volcano is a label, a variable previously defined as being which volcano the user wants to look up. As you've probably figured out you just set the register X to the start of the table (in this case $C880) and then use volcano as a reference point from the start of that table. In this case, register A will wind up with whatever is in memory location $C885 ($C880+$05).

This kind of Indexing can be done using the registers X, Y, U and S. So you can easily be referencing several tables at once.

There are a couple other types of Indexing modes available to you. Instead of a memory address, you can use a register. For example:

```
LDX  #$C880
LDB  #$02
LDA  B,Y
```

Will LoaD the register A with whatever is in memory location $C880 ($C880+$02)

Another Indexed mode automatically increments or decrements the register (X, Y, U or S) being used.

For example:

```
LDX #$C880
```

```
LDA ,X+
```

Will LoaD register A with the contents of memory location $C880. Then, the value of X is increased by 1. This is usefull in working through an entire table in a loop. For example, if you wanted to get an average height of all the volcanoes in your list, you could use this, along with a loop, to quickly read through your entire table.

Similarily,

```
LDA ,-X
```

Will do the same, but decrease X by 1.

Also, because you will often be dealing with two byte values (ie: memory addresses) you can do either:

```
LDA ,X++
```

or

```
LDA ,--X
```

Which increments/decrements X by 2 instead of just one. This way you can easily search through a table of wither large values or memory addresses.

Finally, one last Indexed mode is as follows:

```
LDA [$C880]
```

This LoaDs the register A with the value in the memory location that is pointed to by $C880 and $C881. So, if $C880 has a value of $E000 then register A is LoaDed with the value contained in memory address $E000.

Of course, you are probably wondering why you would ever use all these complicated Indexed modes. Well, there's really good reasons to.

Let's say you're writing a game, and each game has a different number of aliens that fly around and shoot at the player. Now, you could have a section of code for each level that specifically tell how many ships for each level, something like:

```
number_aliens  EQU $C880
.
.
.
level1:
     LDA #10
     STA #number_aliens
.
.
.
level2:
     LDA #12
     STA #number_aliens
.
```

```
.
.
level7:
     LDA #20
     STA #number_aliens
```

And so on. But as you can see, this kind of thing adds up pretty quickly. What's more, this is a VERY simplified example. Suppose you have all kinds of different data for each level. Like how fast the aliens move, how often the shoot, how aggressively the attack. And so on. You can see that this could take an awful lot of code. Particularly when instead you could just:

```
number_aliens  EQU $C880  ;Create a variable to hold the number of aliens
level          EQU $C881  ;Create a variable to hold the level
.
.
LDA  #$01
STA  level                ;Set the level to 1
.
.
LDX #number_aliens_table  ;Setup the start of the table
LDA level,X               ;Get the number of aliens and put it in A
.
.
.
number_aliens_table: fcb #00,#10,#12,#14,#16,#18,#20   ;The table
```

The only thing you haven't seen yet is fcb. All fcb does is tell the assembler to put this ACTUAL VALUE in the code and not interpret it as an instruction. So the values of #00,#10,#12,#14,#16,#18 and #20 are put here. That way you can put static data right into the code. Data like graphics, sound, or volcano heights!

As you can see, this is A LOT simpler than before. We can execute the middle portion of code, the:

```
LDX #number_aliens_table
LDA level,x
```

Any time we want to set up a level, all we need is to do is set the the variable and execute this code. This is a lot simpler and uses a lot less space than doing separate code for each level.