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".

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?

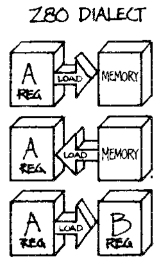
The 6809's data bus is 8 bits wide.

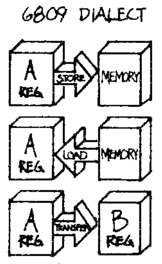
* What is an addressing mode?

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

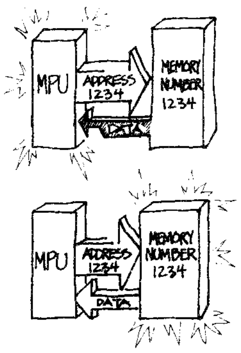
* Where does the processor get its data?

From memory.





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

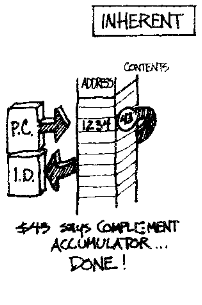


You don't need to know much about this electrical process; for programming purposes, you take it on faith that the machine's designers have organized the connections properly so that when your program wants information from memory location **$1234**, for example, memory location **$1234** will respond 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?

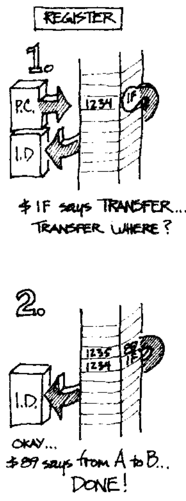
An addressing mode in which the two bytes following the opcode are the address of the data to be used to complete the instruction.

* 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).

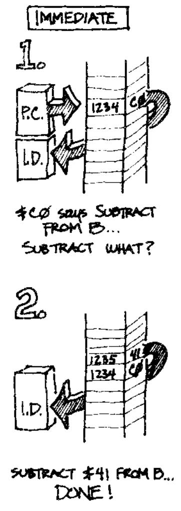


Don't be confused by the MC6809E data booklet; Register Addressing is easy. The data booklet first suggests that Register Addressing can be thought of as either distinct from or the same as Inherent Addressing. I leave that up to you, because the MC6809E data booklet can't make up its mind, either. The 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.

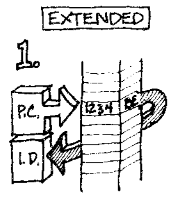
If you wish, review Inherent and Register Addressing in your documentation. For review, turn the tape off *now*.

Immediate Addressing

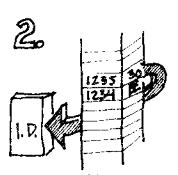


Immediate Addressing is very transparent. The data to be used is found at the address immediately following the instruction itself, in program order. Among examples you have used already are Load X Register with value **$3000** (written **LDX #$3000**), and Subtract the value **$41** from B Accumulator (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!"

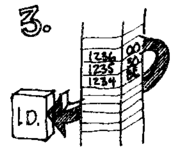
Extended Addressing



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.



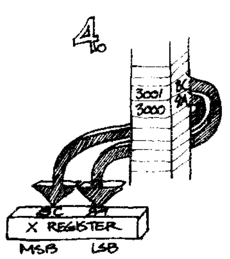
Did a question come to mind? How can the 16-bit X register load the 8-bit data at address **$3000**? Since the data at address **$3000** is only an 8-bit word, and since the X register requires 16 bits, the instruction decoder sees to it that the process is completed correctly. The information loaded 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

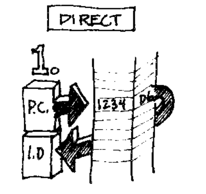
Register; as described by the postbyte.

* CMPA $789A

Extended; at address $789A.

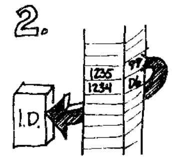
of bytes taken from sequential memory locations matches the size of the register requesting the data.

Direct Addressing



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

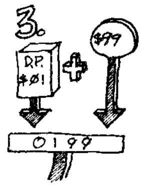
Direct Addressing obtains data for program use with great speed and memory economy. It depends on the organization of memory into pages. A "page" is a specific term in assembly language programming, meaning those 256 contiguous bytes of memory whose most-significant-byte is in common. For example, page **$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.



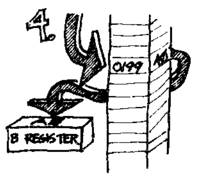
$99 is LSB... LOAD FROM DIRECT PAGE BYTE $99...

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.

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.

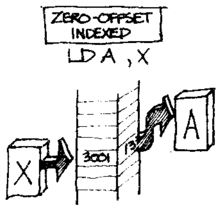
* If the X register is $3000 and the A register is $41, where does the instruction LDB ,X find its data?

At address $3000.

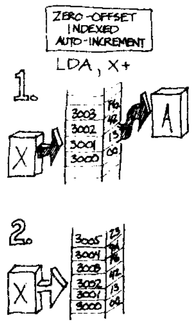
* What kind of addressing is this?

Zero-offset indexed.

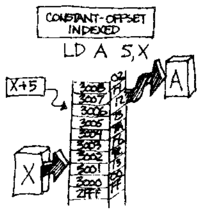
register. You've seen the usefulness of this method in that little code encryption program. The X register was set to the memory location at the start of the encryption table, and the offset added to pick your way through the table was in the B register.



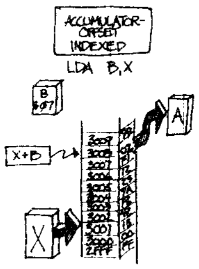
These Indexed Addressing methods are called Zero-Offset Indexed, Constant-Offset Indexed, and Accumulator-Offset Indexed. More jargon. Zero-Offset Indexed means that what you see is what you get; the value in the register is the address of the data. Constant-Offset Indexed means that you're using a fixed 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 $3000 and the A register is $41, where does the instruction LDB $9C,X find its data?

At address $309C.

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

At address $3041.

* 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.

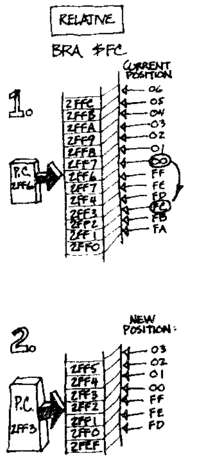
If you've been reviewing this lesson, you probably have an idea that indexed addressing is very flexible and not nearly so difficult as the jargon suggests. And, if you've had a glance at your MC6809E data booklet, then you know there's quite a bit more to the subtlety of indexed addressing. Even so,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

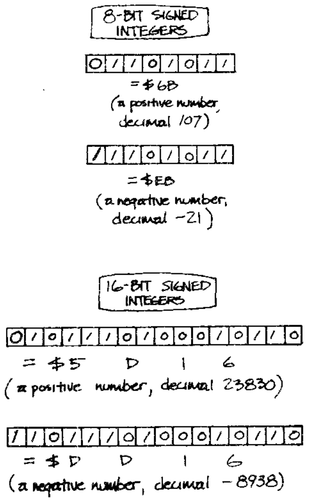
Relative Addressing is a good term, one of the best pieces of jargon you'll encounter. When Relative Addressing is employed, the data needed to complete an instruction is found at a location in memory *relative* to the present position of the Program Counter. Specifically, Relative Addressing is used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

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.


RELATIVE
BRA $FC
CURRENT POSITION
1.
P.C. 2FF6
2FFC 2FFB 2FFA 2FF9 2FF8 2FF7 2FF6 2FF7 2FF4 2FF3 2FF2 2FF1 2FF0
06 05 04 03 02 01 00 FF FE FD FC FB FA
NEW POSITION:
2.
P.C. 2FF3
2FF5 2FF4 2FF3 2FF2 2FF1 2FF0 2FEF
03 02 01 00 FF FE FD

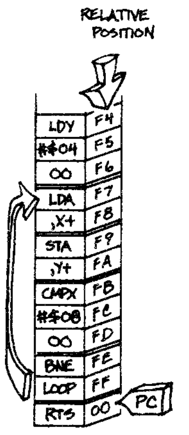


Turn to your documentation. That program is printed with this text; this time, though, the hex code appears with it.

```
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.

Long and short relative

* 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
```

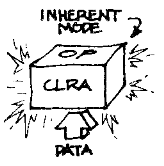
But **$F7** is negative, so the most signficant byte of the address (**$41**) is decremented to **$40**. The result is **$4007**. Glance at the listing. **$4007** is the address where you will find the label "LOOP".

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.

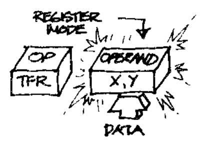
Relative Addressing, then, is unique in that the operand does not provide either an immediate value or a specific address to the processor. Rather, it provides a value which can be used to calculate a specific address in relationship to the present position in the program.

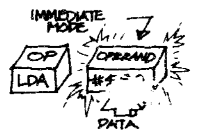
Time to summarize. There are seven major ways your program can obtain the information it needs. These are called the addressing modes.

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

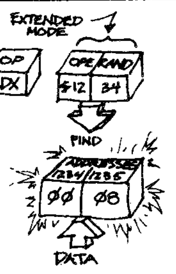


2. The information can deal with internal 6809 registers. This is Register Addressing. **TFR X,Y** (Transfer X Register to Y Register) is an example of Register Addressing.

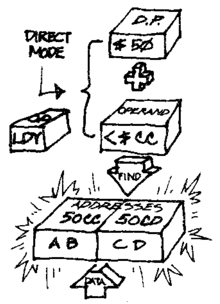




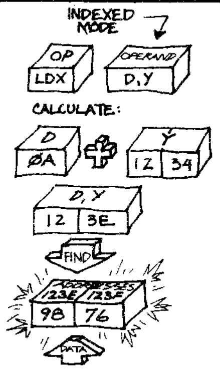
3. The information can be present immediately following the instruction itself. This is Immediate Addressing. **LDA #$80** (Load A Accumulator with the value **$80**) is an example of Immediate Addressing.



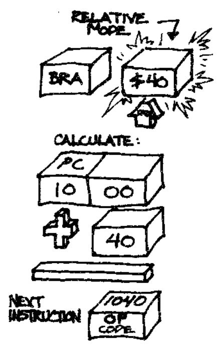
4. The information can take the form of a memory address where data can be found. This is Extended Addressing.
LDX $1234 (Load X Register with the information at Address **$1234**) is an example of Extended Addressing.



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.