CHAPTER 5

BASIC TO MACHINE LANGUAGE

- What Is Machine Language?
- How Do You Write Machine Language Programs?
- Hexadecimal Notation
- Addressing Modes
- Indexing
- Subroutines
- Useful Tips for the Beginner
- Approaching a Large Task
- MCS6510 Microprocessor Instruction Set
- Memory Management on the Commodore 64
- The KERNAL
- KERNAL Power-Up Activities
- Using Machine Language From BASIC
- Commodore 64 Memory Map

WHAT IS MACHINE LANGUAGE?

At the heart of every microcomputer, is a central microprocessor. It's a very special microchip which is the "brain" of the computer. The Commodore 64 is no exception. Every microprocessor understands its own language of instructions. These instructions are called machine language instructions. To put it more precisely, machine language is the ONLY programming language that your Commodore 64 understands. It is the NATIVE language of the machine.

If machine language is the only language that the Commodore 64 understands, then how does it understand the CBM BASIC programming language? CBM BASIC is NOT the machine language of the Commodore 64. What, then, makes the Commodore 64 understand CBM BASIC instructions like PRINT and GOTO?

To answer this question, you must first see what happens inside your Commodore 64. Apart from the microprocessor which is the brain of the Commodore 64, there is a machine language program which is stored in a special type of memory so that it can't be changed. And, more importantly, it does not disappear when the Commodore 64 is turned off, unlike a program that you may have written. This machine language program is called the OPERATING SYSTEM of the Commodore 64. Your Commodore 64 knows what to do when it's turned on because its OPERATING SYSTEM (program) is automatically "RUN."

The OPERATING SYSTEM is in charge of "organizing" all the memory in your machine for various tasks. It also looks at what characters you type on the keyboard and puts them onto the screen, plus a whole number of other functions. The OPERATING SYSTEM can be thought of as the "intelligence and personality" of the Commodore 64 (or any computer for that matter). So when you turn on your Commodore 64, the OPERATING SYSTEM takes control of your machine, and after it has done its housework, it then says:

```
READY.
■
```

The OPERATING SYSTEM of the Commodore 64 then allows you to type on the keyboard, and use the built-in SCREEN EDITOR on the Commodore 64. The SCREEN EDITOR allows you to move the cursor, DELete, INSert, etc., and is, in fact, only one part of the operating system that is built in for your convenience.

All of the commands that are available in CBM BASIC are simply recognized by another huge machine language program built into your Commodore 64. This huge program "RUNs" the appropriate piece of machine language depending on which CBM BASIC command is being executed. This program is called the BASIC INTERPRETER, because it interprets each command, one by one, unless it encounters a command it does not understand, and then the familiar message appears:

```
?SYNTAX ERROR

READY.
■
```

WHAT DOES MACHINE CODE LOOK LIKE?

You should be familiar with the PEEK and POKE commands in the CBM BASIC language for changing memory locations. You've probably used them for graphics on the screen, and for sound effects. Each memory location has its own number which identifies it. This number is known as the "address" of a memory location. If you imagine the memory in the Commodore 64 as a street of buildings, then the number on each door is, of course, the address. Now let's look at which parts of the street are used for what purposes.

SIMPLE MEMORY MAP OF THE COMMODORE 64

ADDRESS	DESCRIPTION
0 & 1	—6510 Registers.
2	—Start of memory.
up to: 1023	—Memory used by the operating system.
1024 up to: 2039	—Screen memory.
2040 up to: 2047	—SPRITE pointers.
2048 up to: 40959	—This is YOUR memory. This is where your BASIC or machine language programs, or both, are stored.
40960 up to: 49151	—8K CBM BASIC Interpreter.
49152 up to: 53247	—Special programs RAM area.
53248 up to: 53294	—VIC-II.
54272 up to: 55295	—SID Registers.
55296 up to: 56296	—Color RAM.
56320 up to: 57343	—I/O Registers. (6526's)
57344 up to: 65535	—8K CBM KERNAL Operating System.

If you don't understand what the description of each part of memory means right now, this will become clear from other parts of this manual.

Machine language programs consist of instructions which may or may not have operands (parameters) associated with them. Each instruction takes up one memory location, and any operand is contained in one or two locations following the instruction.

In your BASIC programs, words like PRINT and GOTO do, in fact, only take up one memory location, rather than one for each character of the word. The contents of the location that represents a particular BASIC keyword is called a *token*. In machine language, there are different tokens for different instructions, which also take up just one byte (memory location=byte).

Machine language instructions are very simple. Therefore, each individual instruction cannot achieve a great deal. Machine language instructions either change the contents of a memory location, or change one of the internal registers (special storage locations) inside the microprocessor. The internal registers form the very basis of machine language.

THE REGISTERS INSIDE THE 6510 MICROPROCESSOR

THE ACCUMULATOR

This is THE most important register in the microprocessor. Various machine language instructions allow you to copy the contents of a memory location into the accumulator, copy the contents of the accumulator into a memory location, modify the contents of the accumulator or some other register directly, without affecting any memory. And the accumulator is the only register that has instructions for performing math.

THE X INDEX REGISTER

This is a very important register. There are instructions for nearly all of the transformations you can make to the accumulator. But there are other instructions for things that only the X register can do. Various machine language instructions allow you to copy the contents of a memory location into the X register, copy the contents of the X register into a memory location, and modify the contents of the X, or some other register directly.

THE Y INDEX REGISTER

This is a very important register. There are instructions for nearly all of the transformations you can make to the accumulator, and the X register. But there are other instructions for things that only the Y register can do. Various machine language instructions allow you to copy the contents of a memory location into the Y register, copy the contents of the Y register into a memory location, and modify the contents of the Y, or some other register directly.

THE STATUS REGISTER

This register consists of eight "flags" (a flag = something that indicates whether something has, or has not occurred).

THE PROGRAM COUNTER

This contains the address of the current machine language instruction being executed. Since the operating system is always "RUN"ning in the Commodore 64 (or, for that matter, any computer), the program counter is always changing. It could only be stopped by halting the microprocessor in some way.

THE STACK POINTER

This register contains the location of the first empty place on the stack. The stack is used for temporary storage by machine language programs, and by the computer.

THE INPUT/OUTPUT PORT

This register appears at memory locations 0 (for the DATA DIRECTION REGISTER) and 1 (for the actual PORT). It is an 8-bit input/output port. On the Commodore 64 this register is used for memory management, to allow the chip to control more than 64K of RAM and ROM memory.

The details of these registers are not given here. They are explained as the principles needed to explain them are explained.

HOW DO YOU WRITE MACHINE LANGUAGE PROGRAMS?

Since machine language programs reside in memory, and there is no facility in your Commodore 64 for writing and editing machine language programs, you must use either a program to do this, or write for yourself a BASIC program that "allows" you to write machine language.

The most common methods used to write machine language programs are *assembler* progams. These packages allow you to write machine language instructions in a standardized *mnemonic* format, which makes the machine language program a lot more readable than a stream of numbers! Let's review: A program that allows you to write machine language programs in mnemonic format is called an *assembler*. Incidentally, a program that displays a machine language program in mnemonic format is called a *disassembler*. Available for your Commodore 64 is a machine language monitor cartridge (with assembler/ disassembler, etc.) made by Commodore:

64MON

The 64MON cartridge available from your local dealer, is a program that allows you to escape from the world of CBM BASIC, into the land of machine language. It can display the contents of the internal registers in the 6510 microprocessor, and it allows you to display portions of memory, and change them on the screen, using the screen editor. It also has a built-in assembler and disassembler, as well as many other features that allow you to write and edit machine language programs easily. You don't HAVE to use an assembler to write machine language, but the task is considerably easier with it. If you wish to write machine language programs, it is strongly suggested that you purchase an assembler of some sort. Without an assembler you will probably have to "POKE" the machine language program into memory, which is totally unadvisable. This manual will give its examples in the format that 64MON uses, from now on. Nearly all assembler formats are the same, therefore the machine language examples shown will almost certainly be compatible with any assembler. But before explaining any of the other features of 64MON, the hexadecimal numbering system must be explained.

HEXADECIMAL NOTATION

Hexadecimal notation is used by most machine language programmers when they talk about a number or address in a machine language program.

Some assemblers let you refer to addresses and numbers in decimal (base 10), binary (base 2), or even octal (base 8) as well as hexadecimal (base 16) (or just "hex" as most people say). These assemblers do the conversions for you.

Hexadecimal probably seems a little hard to grasp at first, but like most things, it won't take long to master with practice.

By looking at decimal (base 10) numbers, you can see that each digit falls somewhere in the range between zero and a number equal to the base less one (e.g., 9). THIS IS TRUE OF ALL NUMBER BASES. Binary (base 2) numbers have digits ranging from zero to one (which is one less than the base). Similarly, hexadecimal numbers should have digits ranging from zero to fifteen, but we do not have any single digit figures for the numbers ten to fifteen, so the first six letters of the alphabet are used instead:

DECIMAL	HEXADECIMAL	BINARY
0	0	00000000
1	1	00000001
2	2	00000010
3	3	00000011
4	4	00000100
5	5	00000101
6	6	00000110
7	7	00000111
8	8	00001000
9	9	00001001
10	A	00001010
11	B	00001011
12	C	00001100
13	D	00001101
14	E	00001110
15	F	00001111
16	10	00010000

Let's look at it another way; here's an example of how a base 10 (decimal number) is constructed:

Base raised by increasing powers: . . .	10^3	10^2	10^1	10^0
Equals:	1000	100	10	1
Consider 4569 (base 10)	4	5	6	9

$$=(4\times 1000)+(5\times 100)+(6\times 10)+9$$

Now look at an example of how a base 16 (hexadecimal number) is constructed:

Base raised by increasing powers: . . .	16^3	16^2	16^1	16^0
Equals:	4096	256	16	1
Consider 11D9 (base 16)	1	1	D	9

$$=1\times 4096+1\times 256+13\times 16+9$$

Therefore, 4569 (base 10) = 11D9 (base 16)

The range for addressable memory locations is 0–65535 (as was stated earlier). This range is therefore 0–FFFF in hexadecimal notation.

Usually hexadecimal numbers are prefixed with a dollar sign ($). This is to distinguish them from decimal numbers. Let's look at some "hex" numbers, using 64MON, by displaying the contents of some memory by typing:

```
SYS 8*4096          (or SYS 12*4096)
B*
    PC SR AC XR YR SP
.; 0401 32 04 5E 00 F6 (these may be different)
```

Then if you type in:

.M 0000 0020 (and press RETURN).

you will see rows of 9 hex numbers. The first 4-digit number is the address of the first byte of memory being shown in that row, and the other eight numbers are the actual contents of the memory locations beginning at that start address.

You should really try to learn to "think" in hexadecimal. It's not too difficult, because you don't have to think about converting it back into decimal. For example, if you said that a particular value is stored at $14ED instead of 5357, it shouldn't make any difference.

YOUR FIRST MACHINE LANGUAGE INSTRUCTION

LDA — LOAD THE ACCUMULATOR

In 6510 assembly language, mnemonics are always three characters. LDA represents "load accumulator with . . . ," and what the accumulator should be loaded with is decided by the parameter(s) associated with that instruction. The assembler knows which token is represented by each mnemonic, and when it "assembles" an instruction, it simply puts into memory (at whatever address has been specified), the token, and what parameters, are given. Some assemblers give error messages, or warnings when you try to assemble something that either the assembler, or the 6510 microprocessor, cannot do.

If you put a "#" symbol in front of the parameter associated with the instruction, this means that you want the register specified in the instruction to be loaded with the "va ue" after the "#." For example:

```
LDA #S05
```

This instruction will put $05 (decimal 5) into the accumulator register. The assembler will put into the specified address for this instruction, $A9 (which is the token for this particular instruction, in this mode), and it will put $05 into the next location after the location containing the instruction ($A9).

If the parameter to be used by an instruction has "#" before it; i.e., the parameter is a "value," rather than the contents of a memory location, or another register, the instruction is said to be in the "immediate" mode. To put this into perspective, let's compare this with another mode:

If you want to put the contents of memory location $102E into the accumulator, you're using the "absolute" mode of instruction:

```
LDA $102E
```

The assembler can distinguish between the two different modes because the latter does not have a "#" before the parameter. The 6510 microprocessor can distinguish between the immediate mode, and the absolute mode of the LDA instruction, because they have slightly different tokens. LDA (immediate) has $A9 as its token, and LDA (absolute), has $AD as its token.

The mnemonic representing an instruction usually implies what it does. For instance, if we consider another instruction, LDX, what do you think this does?

If you said "load the X register with . . . ," go to the top of the class. If you didn't, then don't worry, learning machine language does take patience, and cannot be learned in a day.

The various internal registers can be thought of as special memory locations, because they too can hold one byte of information. It is not necessary for us to explain the binary numbering system (base 2) since it follows the same rules as outlined for hexadecimal and decimal outlined previously, but one "bit" is one binary digit and eight bits make up one byte! This means that the maximum number that can be contained in a byte is the largest number that an eight digit binary number can be. This number is 11111111 (binary), which equals $FF (hexadecimal), which equals 255 (decimal). You have probably wondered why only numbers from zero to 255 could be put into a memory location. If you try POKE 7680,260 (which is a BASIC statement that "says": "Put the number two hundred and sixty, into memory location seven thousand, six hundred and eighty," the BASIC interpreter knows that only numbers 0 – 255 can be put in a memory location, and your Commodore 64 will reply with:

```
?ILLEGAL QUANTITY ERROR

READY.
■
```

If the limit of one byte is $FF (hex), how is the address parameter in the absolute instruction "LDA $102E" expressed in memory? It's expressed in two bytes (it won't fit into one, of course). The lower (rightmost) two digits of the hexadecimal address form the "low byte" of the address, and the upper (leftmost) two digits form the "high byte."

The 6510 requires any address to be specified with its low byte first, and then the high byte. This means that the instruction "LDA $102E" is represented in memory by the three consecutive values:

```
$AD, $2E, $10
```

Now all you need to know is one more instruction and then you can write your first program. That instruction is BRK. For a full explanation of this instruction, refer to *M.O.S. 6502 Programming Manual*. But right now, you can think of it as the END instruction in machine language.

If we write a program with 64MON and put the BRK instruction at the end, then when the program is executed, it will return to 64MON when it is finished. This might not happen if there is a mistake in your program, or the BRK instruction is never reached (just like an END statement in BASIC may never get executed). This means that if the Commodore 64 didn't have a STOP key, you wouldn't be able to abort your BASIC programs!

WRITING YOUR FIRST PROGRAM

If you've used the POKE statement in BASIC to put characters onto the screen, you're aware that the character codes for POKEing are different from CBM ASCII character values. For example, if you enter:

PRINT ASC("A") (and press RETURN)

the Commodore 64 will respond with:

```
65

READY.
■
```

However, to put an "A" onto the screen by POKEing, the code is 1, enter:

SHIFT CLR/HOME to clear the screen

POKE 1024,1:POKE 55296,14 (and RETURN) (1024 is the start of screen memory)

The "P" in the POKE statement should now be an "A."

Now let's try this in machine language. Type the following in 64MON: (Your cursor should be flashing alongside a "." right now.)

.A 1400 LDA #$01 (and press RETURN)

The Commodore 64 will prompt you with:

```
.A 1400 A9 01       LDA #$01
.A 1402 ■
```

Type:

```
.A 1402 STA $0400
```

(The STA instruction stores the contents of the accumulator in a specified memory location.)
The Commodore 64 will prompt you with:

```
.A 1405 ■
```

Now type in:

```
.A 1405 LDA #$0E
.A 1407 STA $D800
.A 140A BRK
```

Clear the screen, and type:

```
G 1400
```

The G should turn into an "A" if you've done everything correctly.

You have now written your first machine language program. Its purpose is to store one character ("A") at the first location in the screen memory. Having achieved this, we must now explore some of the other instructions, and principles.

ADDRESSING MODES

ZERO PAGE

As shown earlier, absolute addresses are expressed in terms of a high and a low order byte. The high order byte is often referred to as the *page* of memory. For example, the address $1637 is in page $16 (22), and $0277 is in page $02 (2). There is, however, a special mode of addressing known as *zero page* addressing and is, as the name implies, associated with the addressing of memory locations in page zero. These addresses, therefore, ALWAYS have a high order byte of zero. The zero page mode of addressing only expects one byte to describe the address, rather than two when using an absolute address. The zero page addressing mode tells the microprocessor to assume that the high order address is zero. Therefore zero page addressing can reference memory locations whose addresses are between $0000 and $00FF. This may not seem too important at the moment, but you'll need the principles of zero page addressing soon.

THE STACK

The 6510 microprocessor has what is known as a *stack*. This is used by both the programmer and the microprocessor to temporarily remember things, and to remember, for example, an order of events. The GOSUB statement in BASIC, which allows the programmer to call a *subroutine*, must remember where it is being called from, so that when the RETURN statement is executed in the subroutine, the BASIC interpreter "knows" where to go back to continue executing. When a GOSUB statement is encountered in a program by the BASIC interpreter, the BASIC interpreter "pushes" its current position onto the stack before going to do the subroutine, and when a RETURN is executed, the interpreter "pulls" off the stack the information that tells it where it was before the subroutine call was made. The interpreter uses instructions like **PHA**, which pushes the contents of the accumulator onto the stack, and **PLA** (the reverse) which pulls a value off the stack and into the accumulator. The status register can also be pushed and pulled with the **PHP** and **PLP**, respectively.

The stack is 256 bytes long, and is located in page one of memory. It is therefore from $0100 to $01FF. It is organized backwards in memory. In other words, the first position in the stack is at $01FF, and the last is at $0100. Another register in the 6510 microprocessor is called the *stack pointer*, and it always points to the next available location in the stack. When something is pushed onto the stack, it is placed where the stack pointer points to, and the stack pointer is moved down to the next position (decremented). When something is pulled off the stack, the stack pointer is incremented, and the byte pointed to by the stack pointer is placed into the specified register.

Up to this point, we have covered immediate, zero page, and absolute mode instructions. We have also covered, but have not really talked about, the "implied" mode. The implied mode means that information is implied by an instruction itself. In other words, what registers, flags, and memory the instruction is referring to. The examples we have seen are PHA, PLA, PHP, and PLP, which refer to stack processing and the accumulator and status registers, respectively.

NOTE: The X register will be referred to as X from now on, and similarly A (accumulator), Y (Y index register), S (stack pointer), and P (processor status).

INDEXING

Indexing plays an extremely important part in the running of the 6510 microprocessor. It can be defined as "creating an actual address from a base address plus the contents of either the X or Y index registers."

For example, if X contains $05, and the microprocessor executes an LDA instruction in the "absolute X indexed mode" with base address (e.g., $9000), then the actual location that is loaded into the A register is $9000 + $05 = $9005. The mnemonic format of an absolute indexed instruction is the same as an absolute instruction except a ",X" or ",Y" denoting the index is added to the address.

EXAMPLE:

```
LDA $9000,X
```

There are absolute indexed, zero page indexed, indirect indexed, and indexed indirect modes of addressing available on the 6510 microprocessor.

INDIRECT INDEXED

This only allows usage of the Y register as the index. The actual address can only be in zero page, and the mode of instruction is called indirect because the zero page address specified in the instruction contains the low byte of the actual address, and the next byte to it contains the high order byte.

EXAMPLE:

Let us suppose that location $02 contains $45, and location $03 contains $1E. If the instruction to load the accumulator in the indirect indexed mode is executed and the specified zero page address is $02, then the actual address will be:

```
Low order  = contents of $02
High order = contents of $03
Y register  =$00
```

Thus the actual address = $1E45 + Y = $1E45.

The title of this mode does in fact imply an indirect principle, although this may be difficult to grasp at first sight. Let's look at it another way: "I am going to deliver this letter to the post office at address $02,MEMORY ST., and the address on the letter is $05 houses past $1600, MEMORY street." This is equivalent to the code:

```
LDA #$00      — load low order actual base address
STA $02       — set the low byte of the indirect address
LDA #$16      — load high order indirect address
STA $03       — set the high byte of the indirect address
LDY #$05      — set the indirect index (Y)
LDA ($02),Y   — load indirectly indexed by Y
```

INDEXED INDIRECT

Indexed indirect only allows usage of the X register as the index. This is the same as indirect indexed, except it is the zero page address of the *pointer* that is indexed, rather than the actual base address. Therefore, the actual base address IS the actual address because the index has already been used for the indirect. Index indirect would also be used if a *table* of indirect pointers were located in zero page memory, and the X register could then specify which indirect pointer to use.

EXAMPLE:

Let us suppose that location $02 contains $45, and location $03 contains $10. If the instruction to load the accumulator in the indexed indirect mode is executed and the specified zero page address is $02, then the actual address will be:

```
Low order  = contents of ($02+X)
High order = contents of ($03+X)
X register  = $00
```

Thus the actual pointer is in = $02 + X = $02.

Therefore, the actual address is the indirect address contained in $02 which is again $1045.

The title of this mode does in fact imply the principle, although it may be difficult to grasp at first sight. Look at it this way:

"I am going to deliver this letter to the fourth post office at address $01,MEMORY ST., and the address on the letter will then be delivered to $1600, MEMORY street." This is equivalent to the code:

```
LDA #$00        — load low order actual base address
STA $06         — set the low byte of the indirect address
LDA #$16        — load high order indirect address
STA $07         — set the high byte of the indirect address
LDX #$05        — set the indirect index (X)
LDA ($02,X)     —load indirectly indexed by X
```

NOTE: Of the two indirect methods of addressing, the first (indirect indexed) is far more widely used.

BRANCHES AND TESTING

Another very important principle in machine language is the ability to test, and detect certain conditions, in a smiliar fashion to the "IF . . . THEN, IF . . . GOTO" structure in CBM BASIC.

The various *flags* in the status register are affected by different instructions in different ways. For example, there is a flag that is set when an instruction has caused a zero result, and is reset when a result is not zero. The instruction:

```
LDA #$00
```

will cause *the zero result flag* to be set, because the instruction has resulted in the accumulator containing a zero.

There are a set of instructions that will, given a particular condition, *branch* to another part of the program. An example of a branch instruction is **BEQ**, which means *Branch if result EQual to zero*. The branch instructions *branch if the condition is true*, and if not, the program continues onto the next instruction, as if nothing had occurred. The branch instructions branch not by the result of the previous instruction(s), but by internally examining the status register. As was just mentioned, there is a *zero result flag* in the status register. The BEQ instruction branches if the zero result flag (known as **Z**) is set. Every branch instruction has an opposite branch instruction. The BEQ instruction has an opposite instruction **BNE**, which means *Branch on result Not Equal to zero* (i.e., Z not set).

The index registers have a number of associated instructions which modify their contents. For example, the **INX** instruction *INcrements the X index register*. If the X register contained $FF before it was incremented (the maximum number the X register can contain), it will "wrap around" back to zero. If you wanted a program to continue to do something until you had performed the increment of the X index that pushed it around to zero, you could use the BNE instruction to continue "looping" around, until X became zero.

The reverse of INX, is **DEX**, which is *DEcrement the X index register*. If the X index register is zero, DEX wraps around to $FF. Similarly, there are **INY** and **DEY** for the *Y index register*.

But what if a program didn't want to wait until X or Y had reached (or not reached) zero? Well there are *comparison instructions,* **CPX** and **CPY**, which allow the machine language programmer to test the index registers with specific values, or even the contents of memory locations. If you wanted to see if the X register contained $40, you would use the instruction:

```
CPX #$40      — compare X with the "value" $40.
BEQ           — branch to somewhere else in the
(some other     program, if this condition is "true."
part of the
program)
```

The compare, and branch instructions play a major part in any machine language program.

The operand specified in a branch instruction when using 64MON is the address of the part of the program that the branch goes to when the proper conditions are met. However, the operand is only an *offset*, which gets you from where the program currently is to the address specified. This offset is just one byte, and therefore the range that a branch instruction can branch to is limited. It can branch from 128 bytes backward, to 127 bytes forward.

NOTE: This is a total range of 255 bytes which is, of course, the maximum range of values one byte can contain.

64MON will tell you if you "branch out of range" by refusing to "assemble" that particular instruction. But don't worry about that now because it's unlikely that you will have such branches for quite a while. The branch is a "quick" instruction by machine language standards because of the "offset" principle as opposed to an absolute address. 64MON allows you to type in an absolute address, and it calculates the correct offset. This is just one of the "comforts" of using an assembler.

NOTE: It is NOT possible to cover every single branch instruction. For further information, refer to the Bibliography section in Appendix F.

SUBROUTINES

In machine language (in the same way as using BASIC), you can call subroutines. The instruction to call a subroutine is **JSR** (Jump to Sub-Routine), followed by the specified absolute address.

Incorporated in the operating system, there is a machine language subroutine that will PRINT a character to the screen. The CBM ASCII code of the character should be in the accumulator before calling the subroutine. The address of this subroutine is $FFD2.

Therefore, to print "HI" to the screen, the following program should be entered:

```
.A 1400 LDA #$48     — load the CBM ASCII code of "H"
.A 1402 JSR $FFD2    — print it
.A 1405 LDA #$49     — load the CBM ASCII code of "I"
.A 1407 JSR $FFD2    — print that too
.A 140A LDA #$0D     — print a carriage return as well
.A 140C JSR $FFD2
.A 140F BRK          — return to 64MON
.G 1400              — will print "HI" and return to 64MON
```

The "PRINT a character" routine we have just used is part of the KERNAL *jump table*. The instruction similar to GOTO in BASIC is **JMP**, which means *JuMP to the specified absolute address*. The KERNAL is a long list of "standardized" subroutines that control ALL input and output of the Commodore 64. Each entry in the KERNAL JMPs to a subroutine in the operating system. This "jump table" is found between memory locations $FF84 to $FFF5 in the operating system. A full explanation of the KERNAL is available in the "KERNAL Reference Section" of this manual. However, certain routines are used here to show how easy and effective the KERNAL is.

Let's now use the new principles you've just learned in another program. It will help you to put the instructions into context:

This program will display the alphabet using a KERNAL routine. The only new instruction introduced here is **TXA** *Transfer the contents of the X index register, into the Accumulator.*

```
.A 1400 LDX #$41      — X = CBM ASCII of "A"
.A 1402 TXA           — A = X
.A 1403 JSR $FFD2     — print character
.A 1406 INX           — bump count
.A 1407 CPX #$5B      — have we gone past "Z" ?
.A 1409 BNE $1402     — no, go back and do more
.A 140B BRK           — yes, return to 64MON
```

To see the Commodore 64 print the alphabet, type the familiar command:

```
.G 1400
```

The comments that are beside the program, explain the program flow and logic. If you are writing a program, write it on paper first, and then test it in small parts if possible.

USEFUL TIPS FOR THE BEGINNER

One of the best ways to learn machine language is to look at other peoples' machine language programs. These are published all the time in magazines and newsletters. Look at them even if the article is for a different computer, which also uses the 6510 (or 6502) microprocessor. You should make sure that you thoroughly understand the code that you look at. This will require perseverence, especially when you see a new technique that you have never come across before. This can be infuriating, but if patience prevails, you will be the victor.

Having looked at other machine language programs, you MUST write your own. These may be utilities for your BASIC programs, or they may be an all machine language program.

You should also use the utilities that are available, either IN your computer, or in a program, that aid you in writing, editing, or tracking down errors in a machine language program. An example would be the KERNAL, which allows you to check the keyboard, print text, control peripheral devices like disk drives, printers, modems, etc., manage memory and the screen. It is extremely powerful and it is advised strongly that it is used (refer to KERNAL section, Page 268).

Advantages of writing programs in machine language:

1. Speed—Machine language is hundreds, and in some cases thousands of times faster than a high level language such as BASIC.
2. Tightness—A machine language program can be made totally "watertight," i.e., the user can be made to do ONLY what the program allows, and no more. With a high level language, you are relying on the user not "crashing" the BASIC interpreter by entering, for example, a zero which later causes a:

```
?DIVISION BY ZERO ERROR IN LINE 830

READY.
■
```

In essence, the computer can only be maximized by the machine language programmer.

APPROACHING A LARGE TASK

When approaching a large task in machine language, a certain amount of subconscious thought has usually taken place. You think about how certain processes are carried out in machine language. When the task is started, it is usually a good idea to write it out on paper. Use block diagrams of memory usage, functional modules of code required, and a program flow. Let's say that you wanted to write a roulette game in machine language. You could outline it something like this:

- Display title
- Ask if player requires instructions
- YES—display them—Go to START
- NO—Go to START
- START Initialize everything
- MAIN display roulette table
- Take in bets
- Spin wheel
- Slow wheel to stop
- Check bets with result
- Inform player
- Player any money left?
- YES—Go to MAIN
- NO—Inform user!, and go to START

This is the main outline. As each module is approached, you can break it down further. If you look at a large indigestable problem as something that can be broken down into small enough pieces to be eaten, then you'll be able to approach something that seems impossible, and have it all fall into place.

This process only improves with practice, so KEEP TRYING.

MCS6510 MICROPROCESSOR

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive-Or" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	Increment Index Y by One
JMP	Jump to New Location

INSTRUCTION SET—ALPHABETIC SEQUENCE

JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift Right One Bit (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer
TYA	Transfer Index Y to Accumulator

The following notation applies to this summary:

A	Accumulator
X, Y	Index Registers
M	Memory
P	Processor Status Register
S	Stack Pointer
✓	Change
_	No Change
+	Add
∧	Logical AND
-	Subtract
⊻	Logical Exclusive Or
↑	Transfer from Stack
↓	Transfer to Stack
→	Transfer to
←	Transfer from
V	Logical OR
PC	Program Counter
PCH	Program Counter High
PCL	Program Counter Low
OPER	OPERAND
#	IMMEDIATE ADDRESSING MODE

Note: At the top of each table is located in parentheses a reference number (Ref: XX) which directs the user to that Section in the MCS6500 Microcomputer Family Programming Manual in which the instruction is defined and discussed.

ADC *Add memory to accumulator with carry* ADC

Operation: A + M + C → A, C

N Ƶ C I D V
✓ ✓ ✓ — — ✓

(Ref: 2.2.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	ADC # Oper	69	2	2
Zero Page	ADC Oper	65	2	3
Zero Page, X	ADC Oper, X	75	2	4
Absolute	ADC Oper	6D	3	4
Absolute, X	ADC Oper, X	7D	3	4*
Absolute, Y	ADC Oper, Y	79	3	4*
(Indirect, X)	ADC (Oper, X)	61	2	6
(Indirect), Y	ADC (Oper), Y	71	2	5*

* Add 1 if page boundary is crossed.

AND *"AND" memory with accumulator* AND

Logical AND to the accumulator

Operation: A ∧ M → A

N Ƶ C I D V
✓ ✓ — —

(Ref: 2.2.3.0)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	AND # Oper	29	2	2
Zero Page	AND Oper	25	2	3
Zero Page, X	AND Oper, X	35	2	4
Absolute	AND Oper	2D	3	4
Absolute, X	AND Oper, X	3D	3	4*
Absolute, Y	AND Oper, Y	39	3	4*
(Indirect, X)	AND (Oper, X)	21	2	6
(Indirect), Y	AND (Oper), Y	31	2	5

* Add 1 if page boundary is crossed.

ASL ASL ASL

ASL *Shift Left One Bit (Memory or Accumulator)*

Operation: C ← |7|6|5|4|3|2|1|Ø| ← Ø

N Ƶ C I D V
✓ ✓ ✓ — — —

(Ref: 10.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	ASL A	ØA	1	2
Zero Page	ASL Oper	Ø6	2	5
Zero Page, X	ASL Oper, X	16	2	6
Absolute	ASL Oper	ØE	3	6
Absolute, X	ASL Oper, X	1E	3	7

BCC BCC BCC

BCC *Branch on Carry Clear*

Operation: Branch on C = Ø

N Ƶ C I D V
— — — — — —

(Ref: 4.1.1.3)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BCC Oper	9Ø	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

BCS BCS BCS

BCS *Branch on carry set*

Operation: Branch on C = 1

N Ƶ C I D V
— — — — — —

(Ref: 4.1.1.4)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BCS Oper	BØ	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to next page.

BEQ

BEQ *Branch on result zero*

BEQ

Operation: Branch on Ƶ = 1 (Ref: 4.1.1.5)

N Ƶ C I D V
_ _ _ _ _ _

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BEQ Oper	FØ	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to next page.

BIT

BIT *Test bits in memory with accumulator*

BIT

Operation: A ∧ M, $M_7 \rightarrow N$, $M_6 \rightarrow V$

Bit 6 and 7 are transferred to the status register. If the result of A∧M is zero then Z = 1, otherwise Z - Ø (Ref: 4.2.1.1)

N Ƶ C I D V
M_7 ✓ _ _ _ M_6

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	BIT Oper	24	2	3
Absolute	BIT Oper	2C	3	4

BMI

BMI *Branch on result minus*

BMI

Operation: Branch on N = 1 (Ref: 4.1.1.1)

N Ƶ C I D V
_ _ _ _ _ _

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BMI Oper	3Ø	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

BNE **BNE** *Branch on result not zero* BNE

Operation: Branch on Z = 0

N Ƶ C I D V
– – – – – –

(Ref: 4.1.1.6)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BNE Oper	DØ	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

BPL **BPL** *Branch on result plus* BPL

Operation: Branch on N = Ø

N Ƶ C I D V
– – – – – –

(Ref: 4.1.1.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BPL Oper	1Ø	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

BRK **BRK** *Force Break* BRK

Operation: Forced Interrupt PC + 2 ↓ P ↓

N Ƶ C I D V
– – – 1 – –

(Ref: 9.11)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	BRK	ØØ	1	7

1. A BRK command cannot be masked by setting I.

BVC

BVC *Branch on overflow clear* **BVC**

Operation: Branch on V = 0 N Ƶ C I D V

(Ref: 4.1.1.8) — — — — — —

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BVC Oper	5Ø	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

BVS

BVS *Branch on overflow set* **BVS**

Operation: Branch on V = 1 N Ƶ C I D V

(Ref: 4.1.1.7) — — — — — —

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BVS Oper	7Ø	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

CLC

CLC *Clear carry flag* **CLC**

Operation: Ø → C N Ƶ C I D V

(Ref: 3.0.2) — — Ø — — —

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLC	18	1	2

CLD CLD *Clear decimal mode* CLD

Operation: Ø → D

N Z C I D V

— — — — Ø —

(Ref: 3.3.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLD	D8	1	2

CLI CLI *Clear interrupt disable bit* CLI

Operation: Ø → I

N Z C I D V

— — — Ø — —

(Ref: 3.2.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLI	58	1	2

CLV CLV *Clear overflow flag* CLV

Operation: Ø → V

N Z C I D V

— — — — — Ø

(Ref: 3.6.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLV	B8	1	2

CMP — CMP *Compare memory and accumulator* — CMP

Operation: A - M

N Z C I D V

✓ ✓ ✓ — — —

(Ref: 4.2.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	CMP #Oper	C9	2	2
Zero Page	CMP Oper	C5	2	3
Zero Page, X	CMP Oper, X	D5	2	4
Absolute	CMP Oper	CD	3	4
Absolute, X	CMP Oper, X	DD	3	4*
Absolute, Y	CMP Oper, Y	D9	3	4*
(Indirect, X)	CMP (Oper, X)	C1	2	6
(Indirect), Y	CMP (Oper), Y	D1	2	5*

* Add 1 if page boundary is crossed.

CPX — CPX *Compare Memory and Index X* — CPX

Operation: X - M

N Z C I D V

✓ ✓ ✓ — — —

(Ref: 7.8)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	CPX #Oper	EØ	2	2
Zero Page	CPX Oper	E4	2	3
Absolute	CPX Oper	EC	3	4

CPY — CPY *Compare memory and index Y* — CPY

Operation: Y - M

N Z C I D V

✓ ✓ ✓ — — —

(Ref: 7.9)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	CPY #Oper	CØ	2	2
Zero Page	CPY Oper	C4	2	3
Absolute	CPY Oper	CC	3	4

DEC

DEC *Decrement memory by one*

DEC

Operation: M - 1 → M

N	Ƶ	C	I	D	V
✓	✓	—	—	—	—

(Ref: 10.7)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	DEC Oper	C6	2	5
Zero Page, X	DEC Oper, X	D6	2	6
Absolute	DEC Oper	CE	3	6
Absolute, X	DEC Oper, X	DE	3	7

DEX

DEX *Decrement index X by one*

DEX

Operation: X - 1 → X

N	Ƶ	C	I	D	V
✓	✓	—	—	—	—

(Ref: 7.6)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	DEX	CA	1	2

DEY

DEY *Decrement index Y by one*

DEY

Operation: Y - 1 → Y

N	Ƶ	C	I	D	V
✓	✓	—	—	—	—

(Ref: 7.7)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	DEY	88	1	2

EOR

EOR *"Exclusive—Or" memory with accumulator* **EOR**

Operation: A ⊻ M → A

N Z C I D V
✓ ✓ — — — —

(Ref: 2.2.3.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	EOR #Oper	49	2	2
Zero Page	EOR Oper	45	2	3
Zero Page, X	EOR Oper, X	55	2	4
Absolute	EOR Oper	4D	3	4
Absolute, X	EOR Oper, X	5D	3	4*
Absolute, Y	EOR Oper, Y	59	3	4*
(Indirect, X)	EOR (Oper, X)	41	2	6
(Indirect),Y	EOR (Oper), Y	51	2	5*

* Add 1 if page boundary is crossed.

INC

INC *Increment memory by one* **INC**

Operation: M + 1 → M

N Z C I D V
✓ ✓ — — — —

(Ref: 10.6)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	INC Oper	E6	2	5
Zero Page, X	INC Oper, X	F6	2	6
Absolute	INC Oper	EE	3	6
Absolute, X	INC Oper, X	FE	3	7

INX

INX *Increment Index X by one* **INX**

Operation: X + 1 → X

N Z C I D V
✓ ✓ — — — —

(Ref: 7.4)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	INX	E8	1	2

INY

INY *Increment Index Y by one*

INY

```
Operation: Y + 1 → Y                          N Z C I D V
                                              ✓ ✓ — — — —
                       (Ref:  7.5)
```

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	INY	C8	1	2

JMP

JMP *Jump to new location*

JMP

```
Operation: (PC + 1) → PCL                     N Z C I D V
           (PC + 2) → PCH   (Ref:  4.0.2)     — — — — — —
                            (Ref:  9.8.1)
```

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Absolute Indirect	JMP Oper JMP (Oper)	4C 6C	3 3	3 5

JSR

JSR *Jump to new location saving return address*

JSR

```
Operation: PC + 2 ↓, (PC + 1) → PCL           N Z C I D V
                     (PC + 2) → PCH                   — —
                          (Ref:  8.1)
```

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Absolute	JSR Oper	20	3	6

LDA **LDA** *Load accumulator with memory* LDA

Operation: M → A

N Z C I D V
✓ ✓ — — — —

(Ref: 2.1.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	LDA # Oper	A9	2	2
Zero Page	LDA Oper	A5	2	3
Zero Page, X	LDA Oper, X	B5	2	4
Absolute	LDA Oper	AD	3	4
Absolute, X	LDA Oper, X	BD	3	4*
Absolute, Y	LDA Oper, Y	B9	3	4*
(Indirect, X)	LDA (Oper, X)	A1	2	6
(Indirect), Y	LDA (Oper), Y	B1	2	5*

* Add 1 if page boundary is crossed.

LDX **LDX** *Load index X with memory* LDX

Operation: M → X

N Z C I D V
✓ ✓ — — — —

(Ref: 7.0)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	LDX # Oper	A2	2	2
Zero Page	LDX Oper	A6	2	3
Zero Page, Y	LDX Oper, Y	B6	2	4
Absolute	LDX Oper	AE	3	4
Absolute, Y	LDX Oper, Y	BE	3	4*

* Add 1 when page boundary is crossed.

LDY LDY

LDY *Load index Y with memory*

Operation: M → Y

N Ƶ C I D V
✓ ✓ — — — —

(Ref: 7.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	LDY #Oper	AØ	2	2
Zero Page	LDY Oper	A4	2	3
Zero Page, X	LDY Oper, X	B4	2	4
Absolute	LDY Oper	AC	3	4
Absolute, X	LDY Oper, X	BC	3	4*

* Add 1 when page boundary is crossed.

LSR LSR

LSR *Shift right one bit (memory or accumulator)*

Operation: Ø → |7|6|5|4|3|2|1|0| → C

N Ƶ C I D V
Ø ✓ ✓ — — —

(Ref: 10.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	LSR A	4A	1	2
Zero Page	LSR Oper	46	2	5
Zero Page, X	LSR Oper, X	56	2	6
Absolute	LSR Oper	4E	3	6
Absolute, X	LSR Oper, X	5E	3	7

NOP NOP

NOP *No operation*

Operation: No Operation (2 cycles)

N Ƶ C I D V
— — — — — —

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	NOP	EA	1	2

ORA **ORA** *"OR" memory with accumulator* ORA

Operation: A ∨ M → A

N Ƶ C I D V
✓ ✓ — — — —

(Ref: 2.2.3.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	ORA #Oper	Ø9	2	2
Zero Page	ORA Oper	Ø5	2	3
Zero Page, X	ORA Oper, X	15	2	4
Absolute	ORA Oper	ØD	3	4
Absolute, X	ORA Oper, X	1D	3	4*
Absolute, Y	ORA Oper, Y	19	3	4*
(Indirect, X)	ORA (Oper, X)	Ø1	2	6
(Indirect), Y	ORA (Oper), Y	11	2	5

* Add 1 on page crossing

PHA **PHA** *Push accumulator on stack* PHA

Operation: A ↓

N Ƶ C I D V
— — — — — —

(Ref: 8.5)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PHA	48	1	3

PHP **PHP** *Push processor status on stack* PHP

Operation: P↓

N Ƶ C I D V
— — — — — —

(Ref: 8.11)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PHP	Ø8	1	3

PLA

PLA *Pull accumulator from stack*

PLA

Operation: A ↑

N	Z	C	I	D	V
✓	✓	–	–	–	–

(Ref: 8.5)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PLA	68	1	4

PLP

PLP *Pull processor status from stack*

PLP

Operation: P ↑

N	Z	C	I	D	V
From Stack					

(Ref: 8.12)

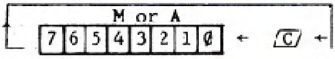
Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PLP	28	1	4

ROL

ROL *Rotate one bit left (memory or accumulator)*

ROL

Operation:



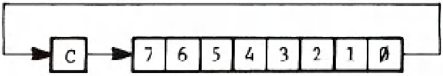
N	Z	C	I	D	V
✓	✓	✓	–	–	–

(Ref: 10.3)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	ROL A	2A	1	2
Zero Page	ROL Oper	26	2	5
Zero Page, X	ROL Oper, X	36	2	6
Absolute	ROL Oper	2E	3	6
Absolute, X	ROL Oper, X	3E	3	7

ROR ROR *Rotate one bit right (memory or accumulator)* ROR

Operation:



(Ref: 10.4)

N Z C I D V
✓ ✓ ✓ - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	ROR A	6A	1	2
Zero Page	ROR Oper	66	2	5
Zero Page,X	ROR Oper,X	76	2	6
Absolute	ROR Oper	6E	3	6
Absolute,X	ROR Oper,X	7E	3	7

Note: ROR instruction is available on MCS650X microprocessors after June, 1976.

RTI RTI *Return from interrupt* RTI

Operation: P↑ PC↑

N Z C I D V
From Stack

(Ref: 9.6)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	RTI	4Ø	1	6

RTS RTS *Return from subroutine* RTS

Operation: PC↑, PC + 1→ PC

N Z C I D V
— — — — — —

(Ref: 8.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	RTS	6Ø	1	6

SBC

SBC *Subtract memory from accumulator with borrow* **SBC**

Operation: A − M − $\overline{C}$ → A

Note: $\overline{C}$ = Borrow (Ref: 2.2.2)

N Z C I D V
√ √ √ — — √

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	SBC #Oper	E9	2	2
Zero Page	SBC Oper	E5	2	3
Zero Page, X	SBC Oper, X	F5	2	4
Absolute	SBC Oper	ED	3	4
Absolute, X	SBC Oper, X	FD	3	4*
Absolute, Y	SBC Oper, Y	F9	3	4*
(Indirect, X)	SBC (Oper, X)	E1	2	6
(Indirect), Y	SBC (Oper), Y	F1	2	5*

* Add 1 when page boundary is crossed.

SEC

SEC *Set carry flag* **SEC**

Operation: 1 → C

(Ref: 3.0.1)

N Z C I D V
— — 1 — — —

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SEC	38	1	2

SED

SED *Set decimal mode* **SED**

Operation: 1 → D

(Ref: 3.3.1)

N Z C I D V
— — — — 1 —

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SED	F8	1	2

SEI — SEI *Set interrupt disable status* — SEI

Operation: 1 → I

N	Z	C	I	D	V
–	–	–	1	–	–

(Ref: 3.2.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SEI	78	1	2

STA — STA *Store accumulator in memory* — STA

Operation: A → M

N	Z	C	I	D	V
–	–	–	–	–	–

(Ref: 2.1.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STA Oper	85	2	3
Zero Page, X	STA Oper, X	95	2	4
Absolute	STA Oper	8D	3	4
Absolute, X	STA Oper, X	9D	3	5
Absolute, Y	STA Oper, Y	99	3	5
(Indirect, X)	STA (Oper, X)	81	2	6
(Indirect), Y	STA (Oper), Y	91	2	6

STX — STX *Store index X in memory* — STX

Operation: X → M

N	Z	C	I	D	V
–	–	–	–	–	–

(Ref: 7.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STX Oper	86	2	3
Zero Page, Y	STX Oper, Y	96	2	4
Absolute	STX Oper	8E	3	4

STY **STY** *Store index Y in memory* STY

Operation: Y → M

N Z C I D V
— — — — — —

(Ref: 7.3)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STY Oper	84	2	3
Zero Page, X	STY Oper, X	94	2	4
Absolute	STY Oper	8C	3	4

TAX **TAX** *Transfer accumulator to index X* TAX

Operation: A → X

N Z C I D V
✓ ✓ — — — —

(Ref: 7.11)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TAX	AA	1	2

TAY **TAY** *Transfer accumulator to index Y* TAY

Operation: A → Y

N Z C I D V
✓ ✓ — — — —

(Ref: 7.13)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TAY	A8	1	2

TSX

TSX *Transfer stack pointer to index X* **TSX**

Operation: S → X

N Z C I D V
✓ ✓ – – – –

(Ref: 8.9)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TSX	BA	1	2

TXA

TXA *Transfer index X to accumulator* **TXA**

Operation: X → A

N Z C I D V
✓ ✓ – – – –

(Ref: 7.12)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TXA	8A	1	2

TXS

TXS *Transfer index X to stack pointer* **TXS**

Operation: X → S

N Z C I D V
– – – – – –

(Ref: 8.8)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TXS	9A	1	2

TYA

TYA *Transfer index Y to accumulator* **TYA**

Operation: Y → A

N Z C I D V
✓ ✓ – – – –

(Ref: 7.14)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TYA	98	1	2

INSTRUCTION ADDRESSING MODES AND

	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect, X)	(Indirect), Y	Absolute Indirect
ADC	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
AND	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
ASL	2	.	5	6	.	6	7	.	.	.	.	.	.
BCC	.	.	.	.	.	.	.	.	.	2**	.	.	.
BCS	.	.	.	.	.	.	.	.	.	2**	.	.	.
BEQ	.	.	.	.	.	.	.	.	.	2**	.	.	.
BIT	.	.	3	.	.	4	.	.	.	.	.	.	.
BMI	.	.	.	.	.	.	.	.	.	2**	.	.	.
BNE	.	.	.	.	.	.	.	.	.	2**	.	.	.
BPL	.	.	.	.	.	.	.	.	.	2**	.	.	.
BRK	.	.	.	.	.	.	.	.	.	.	.	.	.
BVC	.	.	.	.	.	.	.	.	.	2**	.	.	.
BVS	.	.	.	.	.	.	.	.	.	2**	.	.	.
CLC	.	.	.	.	.	.	.	.	2	.	.	.	.
CLD	.	.	.	.	.	.	.	.	2	.	.	.	.
CLI	.	.	.	.	.	.	.	.	2	.	.	.	.
CLV	.	.	.	.	.	.	.	.	2	.	.	.	.
CMP	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
CPX	.	2	3	.	.	4	.	.	.	.	.	.	.
CPY	.	2	3	.	.	4	.	.	.	.	.	.	.
DEC	.	.	5	6	.	6	7	.	.	.	.	.	.
DEX	.	.	.	.	.	.	.	.	2	.	.	.	.
DEY	.	.	.	.	.	.	.	.	2	.	.	.	.
EOR	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
INC	.	.	5	6	.	6	7	.	.	.	.	.	.
INX	.	.	.	.	.	.	.	.	2	.	.	.	.
INY	.	.	.	.	.	.	.	.	2	.	.	.	.
JMP	.	.	.	.	.	3	.	.	.	.	.	.	5

* Add one cycle if indexing across page boundary

** Add one cycle if branch is taken, Add one additional

RELATED EXECUTION TIMES (in clock cycles)

	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect, X)	(Indirect),Y	Absolute Indirect
JSR	.	.	.	.	.	6	.	.	.	.	.	.	.
LDA	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
LDX	.	2	3	.	4	4	.	4*	.	.	.	.	.
LDY	.	2	3	4	.	4	4*	.	.	.	.	.	.
LSR	2	.	5	6	.	6	7	.	.	.	.	.	.
NOP	.	.	.	.	.	.	.	.	2	.	.	.	.
ORA	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
PHA	.	.	.	.	.	.	.	.	3	.	.	.	.
PHP	.	.	.	.	.	.	.	.	3	.	.	.	.
PLA	.	.	.	.	.	.	.	.	4	.	.	.	.
PLP	.	.	.	.	.	.	.	.	4	.	.	.	.
ROL	2	.	5	6	.	6	7	.	.	.	.	.	.
ROR	2	.	5	6	.	6	7	.	.	.	.	.	.
RTI	.	.	.	.	.	.	.	.	6	.	.	.	.
RTS	.	.	.	.	.	.	.	.	6	.	.	.	.
SBC	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
SEC	.	.	.	.	.	.	.	.	2	.	.	.	.
SED	.	.	.	.	.	.	.	.	2	.	.	.	.
SEI	.	.	.	.	.	.	.	.	2	.	.	.	.
STA	.	.	3	4	.	4	5	5	.	.	6	6	.
STX	.	.	3	.	4	4	.	.	.	.	.	.	.
STY	.	.	3	4	.	4	.	.	.	.	.	.	.
TAX	.	.	.	.	.	.	.	.	2	.	.	.	.
TAY	.	.	.	.	.	.	.	.	2	.	.	.	.
TSX	.	.	.	.	.	.	.	.	2	.	.	.	.
TXA	.	.	.	.	.	.	.	.	2	.	.	.	.
TXS	.	.	.	.	.	.	.	.	2	.	.	.	.
TYA	.	.	.	.	.	.	.	.	2	.	.	.	.

if branching operation crosses page boundary

ØØ - BRK

Ø1 - ORA - (Indirect,X)

Ø2 - Future Expansion

Ø3 - Future Expansion

Ø4 - Future Expansion

Ø5 - ORA - Zero Page

Ø6 - ASL - Zero Page

Ø7 - Future Expansion

Ø8 - PHP

Ø9 - ORA - Immediate

ØA - ASL - Accumulator

ØB - Future Expansion

ØC - Future Expansion

ØD - ORA - Absolute

ØE - ASL - Absolute

ØF - Future Expansion

1Ø - BPL

11 - ORA - (Indirect),Y

12 - Future Expansion

13 - Future Expansion

14 - Future Expansion

15 - ORA - Zero Page,X

16 - ASL - Zero Page,X

17 - Future Expansion

18 - CLC

19 - ORA - Absolute,Y

1A - Future Expansion

1B - Future Expansion

1C - Future Expansion

1D - ORA - Absolute,X

1E - ASL - Absolute,X

1F - Future Expansion

2Ø - JSR

21 - AND - (Indirect,X)

22 - Future Expansion

23 - Future Expansion

24 - BIT - Zero Page

25 - AND - Zero Page

26 - ROL - Zero Page

27 - Future Expansion

28 - PLP

29 - AND - Immediate

2A - ROL - Accumulator

2B - Future Expansion

2C - BIT - Absolute

2D - AND - Absolute

2E - ROL - Absolute

2F - Future Expansion

3Ø - BMI

31 - AND - (Indirect),Y

32 - Future Expansion

33 - Future Expansion

34 - Future Expansion

35 - AND - Zero Page,X

36 - ROL - Zero Page,X

37 - Future Expansion

38 - SEC

39 - AND - Absolute,Y

3A - Future Expansion

3B - Future Expansion

3C - Future Expansion

3D - AND - Absolute,X

3E - ROL - Absolute,X

3F - Future Expansion

4Ø - RTI

41 - EOR - (Indirect,X)

42 - Future Expansion

43 - Future Expansion

44 - Future Expansion

45 - EOR - Zero Page

46 - LSR - Zero Page

47 - Future Expansion

48 - PHA

49 - EOR - Immediate

4A - LSR - Accumulator

4B - Future Expansion

4C - JMP - Absolute

4D - EOR - Absolute

4E - LSR - Absolute

4F - Future Expansion

5Ø - BVC

51 - EOR - (Indirect),Y

52 - Future Expansion

53 - Future Expansion

54 - Future Expansion

55 - EOR - Zero Page,X

56 - LSR - Zero Page,X

57 - Future Expansion

58 - CLI

59 - EOR - Absolute,Y

5A - Future Expansion

5B - Future Expansion

5C - Future Expansion

5D - EOR - Absolute,X

5E - LSR - Absolute,X

5F - Future Expansion

6Ø - RTS

61 - ADC - (Indirect,X)

62 - Future Expansion

63 - Future Expansion

64 - Future Expansion

65 - ADC - Zero Page

66 - ROR - Zero Page

67 - Future Expansion

68 - PLA

69 - ADC - Immediate

6A - ROR - Accumulator

6B - Future Expansion

6C - JMP - Indirect

6D - ADC - Absolute

6E - ROR - Absolute

6F - Future Expansion

7Ø - BVS

71 - ADC - (Indirect),Y

72 - Future Expansion

73 - Future Expansion

74 - Future Expansion

75 - ADC - Zero Page,X

76 - ROR - Zero Page,X

77 - Future Expansion

78 - SEI

79 - ADC - Absolute,Y

7A - Future Expansion

7B - Future Expansion

7C - Future Expansion

7D - ADC - Absolute,X

7E - ROR - Absolute,X

7F - Future Expansion

80 - Future Expansion

81 - STA - (Indirect,X)

82 - Future Expansion

83 - Future Expansion

84 - STY - Zero Page

85 - STA - Zero Page

86 - STX - Zero Page

87 - Future Expansion

88 - DEY

89 - Future Expansion

8A - TXA

8B - Future Expansion

8C - STY - Absolute

8D - STA - Absolute

8E - STX - Absolute

8F - Future Expansion

9Ø - BCC

91 - STA - (Indirect),Y

92 - Future Expansion

93 - Future Expansion

94 - STY - Zero Page,X

95 - STA - Zero Page,X

96 - STX - Zero Page,Y

97 - Future Expansion

98 - TYA

99 - STA - Absolute,Y

9A - TXS

9B - Future Expansion

9C - Future Expansion

9D - STA - Absolute,X

9E - Future Expansion

9F - Future Expansion

AØ - LDY - Immediate

A1 - LDA - (Indirect,X)

A2 - LDX - Immediate

A3 - Future Expansion

A4 - LDY - Zero Page

A5 - LDA - Zero Page

A6 - LDX - Zero Page

A7 - Future Expansion

A8 - TAY

A9 - LDA - Immediate

AA - TAX

AB - Future Expansion

AC - LDY - Absolute

AD - LDA - Absolute

AE - LDX - Absolute

AF - Future Expansion

BØ - BCS

B1 - LDA - (Indirect),Y

B2 - Future Expansion

B3 - Future Expansion

B4 - LDY - Zero Page,X

B5 - LDA - Zero Page,X

B6 - LDX - Zero Page,Y

B7 - Future Expansion

B8 - CLV

B9 - LDA - Absolute,Y

BA - TSX

BB - Future Expansion

BC - LDY - Absolute,X

BD - LDA - Absolute,X

BE - LDX - Absolute,Y

BF - Future Expansion

CØ - CPY - Immediate
C1 - CMP - (Indirect,X)
C2 - Future Expansion
C3 - Future Expansion
C4 - CPY - Zero Page
C5 - CMP - Zero Page
C6 - DEC - Zero Page
C7 - Future Expansion
C8 - INY
C9 - CMP - Immediate
CA - DEX
CB - Future Expansion
CC - CPY - Absolute
CD - CMP - Absolute
CE - DEC - Absolute
CF - Future Expansion
DØ - BNE
D1 - CMP - (Indirect),Y
D2 - Future Expansion
D3 - Future Expansion
D4 - Future Expansion
D5 - CMP - Zero Page,X
D6 - DEC - Zero Page,X
D7 - Future Expansion
D8 - CLD
D9 - CMP - Absolute,Y
DA - Future Expansion
DB - Future Expansion
DC - Future Expansion
DD - CMP - Absolute,X
DE - DEC - Absolute,X
DF - Future Expansion

EØ - CPX - Immediate
E1 - SBC - (Indirect,X)
E2 - Future Expansion
E3 - Future Expansion
E4 - CPX - Zero Page
E5 - SBC - Zero Page
E6 - INC - Zero Page
E7 - Future Expansion
E8 - INX
E9 - SBC - Immediate
EA - NOP
EB - Future Expansion
EC - CPX - Absolute
ED - SBC - Absolute
EE - INC - Absolute
EF - Future Expansion
FØ - BEQ
F1 - SBC - (Indirect),Y
F2 - Future Expansion
F3 - Future Expansion
F4 - Future Expansion
F5 - SBC - Zero Page,X
F6 - INC - Zero Page,X
F7 - Future Expansion
F8 - SED
F9 - SBC - Absolute,Y
FA - Future Expansion
FB - Future Expansion
FC - Future Expansion
FD - SBC - Absolute,X
FE - INC - Absolute,X
FF - Future Expansion

MEMORY MANAGEMENT ON THE COMMODORE 64

The Commodore 64 has 64K bytes of RAM. It also has 20K bytes of ROM, containing BASIC, the operating system, and the standard character set. It also accesses input/output devices as a 4K chunk of memory. How is this all possible on a computer with a 16-bit address bus, that is normally only capable of addressing 64K?

The secret is in the 6510 processor chip itself. On the chip is an input/output port. This port is used to control whether RAM or ROM or I/O will appear in certain portions of the system's memory. The port is also used to control the Datassette^{TM}, so it is important to affect only the proper bits.

The 6510 input/output port appears at location 1. The data direction register for this port appears at location 0. The port is controlled like any of the other input/output ports in the system . . . the data direction controls whether a given bit will be an input or an output, and the actual data transfer occurs through the port itself.

The lines in the 6510 control port are defined as follows:

NAME	BIT	DIRECTION	DESCRIPTION
LORAM	0	OUTPUT	Control for RAM/ROM at $A000–$BFFF (BASIC)
HIRAM	1	OUTPUT	Control for RAM/ROM at $E000–$FFFF (KERNAL)
CHAREN	2	OUTPUT	Control for I/O/ROM at $D000–$DFFF
	3	OUTPUT	Cassette write line
	4	INPUT	Cassette switch sense
	5	OUTPUT	Cassette motor control

The proper value for the data direction register is as follows:

BITS	5	4	3	2	1	0
	1	0	1	1	1	1

(where 1 is an output, and 0 is an input).

This gives a value of 47 decimal. The Commodore 64 automatically sets the data direction register to this value.

The control lines, in general, perform the function given in their descriptions. However, a combination of control lines are occasionally used to get a particular memory configuration.

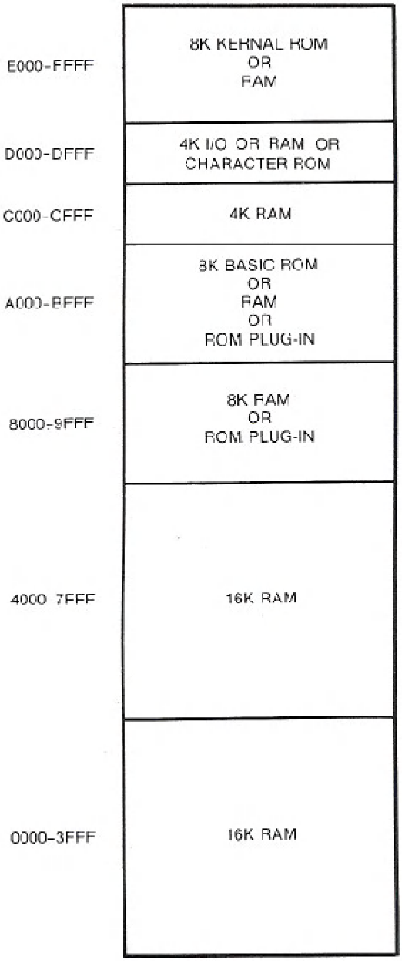
LORAM (bit 0) can generally be thought of as a control line which banks the 8K byte BASIC ROM in and out of the microprocessor address space. Normally, this line is HIGH for BASIC operation. If this line is programmed LOW, the BASIC ROM will disappear from the memory map and be replaced by 8K bytes of RAM from $A000–$BFFF.

HIRAM (bit 1) can generally be thought of as a control line which banks the 8K byte KERNAL ROM in and out of the microprocessor address space. Normally, this line is HIGH for BASIC operation. If this line is programmed LOW, the KERNAL ROM will disappear from the memory map and be replaced by 8K bytes of RAM from $E000–$FFFF.

CHAREN (bit 2) is used only to bank the 4K byte character generator ROM in or out of the microprocessor address space. From the processor point of view, the character ROM occupies the same address space as the I/O devices ($D000–$DFFF). When the CHAREN line is set to 1 (as is normal), the I/O devices appear in the microprocessor address space, and the character ROM is not accessable. When the CHAREN bit is cleared to 0, the character ROM appears in the processor address space, and the I/O devices are not accessable. (The microprocessor only needs to access the character ROM when downloading the character set from ROM to RAM. Special care is needed for this . . . see the section on PROGRAMMABLE CHARACTERS in the GRAPHICS chapter). CHAREN can be overridden by other control lines in certain memory configurations. CHAREN will have no effect on any memory configuration without I/O devices. RAM will appear from $D000–$DFFF instead.

NOTE: In any memory map containing ROM, a WRITE (a POKE) to a ROM location will store data in the RAM "under" the ROM. Writing to a ROM location stores data in the "hidden" RAM. For example, this allows a hi-resolution screen to be kept underneath a ROM, and be changed without having to bank the screen back into the processor address space. Of course a READ of a ROM location will return the contents of the ROM, not the "hidden" RAM.

COMMODORE 64 FUNDAMENTAL MEMORY MAP



E000-FFFF	8K KERNAL ROM OR RAM
D000-DFFF	4K I/O OR RAM OR CHARACTER ROM
C000-CFFF	4K RAM
A000-BFFF	8K BASIC ROM OR RAM OR ROM PLUG-IN
8000-9FFF	8K RAM OR ROM PLUG-IN
4000-7FFF	16K RAM
0000-3FFF	16K RAM

I/O BREAKDOWN

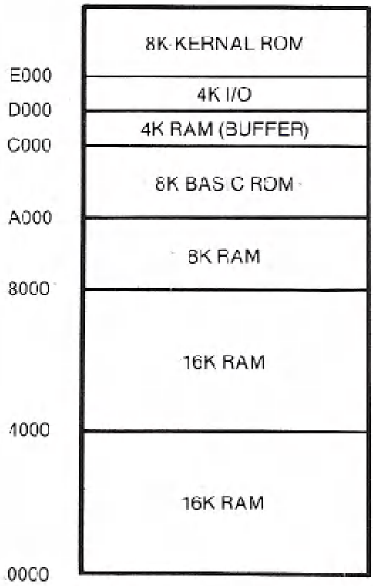
D000-D3FF	VIC (Video Controller)	1K Bytes
D400-D7FF	SID (Sound Synthesizer)	1K Bytes
D800-DBFF	Color RAM	1K Nybbles
DC00-DCFF	CIA1 (Keyboard)	256 Bytes
DD00-DDFF	CIA2 (Serial Bus, User Port/RS-232)	256 Bytes
DE00-DEFF	Open I/O slot #1 (CP/M Enable)	256 Bytes
DF00-DFFF	Open I/O slot #2 (Disk)	256 Bytes

The two open I/O slots are for general purpose user I/O, special purpose I/O cartridges (such as IEEE), and have been tentatively designated for enabling the Z-80 cartridge (CP/M option) and for interfacing to a low-cost high-speed disk system.

The system provides for "auto-start" of the program in a Commodore 64 Expansion Cartridge. The cartridge program is started if the first nine bytes of the cartridge ROM starting at location 32768 ($8000) contain specific data. The first two bytes must hold the Cold Start vector to be used by the cartridge program. The next two bytes at 32770 ($8002) must be the Warm Start vector used by the cartridge program. The next three bytes must be the letters, CBM, with bit 7 set in each letter. The last two bytes must be the digits "80" in PET ASCII.

COMMODORE 64 MEMORY MAPS

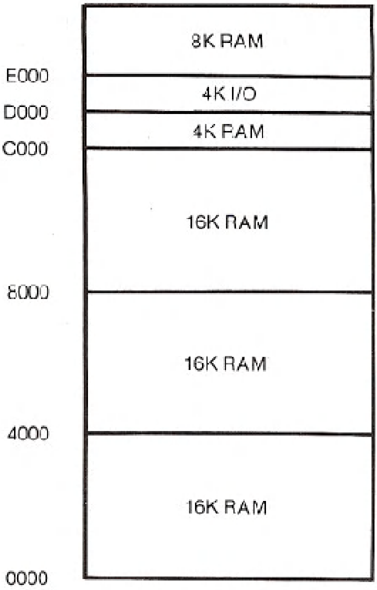
The following tables list the various memory configurations available on the COMMODORE 64, the states of the control lines which select each memory map, and the intended use of each map.



X = DON'T CARE
0 = LOW
1 = HIGH

LORAM	= 1
HIRAM	= 1
GAME	= 1
EXROM	= 1

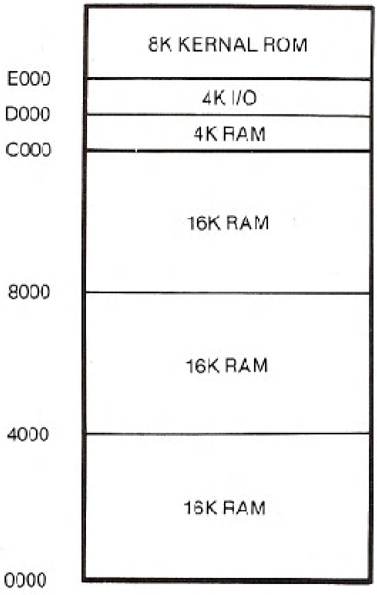
This is the default BASIC memory map which provides BASIC 2.0 and 38K contiguous bytes of user RAM.



X = DON'T CARE
0 = LOW
1 = HIGH

LORAM = 1
HIRAM = 0
GAME = 1
EXROM = X
OR
LORAM = 1
HIRAM = 0
GAME = 0
(THE CHARACTER ROM
IS NOT ACCESSIBLE BY
THE CPU IN THIS MAP)
EXROM = 0

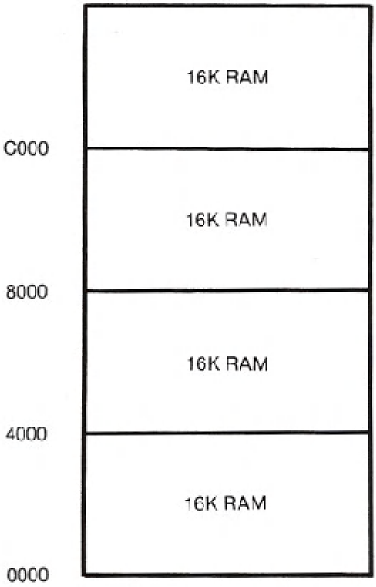
This map provides 60K bytes of RAM and I/O devices. The user must write his own I/O driver routines.



X = DON'T CARE
0 = LOW
1 = HIGH

LORAM = 0
HIRAM = 1
GAME = 1
EXROM = X

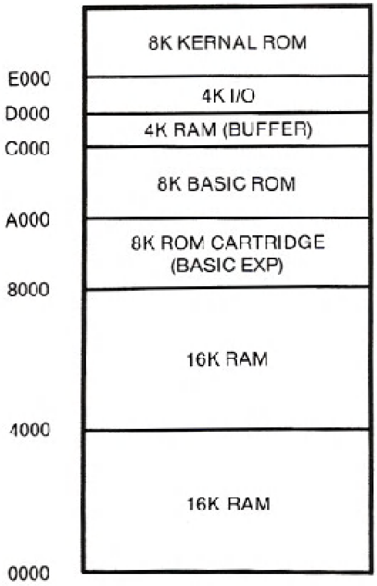
This map is intended for use with softload languages (including CP/M), providing 52K contiguous bytes of user RAM, I/O devices, and I/O driver routines.



X = DON'T CARE
0 = LOW
1 = HIGH

LORAM = 0
HIRAM = 0
GAME = 1
EXROM = X
OR
LORAM = 0
HIRAM = 0
GAME = X
EXROM = 0

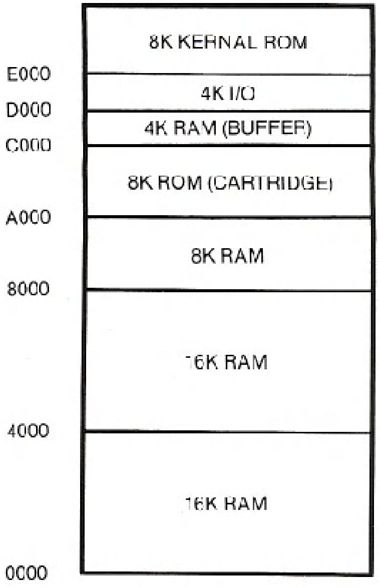
This map gives access to all 64K bytes of RAM. The I/O devices must be banked back into the processor's address space for any I/O operation.



X = DON'T CARE
0 = LOW
1 = HIGH

LORAM = 1
HIRAM = 1
GAME = 1
EXROM = 0

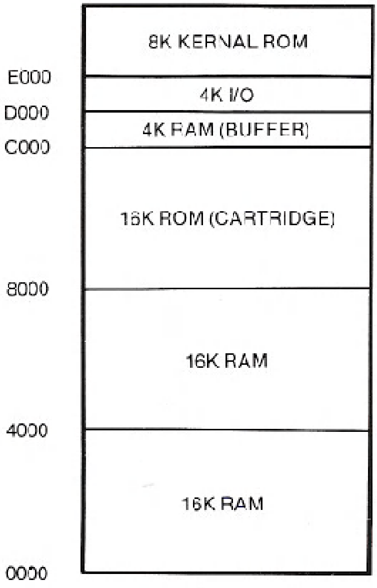
This is the standard configuration for a BASIC system with a BASIC expansion ROM. This map provides 32K contiguous bytes of user RAM and up to 8K bytes of BASIC "enhancement."



X = DON'T CARE
0 = LOW
1 = HIGH

LORAM = 0
HIRAM = 1
GAME = 0
EXROM = 0

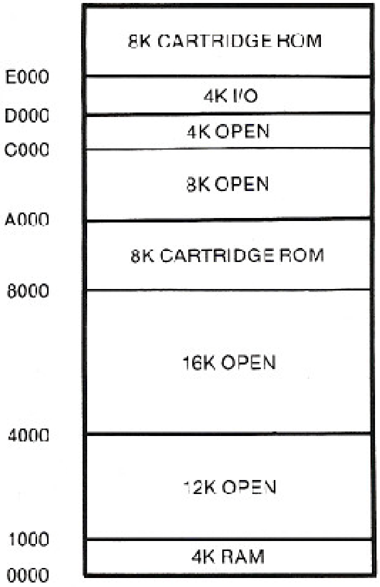
This map provides 40K contiguous bytes of user RAM and up to 8K bytes of plug-in ROM for special ROM-based applications which don't require BASIC.



X = DON'T CARE
0 = LOW
1 = HIGH

LORAM = 1
HIRAM = 1
GAME = 0
EXROM = 0

This map provides 32K contiguous bytes of user RAM and up to 16K bytes of plug-in ROM for special ROM-based applications which don't require BASIC (word processors, other languages, etc.).



X = DON'T CARE
0 = LOW
1 = HIGH

LORAM	= X
HIRAM	= X
GAME	= 0
EXROM	= 1

This is the ULTIMAX video game memory map. Note that the 2K byte "expansion RAM" for the ULTIMAX, if required, is accessed out of the COMMODORE 64 and any RAM in the cartridge is ignored.

THE KERNAL

One of the problems facing programmers in the microcomputer field is the question of what to do when changes are made to the operating system of the computer by the company. Machine language programs which took much time to develop might no longer work, forcing major revisions in the program. To alleviate this problem, Commodore has developed a method of protecting software writers called the **KERNAL**.

Essentially, the KERNAL is a standardized **JUMP TABLE** to the input, output, and memory management routines in the operating system. The locations of each routine in ROM may change as the system is upgraded. But the KERNAL jump table will always be changed to match. If your machine language routines only use the system ROM routines through the KERNAL, it will take much less work to modify them, should that need ever arise.

The KERNAL is the operating system of the Commodore 64 computer. All input, output, and memory management is controlled by the KERNAL.

To simplify the machine language programs you write, and to make sure that future versions of the Commodore 64 operating system don't make your machine language programs obsolete, the KERNAL contains a jump table for you to use. By taking advantage of the 39 input/output routines and other utilities available to you from the table, not only do you save time, you also make it easier to translate your programs from one Commodore computer to another.

The jump table is located on the last page of memory, in read-only memory (ROM).

To use the KERNAL jump table, first you set up the parameters that the KERNAL routine needs to work. Then **JSR** (Jump to SubRoutine) to the proper place in the KERNAL jump table. After performing its function, the KERNAL transfers control back to your machine language program. Depending on which KERNAL routine you are using, certain registers may pass parameters back to your program. The particular registers for each KERNAL routine may be found in the individual descriptions of the KERNAL subroutines.

A good question at this point is why use the jump table at all? Why not just JSR directly to the KERNAL subroutine involved? The jump table is used so that if the KERNAL or BASIC is changed, your machine language programs will still work. In future operating systems the routines may have their memory locations moved around to a different position in the memory map . . . but the jump table will still work correctly!

KERNAL POWER-UP ACTIVITIES

1) On power-up, the KERNAL first resets the stack pointer, and clears decimal mode.
2) The KERNAL then checks for the presence of an autostart ROM cartridge at location $8000 HEX (32768 decimal). If this is present, normal initialization is suspended, and control is transferred to the cartridge code. If an autostart ROM is not present, normal system initialization continues.
3) Next, the KERNAL initializes all INPUT/OUTPUT devices. The serial bus is initialized. Both 6526 CIA chips are set to the proper values for keyboard scanning, and the 60-Hz timer is activated. The SID chip is cleared. The BASIC memory map is selected and the cassette motor is switched off.
4) Next, the KERNAL performs a RAM test, setting the top and bottom of memory pointers. Also, page zero is initialized, and the tape buffer is set up.

   The RAM TEST routine is a nondestructive test starting at location $0300 and working upward. Once the test has found the first non-RAM location, the top of RAM has its pointer set. The bottom of memory is always set to $0800, and the screen setup is always set at $0400.
5) Finally, the KERNAL performs these other activities. I/O vectors are set to default values. The indirect jump table in low memory is established. The screen is then cleared, and all screen editor variables reset. Then the indirect at $A000 is used to start BASIC.

HOW TO USE THE KERNAL

When writing machine language programs it is often convenient to use the routines which are already part of the operating system for input/output, access to the system clock, memory management, and other similar operations. It is an unnecessary duplication of effort to write these routines over and over again, so easy access to the operating system helps speed machine language programming.

As mentioned before, the KERNAL is a jump table. This is just a collection of JMP instructions to many operating system routines.

To use a KERNAL routine you must first make all of the preparations that the routine demands. If one routine says that you must call another KERNAL routine first, then that routine must be called. If the routine expects you to put a number in the accumulator, then that number must be there. Otherwise your routines have little chance of working the way you expect them to work.

After all preparations are made, you must call the routine by means of the JSR instruction. All KERNAL routines you can access are structured as SUBROUTINES, and must end with an RTS instruction. When the KERNAL routine has finished its task, control is returned to your program at the instruction after the JSR.

Many of the KERNAL routines return error codes in the status word or the accumulator if you have problems in the routine. Good programming practice and the success of your machine language programs demand that you handle this properly. If you ignore an error return, the rest of your program might "bomb."

That's all there is to do when you're using the KERNAL. Just these three simple steps:

1) Set up
2) Call the routine
3) Error handling

The following conventions are used in describing the KERNAL routines:

—**FUNCTION NAME:** Name of the KERNAL routine.

—**CALL ADDRESS:** This is the call address of the KERNAL routine, given in hexadecimal.

—**COMMUNICATION REGISTERS:** Registers listed under this heading are used to pass parameters to and from the KERNAL routines.

—**PREPARATORY ROUTINES:** Certain KERNAL routines require that data be set up before they can operate. The routines needed are listed here.

—**ERROR RETURNS:** A return from a KERNAL routine with the CARRY set indicates that an error was encountered in processing. The accumulator will contain the number of the error.

—**STACK REQUIREMENTS:** This is the actual number of stack bytes used by the KERNAL routine.

—**REGISTERS AFFECTED:** All registers used by the KERNAL routine are listed here.

—**DESCRIPTION:** A short tutorial on the function of the KERNAL routine is given here.

The list of the KERNAL routines follows.

USER CALLABLE KERNAL ROUTINES

NAME	ADDRESS		FUNCTION
	HEX	DECIMAL	
ACPTR	$FFA5	65445	Input byte from serial port.
CHKIN	$FFC6	65478	Open channel for input
CHKOUT	$FFC9	65481	Open channel for output
CHRIN	$FFCF	65487	Input character from channel
CHROUT	$FFD2	65490	Output character to channel
CIOUT	$FFA8	65448	Output byte to serial port
CINT	$FF81	65409	Initialize screen editor
CLALL	$FFE7	65511	Close all channels and files
CLOSE	$FFC3	65475	Close a specified logical file
CLRCHN	$FFCC	65484	Close input and output channels
GETIN	$FFE4	65508	Get character from keyboard queue (keyboard buffer)
IOBASE	$FFF3	65523	Returns base address of I/O devices
IOINIT	$FF84	65412	Initialize input/output
LISTEN	$FFB1	65457	Command devices on the serial bus to LISTEN
LOAD	$FFD5	65493	Load RAM from a device
MEMBOT	$FF9C	65436	Read/set the bottom of memory
MEMTOP	$FF99	65433	Read/set the top of memory
OPEN	$FFC0	65472	Open a logical file

NAME	ADDRESS		FUNCTION
	HEX	DECIMAL	
PLOT	$FFF0	65520	Read/set X,Y cursor position
RAMTAS	$FF87	65415	Initialize RAM, allocate tape buffer, set screen S0400
RDTIM	$FFDE	65502	Read real time clock
READST	$FFB7	65463	Read I/O status word
RESTOR	$FF8A	65418	Restore default I/O vectors
SAVE	$FFD8	65496	Save RAM to device
SCNKEY	$FF9F	65439	Scan keyboard
SCREEN	$FFED	65517	Return X,Y organization of screen
SECOND	$FF93	65427	Send secondary address after LISTEN
SETLFS	$FFBA	65466	Set logical, first, and second addresses
SETMSG	$FF90	65424	Control KERNAL messages
SETNAM	$FFBD	65469	Set file name
SETTIM	$FFDB	65499	Set real time clock
SETTMO	$FFA2	65442	Set timeout on serial bus
STOP	$FFE1	65505	Scan stop key
TALK	$FFB4	65460	Command serial bus device to TALK
TKSA	$FF96	65430	Send secondary address after TALK
UDTIM	$FFEA	65514	Increment real time clock
UNLSN	$FFAE	65454	Command serial bus to UNLISTEN
UNTLK	$FFAB	65451	Command serial bus to UNTALK
VECTOR	$FF8D	65421	Read/set vectored I/O

B-1. Function Name: ACPTR

Purpose: Get data from the serial bus
Call address: $FFA5 (hex) 65445 (decimal)
Communication registers: .A
Preparatory routines: TALK, TKSA
Error returns: See READST
Stack requirements: 13
Registers affected: .A, .X

Description: This is the routine to use when you want to get information from a device on the serial bus, like a disk. This routine gets a byte of data off the serial bus using full handshaking. The data is returned in the accumulator. To prepare for this routine the TALK routine must be called first to command the device on the serial bus to send data through the bus. If the input device needs a secondary command, it must be sent by using the TKSA KERNAL routine before calling this routine. Errors are returned in the status word. The READST routine is used to read the status word.

How to Use:

0) Command a device on the serial bus to prepare to send data to the Commodore 64. (Use the TALK and TKSA KERNAL routines.)
1) Call this routine (using JSR).
2) Store or otherwise use the data.

EXAMPLE:

```
;GET A BYTE FROM THE BUS
JSR ACPTR
STA DATA
```

B-2. Function Name: CHKIN

Purpose: Open a channel for input
Call address: $FFC6 (hex) 65478 (decimal)
Communication registers: .X
Preparatory routines: (OPEN)
Error returns:
Stack requirements: None
Registers affected: .A, .X

Description: Any logical file that has already been opened by the KERNAL OPEN routine can be defined as an input channel by this routine. Naturally, the device on the channel must be an input device. Otherwise an error will occur, and the routine will abort.

If you are getting data from anywhere other than the keyboard, this routine must be called before using either the CHRIN or the GETIN KERNAL routines for data input. If you want to use the input from the keyboard, and no other input channels are opened, then the calls to this routine, and to the OPEN routine are not needed.

When this routine is used with a device on the serial bus, it automatically sends the talk address (and the secondary address if one was specified by the OPEN routine) over the bus.

How to Use:

0) OPEN the logical file (if necessary; see description above).
1) Load the .X register with number of the logical file to be used.
2) Call this routine (using a JSR command).

Possible errors are:

#3: File not open
#5: Device not present
#6: File not an input file

EXAMPLE:

```
; PREPARE FOR INPUT FROM LOGICAL FILE 2
LDX #2
JSR CHKIN
```

B-3. Function Name: CHKOUT

Purpose: Open a channel for output
Call address: $FFC9 (hex) 65481 (decimal)
Communication registers: .X
Preparatory routines: (OPEN)
Error returns: 0,3,5,7 (See READST)
Stack requirements: 4+
Registers affected: .A, .X

Description: Any logical file number that has been created by the KERNAL routine OPEN can be defined as an output channel. Of course, the device you intend opening a channel to must be an output device. Otherwise an error will occur, and the routine will be aborted.

This routine must be called before any data is sent to any output device unless you want to use the Commodore 64 screen as your output device. If screen output is desired, and there are no other output channels already defined, then calls to this routine, and to the OPEN routine are not needed.

When used to open a channel to a device on the serial bus, this routine will automatically send the LISTEN address specified by the OPEN routine (and a secondary address if there was one).

How to Use:

REMEMBER: this routine is NOT NEEDED to send data to the screen.

0) Use the KERNAL OPEN routine to specify a logical file number, a LISTEN address, and a secondary address (if needed).
1) Load the .X register with the logical file number used in the open statement.
2) Call this routine (by using the JSR instruction).

EXAMPLE:

```
LDX #3          ;DEFINE LOGICAL FILE 3 AS AN OUTPUT CHANNEL
JSR CHKOUT
```

Possible errors are:

#3: File not open
#5: Device not present
#7: Not an output file

B-4. Function Name: CHRIN

Purpose: Get a character from the input channel
Call address: $FFCF (hex) 65487 (decimal)
Communication registers: .A
Preparatory routines: (OPEN, CHKIN)
Error returns: 0 (See READST)
Stack requirements: 7+
Registers affected: .A, .X

Description: This routine gets a byte of data from a channel already set up as the input channel by the KERNAL routine CHKIN. If the CHKIN has NOT been used to define another input channel, then all your data is expected from the keyboard. The data byte is returned in the accumulator. The channel remains open after the call.

Input from the keyboard is handled in a special way. First, the cursor is turned on, and blinks until a carriage return is typed on the keyboard. All characters on the line (up to 88 characters) are stored in the BASIC input buffer. These characters can be retrieved one at a time by calling this routine once for each character. When the carriage return is retrieved, the entire line has been processed. The next time this routine is called, the whole process begins again, i.e., by flashing the cursor.

How to Use:

FROM THE KEYBOARD

1) Retrieve a byte of data by calling this routine.
2) Store the data byte.
3) Check if it is the last data byte (is it a CR ?).
4) If not, go to step 1.

EXAMPLE:

```
     LDY $#00       ;PREPARE THE .Y REGISTER TO STORE THE DATA
RD   JSR CHRIN
     STA DATA,Y     ;STORE THE YTH DATA BYTE IN THE YTH
                    ;LOCATION IN THE DATA AREA.
     INY
     CMP #CR        ;IS IT A CARRIAGE RETURN?
     BNE RD         ;NO, GET ANOTHER DATA BYTE
```

EXAMPLE:

```
JSR CHRIN
STA DATA
```

FROM OTHER DEVICES

0) Use the KERNAL OPEN and CHKIN routines.
1) Call this routine (using a JSR instruction).
2) Store the data.

EXAMPLE:

```
JSR CHRIN
STA DATA
```

B-5. Function Name: CHROUT

Purpose: Output a character
Call address: $FFD2 (hex) 65490 (decimal)
Communication registers: .A
Preparatory routines: (CHKOUT,OPEN)
Error returns: 0 (See READST)
Stack requirements: 8+
Registers affected: .A

Description: This routine outputs a character to an already opened channel. Use the KERNAL OPEN and CHKOUT routines to set up the output channel before calling this routine. If this call is omitted, data is sent to the default output device (number 3, the screen). The data byte to be output is loaded into the accumulator, and this routine is called. The data is then sent to the specified output device. The channel is left open after the call.

NOTE: Care must be taken when using this routine to send data to a specific serial device since data will be sent to all open output channels on the bus. Unless this is desired, all open output channels on the serial bus other than the intended destination channel must be closed by a call to the KERNAL CLRCHN routine.

How to Use:

0) Use the CHKOUT KERNAL routine if needed (see description above).
1) Load the data to be output into the accumulator.
2) Call this routine.

EXAMPLE:

```
;DUPLICATE THE BASIC INSTRUCTION CMD 4,"A";
LDX #4                                  ;LOGICAL FILE #4
JSR CHKOUT                              ;OPEN CHANNEL OUT
LDA #'A
JSR CHROUT                              ;SEND CHARACTER
```

B-6. Function Name: CIOUT

Purpose: Transmit a byte over the serial bus
Call address: $FFA8 (hex) 65448 (decimal)
Communication registers: .A
Preparatory routines: LISTEN, [SECOND]
Error returns: See READST
Stack requirements: 5
Registers affected: None

Description: This routine is used to send information to devices on the serial bus. A call to this routine will put a data byte onto the serial bus using full serial handshaking. Before this routine is called, the LISTEN KERNAL routine must be used to command a device on the serial bus to get ready to receive data. (If a device needs a secondary address, it must also be sent by using the SECOND KERNAL routine.) The accumulator is loaded with a byte to handshake as data on the serial bus. A device must be listening or the status word will return a timeout. This routine always buffers one character. (The routine holds the previous character to be sent back.) So when a call to the KERNAL UNLSN routine is made to end the data transmission, the buffered character is sent with an End Or Identify (EOI) set. Then the UNLSN command is sent to the device.

How to Use:

0) Use the LISTEN KERNAL routine (and the SECOND routine if needed).
1) Load the accumulator with a byte of data.
2) Call this routine to send the data byte.

EXAMPLE:

```
LDA #'X                         ;SEND AN X TO THE SERIAL BUS
JSR CIOUT
```

B-7. Function Name: CINT

Purpose: Initialize screen editor & 6567 video chip
Call address: $FF81 (hex) 65409 (decimal)
Communication registers: None
Preparatory routines: None
Error returns: None
Stack requirements: 4
Registers affected: .A, .X, .Y

Description: This routine sets up the 6567 video controller chip in the Commodore 64 for normal operation. The KERNAL screen editor is also initialized. This routine should be called by a Commodore 64 program cartridge.

How to Use:

1) Call this routine.

EXAMPLE:

```
JSR CINT
JMP RUN                                   ;BEGIN EXECUTION
```

B-8. Function Name: CLALL

Purpose: Close all files
Call address: $FFE7 (hex) 65511 (decimal)
Communication registers: None
Preparatory routines: None
Error returns: None
Stack requirements: 11
Registers affected: .A, .X

Description: This routine closes all open files. When this routine is called, the pointers into the open file table are reset, closing all files. Also, the CLRCHN routine is automatically called to reset the I/O channels.

How to Use:

1) Call this routine.

EXAMPLE:

```
JSR CLALL ;CLOSE ALL FILES AND SELECT DEFAULT I/O CHANNELS
JMP RUN  ;BEGIN EXECUTION
```

B-9. Function Name: CLOSE

Purpose: Close a logical file
Call address: $FFC3 (hex) 65475 (decimal)
Communication registers: .A
Preparatory routines: None
Error returns: 0,240 (See READST)
Stack requirements: 2+
Registers affected: .A, .X, .Y

Description: This routine is used to close a logical file after all I/O operations have been completed on that file. This routine is called after the accumulator is loaded with the logical file number to be closed (the same number used when the file was opened using the OPEN routine).

How to Use:

1) Load the accumulator with the number of the logical file to be closed.
2) Call this routine.

EXAMPLE:

```
;CLOSE 15
LDA #15
JSR CLOSE
```

B-10. Function Name: CLRCHN

Purpose: Clear I/O channels
Call address: $FFCC (hex) 65484 (decimal)
Communication registers: None
Preparatory routines: None
Error returns:
Stack requirements: 9
Registers affected: .A, .X

Description: This routine is called to clear all open channels and restore the I/O channels to their original default values. It is usually called after opening other I/O channels (like a tape or disk drive) and using them for input/output operations. The default input device is 0 (keyboard). The default output device is 3 (the Commodore 64 screen).

If one of the channels to be closed is to the serial port, an UNTALK signal is sent first to clear the input channel or an UNLISTEN is sent to clear the output channel. By not calling this routine (and leaving listener(s) active on the serial bus) several devices can receive the same data from the Commodore 64 at the same time. One way to take advantage of this would be to command the printer to TALK and the disk to LISTEN. This would allow direct printing of a disk file.

This routine is automatically called when the KERNAL CLALL routine is executed.

How to Use:

1) Call this routine using the JSR instruction.

EXAMPLE:

```
JSR CLRCHN
```

B-11. Function Name: GETIN

Purpose: Get a character
Call address: $FFE4 (hex) 65508 (decimal)
Communication registers: .A
Preparatory routines: CHKIN, OPEN
Error returns: See READST
Stack requirements: 7+
Registers affected: .A (.X, .Y)

Description: If the channel is the keyboard, this subroutine removes one character from the keyboard queue and returns it as an ASCII value in the accumulator. If the queue is empty, the value returned in the accumulator will be zero. Characters are put into the queue automatically by an interrupt driven keyboard scan routine which calls the SCNKEY routine. The keyboard buffer can hold up to ten characters. After the buffer is filled, additional characters are ignored until at least one character has been removed from the queue. If the channel is RS-232, then only the .A register is used and a single character is returned. See READST to check validity. If the channel is serial, cassette, or screen, call BASIN routine.

How to Use:

1) Call this routine using a JSR instruction.
2) Check for a zero in the accumulator (empty buffer).
3) Process the data.

EXAMPLE:

```
;WAIT FOR A CHARACTER
WAIT JSR GETIN
CMP #0
BEQ WAIT
```

B-12. Function Name: IOBASE

Purpose: Define I/O memory page
Call address: $FFF3 (hex) 65523 (decimal)
Communication registers: .X, .Y
Preparatory routines: None
Error returns:
Stack requirements: 2
Registers affected: .X, .Y

Description: This routine sets the X and Y registers to the address of the memory section where the memory mapped I/O devices are located. This address can then be used with an offset to access the memory mapped I/O devices in the Commodore 64. The offset is the number of locations from the beginning of the page on which the I/O register you want is located. The .X register contains the low order address byte, while the .Y register contains the high order address byte.

This routine exists to provide compatibility between the Commodore 64, VIC-20, and future models of the Commodore 64. If the I/O locations for a machine language program are set by a call to this routine, they should still remain compatible with future versions of the Commodore 64, the KERNAL and BASIC.

How to Use:

1) Call this routine by using the JSR instruction.
2) Store the .X and the .Y registers in consecutive locations.
3) Load the .Y register with the offset.
4) Access that I/O location.

EXAMPLE:

```
; SET THE DATA DIRECTION REGISTER OF THE USER PORT TO 0 (INPUT)
JSR IOBASE
STX POINT     ;SET BASE REGISTERS
STY POINT+1
LDY #2
LDA #0        ;OFFSET FOR DDR OF THE USER PORT
STA (POINT), Y ;SET DDR TO 0
```

B-13. Function Name: IOINIT

Purpose: Initialize I/O devices
Call Address: $FF84 (hex) 65412 (decimal)
Communication registers: None
Preparatory routines: None
Error returns:
Stack requirements: None
Registers affected: .A, .X, .Y

Description: This routine initializes all input/output devices and routines. It is normally called as part of the initialization procedure of a Commodore 64 program cartridge.

EXAMPLE:

```
JSR IOINIT
```

B-14. Function Name: LISTEN

Purpose: Command a device on the serial bus to listen
Call Address: $FFB1 (hex) 65457 (decimal)
Communication registers: .A
Preparatory routines: None
Error returns: See READST
Stack requirements: None
Registers affected: .A

Description: This routine will command a device on the serial bus to receive data. The accumulator must be loaded with a device number between 0 and 31 before calling the routine. LISTEN will OR the number bit by bit to convert to a listen address, then transmits this data as a command on the serial bus. The specified device will then go into listen mode, and be ready to accept information.

How to Use:

1) Load the accumulator with the number of the device to command to LISTEN.
2) Call this routine using the JSR instruction.

EXAMPLE:

```
;COMMAND DEVICE #8 TO LISTEN
LDA #8
JSR LISTEN
```

B-15. Function Name: LOAD

Purpose: Load RAM from device
Call address: $FFD5 (hex) 65493 (decimal)
Communication registers: .A,.X,.Y
Preparatory routines: SETLFS, SETNAM
Error returns: 0,4,5,8,9, READST
Stack requirements: None
Registers affected: .A, .X, .Y

Description: This routine LOADs data bytes from any input device directly into the memory of the Commodore 64. It can also be used for a verify operation, comparing data from a device with the data already in memory, while leaving the data stored in RAM unchanged.

The accumulator (.A) must be set to 0 for a LOAD operation, or 1 for a verify. If the input device is OPENed with a secondary address (SA) of 0 the header information from the device is ignored. In this case, the .X and .Y registers must contain the starting address for the load. If the device is addressed with a secondary address of 1, then the data is loaded into memory starting at the location specified by the header. This routine returns the address of the highest RAM location loaded.

Before this routine can be called, the KERNAL SETLFS, and SETNAM routines must be called.

NOTE: You can NOT LOAD from the keyboard (0), RS-232 (2), or the screen (3).

How to Use:

0) Call the SETLFS, and SETNAM routines. If a relocated load is desired, use the SETLFS routine to send a secondary address of 0.
1) Set the .A register to 0 for load, 1 for verify.
2) If a relocated load is desired, the .X and .Y registers must be set to the start address for the load.
3) Call the routine using the JSR instruction.

EXAMPLE:

```
                ;LOAD A FILE FROM TAPE
                LDA     #DEVICE1        ;SET DEVICE NUMBER
                LDX     #FILENO         ;SET LOGICAL FILE NUMBER
                LDY     CMD1            ;SET SECONDARY ADDRESS
                JSR     SETLFS
                LDA     #NAME1-NAME     ;LOAD .A WITH NUMBER OF
                                        ;CHARACTERS IN FILE NAME
                LDX     #<NAME          ;LOAD .X AND .Y WITH
                                        ;ADDRESS OF
                LDY     #>NAME          ;FILE NAME
                JSR     SETNAM
                LDA     #0              ;SET FLAG FOR A LOAD
                LDX     #$FF            ;ALTERNATE START
                LDY     #$FF
                JSR     LOAD
                STX     VARTAB          ;END OF LOAD
                STY     VARTAB+1
                JMP     START
NAME            .BYT    'FILE NAME'
NAME 1          ;
```

B-16. Function Name: MEMBOT

Purpose: Set bottom of memory
Call address: $FF9C (hex) 65436 (decimal)
Communication registers: .X,.Y
Preparatory routines: None
Error returns: None
Stack requirements: None
Registers affected: .X, .Y

Description: This routine is used to set the bottom of the memory. If the accumulator carry bit is set when this routine is called, a pointer to the lowest byte of RAM is returned in the .X and .Y registers. On the unexpanded Commodore 64 the initial value of this pointer is $0800 (2048 in decimal). If the accumulator carry bit is clear (=0) when this routine is called, the values of the .X and .Y registers are transferred to the low and high bytes, respectively, of the pointer to the beginning of RAM.

How to Use:

TO READ THE BOTTOM OF RAM

1) Set the carry.
2) Call this routine.

TO SET THE BOTTOM OF MEMORY

1) Clear the carry.
2) Call this routine.

EXAMPLE:

```
; MOVE BOTTOM OF MEMORY UP 1 PAGE
SEC              ;READ MEMORY BOTTOM
JSR MEMBOT
INY
CLC              ;SET MEMORY BOTTOM TO NEW VALUE
JSR MEMBOT
```

B-17. Function Name: MEMTOP

Purpose: Set the top of RAM
Call address: $FF99 (hex) 65433 (decimal)
Communication registers: .X, .Y
Preparatory routines: None
Error returns: None
Stack requirements: 2
Registers affected: .X, .Y

Description: This routine is used to set the top of RAM. When this routine is called with the carry bit of the accumulator set, the pointer to the top of RAM will be loaded into the .X and .Y registers. When this routine is called with the accumulator carry bit clear, the contents of the .X and .Y registers are loaded in the top of memory pointer, changing the top of memory.

EXAMPLE:

```
;DEALLOCATE THE RS-232 BUFFER
SEC
JSR MEMTOP   ;READ TOP OF MEMORY
DEX
CLC
JSR MEMTOP   ;SET NEW TOP OF MEMORY
```

B-18. Function Name: OPEN

Purpose: Open a logical file
Call address: $FFC0 (hex) 65472 (decimal)
Communication registers: None
Preparatory routines: SETLFS, SETNAM
Error returns: 1,2,4,5,6,240, READST
Stack requirements: None
Registers affected: .A, .X, .Y

Description: This routine is used to OPEN a logical file. Once the logical file is set up, it can be used for input/output operations. Most of the I/O KERNAL routines call on this routine to create the logical files to operate on. No arguments need to be set up to use this routine, but both the SETLFS and SETNAM KERNAL routines must be called before using this routine.

How to Use:

0) Use the SETLFS routine.
1) Use the SETNAM routine.
2) Call this routine.

EXAMPLE:

This is an implementation of the BASIC statement: OPEN 15,8,15,"I/ O"

```
        LDA #NAME2-NAME   ;LENGTH OF FILE NAME FOR SETLFS
        LDY #>NAME        ;ADDRESS OF FILE NAME
        LDX #<NAME
        JSR SETNAM
        LDA #15
        LDX #8
        LDY #15
        JSR SETLFS
        JSR OPEN
NAME    .BYT 'I/O'
NAME2
```

B-19. Function Name: PLOT

Purpose: Set cursor location

Call address: $FFF0 (hex) 65520 (decimal)

Communication registers: .A, .X, .Y

Preparatory routines: None

Error returns: None

Stack requirements: 2

Registers affected: .A, .X, .Y

Description: A call to this routine with the accumulator carry flag set loads the current position of the cursor on the screen (in X,Y coordinates) into the .Y and .X registers. Y is the column number of the cursor location (6–39), and X is the row number of the location of the cursor (0–24). A call with the carry bit clear moves the cursor to X,Y as determined by the .Y and .X registers.

How to Use:

READING CURSOR LOCATION

1) Set the carry flag.
2) Call this routine.
3) Get the X and Y position from the .Y and .X registers, respectively.

SETTING CURSOR LOCATION

1) Clear carry flag.
2) Set the .Y and .X registers to the desired cursor location.
3) Call this routine.

EXAMPLE:

```
; MOVE THE CURSOR TO ROW 10, COLUMN 5 (5,10)
LDX #10
LDY #5
CLC
JSR PLOT
```

B-20. Function Name: RAMTAS

Purpose: Perform RAM test
Call address: $FF87 (hex) 65415 (decimal)
Communication registers: .A, .X, .Y
Preparatory routines: None
Error returns: None
Stack requirements: 2
Registers affected: .A, .X, .Y

Description: This routine is used to test RAM and set the top and bottom of memory pointers accordingly. It also clears locations $0000 to $0101 and $0200 to $03FF. It also allocates the cassette buffer, and sets the screen base to $0400. Normally, this routine is called as part of the initialization process of a Commodore 64 program cartridge.

EXAMPLE:

```
JSR RAMTAS
```

B-21. Function Name: RDTIM

Purpose: Read system clock
Call address: SFFDE (hex) 65502 (decimal)
Communication registers: .A, .X, .Y
Preparatory routines: None
Error returns: None
Stack requirements: 2
Registers affected: .A, .X, .Y

Description: This routine is used to read the system clock. The clock's resolution is a 60th of a second. Three bytes are returned by the routine. The accumulator contains the most significant byte, the X index register contains the next most significant byte, and the Y index register contains the least significant byte.

EXAMPLE:

```
JSR RDTIM
STY TIME
STX TIME+1
STA TIME+2
. . .
TIME *=*+3
```

B-22. Function Name: READST

Purpose: Read status word
Call address: $FFB7 (hex) 65463 (decimal)
Communication registers: .A
Preparatory routines: None
Error returns: None
Stack requirements: 2
Registers affected: .A

Description: This routine returns the current status of the I/O devices in the accumulator. The routine is usually called after new communication to an I/O device. The routine gives you information about device status, or errors that have occurred during the I/O operation.

The bits returned in the accumulator contain the following information: (see table below)

ST BIT POSITION	ST NUMERIC VALUE	CASSETTE READ	SERIAL/RW	TAPE VERIFY + LOAD
0	1		Time out write	
1	2		Time out read	
2	4	Short block		Short block
3	8	Long block		Long block
4	16	Unrecoverable read error		Any mismatch
5	32	Checksum error		Checksum error
6	64	End of file	EOI line	
7	−128	End of tape	Device not present	End of tape

How to Use:

1) Call this routine.
2) Decode the information in the .A register as it refers to your program.

EXAMPLE:

```
;CHECK FOR END OF FILE DURING READ
JSR READST
AND #64                        ;CHECK EOF BIT (EOF=END OF FILE)
BNE EOF                        ;BRANCH ON EOF
```

B-23. Function Name: RESTOR

Purpose: Restore default system and interrupt vectors
Call address: $FF8A (hex) 65418 (decimal)
Preparatory routines: None
Error returns: None
Stack requirements: 2
Registers affected: .A, .X, .Y

Description: This routine restores the default values of all system vectors used in KERNAL and BASIC routines and interrupts. (See the Memory Map for the default vector contents). The KERNAL VECTOR routine is used to read and alter individual system vectors.

How to Use:

1) Call this routine.

EXAMPLE:

```
JSR RESTOR
```

B-24. Function Name: SAVE

Purpose: Save memory to a device
Call address: $FFD8 (hex) 65496 (decimal)
Communication registers: .A,.X,.Y
Preparatory routines: SETLFS, SETNAM
Error returns: 5,8,9, READST
Stack requirements: None
Registers affected: .A, .X, .Y

Description: This routine saves a section of memory. Memory is saved from an indirect address on page 0 specified by the accumulator to the address stored in the .X and .Y registers. It is then sent to a logical file on an input/output device. The SETLFS and SETNAM routines must be used before calling this routine. However, a file name is not required to SAVE to device 1 (the Datassette^{TM} recorder). Any attempt to save to other devices without using a file name results in an error.

> **NOTE:** Device 0 (the keyboard), device 2 (RS-232), and device 3 (the screen) cannot be SAVEd to. If the attempt is made, an error occurs, and the SAVE is stopped.

How to Use:

0) Use the SETLFS routine and the SETNAM routine (unless a SAVE with no file name is desired on "a save to the tape recorder").
1) Load two consecutive locations on page 0 with a pointer to the start of your save (in standard 6502 low byte first, high byte next format).
2) Load the accumulator with the single byte page zero offset to the pointer.
3) Load the .X and .Y registers with the low byte and high byte respectively of the location of the end of the save.
4) Call this routine.

EXAMPLE:

```
LDA #1            ;DEVICE=1:CASSETTE
JSR SETLFS
LDA #0            ;NO FILE NAME
JSR SETNAM
LDA PROG          ;LOAD START ADDRESS OF SAVE
STA TXTTAB        ; (LOW BYTE)
LDA PROG+1
STA TXTTAB+1      ; (HIGH BYTE)
LDX VARTAB        ;LOAD .X WITH LOW BYTE OF END OF SAVE
LDY VARTAB+1      ;LOAD .Y WITH HIGH BYTE
LDA #<TXTTAB      ;LOAD ACCUMULATOR WITH PAGE 0 OFFSET
JSR SAVE
```

B-25. Function Name: SCNKEY

Purpose: Scan the keyboard
Call address: $FF9F (hex) 65439 (decimal)
Communication registers: None
Preparatory routines: IOINIT
Error returns: None
Stack requirements: 5
Registers affected: .A, .X, .Y

Description: This routine scans the Commodore 64 keyboard and checks for pressed keys. It is the same routine called by the interrupt handler. If a key is down, its ASCII value is placed in the keyboard queue. This routine is called only if the normal IRQ interrupt is bypassed.

How to Use:

1) Call this routine.

EXAMPLE:

```
GET   JSR SCNKEY     ;SCAN KEYBOARD
      JSR GETIN      ;GET CHARACTER
      CMP #0         ;IS IT NULL?
      BEQ GET        ;YES . . . SCAN AGAIN
      JSR CHROUT     ;PRINT IT
```

B-26. Function Name: SCREEN

Purpose: Return screen format
Call address: $FFED (hex) 65517 (decimal)
Communication registers: .X,.Y
Preparatory routines: None
Stack requirements: 2
Registers affected: .X, .Y

Description: This routine returns the format of the screen, e.g., 40 columns in .X and 25 lines in .Y. The routine can be used to determine what machine a program is running on. This function has been implemented on the Commodore 64 to help upward compatibility of your programs.

How to Use:

1) Call this routine.

EXAMPLE:

```
JSR SCREEN
STX MAXCOL
STY MAXROW
```

B-27. Function Name: SECOND

Purpose: Send secondary address for LISTEN
Call address: $FF93 (hex) 65427 (decimal)
Communication registers: .A
Preparatory routines: LISTEN
Error returns: See READST
Stack requirements: 8
Registers affected: .A

Description: This routine is used to send a secondary address to an I/O device after a call to the LISTEN routine is made, and the device is commanded to LISTEN. The routine canNOT be used to send a secondary address after a call to the TALK routine.

A secondary address is usually used to give setup information to a device before I/O operations begin.

When a secondary address is to be sent to a device on the serial bus, the address must first be ORed with $60.

How to Use:

1) Load the accumulator with the secondary address to be sent.
2) Call this routine.

EXAMPLE:

```
;ADDRESS DEVICE #8 WITH COMMAND (SECONDARY ADDRESS) #15
LDA #8
JSR LISTEN
LDA #15
JSR SECOND
```

B-28. Function Name: SETLFS

Purpose: Set up a logical file
Call address: $FFBA (hex) 65466 (decimal)
Communication registers: .A, .X, .Y
Preparatory routines: None
Error returns: None
Stack requirements: 2
Registers affected: None

Description: This routine sets the logical file number, device address, and secondary address (command number) for other KERNAL routines.

The logical file number is used by the system as a key to the file table created by the OPEN file routine. Device addresses can range from 0 to 31. The following codes are used by the Commodore 64 to stand for the CBM devices listed below:

ADDRESS	DEVICE
0	Keyboard
1	Datassette^{TM} #1
2	RS-232C device
3	CRT display
4	Serial bus printer
8	CBM serial bus disk drive

Device numbers 4 or greater automatically refer to devices on the serial bus.

A command to the device is sent as a secondary address on the serial bus after the device number is sent during the serial attention handshaking sequence. If no secondary address is to be sent, the .Y index register should be set to 255.

How to Use:

1) Load the accumulator with the logical file number.
2) Load the .X index register with the device number.
3) Load the .Y index register with the command.

EXAMPLE:

```
FOR LOGICAL FILE 32, DEVICE #4, AND NO COMMAND:
LDA #32
LDX #4
LDY #255
JSR SETLFS
```

B-29. Function Name: SETMSG

Purpose: Control system message output
Call address: $FF90 (hex) 65424 (decimal)
Communication registers: .A
Preparatory routines: None
Error returns: None
Stack requirements: 2
Registers affected: .A

Description: This routine controls the printing of error and control messages by the KERNAL. Either print error messages or print control messages can be selected by setting the accumulator when the routine is called. FILE NOT FOUND is an example of an error message. PRESS PLAY ON CASSETTE is an example of a control message.

Bits 6 and 7 of this value determine where the message will come from. If bit 7 is 1, one of the error messages from the KERNAL is printed. If bit 6 is set, control messages are printed.

How to Use:

1) Set accumulator to desired value.
2) Call this routine.

EXAMPLE:

```
LDA #$40
JSR SETMSG                    ;TURN ON CONTROL MESSAGES
LDA #$80
JSR SETMSG                    ;TURN ON ERROR MESSAGES
LDA #0
JSR SETMSG                    ;TURN OFF ALL KERNAL MESSAGES
```

B-30. Function Name: SETNAM

Purpose: Set up file name
Call address: SFFBD (hex) 65469 (decimal)
Communication registers: .A, .X, .Y
Preparatory routines: None
Stack requirements: None
Registers affected: None

Description: This routine is used to set up the file name for the OPEN, SAVE, or LOAD routines. The accumulator must be loaded with the length of the file name. The .X and .Y registers must be loaded with the address of the file name, in standard 6502 low-byte/high-byte format. The address can be any valid memory address in the system where a string of characters for the file name is stored. If no file name is desired, the accumulator must be set to 0, representing a zero file length. The .X and .Y registers can be set to any memory address in that case.

How to Use:

1) Load the accumulator with the length of the file name.
2) Load the .X index register with the low order address of the file name.
3) Load the .Y index register with the high order address.
4) Call this routine.

EXAMPLE:

```
LDA #NAME2-NAME        ;LOAD LENGTH OF FILE NAME
LDX #<NAME             ;LOAD ADDRESS OF FILE NAME
LDY #>NAME
JSR SETNAM
```

B-31. Function Name: SETTIM

Purpose: Set the system clock
Call address: $FFDB (hex) 65499 (decimal)
Communication registers: .A, .X, .Y
Preparatory routines: None
Error returns: None
Stack requirements: 2
Registers affected: None

Description: A system clock is maintained by an interrupt routine that updates the clock every 1/60th of a second (one "jiffy"). The clock is three bytes long, which gives it the capability to count up to 5,184,000 jiffies (24 hours). At that point the clock resets to zero. Before calling this routine to set the clock, the accumulator must contain the most significant byte, the .X index register the next most significant byte, and the .Y index register the least significant byte of the initial time setting (in jiffies).

How to Use:

1) Load the accumulator with the MSB of the 3-byte number to set the clock.
2) Load the .X register with the next byte.
3) Load the .Y register with the LSB.
4) Call this routine.

EXAMPLE:

```
;SET THE CLOCK TO 10 MINUTES = 3600 JIFFIES
LDA #0               ; MOST SIGNIFICANT
LDX #>3600
LDY #<3600           ; LEAST SIGNIFICANT
JSR SETTIM
```

B-32. Function Name: SETTMO

Purpose: Set IEEE bus card timeout flag
Call address: $FFA2 (hex) 65442 (decimal)
Communication registers: .A
Preparatory routines: None
Error returns: None
Stack requirements: 2
Registers affected: None

> **NOTE:** This routine is used ONLY with an IEEE add-on card!

Description: This routine sets the timeout flag for the IEEE bus. When the timeout flag is set, the Commodore 64 will wait for a device on the IEEE port for 64 milliseconds. If the device does not respond to the Commodore 64's Data Address Valid (DAV) signal within that time the Commodore 64 will recognize an error condition and leave the handshake sequence. When this routine is called when the accumulator contains a 0 in bit 7, timeouts are enabled. A 1 in bit 7 will disable the timeouts.

NOTE: The Commodore 64 uses the timeout feature to communicate that a disk file is not found on an attempt to OPEN a file only with an IEEE card.

How to Use:

TO SET THE TIMEOUT FLAG

1) Set bit 7 of the accumulator to 0.
2) Call this routine.

TO RESET THE TIMEOUT FLAG

1) Set bit 7 of the accumulator to 1.
2) Call this routine.

EXAMPLE:

```
;DISABLE TIMEOUT
LDA #0
JSR SETTMO
```

B-33. Function Name: STOP

Purpose: Check if STOP key is pressed
Call address: $FFE1 (hex) 65505 (decimal)
Communication registers: .A
Preparatory routines: None
Error returns: None
Stack requirements: None
Registers affected: .A, .X

Description: If the STOP key on the keyboard was pressed during a UDTIM call, this call returns the Z flag set. In addition, the channels will be reset to default values. All other flags remain unchanged. If the STOP key is not pressed then the accumulator will contain a byte representing the last row of the keyboard scan. The user can also check for certain other keys this way.

How to Use:

0) UDTIM should be called before this routine.
1) Call this routine.
2) Test for the zero flag.

EXAMPLE:

```
JSR UDTIM ;SCAN FOR STOP
JSR STOP
BNE *+5   ;KEY NOT DOWN
JMP READY ;= . . . STOP
```

B-34. Function Name: TALK

Purpose: Command a device on the serial bus to TALK
Call address: $FFB4 (hex) 65460 (decimal)
Communication registers: .A
Preparatory routines: None
Error returns: See READST
Stack requirements: 8
Registers affected: .A

Description: To use this routine the accumulator must first be loaded with a device number between 0 and 31. When called, this routine then ORs bit by bit to convert this device number to a talk address. Then this data is transmitted as a command on the serial bus.

How to Use:

1) Load the accumulator with the device number.
2) Call this routine.

EXAMPLE:

```
;COMMAND DEVICE #4 TO TALK
LDA #4
JSR TALK
```

B-35. Function Name: TKSA

Purpose: Send a secondary address to a device commanded to TALK
Call address: $FF96 (hex) 65430 (decimal)
Communication registers: .A
Preparatory routines: TALK
Error returns: See READST
Stack requirements: 8
Registers affected: .A

Description: This routine transmits a secondary address on the serial bus for a TALK device. This routine must be called with a number between 0 and 31 in the accumulator. The routine sends this number as a secondary address command over the serial bus. This routine can only be called after a call to the TALK routine. It will not work after a LISTEN.

How to Use:

0) Use the TALK routine.
1) Load the accumulator with the secondary address.
2) Call this routine.

EXAMPLE:

```
;TELL DEVICE #4 TO TALK WITH COMMAND #7
LDA #4
JSR TALK
LDA #7
JSR TALKSA
```

B-36. Function Name: UDTIM

Purpose: Update the system clock
Call address: $FFEA (hex) 65514 (decimal)
Communication registers: None
Preparatory routines: None
Error returns: None
Stack requirements: 2
Registers affected: .A, .X

Description: This routine updates the system clock. Normally this routine is called by the normal KERNAL interrupt routine every 1/60th of a second. If the user program processes its own interrupts this routine must be called to update the time. In addition, the STOP key routine must be called, if the STOP key is to remain functional.

How to Use:

1) Call this routine.

EXAMPLE:

```
JSR UDTIM
```

B-37. Function Name: UNLSN

Purpose: Send an UNLISTEN command
Call address: $FFAE (hex) 65454 (decimal)
Communication registers: None
Preparatory routines: None
Error returns: See READST
Stack requirements: 8
Registers affected: .A

Description: This routine commands all devices on the serial bus to stop receiving data from the Commodore 64 (i.e., UNLISTEN). Calling this routine results in an UNLISTEN command being transmitted on the serial bus. Only devices previously commanded to listen are affected. This routine is normally used after the Commodore 64 is finished sending data to external devices. Sending the UNLISTEN commands the listening devices to get off the serial bus so it can be used for other purposes.

How to Use:

1) Call this routine.

EXAMPLE:

```
JSR UNLSN
```

B-38. Function Name: UNTLK

Purpose: Send an UNTALK command
Call address: $FFAB (hex) 65451 (decimal)
Communication registers: None
Preparatory routines: None
Error returns: See READST
Stack requirements: 8
Registers affected: .A

Description: This routine transmits an UNTALK command on the serial bus. All devices previously set to TALK will stop sending data when this command is received.

How to Use:

1) Call this routine.

EXAMPLE:

```
JSR UNTALK
```

B-39. Function Name: VECTOR

Purpose: Manage RAM vectors
Call address: $FFBD (hex) 65421 (decimal)
Communication registers: .X,.Y
Preparatory routines: None
Error returns: None
Stack requirements: 2
Registers affected: .A, .X, .Y

Description: This routine manages all system vector jump addresses stored in RAM. Calling this routine with the the accumulator carry bit set stores the current contents of the RAM vectors in a list pointed to by the .X and .Y registers. When this routine is called with the carry clar, the user list pointed to by the .X and .Y registers is transferred to the system RAM vectors. The RAM vectors are listed in the memory map.

NOTE: This routine requires caution in its use. The best way to use it is to first read the entire vector contents into the user area, alter the desired vectors, and then copy the contents back to the system vectors.

How to Use:

READ THE SYSTEM RAM VECTORS

1) Set the carry.
2) Set the .X and .y registers to the address to put the vectors.
3) Call this routine.

LOAD THE SYSTEM RAM VECTORS

1) Clear the carry bit.
2) Set the .X and .Y registers to the address of the vector list in RAM that must be loaded.
3) Call this routine.

EXAMPLE:

```
;CHANGE THE INPUT ROUTINES TO NEW SYSTEM
LDX #<USER
LDY #>USER
SEC
JSR VECTOR        ;READ OLD VECTORS
LDA #<MYINP       ;CHANGE INPUT
STA USER+10
LDA #>MYINP
STA USER+11
LDX #<USER
LDY #>USER
CLC
JSR VECTOR        ;ALTER SYSTEM
. . .
USER *=*+26
```

ERROR CODES

The following is a list of error messages which can occur when using the KERNAL routines. If an error occurs during a KERNAL routine, the carry bit of the accumulator is set, and the number of the error message is returned in the accumulator.

NOTE: Some KERNAL I/O routines do not use these codes for error messages. Instead, errors are identified using the KERNAL READST routine.

NUMBER	MEANING
0	Routine terminated by the STOP key
1	Too many open files
2	File already open
3	File not open
4	File not found
5	Device not present
6	File is not an input file
7	File is not an output file
8	File name is missing
9	Illegal device number
240	Top-of-memory change RS-232 buffer allocation/deallocation

USING MACHINE LANGUAGE FROM BASIC

There are several methods of using BASIC and machine language on the Commodore 64, including special statements as part of CBM BASIC as well as key locations in the machine. There are five main ways to use machine language routines from BASIC on the Commodore 64. They are:

1) The BASIC SYS statement
2) The BASIC USR function
3) Changing one of the RAM I/O vectors
4) Changing one of the RAM interrupt vectors
5) Changing the CHRGET routine

1) The BASIC statement **SYS X** causes a **JUMP** to a machine language subroutine located at address X. The routine must end with an **RTS** (ReTurn from Subroutine) instruction. This will transfer control back to BASIC.

   Parameters are generally passed between the machine language routine and the BASIC program using the BASIC PEEK and POKE statements, and their machine language equivalents.

   The **SYS** command is the most useful method of combining BASIC with machine language. PEEKs and POKEs make multiple parameter passing easy. There can be many SYS statements in a program, each to a different (or even the same) machine language routine.

2) The BASIC function **USR(X)** transfers control to the machine language subroutine located at the address stored in locations 785 and 786. (The address is stored in standard low-byte/high-byte format.) The value X is evaluated and passed to the machine language subroutine through floating point accumulator #1, located beginning at address $61 (see memory map for more details). A value may be returned back to the BASIC program by placing it in the floating point accumulator. The machine language routine must end with an RTS instruction to return to BASIC.

   This statement is different from the **SYS,** because you have to set up an indirect vector. Also different is the format through which the varicble is passed (floating point format). The indirect vector must be changed if more than one machine language routine is used.

3) Any of the input/output or BASIC internal routines accessed through the vector table located on page 3 (see **ADDRESSING MODES, ZERO PAGE**) can be replaced, or amended by user code. Each 2-byte vector consists of a low byte and a high byte address which is used by the operating system.

   The **KERNAL VECTOR** routine is the most reliable way to change any of the vectors, but a single vector can be changed by POKEs. A new vector will point to a user prepared routine which is meant to replace or augment the standard system routine. When the appropriate BASIC command is executed, the user routine will be executed. If after executing the user routine, it is necessary to execute the normal system routine, the user program must **JMP** (JuMP) to the address formerly contained in the vector. If not, the routine must end with a RTS to transfer control back to BASIC.

4) The **HARDWARE INTERRUPT (IRQ) VECTOR** can be changed. Every 1/60th of a second, the operating system transfers control to the routine specified by this vector. The KERNAL normally uses this for timing, keyboard scanning, etc. If this technique is used, you should always transfer control to the normal **IRQ** handling routine, unless the replacement routine is prepared to handle the CIA chip. (REMEMBER to end the routine with an **RTI** (ReTurn from Interrupt) if the CIA is handled by the routine).

   This method is useful for tasks which must happen concurrently with a BASIC program, but has the drawback of being more difficult.

NOTE: ALWAYS DISABLE INTERRUPTS BEFORE CHANGING THIS VECTOR!

5) The **CHRGET** routine is used by BASIC to get each character/token. This makes it simple to add new BASIC commands. Naturally, each new command must be executed by a user written machine language subroutine. A common way to use this method is to specify a character (@ for example) which will occur before any of the new commands. The new CHRGET routine will search for the special character. If none is present, control is passed to the normal BASIC CHRGET routine. If the special character is present, the new command is interpreted and executed by your machine language program. This minimizes the extra execution time added by the need to search for additional commands. This technique is often called a *wedge*.

WHERE TO PUT MACHINE LANGUAGE ROUTINES

The best place for machine language routines on the Commodore 64 is from $C000–$CFFF, assuming the routines are smaller than 4K bytes long. This section of memory is not disturbed by BASIC.

If for some reason it's not possible or desirable to put the machine language routine at $C000, for instance if the routine is larger than 4K bytes, it then becomes necessary to reserve an area at the top of memory from BASIC for the routine. The top of memory is normally $9FFF. The top of memory can be changed through the KERNAL routine **MEMTOP,** or by the following BASIC statements:

```
10 POKE51,L:POKE52,H:POKE55,L:POKE56,H:CLR
```

Where H and L are the high and low portions, respectively, of the new top of memory. For example, to reserve the area from $9000 to $9FFF for machine language, use the following:

```
10 POKE51,0:POKE52,144:POKE55,0:POKE56,144:CLR
```

HOW TO ENTER MACHINE LANGUAGE

There are 3 common methods to add the machine language programs to a BASIC program. They are:

1) DATA STATEMENTS:

By READing DATA statements, and POKEing the values into memory at the start of the program, machine language routines can be added. This is the easiest method. No special methods are needed to save the two parts of the program, and it is fairly easy to debug. The drawbacks include taking up more memory space, and the wait while the program is POKEd in. Therefore, this method is better for smaller routines.

EXAMPLE:

```
10 RESTORE:FORX=1TO9:READA:POKE12*4096+X,A:NEXT
.
.
.
BASIC PROGRAM
.
.
.
1000 DATA 161,1,204,204,204,204,204,204,96
```

2) MACHINE LANGUAGE MONITOR (64MON):

This program allows you to enter a program in either HEX or SYMBOLIC codes, and save the portion of memory the program is in. Advantages of this method include easier entry of the machine language routines, debugging aids, and a much faster means of saving and loading. The drawback to this method is that it generally requires the BASIC program to load the machine language routine from tape or disk when it is started. (For more details on 64MON see the machine language section.)

EXAMPLE:

The following is an example of a BASIC program using a machine language routine prepared by 64MON. The routine is stored on tape:

```
10 IF FLAG=1 THEN 20
15 FLAG=1:LOAD "MACHINE LANGUAGE ROUTINE NAME",1,1
20
.
.
.
REST OF BASIC PROGRAM
```

3) EDITOR/ASSEMBLER PACKAGE:

Advantages are similar to using a machine language monitor, but programs are even easier to enter. Disadvantages are also similar to the use of a machine language monitor.

COMMODORE 64 MEMORY MAP

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
D6510	0000	0	6510 On-Chip Data-Direction Register
R6510	0001	1	6510 On-Chip 8-Bit Input/Output Register
	0002	2	Unused
ADRAY1	0003–0004	3–4	Jump Vector: Convert Floating—Integer

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
ADRAY2	0005–0006	5–6	Jump Vector: Convert Integer—Floating
CHARAC	0007	7	Search Character
ENDCHR	0008	8	Flag: Scan for Quote at End of String
TRMPOS	0009	9	Screen Column From Last TAB
VERCK	000A	10	Flag: 0 = Load, 1 = Verify
COUNT	000B	11	Input Buffer Pointer / No. of Subscripts
DIMFLG	000C	12	Flag: Default Array DI-Mension
VALTYP	000D	13	Data Type: $FF = String, $00 = Numeric
INTFLG	000E	14	Data Type: $80 = Integer, $00 = Floating
GARBFL	000F	15	Flag: DATA scan/LIST quote/Garbage Coll
SUBFLG	0010	16	Flag: Subscript Ref / User Function Call
INPFLG	0011	17	Flag: $00 = INPUT, $40 = GET, $98 = READ
TANSGN	0012	18	Flag: TAN sign / Comparison Result
	0013	19	Flag: INPUT Prompt
LINNUM	0014–0015	20–21	Temp: Integer Value
TEMPPT	0016	22	Pointer: Temporary String Stack
LASTPT	0017–0018	23–24	Last Temp String Address
TEMPST	0019–0021	25–33	Stack for Temporary Strings
INDEX	0022–0025	34–37	Utility Pointer Area
RESHO	0026–002A	38–42	Floating-Point Product of Multiply
TXTTAB	002B–002C	43–44	Pointer: Start of BASIC Text

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
VARTAB	002D–002E	45–46	Pointer: Start of BASIC Variables
ARYTAB	002F–0030	47–48	Pointer: Start of BASIC Arrays
STREND	0031–0032	49–50	Pointer: End of BASIC Arrays (+1)
FRETOP	0033–0034	51–52	Pointer: Bottom of String Storage
FRESPC	0035–0036	53–54	Utility String Pointer
MEMSIZ	0037–0038	55–56	Pointer: Highest Address Used by BASIC
CURLIN	0039–003A	57–58	Current BASIC Line Number
OLDLIN	003B–003C	59–60	Previous BASIC Line Number
OLDTXT	003D–003E	61–62	Pointer: BASIC Statement for CONT
DATLIN	003F–0040	63–64	Current DATA Line Number
DATPTR	0041–0042	65–66	Pointer: Current DATA Item Address
INPPTR	0043–0044	67–68	Vector: INPUT Routine
VARNAM	0045–0046	69–70	Current BASIC Variable Name
VARPNT	0047–0048	71–72	Pointer: Current BASIC Variable Data
FORPNT	0049–004A	73–74	Pointer: Index Variable for FOR/NEXT
	004B–0060	75–96	Temp Pointer / Data Area
FACEXP	0061	97	Floating-Point Accumulator #1: Exponent
FACHO	0062–0065	98–101	Floating Accum. #1: Mantissa
FACSGN	0066	102	Floating Accum. #1: Sign
SGNFLG	0067	103	Pointer: Series Evaluation Constant

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
BITS	0068	104	Floating Accum. #1: Overflow Digit
ARGEXP	0069	105	Floating-Point Accumulator #2: Exponent
ARGHO	006A–006D	106–109	Floating Accum. #2: Mantissa
ARGSGN	006E	110	Floating Accum. #2: Sign
ARISGN	006F	111	Sign Comparison Result: Accum. #1 vs #2
FACOV	0070	112	Floating Accum. #1. Low-Order (Rounding)
FBUFPT	0071–0072	113–114	Pointer: Cassette Buffer
CHRGET	0073–008A	115–138	Subroutine: Get Next Byte of BASIC Text
CHRGOT	0079	121	Entry to Get Same Byte of Text Again
TXTPTR	007A–007B	122–123	Pointer: Current Byte of BASIC Text
RNDX	008B–008F	139–143	Floating RND Function Seed Value
STATUS	0090	144	Kernal I/O Status Word: ST
STKEY	0091	145	Flag: STOP key / RVS key
SVXT	0092	146	Timing Constant for Tape
VERCK	0093	147	Flag: 0 = Load, 1 = Verify
C3PO	0094	148	Flag: Serial Bus—Output Char. Buffered
BSOUR	0095	149	Buffered Character for Serial Bus
SYNO	0096	150	Cassette Sync No.
	0097	151	Temp Data Area
LDTND	0098	152	No. of Open Files / Index to File Table
DFLTN	0099	153	Default Input Device (0)
DFLTO	009A	154	Default Output (CMD) Device (3)

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
PRTY	009B	155	Tape Character Parity
DPSW	009C	156	Flag: Tape Byte-Received
MSGFLG	009D	157	Flag: $80 = Direct Mode, $00 = Program
PTR1	009E	158	Tape Pass 1 Error Log
PTR2	009F	159	Tape Pass 2 Error Log
TIME	00A0–00A2	160–162	Real-Time Jiffy Clock (approx) 1/60 Sec
	00A3–00A4	163–164	Temp Data Area
CNTDN	00A5	165	Cassette Sync Countdown
BUFPNT	00A6	166	Pointer: Tape I/O Buffer
INBIT	00A7	167	RS-232 Input Bits / Cassette Temp
BITCI	00A8	168	RS-232 Input Bit Count / Cassette Temp
RINONE	00A9	169	RS-232 Flag: Check for Start Bit
RIDATA	00AA	170	RS-232 Input Byte Buffer/Cassette Temp
RIPRTY	00AB	171	RS-232 Input Parity / Cassette Short Cnt
SAL	00AC–00AD	172–173	Pointer: Tape Buffer/ Screen Scrolling
EAL	00AE–00AF	174–175	Tape End Addresses/End of Program
CMP0	00B0–00B1	176–177	Tape Timing Constants
TAPE1	00B2–00B3	178–179	Pointer: Start of Tape Buffer
BITTS	00B4	180	RS-232 Out Bit Count / Cassette Temp
NXTBIT	00B5	181	RS-232 Next Bit to Send/ Tape EOT Flag
RODATA	00B6	182	RS-232 Out Byte Buffer
FNLEN	00B7	183	Length of Current File Name
LA	00B8	184	Current Logical File Number

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
SA	00B9	185	Current Secondary Address
FA	00BA	186	Current Device Number
FNADR	00BB–00BC	187–188	Pointer: Current File Name
ROPRTY	00BD	189	RS-232 Out Parity / Cassette Temp
FSBLK	00BE	190	Cassette Read/Write Block Count
MYCH	00BF	191	Serial Word Buffer
CAS1	00C0	192	Tape Motor Interlock
STAL	00C1–00C2	193–194	I/O Start Address
MEMUSS	00C3–00C4	195–196	Tape Load Temps
LSTX	00C5	197	Current Key Pressed: CHR$(n) 0 = No Key
NDX	00C6	198	No. of Chars. in Keyboard Buffer (Queue)
RVS	00C7	199	Flag: Print Reverse Chars.—1=Yes, 0=No Used
INDX	00C8	200	Pointer: End of Logical Line for INPUT
LXSP	00C9–00CA	201–202	Cursor X-Y Pos. at Start of INPUT
SFDX	00CB	203	Flag: Print Shifted Chars.
BLNSW	00CC	204	Cursor Blink enable: 0 = Flash Cursor
BLNCT	00CD	205	Timer: Countdown to Toggle Cursor
GDBLN	00CE	206	Character Under Cursor
BLNON	00CF	207	Flag: Last Cursor Blink On/Off
CRSW	00D0	208	Flag: INPUT or GET from Keyboard
PNT	00D1–00D2	209–210	Pointer: Current Screen Line Address

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
PNTR	00D3	211	Cursor Column on Current Line
QTSW	00D4	212	Flag: Editor in Quote Mode, $00 = NO
LNMX	00D5	213	Physical Screen Line Length
TBLX	00D6	214	Current Cursor Physical Line Number
	00D7	215	Temp Data Area
INSRT	00D8	216	Flag: Insert Mode, >0 = # INSTs
LDTB1	00D9–00F2	217–242	Screen Line Link Table / Editor Temps
USER	00F3–00F4	243–244	Pointer: Current Screen Color RAM loc.
KEYTAB	00F5–00F6	245–246	Vector: Keyboard Decode Table
RIBUF	00F7–00F8	247–248	RS-232 Input Buffer Pointer
ROBUF	00F9–00FA	249–250	RS-232 Output Buffer Pointer
FREKZP	00FB–00FE	251–254	Free 0-Page Space for User Programs
BASZPT	00FF	255	BASIC Temp Data Area
	0100–01FF	256–511	Micro-Processor System Stack Area
	0100–010A	256–266	Floating to String Work Area
BAD	0100–013E	256–318	Tape Input Error Log
BUF	0200–0258	512–600	System INPUT Buffer
LAT	0259–0262	601–610	KERNAL Table: Active Logical File No's.
FAT	0263–026C	611–620	KERNAL Table: Device No. for Each File
SAT	026D–0276	621–630	KERNAL Table: Second Address Each File
KEYD	0277–0280	631–640	Keyboard Buffer Queue (FIFO)

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
MEMSTR	0281–0282	641–642	Pointer: Bottom of Memory for O.S.
MEMSIZ	0283–0284	643–644	Pointer: Top of Memory for O.S.
TIMOUT	0285	645	Flag: Kernal Variable for IEEE Timeout
COLOR	0286	646	Current Character Color Code
GDCOL	0287	647	Background Color Under Cursor
HIBASE	0288	648	Top of Screen Memory (Page)
XMAX	0289	649	Size of Keyboard Buffer
RPTFLG	028A	650	Flag: REPEAT Key Used, S80 = Repeat
KOUNT	028B	651	Repeat Speed Counter
DELAY	028C	652	Repeat Delay Counter
SHFLAG	028D	653	Flag: Keyb'rd SHIFT Key/ CTRL Key/C= Key
LSTSHF	028E	654	Last Keyboard Shift Pattern
KEYLOG	028F–0290	655–656	Vector: Keyboard Table Setup
MODE	0291	657	Flag: S00=Disable SHIFT Keys, $80 = Enable SHIFT Keys
AUTODN	0292	658	Flag: Auto Scroll Down, 0 = ON
M51CTR	0293	659	RS-232: 6551 Control Register Image
M51CDR	0294	660	RS-232: 6551 Command Register Image
M51AJB	0295–0296	661–662	RS-232 Non-Standard BPS (Time/2-100) USA
RSSTAT	0297	663	RS-232: 6551 Status Register Image
BITNUM	0298	664	RS-232 Number of Bits Left to Send

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
BAUDOF	0299–029A	665–666	RS-232 Baud Rate: Full Bit Time (μs)
RIDBE	029B	667	RS-232 Index to End of Input Buffer
RIDBS	029C	668	RS-232 Start of Input Buffer (Page)
RODBS	029D	669	RS-232 Start of Output Buffer (Page)
RODBE	029E	670	RS-232 Index to End of Output Buffer
IRQTMP	029F–02A0	671–672	Holds IRQ Vector During Tape I/O
ENABL	02A1	673	RS-232 Enables
	02A2	674	TOD Sense During Cassette I/O
	02A3	675	Temp Storage For Cassette Read
	02A4	676	Temp D1IRQ Indicator For Cassette Read
	02A5	677	Temp For Line Index
	02A6	678	PAL/NTSC Flag, 0= NTSC, 1= PAL
	02A7–02FF	679–767	Unused
IERROR	0300–0301	768–769	Vector: Print BASIC Error Message
IMAIN	0302–0303	770–771	Vector: BASIC Warm Start
ICRNCH	0304–0305	772–773	Vector: Tokenize BASIC Text
IQPLOP	0306–0307	774–775	Vector: BASIC Text LIST
IGONE	0308–0309	776–777	Vector: BASIC Char. Dispatch
IEVAL	030A–030B	778–779	Vector: BASIC Token Evaluation
SAREG	030C	780	Storage for 6502 .A Register
SXREG	030D	781	Storage for 6502 .X Register

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
SYREG	030E	782	Storage for 6502 .Y Register
SPREG	030F	783	Storage for 6502 .SP Register
USRPOK	0310	784	USR Function Jump Instr (4C)
USRADD	0311–0312	785–786	USR Address Low Byte/ High Byte
	0313	787	Unused
CINV	0314–0315	788–789	Vector: Hardware IRQ Interrupt
CBINV	0316–0317	790–791	Vector: BRK Instr. Interrupt
NMINV	0318–0319	792–793	Vector: Non-Maskable Interrupt
IOPEN	031A–031B	794–795	KERNAL OPEN Routine Vector
ICLOSE	031C–031D	796–797	KERNAL CLOSE Routine Vector
ICHKIN	031E–031F	798–799	KERNAL CHKIN Routine Vector
ICKOUT	0320–0321	800–801	KERNAL CHKOUT Routine Vector
ICLRCH	0322–0323	802–803	KERNAL CLRCHN Routine Vector
IBASIN	0324–0325	804–805	KERNAL CHRIN Routine Vector
IBSOUT	0326–0327	806–807	KERNAL CHROUT Routine Vector
ISTOP	0328–0329	808–809	KERNAL STOP Routine Vector
IGETIN	032A–032B	810–811	KERNAL GETIN Routine Vector
ICLALL	032C–032D	812–813	KERNAL CLALL Routine Vector
USRCMD	032E–032F	814–815	User-Defined Vector
ILOAD	0330–0331	816–817	KERNAL LOAD Routine Vector

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
ISAVE	0332–0333	818–819	KERNAL SAVE Routine Vector
	0334–033B	820–827	Unused
TBUFFR	033C–03FB	828–1019	Tape I/O Buffer
	03FC–03FF	1020–1023	Unused
VICSCN	0400–07FF	1024–2047	1024 Byte Screen Memory Area
	0400–07E7	1024–2023	Video Matrix: 25 Lines × 40 Columns
	07F8–07FF	2040–2047	Sprite Data Pointers
	0800–9FFF	2048–40959	Normal BASIC Program Space
	8000–9FFF	32768–40959	VSP Cartridge ROM—8192 Bytes
	A000–BFFF	40960–49151	BASIC ROM—8192 Bytes (or 8K RAM)
	C000–CFFF	49152–53247	RAM—4096 Bytes
	D000–DFFF	53248–57343	Input/Output Devices and Color RAM or Character Generator ROM or RAM—4096 Bytes
	E000–FFFF	57344–65535	KERNAL ROM—8192 Bytes (or 8K RAM)

COMMODORE 64 INPUT/OUTPUT ASSIGNMENTS

HEX	DECIMAL	BITS	DESCRIPTION
0000	0	7–0	MOS 6510 Data Direction Register (xx101111) Bit=1: Output, Bit=0: Input, x=Don't Care
0001	1		MOS 6510 Micro-Processor On-Chip I/O Port
		0	/LORAM Signal (0=Switch BASIC ROM Out)

HEX	DECIMAL	BITS	DESCRIPTION
		1	/HIRAM Signal (0=Switch Kernal ROM Out)
		2	/CHAREN Signal (0=Switch Char. ROM In)
		3	Cassette Data Output Line
		4	Cassette Switch Sense 1 = Switch Closed
		5	Cassette Motor Control 0 = ON, 1 = OFF
		6–7	Undefined
D000–D02E	53248–54271		MOS 6566 VIDEO INTERFACE CONTROLLER (VIC)
D000	53248		Sprite 0 X Pos
D001	53249		Sprite 0 Y Pos
D002	53250		Sprite 1 X Pos
D003	53251		Sprite 1 Y Pos
D004	53252		Sprite 2 X Pos
D005	53253		Sprite 2 Y Pos
D006	53254		Sprite 3 X Pos
D007	53255		Sprite 3 Y Pos
D008	53256		Sprite 4 X Pos
D009	53257		Sprite 4 Y Pos
D00A	53258		Sprite 5 X Pos
D00B	53259		Sprite 5 Y Pos
D00C	53260		Sprite 6 X Pos
D00D	53261		Sprite 6 Y Pos
D00E	53262		Sprite 7 X Pos
D00F	53263		Sprite 7 Y Pos
D010	53264		Sprites 0–7 X Pos (msb of X coord.)
D011	53265		VIC Control Register
		7	Raster Compare: (Bit 8) See 53266
		6	Extended Color Text Mode: 1 = Enable

HEX	DECIMAL	BITS	DESCRIPTION
		5	Bit-Map Mode: 1 = Enable
		4	Blank Screen to Border Color: 0 = Blank
		3	Select 24/25 Row Text Display: 1 = 25 Rows
		2–0	Smooth Scroll to Y Dot-Position (0-7)
D012	53266		Read Raster / Write Raster Value for Compare IRQ
D013	53267		Light-Pen Latch X Pos
D014	53268		Light-Pen Latch Y Pos
D015	53269		Sprite Display Enable: 1 = Enable
D016	53270		VIC Control Register
		7–6	Unused
		5	ALWAYS SET THIS BIT TO 0!
		4	Multi-Color Mode: 1 = Enable (Text or Bit-Map)
		3	Select 38/40 Column Text Display: 1 = 40 Cols
		2–0	Smooth Scroll to X Pos
D017	53271		Sprites 0–7 Expand 2× Vertical (Y)
D018	53272		VIC Memory Control Register
		7–4	Video Matrix Base Address (inside VIC)
		3–1	Character Dot-Data Base Address (inside VIC)
D019	53273		VIC Interrupt Flag Register (Bit = 1: IRQ Occurred)
		7	Set on Any Enabled VIC IRQ Condition
		3	Light-Pen Triggered IRQ Flag

HEX	DECIMAL	BITS	DESCRIPTION
		2	Sprite to Sprite Collision IRQ Flag
		1	Sprite to Background Collision IRQ Flag
		0	Raster Compare IRQ Flag
D01A	53274		IRQ Mask Register: 1 = Interrupt Enabled
D01B	53275		Sprite to Background Display Priority: 1 = Sprite
D01C	53276		Sprites 0–7 Multi-Color Mode Select: 1 = M.C.M.
D01D	53277		Sprites 0–7 Expand 2× Horizontal (X)
D01E	53278		Sprite to Sprite Collision Detect
D01F	53279		Sprite to Background Collision Detect
D020	53280		Border Color
D021	53281		Background Color 0
D022	53282		Background Color 1
D023	53283		Background Color 2
D024	53284		Background Color 3
D025	53285		Sprite Multi-Color Register 0
D026	53286		Sprite Multi-Color Register 1
D027	53287		Sprite 0 Color
D028	53288		Sprite 1 Color
D029	53289		Sprite 2 Color
D02A	53290		Sprite 3 Color
D02B	53291		Sprite 4 Color
D02C	53292		Sprite 5 Color
D02D	53293		Sprite 6 Color
D02E	53294		Sprite 7 Color
D400–D7FF	54272–55295		MOS 6581 SOUND INTERFACE DEVICE (SID)

HEX	DECIMAL	BITS	DESCRIPTION
D400	54272		Voice 1: Frequency Control—Low-Byte
D401	54273		Voice 1: Frequency Control—High-Byte
D402	54274		Voice 1: Pulse Waveform Width—Low-Byte
D403	54275	7–4	Unused
		3–0	Voice 1: Pulse Waveform Width—High-Nybble
D404	54276		Voice 1: Control Register
		7	Select Random Noise Waveform, 1 = On
		6	Select Pulse Waveform, 1 = On
		5	Select Sawtooth Waveform, 1 = On
		4	Select Triangle Waveform, 1 = On
		3	Test Bit: 1 = Disable Oscillator 1
		2	Ring Modulate Osc. 1 with Osc. 3 Output, 1 = On
		1	Synchronize Osc. 1 with Osc. 3 Frequency, 1 = On
		0	Gate Bit: 1 = Start Att/Dec/Sus, 0 = Start Release
D405	54277		Envelope Generator 1: Attack / Decay Cycle Control
		7–4	Select Attack Cycle Duration: 0–15
		3–0	Select Decay Cycle Duration: 0–15
D406	54278		Envelope Generator 1: Sustain / Release Cycle Control

HEX	DECIMAL	BITS	DESCRIPTION
		7–4	Select Sustain Cycle Duration: 0–15
		3–0	Select Release Cycle Duration: 0–15
D407	54279		Voice 2: Frequency Control—Low-Byte
D408	54280		Voice 2: Frequency Control—High-Byte
D409	54281		Voice 2: Pulse Waveform Width—Low-Byte
D40A	54282	7–4	Unused
		3–0	Voice 2: Pulse Waveform Width—High-Nybble
D40B	54283		Voice 2: Control Register
		7	Select Random Noise Waveform, 1 = On
		6	Select Pulse Waveform, 1 = On
		5	Select Sawtooth Waveform, 1 = On
		4	Select Triangle Waveform, 1 = On
		3	Test Bit: 1 = Disable Oscillator 2
		2	Ring Modulate Osc. 2 with Osc. 1 Output, 1 = On
		1	Synchronize Osc. 2 with Osc. 1 Frequency, 1 = On
		0	Gate Bit: 1 = Start Att/Dec/Sus, 0 = Start Release
D40C	54284		Envelope Generator 2: Attack / Decay Cycle Control
		7–4	Select Attack Cycle Duration: 0–15

HEX	DECIMAL	BITS	DESCRIPTION
		3–0	Select Decay Cycle Duration: 0–15
D40D	54285		Envelope Generator 2: Sustain / Release Cycle Control
		7–4	Select Sustain Cycle Duration: 0–15
		3–0	Select Release Cycle Duration: 0–15
D40E	54286		Voice 3: Frequency Control—Low-Byte
D40F	54287		Voice 3: Frequency Control—High-Byte
D410	54288		Voice 3: Pulse Waveform Width—Low-Byte
D411	54289	7–4	Unused
		3–0	Voice 3: Pulse Waveform Width—High-Nybble
D412	54290		Voice 3: Control Register
		7	Select Random Noise Waveform, 1 = On
		6	Select Pulse Waveform, 1 = On
		5	Select Sawtooth Waveform, 1 = On
		4	Select Triangle Waveform, 1 = On
		3	Test Bit: 1 = Disable Oscillator 3
		2	Ring Modulate Osc. 3 with Osc. 2 Output, 1 = On
		1	Synchronize Osc. 3 with Osc. 2 Frequency, 1 = On
		0	Gate Bit: 1 = Start Att/Dec/Sus, 0 = Start Release

HEX	DECIMAL	BITS	DESCRIPTION
D413	54291		Envelope Generator 3: Attack / Decay Cycle Control
		7–4	Select Attack Cycle Duration: 0–15
		3–0	Select Decay Cycle Duration: 0–15
D414	54292		Envelope Generator 3: Sustain / Release Cycle Control
		7–4	Select Sustain Cycle Duration: 0–15
		3–0	Select Release Cycle Duration: 0–15
D415	54293		Filter Cutoff Frequency: Low-Nybble (Bits 2–0)
D416	54294		Filter Cutoff Frequency: High-Byte
D417	54295		Filter Resonance Control / Voice Input Control
		7–4	Select Filter Resonance: 0–15
		3	Filter External Input: 1 = Yes, 0 = No
		2	Filter Voice 3 Output: 1 = Yes, 0 = No
		1	Filter Voice 2 Output: 1 = Yes, 0 = No
		0	Filter Voice 1 Output: 1 = Yes, 0 = No
D418	54296		Select Filter Mode and Volume
		7	Cut-Off Voice 3 Output: 1 − Off, 0 − On
		6	Select Filter High-Pass Mode: 1 = On
		5	Select Filter Band-Pass Mode: 1 = On

HEX	DECIMAL	BITS	DESCRIPTION
		4	Select Filter Low-Pass Mode: 1 = On
		3–0	Select Output Volume: 0–15
D419	54297		Analog/Digital Converter: Game Paddle 1 (0–255)
D41A	54298		Analog/Digital Converter: Game Paddle 2 (0–255)
D41B	54299		Oscillator 3 Random Number Generator
D41C	54230		Envelope Generator 3 Output
D500–D7FF	54528–55295		SID IMAGES
D800–DBFF	55296–56319		Color RAM (Nybbles)
DC00–DCFF	56320–56575		MOS 6526 Complex Interface Adapter (CIA) #1
DC00	56320		Data Port A (Keyboard, Joystick, Paddles, Light-Pen)
		7–0	Write Keyboard Column Values for Keyboard Scan
		7–6	Read Paddles on Port A / B (01 = Port A, 10 = Port B)
		4	Joystick A Fire Button: 1 = Fire
		3–2	Paddle Fire Buttons
		3–0	Joystick A Direction (0–15)
DC01	56321		Data Port B (Keyboard, Joystick, Paddles): Game Port 1

HEX	DECIMAL	BITS	DESCRIPTION
		7–0	Read Keyboard Row Values for Keyboard Scan
		7	Timer B: Toggle/Pulse Output
		6	Timer A: Toggle/Pulse Output
		4	Joystick 1 Fire Button: 1 = Fire
		3–2	Paddle Fire Buttons
		3–0	Joystick 1 Direction
DC02	56322		Data Direction Register—Port A (56320)
DC03	56323		Data Direction Register—Port B (56321)
DC04	56324		Timer A: Low-Byte
DC05	56325		Timer A: High-Byte
DC06	56326		Timer B: Low-Byte
DC07	56327		Timer B: High-Byte
DC08	56328		Time-of-Day Clock: 1/10 Seconds
DC09	56329		Time-of-Day Clock: Seconds
DC0A	56330		Time-of-Day Clock: Minutes
DC0B	56331		Time-of-Day Clock: Hours + AM/PM Flag (Bit 7)
DC0C	56332		Synchronous Serial I/O Data Buffer
DC0D	56333		CIA Interrupt Control Register (Read IRQs/ Write Mask)
		7	IRQ Flag (1 = IRQ Occurred) / Set-Clear Flag
		4	FLAG1 IRQ (Cassette Read / Serial Bus SRQ Input)

HEX	DECIMAL	BITS	DESCRIPTION
		3	Serial Port Interrupt
		2	Time-of-Day Clock Alarm Interrupt
		1	Timer B Interrupt
		0	Timer A Interrupt
DC0E	56334		CIA Control Register A
		7	Time-of-Day Clock Frequency: 1 = 50 Hz, 0 = 60 Hz
		6	Serial Port I/O Mode: 1 = Output, 0 = Input
		5	Timer A Counts: 1 = CNT Signals, 0 = System 02 Clock
		4	Force Load Timer A: 1 = Yes
		3	Timer A Run Mode: 1 = One-Shot, 0 = Continuous
		2	Timer A Output Mode to PB6: 1 = Toggle, 0 = Pulse
		1	Timer A Output on PB6: 1 = Yes, 0 = No
		0	Start/Stop Timer A: 1 = Start, 0 = Stop
DC0F	56335		CIA Control Register B
		7	Set Alarm/TOD-Clock: 1 = Alarm, 0 = Clock

HEX	DECIMAL	BITS	DESCRIPTION
		6–5	Timer B Mode Select: 00 = Count System 02 Clock Pulses 01 = Count Positive CNT Transitions 10 = Count Timer A Underflow Pulses 11 = Count Timer A Underflows While CNT Positive
		4–0	Same as CIA Control Reg. A–for Timer B
DD00–DDFF	56576–56831		MOS 6526 Complex Interface Adapter (CIA) #2
DD00	56576		Data Port A (Serial Bus, RS-232, VIC Memory Control)
		7	Serial Bus Data Input
		6	Serial Bus Clock Pulse Input
		5	Serial Bus Data Output
		4	Serial Bus Clock Pulse Output
		3	Serial Bus ATN Signal Output
		2	RS-232 Data Output (User Port)
		1–0	VIC Chip System Memory Bank Select (Default = 11)
DD01	56577		Data Port B (User Port, RS-232)
		7	User / RS-232 Data Set Ready

HEX	DECIMAL	BITS	DESCRIPTION
		6	User / RS-232 Clear to Send
		5	User
		4	User / RS-232 Carrier Detect
		3	User / RS-232 Ring Indicator
		2	User / RS-232 Data Terminal Ready
		1	User / RS-232 Request to Send
		0	User / RS-232 Received Data
DD02	56578		Data Direction Register—Port A
DD03	56579		Data Direction Register—Port B
DD04	56580		Timer A: Low-Byte
DD05	56581		Timer A: High-Byte
DD06	56582		Timer B: Low-Byte
DD07	56583		Timer B: High-Byte
DD08	56584		Time-of-Day Clock: 1/10 Seconds
DD09	56585		Time-of-Day Clock: Seconds
DD0A	56586		Time-of-Day Clock: Minutes
DD0B	56587		Time-of-Day Clock: Hours + AM/PM Flag (Bit 7)
DD0C	56588		Synchronous Serial I/O Data Buffer
DD0D	56589		CIA Interrupt Control Register (Read NMIs/ Write Mask)

HEX	DECIMAL	BITS	DESCRIPTION
		7	NMI Flag (1 = NMI Occurred) / Set-Clear Flag
		4	FLAG1 NMI (User/RS-232 Received Data Input)
		3	Serial Port Interrupt
		1	Timer B Interrupt
		0	Timer A Interrupt
DD0E	56590		CIA Control Register A
		7	Time-of-Day Clock Frequency: 1 = 50 Hz, 0 = 60 Hz
		6	Serial Port I/O Mode: 1 = Output, 0 = Input
		5	Timer A Counts: 1 = CNT Signals, 0 = System 02 Clock
		4	Force Load Timer A: 1 = Yes
		3	Timer A Run Mode: 1 = One-Shot, 0 = Continuous
		2	Timer A Output Mode to PB6: 1 = Toggle, 0 = Pulse
		1	Timer A Output on PB6: 1 = Yes, 0 = No
		0	Start/Stop Timer A: 1 = Start, 0 = Stop
DD0F	56591		CIA Control Register B
		7	Set Alarm/TOD-Clock: 1 = Alarm, 0 = Clock

HEX	DECIMAL	BITS	DESCRIPTION
		6–5	Timer B Mode Select: 00 = Count System 02 Clock Pulses 01 = Count Positive CNT Transitions 10 = Count Timer A Underflow Pulses 11 = Count Timer A Underflows While CNT Positive
		4–0	Same as CIA Control Reg. A for Timer B
DE00–DEFF	56832–57087		Reserved for Future I/O Expansion
DF00–DFFF	57088–57343		Reserved for Future I/O Expansion