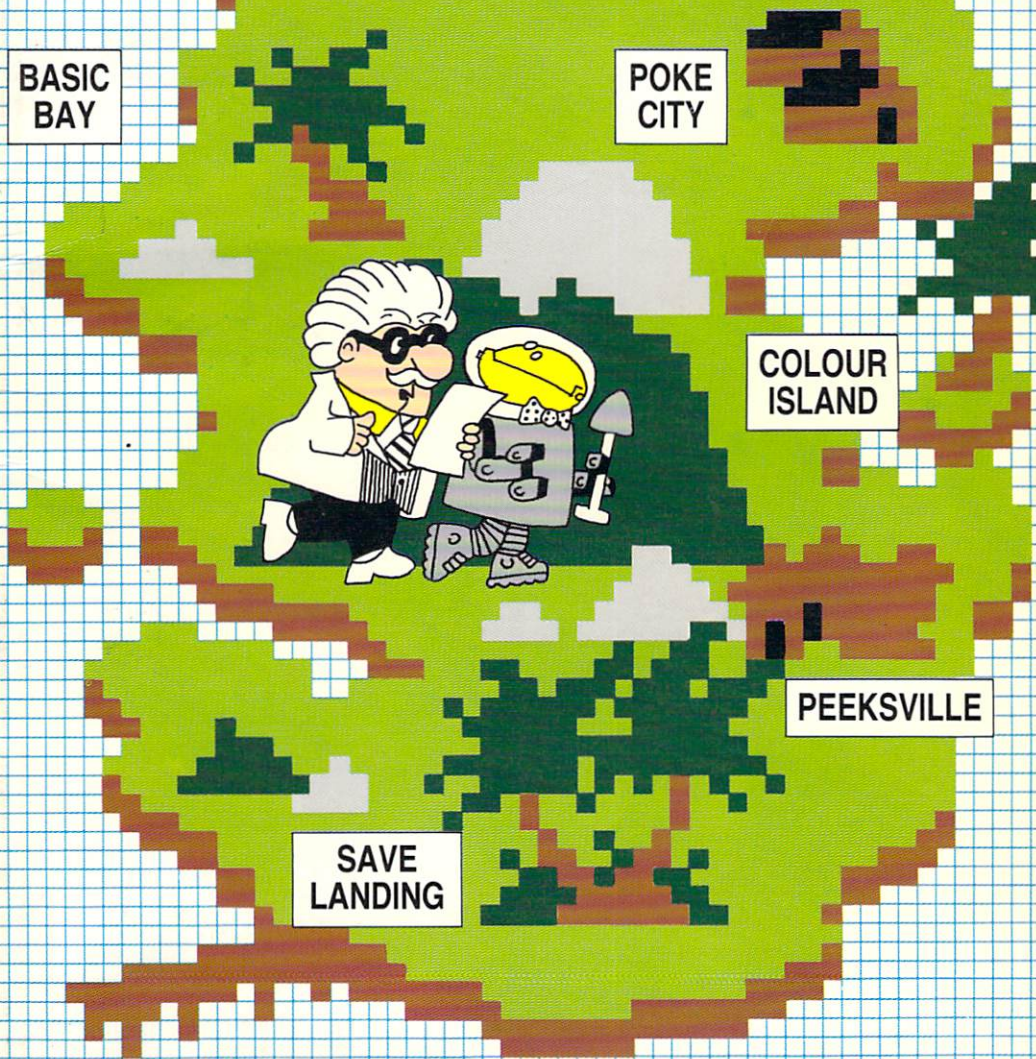
THE MASTER MEMORY MAP FOR THE COMMODORE 64



Paul Pavelko and Tim Kelly

MASTER MEMORY MAP FOR THE COMMODORE 64

A GUIDE TO THE INNER WORKING OF THE COMMODORE 64's BRAIN CELLS

by
PAUL PAVELKO
and
TIM KELLY

Englewood Cliffs, NJ London New Delhi Rio de Janerio
Singapore Sydney Tokyo Toronto Wellington

ISBN 0-13-574351-6

Commodore 64 is a trademark of Commodore Business Machines.
Professor von Chip and Prototype are trademarks of
Educational Software, inc.

10 9 8 7 6 5 4 3 2 1

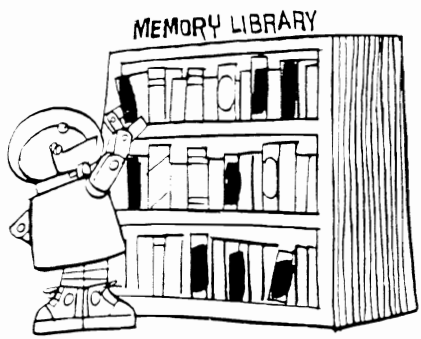
Printed in Great Britain by A. Wheaton & Co. Ltd., Exeter

TABLE OF CONTENTS

PRELUDE
TO THE COMMODORE 64 MASTER MEMORY MAP

Welcome all, beginning or expert programmer, to ESI's COMMODORE 64 MASTER MEMORY MAP! This book will be your guide into the inner working of the Commodore 64's 'brain cells'. This is truly a map, a guide to the special places inside the operating system of the computer. These places will help you add new features to the programs you write, making them really come alive!

Along the way, you have the humor of Professor Von Chip and the friendly alien Prototype to help make the journey a productive one.



The Master Memory Map is divided into sections to aid you. Each section deals with a particular part of the memory. There are lots of programming examples, too, because sometimes it's easier to understand an example when you see it on the screen instead of just reading it. Some of the programming examples add a useful utility to BASIC, like the RENUMBER routine. Others serve as useful programming 'tricks'. In every case, you should study the listings and play with the code to see what happens.

The appendices go into more detail, showing how to do something like create sprites or produce a sound. They give longer programming examples and show you some of the advanced things you can do with the Commodore 64.

This book really isn't a novel, so you can start reading anywhere. But sometime you should read it from cover to cover; sooner or later you'll see new ways to use the computer. This moment of enlightenment - a creative flash - is what makes working with a computer so much fun!

We've worked hard to make the Master Memory Map easy to read and use. Look in the upper right hand page corner for a guide that will show you which locations are covered on those pages. Just flip through the book until you come to the locations you need. The appendix sections are also marked in a similar way.

Prototype, sometimes just called Proto, will help you find locations and routines that the beginning or intermediate programmer will use most often. Look for him in the margin as you flip through the book.

A COPY OF THE PROGRAMS

The programs in this book are used as illustrations for techniques and ideas. You will gain a lot of knowledge if you type in the programs yourself. But if you don't want to tire your fingers, send $9.95 to:

Educational Software, inc.
4565 Cherryvale Ave.
Soquel, CA 95073

A BONUS!

If you discover a new, unpublished use for one of the memory locations send it to Educational Software. In return, we'll send you some software, free.

SOURCES

A few of the program examples in the Master Memory Map come from other sources. The source is identified in the text by using these symbols.

C=: COMMODORE 64 PROGRAMMER'S REFERENCE GUIDE, Commodore Business Machines, Inc., Computer Systems Division, 487 Devon Park Drive, Wayne, PA 19087.

C!: COMPUTE! Magazines, Copyright 1982, Small System Service, Inc., Reprinted by permission from COMPUTE! Magazine, P.O. Box 5406, Greensboro NC 25403, 12 Issue, Subscription $20.00.

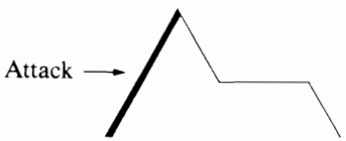
GLOSSARY

ASCII: The American Standard Code for Information Interchange. This is one standard for assigning numbers to the letters and characters on the keyboard. Commodore computers do not follow a true ASCII (pronounced 'ASK KEY') but have their own code instead.

Accumulator: The results of logic and arithmetic operations are stored here temporarily. It acts as a busy bus stop, nobody stays here long!

Address: The number of a given location. It's just like a street address.

Attack: The rate a note or sound changes from 'off' to its highest volume.



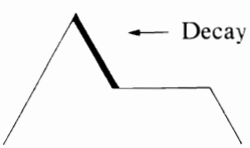
Baud: This is the rate of transmittion of information conveyed over a line . This rate is determined by the bits per second that are being transfered. You encounter this term if you are using a modem or some device that requires special interfacing (RS-232).

Bit: The smallest piece of information the computer can handle. There are eight bits in a byte.

Buffer: A storage place. For example, the keyboard buffer stores your keystrokes and allows you to type faster. The information in a buffer eventually goes somewhere else to be acted upon.

Bus: A bus is a system of electrical lines shared by all devices that are connected to it. This is a convenient way for these devices to share addressing and data. It works just like a party line telephone.

Decay: A musical term meaning the rate of change from the highest level to the sustaining level of sound.



Default: The beginning value of a memory location especially when the power is turned on or other operations are done.

Disable: Turn off. By disabling the RUN/STOP key, you can prevent anyone from accidentally stopping your program.

Enable: To turn on; the opposite of disable.

Flag: A signal that something has happened. Flags can be used in your own programs. For example:

```
If A$ = "ouch" then B = 1
```

B is the flag in that statement.

Floating Point: Arithmetic operations using decimal numbers.

Immediate Mode: Using the computer without running a program. For example:

```
10 PRINT 3 + 2
```

is a program and must be run to get an answer.

```
PRINT 3 + 2
```

will answer "5" when you press RETURN.

Jump: To go from one location to another. In BASIC, the equivalent terms are GOTO and GOSUB.

KERNAL: This is Commodore's word for a series of machine language subroutines that operate the computer. See the Appendix for more information.

Nybble: Pronounced 'nibble'. A nybble is half a byte. Really. The low nybble is composed of bits 0 to 3. The high nybble has bits 4 - 7.

Operating System: Sometimes this is referred to as the OS. Part of this is the KERNAL described above. Its job is to make the computer run.

Page: A page is 256 bytes of memory. The computer often keeps track of different blocks of memory in terms of pages since it is easier for the computer to store.

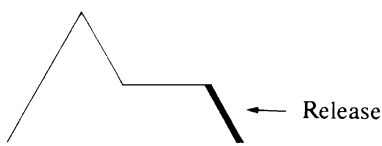
Pointer: It does just what it says, it acts as a signpost, telling the computer where to look for information.

RAM: Random Access Memory. This type of memory can be easily changed. Your programs are stored in RAM, and when the computer is turned off any information in RAM is lost.

ROM: Read Only Memory. This type of memory does not change when the power to the computer is turned off. Examples of ROM memory include BASIC and the KERNAL.

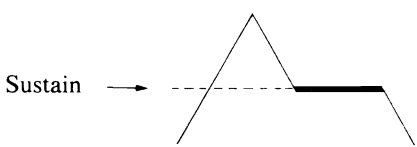
Register: Another name for memory location. Registers can be more than one byte long.

Release: A musical term describing the rate of fall from the sustain level to zero volume.



Reserved Word: Letters that can't be part of your program. Examples of reserved words are the status word, ST, and the time words, TI and TI$. To save yourself from trouble, don't use any variables in your programs that have the same starting letters of any BASIC command or have BASIC words in them.

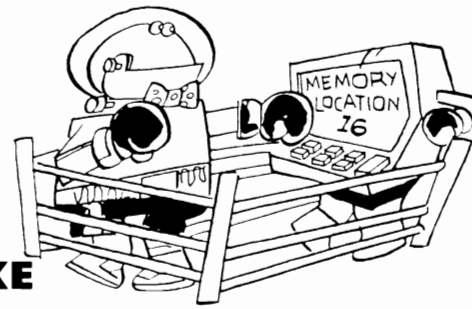
Sustain: Another musical term used with the sound capabilities of the computer. This extends the sound, like the pedals of a piano.



Waveform: This term is a description of the type of sound produced. The computer has four different waveforms; triangle, sawtooth, pulse and noise. Each type produces a different kind of sound.

Vector: This is another kind of pointer. It refers to the starting address of a routine. The computer needs to know where to look for things.

How to PEEK and POKE



This part is for those who have yet to learn how to use a memory map. Basically, a memory map is a list of valuable locations within the computer (in this case a COMMODORE 64), that you can directly use for various purposes. These locations are called bytes (memory locations) of memory at a specific place. With 64K of memory, there are 64*1024 memory locations that you can work with. Although some of these bytes are used for the computer's Operating System, most of them are blank for you to use in your programs. This manual will tell you about the ones that you can do something with.

For example, you can quickly look down this list to find that location 650 controls the repeating of certain keys, like the space bar. By following the included hints, you can change the "normal" value in that location, so that when you press *any* key it will repeat as long as you hold it down. Please note that any of the changes that you make are only temporary and will go away when the computer is turned off.

Now to explain how to make changes from BASIC. Say you look down the list and decide to change location 650 (all numbers are decimal unless marked with a $ symbol, which denotes a HEX-adecimal number, or in a column marked "hex"). 650 is called RPTFLG by Commodore. Its function is to decide which keys on the keyboard to repeat. So, if you would like all the keys to repeat as long as you hold the key down, you simply look up the correct value to POKE into location 650.

In this example, the memory map says to use the number 255 to repeat all keys. The BASIC instruction to put a number into memory is called "POKE". After all this long-winded explanation, you can now see how simple it is to make this change:

POKE 650,255

— HINT —

Always use the decimal numbers with a POKE statement. This means that sometimes you will have to convert between binary, hexadecimal and decimal. Also, any one memory location can only hold a number up to 255. Why?...remember that the COMMODORE uses eight bits per word (memory location), and eight bits in binary counts from 0 to 255 (internally, the machine uses binary). You may want to study the next section of the manual, "Bytes and Bits" to learn about binary. Because of this limitation, sometimes you must POKE numbers into two locations in a row.

For example, look at memory locations 643 & 644 which are called MEMSIZ. These locations hold a number that corresponds to the top of your available memory (called RAM - Random Access Memory). Since the top of memory can be up to 40960, a number well above the limit of 256 for any one memory location, the computer will need two locations to store the value. Yes, I know 256 for the first location plus 256 for the second doesn't seem to add up to a large enough number to hold 64K, but the computer takes each number in the second location and multiplies it by 256. Examples:

```
 11 stored in the low byte
+1 stored in the high byte
---
267
```

The computer "sees" 256*1+11 which equals 267.

Another example:

```
 121 in the low byte
  +7 in the high byte
----
1913
```

Since 7*256 +121=1913.

Sometimes it is desirable to fool the COMMODORE 64 into thinking that the top of memory is lower than it actually is, perhaps to keep it from using the last thousand bytes of memory, thus reserving them for Sprites. You do the same type of POKE here as in the first example, except that you have to do it twice; once for the "LOW" part of the number and once for the "HIGH" part.

I said the LOW part of the number is placed in the first memory location and the HIGH part is next. Although it seems backwards, this is really not hard to understand. The COMMODORE (and most other micro-computers) store multiple part numbers this way. Occasionally this rule is broken, so please don't call me up if you find an exception.

Here's what you do. We want to change the value of MEMSIZ to be 1K less than it currently is.

1) Find current value...

```
10 A = PEEK(643)+PEEK(644) * 256
       LOW Part      HIGH Part
```

This number will be 40960, which is the value of MEMSIZ, when the 64 is turned on.

2) Subtract 1K from this value...

```
20 A = A-(1 * 1024)
```

Remember that one K is actually 1024 bytes.
(You don't have to use 1, you can can change the size by any number.)

3) Break the new value up into LOW and HIGH parts...

```
30 B = INT(A/256): C = A-(B * 256)
```

B would = 156, C would = 0

What this does is make C the LOW part of the number and B the HIGH part.

EX: 40960-1K = 39936

Line 30, when run, will give you 156 for the HIGH part and 0 for the LOW part.

4) POKE these values into memory...

```
40 POKE 643,C: POKE 644,B (#'s in decimal!!)
```

EX: POKE 643,0:POKE 644,156

!! FINAL WORDS OF WISDOM !!

1) Feel free to POKE and PEEK all you want, trying out ideas or testing the effects mentioned in the Master Memory Map. The explanations are only the most basic part of how to do the various effects possible on the COMMODORE. Watch for EDUCATIONAL SOFTWARE'S series of TRICKY TUTORIALS™ that will take you step by step through Sprites, Page Flipping, Sound, Animation and other uses of the computer. These are the techniques that the best programs use, and all of our Tutorials are done in BASIC, although we do sometimes include a machine language subroutine to offer you some advantage like speed.

2) Remember that two numbers are required to tell the computer the value for some locations, and these are stored LOW part first, HIGH part second. This is opposite of what you might think.

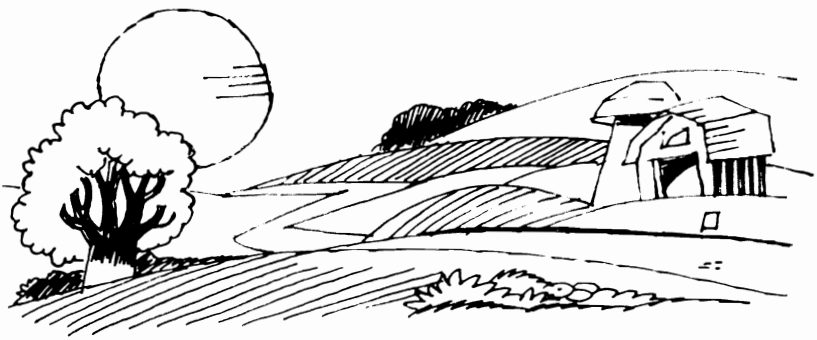
3) All of the memory locations are here, but many are for advanced users only. Don't feel bad if you have no idea what they are for. The idea is to experiment and learn.

4) You can usually press Run/Stop and Restore if trouble occurs. This will restore the original (default) values of many locations.

5) Some locations in the Master Memory Map are used to read from only; that is, you can PEEK to see what is there, but you can't POKE your own number in. This is because part of memory is a type called "ROM" which means "READ ONLY MEMORY". This type is permanent and can't be changed by POKEing, but Commodore has thoughtfully provided a way for you to put a copy of the ROM into memory so you can change it if you wish.

Go back and re-read the last section at least a few hundred times. There are only four lines in the program that both read the old value of MEMSIZ and store a new value. These lines don't have to be part of a program. You could enter them directly.

BYTES and BITS



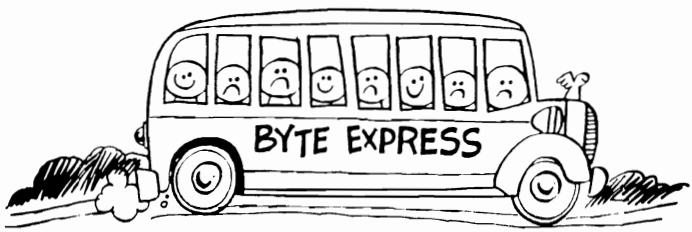
To PEEK & POKE you need to understand what a byte is and how it is structured. It isn't too hard to understand - and you can use the Master Memory Map without learning too much about them - but the more you know about Bytes and Bits the more you'll learn about controlling your Commodore.

A BYTE is really not complicated at all. A BYTE is simply a group of eight BITS. When eight BITS are structured into a BYTE then each of those BITS have special significance. You look puzzled! What, you say, is a BIT?

A BIT is the smallest piece of information a computer can deal with. In fact, the computer only deals with BITs at the most fundamental level. It may be helpful to imagine the microprocessor as a bus station. This bus station has only one single lane road attached to it. That means a bus can only travel in one direction at a time as there is not enough room for two busses to pass each other. Therefore, a bus may either be arriving at the station or departing. The microprocessor, or bus station, can schedule its bus with a signal light that says "I AM ACCEPTING ARRIVALS" or "I AM SENDING DEPARTURES".

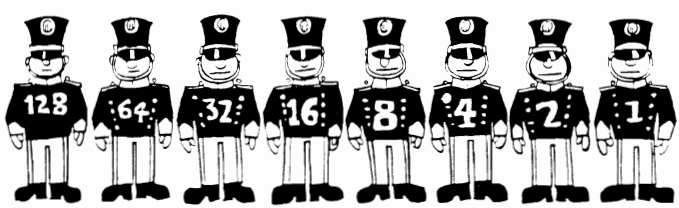
In fact, in real computer hardware architecture, the wires that carry information to and from a microprocessor are called the DATA BUSSES. We don't need eight separate INPUT and eight separate OUTPUT wires because, like the single lane road connected to the bus station, the wires are bi-directional, that is, information can either be arriving (INPUT) or departing (OUTPUT). The microprocessor also has a signal of its own that determines whether it will receive (INPUT) or send (OUTPUT) information.

WHO'S RIDING THE BUS?



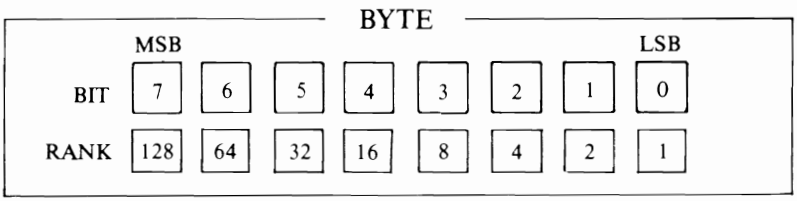
Let's take a closer look at that bus. It is known as the BYTE express, has eight seats, and always carries eight passengers. Those passengers are little messengers known as BITs, and, as a group, they are known as a BYTE. These messengers, or BITs, are rather moody. They are either turned "ON" or they are turned "OFF". That is called BINARY as they are BI-STATE signals, ON being a "1" state or OFF being a "0" state. Their vocabulary is just as limited...the only thing they are willing to tell you is their mood. Now how do we get any meaningful information out of a group of eight little monsters standing in front of us, each screaming "ON" or "OFF" at one time?

Well, when the bus arrives, we could have the whole BYTE stand in front of us and count everyone who is turned "ON". That would give us the capability of counting to eight. Seems pretty limited, doesn't it? Hmmm, the group really needs a leader. That leader will be the first BIT on the left. We'll call that BIT the Most Significant Bit, or the MSB. The last BIT on the right will be the Least Significant Bit, or LSB. Terrific! Now that we have a group leader and group follower, then all the BITs should have a rank.



Handing out ranks is a serious matter and much thought should be given to it. We can start with the LSB and assign that BIT the rank of "1", since it is the Least Significant Bit. We can be easy on everyone if

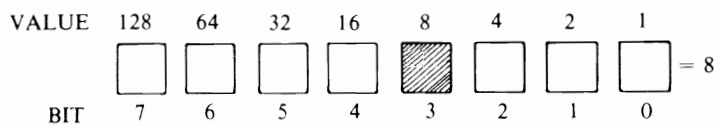
we just double that rank for the next BIT in line. So, why not just keep doubling the rank for the next BIT in line and so on until we get to the MSB or Most Significant Bit. Now out BYTE looks something like this:



MESSAGES WITH MEANING!

What have we gained? More than meets the eye! When the BYTE gets off the BYTE express and each BIT starts telling us what their mood for the day is, we can make a different and more meaningful interpretation out of the ignorant little beasts. If everyone is turned "OFF" except the fourth BIT from the right we can check the rank of that BIT and find it is eight (8). Unknown to the BITs, they have brought us a message and the message is "8".

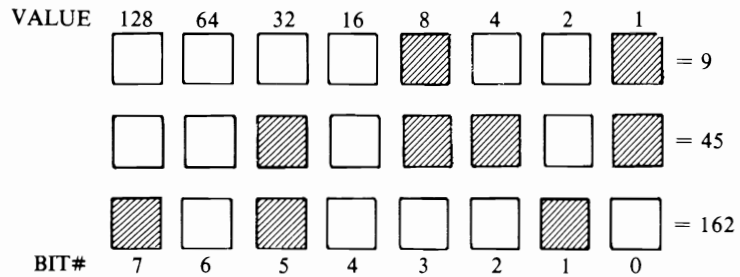
Computers are very efficient and do not like to waste information, therefore, computer related numbers usually start at BASE ZERO (0) since zero is unique. We usually like to start counting with one (1) for the convenience of our thinking process. That way, the number we arrive at when we have counted the last item actually represents the number of items we counted. Normally, we would count the BITs as one (1) through eight (8). The computer thinks a little more efficiently than mortals and sees BITs zero through seven as representing eight (8) individual BITs.



If you SET or turn "ON" only the fourth BIT (BIT #3) from the right you will observe a value of eight (8) in the value box. That was the message we received!

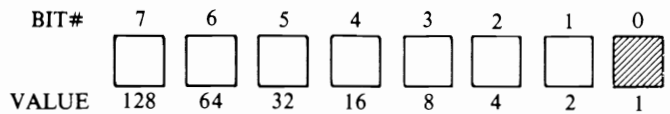
Aha, we now want the BITs to get on the bus and carry a message back to the sender. We want them to tell the sender "9". Oops, a small

problem, there is no BIT with the rank of nine in the BYTE. What to do, what to do? I guess the next best thing is to be very nice to the LSB, pat it on the head, and turn it "ON", then when the sender receives our message from the return BYTE he will find BITs with the rank of "1" and "8" turned "ON". Adding those ranks together gives us a numerical range of 0 to 255. That is a total of 256 cases if you count 0 as a case.



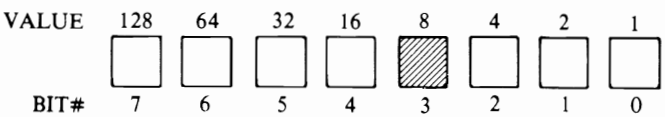
Well, that's how you get the numbers to POKE into the locations in the Memory Map. Let's just take one more example. If Proto was a Sprite, he would have a number from 0 to 7. Sound familiar? To let Proto be Sprite number 0, turn on the zero BIT in location 53269.

POKE 53269,1

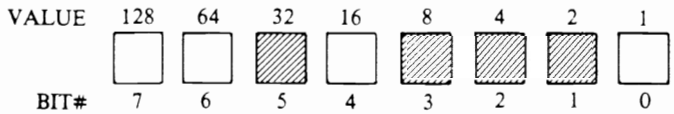


For Proto to be Sprite number 3, turn on BIT 3.

POKE 53269,8

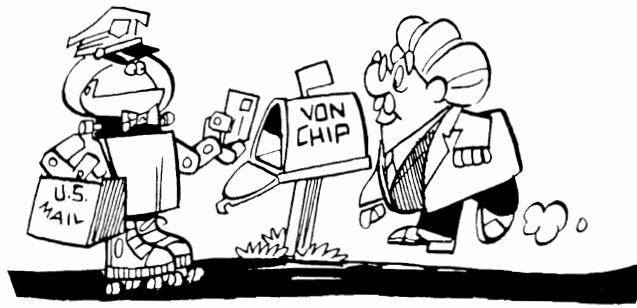


To have 4 different Sprites on the screen, turn on the BIT for each Sprite.



Turns on Sprites #5, 3, 2 and 1.

Confusing? Just read this a few thousand times and play, play, play with the locations in the Master Memory Map. The more you play, the more you learn!



LOWER ADDRESSES

LABEL

Hexadecimal Location	Decimal Location	Decription and How to Use

If you're interested in programming only in BASIC, the first two locations won't be of much interest. Later, when you want to try machine language programming, take a good look here.

D6510

0000 0 This is the place on the computer's main processor (the 6510) or "brain", that monitors or "looks at" the information going in and out of itself. This location is not useful if you program only in BASIC. If you are interested in writing machine language programs this is what you need to know:

NAME	BIT	DIRECTION	DESCRIPTION
LORAM	0	OUTPUT	Control for RAM/ROM at $A000–$BFFF (BASIC)
HIRAM	1	OUTPUT	Control for RAM/ROM at $E000–$EFFF (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

If the bit is set to 0, input is coming from the memory block. If the bit is set to 1, the processor is sending information to the routine. Normally this location is set to 47.

R6510		
0001	1	This location is a little more useful to the BASIC programmer. It is responsible for dividing the memory into pieces the computer can handle.

In addition to 64K bytes of RAM memory, the Commodore 64 has 20K bytes of ROM containing the Kernal, screen editor and BASIC interpreter plus another 4K for the character generator ROM. Another 4K to 8K is available in expansion cartridges. This effectively allows the microprocessor to look at 92K bytes of memory! However, only 64K is accessible to the 6510's address space at any one time.

Before we go too much further, here's a practical use of this location. Using this address and others you can create your own character sets for graphics or animation. To do this you must move the character generator ROM into RAM so you can change an "A" into a tiny Proto! See the appendix for an example of creating your own characters.

If you are interested in what happens when each bit is turned on or off, here's a handy list of the locations and the different kinds of memory configurations that can be made.

Bit#

0 LORAM SIGNAL (0 = switch BASIC ROM OUT)
1 HIRAM signal (0 = switch KERNAL ROM OUT)
2 CHAREN signal (0 = switch character ROM IN)
3 Cassette data output line
4 Cassette switch sense 1 = switch closed
5 Cassette motor control 1 = ON, 0 = OFF

Bits numbers 6 and 7 are not used.

MAP #	BITS 2-1-0	FUNCTION
Map - 0	111	38K BASIC
Map - 1	X01	60K RAM & I/O
Map - 2	110	Z-80 CP/M Cartridge
Map - 3	100	64K free memory
Map - 4	111	Expansion count
Map - 5	110	Assembler/wordprocessor
Map - 6	111	No BASIC
Map - 7	1XX	MAX machine games

Note: X = not available for user

The "Max Machine" is a game machine that Commodore hasn't released in the U.S. yet. The game's cartridges will also work on the 64. If the Max Machine game cartridge is plugged in, map 7 is not available to you.

If it isn't fun figuring out the values for this location, remember that it is affected by different cassette operations (see the appendix "Reconfiguring the Memory Map" for diagrams of the different map possibilities). If you think you really have problems

POKE 1,255

and your ROMs will reset and the system will be its good old self again. This is one place where pressing RUN/STOP and RESTORE won't work.

0002	2	This is used by the computer to temporarily store information it needs. Don't use this address, it could upset the machine.
ADRAY1 0003-0004	3-4	Jump vector: convert floating-point real numbers to integer.
ADRAY2 0005-0006	5-6	Jump vector: convert integer numbers to floating-point numbers.

CHARAC 0007	7	Search character. This location stores the ASCII value for a quote (34).
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 The value changes with the last disk or tape operation.
COUNT 000B	11	Input buffer pointer number of subscripts.
DIMFLG 000C	12	Flag: default array DIMension This location holds the value of the first letter of the most recent dimensioned array.
VALTYP 000D	13	This shows the type of data being read. 255 if it is string data and 0 if it is numeric.
INTFLG 000E	14	This shows data type for numbers. 128 for integer and 0 for floating point.
GARBFL 000F	15	Flag: DATA scan for LIST and garbage collection. Normal value: 4.
SUBFLG 0010	16	Flag: Subscript reference and user function call. This location doubles as flag register for these two functions.

INPFLG 0011	17	Flag: BASIC input types. 0 = INPUT, 64 = GET, 152 = READ This shows 0 when power is turned on. The value will change as the input type changes.
TANSGN 0012	18	Flag: TAN sign/comparison results. This flag tests SIN/COS division results to verify the tangent sign. The normal value is 0. This value becomes 255 when TAN argument is greater than 259.1.
0013	19	Flag: input prompt.
LINNUM 0014-0015	20-21	Temporary storage of integer value. BASIC stores integer variables used in calculations here. The routines in locations 3 - 4 and 5 - 6 use the number stored here.
TEMPPT 0016	22	Pointer: temporary string stack. Normally points to TEMPST (25).
LASTPT 0017-0018	23-24	Last temporary string stack.
TEMPST 0019-0021	25-33	Stack for temporary strings.
INDEX 0022-0025	34-37	The addresses stored here point to machine subroutines stored in the BASIC ROM.
RESHO 0026-002A	38-42	This holds the results of floating point multiplication. These locations are used by the system ROM multiply

routines whenever very large positive or negative numbers are multiplied.

TXTTAB
002B-002C 43-44 Pointer: start of BASIC text. Normal value: $0801(2049).

Now here's a location everyone can use! These addresses point to the place the computer looks for your programs. You can fool the computer into thinking it has less room than it really has so you can "hide" a program. This location combined with others gives you:

AN APPEND ROUTINE

With an append routine you can load one program after another and keep both programs in memory! Why would you want to do this? Here is an example. If you include a joystick routine in every program you write, then typing in the same lines time after time is something you can give up. First, just save the joystick routine separately. Give it high line numbers like 20000 or 30000. Then, whenever you need it, use the append routine to add it to any program you want! The high line numbers should keep your joystick routine above the rest of your program so you won't need to renumber them.

The programs for this location were printed using Midwest Micro Associates printer interface program "Smart ASCII". Each Commodore cursor control character or color character is translated into English and printed within parentheses.

(CLR) means the CLEAR/HOME key and will look like a reversed heart when you enter the program. (DN) means cursor DOWN, (LF) means cursor LEFT, and (HM) is cursor HOME (without clearing the screen). Two color controls are used, (BLU) for blue and (WHT) for white. Use the appropriate keys to produce these colors.

If you use a cassette to save programs,

1. Enter the following program:

```
1 A=PEEK(44)
2 PRINT"(BLU)(CLR)(DN)(DN)(DN)LOAD(DN)(DN)
  (DN)(DN)(DN)(DN)(DN)(DN)(DN)(DN)(LF)(L
  F)(LF)(LF)POKE43,1:POKE44,";A;"(HM)(WHT)"
3 FORI=631TO636:POKEI,13:NEXT
4 POKE198,6
5 IFPEEK(45)<2THENPOKE43,PEEK(45)-2+255:
  POKE44,PEEK(46)-1:END
6 POKE43,PEEK(45)-2:POKE44,PEEK(46):END      C!
```

2. Put the cassette with the first program you want to add into the cassette drive.

3. Run the append program and follow the directions given on the screen.

When the word "READY" appears, type LIST and you will see the second program listed AFTER line 6. To add another, put the new program in the cassette and type RUN again. When you have added all the programs you want, delete lines 1 through 6.

If you have a VIC-1541 disk drive, make the following changes and additions to the program:

```
1 INPUT"ADD WHICH PROGRAM";F$
2 A=PEEK(44):Q$=CHR$(34)
3 PRINT"(BLU)(CLR)(DN)(DN)(DN)LOAD";Q$;F$;
  Q$;",8(DN)(DN)(DN)(DN)(DN)(DN)(DN)(DN)
  (DN)(DN)(LF)(LF)(LF)(LF)POKE43,1:POKE44,
  ";A;"(HM)(WHT)"
4 FORI=631TO640:POKEI,13:NEXT
5 POKE198,10
6 IFPEEK(45)<2THENPOKE43,PEEK(45)-2+255:
  POKE44,PEEK(46)-1:END
7 POKE43,PEEK(45)-2:POKE44,PEEK(46):END      C!
```

Type Run and follow the directions given on the screen. When you finish, be sure to delete lines 1 to 7. If you have programs with the same

line numbers, you will have to renumber both programs. For a handy renumbering program, see address 2040. You can see how the program works by POKEing your screen to a light gray (POKE 53281,12) so all the commands will show.

Briefly, this is what happens. First, adjustments are made in locations 43 to 46 to move this program safely out of the way of the program you are about to add.

Then in line 3, two things happen. First the LOAD command is printed, then the POKE commands that will be needed later to find our hidden program. In line 4, you put carriage returns in the keyboard buffer (locations 631-640) and tell location 198 how many you put there. Line 5 "hides" any program in memory from the computer so the program isn't lost when you add the new one. Then the program ends.

Ending the program causes the computer to act on anything in the input buffer, which we have conveniently stuffed with carriage returns (CHR$ (13)). Since the program has already printed commands on the screen, the carriage return enters these commands just as if you did it yourself. That's a lot for only 7 lines of code!

Here's a scrolling routine that uses some of the same concepts. This routine will let you see one line of program at a time.

```
63000 REM**  +/- LIST  **
63001 SA=PEEK(44)*256+PEEK(43)-1
63002 LN=PEEK(SA+3)+PEEK(SA+4)*256
63003 PRINT"(CLR)(BLU)GOTO 63010":PRINT
      "LIST";LN;
63004 POKE631,19:POKE632,17:POKE633,5:
      POKE634,13:POKE635,19:POKE636,13
63005 POKE198,6:END
63010 IF PEEK(197)=43 THEN 63100:REM TEST
      FOR "-" KEY
63020 IF PEEK(197)=40 THEN 63200:REM TEST
      FOR "+" KEY
63030 GOTO 63010
63100 IF PEEK(SA+5)<>0 THEN SA=SA+1:GOTO
      63100
63110 SA=SA+5:GOTO 63002
63200 SA=SA-1:IF PEEK(SA)=0 AND PEEK(SA-4)
      <>0 AND PEEK(SA-3)<>0 THEN 63002
63210 GOTO 63200
```

Here's how to use it.

1) Load the program you want to scroll.

2) Enter the following BASIC *without* using line numbers (this is called the immediate mode).

CLR: POKE 43,PEEK(45)-2:POKE 44, PEEK (46).

This makes BASIC think it has less memory for the next step.

3) Load the scrolling program.

4) Enter the following in the direct mode.

POKE 43,1:POKE 44,8

5) Type 'RUN' and press return.

6) Press the minus (−) key to scroll down to the next line. Press the plus (+) key to scroll up one line. You must scroll down at least one line before you scroll up or the program will end.

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).

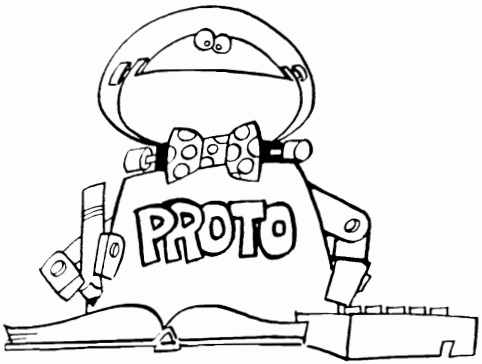
These locations are similar to location 43-46. These give the area where arrays are stored after they are DIMentioned in your program. Being able to find the starting addresses of your string arrays will be needed if you're storing short machine language subroutines in them.

This sample program creates 3 different types of arrays then allows you to see how the computer stores them. Press any key to look at each location. Press RUN/STOP to end.

```
100 REM **** ARRAY SAMPLES ************
110 DIM I%(5,5), C$(5), R(5)
120 :
130 REM **** ARRAY POINTER FN *********
140 DEF FNARRAY(X)=PEEK(X)+256*PEEK(X+1)
150 :
160 REM **** STUFF THE ARRAYS *********
170 FOR I = 1 TO 5
180 R(I)=I
190 C$(I)= CHR$(65+I)
200 FOR J = 1 TO 5: I%(I,J)=J:NEXT
210 :
220 REM **** FIND ARRAYS IN MEMORY ****
230 AS = FNARRAY(47): REM ARRAYS START*
240 AE = FNARRAY(49): REM ARRAYS END***
250 :
260 REM **** PRINT ARRAY ELEMENTS *****
270 FOR T= AS TO AE
280 PRINT T,PEEK(T)
290 WAIT 197,64,64: REM PRESS ANY KEY**
300 NEXTT
310 CLR
320 REM **** PRESS RUN/STOP TO END ****
330 GOTO 330
```

FRETOP		
0033-0034	51-52	Pointer: Bottom of string storage.
FRESPC		
0035-0036	53-54	Utility string pointer.

These two locations point to the end of the Random Access Memory, (RAM), used to hold strings like "Hello" or "My name is Prototype".



Often, programmers use these locations and the next two in a combination that allows them to hide part of the memory from the computer. Typically, you may want to reserve a large portion of your memory for keeping things like longer machine code subroutines, binary data files, or fonts (modified character sets) resident but not accessible by BASIC interpreter routines.

The next two locations show how this is done.

MEMSIZ		
0037-0038	55-56	Pointer: Highest address used by BASIC. In other words, your programs won't be stored beyond the value shown in these 2 bytes. Normally this is set to 40960 but you can change that if you want.

To set aside 4K of memory for machine programs, graphic characters or sprites in the program mode, use this routine in the 'immediate mode'.

```
POKE 51,0:POKE 52,144:POKE 55,0:POKE 56,144:CLR
```

This will save the area from 36864 to 40960. (9000-9FFF in hexadecimal notation)

```
POKE 51,0:POKE 52,160:POKE 55,0:POKE 56,160:CLR
```

This will restore the machine to normal.

CURLIN
0039-003A 57-58 Current BASIC line number.

OLDLIN
003B-003C 59-60 Previous BASIC line number.

These address registers point within memory used by BASIC where the program line numbers are stored. You can determine these line numbers by

PRINT PEEK(57)+256*PEEK(58)

OLDTXT
003D-003E 61-62 This contains the address where BASIC code is to resume after the program has been STOPped. It is used with the BASIC command, CONT. This is normally set at the beginning of BASIC (2048) when there is no program resident.

DATLIN
003F-0040 63-64 Current DATA line number. These locations store the line number of either the first DATA statement in a program or the current DATA statement being read.

```
100 DIM IN%(3)
200 FOR I = 1 TO 3
300 READ IN%(I)
400 NEXT
500 PRINT(PEEK(63)+256*PEEK(64))
600 DATA 1,2,3
```

Running this program will show the number 600 as the current DATA line. By using a routine like this while you're de-bugging your program, you'll be able to see if all of your DATA statements are being read.

DATPTR
0041-0042 65-66 Pointer: This is the address of the current DATA item. When no

DATA statements are encountered, this location defaults to 2048.

Did you know that each type of DATA needs a different amount of memory to store each part? Whole numbers take 2 bytes, characters take 3 bytes and floating point numbers take 4 bytes.

INPPTR

0043-0044	67-68	This address shows where the INPUT statement temporarily stores its data, which is the system INPUT buffer starting from locations 512 to 600 (78 bytes). You can't POKE this location to change the size of the INPUT buffer. By the way, trying to put more information in the buffer than it will hold will cause the first 78 characters to be lost.

VARNAM

0045-0046	69-70	Current BASIC variable name. When a single letter variable has been used, this location returns its ASCII value.

VARPNT

0047-0048	71-72	Pointer: Current BASIC variable data. This is just another location which points to base address of the BASIC variable table (normally 2084).

FORPNT

0049-004A	73-74	Pointer: Index variable for FOR/ NEXT. loops.
004B-0060	75-96	Temporary pointer / data area.

The 16 locations from 97 to 112 are arguments for the floating-point arithmetic routines stored in the KERNAL ROM. They can be used in BASIC, but they provide the assembly language programmer with the ability to perform any arithmetic operations on two numbers (accum #1 and #2) without writing those routines BASIC uses.

These locations are also very useful for machine language subroutines that are called from BASIC. See locations 784-787.

If you are interested in mastering machine level programming, take a look at Rodnay Zak's books on 6502 programming. They are concise and thorough.

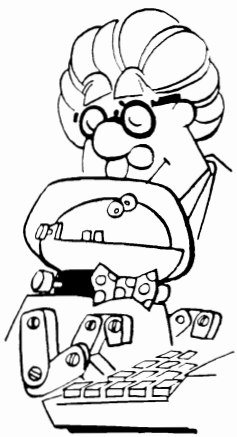
FACEXP 0061	97	Floating-point accumulator #1: exponent.
FACHO 0062-0065	98-101	Floating accumulator #1: mantissa.
FACSGN 0066	102	Floating accumulator #1: sign.
SGNLG 0067	103	Pointer: Series evaluation constant.
BITS 0068	104	Floating accumulator #1: overflow digit.
ARGEXP 0069	105	Floating-point accumulator #2: exponent.
ARGHO 006A-006D	106-109	Floating accumulator #2: mantissa.
ARGSGN 006E	110	Floating accumulator #2: sign.
ARISGN 006F	111	Sign Comparison Result: Accum. #1 vs #2.
FACOV 0070	112	Floating accumulator 1. Low-order (rounding).
FBUFPT 0071-0072	113-114	Pointer: cassette buffer.

Locations from 115 to 138 are part of the computer's operating system that are deliberately put in RAM for the user to be able to change it. For instance, you could add commands to BASIC.

CHRGET
0073-008A 115-138 This machine language subroutine is what the computer uses to get characters or tokens from the keyboard, cassette, disk or modem.

Here is what it looks like disassembled:

```
,0073 E6 7A     INC $7A
,0075 D0 02     BNE $0079
,0077 E6 7B     INC $7B
,0079 AD 60 EA  LDA $EA60
,007C C9 3A     CMP #$3A
,007E B0 0A     BCS $008A
,0080 C9 20     CMP #$20
,0082 F0 EF     BEQ $0073
,0084 38        SEC
,0085 E9 30     SBC #$30
,0087 38        SEC
,0088 E9 D0     SBC #$D0
,008A 60        RTS
```



The next two locations are checked by the CHRGET routine.

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
0900 144 Kernal I/O status word (ST). Normal = 0. ST is a reserved word in BASIC. you can't use it as a variable (like"A" or"B2") in your program. The value of ST will change if there is a problem loading a program from tape or disk.

```
10 REM---ADD THIS BEFORE ANY INPUT# OR
20 REM---GET# STATEMENT TO PREVENT A
30 REM---PROGRAM FROM STOPPING AFTER THE
40 REM---LAST DATUM HAS BEEN READ.
50 REM---FROM THE TAPE OR DISK FILE.
60 IF ST = 64 THEN 90
70 :
80 :
90 PRINT"END OF DATA "
```

ST Bit Position	ST Numeric Value	Cassette Read	Serial/RW	Tape Verify or Load
write 0	1		Time out	
read 1	2		Time out	
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

*Evaluate or Input

STKEY
0091 145 Flag: STOP key and RVS key.
Normal: 255
POKE 145,127 will act as a RUN/STOP key.

SVXT
0092 146 Timing constant for tape.

VERCK
0093 147 Flag: 0 = LOAD, 1 = VERIFY
Stores the last tape or disk LOAD or VERIFY operation.

C3PO		
0094	148	Flag: Serial Bus, output character buffers.
BSOUR		
0095	149	Buffered char. for serial bus.
SYNO		
0096	150	Cassette synchronization number.
0097	151	Temporary data area.
LDTND		
0098	152	Number of open files and index to file table. Only 10 files are allowed open. If you try to open more than 10 files at any time your program will stop and show a "too many files open" error. If you're opening close to ten files in a program, add lines like these to prevent the problem.

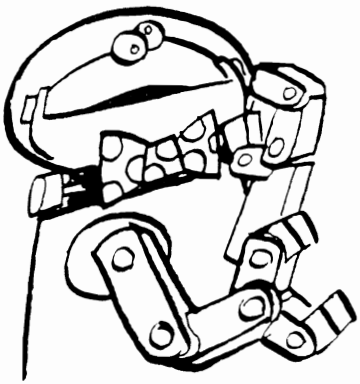
```
200 IF PEEK(152)=10 THEN 400
300 OPEN 1,1,1
310 GOTO 500: REM--CONTINUE
400 PRINT"10 FILES ARE OPEN."
410 PRINT"DO YOU WANT TO CLOSE"
420 INPUT"ANY";R$
```

DFLTN		
0099	153	Default input device normally set to the keyboard (0).
DFLTO		
009A	154	Default ouput (CMD) device normally set to the screen (3).

When you turn the power to the computer on, the screen is where you see information. The BASIC command CMD changes that if you use an open statement. For example, to have a printer send a listing of a program you already have in memory, do this:

OPEN 1,4:CMD 4:LIST

Here's a list of the different numbers used.



DEVICE NUMBER	PERIPHERAL
0	Keyboard
1	Cassette
2	Modem
3	Screen
4 or 5	Printer
8 to 11	Disk Drive(s)

PRTY
009B 155 Tape character parity.

DPSW
009C 156 Flag: Tape byte-received.

MSGFLG
009D 157 Flag: 128 => Direct mode 0 = Program mode.

PTR1
009E 158 Tape pass 1 error log.

PTR2
009F 159 Tape pass 2 error log.

TIME
00A0-00A2 160-162 This is the "JIFFY" clock. A jiffy is 1/60th of a second. The computer uses this to help time the different operations it performs. Here's what these locations do: every 1/60th of a second 1 is added to the number in 162 until it reaches 255. Then, 162 turns to zero and 1 is added to 161 until it reaches 255. Then 161 turns to zero and 1 is added to 160. When 160 reaches 255 all 3 locations are set to zero. You can use this location to make a stopwatch to check your reaction time.

```
100 PRINTCHR$(147)
110 PRINT"        STOPWATCH/REACTION TIME"
120 POKE53280,1:POKE53281,1:POKE646,6
130 PRINT
140 PRINT
150 PRINT"WHEN YOU SEE THE BLUE HEART,"
160 PRINT"PRESS THE SPACE BAR AND YOUR"
170 PRINT"REACTION TIME WILL BE SHOWN."
180 S=INT(10*RND(1))
190 FOR PAUSE= 1 TO S*300:NEXT
200 FOR D=1 TO 3:PRINTCHR$(17);:NEXT
210 FORR=1 TO 12:PRINTCHR$(29);:NEXT
220 PRINTCHR$(115)
230 FORI=160TO162:POKEI,0:NEXT
240 IFPEEK(197)=64 THEN 240
250 PRINT(PEEK(160)*65536+PEEK(161)*256+
    PEEK(162))/60
```

This will show how quickly you can hit a key. This routine will allow you to read time in amounts less than 1 second! In BASIC, time is kept in the string TI$ and in a jiffy counter, TI. TI starts at "000000" when you turn the computer on. The computer uses a "24 hour clock" which means that one o'clock in the afternoon is called 13:00. You can reset the clock to zero by entering TI$ = "000000" or set it to the correct time of day (right to the second) by saying, for example TI$ = "041325" for 4 o'clock, 13 minutes and 25 seconds AM while 4 PM is "161325". As for TI, well it's a counter based on the information in locations 160-162. Both TI$ and TI will reset to zero by POKEing these locations with a zero.

For I = 160TO162:POKEI,0:NEXT

Try substituting this for line 270

```
270 PRINT TI/60;" SECONDS"
```

Here's an example of using TI$ to produce a clock on your screen.

```
100 PRINTCHR$(147)
110 POKE53280,1:POKE53281,1:POKE646,6
120 PRINT,"HOURS","MINUTES","SECONDS"
130 PRINTCHR$(19);
140 PRINT,MID$(TI$,1,2),MID$(TI$,3,2),
    MID$(TI$,5,2)
150 GOTO130
```

TI$ and TI are also reserved words used by BASIC. Don't use these names for your variables.

00A3-00A4	163-164	Tempory data area.
CNTDN 00A5	165	Cassette sync. coutdown.
BUFPNT 00A6	166	Pointer: tape I/O buffer.

These are the zero-page memory locations for the RS-232. The system's software allows compatability with any RS-232 device (printer, modem, etc.). This allows RS-232 programming accessible in BASIC through the Kernal routines. These locations are used directly by the RS-232 device through the system routines and are not controlled by the programmer. See the appendix on the RS-232 in back. During cassette or any other serial bus activities, data can NOT be received from RS-232 devices. This is how temporary cassette data locations can be shared with RS-232 locations.

INBIT 00A7	167	RS-232 input bits/cassette temporary.
BITCI 00A8	168	RS-232 input bit count/cassette temporary.

RINONE
00A9 169 RS-232 flag: Check for start bit.

RIDATA
00AA 170 RS-232 input byte buffer/cassette temporary.

RIPRTY
00AB 171 RS-232 input parity/cassette short count.

SAL
00AC-00AD 172-173 Pointer: Tape buffer/screen scrolling.

EAL
00AE-00AF 174-175 When you LOAD or SAVE, the computer looks at a part of memory to read from or write over. If you or your program changes the start of BASIC variable pointers (locs. 45-46) so you can store more BASIC text, your cassette or disk drive won't be aware of the change and will lose some text. Whenever you change locations 45-46, and you want to SAVE or LOAD a program on cassette. Make sure you POKE these locations with the same values as locations 45 and 46. This will make sure you save all of your program.

CMPO
00B0-00B1 176-177 Tape timing constants.

TAPE1
00B2-00B3 178-179 Pointer: start of tape buffer. This can be used as an indirect zero-page jump routine in the buffer.

BITTS
00B4 180 RS-232 out bit count/cassette temporary.

Locations 180 thru 182 act the same as 166 thru 171.

NXTBIT
00B5 181 RS-232 next bit to send/tape EOT (End-Of-Tape) flag.

RODATA
00B6 182 RS-232 out byte buffer.

FNLEN
00B7 183 Length of current file name. The number of characters in the file name itself. Same as PRINT(LEN (FILE$)) where FILE$ is the name of your file.

LA
00B8 184 Current logical file number.

SA
00B9 185 Current secondary address. In BASIC, the secondary address is used with the OPEN command. The secondary address tells the other device what to do after it is opened.

DEVICE	DEVICE NUMBER	SECONDARY ADDRESS	DESCRIPTION
Cassette	1	0=Input 1=Output 2=Output	file name end of tape
Modem	2	0	control register
Printer	4 or 5	0=uppercase/ graphics 7=upper/lower	PRINT text
Disk	8 to 11	2-14 data channel 15=command channel	drive#, file type, read/write command

For example:

OPEN 1, 4, 7

This OPENS a channel to the printer (device 4) and tells the printer to type upper/lower case letters.

If you want to write some data about your debts on a cassette tape,

OPEN 3, 1, 2, "DEBTS"

tells the computer that file #3 (it could be any number you choose between 1 and 255) will be written as a cassette file (the number 1 after the 3), and, besides writing the data on the tape, put an "End Of Tape" notice (the number 2). The "End Of Tape" (EOT) will tell the computer to stop looking for more data on this tape. **Only use the EOT mark on the last data file or program you put on a cassette.** "DEBTS" will be the name of the file.

FA 00BA	186	Current device number.
FNADR 00BB-00BC	187-189	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	Used for temporary storage of information while information is loading from the cassette.
LSTX 00C5	197	Current key pressed. The last key put into the keyboard buffer. There will be a 64 here if a key isn't held down. The "stopwatch" program under location 160-162 uses this address to see if any key has been pressed.

If more than 1 key is held down at the same time, the key with the highest priority will appear on the screen.

This program will give you a list of the keys and their priority.

```
100 GET A$
110 IF A$= "" THEN 100
120 PRINT PEEK(197)
130 GOTO 100
```

PRIORITY / KEY CHART

#	KEY	#	KEY	#	KEY	#	KEY
0	INST	16	5	32	9	48	
1	RET	17	R	33	I	49	*
2	⇄ CRSR	18	D	34	J	50	]
3	f7	19	6	35	0	51	CLR
4	f1	20	C	36	M	52	
5	f3	21	F	37	K	53	=
6	f5	22	T	38	0	54	↑
7	↑↓ CRSR	23	X	39	N	55	/
8	3	24	7	40	+	56	1
9	W	25	Y	41	P	57	←
10	A	26	G	42	L	58	
11	4	27	8	43	−	59	2
12	Z	28	B	44	.	60	SPC
13	S	29	H	45	[	61	
14	E	30	U	46	@	62	Q
15		31	V	47	,	63	

NDX
00C6 198 Number of characters presently in the keyboard buffer. Normal value: 0. Poking this location with a zero will empty the buffer. This is very useful in games or other applications where keys will be struck quickly. You should always empty the buffer before using the GET statement.

```
100 POKE 198,0
110 PRINT"PRESS ANY KEY TO CONTINUE"
120 GET R$
130 IF R$= "" THEN 120
```

```
100 POKE 198,0
110 PRINT"PRESS 'C' TO CONTINUE"
120 GET R$
130 IF R$ <> "C" THEN 120
```

The first program checks for any key. The second will continue if only one key is pressed.

RVS

00C7	199	Reverse character switch. Normal: 0. The value switches to 18 when reversed characters are printed.

POKE with 1 or 18 before every PRINT to reverse characters. It's easier to use the RVS ON and RVS OFF keys.

INDX

00C8	200	Pointer: end of the logical file for INPUT. This location returns the screen column number of the end of

the INPUT record. Unless you use a semicolon or the cursor keys inside a string, the computer will put the input question mark in the third column of the screen.

Here are some examples of how to change the position of the INPUT.

```
10 INPUT" NAME";N$
10 INPUT" HELLO  DO YOU NEED HELP";N$
10 INPUT"ADDRESS";N$
```

LXSP

00C9-00CA	201-202	Cursor X-Y position at start of INPUT. These two bytes store the X, Y coordinates respectively where the

INPUT statement accepts data. POKEing these locations has no affect on the screen positioning of INPUT records.

SFDX

00CB	203	Flag: Print shifted characters. Normal value is 64. This location will show the value listed in 197 when characters are printed.

CURSOR LOCATIONS

Use these locations to produce a cursor during a GET statement.

BLNSW
00CC 204 Flag: Cursor blink toggle.
0=cursor on 1=cursor off.

```
10 POKE204,0:POKE207,0:GETA$
20 IFA$=""THEN 10
30 PRINTA$;
40 IFA$=CHR$(13)THEN70
50 B$=B$+A$
60 GOTO10
70 PRINTB$
```

BLNCT
00CD 205 Timer: Countdown to toggle cursor.
Normal value: 2.

GDBLN
00CE 206 Character under cursor in ASCII.
Normal value is 32.

BLNON
00CF 207 Flag: Last cursor blink on/off.
Normal value is 0.

Here's how a sample program to add a cursor might look:

```
100 REM--CURSOR WITH GET
200 B$=""
300 POKE204,0:POKE207,0:GET A$
400 IF A$="" THEN 300
500 PRINTA$;
600 IFA$=CHR$(13) THEN 900
700 B$=B$+A$
800 GOTO 300
900 PRINTB$
```

If you want the cursor to blink during a GET, you should have the POKEs on the same line as the GET command. If you press return while the cursor is on it will stay on the screen as a square. This won't change the value of your input. Run the demonstration program and press RETURN while the cursor is on. Notice the variable B$ ignores the cursor.

CRSW 00D0	208	Flag: INPUT or GET from keyboard. Normal value: 0.
PNT 00D1-00D2	209-210	Pointer: Current screen line address. POKEing this address can position word output on the screen. This is similar to using cursor controls, but not as easy.

For example:

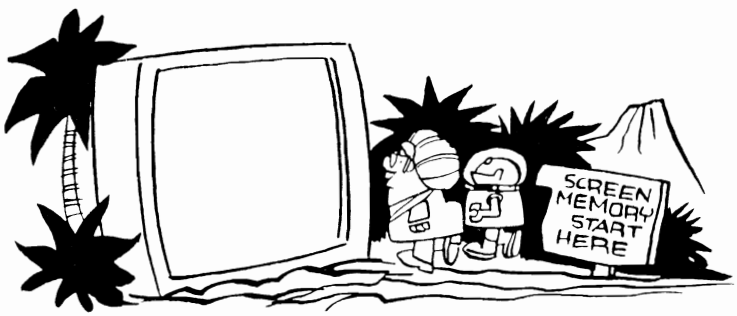
```
10 POKE 209,0
20 POKE 210,4
30 PRINT "HERE I AM"
```

will print the string in Line 20 at the top left part of the screen.

Here's another program example that will show you the screen address.

```
100 REM ************************************
110 REM *          SCREEN ADDRESS          *
120 REM ************************************
130 :
140 REM >> CLEAR SCREEN
150 PRINT CHR$(147)
160 :
170 REM >> STORE CURSOR MOVEMENTS
180 RT$ = CHR$(29):REM --> RIGHT
190 DN$ = CHR$(17):REM --> DOWN
200 :
210 REM >> FIND THE POSITION OFF 11
220 REM >> SCREEN ADDRESSES IN A LOOP
230 FOR I= 1 TO 11
```

```
240 :
250 REM >> FIND CURRENT SCREEN ADRESS
260 SA = PEEK(209) + 256*PEEK(210)
270 :
280 REM >> MOVE CURSOR TO THE RIGHT
290 RGHT$ = RGHT$ + RT$
300 :
310 REM >> PRINT ADRESS OF CHECK BOX
320 PRINT RGHT$ DN$;CHR$(166);
330 PRINT "SCREEN ADRRESS = ";SA
340 :
350 NEXT I
360 GO TO 360
```



PNTR
00D3 211 Cursor column on current line. This location stores the number of spaces on a printed line, just like using the BASIC command TAB(X) where X would be the number of spaces you want to move forward.

QTSW
00D4 212 Flag: Editor in quote mode. 0 means you are not in the quote mode.

LNMX
00D5 213 Physical screen line length. Normal default value: 39. POKEing this location with a value less than 39 will limit printed items on a line to that number by truncation and will NOT be wrapped around.

This program is a demonstration. Line 20 makes a 40 character string by adding the word to itself 4 times.

```
100 REM *** CONCATENATION ***
110 :
120 PRINT CHR$(147): REM - CLEAR SCREEN
130 REM - CONCATENATE A STRING
140 C$="BLIZZARD"
150 C$=C$+C$+C$+C$
160 :
170 REM - TRUNCATE STRING IN LOOP
180 FOR I = 32 TO 8 STEP -1
190 :
200 REM - SHORTEN PHYSICAL LINE LENGTH
210 POKE213,I
220 :
230 REM - EXAMINE STRING CONTENTS
240 :
250 FOR C = 1 TO LEN(C$)
260 :
270 PRINTMID$(C$,C,1);
280 :
290 REM - PAUSE A MOMENT
300 FOR PAUSE = 1 TO 30: NEXT
310 :
320 REM - GET NEXT VALUES
330 NEXT C
335 PRINT
340 NEXT I
```

TBLX

00D6	214	Current cursor physical line number. Normal values: 0-24. POKE this location with values from 0 to 255 to move the cursor's vertical position.

This program will make two words, "SELECT" and "LINE" scroll up and down over each other.

```
100 REM ************************************
110 REM *          SCREEN ROW              *
120 REM ************************************
130 :
140 REM => CLEAR SCREEN
150 PRINT CHR$(147)
160 :
170 REM => INITILIZE SCREEN ROW VALUES
180 Y1 = 0: Y2 = 22
190 :
200 REM => INCREMENT/DECREMENT VALUES
210 FOR I = Y1 TO Y2
220 Y1 = Y1 + 1
230 Y2 = Y2 - 1
240 :
250 REM => SET NEW ROW VALUE & PRINT
260 REM => ANY OLD STRING
270 POKE 214,Y1:PRINT CHR$(13)
280 FOR R=1TO12:PRINTCHR$(29);:NEXT R
290 PRINT"SELECT";CHR$(146)
300 :
310 REM => RESET VALUE AND PLUG IN
320 REM => ANOTHER STRING
330 POKE 214,Y2:PRINT CHR$(144)
340 FOR R=1TO13:PRINTCHR$(29);:NEXT R
350 PRINT"LINE";CHR$(5)
360 :
370 REM => SWITCH VALUES AT THE END
380 IF Y1=22 THEN Y1=0: Y2 = 22
390 :
400 REM => NEXT ROW VALUES
410 NEXT I
420 :
430 REM => RETURN TO REPEAT
440 GO TO 210
```

00D7	215	Temporary data area.
INSRT 00D8	216	Flag: Insert mode or the number of inserts from the left on a given PRINTed line. POKEing with a 0 turns off insert mode.
LDTB1 00D9-00F2	217-242	Screen line link table and also temporarily used as a single editor buffer in the immediate mode.

There are two ways of talking about lines in most computers. The line you just worked with each time after pressing the RETURN key is the logical line. Each line you see on the screen is a physical line. When a logical line takes up more than that one physical line, those lines are "linked" together and stored in a link table. The computer treats them as if they were all one long line. These locations are where those links are stored. To demonstrate how links can be used in a program, see the 'Alternate Screens' example under location 648.

USER 00F3-00F4	243-244	Pointer: Current screen color RAM location. This address points to the area used to hold the colors for each locations on the screen (see 55296 for more information).
KEYTAB 00F5-00F6	245-246	Vector: Keyboard decode table.
RIBUF 00F7-00F8	247-248	RS-232 input buffer pointer.
ROBUF 00F9-00FA	249-250	RE-232 output buffer pointer.
FREKZP 00FB-00FE	251-254	Free zero-page space for user program. This is a COMMODORE freebee! These locations provide

four bytes out of the first page of memory so you can write your assembly language routines in the zero-page indexed mode.

BASZPT 00FF	255	BASIC temporary data area.

Locations 256 to 511 are used by the microprocessor stack. If you're programming in BASIC, you won't be using these locations.

0100-010A	256-266	Floating to string work area.
BAD 0100-013E	256-318	Tape input error log.
013F-01FF	319-511	This area is used by the stack and other system operations and is unavailable for other uses.
BUF 0200-1258	512-600	System INPUT buffer. This is where the information goes when you type on the computer. It only stays here until the information is needed elsewhere.

These next 3 locations also aren't that useful to the BASIC programmer. They store the parameters for channels OPENed and accessed by Kernal routines. These behave in the same way as locations 184-186 except that they store all the file information and not just that of the most recently opened channel. They each take up 10 bytes. This is the limiting number of files you are allowed open at any one time. More on the Kernal is shown in "Addressing the KERNAL".

LAT 0259-0262	601-610	Kernal table: Active logical file numbers.
FAT 0263-026C	611-620	Kernal table: Device number for each file.
SAT 026D-0276	621-630	Kernal table: Second address each file.

KEYD

0277-0280	631-640	Keyboard buffer queue. This area is where characters typed in from the keyboard are temporarily stored. If characters are POKEd in here and location 198 (which holds the number of characters in the buffer) is changed, it will be as if the characters were typed from the keyboard.

The append routine at location 43 uses this location to make the computer think something has been typed in.

MEMSTR

0281-0282	641-642	Pointer: Bottom of memory operating system. Normal value is 2048.

MEMSIZ

0283-0284	643-644	Pointer: Top of memory for operating system. Normal value is 40960.

These are important places! They control the amount of space you have to write programs. These values are set by the INPUT/OUTPUT control register (location 1) during power-up and may be different with each memory map configuration. Don't confuse these locations with the ones that locate parts of the BASIC text (locations 43 thru 56).

There are lots of pointers used by BASIC to find out where the computer stores the different parts of your program. Often, it's good for you to know where, too. If you write programs that modify different places in memory, this little program could be of help. It keeps track of these pointer values.

```
100 PRINTCHR$(147)
110 DEF FNT(X)=PEEK(X)+256*PEEK(X+1)
120 PRINT"START OF PROGRAM--";FNT(43)
130 PRINT"END OF PROGRAM AND"
140 PRINT"START OF VARIABLES--";FNT(45)
150 PRINT"LENGTH OF PROGRAM ="
160 PRINT(FNT(45)-FNT(43));" BYTES."
170 PRINT
180 PRINT"END OF VARIABLES AND"
190 PRINT"START OF ARRAYS =";FNT(47)
200 PRINT
```

```
210 PRINT"NUMBER OF VARIABLES =";
220 PRINT(FNT(47)-FNT(45))/7
230 PRINT
240 PRINT"END OF ARRAYS OR
250 PRINT"START OF FREE RAM =";FNT(47)
260 IF FNT(44)<>FNT(46)THEN 280
270 PRINT"(NO ARRAYS EXIST)"
280 PRINT
290 PRINT"START OF STRINGS:";FNT(49)
300 PRINT"END OF MEMORY =";FNT(51)
310 PRINT"TOP OF MEMORY =";FNT(643)
320 PRINT
330 PRINT
340 PRINT"PRESS THE RUN/STOP KEY"
350 PRINT"TO END THE PROGRAM
360 GOTO360
```

TIMOUT

0285	645	Flag: Enable-disable serial IEEE T.O.

COLOR

0286	646	Current character color code. Normal power-up value: 14 (light blue). Poke with values from 0 to 15, the character color code, to see cursor and characters change color. Using this location is like using the color keys on the keyboard.

GDCOL

0287	647	Background color under cursor.

HIBASE

0288	648	Bottom of screen memory (page). This shows the normal location of the screen in "pages", that is 256 byte chunks. Normally set to 4. This is the top left corner of the screen. 4 x 256 = 1024 which is the beginning of the memory area for the screen.

You can use this information to create alternate screens and switch back and forth. This program is a sample of what two screens could look like.

```
100 REM DUAL SCREEN BY JIM BUTTERFIELD
105 REM TRANSLATED FOR 64 BY PAUL PAVELKO
110 POKE55,0:POKE56,32:CLR
120 DIML%(23)
130 GOSUB400:PRINTCHR$(147):GOSUB400
140 Z$=CHR$(133)
200 GETX$:IFX$=Z$THENGOSUB400
210 PRINTX$;:GOTO200
400 REM SWITCH
410 S=PEEK(648)
420 IFS=4THENS=32:T=128:GOTO500
430 IFS=32THENS=4:T=16:GOTO500
440 STOP:REM ERROR
500 POKE648,S:POKE53272,(PEEK(53272)AND15)
    ORT
510 FORJ=0TO23
520 V=PEEK(J+217):POKEJ+217,L%(J)
530 L%(J)=V
540 NEXTJ
550 PRINT:RETURN
```

C!

Enter the program and run it. The screen will momentarily fill with random characters. You are actually looking at a bit of memory! The screen will clear and return to the original screen. Type in "This is screen one ". Now press the f 1 key on the right side of the keyboard and the screen will blank. Type "This is screen two". Press f 1 and you're back to screen one. This is called "Page Flipping". It is used for such things as word proccessors and games.

XMAX

0289 649 The size of keyboard buffer, normally 10. POKEing this with values from 0 to 255 controls the number of characters the buffer can hold before it loses characters. This isn't like location 198. If you POKE a zero here, the buffer can't hold any letters. Nothing will print on the screen. Pressing RUN/STOP and RESTORE will get you back to normal.

POKE 649,0

will disable the keyboard. Use this if you only want someone to use the joystick or paddles for input.

returns the buffer to the standard size.

RPTFLG

028A 650 Flag: REPEAT key used. POKE 650,100: disables repeat of all keys. POKE 650,255: enables repeat of all keys. POKE 650,0: return to normal operation. Use this location to help write graphics programs. As soon as you turn the computer on, POKE 650,255 and drawing pictures becomes a lot easier when every key repeats.

KOUNT

028B 651 Repeat speed counter. Normal value: 4 POKEing with values from 0 to 255 varies the time before keys repeat. 0 and 255 will give the longest delay times.

DELAY

028C 652 Repeat delay counter.

SHFLAG

028D 653 Flag: Keyboard shift key, CTRL key, Commodore key (C=). This location shows if these keys are being held down.

Key Pressed	Value in 653
Shift	1
Commodore (C=)	2
CTRL (control)	4

If all 3 keys are pressed at the same time the value in 653 would be 7.

Since it would be unusual for all 3 keys to be down at the same time, you could use this as safety stop.

See how this location changes with,

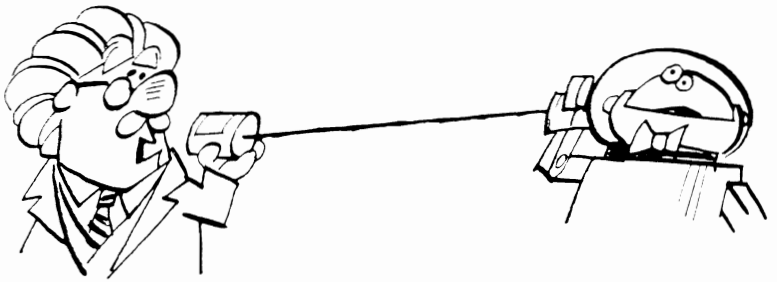
```
10 GET A$:PRINT PEEK(653): IF A$ = "" THEN 10
```

You can use the WAIT command to detect changes in these keys in the same fashion as in location 197. Try these,

1. WAIT 653,1,0 => wait for shift
2. WAIT 653,2,0 => wait for C= key
3. WAIT 653,4,0 => wait for CTRL key

```
200 PRINT"TO DESTROY THIS PROGRAM,"
210 PRINT"HOLD DOWN THE SHIFT, CONTROL,
220 PRINT"AND THE COMMODORE KEYS"
230 PRINT
240 PRINT"TO SAVE PROGRAM, DO NOTHING."
300 FOR PAUSE=1 TO 300
310 IF PEEK(653)=7 THEN 1000
320 NEXT
330 PRINT "PROGRAM SAFE"
340 END
1000 PRINT
1010 PRINT"PROGRAM DESTROYED"
1020 NEW
```

LSTSHF		
028E	654	Last keyboard shift pattern.
KEYLOG		
028F-0290	655-656	Vector: keyboard table setup. Normal value: 60232
MODE		
0291	657	Flag: 0 = disable shift key, 128 = enable. This doesn't turn the shift key off and on, but will show a 128 if CHR$(8) is entered and a 0 if CHR$(9) is entered.
AUTODN		
0292	658	Flag: Auto-scroll down 0 = ON.



RS-232 LOCATIONS 659-670

Locations 659-670 will be used if you write programs that need to communicate through the RS-232 port to other computers, printers etc. Most BASIC programmers won't use these locations. Skip them if you wish. If you do use these locations, remember the OPENing of an RS-232 channel automatically allocates 512 bytes of memory for two buffers which help prevent the loss of data when transmitting or receiving RS-232 information.

If there is not enough free space beyond the end of your program, part of your program will be destroyed to make room for the buffers. Be careful!

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-stantard BPS(time/2-100). 661 and 662 contain the baud rate for the start of the bit test during the interface activity. This is used to calculate baud rate.
RSSTAT 0297	663	RS-232: The RS-232 status register.
BITNUM 0298	664	RS-232: Number of bits left to send.

BAUDOF 0299-029A	665-666	RS-232: Baud rate of full bit time. Two bytes that are equal to the time of one bit cell. (Based on system clock/ baud rate)
RIDBE 029B	667	RS-232: Index to end of input buffer. The byte index to the end of the receiver FIFO buffer.
RIDBS 029C	668	RS-232: Start of input buffer (Page). The byte index to the end of start of the receiver FIFO buffer.
RODBS 029D	669	RS-232: Start of output buffer (Page). The byte index to the start of the transmitter FIFO buffer.
RODBE 029E	670	RS-232: Index to end of output buffer. The byte index to the end of the transmitter FIFO buffer.
IRQTMP 029F-02A0	671-672	This temporarily holds IRQ vector during INPUT or OUTPUT with a cassette. If maskable interrupt is generated during tape operation, this location holds the vector address until the operation is over before servicing the interrupt.
ENABL 02A1	673	RS-232 Enables.
02A2	674	TOD Sense During Cassette I/O.
02A3	675	Temporary storage for cassette read.
02A4	676	Temporary D1IRQ indicator for cassette read.
02A5	677	Temporary 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.

These address registers from 770 - 779 vector directly into the BASIC ROM memory and run these routines to handle important interpreter functions.

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. POKE 775,200 will prevent someone from LISTing your program after it has been run. To restore LIST POKE 775,167.
IGONE 0308-0309	776-777	Vector: BASIC character dispatch.
IEVAL 030A-030B	778-779	Vector: BASIC token evaluation.
SAREG 030C	780	Storage for 6502 .A register. A, Accumulator.
SXREG 030D	781	Storage for 6502 .X register.
SYREG 030E	782	Storage for 6502 .Y register.
STREG 030F	783	Storage for 6502 .SP register .SP, Stack pointer.

These locations shadow the 6502 internal registers. They are loaded with values prior to a SYS command for passing information on to machine language routines or are at the system ROM routines in order to achieve non-standard results. Here is an effective method for recording and updating the current cursor positions. This can be used as a 'PLOT' subroutine.

```
10 X=781 :Y=782 :P=783 :PLOT=65520
20 POKE P,1 :REM SET CARRY FLAG
30 SYS PLOT :REM KERNAL PLOT SUBROUTINE
40 PRINT"CURRENT CURSOR POSITION IS:"
50 PRINT PEEK(X);PEEK(Y)
```

```
10 X=781 :Y=782 :P=783 :PLOT=65520
20 INPUT"WRITE CURSOR POS. (X,Y)";XP,YP
30 POKE P,0 :REM CLEAR CARRY FLAG
40 POKE X,XP:POKE Y,YP
50 SYS PLOT :REM KERNAL PLOT SUBROUTINE
60 PRINT"HI"
```

```
100 A=780 :X=781 :Y=782
110 PLOT  =65520:REM KERNAL PLOT SUB.
120 CHROUT=65490:REM KERNAL CHROUT SUB.
130 PRINT"":POKE X,0
140 FOR YP = 0TO25
150 POKE P,0:REM CLEAR CARRY FLAG
160 POKE Y,YP
170 SYS PLOT
180 POKE A,YP+65:REM SEND OUTPUT CHAR.
190 SYS CHROUT
200 NEXT YP
```

```
10 A = 780
20 SC= 65439 :REM KERNAL SCNKEY ROUTINE
30 GE= 65508 :REM KERNAL GETIN  ROUTINE
40 SYS SC :SYS GE
50 IF PEEK(A)<>0THENPRINTCHR$(PEEK(A));
60 GO TO 20
```

The next three bytes store the information set by the USR(X) command in BASIC. Altering their contents may be useful for passing arguments in assembly routines.

USRPOK 0310	784	This byte stores the 6502 Op-code for the jump instruction (JMP). Normal value: 76 (4C-hex).
USRADD 0311-0312	785-786	This is the low and high byte of the USR(X) starting address.
0313	787	UNUSED
CINV 0314-0315	788-789	Vector: Hardware IRQ vector. Default starting address where IRQ (interrupt request) routines are serviced. Normal value: 59953.

The stop key is in a safe place and isn't usually pressed by mistake. But if you don't want people to accidentally stop your program while it's running, you can turn off, (it's called "disable"), the RUN/ STOP key this way:

POKE 788,52

This also stops the clock (TI$ and TI) so don't use this POKE if you need the time functions in your program. POKE 788,49 turns on the clock and will enable the RUN/STOP key. See location 808 for more information.

CBINV 0316-0317	790-791	Vector: BRK instruction interrupt.
NMINV 0318-0319	792-793	Vector: non-maskable interrupt. Default starting address where the

NMI routine is to be service. Normal value 65095.

This is the jump table to the starting address of the above Kernal routines within the 8K of Kernal. See in the appendix "Addressing the Kernal" for a more detailed explanation of the Kernal operating system. You can intercept these vectors for home made machine language subroutines.

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.

POKE 808,239

will disable the RUN/STOP key *without* stopping the clock.

POKE 808,237

will enable the RUN/STOP key.

POKE 808,225

will disable the RUN/STOP and RESTORE keys. Once your program starts, there isn't any way to break out unless the program ends or this line,

POKE 808,237

is run as part of the program.

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. How this useful address is used is completely up to you! Here is an automatic line numbering routine using a defined vector.

```
63993 INPUT"(CLR)STARTING LINE #";A:INPUT"
      INCREMENT";B:POKE815,B:PRINT"(CLR)"
63994 B=A/256:POKE784,(B-INT(B))*256:POKE
      814,B:PRINTA;
63995 GETA$:PRINTA$;:IFA$<>CHR$(13)THEN63995
63996 PRINT"GO63998":FORA=631TO634:POKEA,
      145:NEXT
63997 POKEA,13:POKE636,13:POKE198,6:END
63998 PRINT"(UP)(UP)":FORA=1TO3:PRINT
      "          ":NEXT:PRINT"(UP)(UP)(UP)";
63999 A=PEEK(784)+PEEK(814)*256+PEEK(815)
      :GOTO63994
```

When you run this, the words "starting line#" appear on the screen. Enter the number you want for the first line number of your program. 1000 would be a good start. Press return. Next, "increment" means how many numbers between each line. 10 would be good here. Press return and your program lines will be automatically numbered.

ILOAD 0330-0331	816-817	Kernal LOAD routine vector. Using the SYS command and this location's address starts LOAD operation on a serial device. It automatically LOADs the first program.
ISAVE 0332-0333	818-819	Kernal SAVE routine vector. Same as above location except it works with the SAVE command.
0334-033B	820-827	Unused.
TBUFFR 033C-03FB	828-1019	Tape I/O buffer. This is the first place your cassette programs go when they are being loaded.

It is also storage area #13, 14 and 15 for sprite data. If you have a sprite on the screen while loading another program from tape, expect your sprite to look strange. This happens because the incoming data from the tape changes the data for the sprite. The sprite must be redefined. If you use a cassette to save programs, sprite areas #192 - 198 would be a better place to put them. They will not be changed by using the cassette. For information on storing sprites, see the appendix "Making a Sprite".

03FC-03FF	1020-1023	Unused.

SCREEN DATA AREA

VICSCN 0400-07E7	1024-2023	This is where data must be put to be seen on the screen. The computer does this automatically. You can do this, too by using the POKE command. POKE 1024,81 will put a heart in the upper left corner of the screen. POKE 55296,7 will make the heart turn yellow. Each screen location from 1024 to 2023 has a matching color location from 55296 to 56295. To make things you POKE onto the screen any color you want and you must also POKE the color locations.

The screen is made up of 1000 locations, 25 rows of 40 columns. To print a specific row and column, use this formula.

POSITION = 1024+COLUMN NUMBER+40*ROW NUMBER

Here's a sample program that will print a cyan heart in the 4th row, 15th column.

```
100 PRINTCHR$(147)
110 REM--1024 IS THE
120 REM--BEGINNING OF
130 REM--SCREEN AREA.
140 S=1024
150 REM--55296 IS THE
160 REM--START OF SCREEN
170 REM--COLOR DATA.
180 SC=55296
190 REM--PUT A CYAN HEART
200 REM--IN THE 4TH ROW
210 REM--AND 15TH COLUMN.
220 POKESC+15+40*4,3
230 POKES+15+40*4,83
```

On the early 64's, any character POKEd to the screen would automatically be white. It's been reported that this has been changed to blue an newer machines. That means the POKEd characters would be the same color as the backround. To make POKEs to the screen visible POKE the color location too. Use:

COLOR POSITION = 55296+COLUMN NUMBER+40*ROW NUMBER

07F8-07FF 2040-2047 Sprite data pointers.

The sprite pointers tell the sprite-making part of the computer where to go to get the information to make a sprite.

For example:

POKE 2040,13

This tells the computer that the data for sprite 0 can be found in the 13th block.

POKE 2043,192

This means data for sprite 3 is in the 192nd block. See the appendix "How to Create a Sprite" for more information.

0800-9FFF	2048-40959	Your BASIC programs are stored here. You can make the space smaller by using locations 43 - 46.

This handy renumber routine checks the area inside this memory to see where the line numbers are stored. Then it goes through and changes all of them to make your program look neater.

Beware though, it doesn't change the line numbers after a GOTO or GOSUB command. You will have to go back and do it by hand.

To renumber a program you have in memory, use the append routine (locations 43-46) or follow these instructions.

1) Type in POKE43,PEEK(45)-2:POKE44,PEEK(46):CLR

2) Load in the renumber routine

3) POKE43,1:POKE44,8

4) Type in RUN 10000 and press RETURN

```
9990 END
10000 REM RENUMBER
10010 Y6=2048:Y7=10
10020 IFPEEK(Y6+3)=6ANDPEEK(Y6+4)=39THENEND
10030 Y8=INT(Y7/256):Y9=Y7-256*Y8
      :POKEY6+3,Y9:POKEY6+4,Y8
10040 IFPEEK(Y6+5)<>0THENY6=Y6+1:GOTO10040
10050 Y7=Y7+10:Y6=Y6+5:GOTO10020
```

The next program is a "cruncher", that is, it combines many program lines into as few lines as possible. This could mean your program would run a bit faster and take up less room in memory. It will also make your program less readable. If you "crunch" a program, always save an expanded version, too. It's much easier to find errors and make corrections on an "un-crunched" program.

```
1 INPUT"COMBINE LINES FROM, TO";L,U:C=2049
  :B=256:E=PEEK(45)+B*PEEK(46)-4
2 LT=PEEK(C+2)+B*PEEK(C+3):PRINTLT;
3 IFLT<LTHENC=PEEK(C)+B*PEEK(C+1):GOTO2
4 IFLT>LTHENPRINT"LINE NOT FOUND":END
5 LINK=C:C=C+4
6 Q=PEEK(C):IFQ<>0THENC=C+1:GOTO6
7 IFPEEK(C+1)+PEEK(C+2)=0THENEND
8 LT=PEEK(C+3)+B*PEEK(C+4):PRINTLT;
9 IFLT>UTHENC=C+1:POKELINK,C-INT(C/8)
  *8:POKELINK+1,C/8:GOTO5
10 POKEC,ASC(":"):FORJ=C+1TOE:Q=PEEK(J+4)
11 POKEJ,Q:NEXT:E=E-4:GOTO6
```

C!

To use this program, first load it in then use the method explained above to hide it. Next, load in the program you want to "crunch". Then POKE43,1:POKE44,8 will allow you to see and use the crunch program.

To save yourself time and trouble use the append routine.

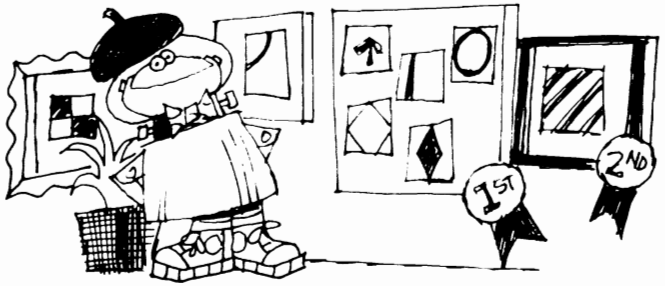
8000-9FFF	32768-40959	VSP (Video/Sound Package) cartridge ROM => 8192 bytes.

The "Video/Sound Package" is a cartridge Commodore plans to release that will make programming of sound and sprites easier. The penalty is a loss of 8192 bytes of memory.

A000-BFFF	40960-49151	BASIC ROM => 8192 bytes (or 8K RAM).

This is where BASIC sits. By using location 1, it is possible to have the computer temporarily pretend BASIC isn't around any more. Most of us will never do something like that, but machine language programs might, in order to have more room for longer programs.

So, in a way, we've come full circle from location 0 to 49151 which points us back to the beginning again. Our next section embarks on an exploration of new territory - the powerful regions of Graphics and Sound.



GRAPHICS

If you're not familiar with bytes and bits, please read the very helpful section on page 10 to 13. Sound and Graphics on the 64 require some knowledge of bytes in order to control the power of the machine.

Hexadecimal Decimal Loc.	Bits	Description
D000-D00F		
53248		Sprite 0 X position
53249		Sprite 0 Y position
53250		Sprite 1 X position
53251		Sprite 1 Y position
53252		Sprite 2 X position
53253		Sprite 2 Y position
53254		Sprite 3 X position
53255		Sprite 3 Y position
53256		Sprite 4 X position
53257		Sprite 4 Y position
53258		Sprite 5 X position
53259		Sprite 5 Y position
53260		Sprite 6 X position
53261		Sprite 6 Y position
53262		Sprite 7 X position
53263		Sprite 7 Y position

These are magic locations! By changing them you can move up to eight sprites across the screen with a FOR/NEXT loop.

Look at the description for sprite #0. The term "X position" means the movement from left to right (or right to left) on the screen. "Y position" is movement up and down.

Here's a way to make programming easier. By calling the first address by a variable name (V) for example, all the other graphic addresses can be reached by adding an appropriate number. That makes it easy to remember. For example, to move sprite #7 you could use V+14 and V+15 instead of 53262 and 53263.

This program creates a solid sprite. By setting V equal to 53248 in line 140, all the other references to the graphics can be easily written. This method of using variable names (like V or S or A1) for frequently used numbers, will also speed up your program.

```
100 REM--CLEAR THE SCREEN.
110 PRINT CHR$(147)
120 REM--'V' EQUALS THE START OF
130 REM--THE GRAPHICS AREA.
140 V=53248
150 REM--GET SPRITE DATA FROM AREA 192.
160 POKE2040,192
170 REM--CREATE A SOLID SPRITE AND
180 REM--PUT SPRITE DATA IN AREA 192.
190 REM--(64*192 = 12288)
200 FORI=0TO62:POKE12288+I,255:NEXT
210 REM--TURN ON SPRITE #0.
220 POKEV+21,1
230 REM--SPRITE HALFWAY ACROSS SCREEN.
240 POKEV,160
250 REM--SPRITE HALFWAY DOWN SCREEN.
260 POKEV+1,150
```

The position of sprite is calculated from the top left corner of the 24 x 21 dot area that make up the unexpanded sprite. Even if only one dot is used as a sprite, and you happen to want it in the middle of the screen, you must still calculate the positioning by starting at the top left corner location.

D010
53264 (V+16) Sprites 0 to 7 X position past position 256

Positioning in the horizontal direction is a little complicated. Even though you have a 320 pixel wide screen for your sprite, if it moves past the 256th pixel from the left (about eight-tenths of the way across the screen) you need a ninth bit to describe the sprite's position, because a byte can only hold a number up to 255.

When a sprite crosses the 256 dot "seam", the proper bit in this location must be turned on. When crossing back, the bit must be turned off.

For example when sprite 0 moves across the seam to the right:

```
POKE V+16, PEEK (V+16) OR 1
```

If sprite 4 crosses then

```
POKE V+16, PEEK (V+16) OR 16
```

To cross sprite 0 back to the LEFT

```
POKE V+16, PEEK (V+16) AND 254
```

Another thing to consider when positioning sprites is that they are visible within certain limits even if you want to move them off screen. These border parameters vary with dimensions of your display and change when your sprites are expanded or unexpanded.

The best way to find the limits is to create a solid sprite and move it back and forth. See the appendix "How To Create A Sprite".

D011
53265 (V+17) VIC control register.

7	register compare: (8 bit) see 53266
6	extended color text mode: 1 = enable
5	bit-map mode: 1 = enable
4	blank screen to border color: 1 = blank
3	select 24/25 row text display: 1 = 25 rows
2-0	smooth scroll to Y dot-position (0-8)

Most beginning programmers can skip this location. This description will interest intermediate or advanced programmers. Think of this register as the main control box for your graphics where each bit acts as an on/off switch for the graphics modes.

BIT 6 - EXTENDED COLOR MODE

In this mode, each character takes on a background color in addition to the color of the character itself. Your screen color may be different, so for example, you could have a white character with a green background on a yellow screen. There is an important limitation in this mode. Since memory out of your character code is used to determine this extra background color, you are limited to the first 64 characters. See location 53270 for information on extended color modes.

BIT 5 - BIT MAP MODE

If you take a magnifying glass to your screen you'll see the countless phosphor dots that comprise the pixels used to fill the screen with color. This is the smallest increment of resolution that is available for graphics, the bit map mode. The standard bit map mode gives you a 320 horizontal dot by 200 vertical dot resolution, with a choice of two colors in each 8 by 8 bit character.

ENABLE HIGH-RESOLUTION BIT MAP MODE:
POKE 53265, PEEK(53265) OR 32

DISABLE HIGH-RESOLUTION BIT MAP MODE:
POKE 53265, PEEK(53265) AND 233

This example of bit mapping is a spiral drawn against a plain background. First the screen will go black, then slowly, the screen will clear. Then a cyan background is produced and the spiral is plotted. When it's finished a black square appears in the top left corner. Press RUN/STOP and RESTORE to return to a normal screen.

```
1000 PRINT CHR$(147)
1010 PRINT
1020 PRINT
1030 PRINT
1040 PRINT"WHEN SPIRAL CURVE IS COMPLETE, A"
1050 PRINT"BLACK BOX WILL APPEAR IN THE"
1060 PRINT"UPPER LEFT CORNER."
1070 PRINT
1080 PRINT"PRESS RUN/STOP AND RESTORE TO"
1090 PRINT"RETURN TO NORMAL SCREEN"
1100 POKE 198,0 :REM CURSOR BLACK
1110 PRINT
1120 PRINT
1130 PRINT"PRESS ANY KEY TO BEGIN"
1140 GET A$: IF A$="" THEN 1140
1150 PRINT CHR$(147)
1160 REM-->PUT BIT MAP AT 8192
1170 BASE=2*4096:POKE53272,PEEK(53272)OR8
1180 REM-->ENTER BIT MAP MODE
1190 POKE 53265,PEEK(53265)OR32
1200 REM-->CLEAR BIT MAP
1210 FORI=BASE TO BASE+7999:POKEI,0:NEXT
1220 REM-->SET BACK. AND BORDER COLORS
1230 POKE 53280,3:REM BORDER TO CYAN
1240 FOR I=1024 TO 2023:POKEI,1:NEXT
1250 REM-->CURVE WILL FILL THE SCREEN
1260 FOR S = 0TO17 STE/30
1270 X=150+INT(150*EXP(-S/20)*SIN(S/4))
1280 Y=90-INT(120*EXP(-S/10)*COS(S/2))
1290 CH = INT(X/8)
1300 RO = INT(Y/8)
1310 LN = Y AND 7
1320 BY = BASE+RO*320+8*CH+LN
1330 BI = 7-(X AND 7)
1340 POKE BY,PEEK(BY)OR(2^BI)
1350 NEXT S
1360 POKE 1024,16
1370 GOTO 1370
```

Here's a formula to turn on and off individual pixels on the screen. Keep in mind that the screen dimensions are 25 rows by 40 columns. That means 1000 (25X40) characters can be printed on the screen. Lets call each one of these places a CHARACTER POSITION. Then this formula is used to plot which pixel to turn on.

The character position number is found this way:

CHARACTER POSITION = INT(X/8)*8+INT(Y/8)

X and Y are the horizontal and vertical positions of the pixel you want to turn on. X will be number between 0 and 320. Y will be a number between 0 and 190.

The row is found by:

ROW = (Y/8 − INT(Y/8))*8

Therefore, the byte in which character memory dot (X,Y) is located is calculated by,

BYTE = 1023 + CHAR*8 +ROW

The bit to be modified is..

BIT = 7 − (X−(INT(X/8)*8)

Finally, to turn on any bit...

POKE BYTE,PEEK(BYTE)OR(2↑BIT)

BIT 4 - SCREEN BLANKING

You can hide anything printed on the screen by:

POKE 53265, PEEK(53265) AND 239

The screen will turn the same color as the border.

POKE 53265, PEEK(53265) OR 16

will make the screen visible again.

With this location you could turn off the screen then have messages or graphics printed on the screen. Then "pop" the screen into view all at once.

This program will:

1) Blank the screen

2) Print a message
3) Turn the screen back on

```
10 PRINTCHR$(147)
20 PRINT"HERE I AM!"
30 PRINT"NOW I'LL GO AWAY"
40 FOR PAUSE = 1 TO 1000: NEXT
50 POKE53265,PEEK(53265)AND 239
60 PRINT
70 PRINT"I'M BACK!"
80 FOR PAUSE = 1 TO 1000: NEXT
90 POKE53265,PEEK(53265)OR 16
```

BIT 2-0 - SCROLLING

You can scroll screen information by moving the screen display in any of four directions, moving as slow as 1 pixel at a time or as fast as 8 pixels.

When you do this, shrink your screen size from 40 to 38 columns wide or from 25 to 24 rows tall depending on the direction you want to scroll. This gives the computer a place to assemble the information before it scrolls onto the screen.

To do truly fine scrolling you'll need a machine language routine. Without that routine, scrolling can only be done with text or keyboard characters.

```
1000 REM ********************************
1010 REM *     PROTO SCROLL ROUTINE     *
1020 REM ********************************
1030 :
1040 DIM C(8,8):REM CHARACTER MATRIX
1050 :
1060 REM >> SHRINK THE SCREEN          <<
1070 POKE 53265,PEEK(53265) AND 247
1080 :
1090 REM >> STORE CHARACTER INFO.      <<
1100 FOR I = 1TO8
1110 FOR J = 1TO8
1120 READ C(I,J)
1130 NEXT J,I
1140 :
```

```
1150 REM >> CLEAR SCREEN/WHITE CURSOR
1160 PRINT CHR$(147);CHR$(5)
1170 :
1180 REM >> MOVE CURSOR TO BOTTOM
1190 FOR X = 1TO24:PRINTCHR$(17);:NEXT
1200 :
1210 REM >> POSITION FOR 1ST SCROLL <
1220 POKE53265,(PEEK(53265)AND248)+7
1230 :
1240 REM >> PRINT CHARACTER MATRIX <<
1250 FOR K = 1TO8
1260 FOR L = 1TO8
1270 IF C(K,L)<0 THEN PRINTCHR$(18);
1275 PRINTCHR$(ABS(C(K,L)));
1280 PRINT CHR$(146);
1290 NEXT L
1300 PRINT
1310 NEXT K
1320 PRINT
1330 :
1340 REM >> PREFORM AN 8-BIT SCROLL <
1350 FOR P = 0TO7
1360 POKE 53265,(PEEK(53265)AND248)+P
1370 :
1380 REM >> PAUSE A MOMENT          <<
1390 FOR W = 0TO50:NEXT W
1400 NEXT P
1410 :
1420 REM >> RETURN TO REPEAT        <<
1430 GO TO 1220
1440 :
1450 REM >>>>>  CHARACTER DATA   <<<<<
1460 DATA 32,32,32,-32,-32,-32,32,32
1470 DATA 32,32,-32,-32,32,-32,-32,32
1480 DATA 32,-32,-32,-32,-32,-32,-32,-32
1490 DATA 32,32,-32,-32,-32,-32,-32,32
1500 DATA 32,32,-32,32,32,32,-32,32
1510 DATA 32,32,-32,32,32,32,-32,32
1520 DATA 32,32,-32,32,32,32,-32,32
1530 DATA 32,-32,32,-32,32,-32,32,-32
```

BIT 3 - 24/25 ROW SELECT

By POKEing this bit off you will make the screen 24 rows from top to bottom, instead of the normal 25. This is used in the fine scrolling mode. Half a row is taken from both the top and bottom of the screen. To see this work

POKE 53265, PEEK (53265) AND 247

To get 24 rows

POKE 53265, PEEK (53265) OR 8

will return you to 25 rows.

D012
53266 (V+18) — Read/write raster value for compare. Lower 8 bits out of 9 bits (see location 53265).

The raster register is a dual purpose register. When you read this register it returns the lower eight bits of the current raster position. The raster position of the most significant bit is in location 53265. You use the raster register to set up timing changes in your display for vertical and horizontal band scrolling, mix-moded display (hi-resolution with characters) and other kinds of interrupts.

The changes of your screen should be made when the raster is not in the visible postion of the display area. The visible positions are those between 51 and 251.

D013
53267 (V+19) — Light-pen latch X position.
D014
53268 (V+20) — Light-pen latch Y position.

These locations return the light-pen X, Y positions across the screen from 0 to 255. Interrupts can be generated when the pen is triggered.

D015
53269 (V+21) — Sprite display

When you want to make a sprite visible, this is the place!

Each bit handles a sprite. Sprite 0 is controlled by bit 0, sprite #1 by bit #1 and so forth.

Here's how to make sprite #4 show on your screen:

```
POKE 53269, PEEK (53269) OR 16
```

```
POKE 53269, PEEK (53269) AND (255-16)
```

will make sprite #4, and only sprite 4, disappear.

```
POKE 53269, PEEK (53269) OR 170
```

will put sprites #1, 3, 5, and 7 on the screen.

```
POKE 53269, PEEK (53269) AND (255-170)
```

will take them off.

D016

53270 (V+22)		VIC control register
	7-6	unused
	5	unused
	4	multi-color mode: 1=enable (text/map)
	3	select 39/40 column text: 1=40 cols.
	2-0	smooth scroll to X position

Here is another place that is used mainly by advanced graphics users. It is an interesting location, but if you program in BASIC only, just take a look at the changes you can make in the scrolling Proto. (Location 53265)

BIT 4 - MULTICOLOR MODE

The multi-color mode for text is turned on by:

```
POKE 53270, PEEK (53270) OR 16
```

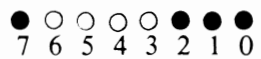
Turn it off with

```
POKE 53270, PEEK (53270) AND 239
```

Normally, each of the 1000 screen positions or "blocks" can have only two colors, a background color and a text color. For example, when the computer is turned on the background of each block is dark blue and the text color is light blue.

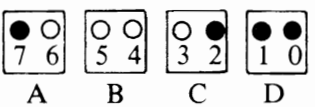
POKEing this location gives you four colors to play with: screen color (loc. 53281), and background colors 1 to 3 (loc. 53282-53284). But there is a penalty, multi-color mode will work for only the first 64 characters in the character set, and it needs two bits to describe any of the colors.

Like this:



Each letter printed on the screen is really an 8x8 block of pixels with some turned on and some off. The ones turned on are the letter you see. In the sample row of pixels above, #1, 4, 6 and 7 are on.

In the multi-color mode they are in pairs.



The pattern "A" tells the computer to get the color from the background #2 (loc. 53283). B gets its information from background #0 (loc. 53281) which is the screen color). C indicates the color comes from background #1 (loc. 53282) and D gets its information from the color memory area from 55296 to 56295 - different for each screen location.

Because of this "two-bit" reading of the data, your resolution will be less than normal but you have more color available.

MULTICOLOR BIT MAP MODE

This uses two locations (53270 and 53265). The same principles of "BIT PAIRS" are used here as in the mode described above.

Enable this with:

POKE 53270, PEEK (53270) OR 16

POKE 53265, PEEK (53265) OR 32

Remember, horizontal resolution is decreased.

BIT 3 - SELECT 38 OR 40 COLUMN MODE

1 = 40 COLUMN

POKE this bit to zero to narrow the screen area. That gives you space on each side of the screen to assemble the characters before you scroll. Doing this will keep the scrolling smooth.

BIT 2 - FINE SCROLLING

By adding

1365 POKE 53270, (PEEK (53270) and 248) + P

to the program in location 53265, you can make Proto scroll diagonally. Again, to do truly fine scrolling you need a machine language routine. The example under location 53265, written in BASIC, allows you to scroll text and keyboard graphic characters only.

D017
53271 (V+23) — Sprites 0 to 7 expand sprite 2 times (vertically)

This location will make your sprites twice as large vertically. This has a one-to-one correspondence, that is, turning on bit #0 expands sprite #0.

POKE 53271, PEEK(53271) OR 1

To return sprite # to normal size:

POKE 53271, PEEK(53271) AND (255-1)

To expand sprites #7 and #1:

POKE 53271, PEEK (53271) OR 130

Also see location 53277.

D018

53272 (V+24)		VIC memory control register
	7-4	video matrix base address (inside VIC)
	3-1	character dot-data base address (in VIC)

This location is responsible for the location of both screen memory and character memory.

POKE 53272,23

will convert the screen display to lower case.

POKE 53272,21

will change the screen display back to upper case.

Another example of the use of this address is seen in location 648, screen flipping.

SCREEN MEMORY LOCATION

Screen memory location is controlled by the last four bits (most significant nybble) of 53272. To move the screen, use the following:

POKE 53272, (PEEK(53272) AND 15) OR A

A must be one of the decimal values on this chart.

A	Location Decimal	Hexadecimal
0	0	$0000
16	1024	$0400
32	2048	$0800
48	3072	$0C00
64	4096	$1000
80	5120	$1400
96	6144	$1800
112	7168	$1C00
128	8192	$2000
144	9216	$2400
160	10240	$2800
176	11264	$2C00
192	12288	$3000
208	13312	$3400
224	14336	$3800
240	15360	$3C00

Normally, A's value is 16, and the screen begins at 1024.

To create your own character set you modify bits 3 to 1.

POKE 53272, (PEEK(53272) AND 240) OR B

where B is the decimal value from this chart.

VALUE of B	DEC	HEX
0	0	$0000-$07FF
2	2048	$0800-$0FFF
4	4096	$1000 -$17FF
6	6144	$1800-$1FFF
8	8192	$2000-$27FF
10	10240	$2800-$2FFF
12	12288	$3000-$37FF
14	14336	$3800-$3FFF

B is normally set to 4.

See the appendix "Being an Artist on the Commodore 64" for an example of creating your own character set.

D019

53273 (V+25)		VIC interrupt flag register (bit=1: IRQ)
	7	set on any enable VIC IRQ condition
	3	light-pen triggered IRQ flag
	2	sprite vs sprite collision IRQ flag
	1	sprite vs background collision IRQ fl
	0	raster compare IRQ flag

This location is best handled with machine language since the values here can change rapidly.

D01A

53274 (V+26)	IRQ mask register: 1=interrupt enabled.

This location is set the same way as the location above. Unless the corresponding bit in the interrupt enable register is set to a 1, no interrupt from that source will take place. Practice your machine code before using this!

D01B

53275 (V+27)	Sprite vs background display priority: 1 = sprite.

With this location you can make a sprite pass in front or behind of printed characters.

POKE 53275,2 PEEK(53275) OR 2

means sprite #1 will pass behind text or graphics on the screen.

POKE 53275,7 PEEK (53275) OR 7

will do the same for sprites #0, 1 and 2.

POKE 53275, PEEK (53275) AND (255-7)

will put sprites #0, 1 and 2 in front of the text.

D01C
53276 (V+28) — Sprites 0-7 multi-color mode select: 1 = M.C.M.

You can create a sprite with up to 3 colors by POKEing on this location.

```
POKE 53276,1
```

puts sprite #0 in the multicolor mode. The three color choices come from the sprite color locations (53287 to 53294, depending on the sprite number you're working with), and the sprite multicolor locations, 53285 and 53286.

See the appendix "How to Create a Sprite" for more information.

D01D
53277 (V+29) — Sprites 0-7 expand sprite 2X (horizontally).

Like 53271, this is the place used to expand a sprite, horizontally.

```
POKE 53277, PEEK (53277) OR 1
```

will expand sprite 0, making it twice as wide.

```
POKE 53277, PEEK (53277) OR 18
```

will expand sprites 4 and 1. Also see location 53271.

D01E
53278 (V+30) — Sprite vs sprite collision detect.

If a sprite is touching another sprite, the bit for that sprite is turned on.

```
IF PEEK (V+30) = 1 THEN ACTION
```

is a typical use of this location. It asks if sprite 0 has bumped into another sprite.

Once you PEEK here, this register resets itself to zero. It's a good idea to save the value here by putting it in a variable.

```
If PEEK (V+30) = 1 THEN B = 1
```

D01F
53279 (V+31) Sprite vs background collision detect.

This works just like the previous location, checking to see if a sprite has bumped into some text or graphics. This register also resets to zero after being read so if you need to save the value, use a variable.

If you PEEK (V+31) and read 128 that means sprite #7 has bumped into some characters.

D020
53280 (V+32) Border color. Normally set to 14

Poking the numbers from 0 to 15 here will change the border color. 
This chart shows the number to POKE to get each color.

0	BLACK	8	ORANGE
1	WHITE	9	BROWN
2	RED	10	Light RED
3	CYAN	11	GREY 1
4	PURPLE	12	GREY 2
5	GREEN	13	Light GREEN
6	BLUE	14	Light BLUE
7	YELLOW	15	GREY 3

These numbers are used *any* time you work with colors.

53281 (V+33) Background color 0.

True, this is called a background color, but it is also the screen color. This location shows a 6 when the computer is turned on or RUN/STOP and RESTORE are pressed.

This program will show you all the combinations of screen and border colors.

```
100 PRINTCHR$(147)
110 FOR A= 0 TO 15
120 FOR B= 0 TO 15
130 POKE53280,A
140 POKE53281,B
150 C=B
160 IFC=15THENC=0
170 POKE646,C+1
180 PRINTCHR$(19)
190 FOR D = 1 TO 12:PRINTCHR$(17);:NEXT D
200 PRINT"HELLO!"
210 FOR E =1 TO 300: NEXT E
220 NEXTB
230 NEXTA
```

53282 (V+34)	Background color 1.
53283 (V+35)	Background color 2.
53284 (V+36)	Background color 3.

These are the color registers used with the multi-color modes in locations 53265 and 53270. POKE the numbers on the colors you want to use. See location 53270.

D025-D026

53285 (V+37)	Sprite multi-color register 0.
53286 (V+38)	Sprite multi-color register 1.

When the sprite multi-color mode (loc. 53276) has been selected the colors in these registers are used in addition to the usual sprite color (loc. 53287-53294). See the appendix "How to Create a Sprite" for more information.

D027-D02E

53287 (V+39)	Sprite 0 color.
53288 (V+40)	Sprite 1 color.
53289 (V+41)	Sprite 2 color.
53290 (V+42)	Sprite 3 color.
53291 (V+43)	Sprite 4 color.

53292 (V+44) Sprite 5 color.

53293 (V+45) Sprite 6 color.

53294 (V+46) Sprite 7 color.

These are the color registers for each sprite.

Each sprite has a "default" color, that is, if you don't POKE a color for your sprite, the 64 will automatically give it a color.

Sprite	Default Colors
0	White
1	Red
2	Cyan
3	Purple
4	Green
5	Blue
6	Yellow
7	Grey 2

SOUND

If you are new to sound programming, get ready to be amazed! The Commodore 64 has sound capabilities unheard in other computers.

You need to use the BASIC commands PEEK and POKE to produce sound, so if you're not sure what these commands do, just re-read the sections "How to PEEK and POKE" and "BYTES and BITS" on pages 6 to 13.

By calling the first address by a variable name (S), all other sound addresses can be reached by adding an appropriate number.

```
10 LET S = 54272
```

It is also a good idea to clear the voices by POKEing all the sound locations to zero. That keeps unwanted settings out of your program.

```
100 S=54272
200 FOR I = 0 TO 28
300 POKE S+I,0
400 NEXT I
```

VOICE #1 REGISTERS

D400 54272 (S=54272)	Voice 1: frequency control, low-byte.
D401 54273 (S+1)	Voice 1: frequency control, high-byte.

The values to POKE here to produce musical notes are in the note table on page 175.

An example of values to POKE:

```
100 POKE 54272,75
200 POKE 54273,34
```

Produces a C note in the 5th octave. Not every sound you want to make will be a musical note. To produce a frequency not on the musical note table, use this program. It will give the numbers to POKE in the high and low bytes.

```
10 PRINTCHR$(147)
20 INPUT"WHAT FREQUENCY DO YOU WANT";F1
30 F1=ABS(F1)
40 REM--COMPUTE HIGH BYTE
50 F2=INT(F1/256)
60 REM--COMPUTE LOW BYTE
70 F3=F1-(F2*256)
80 PRINT"POKE LOW BYTE -----";F3
90 PRINT"POKE HIGH BYTE -----";F2
```

D402

54274 (S+2)		Voice 1: pulse waveform width, low-byte

D403

54275 (S+3)		Voice 1: pulse waveform width.
	7-4	unused
	3-0	high-nybble

Another quality that is given to any frequency of the voice is how distinct sound peaks are. This is how you produce tonal textures such as vibrato (the vibrating aspect of sound that gives it a singing quality). This is done with pulse waveform. The different types of waveforms are discussed in the next register.

In location 54274, POKE a number in the range 0-255.

In location 54275, POKE a number between 0-15.

See the "airplane" program in the appendix for an example of pulse waveform manipulation.

D404
54276 (S+4)

	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

This is the place! Here you can decide what kind of note you want to create, from the muted sound of a violin to the crash of ocean surf. It all depends on the type of waveform you choose.

Take a look at all the bits in this location. Bits 0, 4, 5, 6 and 7 will be used often by everyone. Bits 1, 2 and 3 are used for advanced sound techniques.

Bit 0: This is the ON/OFF switch of voice one, but it should be used in union with one of the waveform bits. When this bit is set to 1, the attack, decay and sustain cycle begins. When it is poked with a 0, the release of the note starts.

```
POKE 54276,17
```

will begin the A/D/S cycle for the triangle waveform.

```
POKE 54276,16
```

will start the Release cycle.

The waveforms are designed to be used one at a time. You should try to add two or more together to experiment.

Bit 1: This bit, when turned on, will allow the interaction of voice 1 and voice 3, blending both waveforms. To hear this, the frequency of voice three should be lower than voice 1.

Bit 2: The ring modulation effect creates the sounds of bells or gongs. It is also used with voice 3. In order to hear the ring modulation, voice 1 must use the triangle waveform and voice 3 must be set to a frequency higher than zero.

Bit 3: This bit, if turned on, will reset voice 1 until it is POKEd back to zero.

Bit 4: The triangle waveform is smooth and flute-like. The sound is made by POKEing this bit and the gate bit on.

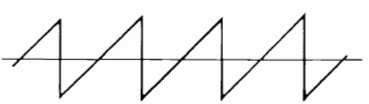
ON: POKE 54276,17

OFF: 54276,16

Bit 5: Listen to the brass sound of the sawtooth waveform.

ON: POKE 54376,33

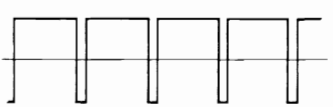
OFF: POKE 54276,36



Bit 6: The pulse waveform can be changed by using location 54275 to vary the width of the pulse. You can create sounds from a piano to a clarinet.

ON: POKE 54276,65

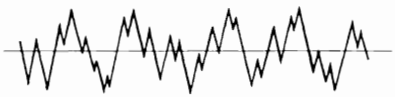
OFF: POKE 54276,64



Bit 7: The noise bit. Use this to make rocket blasts to surf sounds.

ON: POKE 54276,129

OFF: POKE 54276,128



D405

54277 (S+5)		Envelope generator 1: att/dec cycles.
	7-4	select attack cycle duration: 0-15
	3-0	select decay cycle duration: 0-15

The attack cycle is one of 4 parts that make up any note played by an instrument. In the attack, the volume of the note rises to its highest volume. Then it begins to 'decay' or fall in volume to lower level. Decay is the second part of a note.

Some instruments, like a trumpet, have a fast attack. Others, like the violin, have a very slow attack!

Here's an example of how to set both attack and decay.

```
100 REM - A IS THE ATTACK RATE.
110 REM - IT CAN BE ANY NUMBER FROM
120 REM - 0 TO 15.
130 A=9
140 REM - D IS THE DECAY RATE.
150 REM - IT CAN BE ANY NUMBER FROM
160 REM - 0 TO 15, TOO.
170 D=5
180 REM - POKE THE ATTACK/DECAY
190 POKE 54277,PEEK(54277)AND(A*16)
200 REM - ADD THE DECAY NEXT
210 POKE 54272,PEEK(54272) + D
```

This sets a medium attack and decay.

D406

54278 (S+6)	Envelope generator 1: sus/rel cycles.

7-4	select sustain cycle duration: 0-15
3-0	select release cycle duration: 0-15

Sustain and release are the other parts of a sound. When a sound decays, the volume falls to the sustain level. The volume stays at this level until the release cycle begins.

Use bits 7-4 to set the sustain level from 0 to 15, zero would be the lowest sustain, 15 is the highest sustain.

The release cycle starts the fall in volume from the sustain level to zero. The release cycle follows an exponential curve that mimics the way instruments that are blown or bowed actually respond.

Here's a sample program to produce a single tone.

```
100 REM - SOUND CHIP STARTING ADDRESS.
110 SID=54272
120 REM - CLEAR THE REGISTERS.
130 FOR I = 0 TO 28
140 POKE SID+I,0
150 NEXT
160 REM - SET THE HIGHEST VOLUME.
170 POKESID+24,15
180 REM - SET HIGH BYTE AND LOW BYTE
190 REM - TO PRODUCE MIDDLE C IN VOICE
200 REM - ONE.
210 POKESID,34
220 POKESID+1,75
230 REM - SET THE ATTACK/DECAY RATE.
240 REM > A = ATTACK; D = DECAY
250 A=0
260 D=15
270 POKESID+5,A*16+D
280 REM - SET THE SUSTAIN/RELEASE RATE.
290 REM > S = SUSTAIN; R = RELEASE
300 S=15
310 R=9
320 POKESID+6,S*16+R
330 REM - CHOOSE THE SAWTOOTH WAVEFORM
340 REM - AND TURN IT ON.
350 POKESID+4,33
```

```
360 REM > HOLD THE NOTE A WHILE
370 FOR PAUSE = 1 TO 128: NEXT
380 REM - START THE RELEASE CYCLE
390 POKESID+4,32
```

The registers of voices 2 and 3 are handled in the same manner as voice 1. There are a few differences in the voice control registers for ring modulation and syncronization. The changes are marked in bold.

VOICE #2 REGISTERS

D407-D40D		
54279 (S+7)		Voice 2: frequency control, low-byte.
54280 (S+8)		Voice 2: frequency control, high-byte.
54281 (S+9)		Voice 2: pulse waveform width, low-byte.
54282 (S+10)		Voice 2: pulse waveform width.
	7-4	unused
	3-0	high-nybble
54283 (S+11)		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 1
	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
54284 (S+12)		Envelope generator 2: att/dec
	7-4	select attack cycle duation: 0-15
	3-0	select decay cycle duration: 0-15

54285 (S+13)		Envelope generator 2: sus/rel
	7-4	select sustain cycle duation: 0-15

VOICE #3 REGISTERS

D40E-D414

54286 (S+14)		Voice 3: frequency control, low-byte.
54287 (S+15)		Voice 3: frequency control, high-byte.
54288 (S+16)		Voice 3: pulse waveform width, low-byte
54289 (S+18)		Voice 3: pulse waveform width.
	7-4	unused
	3-0	high-nybble
54290 (S+18)		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 1
	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
54291 (S+19)		Envelope generator 3: att/dec
	7-4	select attack cycle duration: 0-15
	3-0	select decay cycle duration: 0-15
54292 (S+20)		Envelope generator 3: sus/rel
	7-4	select sustain cycle duration: 0-15
	3-0	select release cycle duration: 0-15
54293 (S+21)		Filter cutoff frequency: low-nybble. (bits 2-0)

54294 (S+22) Filter cutoff frequency: high-byte.

These locations set the cutoff frequency used by the filters.

For example, to make 1000 Hz the cutoff frequency:

```
POKE 54293,3

POKE 54294,232
```

In other words, (3 * 256) + 232 = 1000. Only numbers from 0 to 7 can be POKEd in location 54293. The filtering location at 54296 will look here to find the frequency to work with.

D417

54295 (S+23)		Filter resonance/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

This is the switch box that turns on and off the filter for the voices you select. To turn on the filter if "V" is the voice number:

```
POKE 54295, PEEK (54295) OR 2↑V
```

Example:

```
POKE 54295, PEEK (54295) OR 4
```

will turn on the filter for voice 2.

```
POKE 54295, PEEK (54295) AND (255-V↑2)
```

turns off the filter.

```
POKE 54295, PEEK (54295) AND 251
```

turns off the filter for voice 2.

You can filter more that one voice at a time by turning on the proper bits. The filter resonance area, bits 4-7, can make the voice sound either sharp or dull. There are 15 possible resonance settings. To enter them let R = the resonance desired.

POKE 54295, PEEK (54295) + (R * 16)

D418

54296 (S+24)		Select filter mode and volume.
	7	cut-off voice 3 output: 1=OFF
	6	select filter high-pass mode: 1=ON
	5	select filter band-pass mode: =ON
	4	select filter low-pass mode: 1=ON
	3-0	select output volume: 0 (off) – 15(max)

The volume of the sound is controlled here. Poke a number between 0 and 15. 0 is off, 15 is the loudest volume. The volume is set the same for all voices, so if you want one voice to sound louder than another, this register will have to be rePOKEd every time you change the volume.

POKE 54296,15

sets the highest volume.

Bit 4: You turn on the low pass filter at bit 4 with:

POKE 54296, PEEK (54296) AND 16

This passes any frequency lower than the cut off set in location 54293 and 54294. The frequencies above the cut off are reduced in volume. The higher the frequency, the greater the reduction in volume.

Bit 5: The high pass filter will pass on frequencies higher than the cut off. Low frequencies will be lower in volume.

POKE 54296, PEEK (54246) AND 32

will start the high pass filter.

POKE 54296, PEEK (54296) AND 48

will start both the high pass and low pass filters. This is called a notch

reject filter. All frequencies except those near the cut off will pass through.

Bit 6: The bandpass filter is the opposite of the notch reject filter. It will pass only frequencies near the cut off.

POKE 54296, PEEK (54296) AND 64

will turn it on.

Bit 7: Voice 3 can be set so its voice can't be sent to a speaker. This is useful if you want to use the output of voice 3 in modulation with other voices.

POKE 54296, PEEK (54296) AND 128

will turn off the output of voices.

D419-D41A

54297 (S+25) — A/D converter: game paddle 1 (0-255).

54298 (S+26) — A/D converter: game paddle 2 (0-255).

Games using paddles must use a machine language paddle routine because of the complexity of the conversion from reading these locations. See location 56320 for a machine language game paddle routine.

D41B

54299 (S+27) — Oscillator 3 random number generator.

This location produces a random number from 0 to 255 when voice 3 is set to the noise waveform.

D41C

54300 (S+28) — Envelope generator 3 output.

If voice 3 is turned on, this location will have digital output of the A/D/S/R "envelope" for voice 3. This can be added to the filter frequency, for example, to produce a range of interesting sounds.

D500-D7FF
54528-55295 RESERVED FOR FUTURE I/O EXPANSION

Commodore has plans for this area...someday. But until they do, this would be a great place to put machine language programs.

SCREEN COLOR AREA

D800-DBFF
55296-56319 SCREEN COLOR CONTROL RAM
(ONLY BITS 3-0 USED)

This area parallels the screen memory area 1024 to 2023. If you are POKEing characters to the screen then you must also POKE the color of the character here.

POKE 1024,83

puts a heart in the top left corner of the screen.

POKE 55296,2

will make it a red heart.

Here are the values to POKE into a color memory location to change a character's color:

0	BLACK	8	ORANGE
1	WHITE	9	BROWN
2	RED	10	Light RED
3	CYAN	11	GREY 1
4	PURPLE	12	GREY 2
5	GREEN	13	Light GREEN
6	BLUE	14	Light BLUE
7	YELLOW	15	GREY 3

This program will fill the screen with hearts, then use all the colors available on the computer.

```
100 REM ********************************
110 REM *    RANDOM COLOR HEARTS       *
120 REM ********************************
130 :
140 REM >>      CLEAR SCREEN        <<
150 PRINT CHR$(147)
160 :
170 REM >> START OF COLOR MEMORY <<
180 MIN = PEEK(243) + 256*PEEK(244)
190 MAX = MIN + 999
200 :
210 REM >> POKE HEARTS ONTO SCREN <
220 FOR H=1024 TO 2023:POKE H,83:NEXT H
230 :
240 REM >> TOGGLE THROUGH COLORS <<
250 FOR CO=0 TO 15
260 :
270 REM >>  POKE COLORS IN WITH  <<
280 REM >>   RANDOM INCREMENTS   <<
290 FOR CM=MIN TO MAX STEP RND(1)*1+4
300 POKE CM,CO
310 :
320 REM >> NEXT COLOR TO POKE IN <<
330 NEXT CM,CO
340 :
350 REM >>   RETURN TO REPEAT    <<
360 GO TO 210
9990 END
```

COMPLEX INTERFACE ADAPTER (CIA) #1

This is a pretty technical area. But in this high-tech world are the controls for joysticks and paddles. See location 56321 if you want to add these routines to the programs you write.

DC00
56320

	PEEK at this address to read the joystick at control port 2.
	Data port A: keyboard, joystick, paddles
7-0	write keyboard column values for keyboard scan.
7-6	select paddle input port: 01=port A, 10=port B
4	joystick 2 fire button: 1=fire
3-0	joystick 2 direction (0-15)

DC01
56321

	PEEK here to read the joystick values at control port 1.
	Data port B: keyboard, joystick, paddles and lightpen
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/lightpen trigger (1=fire)
3-0	joystick 1 direction (0-15)

These two locations have a lot of work to do like scanning the keyboard to see if a key has been pressed, and checking to see if a joystick, paddle or lightpen is in use.

Plug your joystick into control port 1 and run this program.

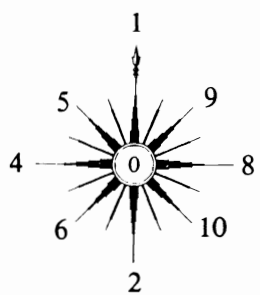
```
100 FORK=0TO10
110 READDR$(K):NEXT
120 DATA"","N","S","","W","NW"
130 DATA"SW","","E","NE","SE"
140 PRINT"GOING...";
150 GOSUB200
160 IFDR$(JV)=""THEN 180
170 PRINTDR$(JV);" ";
180 IFFR=16THEN 150
190 PRINT"***F***I***R***E***":GOTO150
200 JV=PEEK(56321)
210 FR=JVAND16
220 JV=15-(JVAND15)
230 RETURN
```

Moving the joystick will print out the direction you're headed on the screen. Press the fire button too.

Since the keyboard is scanned, these 4 keys will act as a "mock joystick": the back arrow (←), the CTRL, 1 and 2.

Run the program again and try these keys instead of the joystick. Some combinations aren't allowed, like trying to go east and west at the same time.

This is the compass rose that shows which direction the joystick is pointed. Check these values for movement in your subroutines.



A machine language routine is needed to check the paddles because of the complexity of reading them.

The values in each location range from 0 to 255 depending on the rotation of the paddles.

```
100 PRINTCHR$(147)
110 C=12*4096
120 FORI=0TO63:READA:POKEC+I,A:NEXT
130 SYS C
140 P1=PEEK(C+257)
150 P2=PEEK(C+258)
160 P3=PEEK(C+259)
170 P4=PEEK(C+260)
180 W1=PEEK(C+261):W2=PEEK(C+262)
190 PRINTP1,P2,P3,P4
200 PRINT:PRINT"FIRE A";W1,"FIRE B";W2
210 FORW=1TO50:NEXT
220 PRINTCHR$(19):GOTO  130
230 DATA 162,1,120,173,2,220,141,0,193
240 DATA 169,192,141,2,220,169
250 DATA 128,141,0,220,160,128,234,136
260 DATA 16,252,173,25,212,157
270 DATA 1,193,173,26,212,157,3,193,173
280 DATA 0,220,9,128,141,5,193
290 DATA 169,64,202,16,222,173,0,193
300 DATA 141,2,220,173
310 DATA 1,220,141,6,193,88,96
```

When plugged into port 1, the paddles are read at (C+257) and (C+259) and the fire button is read at (C+262). The value in (C+262) will change depending on which fire button is pressed or if both are pressed at the same time.

Paddles in port 2 are read at (C+258) and (C+260). The fire button is (C+261). It is read the same way indicated above.

DC02
56322 — Data direction register, port A

DC03
56323 Data direction register, port B
Normal values for each, 241

These are the data direction bytes for the control port registers. When a bit is set to one, that means that the port is receiving input. A zero turns on that bit for output.

Bit 5 is not counted because line five carries voltage. These data direction registers must be POKEd before you set the proper values of the control registers.

DC04
56324 Timer A: low-byte

DC05
56325 Timer A: high-byte

DC06
56326 Timer B: low-byte

DC07
56327 Timer B: high-byte

Both CIA chips have two powerful 16-bit timing devices. They can be used to time various waveforms, pulse widths and frequencies for internal and external signal generation. These timers can be used individually or linked together to expanded timing durations. These registers are read in BCD (binary-coded decimal), each nybble describes a digit in the timer value. BCD format is faster for I/O operations.

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
7 AM/PM flag

These four registers store a real-time AM/PM TOD, time-of-day clock. This is another programmable CIA timing feature which, when read, returns the respective TOD values. When written to (via setting bit 7 in control register 56335), these values latch on the ALARM. This programmable alarm allows the CIA to generate an interrupt at a specified time. Only the MSB (most significant bit) of the hours register is used to specify AM or PM. The values must be latched in one at a time starting with the hours register and the clock will not start until the 1/10 sec. register is set. This ensures that the proper time is specified.

DC0C
56332 — Synchronous serial I/O data buffer.

This register stores the values of the serial port which is a buffered, 8-bit synchronous shift register system. With every eight clock counts (CNT), the shift register deposits a value in this register. The clock counts are generated by TIMER A which is also used as a baud rate generator. After eight clock counts an interrupt is enabled to send for more data. This constitutes a double-buffered I/O system where the microprocessor stays one byte ahead of the shift register which stays a byte ahead of the serial port buffer. This lets you load new data on the serial bus before the shift register clears.

DC0D
56333 — CIA interrupt control: read IRQs/write mask (to IRQ)

7	IRQ flag (1=IRQ occurred)/set-clear flag
4	FLAG1 IRQ (cassette read/serial IEEE SRQ input)
3	serial port interrupt
2	time-of-day clock alarm interrupt
1	timer B underflow interrupt
0	timer A underflow interrupt

This is the register that contains the interrupt and masking information for the five sources of interrupts from the 6526. These interrupts are the underflow from TIMER A, the underflow from TIMER B, TOD ALARM, FLAG and serial port full/empty conditions. When read, this location becomes a data register which accepts the interrupts being generated. When written to, this location creates a mask for the IRQ line which provides selective control over the interrupt system. If bit 7 is

zeroed, any mask bit which is on (1) is cleared while off bits are sent through.

DC0E 56334		CIA control register timer A
	7	time-of day clock frequency: 1=50Hz, 0=60Hz
	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

This is the control register for the internal TIMER A and the TOD clock talked about under locations 56324 thru 56331.

(BIT 0) START - This bit enables and disables TIMER A. When an underflow condition occurs in the one-shot mode, this bit is automatically reset.

(BIT1) PBON - When on, this bit allows the timer output of A to appear on Port B.

(BIT 2) OUTMODE - This allows the ouput of PORT B to either toggle (flip on and off) or pulse singly over one cycle duration.

(BIT 3) RUNMODE - This chooses the one-shot or continuous modes. In the one-shot mode, the timer will count down to zero from the value latched into it, enable an interrupt, and then stop. In continuous mode, the value is re-latched and done again.

(BIT 4) INMODE - This bit controls which clock is used to decrement the timer; either the 02 clock pulses or the external pulses applied to the count (CNT) pin.

(BIT 6) SPMODE - This bit controls how TIMER A clocks the serial bus. When one, the timer writes out to the serial bus. When zero, the serial bus provides input.

(BIT 7) TODIN - Sets the TOD pin for accurate time.

DC0F		
56335		CIA control register timer B
	7	set alarm/TOD clock 1=alarm, 0=clock

This is the same register for TIMER B with the exception of bits 5 and 6. These bit pairs are used for timer count transitions and extended timer use (using both timers together).

COMPLEX INTERFACE ADAPTER (CIA) #2

DD00		
56576		Data port A (serial IEEE, RS-232, VIC memory control).
	7	serial IEEE data input
	6	serial IEEE clock pulse input
	5	serial IEEE data output
	4	serial IEEE clock pulse output
	3	serial IEEE ATN signal output
	2	RS-232 data output (user port)
	1-0	VIC chip system memory bank select

This multi-functional register controls video bank-select, and is the control register when an IEEE-488 interface is present on the expansion port. This register must be set in correspondence with its data direction register (56578).

DD01		
56577		Data port B (user port, RS-232)
	7	user / RS-232 data set ready
	6	user / RS232 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
	user / RS-232 receive: start-bit (IRQ flag)

Similar to location DD00, this register returns values of user PORT B. It also handles the RS-232 connection. This register, too, must be POKEd in conjunction with its data direction register (56579) to achieve results.

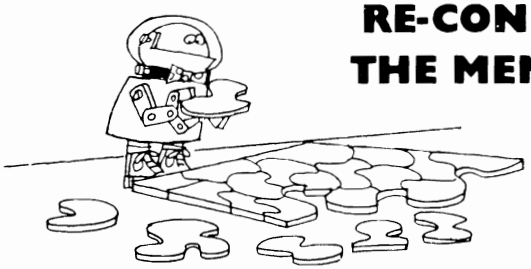
The following registers behave the same as on CIA #1. Both 6526 chips have identical timing and clock capabilities.

THE FOLLOWING LOCATION DECRIPTIONS APPLY FROM CIA #1

DD02 56578	Data direction register, port A
DD03 56579	Data direction register, port B
DD04 56580	Timer A: low-byte
DD05 56581	Timer A: low-byte
DD06 56582	Timer B: high-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: read NMIs/ write mask (to IRQ)
	7	IRQ flag (1 => IRQ occurred) / set-clear flag
	4	FLAG1 IRQ: cassette read/ serial IEEE SRQ input
	3	serial port interrupt
	2	time-of-day clock alarm interrupt
	1	timer B underflow interrupt
	0	timer A underflow interrupt
DD0E 56590		CIA control register A, same as CIA 1
DD0F 56591		CIA control register B, same as CIA 1
DE00-DEFF 56832-57087		RESERVED FOR FUTURE I/O EXPANSION
DF00-DFFF 57088-57343		RESERVED FOR FUTURE I/O EXPANSION

APPENDICES



RE-CONFIGURING THE MEMORY MAP

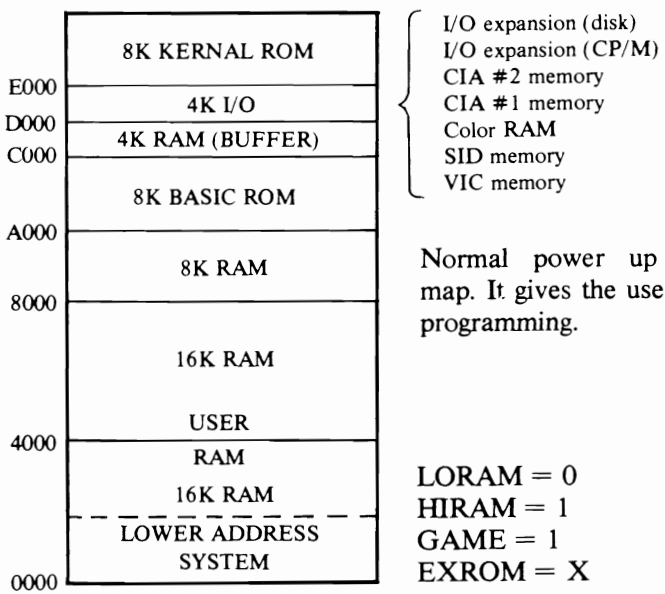
With the COMMODORE 64 you get more than one kind of machine. You have the ability to rearrange sections of memory into eight different memory maps.

Admittedly, this isn't for the novice, but if you are the kind of programmer that can make a computer dance, the 64 will do a fine jig.

There are 8 memory map possibilities. Here's a chart:

X = Don't Care
0 = OFF
1 = ON

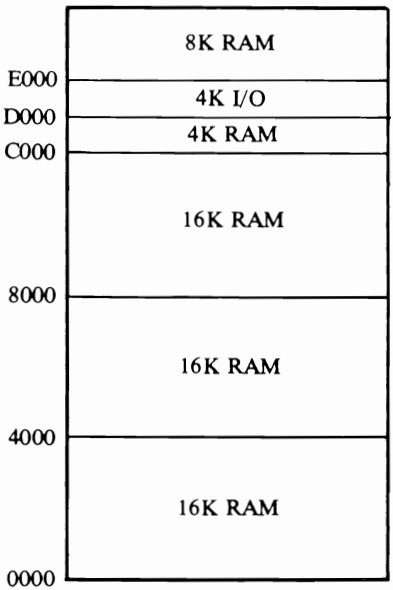
Map #1



Normal power up memory map. It gives the user 38K for programming.

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

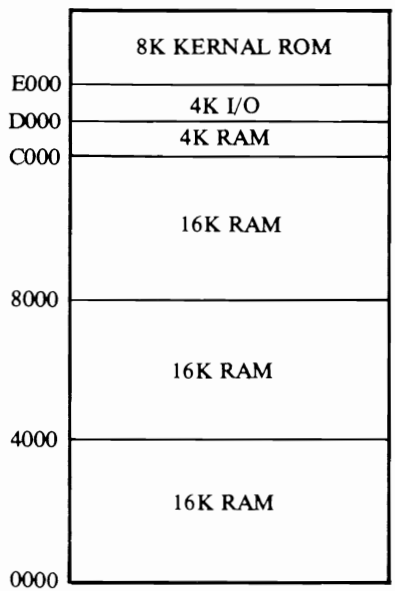
Map #2



60K RAM for I/O devices w/o system routines.

LORAN = 1
HIRAM = 0
GAME = 1 or 0
EXROM = X

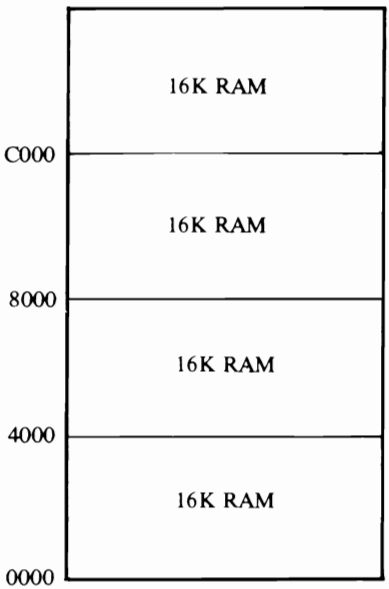
Map #3



52K for I/O devices and other languages including CP/M.

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

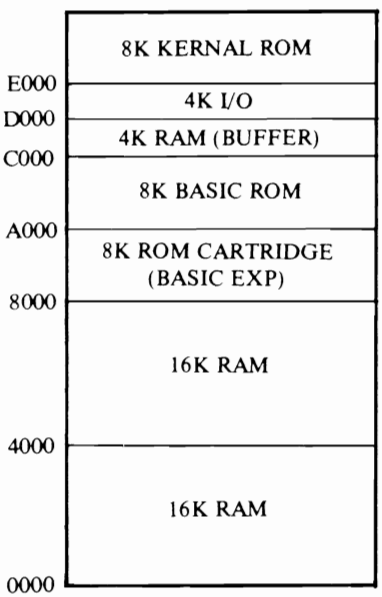
Map #4



Full 64K free RAM. No I/O operators can be done here.

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

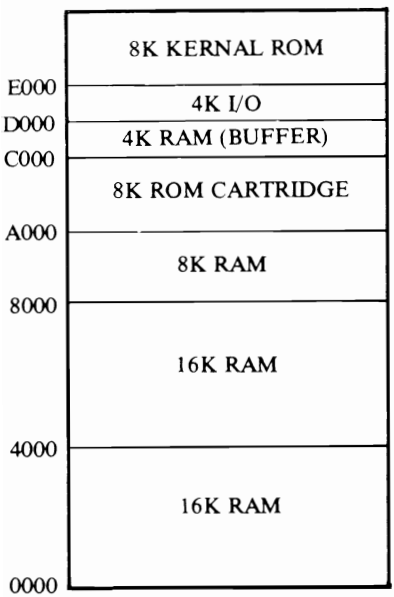
Map #5



32K RAM for BASIC user with 8K taken up for expansion cartridges.

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

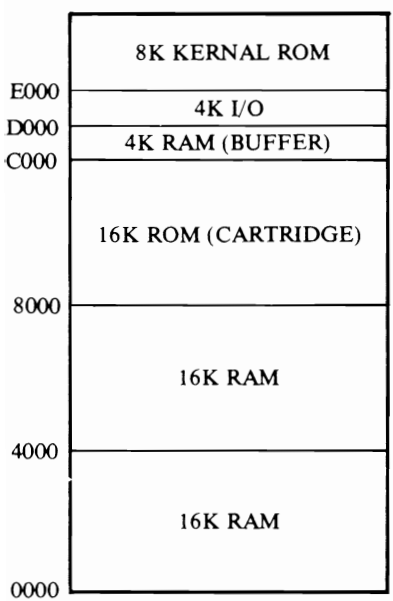
Map # 6



40K of user RAM with 8K of expansion ROM that does not reduce BASIC.

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

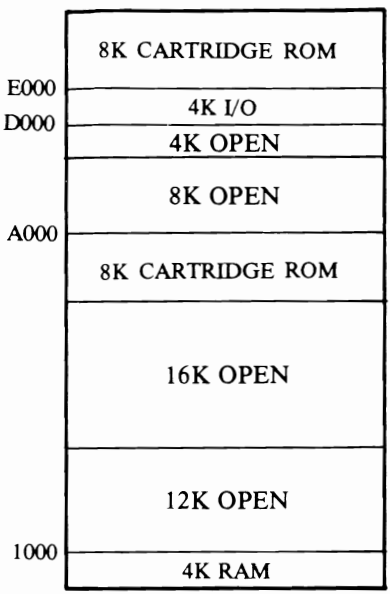
Map #7



Same as Map #6 except that 32K is for the user and 16K ROM for expansion.

LORAN = X
HIRAM = X
GAME = 0
EXROM = 1

Map #8



This configuration is used for the ULTIMAX video game cartridges so that they are compatible on the Commodore 64.

You have the capability of flipping some of these memory sections in and out, freeing that area for other uses such as free RAM for programming. Second, you can internally re-locate some smaller sections such as screen and character memories (see loc. 648). This gives flexibility to several kinds of programming environments.

For example, locations 0 and 1 in the memory show how to switch the 3 important ROMS, the KERNAL, BASIC and the Character Generator in and out. This is done by the lower three bits of location 1. This location is actually the control register for some of the processor's (6510) addressing lines.

You can even move BASIC into RAM and make your own modifications to it.

There are other addressing lines monitored by location 1 which are connected to the expansion port that you have no control over (at least not with software). These lines automatically reconfigure the memory map to the specifications determined by what expansion cartridge is in use. The cartridge could be a game or word processor, for example.

ROM MEMORY MAP

To start out with, the BASIC and KERNAL ROMs share the 64K addressing space. This means that while your ROMs are present upon turning on the computer, there is still the RAM hidden 'behind' it. When you read this area, you get the contents of the ROM routines. If, however you write to it, your information is stored in the RAM behind it. This is convenient if you want to store something in this RAM, but how can you get to it? The solution is to flip out any or all of the ROMs present in the overall memory configuration using the Commodore 64's impressive 'bank-switching' feature (see loc. 1). You can free up to 16K ROM memory this way and this amount of memory for extra RAM.

WHAT IS INSIDE THE BASIC AND KERNAL ROMS?

Let's start with the BASIC interpreter which everyone gets automatically when they turn on their Commodore 64. An interpreter is a large library of routines and subroutines which break down the BASIC commands (tokens) and execute the proper functions in machine code.

An interpreter is different from a compiler which must methodically break down the 'source code' several times (passes) to be reconstructed in another binary 'object code' file. An interpreter, while not as efficient as a compiler, is certainly a lot easier to use because you don't need to wait each time for a compilation process (which can take several

minutes) each time you debug your program. In addition, the interpreter accepts and processes most of the commands that you execute directly into the computer in the 'immediate mode'.

The Kernal is a similar bag of tricks. It is a miniature operation system, not as versatile as the BASIC interperter, but built for speed and efficiency of operation. This is useful if you need to write machine routines and you do not want to re-invent the wheel, that is, constantly writing the same input or output routines. Just use the built-in KERNAL subroutines.

When you use BASIC, the computer "automatically" knows what to do with what you wrote and where to go to get the information. The KERNAL, however, isn't so automatic. There are certain "calling" procedures that must be followed. "Calling" means setting up the information the KERNAL routine needs to have before the routine can be used. See Commodore's "Programmers Reference Guide" for the correct calling procedures for each KERNAL routine.

The list of locations in this section contain the starting address of all the BASIC Interpreter and Kernal operating system routines. They are important to know because you can change them or at least alter their inputs to non-standard results. You can map all or just a section of these ROM routines with this FOR/NEXT loop. For instance, to map BASIC into RAM:

```
FOR I = 40960 TO 49151 : POKE I,I: NEXT I
```

That's right. You just POKE an area with its own contents! Remember you must take out the ROM when you are done so that you can read the RAM. Take the ROM out of operation. You can change the ROM routines if you wish. You can modify or even create your own BASIC.

A few last things to note. First, the sharp-eyed reader will take notice of the fact that a number of BASIC ROM routines are in the Kernal 8K section. As a result, if you rid yourself of the Kernal, your BASIC will not function normally. Second, the ROM memory map locations are not all in chronological order. This is to consolidate certain routines which are used together.

THE KERNAL

The Kernal is Commodore's name for a table of standard subroutines. Everything that goes in or out of the computer uses these routines.

If you write programs in machine language, the list of routines here will enable you to use some of the power of the machine instead of writing your own. These locations can be called by BASIC using the 'SYS' command after loading locations 780-783 with necessary inputs. (see program)

The Kernal is like a wall between you and the inner workings of the operating system. These routines are like doors in the wall, allowing you to use different parts of the system. Advanced programmers may want to cut some windows in the wall to get to some of the subroutines that the Kernal uses. Don't do this. Commodore is noted for ROM upgrades, which means they will rebuild that wall. They promise to keep the doors, but the windows will be gone!

Here is a brief summary of the Kernal routines.

Label Hex. Addr.	Dec. Loc.	Description
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.

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 buffer.
IOBASE FFF3	65523	Returns base address of I/O device.
IOINIT FF84	65412	Initialize input/output.
LISTEN FFB1	65457	Command serial bus device 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.
PLOT FFF0	65520	Read/set X,Y cursor position.
RAMTAS FF87	65415	Initialize RAM, allocate tape buffer, set screen $0400.
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	Send logical-file, device, secondary address.
SETMSG FF90	65424	Control Kernal messages.
SETNAM FFBD	65469	Set file name.
SETTIM FFDB	65499	Set real time 'jiffy' 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 device to UNLISTEN.
UNTLK FFAB	65451	Command serial bus device to UNTALK.
VECTOR FF8D	65421	Read/set vectored I/O.

BASIC ROM ROUTINE STARTING ADDRESSES

This list of ROM addresses was published by Commodore in the October/November 1982 issue of their magazine. According to some sources, a few of the routines listed here have been changed. Try these first to see if what you want to do will work. For detailed information, you should contact customer service at Commodore either through regular mail or electronic mail on Compuserve.

One location you should try is 64738. By using the command SYS 64738, the 64 will reset itself and display the same message as if you had just turned on the computer. This is called a 'cold start'. It's just like turning the computer off and then on again. Everything is set to its default value and any program in the computer is destroyed. This SYS will save wear and tear on the switch and the power supply.

HEX	DEC.	ROUTINE
A000	40960	ROM control
A00C	40972	Keyword action vectors
A052	41042	Function vectors
A080	41088	Operator vectors
A09E	41118	Keywords
A19E	41374	Error messages
A328	41768	Error message vectors
A365	41829	Misc. messages
A38A	41866	Scan stack for FOR/GOSUB
A3B8	41912	Move memory
A3FB	41979	Check stack depth
A408	41992	Check memory space
A435	42037	*out of memory*
A437	42039	Error routine
A469	42089	BREAK entry
A474	42100	*ready*
A480	42112	Ready for BASIC
A49C	42140	Handle new line
A533	42291	Re-chain lines
A560	42336	Receive input line
A579	42361	Crunch tokens
A613	42515	Find BASIC line
A642	42562	Perform [NEW]

A65E	42590	Perform [CLR]
A68E	42638	Backup text pointer
A69C	42654	Perform [LIST]
A742	42818	Perform [FOR]
A7ED	42989	Execute statement
A81D	43037	Perform [RESTORE]
A82C	43052	Break
A82F	43055	Perform [STOP]
A831	43057	Perform [END]
A857	43095	Perform [CONT]
A871	43121	Perform [RUN]
A883	43139	Perform [GOSUB]
A8A0	43168	Perform [GOTO]
A8D2	43218	Perform [RETURN]
A8F8	43256	Perform [DATA]
A906	43270	Scan for next statement
A928	43304	Perform [IF]
A93B	43323	Perform [REM]
A94B	43339	Perform [ON]
A96B	43371	Get fixed point number
A9A5	43429	Perform [LET]
AA80	43648	Perform [INPUT#]
AA86	43654	Perform [CMD]
AAA0	43680	Perform [PRINT]
AB1E	43806	Print string form (y.a)
AB3B	43835	Print format character
AB4D	43853	Bad input routine
AB7B	43899	Perform [GET]
ABA5	43941	Perform [INPUT#]
ABBF	43967	Perform [INPUT]
ABF9	44025	Prompt & input
AC06	44041	Perform [READ]
ACFC	44284	Input error messages
AD1E	44318	Perform [NEXT]
AD78	44408	Type match check
AD9E	44446	Evaluate expression
AEA8	44712	Constant - Pi
AEF1	44785	Evaluate within brackets
AEF7	44791	*) *
AEFF	44799	comma..
AF08	44808	Syntax error
AF14	44820	Check range
AF28	44840	Search for variable

AFA7	44967	Setup FN reference
AFE6	44790	Perform [OR]
AFE9	45033	Perform [AND]
B016	45078	Compare
B081	45185	Perform [DIM]
B08B	45195	Locate variable
B113	45331	Check alphabetic
B11D	45341	Create variable
B194	45460	Array pointer subroutine
B1A5	45477	Value 32768
B1B2	45490	Float-fixed
B1D1	45521	Set up array
B245	45637	*bad subscript*
B248	45640	*illegal quantity*
B34C	45900	Compute array size
B37D	45949	Perform [FRE]
B391	45969	Fix-float
B39E	45982	Perform [POS]
B3A6	45990	Check direct
B3B3	46003	Perform [DEF]
B3E1	46049	Check FN syntax
B3F4	46068	Perform [FN]
B465	46181	Perform [STR$]
B475	46197	Calculate string vector
B487	46215	Set up string
B4F4	46324	Make room for strings
B526	46374	Garbage collection
B5BD	46525	Check salvageability
B606	46598	Collect string
B63D	46653	Concatenate
B67A	46714	Build string to memory
B6A3	46755	Discard unwanted string
B6DB	46811	Clean descriptor stack
B6EC	46828	Perform [CHR$]
B700	46949	Perform [LEFT$]
B72C	46902	Perform [RIGHT$]
B72C	46892	Perform [RIGHT$]
B737	46903	Perform [MID$]
B761	46945	Pull string parameters
B77C	46972	Perform [LEN]
B782	46978	Exit string-mode
B78B	46987	Perform [ASC]
B79B	47003	Input byte parameter

B7AD	47021	Perform [VAL]
B7EB	47083	Parameters: POKE/WAIT
B7F7	47095	Float-fixed
B80D	47117	Perform [PEEK]
B824	47140	Perform [POKE]
B82D	47149	Perform [WAIT]
B849	47177	Add 0.5
B850	47184	Subtract-from
B853	47187	Perform [subtract]
B86A	47210	Perform [add]
B947	47431	Complement FAC#1
B97E	47486	* overflow *
B983	47491	Multiply by zero byte
B9EA	47594	Perform [LOG]
BA2B	47659	Perform [multiply]
BA59	47705	Multiply-a-bit
BA8C	47756	Memory to FAC#2
BAB7	47799	Adjust FAC#1/#2
BAD4	47828	Underflow/overflow
BAE2	47842	Multiply by 10
BAF9	47865	+ 10 in floating point
BAFE	47870	Divide by 10
BB12	47890	Perform [divide]
BBA2	48034	Memory to FAC#1
BBC7	48071	FAC#1 to memory
BBFC	48124	FAC#2 to FAC#1
BC0C	48140	FAC#1 to FAC#2
BC1B	48155	Round FAC#1
BC2B	48171	Get sign
BC39	48185	Perform [SGN]
BC58	48216	Perform [ABS]
BC5B	48219	Compare FAC#1 to memory
BC9B	48283	Float-fixed
BCCC	48332	Perform [INT]
BCF3	48371	String to FAC
BD7E	48510	Get ASCII digit
BDC2	48578	Print *IN..*
BDCD	48589	Print line number
BDDD	48605	Float to ASCII
BF16	49818	Decimal constants
BF3A	48954	TI constants
BF71	47089	Perform [SQR]
BF7B	49019	Perform [power]

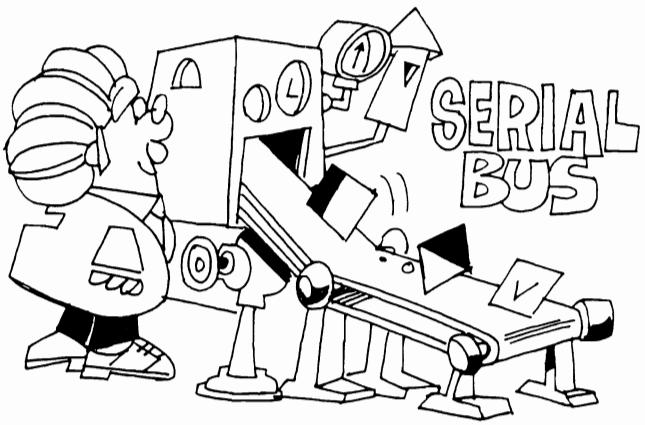
BFB4	49076	Perform [negative]
BFED	49133	Perform [EXP]
E043	57411	Series evaluation 1
E059	56433	Series evaluation 2
E097	57495	Perform [RND]
E0F9	57593	?? breakpoints ??
E12A	57642	Perform [SYS]
E156	57686	Perform [SAVE]
E165	57702	Perform [VERIFY]
E168	57704	Perform [LOAD]
E1BE	57790	Perform [OPEN]
E1C7	57799	Perform [CLOSE]
E1D4	57812	Parameters for LOAD/SAVE
E206	57862	Check default parameters
E20E	57870	Check for comma
E219	57881	Parameters for OPEN/CLOSE
E264	57956	Perform [COS]
E26B	57963	Perform [SIN]
E2B4	58036	Perform [TAN]
E30E	58126	Perform [ATN]
E37B	58235	Warm restart
E394	58260	Initialize
E3A2	58264	CHRGET for zero page
E3BF	58303	Initialize BASIC
E447	58439	Vectors for $300
E452	58451	Initialize vectors
E45F	58463	Power-up message
E500	58624	Get I/O address
E505	58629	Get screen size
E50A	58634	Put/get row/column
E518	58648	Initialize I/O
E544	58692	Clear screen
E566	58726	Home cursor
E56C	58732	Set screen pointers
E5A0	58784	Set I/O defaults
E5B4	58800	Input from keyboard
E632	58930	Input from screen
E694	59012	Quote test
E691	59025	Setup screen print
E6B6	59062	Advance cursor
E6ED	59117	Retreat cursor
E701	59127	Back into previous line
E716	59158	Output to screen

E87C	59516	Go to next line
E891	59537	Perform [RETURN]
E8A1	59553	Check line decrement
E8B3	59571	Check line increment
E8CB	59595	Set color code
E8DA	59610	Color code table
E8EA	59626	Scroll screen
E965	59749	Open space on screen
E9C8	59848	Move a screen line
E9E0	59872	Sync. the color transfer
E9F0	59888	Set start-of-line
E9FF	59903	Clear screen line
EA13	59923	Print to screen
EA24	59940	Sync. color pointer
EA31	59953	Interrupt - clock etc.
EA87	60039	Read keyboard
EB79	60281	Keyboard select vectors
EB81	60289	Keyboard 1 - unshifted
EBC2	60354	Keyboard 2 - shifted
EC03	60419	Keyboard 3 - COMMODORE
EC44	60484	Graphics/text control
EC4F	60495	Set graphics/text mode
EC78	60536	Keyboard 4
ECB9	60601	Video chip setup
ECE7	60647	Shift/run equivalent
ECF0	60656	Screen In address low
ED09	60681	Send *talk*
ED0C	60684	Send *listen*
ED40	60736	Send to serial bus
EDB2	60850	Serial timeout
EDB9	60857	Send listen SA
EDBE	60862	Clear ATN
EDC7	60871	Send talk SA
EDCC	60876	Wait for clock
EDDD	60893	Send serial deferred
EDEF	60911	Send *untalk*
EDFE	60926	Send *unlisten*
EE13	60947	Receive from serial bus
EE85	61061	Serial clock on
EE8E	61070	Serial clock off
EE97	61079	Serial output *I*
EEA0	61088	Serial output *O*
EEA9	61097	Get serial in & clock

EEB3	61107	Delay 1 ms.
EEBB	61115	RS-232 send
EF06	61190	Send new RS-232 byte
EF2E	61230	No-DSR error
EF31	61233	No-CTS error
EF3B	61243	Disable timer
EF4A	61258	Compute bit/count
F8D0	63696	Check tape stop
F8E2	63714	Set read timing
F92C	63788	Read tape bits
FA60	64096	Store tape characters
F8BE	64398	Reset pointers
FB97	64407	New character setup
FBA6	64422	Send transition to tape
FBC8	64456	Write data to tape
FBCD	64461	IRQ entry point
FC57	64599	Write tape leader
FC93	64659	Restore normal IRQ
FCB8	64696	Set IRQ vector
FCCA	64714	Kill tape motor
FCD1	64721	Check R/W pointer
FCDB	64731	Bump R/W pointer
FD50	64848	Initialize sys. constraints
FCE2	64738	Power reset entry
FD02	64770	Check 8K-ROM
FD10	64784	8K-ROM mask
FCDB	64731	Bump R/W pointer
FCE2	64738	Power reset entry
FD02	64770	Check 8K-ROM
FD15	64789	Kernal reset
FD1A	64794	Kernal move
FD30	64816	Vectors
FD9B	64923	IRQ vectors
FDA3	64931	Initialize I/O
FDDD	64989	Enable timer
FDF9	65017	Save filename data
FE00	65024	Save file details
FE07	65031	Get status
FE18	65048	Flag status
FE1C	65052	Set status
FE21	65067	Set timeout
FE25	65061	Read/set top of memory

FE27	65063	Read top of memory
FE2D	65069	Set top of memory
FE34	65076	Read/set bottom of memory
FE43	65091	NMI entry
FE66	65126	Warm start
FEB6	65206	Reset IRQ & exit
FEBC	65212	Interrupt exit
FEC2	65218	RS-232 timing table
FED6	65238	NMI RS-232 in
FF07	65287	NMI RS-232 out
FF43	65347	Fake IRQ
FF48	65352	IRQ entry
FF81	65409	Jumbo jump table
FFFA	65530	Hardwire vectors
FE59	65113	RS-232 receive
EF7E	61310	Setup to receive
EFC5	61381	Receive parity error
EFCA	61386	Receive overflow
EFCD	61389	Receive break
EFD0	61392	Framing error
EFE1	61409	Submit to RS-232
FOOD	61453	Send to RS-232
FOOD	61453	No-DSR error
F017	61463	Send to RS-232 buffer
F04D	61517	Input from RS-232
F086	61574	Get from RS-232
F0A4	61604	Check serial bus idle
F0BD	61629	Messages
F12B	61739	Print if direct
F13E	61758	Get..
F14E	61774	..from RS-232
F157	61783	Input
F199	61849	Get..tape/serial/RS-232
F1CA	61909	Output..
F1DD	61917	..to tape
F20E	61966	Set input device
F250	62032	Set output device
F291	62097	Close file
F30F	62223	Find file
F31F	62239	Set file values
F32F	62255	Abort all files
F333	62259	Restore default I/O
F34A	62282	Do file open

F3D5	62421	Send SA
F409	62473	Open RS-232
F49E	62622	Load program
F5AF	62895	*searching*
F5C1	62913	Print filename
F5D2	62930	*loading/verifying*
F5DD	62941	Save progrram
F68F	63119	Print *saving*
F69B	63131	Bump clock
F6BC	63164	Log PIA key reading
F6DD	63197	Get time
F6E4	63204	Set time
F6ED	63213	Check stop key
F6FB	63227	Output error messages
F72D	63277	Find any tape header
F76A	63338	Write tape header
F7DO	63440	Get buffer address
F7D7	63447	Set buffer pointers
F7EA	63466	Find specific header
F80D	63501	Bump tape pointer
F817	63511	*press play..*
F82E	63534	Check tape status
F838	63544	*press record*
F841	63553	Initiate tape read
F864	63588	Initiate tape write
F875	63605	Common tape code



THE SERIAL BUS

The 64 communicates with a printer or disk drive through the serial port, the six-pin socket on the back of the computer. This connects the peripheral device to the serial "Bus". A "Bus" is a collection of communication lines shared by such things as disk drives and printers.

Imagine youself as a switchboard operator in a train station. Your job is to tell when and how the trains are to enter and exit the station. It is obvious you must be able to monitor each train's course and in turn, each train engineer must notify you of his intentions. Among other things, your most critical function is to make sure that trains are not on the same track at the same time. The consequences may be disastrous.

Back to the Commodore 64. It has the role of being the switchboard operator of the serial bus. Devices on the serial bus can either 'talk' or 'listen', that is, send or receive information but never at the same time or else all the data would be scrambled. You know how hard it is to understand two people speaking at the same time! The 64 not only talks and listens but 'controls' who will talk and who will listen. Only the computer has this privilege.

The bus has three input lines that bring in data and three output lines that send. Of each three, one line wakes up the device, one line controls the timing of data sent on the serial bus, while the third conveys the data. Recall the train station analogy. In serial transmission, each car of the train is a bit of information.

Here's two more points to keep in mind. First, each device on the serial bus must be recognized by its device address. That is the second number used in the BASIC OPEN statement. For example,

OPEN 1,4

tells the printer, device number 4, to get ready to do some work. The second thing the bus can do is set the device into its selective modes as specified by the third number in the OPEN statement.

OPEN 1, 4, 7

tells the printer to print in upper and lower case letters.

Using BASIC or machine language you can use the serial bus to control other devices.

THE COMPLEX INTERFACE ADAPTERS (CIA)

THIS CIA is not secret, but it is complex. Ask yourself this question: Why is a separate chip required to handle communication tasks? There is no obvious answer. Let's start by saying that there are a number of tasks that the computer must perform in the right sequence in order to 'talk' and 'listen' to another device such as the printer, drives and modem (modulator-demodulator). Each device shares a number of lines with other devices; when one device 'talks' all others 'listen'. These shared lines of communication are known as the serial bus.

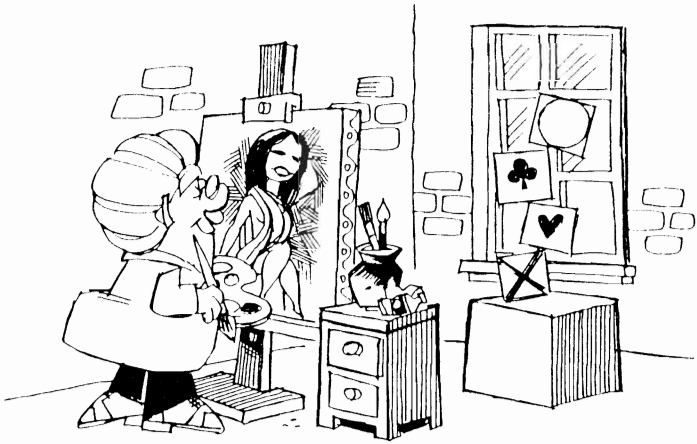
Another dilemma facing communications is that these devices may speak in different 'dialects' at different rates to one another. Whenever communications are not dependent on each other's timing, then a method for coordinating their operations are required. This is known as 'asynchronous communication'. It requires special software and/or hardware support.

COMPLETELY CONFUSED?

Now, pretend you are the computer. You have data you transmit to one of the peripherals, say the printer. But the printer is busily chattering away, printing something else that you told it and hasn't finished. The printer tells you that it is not ready to print what you have, but to 'wait' until it is ready. So you, the computer, put your 'task' on what is known as a 'queue' which tells the printer what it has to do next. Meanwhile, you periodically ask the printer if it is ready to accept the information you want it to process. When the printer responds positively, you can now

empty the queue and process your task. This is what is known as a 'handshaking' protocol and there are several kinds which are constantly occuring in a system which interfaces one or more devices.

This computer has two dedicated CIA chips. They handle the required memory locations in the 64K user-accessible memory space which the computer or user needs to interrogate and modify. When a device or any one of the internal dedicated chips need processing from the 6510 central processor, a signal is sent to one of the CIA chips to request the processor to give some of its time to the task. The CIA has the special privilege of telling the 6510 what it can do with its time on certain occasions. So a big part of the CIA's task is to process the information from these other devices.



BEING AN ARTIST
WITH COMMODORE 64 GRAPHICS

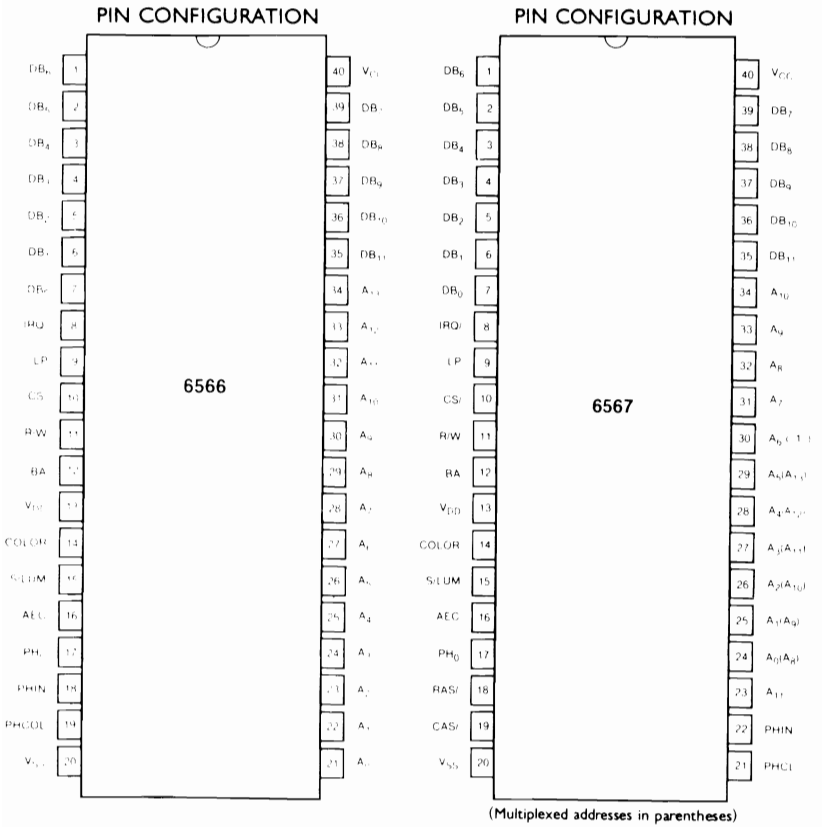
Often the creative mind has limited channels to express itself. Suppose you tried to paint a masterpiece or write a symphony with no prior training in those skills and only conventional tools at your disposal. It may take years of time and labor to achieve your aim. Lots of us have found, in micro-computers, direction in our creativity that we could not achieve elsewhere. With the Commodore 64 you may find for yourself a whole new dimension beyond paintbrush and manuscript for creating graphics and sound. The extentions of your creativity need only conform to understanding the practices for constructing programs that sing and paint.

Now the key to all of this, or at least the graphics end of things is the 6567. In other circles it is called the VIC-II chip. If you wondered what ever happened to the VIC-I chip, it was put in the VIC-20. Remember V.I.C. stands for Video Interface Controller. The VIC-I or 6560 used in the VIC-20 is used for both graphics and sound without the powerful ability to handle things like Sprites like the Commodore 64 can. There is a lot of stuff to play with here, such as the ability to build your own characters and even write new alphabets. That is only the beginning! You can control the destinies of eight different movable objects called Sprites. You control how they move, what they do when they hit other objects, or what they do when they hit each other.

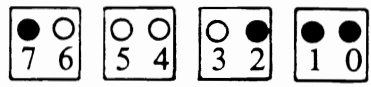
LET'S GET TECHNICAL FOR A MOMENT...

First of all, the VIC-II is, in fact, a two chip processor. The companion chip is the 6566 chip. About the only difference between them is how

they access the system's address busses. Other than that, normal operations between the two are completely transparent to the system and for all practical purposes they can be considered as a single functional unit. The VIC-II, like other dedicated graphics chips on the market, has direct memory access (DMA) to the central processor (6510). This means that when the VIC-II needs to get something done it has to ask the 6510 if it can use the system's 8-bit bi-directional data busses to convey information to the 64K of RAM. The 6510 has to give up machine cycles to comply with the request, thereby slowing down processing. Incidentally, the user can disable the DMA to speed up the system's processing through screen blanking (see location 53265) and disable the interrupts along the address busses so that graphics processing won't interfere with the rest of the system (see location 56334). For those who insist on seeing a simple hardware overview:



In addition, the chip takes care of test display, high resolution graphics, sprites, sprite priorities and sprite collision direction. All of the Commodore 64's graphics are available in multi-color mode which give your display objects more colors to chose from. A rule of thumb is what you gain in colors you lose in resolution. That's because the 8 bits in the byte are read in 'bit pairs'



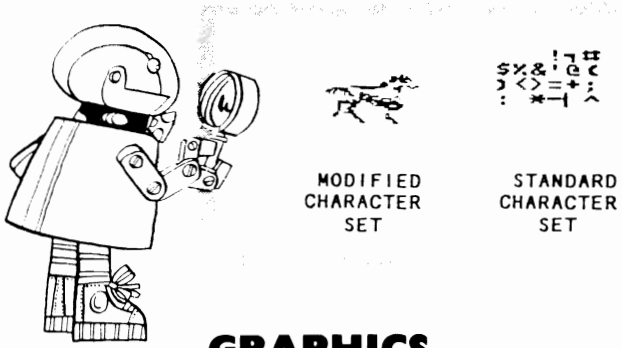
A B C D

A gets color from location 53281 - the screen color.

B gets color from location 33282 - background #1 color.

C gets color from location 53283 - background #2 color.

D gets color depending on where it is located on the screen. This comes from the color RAM area 55296 to 56295.



GRAPHICS PROGRAMMING

VIDEO BANK SELECTON

With the computer, we think that all the memory is available at the same time. 64K is 64K, right? To the VIC-II chip's addressing lines, only 16K chunks of memory can be considered at any one time. There are four sections of 16K banks to choose from when changing the starting locations. The VIC-II chip, however does not have control over what section of memory it sees. This is accomplished by the 6526 Complex Interface Adapter #2 (CIA #2). Two bits used out of control PORT A are used to select the start address location. Here is how to modify these bit selections:

To change video bank:

POKE 53272, PEEK (53272) AND 15 OR A

The values of 'A' control the following values:

Value Of A	Bits	Bank	Starting Location	VIC-II Chip Range
0	00	3	49152	($C000-$FFFF)*
1	01	2	32768	($8000-$BFFF)
2	10	1	16384	($4000-$7FFF)*
3	11	0	0	($0000-$3FFF) (DEFAULT VALUE)

*NOTE: The Commodore 64 character set is not available to the VIC-II chip in BANKS 1 and 3.

The only part of graphics memory that is not relocatable is screen color memory. It always starts at location 55296 ($DBE7) and ends at location 56295 ($DBE7). Actual color is determined by the nybbles in each color byte since a nybble can describe sixteen colors. Whenever a character is POKEd onto the video matrix with the proper screen color code (see screen color code table at the end of this section), its color is determined by the values in the color memory.

PROGRAMMABLE CHARACTERS

Perhaps one of the most versatile techniques in modern graphics programming is the ability to create custom characters for any one variety of purposes. With this technique you can create an entirely new character set, different from the one found in the character generator ROM. Think of the possibilities! An entirely new alphabet, or font, of special characters used for building certain kind of displays. Fonts are particularly useful for writing arcade style games or creating several characters to animate. Fonts are a kind of portfolio for the re-defined character information employed in these more advanced programming techniques.

A PROTO EXAMPLE

This example changes 4 characters: the 2, <, 7 and = into Prototype. The steps followed here will work with any characters you want to modify from the standard character set.

The first step is to turn off the keyboard and all input and output. Then move the characters from ROM (Read Only Memory) to RAM (Randam Access Memory - memory you can modify) so you can make the changes. Lines 1040 to 1260 do just that.

```
1040 REM >> TURN OFF I/O, BRING IN ROM
1050 POKE 56334,PEEK(56334) AND 254
1060 POKE 1,PEEK(1) AND 251
1070 :
1080 REM >> NO. OF BITS FROM CHAR. ROM
1090 FOR I=0 TO 63
1100 :
1110 REM >> RANGE OF BYTES/CHARACTER
1120 FOR J=0 TO 7
```

```
1130 :
1140 REM >> COPY BYTE OVER INTO RAM  <<
1150 POKE 12288+I*8+J,PEEK(53248+I*8+J)
1160 :
1170 REM >> NEXT BYTE OR CHARACTER   <<
1180 NEXT J,I
1190 :
1200 REM >> TURN ON I/O AND ROM OUT  <<
1210 POKE 1, PEEK(1)OR4
1220 POKE 56334, PEEK(56334) OR 1
1230 :
1240 REM >> RESET CHARACTER MEMORY   <<
1250 REM >>     POINTER TO 12288     <<
1260 POKE 53272,(PEEK(53272)AND240)+12
1270 :
```

Next, modify the character. Each one is stored in 8 bytes. That is the job of these lines.

```
1280 REM >> NESTED LOOPS STORE       <<
1290 REM >> BYTES PER CHARACTER      <<
1300 FOR CHAR = 60 TO 63
1310 FOR BYTE =  0 TO 7
1320 :
1330 REM >> READ RE-DEFINED DATA     <<
1340 READ NUM
1350 :
1360 REM >> STORE NEW DATA INTO CHAR. <
1370 POKE 12288+(8*CHAR)+BYTE, NUM
1380 :
1390 REM >> NEXT BYTE OR NEXT CHAR.  <<
1400 NEXT BYTE,CHAR
1410 :
```

This puts the redefined character on the screen. By pressing any key the characters will change back to the standard character set.

```
1420 REM >> PUT RE-DEFINED CHAR ON   <<
1430 PRINTCHR$(147)TAB(255)CHR$(60);
1440 REM >> USER PRESSES KEY TO SEE  <<
1450 PRINTCHR$(61)TAB(55)CHR$(62)CHR$(63)
1460 REM >>    NORMAL CHARACTERS     <<
```

```
147Ø GETA$:IF A$="" THEN GOTO 147Ø
148Ø :
149Ø REM >> RESET CHAR MEM POINTER    <<
15ØØ REM >>       TO NORMAL          <<
151Ø POKE 53272,21
152Ø :
```

This last section is the data necessary to change the standard characters into Proto.

```
153Ø REM >>      NEW CHAR. DATA      <<
154Ø DATA 7,7,3Ø,3Ø,127,127,31,31
155Ø DATA 224,224,12Ø,12Ø,225,225,248,248
156Ø DATA 24,24,24,24,24,24,1Ø2,1Ø2
157Ø DATA 24,24,24,24,24,24,1Ø2.1Ø2
158Ø END
```

Character Graphics is a powerful tool for programmers who want to customize programs. It gives added dimension to games, too.

ASCII and CHR$ CODES

PRINT	CHR$	PRINT	CHR$	PRINT	CHR$	PRINT	CHR$
	0	RED	28	8	56	T	84
	1	CURSOR RIGHT	29	9	57	U	85
	2	GREEN	30	:	58	V	86
	3	BLUE	31	;	59	W	87
	4	SPACE	32	<	60	X	88
WHITE	5	!	33	=	61	Y	89
	6	"	34	>	62	Z	90
	7	#	35	?	63	[	91
DISABLES SHIFT	CTRL 8	$	36	@	64	£	92
ENABLES SHIFT	CTRL 9	%	37	A	65	]	93
	10	&	38	B	66	↑	94
	11	'	39	C	67	←	95
	12	(	40	D	68		96
RETURN	13	)	41	E	69		97
SWITCH TO LOWER CASE	14	*	42	F	70		98
	15	+	43	G	71		99
	16	,	44	H	72		100
CURSOR DOWN	17	-	45	I	73		101
REVERSE ON	18	.	46	J	74		102
HOME	19	/	47	K	75		103
INSERT/ DELETE	20	0	48	L	76		104
	21	1	49	M	77		105
	22	2	50	N	78		106
	23	3	51	O	79		107
	24	4	52	P	80		108
	25	5	53	Q	81		109
	26	6	54	R	82		110
	27	7	55	S	83		111

PRINT	CHR$	PRINT	CHR$	PRINT	CHR$	PRINT	CHR$
	112		132	GREY 2	152		172
	113	f1	133	LT. GREEN	153		173
	114	f3	134	LT. BLUE	154		174
	115	f5	135	GREY 3	155		175
	116	f7	136	PURPLE	156		176
	117	f2	137	CURSOR LEFT	157		177
	118	f4	138	YELLOW	158		178
	119	f6	139	CYAN	159		179
	120	f8	140	REVERSE SPACE	160		180
	121	SHIFT RETURN	141		161		181
	122	SWITCH TO UPPER CASE	142		162		182
	123		143		163		183
	124	BLACK	144		164		184
	125	CURSOR UP	145		165		185
	126	REVERSE OFF	146		166		186
	127	CLEAR/ HOME	147		167		187
	128	INSERT DELETE	148		168		188
ORANGE	129	BROWN	149		169		189
	130	LT. RED	150		170		190
	131	GREY 1	151		171		191

CODES	192-223	SAME AS	96-127
CODES	224-254	SAME AS	160-190
CODE	255	SAME AS	126

SCREEN DISPLAY CODES

SET1	SET2	POKE
@		0
A	a	1
B	b	2
C	c	3
D	d	4
E	e	5
F	f	6
G	g	7
H	h	8
I	i	9
J	j	10
K	k	11
L	l	12
M	m	13
N	n	14
O	o	15
P	p	16
Q	q	17
R	r	18
S	s	19
T	t	20
U	u	21
V	v	22
W	w	23
X	x	24
Y	y	25
Z	z	26
[		27

SET1	SET2	POKE
£		28
]		29
↑		30
←		31
SPACE		32
!		33
"		34
#		35
$		36
%		37
&		38
'		39
(		40
)		41
*		42
+		43
,		44
–		45
.		46
/		47
0		48
1		49
2		50
3		51
4		52
5		53
6		54
7		55

SET1	SET2	POKE
8		56
9		57
:		58
;		59
<		60
=		61
>		62
?		63
		64
♠	A	65
	B	66
	C	67
	D	68
	E	69
	F	70
	G	71
	H	72
	I	73
	J	74
	K	75
	L	76
	M	77
	N	78
	O	79
	P	80
●	Q	81
	R	82
♥	S	83

SET1	SET2	POKE	SET1	SET2	POKE	SET1	SET2	POKE
	T	84			99			114
	U	85			100			115
	V	86			101			116
	W	87			102			117
	X	88			103			118
	Y	89			104			119
	Z	90			105			120
		91			106			121
		92			107			122
		93			108			123
		94			109			124
		95			110			125
SPACE		96			111			126
		97			112			127
		98			113			

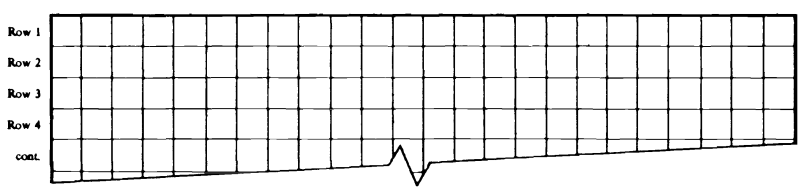
HOW TO CREATE SPRITES

Sprites are special movable objects you can design yourself. After the design work - and it *is* a bit of work - the computer will help you easily move them, check for collisions with other sprites, with text or other graphics, and even keep track of priorities, in other words, which sprites will pass in front of the others.

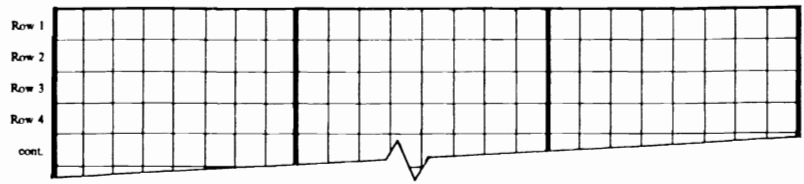
DESIGN THE SPRITE

A sprite is designed using 63 bytes of memory - 21 rows with 3 bytes in each row. With eight positions in each byte, you have 24 (3 times 8) spaces in each row.

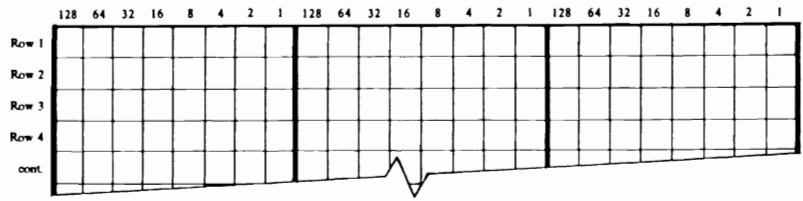
Mark off a 24 x 21 section of graph paper. This is the grid needed to represent your sprite.



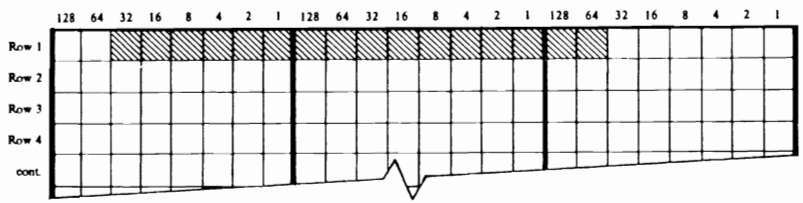
Now draw lines dividing the grid into 3 equal columns of 8 squares each.



These markings are used later to calculate the byte values needed for the data statements. The first row and first column looks like this:



Of course, all the 62 other pieces of the graph look like this, too. To turn on the parts of the sprites marked, you would write a data statement for each of the 63 bytes needed. To have a first row that looked like this



would need a DATA statement like this:

DATA 63,255,192

Finish designing your sprite then calculate the value of the 63 bytes needed for each sprite.

STORING THE SPRITE IN MEMORY

After building the sprite and calculating the DATA statements, you must POKE the data into memory. You can put sprites safely in the locations shown on the following chart. Other places can be used, too, but require that you reserve space in the computer. See locations 55 and 56 for a routine to do this.

LOCATION (L)	DATA POINTER
832	13
896	14
960	15
12288	192
12352	193
12416	194
12480	195
12544	196
12608	197
12672	198

If you use a cassette to store programs, then remember location 13, 14 and 15 are used by the cassette for temporary data storage when loading or saving programs. The two operations will cause the sprite data stored there to be changed. The sprites should be redefined by reading the spite data again and rePOKEing it into memory.

Use a FOR/NEXT loop to POKE in the data; for example;

```
100 REM ** L = THE START OF SRITE DATA
110 L=12288
120 FOR I = 0 TO 62
130 REM ** SD = SPRITE DATA ELEMENT
140 READ SD
150 POKE L+I,SD
160 NEXT
170 REM ** SAMPLE DATA
180 DATA 63,255,192,35,170,85,120, ETC
190 :
200 :
```

SETTING THE SPRITE POINTERS

Each sprite has a location that tells the computer where to go to get the data necessary to put the sprite on the screen. The sprite pointers are locations 2040 to 2047 and they point for sprites 0 to 7, respectively. If you POKE for information for sprite 0 in location 960 then:

POKE 2040,15

See the chart above to find the right data pointer. If you don't want to use the chart, just divide the location address by 64 to get the pointer number.

960/64 = 15

If you want 4 sprites to look exactly the same, just POKE the data once and set the pointers for the 4 sprites to the same area. This makes sprites 0, 1, 2 and 3 look exactly alike.

```
100 FOR I = 2040 TO 2043
110 POKE I,15
120 NEXT I
```

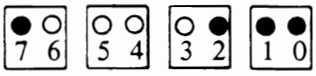
CHOOSING THE COLOR

Once you have the sprite data in memory then decide on the color the sprite will be. Each sprite has a default color, that is, a color already set by the computer. If you want, choose a color from this chart and POKE it into the sprite color locations (55287-55294).

Sprite	Default Colors
0	White
1	Red
2	Cyan
3	Purple
4	Green
5	Blue
6	Yellow
7	Grey 2

MULTICOLOR SPRITES

A mulitcolored sprite is a more advanced technique. In this mode, the computer reads your sprite data in a slightly different way, in bit pairs. Two bits are read as one piece of data.



Bit pair 'A' tells the computer to get the color for the area from the sprite color register (from 55287-55294, depending on the sprite you're working on).

B is screen color, so it will look like a blank space.

C is the color value POKEd into location 53285, multicolor register #0.

D is the color in multicolor register #1, 53286.

The program below switches a sprite from standard mode to multicolor mode.

VARIATIONS ON A THEME OF PROTO

```
1060 REM >>>>>    INITIALIZTION     <<<<<
1070 REM >> CLEAR SCREEN/BLUE CURSOR <<
1080 PRINT CHR$(147) :POKE 214, 6
1090 GO TO 1390
1100 :
1110 REM >>  SUBROUTINES GO ON TOP    <<
1120 REM >> RESET SWITCH SUBROUTINE   <<
1130 IF N=1 THEN N=0 :RETURN
1140 :  N=1 :RETURN
1150 :
1160 REM >>     EXPAND SUBROUTINE      <<
1170 REM >> CHECK FOR 'F1' KEY FIRST <<
1180 IF PEEK(197)<>4 THEN RETURN
1190 POKE 53265,11 :REM BLANK SCREEN
1200 POKE VIC+23,N :POKE VIC+29,N
```

```
1210 POKE 53265,27 :REM SCREEN ON
1220 GOSUB 1130    :REM SWITCH SUBR.
1230 RETURN
1240 :
1250 REM >> SINGLE/MULTI-COLOR SUBR. <<
1260 REM >> CHECK FOR 'F2' KEY FIRST <<
1270 IF PEEK(197) <>5 THEN RETURN
1280 POKE 53265,11 :REM BLANK SCREEN
1290 POKE VIC+28,N :REM M.C.M. SPR 0
1300 POKE 53265,27 :REM SCREEN ON
1310 GOSUB 1130    :REM SWITCH SUBR.
1320 RETURN
1330 :
1340 REM >> HEADER INFO. SUBROUTINE  <<
1350 READ HD$,HR$ :PRINT HD$" "HR$
1360 RETURN
1370 :
1380 REM >>      PRINT 1ST LINE       <<
1390 PRINTCHR$(17)CHR$(29);
1400 GOSUB 1350
1410 :
1420 REM >>      PRINT 2ND LINE       <<
1430 FORI=1TO3:PRINTCHR$(17)CHR$(29);
1440 NEXT I
1450 GOSUB 1350
1460 :
1470 REM >>   LAST LINE TO GO ON      <<
1480 FORI=1TO3:PRINTCHR$(17)CHR$(29);
1490 NEXT I
1500 FORI=1TO5:PRINTCHR$(29);:NEXT I
1510 PRINT"HIT ANY KEY TO CONTINUE"
1520 GETA$:IF A$="" THEN 1520
1530 PRINT CHR$(147):N=0
1540 :
1550 REM >>   SET SPRITE CONSTANTS   <<
1560 VIC = 53248 :REM BASE OF VIC CHIP
1570 TBL = 13    :REM NUMBER OF BLOCKS
1580 MEM = 64*TBL:REM LOC. OF SPRITES
1590 :
1600 REM >> TURN OFF SCREEN          <<
1610 REM POKE 53265, 11
1620 :
```

```
1630 REM >> SET BACK. AND BORDER COLORS
1640 POKE 53280, 1 :POKE 53281, 1
1650 :
1660 REM >> SET SPRITE DATA POINTER  <<
1670 POKE 2040, TBL
1680 :
1690 REM >> TURN ON SPRITE # 0       <<
1700 POKE VIC+21, 1
1710 :
1720 REM >> SET SPRITE 0'S POSITION  <<
1730 POKE VIC, 160 :POKE VIC+1, 120
1740 :
1750 REM >> SET SPRITE COLORS        <<
1760 POKE VIC+37, 2:REM M.C.M. REG.#1
1770 POKE VIC+39, 4:REM SPRITE #0 COLOR
1780 :
1790 REM >> OPTION TO EXPAND SPRITE #0
1800 POKE VIC+23, N :REM --> EXPAND X
1810 POKE VIC+29, N :REM --> EXPAND Y
1820 :
1830 REM >> SELECT M.C.M. FOR SPRITE #0
1840 POKE VIC+28, N
1850 :
1860 REM >> READ SPRITE DATA WITH A  <<
1870 REM >>    (10101010) BIT MASK   <<
1880 FOR  I = 0 TO 23
1890 READ P :POKE(MEM+I), P AND 170
1900 NEXT I
1920 REM >> READ DATA W/(01010101) MASK
1930 FOR  J = 24 TO 55
1940 READ P :POKE(MEM+J), P AND 85
1950 NEXT J
1970 REM >> READ DATA W/(10101010) MASK
1980 FOR  K = 56 TO 62
1990 READ P :POKE(MEM+K), P AND 170
2000 NEXT K
2010 :
2020 REM >> TURN SCREEN BACK ON      <<
2030 POKE 53265, 27
2040 :
2050 REM >> SET SWITCH/WHITE CURSOR  <<
2060 N=1 :POKE 646, 0
2070 :
```

```
2080 REM >> FLASH 'PROTO', THEN WAIT <<
2090 PRINT CHR$(147) :REM CLEAR SCREEN
2100 FORI=1TO10 :PRINTCHR$(17)CHR$(29);
2110 NEXT I
2120 PRINT"PROTO"  :FOR W=0TO50:NEXT W
2130 PRINTCHR$(147):FOR W=0TO50:NEXT W
2140 :
2150 REM >> CHECK SUBROUTINES          <<
2160 GOSUB 1180:REM EXPAND SPRITE 0 X&Y
2170 GOSUB 1270:REM SINGLE/MULTI-COLOR
2180 GO TO 2090
2190 :
2200 REM >>>>> HEADER INFO. DATA  <<<<<
2210 DATA HIT 'F1' TO EXPAND OR
2220 DATA UN-EXPAND SPRITE
2230 DATA HIT 'F3' TO SEE MULTI COLOR
2240 DATA SPRITE
2260 REM >>>>>    PROTO'S DATA    <<<<<
2270 DATA 0,60,0,1,255,128,7,255,224,31
2275 DATA 231,252,63,231,254,15,255,248
2276 DATA 7,255,240,0,255,128,0,255,128
2277 DATA 0,193,128,0,193,128,0,193,128
2278 DATA 0,193,128,0,193,128,0,193,128
2279 DATA 0,193,128,0,193,128,0,193,128
2280 DATA 0,193,128,3,247,224,3,54,96
```

The other parts of sprite programming such as priorities and collision detection are rather straight forward. Just refer to the necessary location for more details on their use. With priorities, just remember that they are set automatically with sprite 0 having the highest priority. Sprite 0 will pass in front of sprites 1 to 7. Sprite 1 will pass behind #0 and in front of 2 to 7. Sprite #2 will pass behind #0 and #1 and in front of 3 to 7. You can see the pattern. All sprites have priority over background data. If you want to change any particular sprite's priority with respect to the background and make the sprites pass behind the text, the proper bit in the sprite priority register must be set (see loc. 53275).

Collision detection is no harder to use. The sprite collision registers are "read only" registers whose bits change to 1 whenever a collision has occurred (see locs. 53278 and 53279). It is easy to test for the collision, but that test must be done strategically in your code such as in a main motion loop. Collisions can be determined by one line of code.

FOR SPRITE-TO-SPRITE COLLISIONS USE:

IF PEEK(V+30) AND X THEN [ACTION]

FOR SPRITE-TO-BACKGROUND COLLISIONS USE:

IF PEEK(V+31) AND X THEN [ACTION]

where 'X' is the decimal value of the bit in the collision register.

SPRITE #	BIT	VALUE
0	0	1
1	1	2
3	2	4
3	3	8
4	4	16
5	5	32
6	6	64
7	7	128

In a very brief outline form, here's how to create a sprite. Part I (A) is true for all programs you write, the remainder deals with only the sprites.

Those marked with an * are optional depending on your level of ability and how you want the sprites to look. You can create and move sprites without them.

SPRITE ALGORITHM OUTLINE

1. PROGRAM INITIALIZATION
   - A) Declare Program Parameters
     1. Dimension arrays
        - a. integer
        - b. string
        - c. real
     2. Set constants and variables (same items as under arrays)
   - B) Declare Sprite parameters
     1. Set constants and variables pertinant to sprites
        - a. VIC=53248 => start address of the VIC-II

     *2. Turn off screen (POKE VIC+17,11)

     3. Store sprite data
        - a. set data pointers (locs. 2040 - 2047)
        - b. use FOR/NEXT loop to read in data

     4. Enable Sprites (VIC+21)
     5. Set sprite color registers
        - a. single color sprites (POKE locs. VIC+39 + VIC+46)
        - *b. multi-color sprites (POKE locs. VIC+37 + VIC+38)

     *6. Enable sprite color mode
        - a. Turn bit four on in multi-color register (VIC+22)

     *7. Select expanded sprites (VIC+23 and VIC+29)
     *8. Turn screen back on (POKE VIC+17,27)
2. MAIN PROGRAM LOOP
   - A) Set sprite motion FOR/NEXT loop
     1. PEEK at sprite positions (VIC + VIC+16)
        - a. call them X and Y

     2. Determine how far you want to move
        - a. call the amount of motion DX and DY

     3. Determining direction of motion

     *4. Monitor sprite collision registers (DX=−DX, DY=−DY)
        - a. sprite-sprite collision
        - b. sprite-data collision

     5. Check X positions of sprites whether to POKE MSB in VIC+16
        - a. Check position border constraints (see sprite pos. locs.)

     6. Add offsets to original position: X=X+DX, Y=Y+DY
     7. Exit position update loop

B) Evaluate multiple sprite displays

1. POKE VIC+16 with the value determined by the sprites whose X position is greater than 256.

*2. Check raster register for special interrupt conditions

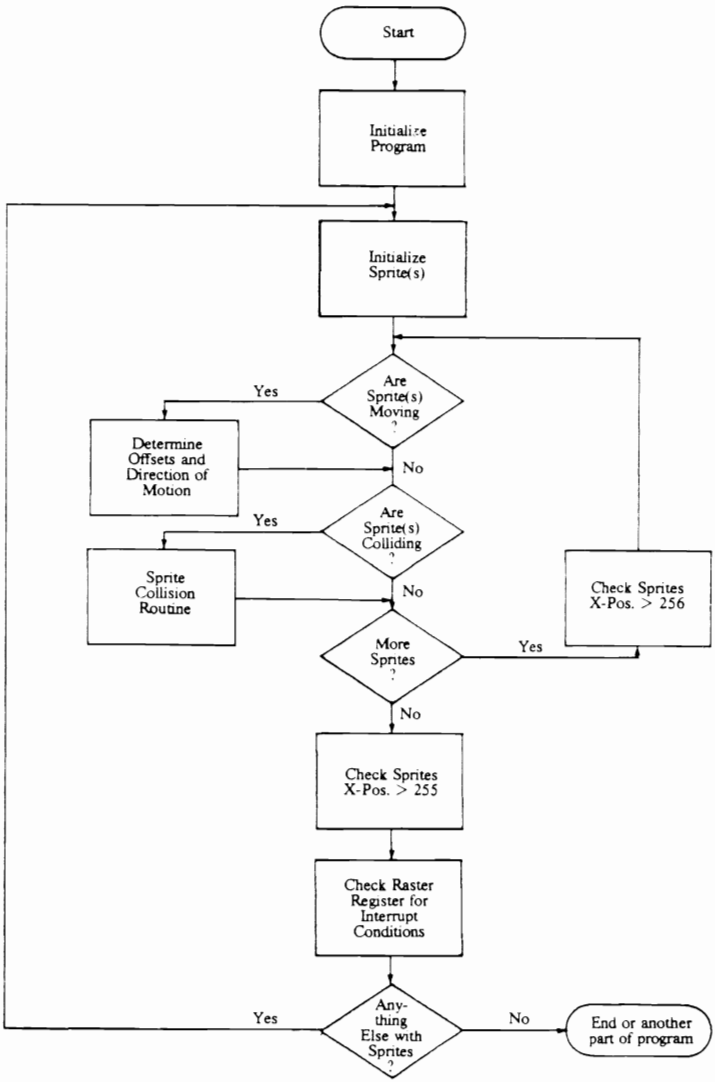
   a. mix-moded displays (character w/ bit-map mode)

   b. moving more than eight sprites

   c. light-pen

3. POKE updated position (X,Y) back into position registers

4. Go back to II-A) if y



This last program contains a machine language routine to move sprites quickly across the screen. The speed range is from 1 to 5. Press RUN/STOP to end the program.

```
1000 REM ********************************
1010 REM *   THEME ON A VARIATION OF    *
1020 REM *           P R O T O          *
1030 REM * MOVING PROTO WITH JOYSTICK *
1040 REM * ALGORYTHM BY SHELDON LEEMON*
1050 REM ********************************
1060 :
1070 PRINTCHR$(147);CHR$(5):SP=53248
1080 INPUT"SPEED (1-5) ";S:GOTO 1190
1090 :
1100 REM >>>>>    SELECT SPEED    <<<<<
1110 ON S GOTO 1120,1130,1140,1150,1160
1120 SYS(49409) :GOTO 1120
1130 SYS(49406) :GOTO 1130
1140 SYS(49403) :GOTO 1140
1150 SYS(49400) :GOTO 1150
1160 SYS(49413) :GOTO 1160
1170 :
1180 REM >>>>> STORE SPRITE DATA  <<<<<
1190 FORI=871TO895:POKE I,0:NEXT
1200 FORI=832TO894:READ A:POKE I,A:NEXT
1210 :
1220 REM >>>>> SET SPRITE PARAMETERS <<
1230 POKE SP+21,1 :POKE 2040,13
1240 POKE SP+39,6 :POKE SP+29,1
1250 POKE SP,160  :POKE SP+1,100
1260 POKE SP+32,0 :POKE SP+33,0
1270 PRINT CHR$(147)
1280 :
1290 REM >>>>>   PLACE STARS IN   <<<<<
1300 FORI=1TO50
1310 : POKE 1024+INT(RND(0)*1000),46
1320 NEXT
1330 :
1340 REM >>>>>    PROTO'S DATA    <<<<<
1350 DATA 0,60,0,1,255,128,7,255,224,31
1360 DATA 231,252,63,231,254,15,255,248
1370 DATA 7,255,240,0,255,128,0,255,128
1380 DATA 0,193,128,0,193,128,0,193,128
```

```
1390 DATA 0,193,128,0,193,128,0,193,128
1400 DATA 0,193,128,0,193,128,0,193,128
1410 DATA 0,193,128,3,247,224,3,54,96
1420 :
1430 REM >>>>>         STORE          <<<<<
1440 REM >>>>>  MACHINE LANGUAGE     <<<<<
1450 REM >>>>>       ROUTINE          <<<<<
1460 FORI=1TO101
1470 : READ A:POKE 49151+I,A
1480 NEXT
1490 :
1500 FORI=1TO19
1510 : READ A:POKE 49399+I,A
1520 NEXT
1530 :
1540 REM >>>>> RETURN TO UPDATE      <<<<<
1550 REM >>>>>      POSITION         <<<<<
1560 GO TO 1110
1570 :
1580 REM >>>>> MACHINE LANG. DATA <<<<<
1590 DATA 173,1,220,74,176,3,206,1,208
1600 DATA 74,176,3,238,1,208,74,176,38
1610 DATA 173,0,208,208,15,173,16,208
1620 DATA 41,1,240,12,173,16,208,41,254
1630 DATA 141,16,208,206,0,208,96,173
1640 DATA 16,208,9,1,162,63,141,16,208
1650 DATA 142,0,208,96,74,176,32,238
1660 DATA 0,208,240,28,173,16,208,41,1
1670 DATA 240,20,169,64,205,0,208,208
1680 DATA 13,173,16,208,41,254,162,0
1690 DATA 141,16,208,142,0,208,96,173
1700 DATA 16,208,9,1,141,16,208,96
1710 DATA 32,0,192,32,0,192,32,0,192,32
1720 DATA 0,192,96,32,0,192,76,5,193
```

COMPOSING MUSIC

Rattle off a quick tune in your head. Now think for a moment all the variations of pitch, volumn and tonal qualities that go into just a few bars of it. If you pick up the manuscript of a fully scored symphony, you may find overwhelming amounts of description for all the sounds that each instrument is responsible for. Little wonder how much training a modern composer or conducter must have to to attain mastery over musical expression.

With the advent of electronic synthesizers in the mid-1970's, keyboard musicians were introduced to new vista of musical performance with a wide range of dynamics, phrasing and articulation. The resulting explosion of creativity has lead to altogether new musical forms and ideas. And now in the 80's this has taken another step with the introduction of musical synthesis in home computers.

SOUND PROGRAMMING TECHNIQUES

The nice thing about programming sound on the Commodore 64 is that all its given features can be programmed in BASIC up to three voices. The 3 voices are just like having 3 musicians ready to make any sound you want. The only difficulty involved in sound programming consists of closely watching which values have been POKEd into the sound registers.

Needless to say, BE PATIENT. You will find after sound programming experimentation that even your mistakes may produce useful sound.

Let's get to some basic techniques. For practical purposes, there are only two ways to provide the information you need to make sounds. You can generate the sounds by POKEing in numbers one at a time, or put this information in data statements to be read as the program runs.

Before a sound can be made, be sure to do these things first.

1) Clear the SID chip to get rid of any unwanted values in the registers. A FOR/NEXT loop will do the job.

```
100 REM - SOUND CHIP STARTING ADDRESS.
110 SID=54272
120 REM - CLEAR THE REGISTERS.
130 FOR I = 0 TO 28
140 POKE SID+I,0
150 NEXT
```

2) Turn on the volume control on the chip.

```
160 REM   SET THE HIGHEST VOLUME.
170 POKESID+24,15
```

is the highest volume.

3) Set the frequency you want. Check the note table at the end for the values you need.

```
180 REM - SET HIGH BYTE AND LOW BYTE
190 REM - TO PRODUCE MIDDLE C IN VOICE
200 REM - ONE.
210 POKESID,34
220 POKESID+1,75
```

These are the values for C in the 5th octave, middle C.

4) Set the Attack, Decay, Sustain and Release values.

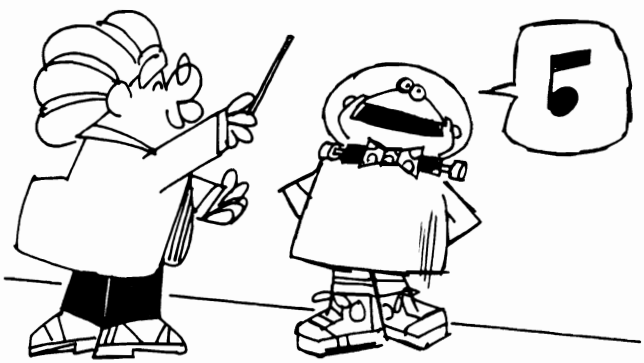
```
230 REM - SET THE ATTACK/DECAY RATE.
240 POKESID+5,16
250 REM - SET THE SUSTAIN/RELEASE RATE.
260 POKESID+6,249
```

With those steps taken care of, you can now work with filtering, modulation and syncronization or just enter all the above lines and turn a specific waveform on.

```
270 REM - CHOOSE THE SAWTOOTH WAVEFORM
280 REM - AND TURN IT ON.
290 POKESID+4,33
```

and off

```
300 REM - START THE RELEASE CYCLE
310 POKESID+4,32
```



OUTLINE FOR SINGLE VOICE

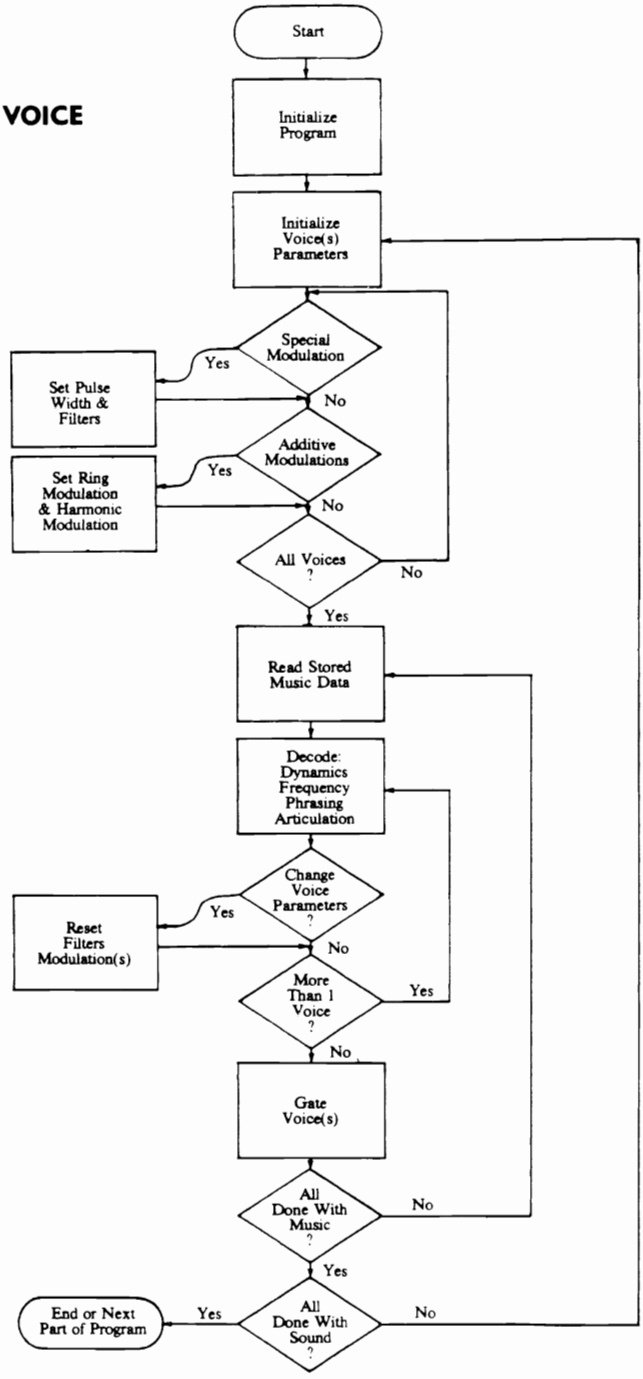
This short outline contains all the steps necessary to produce a sound with a single voice. Part I (A) is true for all programs. You write,

I. PROGRAM INITIALIZATION
- A) Declare Program Parameters
  1. Dimension arrays
     a. integer
     b. string
     c. real
  2. Set constants and variables for the other parts of your program.
- B) Declare Sound Parameters
  1. Assign constants and variables pertinent to sound
     a. SID=54272, the start address of SID chip. See location 54272 for an example of using this method.
  2. Clear the sound chip
     a. use FOR/NEXT to POKE zero into the SID chip registers (SID to SID+24)

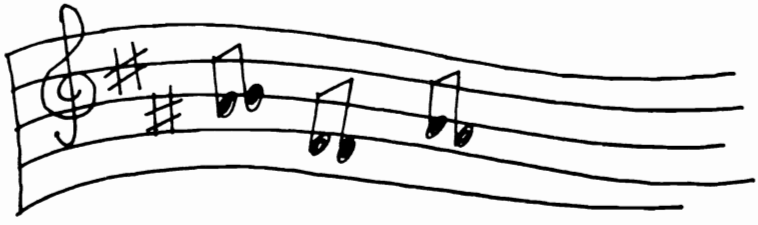
II. MAIN PROGRAM LOOP TO PLAY NOTES
- A) FOR/NEXT loop for voice production
  1. Define functions for any one or all of the voice parameters
     a. Define frequency (SID + voice frequency used)
     b. Define waveform (SID + voice control register)
     c. Define A/D/S/R (SID + A/D/S/R registers)
     d. Define filter (SID + filter control register)
     e. Define filter (SID + filter control register bits 0-3)
  2. Define sound duration
  3. GATE ON voice control bit for specified duration
  4. GATE OFF voice control bit for specified duration (rest)
- B) FOR/NEXT loop for voice production stored in DATA statements
  1. Store above voice parameters in DATA statements
     a. Store one or more parameters individually or,
     b. consolidate more than 1 parameter per given DATA element.
  2. READ voice information into voice parameter variables
  3. POKE voice information into respective voice parameter register

III. DATA STATEMENTS OR NEXT PART OF PROGRAM

VOICE

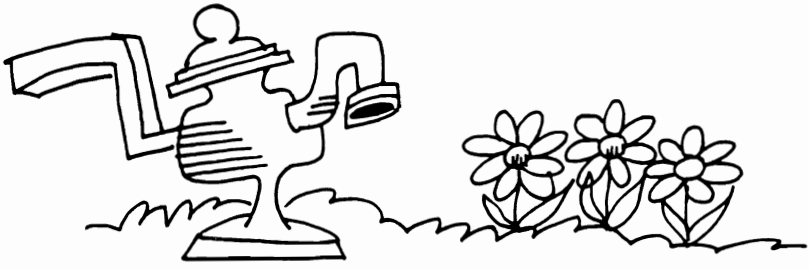

Start
Initialize Program
Initialize Voice(s) Parameters
Special Modulation
Yes
Set Pulse Width & Filters
No
Additive Modulations
Yes
Set Ring Modulation & Harmonic Modulation
No
All Voices ?
No
Yes
Read Stored Music Data
Decode: Dynamics Frequency Phrasing Articulation
Change Voice Parameters ?
Yes
Reset Filters Modulation(s)
No
More Than 1 Voice ?
Yes
No
Gate Voice(s)
All Done With Music ?
No
Yes
End or Next Part of Program
Yes
All Done With Sound ?
No

The programs below are samples of single voice programming. Play with these as much as possible, POKEing different values into the sound registers. Even your mistakes will lead you down new paths.



```
100 REM ************************************
110 REM * SOUND EXAMPLE #1    SCALES     *
120 REM ************************************
130 :
140 SID=54272:REM --> START OF SOUND
150 :
160 REM >>>>> CLEAR SOUND CHIP  <<<<<
170 FOR S=SIDTOSID+24:POKE S,0:NEXT S
180 :
190 REM >>>>> SET VOICE VOLUME  <<<<<
200 POKE SID+24,79:REM --> MAXIMUM
210 :
220 REM >>>>> SET A/D/S/R CYCLE <<<<<
230 POKE SID + 5,9: REM SID + 6, 65
240 :
250 REM >>>>> READ FREQ. DATA   <<<<<
260 READ HF, LF :REM HI FREQ.,LOW FREQ.
270 :
280 REM >>>>>  CHECK FOR END    <<<<<
290 IF HF < 0 THEN POKE SID+4,0: END
300 :
310 REM >>>>>  POKE IN FREQ.    <<<<<
320 POKE SID, LF: POKE SID+1, HF
330 :
340 REM >>>>>  GATE VOICE 1 W/  <<<<<
350 REM >>>>> SAWTOOTH WAVEFORM <<<<<
360 POKE SID + 4, 33
370 :
380 REM >>>>>  DELAY LOOP FOR   <<<<<
390 REM >>>>>  NOTE DURATION    <<<<<
```

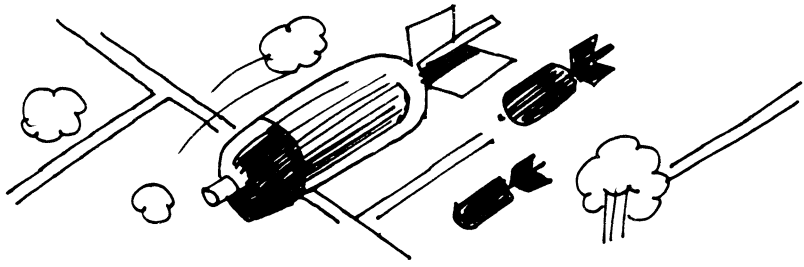
```
400 FOR DUR = 1 TO 250: NEXT DUR
410 :
420 REM >>>>>  TURN OF VOICE 1  <<<<<
430 POKE SID + 4, 32
440 :
450 REM >>>>>   GET NEXT NOTE   <<<<<
460 GOTO 230
470 :
480 REM >>>>>      MUSIC DATA    <<<<<
490 DATA 34,75,38,126,43,52,45,198,51
500 DATA 97,57,172,64,188,68,149
510 DATA 76,252,86,105,91,140,102,194
520 DATA 115,88,129,120,137,43,-1,-1
```



This example uses a high-pass filter and volume manipulation to produce the sound of an old water pump.

```
100 REM ***********************************
110 REM * SOUND EXAMPLE #2     PUMP       *
120 REM ***********************************
130 :
140 SID=54272:REM --> START OF SOUND
150 :
160 REM >>>>> CLEAR SOUND CHIP  <<<<<
170 FOR S=SIDTOSID+24:POKE S,0:NEXT S
180 :
190 REM >>>>> SET VOICE VOLUME  <<<<<
195 REM >>>>>& HIGH PASS FILTER <<<<<
200 POKE SID+24,79:REM --> MAX. VOL.
210 :
220 REM >>>>> SET A/D/S/R CYCLE <<<<<
230 POKE SID + 5,148: POKE SID + 6, 26
270 :
310 REM >>>>>  POKE IN FREQ.    <<<<<
320 POKE SID, 240: POKE SID+1, 33
330 :
340 REM >>>>>  GATE VOICE 1 W/  <<<<<
350 REM >>>>>  NOISE WAVEFORM   <<<<<
360 POKE SID + 4, 131
370 :
380 REM >>>>>  DECREASE VOLUME  <<<<<
390 FOR VOL=15 TO 0 STEP -1
400 POKE SID + 24, VOL
410 :
420 REM >>>>> SUSTAIN VOL WITH  <<<<<
430 REM >>>>>    A DELAY LOOP    <<<<<
```

```
440 FOR DUR = 0 TO 25 :NEXT DUR
450 :
460 REM >>>>>  NEXT VOLUME SET  <<<<<
470 NEXT VOL
480 :
490 REM >>>>> TURN OFF VOICE 1  <<<<<
500 POKE SID + 4, 0
510 :
520 REM >>>>> RETURN TO REPEAT  <<<<<
530 GO TO 360
```



A low pass filter plus the interaction of voices 2 and 3 for the falling bomb effect - complete with explosion.

```
100 REM ********************************
110 REM * SOUND EXAMPLE 3     BOMB     *
120 REM ********************************
130 :
140 SID=54272:REM ---> START OF SOUND
150 :
160 REM >>>>> CLEAR SOUND CHIP <<<<<
170 FOR S=SIDTOSID+24:POKE S,0:NEXT S
180 :
190 REM >>>>> VOICE 3 A/D/S/R  <<<<<
200 POKE SID+19,20:POKE SID+20,20
210 :
220 REM >>>>> FILTER CUTOFF FREQ. <<
230 POKE SID+21,1:POKE SID+22, 110
240 :
250 REM >>>>> SET VOICE 3 FILTER <<<
260 POKE SID+23, 4
270 :
280 REM >>>>> SET VOICE 3 VOLUME <<<
290 REM >>>>> W/ LOW-PASS FILTER <<<
300 REM >>>>>       & CUTOFF     <<<
310 POKE SID+24, 159
320 :
330 REM >>>>> GATE VOICE 3 WITH  <<<
340 REM >>>>> TRIANGLE WAVEFORM  <<<
350 POKE SID + 18, 17
360 :
370 REM >>>>> SLIDE DOWN VOICE 3 <<<
```

```
380 REM >>>>>       FREQUENCY      <<<
390 FOR HF=230TO20STEP-1
400 FOR LF=255TO0 STEP-100
410 POKE SID+14,LF: POKE SID+15,HF
420 NEXT LF
430 IF LF<0 THEN NEXT HF
440 :
450 REM >>>>> TURN OFF VOICE 3 <<<<<
460 POKE SID+18,16:REM --> RELEASE
470 :
480 REM >>>>>  PAUSE A MOMENT  <<<<<
490 FOR DUR = 1 TO 100:NEXT DUR
500 :
510 REM >>>>> SET UP VOICE 2 FOR <<<
520 REM >>>>> HARD SYNC. WITH 3  <<<
530 POKE SID+8,5:REM     ----> FREQ.
540 POKE SID+12,15:REM  ----> A/D/
550 POKE SID+13,255:REM ----> S/R CYCLE
560 :
570 REM >>>>> GATE VOICES 2 & 3 <<<<
580 POKE SID+11, 131: POKE SID+18, 132
590 :
600 REM >>>>> SOUND AN EXPLOSION <<<
610 REM >>>>> W/ VOICE 3 DECREASE<<<
620 REM >>>>>       OF VOLUME      <<<
630 FOR L=10TO0 STEP -.01
640 POKE SID + 24, L
650 NEXT L
660 :
670 REM >>>>> TURN OFF VOICE 2   <<<
680 POKE SID+11,0:REM --> RELEASE
690 END
```

An example of ring modulation using voices 1 and 3 produces this sound.

```
100 REM ********************************
110 REM * SOUND EXAMPLE 4 BUSY SIGNAL*
120 REM ********************************
130 :
140 SID=54272:REM --> START OF SOUND
150 :
160 REM >>>>> CLEAR SOUND CHIP <<<<<
170 FOR S=SIDTOSID+24:POKE S,0:NEXT S
180 :
190 REM >>>>> SET FREQ. VOICE 1 <<<<<
200 POKE SID+1, 75
210 :
220 REM >>>>> VOICE 1 A/D/S/R  <<<<<
230 POKE SID+5,20: POKE SID+6,200
240 :
250 REM >>>>> SET FREQ. VOICE 3 <<<<<
260 POKE SID+15, 36
270 :
280 REM >>>>>     SET VOLUME     <<<<<
290 POKE SID+24,15:REM --> MAXIMUM
300 :
310 REM >>>>>  SET DELAY LOOP  <<<<<
320 FOR DUR=1TO100:NEXT DUR
330 :
340 REM >>>>> GATE VOICE 1 W/  <<<<<
350 REM >>>>> TRIANGLE WAVEFORM <<<<<
360 REM >>>>> & RING MODULATION <<<<<
370 POKE SID+4, 21
380 :
390 REM >>>>> ANOTHER DELAY LOOP <<<<<
400 FOR DUR=1TO100:NEXT DUR
410 :
420 REM >>>>> TURN OFF VOICE 1 <<<<<
430 POKE SID + 4, 0:REM --> RELEASE
440 :
450 REM >>>>> RETURN TO REPEAT <<<<<
460 GO TO 290
```

Two voices, one siren with hard syncronization.

```
100 REM ********************************
110 REM * SOUND EXAMPLE 5     SIREN    *
120 REM ********************************
130 :
140 SID=54272:REM --> START OF SOUND
150 :
160 REM >>>>> CLEAR SOUND CHIP <<<<<
170 FOR S=SIDTOSID+24:POKE S,0:NEXT S
180 :
190 REM >>>>> VOICE 1 A/D/S/R  <<<<<
200 POKE SID+5,100: POKE SID+6,100
210 :
220 REM >>>>>     SET VOLUME     <<<<<
230 POKE SID+24,15:REM --> MAXIMUM
240 :
250 REM >>>>> SET FREQ. VOICE 3 <<<<<
260 POKE SID+15,30
270 :
280 REM >>>>> GATE VOICE 1 W/   <<<<<
290 REM >>>>> TRIANGLE WAVEFORM <<<<<
300 REM >>>>>     & HARD SYNC.  <<<<<
310 POKE SID+4, 19
320 :
330 REM >>>>> INCREASE VOICES 1 <<<<<
340 REM >>>>>  & 3 FREQUENCIES  <<<<<
350 FOR HF = 30 TO 40
360 FOR LF = 0  TO 255 STEP 20
370 :
380 POKE SID+1, HF: POKE SID,   LF
390 POKE SID+15,HF: POKE SID+14,LF
400 :
```

```
410 NEXT LF,HF
420 :
430 REM >>>>>  SET DELAY LOOP  <<<<<
440 FOR DUR=1TO250:NEXT DUR
450 :
460 REM >>>> DECREASE VOICES 1 <<<<<
470 REM >>>>  & 3 FREQUENCIES  <<<<<
480 FOR HF = 40 TO 30 STEP-1
490 FOR LF = 255TO 0  STEP-20
500 :
510 POKE SID+1, HF:POKE SID,LF
520 POKE SID+15,HF:POKE SID+14,LF
530 :
540 NEXT LF,HF
550 :
560 REM >>>>>  SET DELAY LOOP  <<<<<
570 FOR DUR=1TO250:NEXT DUR
580 :
590 REM >>>>> RETURN TO REPEAT <<<<<
600 GO TO 350
```

The 'drip' uses a pulse waveform and ring modulation.

```
100 REM ******************************
110 REM * SOUND EXAMPLE 6  DRIPPING *
120 REM ******************************
130 :
140 SID=54272:REM --> START OF SOUND
150 :
160 REM >>>>> SET FOR/NEXT STEP <<<<<
170 SP = 0.1
180 :
190 REM >>>>> CLEAR SOUND CHIP  <<<<<
200 FOR S=SIDTOSID+24:POKE S,0:NEXT S
210 :
220 REM >>>>> SET VARIABLE DELAY <<<<
230 FOR DUR=1TO1000 STEP SP+0.1
240 :
250 REM >>>>> SET VOICE 1 FREQ. <<<<<
260 POKE SID+1, RND(1)*200+1
270 :
280 REM >>>>>  VOICE 1 A/D/S/R  <<<<<
290 POKE SID+5,10: POKE SID+6,255
300 :
310 REM >>>>> SET VOICE 3 FREQ. <<<<<
320 POKE SID+15, RND(1)*200+1
330 :
340 REM >>>>>     SET VOLUME      <<<<<
350 POKE SID+24,15:REM --> MAXIMUM
360 :
370 REM >>>>>  GATE VOICE 1 W/   <<<<<
380 REM >>>>>  SQUARE WAVEFORM   <<<<<
390 REM >>>>>     & RING MOD.     <<<<<
400 POKE SID+4, 69
```

```
410 :
420 REM >>>>> SET VARIABLE DELAY <<<<
430 FOR T=1TO1000/DUR: NEXT T
440 :
450 REM >>>>>  GATE VOICE 1 W/  <<<<<
460 REM >>>>> TRIANGLE WAVEFORM <<<<<
470 REM >>>>>     & RING MOD.    <<<<<
480 POKE SID+4, 21
490 :
500 REM >>>>> INCREMENT STEP VALUE <<
510 SP = SP + 0.9
520 :
530 REM >>>>>  NEXT DURATION    <<<<<
540 NEXT DUR
```

If you put this airplane example with the bomb you're halfway to some game sound effects.

```
100 REM ******************************
110 REM * SOUND EXAMPLE 7  PLANE     *
120 REM ******************************
130 :
140 SID=54272:REM --> START OF SOUND
150 :
160 REM >>>>> CLEAR SOUND CHIP <<<<<
170 FOR S=SIDTOSID+24:POKE S,0:NEXT S
180 :
190 REM >>>>> VOICE 1 A/D/S/R  <<<<<
200 POKE SID+5,20: POKE SID+6,20
210 :
220 REM >>>>>     SET VOLUME     <<<<<
230 POKE SID+24,15:REM --> MAXIMUM
240 :
250 REM >>>>> SET CONSTANT FREQ. <<<
260 LF = 255: HF = 7
270 :
280 REM >>>>> GATE VOICE 1 W/   <<<<<
290 REM >>>>> SQUARE WAVEFORM   <<<<<
300 POKE SID+4, 65
310 :
320 REM >>>>>  INCREASE PULSE   <<<<<
330 REM >>>>> WIDTH MODULATION <<<<<
340 FOR PH = 0 TO 15
350 FOR PL = 0 TO 255 STEP 10
360 POKE SID+2, PL: POKE SID+3, PH
370 :
380 REM >>>>> SET VOICE 1 FREQ. <<<<
390 POKE SID, LF:POKE SID+1, HF
400 :
410 REM >>>>>  DECREASE FREQ.   <<<<<
420 LF = LF - 5
430 IF LF<0 THEN LF=255: HF = HF-1
440 :
450 NEXT PL,PH
```

APPENDIX J

MULTIPLE VOICE PROGRAMMING

Consider for a moment what might be involved in programming more that one voice. Here again, plan well or you may run into complications. In setting up a multi-voice program, you should introduce more variables and arrays for easy storage and computation os each separate element of voice information.

This presents a problem if you are getting all your information from data statements. Think of how long your data statements might be if you had to store the frequency, waveform, A/D/S/R cycle, duration, or any other involved voice information sepaprately for each voice! If nothing else, it would take forever for it to be READ into arrays.

For long sound program, this time to load your voice information cannot be avoided. However, there are several ingenious methods for using a given data element for describing more that one voice parameter. This cuts down on DATA elements, hence the time it takes to read them in. Here is just one such method provided in Commodore's "Programmer's Reference Guide". It lets you write a multi-voice program by creating proper DATA tables for all your sound information. These are the steps:

1) Take each note's duration (the number of 1/16ths which it constitutes) and multiply it by 8.

2) Add the result of step 1 to the octave (0-7) from the note table.

3) Take this result and multiply it by 16

4) Take this result and add your note to it from the note table

Another way of saying this is to have D, O and N represent duration, octave and note respectively. The result of this little formula is one DATA element for one voice's information:

$$\text{Data Element} = (((D*8)+0)*16)+N)$$

Now all that your code has to do is disassemble this information to be POKEd into the correct registers.

The next thing to consider is to coordinate the two or three voices together. In other words we must determine the durations and rests of multi-voices at one time.

1) Divide each musical measure into 16 parts.

2) Store the events that occur in each 1/16th measure interval in three separate arrays.

3) Process array information using the waveform control byte as a starting signal for beginning a note or continuing a note that is already playing.

The multi-voice outline is more complex than the single voice, but it really does more.

OUTLINE FOR MULTI-VOICE

I. PROGRAM INITIALIZION
- A. Declare program parameters (same as the single voice outline)
- B. Declare multi-voice parameters
  1. Dimension arrays to contain activity of music
     a. 16th measure per location
  2. Dimension array to contain the base frequency for each note

II. MAIN PROGRAM LOOP FOR MULTI-VOICES
- A. FOR/NEXT loops for storing into 3-voice information arrays
  1. frequency start address array
  2. waveform control byte array
     a. one element/voice

- B. Set parameters inherent of all three voices
  1. Set voice registers that do not change in rest of program
     a. filter values
     b. pulse-width values
     c. etc.
  2. Special modulations
     a. ring modulation or hard sync.
     b. harmonic modulation (see S+28)
     c. any other additive modulations
- C. Nested FOR/NEXT loops for actual music execution
  1. Set pointer to 3-voice activity array
  2. Begin decode loop for each voice's parameter:
     a. assign variables to each decoded parameter.
  3. POKE in respective values into respective voice control register
  4. Increment 3-voice activity pointer
  5. Go to II.C-1 to process next measure

III. DATA STATEMENTS
- A. Set up DATA tables
  1. Put starting frequencies per voice in one table
  2. Put encoded voice information in another table
  3. Use an element(s) to delimit end of program

IV. GO ON TO NEXT PART OF PROGRAM OR END

If you can think of any brilliant alternative algorythms for multi-voices, you may have poineered a whole new approach, so please tell the world. About the only limitation using multi-voices is that you don't have the diversity of special modulation you have with one voice because all voices are dedicated to their assigned waveforms and are not free to be added to others to produce some desired blends. However, you probably won't find this to be a problem for programming sound ideas.

This sample program from Commodore's Programmers Reference Guide is a fine example of a multi-voice song. After you type 'RUN', the program will take 30 seconds to set up the data arrays necessary for the voices.

```
10 S=54272:FORL=STOS+24:POKEL,0:NEXT
20 DIMH(2,200),L(2,200),C(2,200)
30 DIMFQ(11)
40 V(0)=17:V(1)=65:V(2)=33
50 POKES+10,8:POKES+22,128:POKES+23,244
60 FORI=0TO11:READFQ(I):NEXT
100 FORK=0TO2
110 I=0
120 READNM
130 IFNM=0THEN250
140 WA=V(K):IFNM<0THENNM=-NM:WA=1
150 DR%=NM/128:OC%=(NM-128*DR%)/16
160 NT=NM-128*DR%-16*OC%
170 FR=FQ(NT)
180 IFOC%=7THEN200
190 FORJ=6TOOC%STEP-1:FR=FR/2:NEXT
200 HF%=FR/256:LF%=FR-256*HF%
210 IFDR%=1THENH(K,I)=HF%:L(K,I)=LF%:C(K,I)
    =WA:I=I+1:GOTO120
220 FORJ=1TODR%-1:H(K,I)=HF%:L(K,I)=LF%:C
    (K,I)=WA:I=I+1:NEXT
230 H(K,I)=HF%:L(K,I)=LF%:C(K,I)=WA-1
240 I=I+1:GOTO120
250 IFI>IMTHENIM=I
260 NEXT
500 POKES+5,0:POKES+6,240
510 POKES+12,85:POKES+13,133
520 POKES+19,10:POKES+20,197
530 POKES+24,31
```

```
540 FORI=0TOIM
550 POKES,L(0,I):POKES+7,L(1,I):POKES+14,
    L(2,I)
560 POKES+1,H(0,I):POKES+8,H(1,I):
    POKES+15,H(2,I)
570 POKES+4,C(0,I):POKES+11,C(1,I):
    POKES+18,C(2,I)
580 FORT=1TO80:NEXT:NEXT
590 FORT=1TO200:NEXT:POKES+24,0
600 DATA34334,36376,38539,40830
610 DATA43258,45830,48556,51443
620 DATA54502,57743,61176,64814
1000 DATA594,594,594,596,596
1010 DATA1618,587,592,587,585,331,336
1020 DATA1097,583,585,585,585,587,587
1030 DATA1609,585,331,337,594,594,593
1040 DATA1618,594,596,594,592,587
1050 DATA1616,587,585,331,336,841,327
1060 DATA1607
1999 DATA0
2000 DATA583,585,583,583,327,329
2010 DATA1611,583,585,578,578,578
2020 DATA196,198,583,326,578
2030 DATA326,327,329,327,329,326,578,583
2040 DATA1606,582,322,324,582,587
2050 DATA329,327,1606,583
2060 DATA327,329,587,331,329
2070 DATA329,328,1609;578,834
2080 DATA324,322,327,585,1602
2999 DATA0
3000 DATA567,566,567,304,306,308,310
3010 DATA1591,567,311,310,567
3020 DATA306,304,299,308
3030 DATA304,171,176,306,291,551,306,308
3040 DATA310,308,310,306,295,297,299,304
3050 DATA1586,562,567,310,315,311
3060 DATA308,313,297
3070 DATA1586,567,560,311,309
3080 DATA308,309,306,308
3090 DATA1577,299,295,306,310,311,304
3100 DATA562,546,1575
3999 DATA0
```

ADVANCED SOUND PROGRAMMING TECHNIQUES

Advanced techniques center on the use of modulation, filtering and variations of the A/D/S/R envelope. Here are some sample programs using these special features. The rest is up to your ability to POKE different values in to appropriate locations to hear the differences. Have a fun time programming sound!

Sound in the commodore 64 lives in the MOS 6581 chip called SID. SID stands for Sound Interface Device and is a single-chip 3-voice full electronic music synthesizer/sound effects generator. This chip provides all pitches, tonal qualities and dynamics that you can dream of in a computer. Specialized control circuitry has reduced the overhead in software required to produce wide sound variations. Here is a brief description of SID's chip specifications and pin configuration:

3 TONE OSCILLATORS
Range: 0-4 kHz

4 WAVEFORMS PER OSCILLATOR
Triangle, Sawtooth, Variable Pulse, Noise

3 AMPLITUDE MODULATORS
Range: 48 dB

3 ENVELOPE GENERATORS
Exponential response
Attack Rate: 2mS-8S
Decay Rate: 6mS-24S
Sustain Level: 0-peek volume
Release Rate: 6mS-24S

OSCILLATOR SYNCHRONIZATION

RING MODULATION

PROGRAMMABLE FILTER
Cutoff range: 30 Hz-12 kHz
12 dB/octave Rolloff
Low pass, Band pass,
High pass, Notch outputs
Variable Resonance

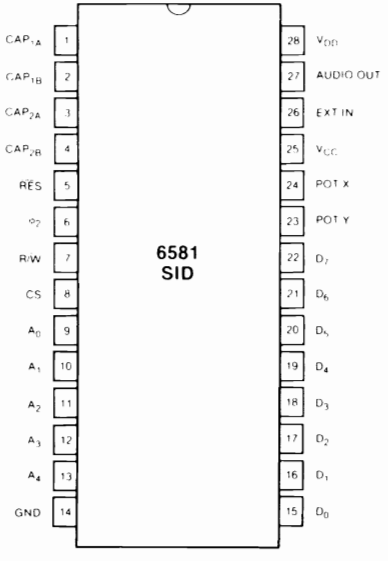
MASTER VOLUME CONTROL

2A/D POT INTERFACES

RANDOM NUMBER/MODULATION GENERATOR

EXTERNAL AUDIO INPUT

PIN CONFIGURATION



MUSIC NOTE VALUES

MUSICAL NOTE		OSCILLATOR FREQ		
NOTE	OCTAVE	DECIMAL	HI	LOW
0	C–0	268	1	12
1	C#–0	284	1	28
2	D–0	301	1	45
3	D#–0	318	1	62
4	E–0	337	1	81
5	F–0	358	1	102
6	F#–0	379	1	123
7	G–0	401	1	145
8	G#–0	425	1	169
9	A–0	451	1	195
10	A#–0	477	1	221
11	B–0	506	1	250
16	C–1	536	2	24
17	C#–1	568	2	56
18	D–1	602	2	90
19	D#–1	637	2	125
20	E–1	675	2	163
21	F–1	716	2	204
22	F#–1	758	2	246
23	G–1	803	3	35
24	G#–1	851	3	83
25	A–1	902	3	134
26	A#–1	955	3	187
27	1	1012	3	244
32	C–2	1072	4	48
33	C#–2	1136	4	112
34	D–2	1204	4	180
35	D#–2	1275	4	251
36	E–2	1351	5	71
37	F–2	1432	5	152
38	F#–2	1517	5	237
39	G–2	1607	6	71
40	G#–2	1703	6	167
41	A–2	1804	7	12
42	A#–2	1911	7	119
43	B–2	2025	7	233
48	C–3	2145	8	97

MUSICAL NOTE		OSCILLATOR FREQ		
NOTE	OCTAVE	DECIMAL	HI	LOW
49	C#–3	2273	8	225
50	D–3	2408	9	104
51	D#–3	2551	9	247
52	E–3	2703	10	143
53	F–3	2864	11	48
54	F#–3	3034	11	218
55	G–3	3215	12	143
56	G#–3	3406	13	78
57	A–3	3608	14	24
58	A#–3	3823	14	239
59	B–3	4050	15	210
64	C–4	4291	16	195
65	C#–4	4547	17	195
66	D–4	4817	18	209
67	D#–4	5103	19	239
68	E–4	5407	21	31
69	F–4	5728	22	96
70	F#–4	6069	23	181
71	G–4	6430	25	30
72	G#–4	6812	26	156
73	A–4	7217	28	49
74	A#–4	7647	29	223
75	B–4	8101	31	165
80	C–5	8583	33	135
81	C#–5	9094	35	134
82	D–5	9634	37	162
83	D#–5	10207	39	223
84	E–5	10814	42	62
85	F–5	11457	44	193
86	F#–5	12139	47	107
87	G–5	12860	50	60
88	G#–5	13625	53	57
89	A–5	14435	56	99
90	A#–5	15294	59	190
91	B–5	16203	63	75
96	C–6	17167	67	15
97	C#–6	18188	71	12
98	D–6	19269	75	69
99	D#–6	20415	79	191
100	E–6	21629	84	125

APPENDIX J

MUSICAL NOTE		OSCILLATOR FREQ		
NOTE	OCTAVE	DECIMAL	HI	LOW
101	F–6	22915	89	131
102	F#–6	24278	94	214
103	G–6	25721	100	121
104	G#–6	27251	106	115
105	A–6	28871	112	199
106	A#–6	30588	119	124
107	B–6	32407	126	151
112	C–7	34334	134	30
113	C#–7	36376	142	24
114	D–7	38539	150	139
115	D#–7	40830	159	126
116	E–7	43258	168	250
117	F–7	45830	179	6
118	F#–7	48556	189	172
119	G–7	51443	200	243
120	G#–7	54502	212	230
121	A–7	57743	225	143
122	A#–7	61176	238	248
123	B–7	64814	253	46

INDEX TO MEMORY LOCATIONS

Because of the unique way that this book presents its information, we have grouped the memory locations by major topic, and then alphabetized them within the groupings. Names are given to locations as they are commonly called by either Commodore or a consensus of the literature. Many locations have not been named. Also, many locations do not have a simple common use that they could be listed hereby. If you don't see what you are looking for under a topic such as "Input Output", then look at the locations without descriptions. You may, of course, "fill in the blanks" as you discover uses for locations.

DESCRIPTION	MEMORY LOCATION	NAME
COMPLEX INTERFACE ADAPTER (CIA)		
General Description		Pages 128-12
CIA 1		
Control register		
Control register timer A	56334	
Data direction register port A	56322	
Data direction register port B	56323	
I/O	56332	
Interrupt control	56333	
Joystick 1 fire button/lightpen trigger	56321	
Joystick 2 direction	56320	
Joystick 2 fire button	56320	
Read keyboard row values	56321	
Time-of-day clock: hours	56331	
Time-of-day clock: minutes	56330	
Time-of-day clock: seconds	56329	
Time-of-day clock: 1/10 seconds	56328	
Timer A high-byte	56325	
Timer A low-byte	56324	
Timer B high-byte	56327	
Timer B low-byte	56326	
Write keyboard column values	56320	
CIA 2		
Control register A	56590	
Control register B	56591	
Data direction register Port B	56576	
Data direction register Port B	56579	
Data port A:		
serial IEEE/RS-232 output	56576	
Data port B: RS-232	56577	
Future I/O expansion	56832-57087	
Future I/O expansion	57088-57343	
I/O	56588	
Time-of-day clock: hours	56587	
Time-of-day clock: minutes	56586	
Time-of-day clock: seconds	56585	
Time-of-day clock: 1/10 seconds	56584	
Timer A high-byte	56581	
Timer A low-byte	56580	
Timer B high-byte	56583	
Timer B low-byte	56582	

DESCRIPTION	MEMORY LOCATION	NAME
CURSOR		
Character under cursor in ASCII	206	GDBLN
Countdown to toggle cursor	205	BLNCT
Cursor blink flag	207	BLNON
Cursor blink toggle	204	BLNSW
Glossary	Pages 3-5	
INPUT/OUTPUT		
Buffer start pointer	178-179	TAPE1
Byte received flag	156	DSPW
Character parity	155	PRTY
Current device number	186	FA
Current file name	187-188	FNADR
Current secondary address	185	SA
Default input device	153	DFLTN
Default input device	154	DFLTO
I O buffer pointer	166	BUFPNT
I O start address	193-194	STAL
Input error log	256-318	BAD
Length of current file name	183	FNLEN
Load save memory pointers	174-175	EAL
Load verify flag	147	VERCK
Logical file number	184	LA
Number of open files	152	LDTND
Pass 1 error log	158	PTR1
Pass 2 error log	159	PTR2
Read write block count	190	FSBLK
Serial word buffer	191	MYCH
Sync. countdown	165	CNTDN
Sync. number	150	SYNO
Tape I O buffer	828-1029	CASI
Tape motor interlock	192	
Temporary data area	151	
Temporary D1IRQ indicator for cassette read	676	
Temporary storage for cassette read	176-177	CMPO
Timing constants	674	
TOD sense during cassette I O	0	D6510
	148	C3PO
	149	BSOUR
	172-173	SAL
	180	BITTS
	181	NXTBIT

DESCRIPTION	MEMORY LOCATION	NAME
	182	RODATA
	195-196	MEMUSS
	200	INDX
	645	TIMOUT
INTERNAL REGISTERS		
Storage for 6502 .SP register	783	STREG
Storage for 6502 .X register	781	SXREG
Storage for 6502 .Y register	782	SYREG
Storage for 6502 .A register	780	SAREG
KERNAL		
Active logical file numbers	601-610	LAT
BRT instruction interrupt	790-791	CBINV
CHKIN routine vector	798-799	ICHKIN
CHKOUT routine vector	800-801	ICKOUT
CHRIN routine vector	804-805	IBASIN
CHROOUT routine vector	806-807	IBSOUT
CLALL routine vector	812-813	ICLALL
CLOSE routine vector	796-797	ICLOSE
CLRCHN routine vector	802-803	ICLRCH
Device number for each file	611-620	FAT
GETIN routine vector	810-811	IGETIN
Kernal routines	Pages 113-116	
LOAD routine vector	816-817	ILOAD
Non-maskable interrupt	792-793	NMINV
OPEN routine vector	794-795	IOPEN
SAVE routine vector	818-819	ISAVE
Second address each file	621-630	SAT
STOP routine vector	808-809	ISTOP
KEYBOARD		
Current key	197	LSTX
INPUT GET from keyboard	208	CRSW
Keyboard decode table	243-244	KEYTAB
Keyboard shift key flag	653	SHFLAG
Keyboard table setup vector	655-656	KEYLOG
Last keyboard shift pattern	654	LSTSHF
Number of characters in keyboard buffer	198	NDX
Repeat delay counter	652	DELAY
Repeat speed counter	651	KOUNT
REPEAT key flag	650	RPTFLG
Shift key flag	657	MODE

DESCRIPTION	MEMORY LOCATION	NAME
Size of keyboard buffer	649	XMAX
STOP RVS key flag	145	STKEY
MATH		
Floating RND function seed value	139-143	RNDX
	3-4	ADRAY1
	5-6	ADRAY2
	13	VALTYP
	14	INTFLG
	18	TANSGN
	20-21	LINNUM
	38-42	RESHO
	97	FACEXP
	98-101	FACHO
	102	FACSGN
	103	SGNFLG
	104	BITS
	105	ARGEXP
	106-109	ARGHO
	110	ARGSGN
	111	ARISGN
	112	FACOV
	113-114	FBUFPT
	256-266	
MEMORY MANAGEMENT		
Bottom of memory for operating system	641-642	STRMEM
Bottom of string storage	51-52	FRETOP
End of BASIC arrays (+1)	49-50	STREND
Highest address used by BASIC	55-56	MEMSIZ
Load save memory pointers	174-175	EAL
Start of BASIC arrays	47-48	ARYTAB
Start of BASIC text	43-44	TXTTAB
Start of BASIC variables	45-46	VARTAB
Top of memory for operating system	643-644	MEMSIZ
Utility string pointer	53-54	FRESPC
MISCELLANEOUS		
	15	GARBFL
	146	SVXT
Temporary data area	163-164	
Unused	679-767	
Unused	787	

DESCRIPTION	MEMORY LOCATION	NAME
Unused	820-827	
Unused	1020-1023	
	671-672	IRQTMP
Peek and Poke	Pages 6-9	
Reconfiguring the memory map	Pages 106-110	
RS-232		
Enables	673	ENABL
Index to end of input buffer	667	RIDBE
Index to end of output buffer	670	RODBE
Input bit count	168	BITCI
Input bits	167	INBIT
Input byte buffer	170	RIDATA
Input parity	171	RIPRTY
Next bit to send	181	NXTBIT
Nonstandard BPS	661-662	M51AJB
Number of bits left to send	664	BITNUM
Out bit count	180	BITTS
Out byte buffer	182	RODATA
Out parity	189	ROPRTY
RS-232 status register	663	RSTAT
Start bit flag	169	RINONE
Start of input buffer (Page)	668	RIDBS
Start of output buffer (Page)	669	RODBS
6551 command register image	660	M51CDR
6551 control register image	659	M51CTR
SCREEN		
Background color	647	GDCOL
Background color	53281	
Background color 1	53282	
Background color 2	53283	
Background color 3	53284	
Border color	53280	
Bottom of screen memory	648	HIBASE
Character color code	646	COLOR
Current cursor line number	214	TBLX
Cursor column on current line	211	PNTR
Cursor X-Y position	201-202	LXSP
Editor in quote mode flag	212	QTSW
Insert mode flag	216	INSRT
Print shifted characters flag	203	SFDX
Programmable characters	Page 134	

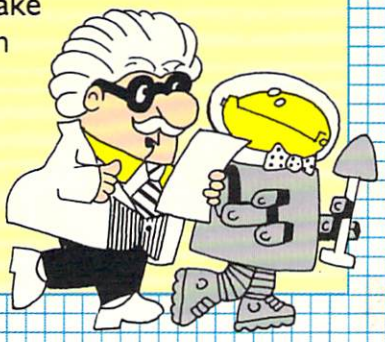
FOR BEGINNERS OR EXPERTS - THIS BOOK IS FOR YOU!

At last! An easy-to-read book that shows you the technical tricks to get the most out of your computer.

The MASTER MEMORY MAP is a clearly written, friendly guide to the inner workings of the Commodore 64 computer.

Full of useful explanations and examples, this book is a guided tour of all the memory locations - places inside the computer that make it act in special ways. You'll learn lots of uses for the Commodore 64, including how to make music; even how to create the special characters used in games.

If you're just beginning to program, we'll give you the information you need to write exciting programs - even add sound effects! If you've been programming for a while, the book will take you farther, allowing you to learn even more. Advanced programmers will use this book again and again as a powerful reference tool.



ISBN 0-13-574351-6

9 780135 743515

Prentice Hall International

£4.95