February 1995

Commodore Users Group of Saskatchewan

3617 29th Avenue ● Regina, SK ● S4S 2P8 ● Tel: (306)584-1736 ● BBS: (306)565-8562

the MONITOR

Newsletter Editor
Tristan Miller 584-1736
Secretary/Treasurer
Lyndon Soerensen n/a
64 Librarian
Stan Mustatia 789-8167
128 Librarian
Harvey Klyne 522-8694

The Monitor is published monthly by the **Commodore User's Group of Saskatchewan** (CUGS). Meetings are held on the first Wednesday of every month in **Room 173** of Miller High School unless otherwise noted. The next meetings will be held on **March 1** and **April 5**, 1995, from **7:30 to 9:30 P.M.**

CUGS is a nonprofit organization comprised of 64 and 128 users interested in sharing ideas, programs, knowledge, problems, and solutions with each other. Membership dues ($15) are pro-rated, based on a January to December year.

Anyone interested in computing is welcome to attend any meeting. Members are encouraged to submit **public domain** and **shareware** software for inclusion in the **CUGS Disk Library**. These programs are made available to members at $3.00 each (discounted prices when buying bulk). Since some programs on the disks are from magazines, individual members are responsible for deleting any program that they are not entitled to by law (you must be the owner of the magazine in which the original program was printed). To the best of our knowledge, all such programs are identified in their listings.

Other benefits of club membership include access to our disk copying service to make backups of copy-protected software, and any members who own a modem and wish to call our BBS will receive increased access. The board operates 300-2400 baud, 24 hours a day. The number is **565-8562**.

Editorial

Greetings, members. I'd like to begin this month's editorial by extending a warm thank you to Art Hamer, who was kind enough to allow us the use of his house for the meeting last month when we inadvertently found ourselves denied access to our usual place. I have been assured that we will not find ourselves in a similar situation this month.

February is upon us once again. Many of our members will find themselves back in school after a gruelling week of final exams, complementing the relief brought about by the wonderfully warm weather we've been having. With that in mind, there should be no excuses not to attend this meeting! :)

I have learned through our bulletin board that COMPUTE!'s Gazette is no longer in print; their February issue will be their last, marking about 12 years of quality programs and articles. Not bad for a magazine covering a computer that supposedly "died" six years ago. We'll all look forward to receiving the remaining Commodore publications in print — LoadStar, Twin Cities, and, of course, the CUGS Monitor.

I've sort of rediscovered my 64 as of late, having finally decided to finish off an adventure game I had been playing for eight years. I must say I was rather impressed with the ending; it has incited me to blow the dust off and restart its predecessor, Ultima IV, which I am now seemingly possessed by once again. A friend of mine just offered me the MS-DOS version of the game, but I flatly refused, explaining that I would not be content with replacing the brilliantly composed music and sound effects for the monotonous beeps of the PC speaker.

A Look Into the C128 Video Display Controller — Part One

E. Carl Reilly

Introduction
=========

It is not necessary to have an understanding of Assembler and BASIC 7.0 to grasp what this paper is trying to get across. For those of you who have had limited access to these languages, this introduction will provide you with enough knowledge to get you through the pages of this paper.

The Numbering System
==================

When working with computers it is often that we must resort to a different numbering scheme other than our normal 0 - 9 system of counting (also called decimal). Other frequent schemes include Hexadecimal (base 16) and Binary (base 2).

Your C128 is an 8-bit machine and each address can contain a value of 0 to 255 (decimal) for a total of 256 numbers. In binary 255 would look like %11111111 and in hex it would look like $FF. The symbols in front of the numbers depict what base you are working in. The percent sign (%) meaning binary, the dollar sign ($) meaning hex and the number sign (#) meaning decimal. Binary digits range from 0 to 1 and hex digits range from 0 to 9 and then A to F.

To relate this to our normal numbering scheme of decimal, $0A would equal #10, $0B would equal #11, and so on.
Figure I-1 shows the decimal equivalent to each binary placement in an 8-bit binary word.

FIGURE I-1

```
Decimal Values   128   64   32   16   8   4   2   1
                +----+----+----+----+---+---+---+---+
                |  |   |  |  |  |  |  |  |  |  |  |
                +----+----+----+----+---+---+---+---+
Bit Number        7    6    5    4   3   2   1   0
```

The numbers on top of each box represent the decimal equivalent to that particular placement of an 8-bit word. Just think of it as the hundredth, thousandth, etc. position as in decimal and you should have no problem.

If we took a binary word like %10010101, for example, and placed it into the boxes below the decimal numbers, we would be able to convert the number to something a little more coherent to us.
To use the table of Figure I-1, just add the decimal numbers that are set with a binary 1. The binary number of %10010101 would equal 128 + 16 + 4 + 1 = 149 in decimal. This process of converting numbers while working with computers is very common and performing the above calculations can be quite tedious. This is where hex is nice to work with. Hex is really just another way of writing binary.

If we take one nibble (4 bits) of a binary word, the maximum number we can obtain would be fifteen (%1111 = $F). With this in mind, converting the 8-bit binary word is very simple to acheive. See Figure I-2 for the conversion table.

FIGURE I-2

NIBBLE (%)	Hex ($)	Decimal (#)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

The high nibble of the previous binary word of %10010101 is the first four bits of the 8-bit word: %1001 ($9, as shown in Figure I-2); and the low nibble is %0101 ($5, as shown in Figure I-2, as well). To obtain the whole hex value we just combine the two converted nibbles together like so: $95. With this conversion, we've eliminated the need to write down the binary word into a table and then add the set bits together with their decimal equivalents to obtain a proper conversion. With the Hex conversion, it's easy, simple, and practically error free.

Why do we have to work with these numbers? Computers work with binary data. To write out and work with binary data as humans, it becomes very cumbersome and error prone. Hex is a nice and easy way of working with the computer and to ease your mind, whenever possible, I have included the decimal equivelants to the hex and binary numbers.

The Language Connection

The C128 has both BASIC 7.0 and an 8502 Assembler/Monitor built into its' operating system. The BASIC source code can be typed right into your C128 and then saved for later use. However, to keep things functional and easy, I've used the Power Assembler (also known as Buddy) to produce the Assembler source code in this paper. If you do not have this popular Assembler, then a few easy modifications are required if you choose to use the built in Assembler/Monitor.
All BASIC instructions (ie: SYS), Assembler directives (ie:.ORG, .MEM, etc.), labels (ie: READ, WRITE, etc.), and equates (ie: COLUMNS, ROWS, ect.) must be removed from the source code while typing it into the Assembler. You must begin to assemble the source code at the number referred by the .ORG command.

While Assembler may seem tedious and slow to program, it succeeds BASIC with its' speed, its' ability to access much more of the computer's functions, and its' versitility by being able to run more than one program at once (with the help of IRQ's, etc.).

This paper is written to take advantage of the Assembler code.

While there is a section with the equivelant BASIC source code, I recommend that you try to stick with the Assembler source code. When working with the VDC, BASIC is too slow and in- efficient at times. Keep this in mind if you begin to experiment with other VDC registers.

Assembler
=========

The C128 has one Accumulator and two registers, the X and Y registers. These are eight bit addresses that we can manipulate to make the C128 do what we need it to do. The Accumulator (A) will be our main worker and source of information, while the X register (X) will be used as a pointer and data holder. The Y register (Y) is used in the same manner as that of the X register, but for this paper, we really don't need to worry about using the Y register.

A bit can be set or cleared. To set a bit, means to make it a logical 1. And to clear a bit, means to make it a logical 0. Remember that old light bulb story about 0 meaning off and 1 meaning on? Well, keep it mind when you're working with binary notation and tests.
Such statements as: LDA, STA, INC, DEX, STX, BIT, etc. are all Assembler instructions called mnemonics (pronounced: new- mon-iks). Each mnemonic has a special purpose and meaning. For example, LDA means LoaD A (or more appropriately, LoaD Accumulator A), STA means to STore A, INC means to INCrement A by 1, DEX means to DEcrement X by 1, etc. You can just look at each mnemonic and decifer what each one means by it's three letter abbreviation (most of the time).
When we want to load A with a number for whatever reason, we can enter that number directly into the Accumlator like so:

```
LDA  #$9F
```

This instruction will tell the C128 to load A with the number $9F. We can also tell it to load the Accumulator with data from another address in memory like so:

```
LDA  $D601
```

This instruction will tell the C128 to load the Accumulator with whatever number was in address $D601. The difference between these last two instructions is subtle, but the advent of the number sign (#) in the first instruction lets the Assembler know whether the data is a direct number or a number stored in an address. These are the syntaxs you should look out for when typing the program into your C128.
I am not going to spend a lot of time introducing you to Assembler (or BASIC). There is more than enough information on the appropriate syntaxes in your Users Manual that came with your C128.

However, let me explain the BIT instruction. This instruction does not expand into a string of words, it is simply BIT. The BIT instruction is used (with another number) to test bits 6 and 7 of a number. If the test is true (meaning that one or both of the bits are set to 1) then a flag is set. For example, let's say that address $2000 is holding the number $9F. If we convert this number into binary it would look like %10011111. So, let's put it into Assembler source code to see what it would look like:

```
BIT  $2000
```

We used the BIT instruction to test bits 6 and 7 against the number $9F. The result of the BIT instruction would be true because bit 7 of the address is set.

Commodore Users Group of Saskatchewan

3617 29th Avenue ● Regina, SK ● S4S 2P8 ● Tel: (306)584-1736 ● BBS: (306)565-8562

The VDC has a status byte located at $D600. We must test bit 7 of this address to find out if it is OK for the VDC to communicate with us. This BIT instruction should perform quite adequately for this task.

The best learning tool a beginner can have is to get involved. If you would like to understand how Assembler works, the best way is to type in a nice, short program like this one, and see what makes it tick. You'll be surprised just how easy it is. The problem comes when a little patience is needed to get through a problem in your code. Sit back, relax and get away from the computer for a few moments to have a chance to think about the problem.

Let's begin.

A Look into The C128
Video Display Controller

By E. Carl Reilly

Getting through the basics!
==========

The C128 is quite a remarkable machine. In-point-of-fact, the C128 is four machines in one shell! It will flawlessly emulate a C64, modes for CP/M V3.0+ (both forty and eighty columns), a 40 column C128 with video controlled under the imphomus VIC II chip, and finally the 80 column C128 with its' video controlled by the 8563 Video Display Controller (or VDC for short). It is the latter subject that we are going to discuss in detail.

To begin, the VDC in your C128 was custom made to display text in 80 columns RGBI only (so we were told). RGBI stands for Red-Green-Blue-Intensity and is a nine pin DIN connector that's right beside your User Port. Because this is different technology, the colours in 80 columns differ slightly from that of the 40 column VIC II display (whether in 64 mode or 40 column 128 mode).

The great thing about the VDC is that it has its' own RAM separate from the C128's Operating System RAM. Depending on the design of your C128, you could have either 16K or 64K. If you own a C128D, then you have 64K (or should!). However, if you own a stock C128, then you have only 16K of VDC RAM. I must stress the fact that this is not part of the System RAM available to us (directly). Therefore, in actual fact, you could have a 192K C128 or a 144K C128.

This VDC has even more features! It has the ability to display both Uppercase/Lowercase and Uppercase/Graphics characters on the same screen. This is unlike the VIC II chip which must take RAM from the Operating System's RAM and only display one or the other character set at one time. However, the VIC II chip is much more accessible than the VDC is (sort of).

So why the difference in RAM size and what can it do for you? Well, the less RAM you have, the less you can do. For example, with 64K of VDC RAM, you can display over 65,000 different colours on the screen! You may store a Help screen in the VDC RAM and call it up instantly without having to waste space in your program redrawing the screen. You can store more screen clips while opening a window and restoring the clip underneath the window instantly. We'll get into the VDC's internal registers and functions in detail a little later on.

What if your C128 is only equipped with 16K RAM and how do you tell? Well, if you own a stock C128 (the one that has the style of the slim line C64c), then you only have 16K. If you own the C128D (the one with the internal 1571 disk drive and detachable keyboard), then you have 64K. The easiest way to check is to open up your C128 (voiding your warranty, if one is still active), and see. Not to fear! If you only have 16K of RAM for your VDC, you can still upgrade to 64K with a little work. SSI in Vancouver, Washington sells a 64K RAM module for around $50 (ouch!) or you can install your own 64K DRAM's for about $10.00. For information on upgrading and installing 64K into your VDC, just ask for the info at the address at the end of this paper.

Getting in Deeper
==========

The VDC is more than an 80 column text display driver. It has the ability to move blocks of data in its' own RAM, display attributes (like blinking text, underlined text, extended screen editing, etc.), bit map graphics, etc. It is a great piece of engineering.
The VDC has 37 registers that we can manipulate. A complete list of these registers and their functions are detailled later in this paper. The VDC is sort of picky on how you access its' registers. You cannot just POKE a number into an address and expect the VDC to react; you have to ask permission (for lack of better words). The VDC has two "Ports" that let us read and write to the 37 registers. Through the addresses of $D600 the address register (54784 in Decimal) and $D601 the data register (54785), we can talk to the VDC.

First we must tell it which register we want to access and then we have to wait until the VDC has completed its' housekeeping before we can do what we want.

We'll get into the BASIC instructions later. For now we will discuss the Assembler code. This will give us a better understanding as to how the VDC works and how we must access it.

When working with the VDC, we must understand how to communicate with it. The X-pointer must contain the register number we wish to talk to and the Accumulator will contain the information. To explain this concept a little better, we are going to find out how to read a register in the VDC. Figure 1 displays the Assembler code involved to read register 6.

FIGURE 1

```
            LDX  #$06
WAIT        STX  $D600
            BIT  $D600
            BPL  WAIT
            LDA  $D601
            STA  BUFFER
            BRK
BUFFER      .BYTE $00
            BRK
```

In the first line, the number $06 is loaded into the X-Pointer. This is the register we want to communicate with. Next, the X-Pointer is then stored into address $D600, the address register of the VDC. This will tell the VDC which register will be accessed next. The BIT instruction in the

next line will test bit 7 of the address register to find out whether or not it is OK to communicate with register 6. Following through, if it is not OK, we go back to the beginning of the WAIT subroutine and do it all over again until it is OK. Finally, when it is OK, we load the information into the Accumulator, store the value of the Accumulator into the BUFFER, and break out of the program.
Now, let's write to register 6 of the VDC. It is done the same way as above, except that we must store a value from the Accumulator into the register instead of loading the value from the register and into the Accumulator. So, let's put a value of $52 into register 6.
FIGURE 2

```
        LDX  #$06
        LDA  #$52
  WAIT  STX  $D600
        BIT  $D600
        BPL  WAIT
        STA  $D601
        BRK
```

Basically the same except that we had loaded the accumulator with our value of $52 at the start and then stored it into register 6 after it was OK to communicate. Simple!

Understanding By Diving In
==========

Over the many years of programming and understanding new concepts and ideas, I have found that it is a lot easier to learn something while performing a task involving the actual concept. It is almost a must to get the "learnee" involved with the process. So here we go. We are going to develop a small program in both BASIC 7.0 and Assembler that will let us toggle the 80 column screen off and on, basically creating a simple 80 column screen saver. The Assembler format will be edited using the POWER ASSEMBLER (or BUDDY, for short) from Spinnaker.

So, what is first? Well, before we begin, we're going to have to determine what we will need to perform a screen blanker. From looking at the register map, it looks as though we are going to need register 1 and register 6. Register 1 takes care of how many characters wide, from left to right, the display is. While register 6 takes care of how many characters long, from top to bottom, the display is. Essentially, we want to toggle these register values from their initial values to zero and then back again when need be.

How are we going to do it? Well, a really great function that is present in both 8502 Assembler and BASIC 7.0 is the Exclusive OR function (EOR). This function has the ability to toggle bits on and off. See Table 1 below to get an idea of what I mean.

TABLE 1

A	B	EOR	RESULT
0	0	=	0
0	1	=	1
1	0	=	1
1	1	=	0

If we inspect this little wonder a bit closer, we can see that if any two bits are the same and EOR'd, the result is a 0. And if any two bits are not the same, the result of the EOR is a 1. Huh? Don't concern yourself with how this function works, just accept that is does work this way

and it'll be a lot easier.

Let me explain how this is going to help us. We want to read the value of register 1 and then toggle it from $50 (80 columns) to $00 and then back to $50 again. $50 in binary is 01010000 and $00 is 00000000. Let's put the value $50 into a variable called COLUMNS to help illustrate this idea. Starting on the right most bit, just follow through as you would an addition problem and refer back to Table 1.

```
          01010000  This is our variable COLUMNS
 EOR      01010000  This is the current value of register 1
          ----------
RESULT 00000000  The effective result is $00
```

Starting from the right hand side, we see a 0 at the top and a 0 at the bottom. From Table 1, we see that a 0 EOR'd with another 0 will result with a 0. Four more bits to the left, we can see that there's a 1 on the top EOR'd with a 1 on the bottom. From Table 1, we can see that a 1 EOR'd with a 1 will result in a 0. And so on down the bits.
To toggle back to the initial number, we must do the same thing.

```
          01010000   This is our variable COLUMNS
 EOR      00000000   This is the current value of register 1
          ----------
RESULT  01010000   The effective result is $50
```

Again, just go through it like an addition problem.

With that out of the way, let's outline this program now. We are going to have to read a register to obtain a value to toggle and then write the result back to the register. We are going to need two values, one for the number of columns ($50), and one for the number of rows ($19). In our Assembler program, we are also going to have to find a place to put this subroutine and then return back to the editor when the task is done.

Let's begin by constructing the Assembler version first. We are going to place this subroutine at address $1300 and declare our variables (more properly named, "equates").

```
10 SYS 4000
20 .ORG $1300
25 .MEM
30 COLUMNS =$50
40 ROWS =$19
```

Line 10 tells Buddy to begin compiling the program. Line 20 lets Buddy know where to put the compiled program in memory. Next, we tell Buddy that we want to compile the code straight to RAM. Completing this part, we declare our equates.

OK, now we have to read each register and then toggle the display. We'll start with register 1.

```
50 LDX #$01
60 JSR READ
70 EOR COLUMNS
```

Commodore Users Group of Saskatchewan

3617 29th Avenue ● Regina, SK ● S4S 2P8 ● Tel: (306)584-1736 ● BBS: (306)565-8562

We loaded the X-Pointer with #$01 and then jumped to the subroutine READ. The READ subroutine will be placed at the end of our main code and is the same READ subroutine that we looked at earlier in FIGURE 1. When the READ subroutine has done its' chore, the value of register 1 is stored in the Accumulator. The Accumulator will be EOR'd with the value of that in our equate COLUMNS. The result will be placed back into the Accumulator automatically for us by the C128. So all that is left is to write the new value back into the same register. Remember that our X-Pointer was not tampered with and still contains $01, so we don't need to re-initialize the X-Pointer.

```
80 JSR WRITE
```

Now, we have to do the same thing for register 6:

```
90 LDX #$06
100 JSR READ
110 EOR ROWS
120 JSR WRITE
```

And finally, we have to return back to wherever we had called this code from:

```
130 RTS
```

There! Now to put all of this code into one complete program, it should look like this [don't enter the information after the semi-colons (;). These are just there to help you understand what that particular line of code is trying to accomplish]:

```
10 SYS 4000
20 .ORG $1300
25 .MEM
30 COLUMNS =$50        ;Set up equates
40 ROWS =$19
50 LDX #$01            ;Load X-Pointer with $01
60 JSR READ            ;Jump to subroutine to read register 1
70 EOR COLUMNS     ;EOR contents of accumulator with equate
80 JSR WRITE          ;Write new value of Accumulator back to register 1
90 LDX #$06           ;Load X-Pointer with #$06
100 JSR READ          ;Jump to subroutine to read register 6
 110 EOR ROWS            ;EOR contents of Accumulator with the current number of rows
120 JSR WRITE                ;Write new value of Accumulator back to register 6
130 RTS               ;Exit
140 READ STX $D600  ;Store value of X-Pointer
150 BIT $D600
160 BPL READ          ;Go back and wait if not ready
170 LDA $D601         ;Load value of current register into Accumulator
180 RTS               ;Return to call
190 WRITE STX $D600 ;Store value of X-Pointer
200 BIT $D600
210 BPL WRITE
220 STA $D601         ;Write value of Accumulator into the current register
230 RTS               ;Return back to call
```

If we call this program from BASIC, we would enter the following (Clear the screen and make sure that the cursor is in the home position!):

```
BANK 15:SYS DEC("1300")          <RETURN>
```

Your 80 column screen should go blank! If it didn't, then you may have made a mistake in the Assembler code. You should go back and recheck the program for proper syntax.

However, if everything went as planned, just press the <HOME> key and then the <RETURN> key again. The display should re-appear.

*The conclusion to Carl's **A Look Into the C128 Video Display Controller** will appear in the next issue of the MONITOR.*

Addendum

Just as I was finishing up this month's issue of the Monitor, I received a call from a Mr. Ron Haslam of Texas, who wanted to inform me that, as a "die-hard Commie user", he had been "stockpiling" a wide range of Commodore equipment over the past eleven years. His purpose, he tells me, is to make available spare parts to other Commodore users who find themselves at a loss now that Commodore Electronics has gone under. Anyone requiring Commodore 8-bit or Amiga disk drives, computers, modems, printers, etc. is encouraged to contact him at the following address or phone number.

Ron Haslam
203 Humble Avenue
Midland, TX
USA 79705
Tel: (915) 686-8223