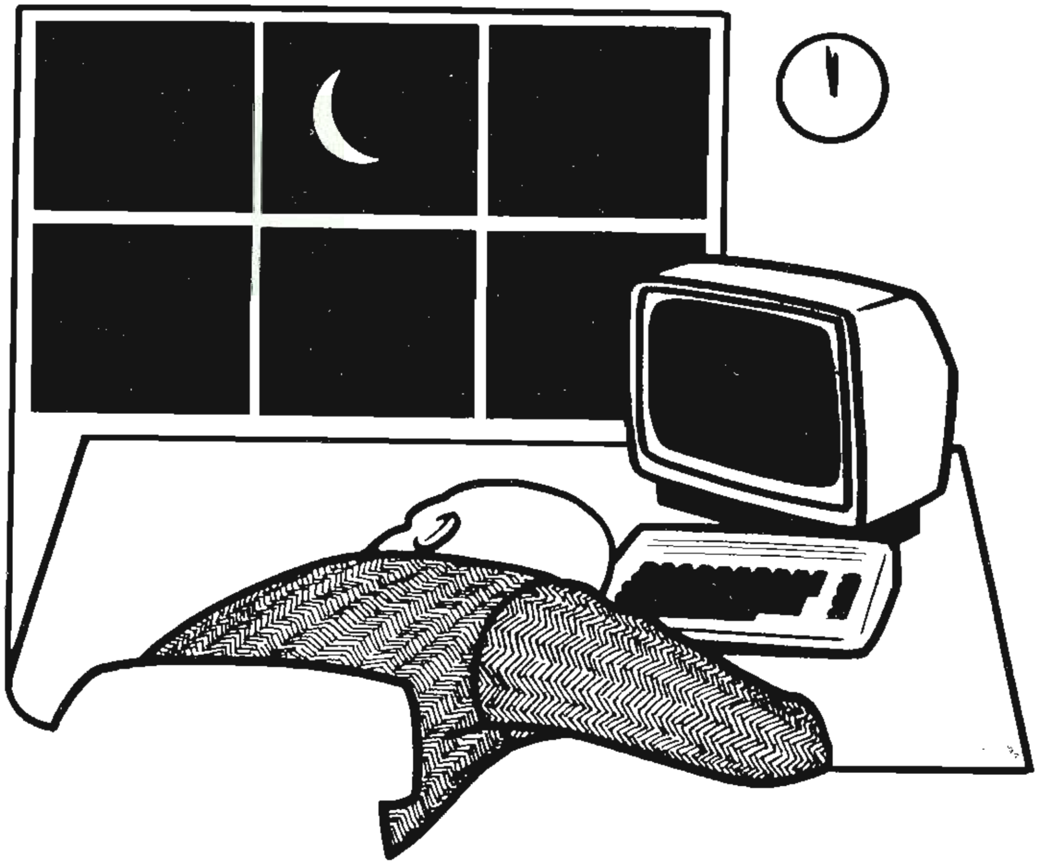
PAY NO MORE

$2.00 US

ISSUE 36 1986

Midnite Software Gazette

The First Independent U.S. Magazine for users of Commodore brand computers.



I N T R O D U C I N G

SUPER KIT/1541

BY MARTY FRANZ & JOE PETER

SINGLE/DUAL NORMAL COPIER
Copies a disk with no errors in 32.68 seconds. dual version has graphics & music.

SINGLE/DUAL NIBBLE COPIER
Nibble Copies a disk in 34.92 seconds. dual version has graphics & music.

SINGLE/DUAL FILE COPIER
6 times normal DOS speed Includes multi-copy, multi-scratch, view/edit BAM, & NEW SUPER DOS MODE

TRACK & SECTOR EDITOR
Full editing of t&s in hex, dec, ascii, bin. Includes monitor/disassembler with printout commands.

GCR EDITOR
Yes disk fans, a full blown sector by sector or track by track GCR Editor. Includes Bit Density Scan.

SUPER DOS I
Fast boot for SUPER DOS. 150 blks in 10.12 seconds.

SUPER DOS II
Screen on and still loads 150 in 14.87 seconds.

SUPER NIBBLER
Quite frankly, if it can be copied on a 1541 this will do it! Including Abacus, Timeworks, Accolayde, Epyx, Acti-vision, Electronic Arts.

$29.95

PLUS $3.00 SHIPPING/HANDLING CHARGE — $5.00 C.O.D. CHARGE

PRISM SOFTWARE

401 LAKE AIR DR., SUITE D • WACO, TEXAS 76710
ORDERS (817) 751-0200
MASTERCARD & VISA ACCEPTED

(C)opyright 1986 Micro-PACE, All Rights reserved

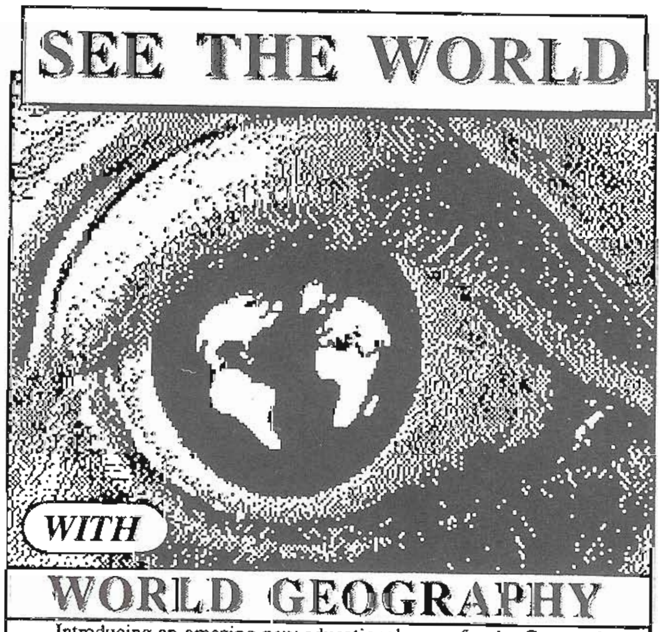
Published by:	Micro-PACE Comptuters 1212 Hagan St. Champaign, IL 61820 (217) 356-1885 Bob Wolter, Owner/President
Editor-in-Chief:	James A. Oldfield, Jr., V.P.
Editor:	Tim Sickbert
Assc. Editors:	Art Lewis Kimball Robert Baker Dr. Richard Immers Mike Stout
Educational Editor:	Elizabeth Kasper

Address for Correspondence:
Post Office Box 1747
Champaign, IL 61820

Voice Telephone:	(217) 356-1885
Starship MPC BBS:	(217) 356-8056 (300/1200 baud Punternet #35!)
Issue 36	August/September 1986

Ad rates are low: Full page, $200; Half page, $100; etc.

Circulation this issue: 1500



Introducing an amazing new educational game for the Commodore 64/128 featuring a FULL COLOR 3D ROTATING GLOBE and detailed graphics of over 175 countries for 1 or 2 players.

only **$24.95**

Add $2 shipping and handling. CA residents add 6.5% sales tax.

BOBCO

200 7th Ave., suite 111 Santa Cruz, CA 95062

1-800-331-4321 **(CA) 1-800-851-1986**

Now available on microfiche.

Only the finest, most respected publications are on microfiche.

Bell & Howell chooses only those publications with the highest reference or research value to be made available on microfiche. This publication, among many other respected titles, has been awarded that honor.

Call or write Ms. Pam Shea, Quotations Editor
Bell & Howell, Micro Photo Division
Old Mansfield Road, Wooster, OH 44691
or call toll free 1-800-321-9881.
In Ohio call collect 1-216-264-6666.

Please send complete information about microfiche.

Title of Publication ____________

Name ____________ Title ____________

Company ____________

Co. Address ____________

City ____________ State ____________ Zip ____________

GETTING STARTED IN ASSEMBLY LANGUAGE PROGRAMMING

by Robert W. Baker

Many VIC-20 and Commodore-64 owners are newcomers to the world of computer programming. They've had enough trouble learning how to program in BASIC without being further confused with assembly or machine language programming. However, by not knowing how to enter and use prepared assembly language programs, they are missing out on a number of valuable utilities and useful routines published in magazines from time to time. I thought it might be valuable to provide a basic introduction to assembly language programming and describe how to enter and use an assembly language program.

The current and previous Commodore systems are based on the 6502 microprocessor or some newer member of the same family. The 6502, and all other processors, understand only the ones and zeroes that correspond to on or off states. Thus, all data and instructions are binary. Users find it hard to work with the binary number system, and therefore use a more convenient representation such as hexadecimal (base 16) or decimal. A typical 6502 instruction to load the value 21 (decimal) into the accumulator may be shown as:

```
10101001    00010101   in binary
 A9            15      in hexadecimal
169            21      in decimal
```

The hexadecimal system is commonly used for machine language programming because it is the easiest to use when talking about addresses within the computer. For example, the BASIC text area begins at the decimal address 2048. The hexadecimal address $0800 specifies the same location, but is easier to work with since the hexadecimal number is round-ending in two zeros. This becomes more important when working with addresses like 56576 (decimal); it is usually easier to remember the same address as $DD00. Most assembly language programmers use hexadecimal, and most articles give programs in hex. I shall use it here as well.

Even with the convenience of hex, programmers find numeric representation of instructions tedious to work with. So a symbolic representation is commonly used. For example, the preceding instruction might be written as:

```
        LDA     #21
```

In this case, LDA is the symbol for the instruction to load the accumulator. A computer program called an assembler is used to translate the symbolic form LDA to the numeric form $A9. The symbolic program is referred to as source code, and the numeric program generated by the assembler is called the object code. (The numeric code produced by the assembler is actually in binary but the assembler program shows it to you as hex so it will be easier for you to read.) Some assemblers also produce a listing which shows the correlation between the source code and the object code. Only the binary code can be executed on the computer, but a special loader program may be needed to load and run the object code.

Each machine instruction has a symbolic name, referred to as an operation code, op code, or mnemonic. Some op codes require an operand to specify the data on which the operation is to be performed. The operation portion of an instruction specifies either an address or a value, and may contain an expression such as L2+2 for computed values.

Additionally, any instruction may be labeled for reference by other instructions, as shown by:

```
Label2      LDA     #21
```

In this example, the label is Label2, the op code is LDA, and the operand is #21. Labels are used as targets by branch instructions (the machine language equivalent of 'GOTO') and as the data elements (variables) within operands.

Most assemblers allow comments following the instruction operands. This provides a convenient way to document the program flow for later reference.

Assembler directives are another important feature of most assemblers. These are special instructions to the assembler to reserve storage space, generate data constants, or otherwise control the assembler operation.

Many different assemblers are available for the Commodore systems. Simple assemblers may assemble source code from BASIC DATA statements and POKE the object code directly into memory. Others may read the source code from tape or disk files and create an object file that must later be loaded by a special program (such as with the assembler offered by Commodore).

Assemblers written in BASIC are inexpensive and relatively slow, while those written in machine code run much faster and usually provide a number of features. Be aware of any limitations or requirements of an assembler before considering it for your particular needs.

Sample listing 1 shows an assembly listing for a very simple machine language routine. Let's disregard the actual function of the routine and just look at the listing and what it tells us. The first column of numbers indicates the hexadecimal memory locations where every instruction or data constant is located. The next three columns of two-digit hexadecimal values indicate the object code, starting at the corresponding location indicated on that line.

The next column shows the source code line numbers. These are generally shown merely for convenience They may, however, be used by a special editor for editing or creating the source code. The remainder of the line is the actual source code that was used to generate the object code shown on that line. There may be comments on the source code line to document the program operation.

To get this routine into your machine, if you have an assembler program, you might just have to type in the source code and assemble the program. At worst, you might have to convert the assembler syntax from that used in the article to the form used by your assembler.

However, if you don't have an assembler you can still enter and use the routine. If the magazine doesn't use some standard loader program for machine language programs, just use one of the available monitor programs to enter the object code directly into the memory locations specified in the assembly. If the program is short (like Sample listing 1), entering the object code should be fairly simple; entering larger programs may not be a very pleasant task.

To enter the object code directly, first activate the monitor program as normal. With most monitors you then have to display the memory area to be modified, entering the desired starting and ending addresses. You can get the hexadecimal addresses directly from the first column of the assembly listing.

If the program is long, choose a block of memory that will fit on the screen. Now enter the data in the appropriate locations, as indicated in the assembly listing. The data will normally be shown in hexadecimal in the listing, the form in which it will be accepted by most monitors. Figure 1 is a memory display showing how the sample program might be entered using a typical monitor.

When you've finished entering a machine language program into memory, it's always a good idea to first save it on tape or disk before doing anything else. If you make a mistake in entering the program, it could cause the entire system to hang when the program is run. If this were to happen, you could lose the program you just entered when you reset or power off the system to regain control.

Most monitor programs provide the necessary command to save an area of memory to tape or disk. You'll normally have to specify the starting and ending addresses, with the ending address possibly one higher than the last location to be saved.

There's one point that many Commodore owners are not aware of. You do not need to use the monitor to load a machine language program that was saved on tape or disk. The normal BASIC Load command will load either a BASIC or a machine language program. Both are saved as a simple memory dump between two locations with the same file header. Just be sure to use the secondary address of 1 in the LOAD command if you don't want the program to be relocated by the loader.

Once you've saved it, try the program for proper operation. If you have a problem, reload the program and check the values entered using the monitor. Also, be sure the proper starting address was used to execute the program or routine. The normal starting address should be given in the article.

Occasionally, there may be alternate starting addresses for different functions or options. Also, some programs may expect parameters at specific locations set by a BASIC program or certain variables defined in a specific order. Be sure to read the article for details what it the program expects.

Another way to double-check a machine language program entered by a monitor is to use a disassembler, which is a simple program that converts object code back into symbolic assembler form. For those interested, a simple BASIC disassembler program is included. This program requests a starting address and then asks if printed output is desired. It then produces a disassembly starting from the location specified.

This program can also be helpful in looking at routines in the BASIC or Kernel ROMs of the system itself, if the appropriate starting address is given. I should warn, however, that if the starting address is not the first byte of an instruction (if it's actually an instruction operand), the output may be unpredictable. You may have to experiment with the starting address to get desirable results.

Sample listing 3 shows the disassembler output. As you can see, the disassembly listing is much like the assembly listing except there are no labels. All addresses are shown as absolute addresses in hexadecimal notation. This disassembler provides the decimal as well as the hexadecimal location of each instruction for added convenience.

Also, on all branch instructions the actual target address, rather than the relative offset, is indicated. Thus, the disassembler can be used to verify that the correct instructions have been entered at the appropriate places.

When using the disassembler, it first takes a few seconds to set up an internal table that is later used to speed up the disassembly process. It then requests a starting address that can be entered as a decimal number or as a hexadecimal number with a leading dollar sign ($). If desired, the output can be printed or displayed.

When printing the disassembly output, press any key on the computer keyboard at any time to suspend printing. Printing will stop at the end of the current line. You then have the option of continuing the disassembly, terminating the printing and restarting another disassembly, or stopping the program and return to Basic. Enter the appropriate letter for the desired action as indicated.

When displaying the disassembly output, the same options are available at the prompt at the bottom of each display screen. This lets you easily page through memory as long as desired. Restarting a disassembly allows specifying a new starting address and re-directing the output to the printer or screen.

Sample listing 1 - Typical assembler output listing

```
               0010            .LS      ;TURN ON LISTING
               0020 ;***************************
               0030 ;
               0040 ;  SAMPLE ML PROGRAM SOURCE
               0050 ;
               0060 ;***************************
               0070
               0080 SPC       .DI $0020     ;DEFINE SPACE CHAR
               0090
               0100           .BA $0073     ;STARTING LOCATION
               0110
0073- E6 7A    0120 CHRGET    INC CHRGOT+1  ;INCREMENT
0075- D0 02    0130           BNE CHRGOT    ; ADDRESS &
0077- E6 7B    0140           INC CHRGOT+2  ;  BYTE IF CARRY
0079- AD 34 12 0150 CHRGOT    LDA $1234     ;GET BYTE (DUMMY ADR)
007C- C9 3A    0160           CMP #$3A      ;CHECK FOR COLON CHAR
007E- B0 0A    0170           BCS CHRX      ;EXIT IF COLON
0080- C9 20    0180           CMP #SPC      ;CHECK FOR SPACE CHAR
0082- F0 EF    0190           BEQ CHRGET    ;IF SPC, READ AGAIN
0084- 38       0200           SEC
0085- E9 30    0210           SBC #$30      ;CHECK CHAR NUM/LTR
0087- 38       0220           SEC
0088- E9 D0    0230           SBC #$D0
008A- 60       0240 CHRX      RTS      ;EXIT WHEN DONE
```

Sample listing 3 - Disassembler output

```
LOC-DEC/HEX OBJECT DISASSEMBLY

115  0073: E6 7A     INC $7A
117  0075: D0 02     BNE $0079
119  0077: E6 7B     INC $7B
121  0079: AD 34 12  LDA $1234
124  007C: C9 3A     CMP #$3A
126  007E: B0 0A     BCS $008A
128  0080: C9 20     CMP #$20
130  0082: F0 EF     BEQ $0073
132  0084: 38        SEC
133  0085: E9 30     SBC #$30
135  0087: 38        SEC
136  0088: E9 D0     SBC #$D0
138  008A: 60        RTS
```

Fig. 1 - Memory display showing sample routine using monitor

```
:  0073 E6 7A D0 02 E6 7B AD 34
:  007B 12 C9 3A B0 0A C9 20 F0
:  0083 EF 38 E9 30 38 E9 D0 60
```

```
100 rem machine language dis-assembler
110 rem       by: robert w. baker
120 :
130 print"[CLR]dis-assembler"
140 print"[DOWN][DOWN][DOWN]initializing ..
.."
150 dim m$(255): h$="0123456789abcdef"
160 for x=0 to 255: read a$: if a$="*" then
 a$="0*?*"
170 a$=left$(a$+"      .",6): m$(x)=a$: nex
t x: rem <-- 6 spaces
180 print"[CLR]enter decimal starting addre
ss[DOWN]"
190 print"or '$' followed by hex address[DO
WN]"
200 input a$: if left$(a$,1)="$" then 230
210 for x=1 to len(a$): c$=mid$(a$,x,1): if
 c$ < "0" or c$ > "9" then 180
220 next x: a=int(val(a$)/8)*8: goto 290
230 a=0: if len(a$)<2 then 180
240 for x=2 to len(a$): c$=mid$(a$,x,1): if
 c$ < "0" then 180
250 if c$<="9" then a=a*16+val(c$): goto 28
0
260 if c$<"a" or c$>"f" then 180
270 a=a*16+asc(c$)-55
280 next x
290 print"[DOWN]want printed copy": input"(
y/n)   n[LEFT][LEFT][LEFT]";c$
300 p=3:if left$(c$,1)="y" then p=4
310 open 4,p
320 print"[CLR]";: if p=3 then 340
330 print"depress any key to halt printer":
print#4
340 print#4,"[RVON]loc-dec/hex  object   di
ssassembly    ": print#4
350 if p=3 then for n=1 to 20: rem ***** ch
ange 20 to 8 for vic-20 *****
360 if a>65536 then a=a-65536
370 a$=str$(a): print#4,right$("      "+a$,
6);" "; : rem <-- 6 spaces
380 y=a:gosub 690: print#4,": ";
390 v=peek(a): gosub 700: print#4," ";:  a=
a+1:  a$=m$(v)
400 if left$(a$,1)="0" then print#4,"";spc(
7);mid$(a$,2,3): goto 580
410 v=peek(a): gosub 700: print#4," ";: a=a
+1
420 if left$(a$,1)="2" then 510
430 print#4,"";spc(4);mid$(a$,2,3);" ";: if
 mid$(a$,5,1)<>"r" then 460
440 if v>127 then v=v-256
450 y=a+v: gosub 680: goto 570
460 if mid$(a$,5,1)="#" then print#4,"#$";:
 gosub 700: goto 570
470 if mid$(a$,6,1)=")" then print#4,"(";
480 print#4,"$";: gosub 700: if mid$(a$,5,1
)=" " then 570
490 if mid$(a$,5,2)="y)" then print#4,"),y"
: goto 580
500 print#4,",";mid$(a$,5,2): goto 580
510 v1=v: v=peek(a): gosub 700: a=a+1: prin
t#4,"  ";mid$(a$,2,3);" ";
520 y=v1+(256*v)
530 if mid$(a$,5,1)=")" then print#4,"(";:
gosub 680: print#4,")": goto 580
540 gosub 680
550 if mid$(a$,5,1)=" " then 570
560 print#4,",";mid$(a$,5,1);
570 print#4
580 if p=3 then next n: goto 600
590 get c$: if c$="" then 360
600 print"[DOWN][RVON]continue, restart, or
 stop (c,r,s) ?  [RVOF]";
610 get c$: if c$="c" then 320
```

```
620 if c$="r" then close 4: goto 180
630 if c$<>"s" then 610
640 close 4: end
650 :
660 rem subroutines
670 :
680 print#4,"$";
690 v=int(y/256): gosub 700: v=y-(v*256)
700 h=int(v/16): l=v-(h*16)
710 print#4,mid$(h$,h+1,1);mid$(h$,l+1,1);:
 return
720 :
730 rem ========================
740 rem 6502 instruction set data
750 rem ========================
760 :
770 data 0brk,1orax),*,*,*,1ora,1asl,*
780 data 0php,1ora#,0asl,*,*,2ora,2asl,*
790 data 1bplr,1oray),*,*,*,1oray,1aslx,*
800 data 0clc,2oray,*,*,*,2orax,2aslx,*
810 data 2jsr,1andx),*,*,1bit,1and,1rol,*
820 data 0plp,1and#,0rol,*,2bit,2and,2rol,*
830 data 1bmir,1andy),*,*,*,1andx,1rolx,*
840 data 0sec,2andy,*,*,*,2andx,2rolx,*
850 data 0rti,1eorx),*,*,*,1eor,1lsr,*
860 data 0pha,1eor#,0lsr,*,2jmp,2eor,2lsr,*
870 data 1bvcr,1eory),*,*,*,1eorx,1lsrx,*
880 data 0cli,2eory,*,*,*,2eorx,2lsrx,*
890 data 0rts,1adcx),*,*,*,1adc,1ror,*
900 data 0pla,1adc#,0ror,*,2jmp),2adc,2ror,
*
910 data 1bvsr,1adcy),*,*,*,1adcx,1rorx,*
920 data 0sei,2adcy,*,*,*,2adcx,2rorx,*
930 data *,1stax),*,*,1sty,1sta,1stx,*
940 data 0dey,*,0txa,*,2sty,2sta,2stx,0
950 data 1bccr,1stay),*,*,1styx,1stax,1stxy
,*
960 data 0tya,2stay,0txs,*,*,2stax,*,*
970 data 1ldy#,1ldax),1ldx#,*,1ldy,1lda,1ld
x,*
980 data 0tay,1lda#,0tax,*,2ldy,2lda,2ldx,*
990 data 1bcsr,1lday),*,*,1ldyx,1ldax,1ldxy
,*
1000 data 0clv,2lday,0tsx,*,2ldyx,2ldax,2ld
xy,*
1010 data 1cpy#,1cmpx),*,*,1cpy,1cmp,1dec,*
1020 data 0iny,1cmp#,0dex,*,2cpy,2cmp,2dec,
*
1030 data 1bner,1cmpy),*,*,*,1cmpx,1decx,*
1040 data 0cld,2cmpy,*,*,*,2cmpx,2decx,*
1050 data 1cpx#,1sbcx),*,*,1cpx,1sbc,1inc,*
1060 data 0inx,1sbc#,0nop,*,2cpx,2sbc,2inc,
*
1070 data 1beqr,1sbcy),*,*,*,1sbcx,1incx,*
1080 data 0sed,2sbcy,*,*,*,2sbcx,2incx,*
```

```
100 rem machine language dis-assembler
110 rem       by: robert w. baker
120 :
130 print"[CLR]dis-assembler"
140 print"[DOWN][DOWN][DOWN]initializing ..
.."
150 dim m$(255): h$="0123456789abcdef"
160 for x=0 to 255: read a$: if a$="*" then
 a$="0*?*"
170 a$=left$(a$+"      .",6): m$(x)=a$: nex
t x: rem <-- 6 spaces
180 print"[CLR]enter decimal starting addre
ss[DOWN]"
190 print"or '$' followed by hex address[DO
WN]"
200 input a$: if left$(a$,1)="$" then 230
210 for x=1 to len(a$): c$=mid$(a$,x,1): if
 c$ < "0" or c$ >
```