FREEDOM ASSEMBLER 128

SYMBOLIC ASSEMBLER FOR THE 6502/10/8502/65C02 MICROPROCESSOR FAMILY

Freedom Assembler-128

HUGHES ASSOCIATES

45341 HARMONY LANE

BELLEVILLE, MI 48111

313-699-1931

TABLE OF CONTENTS

FREEDOM ASSEMBLER FOR THE COMMODORE 128 COMPUTER

The FREEDOM ASSEMBLER-128 is a powerful, symbolic assembler for the 6502/6510/8502/65C02 microprocessor family. It is professionally written in assembly language to give very fast assemblies and to leave the maximum RAM available for source file and object code.

The FREEDOM ASSEMBLER-128 will operate in the Commodore 128 (in 128 Mode) by inserting the cartridge, label up, in the expansion slot in the back of the C-128 computer. With the FREEDOM ASSEMBLER-128, microprocessor code can be written to execute in any 6502/6510/8502/65C02 based system, including the VIC-20, C-64, and C-128 as well as other hardware such as Apple IIe & IIc, Atari computers, video games, and industrial applications.

In addition to the base instruction set used with the 6502/6510/8502 NMOS microprocessors, the FREEDOM ASSEMBLER-128 correctly cross-assembles, disassembles, and "walks" 3 other instruction sets, including the sets used in 2 different versions of the CMOS 65C02.

The FREEDOM ASSEMBLER-128 is designed to operate on a source file created using the editor which comes with the C-128 or any other "tokenizing" editor expansion program designed for use with BASIC in the Commodore computers. The FREEDOM EDITOR for the C-128 works especially well with the FREEDOM ASSEMBLER-128.

OBJECT CODE can be assembled into a memory area different from where it is later intended to be used.

Excellent ERROR DETECTION has been built in, with specific error messages generated pointing to the line number in the source file where the error was discovered. Priority has been placed on not generating unintended code as a result of a syntax error.

In cases where two opcodes would both work (i.e. zero-page or not), the FREEDOM ASSEMBLER-128 will use the lower byte count zero-page opcode, unless told to do otherwise.

For those with a printer, a full-featured assembly listing PRINT UTILITY is provided, which includes an alphabetical cross-reference symbol list, keyed to line number and operation type. For those without a printer, or just for a quick look, cross-reference information can be displayed on the video screen. The assembler itself, being ROM chip based, uses very little RAM. The size of programs which can be handled by the FREEDOM

ASSEMBLER-128 is limited only by the amount of RAM available for the object file, and symbol table.

A LINE DISASSEMBLER is included, which displays opcodes, values, and raw hexadecimal (base 16) machine code starting at any desired location. It can be used independently of the assembler. When used after an assembly, it can disassemble code starting from a label name (no need to determine the address).

The FREEDOM ASSEMBLER-128 is designed to work in either 40 or 80 column mode.

Included within the FREEDOM ASSEMBLER-128 program are the following commands:

SYS 57000 Enter the freedom assembler-128

A Assemble from source

B Set/Display/Clear Breakpoint

C Cross-reference printout, printer or screen

D Disassemble

G Go (Execute code)

K Kill Freedom Assembler

L List specific cross-reference information.

M Monitor (C-128 resident monitor access)

P Assemble with Printout

Q Quicktrace code

R Register Display (Semi-Colon(;) Register Display Edit Mode)

W Walk machine code

Change instruction set mode

+ Checksum memory

> High Speed (2 MHz, default for 80 column display.)

< Normal Speed (1 MHz, default for 40 column display.)

OVERVIEW OF USING THE ASSEMBLER

(1) INSERT CARTRIDGE (label up!) with your computer turned off. Select either 40 or 80 column mode and turn your computer on. (Note: You must have a high resolution monitor and cable in order to use the 80 column mode.) The FREEDOM ASSEMBLER-128 is activated with the SYS 57000 (RETURN) command.

(2) CREATE THE SOURCEFILE. Before activating the cartridge, however, you need a sourcefile. Either create it from scratch or load a sourcefile in from Disk. If you create a new sourcefile, it is a good idea to save it to disk. (These steps can also be done without the cartridge installed.

(3) ENTER THE FREEDOM ASSEMBLER-128 BY TYPING:

SYS 57000 (followed by a RETURN)

The screen colors should change and his title screen should appear.

FREEDOM ASSEMBLER-128 C1986 JL HUGHES

READY

(4) COMMAND AN ASSEMBLY (base instruction set) by typing:

A
(followed by a RETURN)

The assembler title screen will appear.

FREEDOM ASSEMBLER

RUNNING--PASS 1

BANK 0 ,------BANK 1-----,
SOURCE XREF. LABEL OBJECT

The assembly will either complete successfully or stop at an error. It will display either:

FOUND END AT LINE # XXXXX

The assembly was successful!

Go on to step (6).

or:

ERROR #XX AT LINE # XXXXX

A SPECIFIC ERROR MESSAGE

There is an "assembly error"!

Go on to step (5).

(5) IF THERE WAS AN ERROR, go back to the editor and correct it.

Go back to the editor by typing:

K or KILL (either will work)
(followed by a RETURN.)

List the line number indicated, and correct the error.

Now go back to step (3).

(6) ONCE THE ASSEMBLY IS SUCCESSFUL you may wish to do other things such as:

(A) DO A PRINTING ASSEMBLY to get an assembly listing. Make sure the printer is turned on and type:

P
(followed by a RETURN.)

After pressing the RETURN, the assembly will proceed as before except it will do one more (much slower) pass while the listing is being printed.

(B) PRINT OUT ONLY THE CROSS-REFERENCE LISTING (Make sure the printer is on. TYPE:

C
(followed by a RETURN.)

If you wish to print the cross-reference list only to the screen, type:

C3
(followed by a RETURN.)

(C) DISPLAY SPECIFIC CROSS-REFERENCE INFORMATION to the screen by typing:

L"symbol"

(followed by a RETURN.)

After pressing the RETURN, the name, value, and all reference line numbers for the symbolic label will be displayed. To see the next label, push the down-cursor key, or hold down to scroll.

To go back to command level, press the RETURN key.

(D) SAVE THE MACHINE LANGUAGE CODE by typing:

M (This takes you to the C-128 resident Monitor)
(followed by return)

Then type:

S"OBJECT FILE NAME",d,1XXXX,1YYYY
(followed by a RETURN.)

d is device #, usually 1 for tape, 8 for disk.
1 is the bank number
XXXX is first HEXADECIMAL address to save.
YYYY is one past the last HEXADECIMAL address to save.

(E) DISASSEMBLE the code you have just assembled by typing:

D 1ZZZZ
(followed by a RETURN.)

1 is the bank number
ZZZZ is the HEXADECIMAL address of the first location to disassemble.

To disassemble the next instruction, push the cursor-down key or hold it down to scroll it onto the screen.

To go back to command level, press the RETURN key.

(F) USE ANY OF THE MANY MONITOR COMMANDS. (see MEMORY MANIPULATION AND DEBUGGING SECTIONS for these commands)

(G) SAVE THE SOURCE FILE (if you haven't saved the final version yet) by leaving the FREEDOM ASSEMBLER-128. TYPE:

KILL (RETURN)

SAVE"SOURCE FILE NAME",d (normal operation)

d is device #.

(H) RUN THE MACHINE CODE you have just assembled by using the MONITOR to transfer the code to a suitable location (such as $2000 through $4000 in bank 0), KILLing the FREEDOM ASSEMBLER-128, and typing:

SYS DDDDD (normal operation) (RETURN)

DDDDD is decimal address of first instruction of the object file.

The SYS COMMAND can be used to call Machine Language Subroutines from within a BASIC program. The SYS takes the place of a GOSUB in BASIC.

ASSEMBLER CONCEPTS

The purpose of an assembler is to operate on an assembly language SOURCE FILE (user-friendly but non-executable) to create an OBJECT FILE (user-unfriendly but executable by the computer). An assembler is fundamentally different from a compiler which translates a higher level language such as BASIC into machine code. Although compilers are very useful for quickly converting high level language code to machine code, they generate code which is inherently inefficient (slower executing and requiring more memory) compared to properly written assembly language programs.

For a "line assembler", as found in many monitor programs, the source file is a single line. For example:

LDA #$32

The opcode LDA is a mnemonic for "load the accumulator with the hexadecimal (base 16) value 32". Simple line assemblers convert each line separately. The above is assembled into the following two-byte object file:

A9 32

The FREEDOM ASSEMBLER-128 operates on a SOURCE FILE containing many lines such as those found in the example program in the APPENDIX. The source file is assembled as a whole instead of each line separately.

Since the FREEDOM ASSEMBLER-128 accepts symbols such as "NAME", it is a "symbolic assembler". This is a very important capability, which is almost mandatory for large assembly programs. The most important reason for using a symbolic assembler has to do with BRANCH and JUMP instructions. Adding or deleting instructions alters the absolute address of all subsequent instructions. It is, therefore, necessary to change the actual value of many of the branch and jump instructions in the program. With a symbolic assembler, and using an "assembler pseudo-opcode" called an ADDRESS LABEL, the correct branch and jump values are computed automatically, thus preserving the sanity of the software designer.

An ADDRESS LABEL is a name for a location in the program. In the FREEDOM ASSEMBLER-128, an AT (@) sign followed by a name in quotes signifies an address label. This must be entered on a separate line just before the instruction being referenced. For example, we can give a name to the address of a load like this:

3220 @ "HEREIS"

```
3230 LDA #$32
```

The line number 3220 does not create any object code. Instead it loads another area in memory called the SYMBOL TABLE with the proper memory address where the LDA #$ instruction A9 is placed. Because the assembler can then look up this address in the symbol table, it is possible to reference this location by name, only from anywhere in the program. For example, we could jump to that location like this:

```
1376 JMP $"HEREIS"
```

NOTE: In this example the JUMP instruction at line number 1376 will be encountered by the assembler before the address label is defined at line number 3220. This is not a problem because the FREEDOM ASSEMBLER-128 is a MULTI-PASS assembler. During the first pass, the assembler will simply use a dummy value for "HEREIS". On the second pass, the symbol table should contain the correct value for "HEREIS" and the instruction will assemble correctly. If, however, we have forgotten to put in line 3220, the assembler will pause on the second pass and tell us that there is an UNDEFINED LABEL @ LINE 01376. Because the FREEDOM ASSEMBLER-128 uses source files compatible with the resident editor, we can, at this point, "KILL" the assembler (with the K COMMAND), add the required line to the source file, SYS 57000, back into the FREEDOM ASSEMBLER-128 and try another assembly -- all in a matter of seconds.

SYMBOLS are useful for many other purposes than just address labels. Another use is to help the software designer remember what he/she was trying to do with a certain instruction. The use of a descriptive name can save hours in trying to understand a section of code. For this reason, the FREEDOM ASSEMBLER-128 accepts labels containing from one to six characters, which may be anything except the quotation mark ("), which is used to signal the start and end of the label (other punctuation marks, numbers, etc. are OK). Symbols also allow things to be changed easily by redefining the symbol in one place, even though it may be referenced many times in the program.

Another important benefit of using SYMBOLS is that every reference to a symbol will be listed in the CROSS-REFERENCE list. Without this capability it would be very difficult to follow the code in any sizable program.

The FREEDOM ASSEMBLER-128 also accepts arithmetic operations with labels and numbers mixed in a string. For example, the previous load could have been:

```
3230 LDA #$"BEGBLK"-"ENDBLK"+1
```

Experienced software designers use symbolic labels for almost all values.

When a program is being run, the computer keeps track of where it is by means of a 2-byte register called the PROGRAM COUNTER. This is continually loaded with the address of the next instruction to be run. During an assembly the assembler must maintain a simulated program counter to keep track of where the computer would be if actually running the object code. This is called the **INSTRUCTION LOCATION COUNTER.**

A special feature of the FREEDOM ASSEMBLER-128 allows assembly to a different address than where the code is intended to be run. In order to allow this feature, there is a second assembler counter called the **STORAGE LOCATION COUNTER** which determines where the object code is stored during assembly. These counters may be initialized equal or not equal to each other by the sourcefile. During assembly whenever one counter is incremented, so is the other.

In the process of developing an assembly language program, it is normal to make many revisions to the source file and run many test assemblies. During this process, only the source file needs to be saved (and backed up!), since the object file can be quickly recreated by the assembler. Once the program has been completed, only the object code is required to run the program. The source file should be saved in case future changes are needed, to help debug any problems in the program, or to help answer questions about the program.

CREATING THE SOURCEFILE

To create the source file use the normal Commodore editor. Enter the lines with line numbers in the same way as for a BASIC program. A sample program is shown in the Appendix. It may be helpful to refer to the sample program for clarification of some of the details to be described further on. If you are already experienced in using an assembler, much of the syntax may be obvious.

$ OPERATOR -- VALUE STRINGS

Value strings are used to tell the assembler what value to assemble for the operand of an instruction. Take the following lines from the Example Program found in the APPENDIX:

```
740 CLC
750 ADC #$01
780 AND #$%00001111
790 CMP $"COLOR"
```

NOTE: This format is simply standard 6502 opcodes with a string after the DOLLAR SIGN ($) specifying what value to use.:

```
CLC clean carry bit of PSW
ADC #$ add to accumulator with carry--immediate
AND #$ logical AND with accumulator--immediate
CMP $ compare with accumulator--absolute
```

With the FREEDOM ASSEMBLER-128 the **DOLLAR SIGN ($) SYMBOL IS REQUIRED** to signify the start of a value string whenever the standard opcode includes a $ sign. See the OPCODE REFERENCE INFORMATION (in the APPENDIX) for a list of these opcodes. The value string is ended by either a space or a comma (spaces or commas inside a label do not end the value string however).

! AND % OPERATORS -- NUMBER BASES

THE DEFAULT BASE FOR NUMBERS IN THE FREEDOM ASSEMBLER-128 IS HEXADECIMAL (base 16). There is, however, a provision for entering numbers in decimal (base 10) or binary (base 2). For example the Hex instruction:

```
100 LDA #$32
```

becomes in decimal:

```
100 LDA #$!50
```

is written in binary as:

```
100 LDA #$%00110010
```

or also in binary as:

```
100 LDA #$%110010
```

All of the above are equivalent. The exclamation point (!) signifies DECIMAL. The percent sign (%) signifies BINARY.

NOTE: that the $ sign was still required.

STRING ADDITION AND SUBTRACTION

The FREEDOM ASSEMBLER-128 will also accept a value string of numbers and/or labels separated by PLUS (+) or MINUS (-) signs. Our previous example could have also been:

```
100 LDA #$"ENDBLK"-"BEGBLK"+!92-40-%1011
```

NOTE: The 40 in the above example would be interpreted by the assembler as hexadecimal. The effect of the % or ! sign is terminated by either a + or - sign, as well as by a comma or space, thus allowing mixed mode strings. Note also that there must be no spaces until the end of the TOTAL VALUE STRING. There is no provision for multiplication or division. Numbers must be integers only (no fractions or decimal points).

In the above example, if the value string is computed by the assembler as greater than 255 (FF hex), a "BAD VALUE" error is generated. The assembler knows that the instruction LDA #$ is a two-byte instruction that can only have a single byte operand.

NEGATIVE NUMBERS

If the value string computes as negative, the result will be a "2's complement" number. The following are equivalent:

```
100 LDY #$-3
100 LDY #$FD
```

< AND > OPERATORS -- ONE BYTE OF WORD

In many cases it is necessary to load either the MOST SIGNIFICANT (high) or LEAST SIGNIFICANT (low) byte of a two-byte value. The GREATER-THAN SIGN (>) signifies the most significant byte. For example, these two are equivalent:

```
100 LDA #$>89AB
100 LDA #$89
```

For loading the least significant byte, the LESS-THAN SIGN (<) is used. These two are equivalent:

```
100 LDA #$<89AB
100 LDA #$AB
```

In the case of long value strings the < or > signs are understood by the assembler as applying to the whole string. For example, these three are equivalent:

```
100 LDA #$>89AB-70
100 LDA #$>893B
100 LDA #$89
```

The example above does not reduce to:

```
100 LDA #$19
```

Of course, the real usefulness of the < and > operators is with symbols.

£ OPERATOR -- FORCE WORD OPERAND

The standard opcode mnemonics for the 6502 are somewhat ambiguous, in that the same mnemonic can often stand for two possible opcodes. For example, the instruction:

```
LDA $,X
```

could be either a 2-byte "zero-page" instruction or a 3-byte "non-zero-page" instruction. In this case, if the address for the data to be loaded into the accumulator is greater than FF (hexadecimal), then the assembler knows that the

3-byte instruction is required. In most cases, if the address denoted by the value string is less than 100 (hexadecimal), the "zero-page" instruction will be equivalent, so the assembler will generate the more memory efficient 2-byte instruction, if it exists. In a few cases, the two instructions do not give equivalent results. For this reason, the ENGLISH POUND SIGN OPERATOR (£) (which prints as a backslash (\) on some printers) has been provided. The £ operator tells the assembler that the value string should be treated as greater than FF (hexadecimal) even if it is not. For example, the instruction:

```
8730 LDA $£00F8,X
```

will assemble as:

```
BD F8 00
```

and in the case of X=8, will load the contents of location F8+08=0100 into the accumulator.

On the other hand, the instruction:

```
8730 LDA $00F8,X
```

will assemble as:

```
B5 F8
```

and in the case of X=8, will wrap around on the zero page and load the contents of location 0000 into the accumulator.

Depending on the circumstances, either opcode may be desired.

' SINGLE QUOTE -- ASCII BYTE OPERATOR

The SINGLE QUOTE MARK (') will cause a Commodore ASCII value to be assembled--Single Byte Only. Example:

```
LDA #$'A
```

is the same as:

```
LDA #$41
```

@ OPERATOR -- PRESENT LOCATION

For convenience the AT (@) operator can be used within a value string to signify the value of the "object program counter" at the beginning of the current instruction. For example, the following instructions, when executed, would cause the processor to skip the instruction at line 5960 if the "negative bit" in the "Program Status Word" is set.:

```
5950 BMI $@+3   (2-byte instruction)
5960 INY        (1-byte instruction)
                ---------------
                3-bytes must be added
```

REM PSEUDO-OP -- REMARKS

The only other thing that the assembler will allow on the same line with an opcode is a remark.

A REMARK is signified by the three characters REM on a line. As mentioned above, a remark can be on the same line with an opcode:

```
100 LDA #$32 REM#50 ITEMS TO START
```

NOTE: There must be a space between the operand and the R in REM. Long remarks should be put on a separate line:

```
98 REMLOAD 50 ITEMS TO SET UP THE LOOP
```

The Commodore editor will convert the three characters REM to a single byte "token", thus saving memory and speeding assembly. Any expansion editor compatible with Commodore BASIC (such as the FREEDOM EDITOR-128) will do the same thing. In fact, these expansion editors are very helpful in creating source files. "Non-tokenizing" editors supplied with some other assemblers are <u>not compatible</u> with the FREEDOM ASSEMBLER-128.

When doing a PRINTING (P) assembly, the "token" for REM will be converted to a space, so that only the actual remark will be printed.

SYMBOLIC LABELS

A symbolic label is simply a name which represents a value. It is from 1 to 6 characters enclosed in quotation marks.

Any characters are allowed in a label except quotation marks. The label must be unique (not defined twice with different values). A statement with a label in it must be either defining the value of the label or using the value of the label. The following pseudo-ops all DEFINE THE VALUE OF A LABEL:

=	Equate
@	Address label
DEF	Define memory
DEFW	Define memory--double byte (word)

The following pseudo-ops all may USE THE VALUE OF A STRING CONTAINING A LABEL:

USR	Origin
TO	Assembly origin
GOTO	Fill memory
POKE	Direct load

And of course a label may be used as all or part of a value string in an opcode.

= PSEUDO-OP -- EQUATES

An EQUATE (=) is simply a statement that directly sets the value of a label. The format is:

```
line# =$valuestring,"A NAME"
```

For example:

```
170 =$033C,"TBUFFR"
```

A label can even be "equated" to a value string containing another label. For example:

```
309 =$8000,"BASE"
310 =$"BASE"+127,"TABSTT"
320 =$"BASE"+146,"FILSUB"
```

The above example is equivalent to:

```
310 =$8127,"TABSTT"
320 =$8146,"FILSUB"
```

The above is a useful trick for changing many label values by changing only one line.

@ PSEUDO-OP -- ADDRESS LABEL

An ADDRESS LABEL is the name for a location in memory. It is defined by an @ sign followed by the desired label (name of the location enclosed in quotation marks). Examples of using an address label is:

```
5870 @ "WAITHR"
5880 JMP $"WAITHR"
```

(If executed, the above code would cause the processor to run in place, waiting for an interrupt.)

When the assembler encounters an ADDRESS LABEL, it loads the SYMBOL TABLE with the label name and the current value of the INSTRUCTION LOCATION COUNTER. The INSTRUCTION LOCATION COUNTER is a counter that the assembler maintains to mimic what the actual program counter would be during execution of the object code.

The space after the @ sign is optional.

DEF AND DEFW PSEUDO-OPS -- DEFINE MEMORY

A location in memory may be given a name using the DEF or DEFW "pseudo-op". This is usually used to set-up RAM Locations as scratchpad memory for the object program. For example:

```
130 USR $100 REM*START RAM AT START OF STACK AREA
140 DEF "XSAVE"
150 DEFW "CNTLOW" REM*2-BYTE
160 DEF "YSAVE"
```

The above example would be equivalent to:

```
140 =$100,"XSAVE"
150 =$101,"CNTLOW"
160 =$103,"YSAVE"
```

When the assembler encounters a DEF pseudo-op, it loads the SYMBOL TABLE with the name of the label and the current value of the INSTRUCTION LOCATION COUNTER. This is similar to the address label, except that, for a DEF, the assembler adds 1 to the INSTRUCTION LOCATION COUNTER. A DEFW is handled the same, except that 2 is added to the INSTRUCTION LOCATION COUNTER, thus reserving memory space for a word

(double-byte).

NOTE: Only the SYMBOL TABLE is written to by these pseudo-ops (No actual machine code is stored.).

If desired, DEFB is accepted by the assembler as equivalent to DEF.

USR & TO PSEUDO-OPS -- ORIGIN

The USR pseudo-op is used to tell the assembler at what address you wish the object code to start, when actually being used. The format for this pseudo-op is:

```
line# USR $valuestring
```

The TO pseudo-op is used to tell the assembler where you want to start storing the object file during assembly. In many cases this is the same as the USR address. The format for this pseudo-op is:

```
line# TO $valuestring
```

In some cases, where it is desired to use and load the code at the same address, the TO pseudo-op is not needed, since it is generated automatically by the USR command. Because of this, when the TO pseudo-op is used, it should come after the USR pseudo-op.

Some examples of using these pseudo-ops are:

```
1380 USR $A000
```

OR:

```
1380 USR $"BASE"
1390 TO $6000
```

If a label is used in the value string it should be defined before use in a USR or TO pseudo-op.

The default address for the USR and TO values on the Commodore 128 is 6000 (hex) in memory bank 1.

The USR pseudo-op may be used, as described, for the DEF and DEFW pseudo-ops, to set the starting location of the scratchpad memory. Later, however, the USR pseudo-op MUST BE USED (before any instructions that create object code) to define the start of assembly, since the default for USR will be overridden.

RELOCATED ASSEMBLY

A feature of the FREEDOM ASSEMBLER-128 is the ability to assemble code for a location that is DIFFERENT from where it is stored during the assembly. The most usual reason for doing this is that free RAM does not exist in the address area where the program will actually be used later. This is often the case when the target application is not the Commodore computer (maybe you're writing an Atari game) or in preparing object code for "burning" EPROM chips.

Of course if object code is stored in a location different from where it is intended to be used, it is not likely to operate properly in its temporary storage location. In other words, it will not be executable until it is properly located.

The USR pseudo-op controls the memory where the object code is intended to be used AFTER ASSEMBLY. The TO pseudo-op controls the memory where the object code is stored DURING ASSEMBLY.

As an example, you wish to write code to execute on a C-128 starting at address 2000 (hex). Since the XRF table might need to occupy this area, you could store the OBJECT code at $6000, then use the resident monitor to transfer the object code to $2000 in Bank 0, BEFORE TRYING TO RUN IT. For example:

```
180 USR $2000
190 TO $6000
```

LIST PSEUDO-OP -- LOCATING THE SYMBOL TABLE

The FREEDOM ASSEMBLER-128 stores the symbolic names and associated values in the symbol table. This requires 8 bytes per label. The symbol table must be located in a safe place during the assembly, so RAM Bank 1 is used. The starting location of the symbol table within Bank 1 is controlled by the LIST pseudo-op. The format for this pseudo-op is:

```
line# LIST $value
```

The LIST pseudo-op should be used BEFORE any labels are defined, and it cannot reference any labels. It is suggested that only remarks (REM) precede a LIST pseudo-op.

An example of using the LIST pseudo-op is:

```
100 LIST $7800
```

As with the USR and TO pseudo-ops, it is not always necessary to use the LIST pseudo-op, since the FREEDOM ASSEMBLER-128 has an intelligent default. On the Commodore 128 the default for the start of the symbol table is address A000 in memory bank 1.

The LIST pseudo-op should not be used more than once in a source file.

LOAD PSEUDO-OP

This pseudo-op is used when you wish to load in a file within a program. For example:

```
1500 LOAD "FILENAME",8
```

This loads the file temporarily during each pass and assembles it as though it was inserted in the source file at the position of the LOAD pseudo-op. An END pseudo-op in the file will terminate assembly of that file only and cause assembly to continue at the next line in the main source file.

The LOAD pseudo-op allows very large programs to be assembled since the entire source file does not have to fit in the C-128 RAM.

A source file may even consist only of lines containing LOAD pseudo-ops, however, do not attempt to nest the LOAD sseudo-op.

POKE PSEUDO-OP -- DIRECT MEMORY STORAGE

Any value from 0 to 65535 (FFFF hex) can be loaded directly into the object file by using the POKE pseudo-op. The format for the POKE pseudo-op is:

```
line# POKE $valuestring
```

The POKE pseudo-op will load 1 byte if the valuestring is less than or equal to 255 (FF hex). Otherwise 2 bytes will be loaded. It is important to note that in line with 6502 convention, a value loaded into 2 bytes will be loaded least significant byte first. For example, the following line:

```
2780 POKE $1234
```

will assemble as:

```
34 12
```

The ENGLISH POUND sign operator (£) can be used for the special case where it is necessary to load 2 bytes with a value less than or equal to 255. For example, the following line:

```
2790 POKE $£0056
```

will assemble as:

```
56 00
```

whereas this line:

```
2800 POKE $0056
```

will assemble as:

```
56
```

Since the INSTRUCTION LOCATION COUNTER and the "storage location counter" are kept in step by the POKE pseudo-op, it may be used as many times as desired without resetting the location with USR or TO.

ASC PSEUDO-OP — LOADING ASCII

The values corresponding to "Commodore ASCII" can be directly loaded into the object file by using the ASC pseudo-op. The format for this pseudo-op is:

```
ASC "ANYTHING YOU TYPE"
```

For example:

```
5470 ASC "6502 ASSEMBLER"
```

For capital letters, numbers, and most punctuation, Commodore ASCII is the same as standard ASCII. For other characters, some experimentation may be required.

Quotation marks may be included within the outer quotation marks, unlike with labels.

The ASC pseudo-op keeps the location counters in step.

GOTO PSEUDO-OP -- FILLING MEMORY

The GOTO pseudo-op will continue to store the value of the last byte stored into the object file until just before the address specified by a value string. For example, to load NOPs (EA) from the current location through CFFF (hex) we could do this:

```
5780 NOP
5790 GOTO $D000
```

We could fill a space with FF (hex) like this:

```
5780 POKE $FF
5790 GOTO $D000
```

The GOTO pseudo-op keeps the location counters in step.

In cases where the TO pseudo-op was used to make the INSTRUCTION LOCATION COUNTER different from the STORAGE LOCATION COUNTER, it is the INSTRUCTION LOCATION COUNTER which is compared with the operand of the GOTO pseudo-op.

The GOTO pseudo-op also provides a check for object code assembling beyond a desired point. An error is generated if already beyond the GOTO address.

↑ PSEUDO-OP -- PAGE EJECT

The UP ARROW (↑) pseudo-op is provided for the purpose of allowing a form feed when doing a PRINTING (P) assembly. For example:

```
1368 ↑
```

INTERPRETING THE ASSEMBLY DISPLAY SCREEN

After entering the FREEDOM ASSEMBLER-128 command mode by using the SYS 57000 command and then starting an assembly by using A (non-printing) or P (printing), you will see a display in the upper part of the screen that looks like this:

```
FREEDOM ASSEMBLER

COMPLETE-PASS 2

BANK 0  ,-----BANK 1---,

SOURCE XREF. LABEL OBJECT

$1C01  $0400 $A000 $6E10

$2336  $04C3 $A080 $6E59

FOUND END @ LINE 00870
```

The second line tells you what PASS the FREEDOM ASSEMBLER-128 is currently running. A NON-PRINTING (A) assembly usually requires two passes. In unusual circumstances, extra passes may be required to give a correct assembly. This is perfectly okay, and is a feature of the FREEDOM ASSEMBLER-128. Many other assemblers "bomb" under these circumstances. A PRINTING (P) assembly will always take one more pass, during which the printout occurs providing the printer is turned on.

The last line does not appear until the assembly is either complete or aborted due to an error. If the "FOUND END..." message appears as shown above, the assembly completed successfully. If, however, an "ERROR #0X..." message appears, there was an assembly error, and the assembly was terminated at the line shown. If there was an error, an explicit error message and the line on which it occurred will be printed, for example:

```
FREEDOM ASSEMBLER

HALTED--PASS 2

BANK 0  ,-----BANK 1---,

SOURCE XREF. LABEL OBJECT
```

```
$1C01  $0400 $A000 $6000

$1C23  $0405 $A000 $9FFF

ERROR #8 @ LINE 00120 OBJ RANGE ERROR
```

If an error (other than ERROR #5) was encountered, a printout listing will not be made.

The third line lists DYNAMIC MEMORY AREAS used in an assembly. The fourth and fifth lines give the START and FINISH addresses (in hexadecimal) for each memory area.

* The SOURCE AREA is the memory area used by the source file. Both the start and finish addresses of the source file are displayed at the beginning of the assembly.

* The XRF AREA is the memory area used to create the cross-reference list. This file starts writing in pass 2 at (1)400. The FREEDOM ASSEMBLER-128 will not allow this file to write over the OBJECT or LABEL areas. If the XRF file is truncated, which is indicated by a plus (+) sign on the screen, the only harm done is that some cross-reference information will not be shown.

* The LABEL AREA is the memory area used for the symbol table. These addresses will change during the assembly as symbols are defined. The start of the LABEL area is controlled by the LIST pseudo-op.

* The OBJECT AREA is the memory area used to store the object machine code during the assembly. These addresses will change during the assembly as code is assembled. The start of the OBJECT area is controlled by the USR or TO pseudo-op.

MEMORY MANAGEMENT CONSIDERATIONS

The C-128 computer contains 2 full 64k banks of RAM, designated as bank 0 and bank 1. In addition various other logical banks are obtained by combining parts of bank 0 or 1 with parts of the operating system ROM, internal or external function ROM, and/or

I/O memory.

When using the FREEDOM ASSEMBLER-128, bank 0 is used only for the source file. The 56k of memory in bank 0 from $1C01 to $FF00 can contain a source file sufficient to create about 4k of object code. Since this is insufficient for many large programs, the LOAD pseudo-op has been provided to allow assembly from source files on disk. With the LOAD pseudo-op there is no practical limit to the size of the source file.

Bank 1 is used to store the OBJECT, LABEL, and XREF files. The LABEL and OBJECT files are usually about the same size, with the XREF file being somewhat larger. The full capacity of bank 1 will be reached for all 3 of these files, when the OBJECT file reaches about 16k, which is a very large assembly language program. If necessary, it is possible to create even larger programs by realizing that the XREF file is not strictly necessary to do an assembly; it is only used to create the cross-reference information. If necessary, locate the OBJECT and LABEL files as required; the assembler will not allow the XREF file to overwrite either of these.

The normal operating bank of the computer is bank 15 (F hex), which combines bank 0 up to $3FFF with ROM from $4000 on up, except for the I/O block from $D000 to $DFFF. The easiest place from which to execute programs is in this bank 0 RAM below $4000. Since the assembler can only assemble directly into bank 1, it is necessary to use the resident monitor to transfer this code before running it. This is very easy and fast.

OBJECT files saved to disk from bank 1 will always load into bank 0, since the disk binary program format does not save the bank number. Before saving, however, the file should be transferred to the address range where it is to be run, since it will load into the same location from which it was saved.

MEMORY MANAGEMENT SUMMARY

If all of the previous explanation seems complicated, there is no cause for worry. The FREEDOM ASSEMBLER-128 has intelligent defaults which make it easy to get started. Assuming that you have created a source file and saved it on tape or disk, no real harm can be done by trying a test assembly, perhaps without any LIST, USR, or TO pseudo-ops in the SOURCE file. This will allow the defaults to operate.

EXPLANATION OF ERROR CODES

When an assembly error occurs, the assembly is usually terminated at the line where the error was discovered. When this happens, the following information is given:

* The assembler pass during which the error was discovered.
* Error reference #.
* Source line # where error was discovered
* Brief error description.

The following information provides an expanded explanation of the nine types of error messages.

ERROR #01 — PAST GOTO VALUE

The object of the GOTO command is to fill a range of memory up to, but not including, the address given by the value string with the same data used in the previous address. Before the assembler fills the next location, it checks to see if it has reached it. ERROR #01 is issued if it is already past the specified address.

ERROR #02 — ILLEGAL OPCODE

Before the assembler tries to assemble a line as an opcode, it first looks for all legal pseudo-ops. If the line is not a legal pseudo-op, the assembler will try to match it with a legal opcode. ERROR #02 is issued if the assembler cannot find any matching legal pseudo-op or opcode. Hence there are a number of possibilities:

* An attempt was made to use an illegal pseudo-op.
* An attempt was made to use a non-existent opcode.
* The software designer was designing for one of

the alternate instruction sets, but forgot to use the # command to override the default base 6502 instruction set.

* The software designer had the right idea, but typed a bad character.

* The opcode attempted exists for a value less than 256, but the value string computed is greater than 255.

* A value string contained a space (remainder of value string was considered opcode by assembler).

ERROR #03 -- ILLEGAL VALUE

When evaluating a value string, the assembler looks for obviously incorrect input. ERROR #03 will be issued if:

* A hexadecimal value contains digits other than 0-9 and A-F.

* A decimal value contains digits other than 0-9.

* A binary value contains digits other than 0 and 1.

* An "immediate" mode opcode was attempted, but the value string computed was greater than 255.

* A hexadecimal value contains more than 4 digits.

* A decimal value contains more than 5 digits.

* Any subvalue of the value string is greater than FFFF (hex), 65535 (decimal), or 1111111111111111 (binary).

* A value string is not properly terminated (space, comma, or end of line).

* One of the operators (<, >, £, %, !) was encountered after the beginning of a value within a value string.

* A quotation mark was omitted at the beginning of a label.

ERROR #04 -- BRANCH TOO LONG

The branch instructions are limited in range to between +127 and -128 locations from the beginning of the next instruction. ERROR #04 is issued if the branch value required exceeds this range. To correct this either rearrange the code or rewrite it to use a JMP $ (which has total span of the memory).

* Another possibility is that the label is defined twice, but the assembler has not discovered it yet, and is trying to branch to an unintended location.

ERROR #05 -- UNDEFINED LABEL

All labels must be defined somewhere in the SOURCE. ERROR #05 is issued if an attempt is made to use a label that is not defined. NOTE: This is the only error that does not immediately terminate the assembly. This error will only occur on the second pass. Several possibilities exist:

* The label has been misspelled on the line number referenced or the defining line.

* The label name was not properly closed with a quotation mark(").

* The line required to define the label was omitted.

In the case of having multiple undefined labels, only the first 10 will be shown.

ERROR #06 -- LABEL TOO LONG

Labels cannot exceed 6 characters in length. ERROR #06 will be issued if the assembler encounters a 7th character before encountering the closing quotation mark. Several possibilities exist:

* An attempt was made to use too long a label.

* The label was not properly closed with a quotation mark (").

* A quotation mark was used without any intention to indicate a label.

ERROR #07 -- DEFINE CONFLICT

Normally labels are defined in only one location, however this assembler will accept multiple definitions as long as the values agree. ERROR #07 is issued if the same label is defined in several places with DIFFERENT values.

ERROR #08 -- OBJ RANGE ERROR

Object overflowing into label area or <$2000 or >$FF00

ERROR #09 -- BAD LABEL RANGE

Label address <$2000 or >$FF00

CREATING A LISTING PRINTOUT

To get a printout of an assembly listing of your program, it is necessary to substitute the P command for the A command.

When the P command is used for an assembly, an extra assembly pass will be made. Assuming a printer is powered up and defined as device #4, a listing will be printed.

P (RETURN)

In order to accommodate various device #s and types of printers, an expanded form of the P command can be used to redefine the device #, etc. The P command is equivalent to:

P4,FF

(NOTE: FF is equal to a null secondary command)

This command sets the file # to 4, the device # to 4, with no "secondary command" If it is necessary to change these defaults, the command may be typed out in full, with appropriate changes. Because of the many different printers and interfaces available, some experimentation may be required.

NOTE: The first number makes the file and the device number the same (only one number is needed for both.)

The listing printout may be sent to the screen by typing P3 (3 is the device # for the screen).

INTERPRETING THE LISTING PRINTOUT

The listing will display the lines from the SOURCE file along with the assembled code, addresses and cross-reference information. Refer to the printout listing of the example program in the Appendix.

Starting at the left side of the listing, the first column is the SOURCE file line number. Next will be the opcode or pseudo-op from the SOURCE file, and/or a remark. On the right side, the

leftmost column is the value of the STORAGE LOCATION COUNTER, followed by the value of the INSTRUCTION LOCATION COUNTER.
NOTE: These may differ when the TO command differs from the USR command. These are 4-digit hexadecimal numbers preceded by a $. Last comes the actual assembled code, listed as a 2-digit hexadecimal number for each byte. For example:

```
00200 =$D021,"BKGRND" *BACKGROUND COLOR(40C) $6000  $6000

OR:

00360 LDA $"BKGRND"                          $6E14  $0E14 AD 21 D0
```

After the last line of the SOURCE file has been assembled, the FREEDOM ASSEMBLER-128 will continue on to print an alphabetical (or more correctly numero-alphabetical) cross-reference listing of all labels. Starting at the left is the label name followed by its 4-digit hexadecimal value preceded by a $. After that comes a list of all line numbers where that label was referenced in the program. For example:

```
00870 ******************
ADD&CP  $0E34  J 00390  J 00540  @ 00720  B 00800
BKGRND  $D021  = 00200  L 00360  S 00420
```

A feature of the FREEDOM ASSEMBLER-128 is that in front of each line number is a letter or symbol indicating how the label is referenced IN THAT LINE. This symbol is typically the first letter of the opcode or the printing symbol for a pseudo-op as described above. In the above example:

J is JSR
S is STA
B is BEQ
@ is address label
= is equate

(Refer to the actual code lines in the example program in the APPENDIX.)

The cross-reference listing can be printed out separately by using the C command, assuming that an (A) assembly has been done already.

C (RETURN)

As with the P command, there is an expanded form of the C command in order to accommodate various device #'s and types of printers. For example, to print the cross-reference list to a printer defined as device #5 use the command:

C5,FF

(Where the FF is equal to a null secondary command)

TYPING:

C3 (RETURN)

will cause the CROSS-REFERENCE listing to be displayed on the screen. This can be halted temporarily with the "NO SCROLL" key.

> COMMAND--HIGH SPEED

The High Speed (2MHz) is the default state for assemblies in 80 column mode.
The High Speed Command (>) can be used before typing A for an Assembly or P for a Printing Assembly in 40 column mode.

NOTE: You can switch back and forth between the fast and normal speeds in either 40 or 80 column mode by issuing the > or the < commands.

> (RETURN)

Use of the > COMMAND puts the computer into high gear giving extremely fast assemblies. Screen blanking will occur in 40 column mode during assembly when this command is in effect.

In 80 column mode the screen display of LABEL and XREF addresses is not continuously updated in the High Speed Mode.

< COMMAND--NORMAL SPEED

< (RETURN)

The Normal Speed (1MHz) is the default state for assemblies in 40 column mode.

Use of this commands will eliminate screen blanking in 40 column mode and allow LABEL and XREF addresses to be continuously updated on the screen in 80 column mode.

USING THE DISASSEMBLER

As an aid in developing assembly language programs, the FREEDOM ASSEMBLER-128 also contains a DISASSEMBLER.

After entering command level by using the SYS 57000 command, one can disassemble any machine language in the computer using the following command followed by a RETURN:

D valuestring

The D COMMAND will accept up to 5 hex digits. If there are 5 digits the first digit must be the bank number (0-F). The default is bank 1 (where the object code is located.) If 4 hex digits or a symbol name is used, the current bank will be maintained.

The quantity "valuestring" can be a single number, or a string of mixed base numbers. In fact, if you have run an assembly to create a LABEL file, the valuestring can even contain LABELS. The rules for the valuestring are just like those for using a valuestring in a SOURCE FILE for assembly, except that it is not necessary to start out with a $:

* A space after the D is optional.
* A leading $ is optional.
* Default base is hexadecimal.
* % is for binary.
* ! is for decimal.
* No spaces till the end of the valuestring.
* Labels need quotation marks (") around them

If you have just assembled the example program and wish to see how it assembled starting at address label "INC80B", just type:

D "INC80B" +6000 (RETURN)

The computer will respond:

16E1E LDX #$1A A2 1A

Now press the down cursor to get:

16E20 JSR $0E4E 20 4E 0E

And again to get:

16E23 PHA 48

To get back to command level press RETURN.

NOTE: The disassembly display starts from the left with the bank number and address, followed by the opcode and numeric operand, and finally 1 to 3 bytes of raw machine code, all in hexadecimal.

The D COMMAND can be used to convert decimal or binary numbers to hexadecimal. For example, to convert the decimal number 31346 to hexadecimal, type D!31346. The address displayed by the disassembler will be the hexadecimal equivalent 7A72.

It is possible to print the disassembly display on a printer. BEFORE entering the FREEDOM ASSEMBLER-128 with a SYS 57000, type:

```
OPEN 4,4:CMD4 (return)
```

Then use the SYS 57000 command to enter the FREEDOM ASSEMBLER-128. All computer responses will go to the printer. When you are through printing, KILL the FREEDOM ASSEMBLER-128, then close the printer channel. Type:

```
K (return)
PRINT#4:CLOSE4 (return)
```

USING THE MEMORY MANIPULATION COMMANDS

M COMMAND--MONITOR

THIS COMMAND TAKES YOU TO THE RESIDENT C-128 MONITOR. FROM THERE YOU CAN ACCESS ALL THE COMMANDS FOUND IN THE MONITOR SECTION OF YOUR COMMODORE USER'S MANUAL, APPENDIX J, PAGE 369.

M (RETURN)

X (RETURN) Will take you back to the FREEDOM ASSEMBLER.

+ COMMAND--CHECKSUM MEMORY

The memory checksum command (+) is used to add up all memory bytes within a certain range. This is often used to verify that the bit pattern of a program has not changed. If any location were to be altered, it is very unlikely that the checksum would remain the same.

To checksum a section of memory, for example in bank 0 from $4000 up to and including $5FFF, type the following:

+ 04000 -06000 (followed by a RETURN)

After about a second, the computer will respond with a 3-byte checksum, such as:

10E23B

Three bytes is enough to contain any possible value. In many cases, only the least significant byte would be used.

USING THE DEBUGGING COMMANDS

B COMMAND -- SETTING/DISPLAYING/CLEARING BREAKPOINT

A BREAKPOINT is a flag at a certain address that tells the computer to stop. This is useful in debugging software. Some uses of a breakpoint are:

* To see if the program is actually getting to a certain point in the code.

* To stop the program and examine the registers and memory contents at a certain point.

* To stop the program so that it may be "walked" (single stepped) through a questionable portion of the code.

There are actually two different types of breakpoints that are used in debugging. These are referred to as "hardware" and "software" breakpoints.

* A HARDWARE BREAKPOINT is obtained by actually writing a 00 (BRK) instruction into the code. This is the only kind of breakpoint which can be used in executing code at full speed.

  NOTE: A HARDWARE BREAKPOINT can usually only be used for debugging code in RAM, since ROM cannot easily be altered.

* A SOFTWARE BREAKPOINT is obtained when executing code one instruction at a time by having the debugging program constantly compare the program counter with the desired breakpoint. Hence a SOFTWARE BREAKPOINT is useful in debugging RAM or ROM based software, but not during full speed operation.

The B COMMAND, when used with an address, simultaneously sets both a "hardware" and a "software" breakpoint. For example:

To set a breakpoint at location $1734 in bank 0, type:

B 01734 (followed by a RETURN)

The above command will load a BRK instruction into RAM at location $1734 in bank 0 and also load that address into memory for use as a "software" breakpoint with the QUICKTRACE and WALK commands.

To display the breakpoint, type:

B (followed by a RETURN)

To clear the breakpoint, type:

BX (followed by a RETURN)

The program will replace the BRK instruction in RAM with the original value, which was saved at the time the breakpoint was set. The breakpoint flags used for the SOFTWARE BREAKPOINT will also be cleared. Following the clearing of the breakpoint the program will respond with "OK". In the event an attempt is made to clear the breakpoint when it was not set, the message "BREAKPOINT NOT SET" will be displayed.

ONLY ONE BREAKPOINT AT A TIME CAN BE USED. If a new breakpoint is requested without clearing the previous one, it will be cleared automatically. If it is desired, the resident monitor (M COMMAND) can be used to set multiple HARDWARE BREAKPOINTS by loading 00 values into various addresses in RAM.

R COMMAND -- REGISTER DISPLAY/EDIT

The FREEDOM ASSEMBLER-128 program maintains a set of pseudo-registers for use with the debugging commands. Although the actual machine registers are constantly changing, the pseudo-registers can represent the state of the registers at critical times. These pseudo-registers are used to:

* Provide values to load into the real registers to begin code execution with the GO, QUICKTRACE, and WALK commands.

* Contain the contents of the real registers after encountering a breakpoint.

* During single stepping (WALK and QUICKTRACE modes), contain the contents of the real registers that would exist after each instruction is executed.

The R command can be used to display the contents of the pseudo-registers at any time. For example, the following command may be typed immediately after entering the FREEDOM ASSEMBLER-128:

R (followed by a RETURN)

The following will be displayed:

```
  A. X. Y. NV-BDIZC SP .PC.
; 00 00 00 00100000 F8 1FFFF BRK
```

Moving from left to right, the meaning of these is:

```
;  put in for ease of editing the register contents
A. contents of Accumulator
X. contents of X register
Y. contents of Y register
Program Status Word bits
------------------------
N  status of "negative" bit
V  status of "overflow" bit
-  undefined bit--always 1
B  status of "break" bit
D  status of "decimal" bit
I  status of "interrupt" (disable) bit
Z  status of "zero" bit
C  status of "carry" bit
------------------------
SP value of StackPointer
PC value of Program Counter (with leading
                             Bank Number)
BRK this could be the disassembled contents of
location FFFF in bank 1 RAM
```

Of course the above values were only the initialized values of the pseudo-registers, since none of the other debugging operations had yet been used.

The R Command shows the Bank as the first digit of the 5 digit program counter value. A non-standard Bank shows as *. The bank number may be edited like other things on the register line.

The contents of the pseudo-registers may be easily changed at any time. This is easiest done by first displaying the current values with just the R command by itself, then cursoring up and over to the value(s) to be altered. The

value of the stackpointer is restricted to a safe range to prevent the program from getting lost. Any information in the disassembly field will be ignored in the edit operation. Following a RETURN from anywhere on the line, the new values will be displayed as confirmation.

G COMMAND -- FULL SPEED EXECUTION

The Go (G) COMMAND is used to begin full speed execution of code from inside the FREEDOM ASSEMBLER-128 program. It is similar to the SYS command, but has several advantages:

* One advantage is that the pseudo-registers may be preset to any values desired, and are loaded into the real registers just before executing the code segment.

* Another advantage is that a break - inducing return address is pushed onto the stack so that subroutines can be executed with a return to the FREEDOM ASSEMBLER-128 for register display, etc.

If the code being executed is in RAM, the "hardware" breakpoint will, if encountered, force a break and return to the FREEDOM ASSEMBLER-128, where the registers will be displayed. By looking at the value in the Program Counter, it can be determined whether the the break was induced by an unbalanced return from subroutine, or by the breakpoint utility (B command).

The Go COMMAND is invoked by typing:

G address (followed by a RETURN)

For example, to begin executing code at location $C000, in bank 15 (F hex), first set up the pseudo-registers as desired, then type:

G FC000 (followed by a RETURN)

W COMMAND -- SINGLE-STEP EXECUTION

Single-stepping through questionable code with the Walk (W) COMMAND is a powerful, and relatively easy to use debugging technique. This operation works on the principle of executing the instructions one at a time, displaying the register contents each time. While not all of the

instructions are actually executed, the ones that are not are emulated with other instructions, so that the intended operations are actually done.

The W COMMAND will accept 5 hex digits. (The first will be Bank 0-F, only if there are five digits. If there are less than five digits the current bank is used.) The default is bank 1 (Where the object code is located after an assembly.)

To invoke single-stepping, use the following command:

W address (followed by a RETURN)

If the W is used without an address, WALK will begin the "walk" at the location indicated in the pseudo Program Counter (PC) register.

For example, to begin "walking" code at location $FFD2, first set up the pseudo-registers as desired (e.g. put $41--ASCII "A"--into accumulator), then type:

W FFFD2 (followed by a RETURN)

```
  A. X. Y. NV-BDIZC SP .PC.
; 41 00 00 00100000 D0 FFFD2 JMP ($0326)
```

Then pushing the down cursor key will give:

```
; 41 00 00 00100000 D0 FEF79  PHA
; 41 00 00 00100000 CF FEF7A  LDA  $9A
; 03 00 00 00100000 CF FEF7C  CMP #$03
; 03 00 00 00100011 CF FEF7E  BNE  $EF84
; 03 00 00 00100011 CF FEF80  PLA
; 41 00 00 00100001 D0 FEF81  JMP  $C00C
; 41 00 00 00100001 D0 FC00C  JMP  $C72D
; 41 00 00 00100001 D0 FC72D  STA  $EF
```

Pressing the RUN/STOP key ends the walk and prints:

```
OK
READY
```

After each instruction, there are a number of options:

* As implied, the DOWN CURSOR key will cause instructions to be executed as long as it is held down.

* The RETURN key will cause one instruction to be executed each time the key is pushed.

* The J key is active when the current instruction is a JSR $. It will cause the subroutine to be "jumped" (executed at full speed with no display). (This key should not be used if you are using one of the optional instruction sets, unless you know that the subroutine only contains normal 6502 opcodes.)
  Use of the J key greatly speeds up debugging of code when the subroutines are known to be working properly.

* The RIGHT CURSOR key is like the "down cursor" key except that any subroutine call will be QUICK-TRACED.

* The S key will skip execution of the instruction being displayed, and go on to the next instruction.

* The STOP key will stop the "walk" without executing the current instruction. This allows the registers to be changed, and the "walk" resumed at the same location, if desired. It is also useful when it is desired to use the Resident Monitor (M Command) to display memory.

NOTE: That at any point during the WALK, the instruction currently being displayed is not yet executed.

The optional instruction sets are fully supported for the W command by emulating them with normal 6502 instructions in a manner that is transparent to the user. It is necessary to first use the # command to activate the optional instructions.

Q COMMAND -- QUICKTRACE EXECUTION

The QUICKTRACE (Q) mode can be thought of as a cross between the GO mode and the WALK mode. It is far faster than the WALK mode, but far slower than full speed execution.

In actual fact QUICKTRACE mode is the same as the WALK mode, except that it continues automatically -- and without any register display. Hence the software breakpoint is operational, and the optional instructions (if used) are all emulated.

The Q COMMAND will accept 5 hex digits. (The first digit will be Bank 0-F, only if there are five digits. If there are less than five digits the current bank is used.) The default is bank 1 (Where the object code is located after an assembly.)

To call the QUICKTRACE, use the following command:

Q address (followed by a RETURN)

For example, to begin a QUICKTRACE at location $FFD2, in the normal ROM bank, first set up the pseudo-registers as desired (e.g. put $42 into accumulator), then type:

Q FFFD2 (followed by a RETURN)

Since location $FFD2 is the start of the "CHROUT" serial output kernal the letter B should be displayed to the screen if the Accumulator contained $42 at entry. Following that, the Register display should appear, indicating a return from the subroutine.

REMEMBER: QUICKTRACE is much slower than full-speed execution, so give it time to work.

QUICKTRACE mode may be stopped at any time by pressing the RUN/STOP key, and the value of the registers at that time can be displayed by typing R (RETURN).

QUICKTRACE mode is interesting to watch when the code displays to the screen, because the action is slowed down, so that things which appear to be instantaneous in normal operation can be observed as many separate actions.

CAUTIONS

As stated previously, the G, W, and Q COMMANDS all actually execute code. This gives these commands the power to "lock-up" the computer if bad code is executed, or if interactions occur.

It is suggested that these instructions should not be used unless an up-to-date copy of the source file has already been made.

During execution of WALK and QUICKTRACE, 3 programs are being run at once:

* The operating system.

* The FREEDOM ASSEMBLER-128 program.
* The object program under test.

Since the object program is unconstrained, and can operate in the entire memory space, it is impossible to guarantee no interaction between the three programs.

Difficulty will probably be encountered if the object program uses the stack area from $100 to $170, or the bank 0 RAM from $BC0 to $BF0, which are used for scratchpad memory by the FREEDOM ASSEMBLER-128.
Certain zero-page registers are also used by the FREEDOM ASSEMBLER-128. Since these registers are also needed for object programs, these are saved in the stack area, so that WALK and QUICKTRACE can work correctly. These registers, and their corresponding save locations are as follows:

Register	Saved @
$1B	$BE2
$1C	$BE3
$1D	$BE4
$1E	$BE5
$1F	$BE6
$2C	$BE1
$FA	$BD1
$FB	$BD2
$FC	$BD3
$FD	$BD4
$FE	$BD5

If a program contains a STORE to one of these zero-page addresses, followed by a LOAD, operation will be as normally expected. If however, it is desired to use the resident monitor to pre-load or post-examine these registers, the above-listed save locations must be used.

Operations which alter the memory banking may cause difficulty. The thing to bear in mind is that in WALK and QUICKTRACE, the operations are all REALLY BEING EXECUTED (not merely simulated).

Despite some apparent limitations, these utilities are invaluable in debugging programs. The only better arrangement is to have an expensive logic analyzer with its own separate CPU and memory.

Also it should be noted that the W and Q COMMANDS inherently cause three programs to be operating at the same time, and there can be some interaction. To avoid this it is recommended that the area of the stack from $100 to $170 and RAM from $8C0 to $BF0 not be used for programs in order to avoid interaction with the debugging software. If necessary to debug software using these locations, it is suggested that the code be altered temporarily to use different ones.

Despite these minor cautions, the FREEDOM ASSEMBLER-128 debugging utilities are very powerful and useful.

TIPS FOR WRITING AND DEBUGGING CODE

* Start out with a small program and build up on it, checking it out at each step.

* The use of subroutines helps to organize thinking, often saves memory, and results in code that is easier to debug.

* Make temporary versions with special code to help debug a troublesome section. For example, it may be helpful to store an intermediate value into some memory location that can be read later.

* Be especially careful not to omit the # symbol if you intend an IMMEDIATE MODE instruction. The assembler cannot catch this error since the result is a legal "zero-page" instruction.

* Use descriptive remarks and labels in the Source File to make the program easier to follow.

* Use dedicated storage locations for variables. (Do not share locations unless really necessary).

* Be especially careful about balancing stack operations. Generally there must be a "push"

for every "pull". If something is "pushed" onto the stack, and then there is a branch, both branches may need a "pull".

* Use the WALK utility to check out your code. Many problems are due to some false assumption on the part of the software designer that he/she will make over and over when just looking through the code. The WALK utility will generally bring these errors to one's attention quite quickly.

USING THE ALTERNATE INSTRUCTION SETS

Although the majority of users will probably only use the standard 6502/6510/8502 instruction set, the FREEDOM ASSEMBLER-128 can correctly ASSEMBLE, DISASSEMBLE, WALK, and QUICKTRACE three other enhanced instruction sets, which are used in some other 6502 based microprocessors. The additional instructions available in these other microprocessors are very nice, and allow easier software design and more efficient programs. Of course these advanced instructions will only operate in a CPU chip designed to use them.

These alternate instruction sets should not be confused with the "undocumented opcodes" which are said to exist in normal 6502 microprocessors. The use of "undocumented opcodes" is NOT SUPPORTED by the FREEDOM ASSEMBLER-128, since there is no guarantee that they work in all units, or are even the same between various manufacturers, etc.

If you want to use the base 6502/6510/8502 instruction set, no action need be taken, as this is the default state. Any attempt to assemble the enhanced instructions will result in an ILLEGAL OPCODE error. Similarly the disassembler will only disassemble standard 6502 instructions.

If you want to use one of the alternate instruction sets, it is necessary to use the # command to specify which one. Once specified this instruction set will be used for all operations until changed by the # COMMAND or the FREEDOM ASSEMBLER-128 is KILLed by the K command.

The instructions which exist in each instruction set are shown in the Appendix under SUPPLEMENTAL INSTRUCTION SET. The general usage of these instruction sets is as follows:

INSTRUCTION SET #	KNOWN USAGE
#0	NORMAL (NMOS) 6502/6510/8502
#1	65C02 (NCR MANUFACTURE) (CMOS)
#2	65C02 (ROCKWELL MANUFACTURE) (CMOS)
#3	6500/11,/12,/13 (NCR MANUFACTURE)

Besides the 6502/6510 and 8502, there are many other microprocessors which use the base 6502 instruction set. Many of these are special purpose industrial or consumer controllers with on-board ROM and RAM, dedicated output ports, etc.

It is possible to change back and forth between the various instruction sets at any time by using the NUMBER (#) COMMAND.

STANDARD 6502/6510/8502

OPCODE REFERENCE

INCLUDED IN ALL INSTRUCTION SETS

MODE	MNEMONIC	HEX	BYTES	CYCLES
ADD MEMORY TO ACCUMULATOR WITH CARRY (A = A+M+C)				
immediate	ADC #$	69	2b	2c
zero page	ADC $	65	2b	3c
zero page,X	ADC $,X	75	2b	4c
absolute	ADC $	6D	3b	4c
absolute,X	ADC $,X	7D	3b	4c*
absolute,Y	ADC $,Y	79	3b	4c*
(indirect,X)	ADC ($,X)	61	2b	6c
(indirect),Y	ADC ($),Y	71	2b	5c*
condition codes	affected - N, Z, C, V			
"AND" MEMORY WITH ACCUMULATOR (A = A.and.M)				
immediate	AND #$	29	2b	2c
zero page	AND $	25	2b	3c
zero page,X	AND $,X	35	2b	4c
absolute	AND $	2D	3b	4c
absolute,X	AND $,X	3D	3b	4c*
absolute,Y	AND $,Y	39	3b	4c*
(indirect,X)	AND ($,X)	21	2b	6c
(indirect),Y	AND ($),Y	31	2b	5c
condition codes	affected - N, Z			
SHIFT LEFT ONE BIT -- ZERO FILL				
accumulator	ASL A	0A	1b	2c
zero page	ASL $	06	2b	5c
zero page,X	ASL $,X	16	2b	6c
absolute	ASL $	0E	3b	6c
absolute,X	ASL $,X	1E	3b	7c
condition codes	affected - N, Z, C			
BRANCH IF CARRY CLEAR (Carry bit=0)				
relative	BCC $	90	2b	2c*
condition codes	affected - none			
BRANCH IF CARRY SET (Carry bit=1)				
relative	BCS $	B0	2b	2c*
condition codes	affected - none			

MODE	MNEMONIC	HEX	BYTES	CYCLES

BRANCH IF RESULT IS ZERO (Zero bit=1)

relative	BEQ $	F0	2b	2c*

condition codes affected - none

TEST BITS IN MEMORY WITH ACCUMULATOR (A.and.M)

zero page	BIT $	24	2b	3c
absolute	BIT $	2C	3b	4c

Bits 6 and 7 are transferred to the Program Status Word
The accumulator and memory are both unchanged
condition codes affected - N=m7, Z, V=m6

BRANCH IF RESULT IS MINUS (Negative bit=1)

relative	BMI $	30	2b	2c*

condition codes affected - none

BRANCH IF RESULT IS NOT ZERO (Zero bit=0)

relative	BNE $	D0	2b	2c*

condition codes affected - none

BRANCH IF RESULT IS PLUS (Negative bit=0)

relative	BPL $	10	2b	2c*

condition codes affected - none

BREAK

implied	BRK	00	1b	7c

Forced interrupt -- PC+2 pushed onto stack.
Program Status Word pushed onto stack.
condition codes affected - I=1

BRANCH IF OVERFLOW CLEAR (Overflow bit(V)=0)

relative	BVC $	50	2b	2c*

condition codes affected - none

BRANCH IF OVERFLOW SET (Overflow bit(V)=1)

relative	BVS $	70	2b	2c*

condition codes affected - none

CLEAR CARRY

implied	CLC	18	1b	2c

condition codes affected - C=0

MODE	MNEMONIC	HEX	BYTES	CYCLES

CLEAR DECIMAL MODE

implied	CLD	D8	1b	2c

condition codes affected - D=0

CLEAR INTERRUPT DISABLE BIT

implied	CLI	58	1b	2c

condition codes affected - I=0

CLEAR OVERFLOW FLAG

implied	CLV	B8	1b	2c

condition codes affected - V=0

COMPARE MEMORY WITH ACCUMULATOR (A-M)

immediate	CMP #$	C9	2b	2c
zero page	CMP $	C5	2b	3c
zero page,X	CMP $,X	D5	2b	4c
absolute	CMP $	CD	3b	4c
absolute,X	CMP $,X	DD	3b	4c*
absolute,Y	CMP $,Y	D9	3b	4c*
(indirect,X)	CMP ($,X)	C1	2b	6c
(indirect),Y	CMP ($),Y	D1	2b	5c*

The accumulator and the memory are both unchanged.
condition codes affected - N, Z, C

COMPARE MEMORY WITH X REGISTER (X-M)

immediate	CPX #$	E0	2b	2c
zero page	CPX $	E4	2b	3c
absolute	CPX $	EC	3b	4c

condition codes affected - N, Z, C

COMPARE MEMORY WITH Y REGISTER (Y-M)

immediate	CPY #$	C0	2b	2c
zero page	CPY $	C4	2b	3c
absolute	CPY $	CC	3b	4c

condition codes affected - N, Z, C

DECREMENT MEMORY BY 1 (M = M-1)

zero page	DEC $	C6	2b	5c
zero page,X	DEC $,X	D6	2b	6c
absolute	DEC $	CE	3b	6c
absolute,X	DEC $,X	DE	3b	7c

condition codes affected - N, Z

MODE	MNEMONIC	HEX	BYTES	CYCLES
DECREMENT X REGISTER BY 1 (X=X-1)				
implied	DEX	CA	1b	2c
condition codes affected - N, Z				
DECREMENT Y REGISTER BY 1 (Y=Y-1)				
implied	DEY	88	1b	2c
condition codes affected - N, Z				
"EXCLUSIVE-OR" MEMORY WITH ACCUMULATOR (A = A.xor.M)				
immediate	EOR #$	49	2b	2c
zero page	EOR $	45	2b	3c
zero page,X	EOR $,X	55	2b	4c
absolute	EOR $	4D	3b	4c
absolute,X	EOR $,X	5D	3b	4c*
absolute,Y	EOR $,Y	59	3b	4c*
(indirect,X)	EOR ($,X)	41	2b	6c
(indirect),Y	EOR ($),Y	51	2b	5c* .
condition codes affected - N, Z				
INCREMENT MEMORY BY 1 (M = M+1)				
zero page	INC $	E6	2b	5c
zero page,X	INC $,X	F6	2b	6c
absolute	INC $	EE	3b	6c
absolute,X	INC $,X	FE	3b	7c
condition codes affected - N, Z				
INCREMENT X REGISTER BY 1 (X = X+1)				
implied	INX	E8	1b	2c
condition codes affected - N, Z				
INCREMENT Y REGISTER BY 1 (Y = Y+1)				
implied	INY	C8	1b	2c
condition codes affected - N, Z				
JUMP TO NEW LOCATION				
absolute	JMP $	4C	3b	3c
(PC+1) goes into PCL.				
(PC+2) goes into PCH.				
indirect	JMP ($)	6C	3b	5c
((PC+2),(PC+1)) goes into PCL				
((PC+2),(PC+1)+1) goes into PCH				
condition codes affected - none				

MODE	MNEMONIC	HEX	BYTES	CYCLES

JUMP TO SUBROUTINE (CALL)

absolute	JSR $	20	3b	6c

PC+2 is pushed onto stack.
(PC+1) goes into PCL.
(PC+2) goes into PCH.
SP = SP-2
condition codes affected - none

LOAD ACCUMULATOR WITH MEMORY (A = M)

immediate	LDA #$	A9	2b	2c
zero page	LDA $	A5	2b	3c
zero page,X	LDA $,X	B5	2b	4c
absolute	LDA $	AD	3b	4c
absolute,X	LDA $,X	BD	3b	4c*
absolute,Y	LDA $,Y	B9	3b	4c*
(indirect,X)	LDA ($,X)	A1	2b	6c
(indirect),Y	LDA ($),Y	B1	2b	5c*

condition codes affected - N, Z

LOAD X REGISTER WITH MEMORY (X = M)

immediate	LDX #$	A2	2b	2c
zero page	LDX $	A6	2b	3c
zero page,Y	LDX $,Y	B6	2b	4c
absolute	LDX $	AE	3b	4c
absolute,Y	LDX $,Y	BE	3b	4c*

condition codes affected - N, Z

LOAD Y REGISTER WITH MEMORY (Y = M)

immediate	LDY #$	A0	2b	2c
zero page	LDY $	A4	2b	3c
zero page,X	LDY $,X	B4	2b	4c
absolute	LDY $	AC	3b	4c
absolute,X	LDY $,X	BC	3b	4c*

condition codes affected - N, Z

SHIFT RIGHT ONE BIT -- ZERO FILL

accumulator	LSR A	4A	1b	2c
zero page	LSR $	46	2b	5c
zero page,X	LSR $,X	56	2b	6c
absolute	LSR $	4E	3b	6c
absolute,X	LSR $,X	5E	3b	7c

condition codes affected - N=0, Z, C

NO OPERATION

implied	NOP	EA	1b	2c

condition codes affected - none

MODE	MNEMONIC	HEX	BYTES	CYCLES
"OR" MEMORY WITH ACCUMULATOR (A=A.or.M)				
immediate	ORA #$	09	2b	2c
zero page	ORA $	05	2b	3c
zero page,X	ORA $,X	15	2b	4c
absolute	ORA $	0D	3b	4c
absolute,X	ORA $,X	1D	3b	4c*
absolute,Y	ORA $,Y	19	3b	4c*
(indirect,X)	ORA ($,X)	01	2b	6c
(indirect),Y	ORA ($),Y	11	2b	5c
condition codes affected - N, Z				
PUSH ACCUMULATOR ONTO STACK				
implied	PHA	48	1b	3c
SP decremented				
condition codes affected - none				
PUSH PROGRAM STATUS WORD ONTO STACK				
implied	PHP	08	1b	3c
SP decremented				
condition codes affected - none				
PULL ACCUMULATOR FROM STACK				
implied	PLA	68	1b	4c
SP incremented				
condition codes affected - N, Z				
PULL PROGRAM STATUS WORD FROM STACK				
implied	PLP	28	1b	4c
SP incremented				
condition codes affected - all from stack				
ROTATE LEFT ONE BIT (WITH CARRY)				
accumulator	ROL A	2A	1b	2c
zero page	ROL $	26	2b	5c
zero page,X	ROL $,X	36	2b	6c
absolute	ROL $	2E	3b	6c
absolute,X	ROL $,X	3E	3b	7c
condition codes affected - N, Z, C				

MODE	MNEMONIC	HEX	BYTES	CYCLES

ROTATE RIGHT ONE BIT (WITH CARRY)

accumulator	ROR A	6A	1b	2c
zero page	ROR $	66	2b	5c
zero page,X	ROR $,X	76	2b	6c
absolute	ROR $	6E	3b	6c
absolute,X	ROR $,X	7E	3b	7c

condition codes affected - N, Z, C

RETURN FROM INTERRUPT

implied	RTI	40	1b	6c

Program Status Word popped from stack.
PC popped from stack.
SP = SP+3
condition codes affected-all from stack

RETURN FROM SUBROUTINE

implied	RTS	60	1b	6c

PC popped from stack.
PC+1 goes into PC.
SP = SP+2
condition codes affected - none.

SUBTRACT MEMORY FROM ACCUMULATOR WITH BORROW
(A = A-M-B)

immediate	SBC #$	E9	2b	2c
zero page	SBC $	E5	2b	3c
zero page,X	SBC $,X	F5	2b	4c
absolute	SBC $	ED	3b	4c
absolute,X	SBC $,X	FD	3b	4c*
absolute,Y	SBC $,Y	F9	3b	4c*
(indirect,X)	SBC ($,X)	E1	2b	6c
(indirect),Y	SBC ($),Y	F1	2b	5c*

Borrow is inverse of carry.
condition codes affected - N, Z, C, V

SET CARRY

implied	SEC	38	1b	2c

condition codes affected - C=1

SET DECIMAL MODE

implied	SED	F8	1b	2c

condition codes affected - D=1

MODE	MNEMONIC	HEX	BYTES	CYCLES

SET INTERRUPT DISABLE STATUS

implied	SEI	78	1b	2c

condition codes affected - I=1

STORE ACCUMULATOR INTO MEMORY (M = A)

zero page	STA $	85	2b	3c
zero page,X	STA $,X	95	2b	4c
absolute	STA $	8D	3b	4c
absolute,X	STA $,X	9D	3b	5c
absolute,Y	STA $,Y	99	3b	5c
(indirect,X)	STA ($,X)	81	2b	6c
(indirect),Y	STA ($),Y	91	2b	6c

condition codes affected - none

STORE X REGISTER INTO MEMORY (M = X)

zero page	STX $	86	2b	3c
zero page,Y	STX $,Y	96	2b	4c
absolute	STX $	8E	3b	4c

condition codes affected - none

STORE Y REGISTER INTO MEMORY (M = Y)

zero page	STY $	84	2b	3c
zero page,X	STY $,X	94	2b	4c
absolute	STY $	8C	3b	4c

condition codes affected - none

TRANSFER ACCUMULATOR TO X REGISTER (X = A)

implied	TAX	AA	1b	2c

condition codes affected - N, Z

TRANSFER ACCUMULATOR TO Y REGISTER (Y = A)

implied	TAY	A8	1b	2c

condition codes affected - N, Z

TRANSFER STACKPOINTER TO X REGISTER (X = SP)

implied	TSX	BA	1b	2c

condition codes affected - N, Z

TRANSFER X REGISTER TO ACCUMULATOR (A = X)

implied	TXA	8A	1b	2c

condition codes affected - N, Z

MODE	MNEMONIC	HEX	BYTES	CYCLES

TRANSFER X REGISTER TO STACKPOINTER (SP = X)

implied	TXS	9A	1b	2c

condition codes affected - none

TRANSFER Y REGISTER TO ACCUMULATOR (A = Y)

implied	TYA	98	1b	2c

condition codes affected - N, Z

SPECIAL NOTES

b means bytes

c means cycles

PC is Program Counter

() means content of location.

* means add 1 more cycle if crossing page boundary.

For branches, also add 1 more cycle if branch is taken.

PROGRAM STATUS WORD BITS

C bit 0 - carry flag

Z bit 1 - zero flag

I bit 2 - interrupt disable

D bit 3 - decimal mode flag

B bit 4 - break flag

bit 5 - always 1

V bit 6 - overflow flag

N bit 7 - negative flag

SUPPLEMENTAL 65C02 OPCODE REFERENCE

INSTRUCTION SETS 1 & 2 ONLY

MODE	MNEMONIC	HEX	BYTES	CYCLES

ADD MEMORY TO ACCUMULATOR WITH CARRY (A = A+M+C)

0-page indirect	ADC ($)	72	2b	5c

Add 1 cycle if in decimal mode.
condition codes affected - N, Z, C, V

"AND" MEMORY WITH ACCUMULATOR (A = A.and.M)

0-page indirect	AND ($)	32	2b	5c

condition codes affected - N, Z,

TEST BITS IN MEMORY WITH ACCUMULATOR (A.and.M)

immediate	BIT #$	89	2b	2c
zero page,X	BIT $,X	34	2b	4c
absolute,X	BIT $,X	3C	3b	4c

Bits 6 and 7 are transferred to the Program Status Word (except immediate mode)
The accumulator and memory are both unchanged.
condition codes affected - N=m7, Z, V=m6 (in immediate mode, N & V are either unaffected or indeterminate, depending on manufacturer.

BRANCH ALWAYS

relative	BRA $	80	2b	3c

Add 1 cycle if page boundary crossed.
condition codes affected - none

COMPARE MEMORY WITH ACCUMULATOR (A-M)

0-page indirect	CMP ($)	D2	2b	5c

The accumulator and memory are both unchanged.
condition codes affected - N, Z, C

MODE	MNEMONIC	HEX	BYTES	CYCLES

DECREMENT ACCUMULATOR BY 1 (A = A-1)

implied	DEA	3A	1b	2c

condition codes affected - N, Z

"EXCLUSIVE-OR" MEMORY WITH ACCUMULATOR (A = A.xor.M)

0-page indirect	EOR ($)	52	2b	5c

condition codes affected - N, Z

INCREMENT ACCUMULATOR BY 1 (A = A+1)

implied	INA	1A	1b	2c

condition codes affected - N, Z

JUMP TO NEW LOCATION

X-indirect	JMP ($,X)	7C	3b	6c

($+X) goes into PCL.
($+X+1) goes into PCH.
condition codes affected - none

LOAD ACCUMULATOR WITH MEMORY (A = M)

0-page indirect	LDA ($)	B2	2b	5c

condition codes affected - N, Z

"OR" MEMORY WITH ACCUMULATOR (A = A.or.M)

0-page indirect	ORA ($)	12	2b	5c

condition codes affected - N, Z

PUSH X REGISTER ONTO STACK

implied	PHX	DA	1b	3c

SP = SP-1
condition codes affected - none

PUSH Y REGISTER ONTO STACK

implied	PHY	5A	1b	3c

SP = SP-1
condition codes affected - none

MODE	MNEMONIC	HEX	BYTES	CYCLES

PULL X REGISTER FROM STACK

implied	PLX	FA	1b	4c

SP = SP+1
condition codes affected - N, Z

PULL Y REGISTER FROM STACK

implied	PLY	7A	1b	4c

SP = SP+1
condition codes affected - N, Z

SUBTRACT MEMORY FROM ACCUMULATOR WITH BORROW (A = A-M-B)

0-page indirect	SBC ($)	F2	2b	5c

Add 1 cycle if in decimal mode.
Borrow is inverse of carry.
condition codes affected - N, Z, C, V

STORE ACCUMULATOR INTO MEMORY (M = A)

0-page indirect	STA ($)	92	2b	5c

condition codes affected - none

STORE ZERO INTO MEMORY (M = 0)

zero page	STZ $	64	2b	3c
zero page,X	STZ $,X	74	2b	4c
absolute	STZ $	9C	3b	4c
absolute,X	STZ $,X	9E	3b	5c

condition codes affected - none

TEST & RESET MEMORY BITS WITH ACCUMULATOR (M = M.and.(inv.of A))

zero page	TRB $	14	2b	5c
absolute	TRB $	1C	3b	6c

condition codes affected - Z

TEST & SET MEMORY BITS WITH ACCUMULATOR (M = M.or.A)

zero page	TSB $	04	2b	5c
absolute	TRB $	0C	3b	6c

condition codes affected - Z

SUPPLEMENTAL OPCODE REFERENCE

INSTRUCTION SETS 2 & 3 ONLY

MODE	MNEMONIC	HEX	BYTES	CYCLES
BRANCH MEMORY BIT RESET				
zero page-bit0	BBR0 $,$	0F	3b	5c*
zero page-bit1	BBR1 $,$	1F	3b	5c*
zero page-bit2	BBR2 $,$	2F	3b	5c*
zero page-bit3	BBR3 $,$	3F	3b	5c*
zero page-bit4	BBR4 $,$	4F	3b	5c*
zero page-bit5	BBR5 $,$	5F	3b	5c*
zero page-bit6	BBR6 $,$	6F	3b	5c*
zero page-bit7	BBR7 $,$	7F	3b	5c*
condition codes affected - none				
BRANCH MEMORY BIT SET				
zero page-bit0	BBS0 $,$	8F	3b	5c*
zero page-bit1	BBS1 $,$	9F	3b	5c*
zero page-bit2	BBS2 $,$	AF	3b	5c*
zero page-bit3	BBS3 $,$	BF	3b	5c*
zero page-bit4	BBS4 $,$	CF	3b	5c*
zero page-bit5	BBS5 $,$	DF	3b	5c*
zero page-bit6	BBS6 $,$	EF	3b	5c*
zero page-bit7	BBS7 $,$	FF	3b	5c*
condition codes affected - none				
SET MEMORY BIT				
zero page-bit0	SMB0 $	87	2b	5c
zero page-bit1	SMB1 $	97	2b	5c
zero page-bit2	SMB2 $	A7	2b	5c
zero page-bit3	SMB3 $	B7	2b	5c
zero page-bit4	SMB4 $	C7	2b	5c
zero page-bit5	SMB5 $	D7	2b	5c
zero page-bit6	SMB6 $	E7	2b	5c
zero page-bit7	SMB7 $	F7	2b	5c
condition codes affected - none				
RESET MEMORY BIT				
zero page-bit0	RMB0 $	07	2b	5c
zero page-bit1	RMB1 $	17	2b	5c

zero page-bit2	RMB2 $	27	2b	5c
zero page-bit3	RMB3 $	37	2b	5c
zero page-bit4	RMB4 $	47	2b	5c
zero page-bit5	RMB5 $	57	2b	5c
zero page-bit6	RMB6 $	67	2b	5c
zero page-bit7	RMB7 $	77	2b	5c

condition codes affected - none

NOTE

The FREEDOM ASSEMBLER also accepts a symbolic designation of the bit # for the above instructions. For example:

```
BBS$"AFLAG" $"ZPFLGS",$"AWASST"
```

EXAMPLE PROGRAM

The small program included in this manual is mainly provided as an example of syntax for creating source files for the FREEDOM ASSEMBLER-128. It also illustrates the use of the LOAD pseudo-op to assemble from disk files.

To create the machine language OBJECT file, first create the SOURCE file as shown and the LOAD file (shown only in the printout listing) and SAVE them.

To assemble the program, load the SOURCE file and, with the LOAD file on disk in the disk drive, enter the FREEDOM ASSEMBLER-128 using the SYS57000 command, and type A (return). If you have made no typographical errors, the assembly will complete in about four seconds. Note the first address where the OBJECT file assembled.

After a successful assembly is completed, the OBJECT file can be saved for later use with the resident monitor. Type M (return) to enter the monitor. Then use the "T" (transfer) command to locate the binary code properly in memory (use of bank 0 or 1 here OK). Then use the "S" (save) command to save it in binary format:

M (return)

T 16E10 16E5A 10E10 (return)

S"CHANGE-BKGRD-BIN",8,10E10,10E5A (RETURN)

You do not need to save the program first if you wish to run it right away. You must however switch the program to Bank 0. Do this by changing the T command above to read:

T 16E10 16E5A 00E10 (return)

Then kill the MONITOR by typing X RETURN and the FREEDOM ASSEMBLER-128 by typing K RETURN. Then type: SYS 3600 to run the background color change program.

To use this program from the disk, load it into bank 0; e.g. BLOAD "CHANGE-BKGRD-BIN" (return), then type SYS3600. Each time you execute this program by typing SYS3600, the background color will change.

The code for handling the 40 column mode is quite simple, however the code for 80 column mode may not be obvious. The code in the

LOAD file is the method recommended by Commodore for direct high speed reading and writing to registers of the 8563 video display chip used for the 80 column display.

A SOURCE file and a PRINTOUT LISTING are provided for this example program on the following pages.

```
100 REM       [$A0] "CHANGE BACKGRD"
110 REM=======================================
120 REM$PROGRAM TO CHANGE BACKGROUND COLOR
130 REM---------------------------------------
140 REM$EQUATE DEFINITIONS
150 =$!26,"VDCR26" REM$8563 FOR/BKGRD COLOR REG
160 REM---------------------------------------
170 REM$CBM OPERATING SYSTEM LOCATIONS
180 =$D7,"MODE" REM=$80 IF 80 COLUMN
190 =$F1,"COLOR" REM$CURRENT CHAR COLOR
200 =$D021,"BKGRND" REM$BACKGROUND COLOR (40C)
210 REM---------------------------------------
220 REM$SCRATCHPAD DEFINITIONS
230 USR $0E09
240 DEF "LOWNIB"
250 REM---------------------------------------
260 USR $0E10 REM$ORIGIN @ 3600 DECIMAL
270 TO $6E10
280 REM$$$$$$$$$$$$$$
290 REM$START OF CODE
300 REM$$$$$$$$$$$$$$
310 REM$BRANCH IF 80 COLUMN MODE
320 LDA $"MODE"
330 BMI $"INC80B"
340 REM $ $ $ $ $ $
350 REM$ELSE GET 40C BACKGROUND COLOR
360 LDA $"BKGRND"
370 REM $ $ $ $ $ $
380 REM$ADD 1 OR MORE
390 JSR $"ADD&CP"
400 REM $ $ $ $ $ $
410 REM$COLOR OK TO WRITE
420 STA $"BKGRND"
430 REM $ $ $ $ $ $
440 RTS REM$GO BACK TO COMMAND LEVEL
450 ^
460 @ "INC80B"
470 REM$GET 80C BACKGROUND COLOR
480 LDX #$"VDCR26"
490 JSR $"VDCIN"
500 REM $ $ $ $ $ $
510 PHA REM$SAVE FOR UPPER NIBBLE
520 REM $ $ $ $ $ $
530 REM$ADD 1 OR MORE
540 JSR $"ADD&CP"
550 REM $ $ $ $ $ $
560 REM$SAVE TEMPORARILY
570 STA $"LOWNIB"
580 REM $ $ $ $ $ $
590 PLA REM$UNSAVE ORIGINAL
600 AND #$F0 REM$MASK OFF LOW NIBBLE
610 REM $ $ $ $ $ $
620 REM$COMBINE WITH LOW NIBBLE
630 ORA $"LOWNIB"
640 REM $ $ $ $ $ $
650 REM$SEND BACK TO VDC CHIP
660 REM$X=VDCR26 STILL
670 JSR $"VDCOUT"
680 REM $ $ $ $ $ $
690 RTS REM$RETURN TO COMMAND LEVEL
700 REM$$$$$$$$$$$$$$$$$$
710 REM$$$ SUB ADD&CP $$$
720 @ "ADD&CP"
730 REM$ADD 1
740 CLC
750 ADC #$01
760 REM $ $ $ $ $ $
770 REM$IF BACKGROUND SAME AS CHAR COLOR, DO AGAIN
780 AND #$%00001111
790 CMP $"COLOR"
800 BEQ $"ADD&CP"
810 REM $ $ $ $ $ $
820 REM$ELSE OK TO KEEP
830 RTS
840 REM$$$$$$$$$$$$$$$$$$
850 ^
860 LOAD"80C CHIP SUBS",8
870 REM$$$$$$$$$$$$$$$$$$
```

```
00100          "CHANGE BACKGRD"
00110  ==================================
00120  *PROGRAM TO CHANGE BACKGROUND COLOR
00130  ----------------------------------
00140  *EQUATE DEFINITIONS
00150 =$!26,"VDCR26"  *8563 FOR/BKGRD COLOR REG     $6000   $6000
00160  ----------+-----------------------
00170  *CBM OPERATING SYSTEM LOCATIONS
00180 =$D7,"MODE"  =$80 IF 80 COLUMN                $6000   $6000
00190 =$F1,"COLOR"  *CURRENT CHAR COLOR             $6000   $6000
00200 =$D021,"BKGRND"  *BACKGROUND COLOR (40C)      $6000   $6000
00210  ----------------------------------
00220  *SCRATCHPAD DEFINITIONS
00230 USR $0E09                                     $6000   $6000
00240 DEF "LOWNIB"                                  $0E09   $0E09
00250  ----------------------------+-----
00260 USR $0E10  *ORIGIN @ 3600 DECIMAL             $0E0A   $0E0A
00270 TO $6E10                                      $0E10   $0E10
00280  ************
00290  *START OF CODE
00300  ************
00310  *BRANCH IF 80 COLUMN MODE
00320 LDA $"MODE"                                   $6E10   $0E10   A5  D7
00330 BMI $"INC80B"                                 $6E12   $0E12   30  0A
00340    * * * * * *
00350  *ELSE GET 40C BACKGROUND COLOR
00360 LDA $"BKGRND"                                 $6E14   $0E14   AD  21  D0
00370    * * * * * *
00380  *ADD 1 OR MORE
00390 JSR $"ADD&CP"                                 $6E17   $0E17   20  34  0E
00400    * * * * * *
00410  *COLOR OK TO WRITE
00420 STA $"BKGRND"                                 $6E1A   $0E1A   8D  21  D0
00430    * * * * * *
00440 RTS  *GO BACK TO COMMAND LEVEL                $6E1D   $0E1D   60
00450 FF                                            $6E1E   $0E1E
```

```
00460  @ "INC80B"                                  $6E1E  $0E1E
00470   *GET 80C BACKGROUND COLOR
00480  LDX #$"VDCR26"                              $6E1E  $0E1E  A2  1A
00490  JSR $"VDCIN"                                $6E20  $0E20  20  4E  0E
00500    * * * * * *
00510  PHA  *SAVE FOR UPPER NIBBLE                 $6E23  $0E23  48
00520    * * * * * *
00530   *ADD 1 OR MORE
00540  JSR $"ADD&CP"                               $6E24  $0E24  20  34  0E
00550    * * * * * *
00560   *SAVE TEMPORARILY
00570  STA $"LOWNIB"                               $6E27  $0E27  8D  09  0E
00580    * * * * * *
00590  PLA  *UNSAVE ORIGINAL                       $6E2A  $0E2A  68
00600  AND #$F0  *MASK OFF LOW NIBBLE              $6E2B  $0E2B  29  F0
00610    * * * * * *
00620   *COMBINE WITH LOW NIBBLE
00630  ORA $"LOWNIB"                               $6E2D  $0E2D  0D  09  0E
00640    * * * * * *
00650   *SEND BACK TO VDC CHIP
00660   *X=VDCR26 STILL
00670  JSR $"VDCOUT"                               $6E30  $0E30  20  40  0E
00680    * * * * * *
00690  RTS  *RETURN TO COMMAND LEVEL               $6E33  $0E33  60
00700   ******************
00710   *** SUB ADD&CP ***
00720  @ "ADD&CP"                                  $6E34  $0E34
00730   *ADD 1
00740  CLC                                         $6E34  $0E34  18
00750  ADC #$01                                    $6E35  $0E35  69  01
00760    * * * * * *
00770   *IF BACKGROUND SAME AS CHAR COLOR, DO AGAIN
00780  AND #$%00001111                             $6E37  $0E37  29  0F
00790  CMP $"COLOR"                                $6E39  $0E39  C5  F1
00800  BEQ $"ADD&CP"                               $6E3B  $0E3B  F0  F7
00810    * * * * * *
00820   *ELSE OK TO KEEP
00830  RTS                                         $6E3D  $0E3D  60
00840   ******************
00850  FF                                          $6E3E  $0E3E
```

```
00860  LOAD"80C CHIP SUBS",8                                  $6E3E  $0E3E

01000   *VIDEO DISPLAY CHIP SUBROUTINES
01010   ********************************
  020   *EQUATE DEFINITIONS
01030  =$D600,"VDCADR"   *8563 ADDRESS REG.                   $6E3E  $0E3E
01040  =$D601,"VDCDAT"   *8563 DATA REGISTER                  $6E3E  $0E3E
01050  =$!31,"VDCR31"   *8563 READ/WRITE REG                 $6E3E  $0E3E
01060   ******************
01070   *** SUB VDCPUT ***
01080  @ "VDCPUT"                                            $6E3E  $0E3E
01090   *WRITE A TO CURRENT 80C LOC.
01100  LDX #$"VDCR31"                                        $6E3E  $0E3E  A2  1F
01110    * * * * * * * * *
01120   *** SUB VDCOUT ***
01130  @ "VDCOUT"                                            $6E40  $0E40
01140   *WRITE A TO REGISTER IN X
01150  STX $"VDCADR"                                         $6E40  $0E40  8E  00  D6
01160  @ "VDOWT"                                             $6E43  $0E43
01170  BIT $"VDCADR"                                         $6E43  $0E43  2C  00  D6
01180   *BRANCH UNTIL VDC READY
01190  BPL $"VDOWT"                                          $6E46  $0E46  10  FB
01200   *READY FOR STORE
01210  STA $"VDCDAT"                                         $6E48  $0E48  8D  01  D6
01220  RTS                                                   $6E4B  $0E4B  60
01230   ******************
01240   *** SUB VDGET  ***
01250  @ "VDCGET"                                            $6E4C  $0E4C
01260   *READ FROM CURRENT 80C LOC.
01270  LDX #$"VDCR31"                                        $6E4C  $0E4C  A2  1F
01280    * * * * * * * * *
01290   *** SUB VDCIN  ***
01300  @ "VDCIN"                                             $6E4E  $0E4E
01310   *READ FROM REGISTER IN X
01320  STX $"VDCADR"                                         $6E4E  $0E4E  8E  00  D6
01330  @ "VDIWT"                                             $6E51  $0E51
01340  BIT $"VDCADR"                                         $6E51  $0E51  2C  00  D6
01350   *BRANCH UNTIL VDC READY
01360  BPL $"VDIWT"                                          $6E54  $0E54  10  FB
01370   *READY FOR READ
01380  LDA $"VDCDAT"                                         $6E56  $0E56  AD  01  D6
01390  RTS                                                   $6E59  $0E59  60
01400   ******************
00870   ******************
ADD&CP  $0E34  J 00390  J 00540  @ 00720  B 00800
BKGRND  $D021  = 00200  L 00360  S 00420
COLOR   $00F1  = 00190  C 00790
INC80B  $0E1E  B 00330  @ 00460
LOWNIB  $0E09  DEF 00240  S 00570  O 00630
MODE    $00D7  = 00180  L 00320
VDCADR  $D600  = 01030  S 01150  B 01170  S 01320  B 01340
VDCDAT  $D601  = 01040  S 01210  L 01380
VDCGET  $0E4C  @ 01250
VDCIN   $0E4E  J 00490  @ 01300
VDCOUT  $0E40  J 00670  @ 01130
VDCPUT  $0E3E  @ 01080
VDCR26  $001A  = 00150  L 00480
VDCR31  $001F  = 01050  L 01100  L 01270
VDIWT   $0E51  @ 01330  B 01360
VDOWT   $0E43  @ 01160  B 01190
```