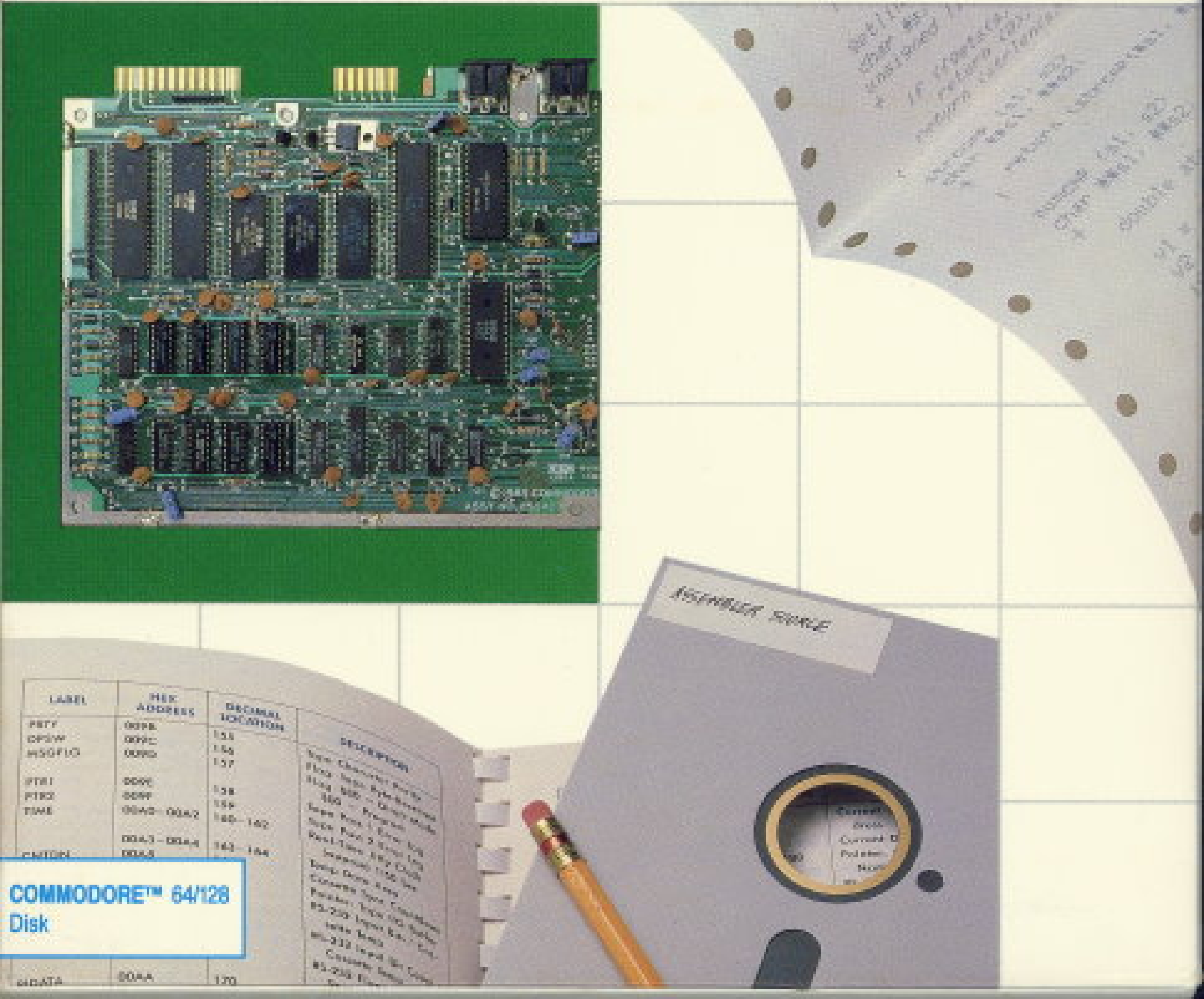
# BetterWorking™
From Spinnaker®

# POWER ASSEMBLER

The first professional quality assembler for both the Commodore 64 or Commodore 128.



COMMODORE™ 64/128
Disk

BetterWorking™

From Spinnaker®

# POWER ASSEMBLER

The first professional quality assembler for both the Commodore 64 or Commodore 128.

POWER ASSEMBLER is a very fast and versatile assembler designed specifically for the Commodore. Its rich body of commands and wide range of abilities set the standard for convenience and flexibility in assembly language programming. Capable of assembling itself in under 80 seconds, POWER ASSEMBLER will assemble to disk or to memory. Input to the assembler can come from any combination of memory resident and disk resident files. POWER ASSEMBLER possesses advanced symbol table facilities which support symbols of any length and allows all components of a program to reference all other components of the program using symbols rather than direct addresses.

Plus, with POWER ASSEMBLER a program can be written partially in BASIC for convenience and partially in assembler for speed.

POWER ASSEMBLER comes with a powerful full screen editor to make program input and debugging simple and easy. Also, its full assembly listing provides a complete description of any errors rather than just spewing out error numbers.

POWER ASSEMBLER DELIVERS:

- Full screen editor.
- Support of CBM and ASCII printers.
- Assembly to memory or disk.
- Support of separate assembly.
- Compatibility with BASIC.
- Support of conditional assembly.
- Support of long symbols.
- Cross assembly for the Z80 on the C-128.
- Support of four or more Commodore 1541 (or 1571) disk drives.
- Un-copy protected disks so back-up copies are hassle free.

0 87659 00155
PWA-C6-01
1142-20

BX-PWA

BetterWorking™

From Spinnaker®

# Power Assembler

## COPYRIGHT NOTICE

**Copyright Notice**

## TABLE OF CONTENTS

## SPECIFICATIONS

Your POWER ASSEMBLER actually encompasses two stand-alone machine language development systems for the Commodore 64 and three for the Commodore 128. One uses the Basic editor (enhanced by string SEARCH & REPLACE commands) for writing its memory based source and the other its own powerful ASCII editor. For the Commodore 128, there is also a complete Z/80 (C/PM's micro processor) cross assembler which has all the powerful commands and features of the other two programs.

Here is a brief rundown of the programs you will find on your system disk.

BUD is the boot for the Basic source compatible version of the assembler.

EBUD is the boot for the ASCII editor and its version of the assembler.

ZBUD (C-128 only) is the boot for the Z/80 cross assembler.

BUDDY.64 is the body of the Basic source compatible version of the assembler which is loaded into memory when "BUD" is run.

BUDDY.ML (C-128 only) is the body of the Basic source compatible version of the assembler which is loaded into memory when "BUD" is run.

BUDDYSYMS is the symbol table for "BUDDY.64" or "BUDDY.ML" as generated by its assembly. You can use it to explore the code and to create your own commands.

CREATE-BOOT (C-128 only) can be used to generate an autoboot for "BUDDY.ML" that will not disturb source already in memory.

ED-BUDDY.64 or ED-BUDDY.ML is the body of the ASCII editor compatible version of the assembler. It is loaded into memory along with "EDITOR.64" or "EDITOR.128" when "EBUD" is run.

ED-BUDDYSYMS is the symbol table for "ED-BUDDY.64" or "EBUDDY.ML" as generated by its assembly. You can use it to explore the code and to create your own commands.

EDITOR.64 or EDITOR.128 is a multi-featured ASCII text editor, one that does not clash with the Basic operating system.

MAKE-ASCII can be used to convert Basic style source to EDITOR.64 or EDITOR.128 compatible ASCII source.

TEST.MNE is a complete source listing of all standard and non-standard 8500 mnemonics. Use these to familiarize yourself with 8500 command syntax and to test the assembler.

TEST.ZMNE (C-128 only) is a complete source listing of Z/80 instructions. Use it to test ZBUD as well as to examine Z/80 assembly language syntax.

INVOKE-Z80.BAS (C-128 only) is a short example of a Basic program calling Z/80 routines.

SWITCHER-SOURCE (C-128 only) is a POWER ASSEMBLER source program to generate the 8500 "pivot" code used by the above. These demonstrate dual processing technique in the C-128 and will help you to make full use of the ZBUD Z/80 cross assembler.

UNASM-SOURCE is the complete, documented source program for our powerful unassembler that will convert raw code to source that you can LIST, SAVE, LOAD and, best of all, re-assemble using BUD.

ZBUDDY.ML (C-128 only) is the body of the Z/80 cross assembler which is installed when "ZBUD" is run.

ASM.SH (C-128 only) is the version of POWER ASSEMBLER that was designed to run under the SHELL program from POWER C 128 for C language programmers. It is fully compatible with the Shell, Editor, Ramdisk and Linker, and can be used both as a stand-alone assembler in this environment or to write custom C language functions.

## COMPATIBILITY

Your POWER ASSEMBLER is completely compatible with the Basic 2.0 source format used in the Commodore 64 and with the Basic 7.0 source format used in the Commodore 128, and with the Commodore disk operating system used in both.
Assembly language programs can be written on the much enhanced C-128 editor. In addition to using this new editor's ability to renumber and auto-number lines, delete line ranges, pause scroll and much more, with Bud in memory users will be able to execute powerful string (label) search and replace commands.

Pure ASCII SEQ or PRG files can also be assembled from disk or memory, allowing source to be written on virtually any text editor or word processor. POWER ASSEMBLER's own EDITOR.64 and EDITOR.128 provided supports 4-WAY, bidirectional scrolling and paging as well as CUT & PASTE, SEARCH & REPLACE and much more.

POWER ASSEMBLER for the Commodore 64 is fully compatible with FASTLOAD/SAVE disk utilities such as BETTER WORKING's TURBO LOAD AND SAVE.

## SPEED

On the Commodore 64, symbols are organized as a binary tree for very fast access of even gigantic tables. 6510 Mnemomics are divided into numerous short lists which are selected via a quickly generated hash value for virtually instantaneous command lookups. BUD LINK assembles its own ten source files creating two ML programs containing over 8K of code in about two minutes with Turbo Load and Save installed.

On the Commodore 128, source files can be linked or disk assembled using 1571 burst mode access through a quasi RAM DISK maintained by the assembler for very fast chaining. A binary structured symbol table and hash code access to multiple, very short mnemonic lists insure near instantaneous memory based operations.

## INPUT

Single, large source programs can be assembled directly from memory, or many source files can be assembled as one, either by load linking or direct disk based assembly. Many combinations of memory and disk based assembly are also possible.

Multiple device handling, allows for the application of any number of disk drives.

## OUTPUT

Code can be sent directly to memory allowing for fast, diskless testing of fairly large programs.

Any number of LOADable, machine language programs, all sharing a common symbol table, can be created in a single operation.

Symbol tables can be automatically saved in part or in full and used by other source programs. New modules for very large ML projects can be designed, tested and retested in memory without having to re-assemble the whole system each time.

POWER ASSEMBLER can be instructed to direct all output through custom user routines for special handling.

## DISPLAY

Show full assembly process including source lines, object code and symbol table listings for all or any portion of an assembly.

Paginated output may also be directed to a printer.

Error checking is complete and error messages are full and descriptive. Where many errors are anticipated a Display-Errors-Only-To-Printer mode is supported.

## LANGUAGE FEATURES

If/else conditional assembly is supported

Temporarily offset program counter assembling generates patches of code that will be relocated before execution.

Setup internal buffers as well as passive external variable tables effortlessly.

Automatically merge Basic and assembler source programs. BUD allows Basic to SYS, POKE, and PEEK assembled symbol table values by name.

Work with non-standard opcode using all of their unofficially yet generally agreed on mnemonic forms.

Macro-ops to move memory (up or down any distance), fill memory and test pointers make short work of these common and often tedious procedures.

Data can be in the form of word tables, byte tables, ASCII text, even screen-code text.

Multiplication, division, addition and subtraction of any combination of hex, binary, decimal, ASCII, screen code or symbolic values is supported.

Symbols may be of any length and remain unique.

Temporary (reusable), character ( + - / ) symbols allow for easier coding of routine short branches and result in smaller symbol tables and even faster assemblies.

## GETTING STARTED

If you are like most people you will want to try something right away just to feel the program out and get on the right track. Type in the following:

| For Commodore 64 | For Commodore 128 |
|---|---|
| LOAD"BUD",8 <RETURN> | DLOAD"BUD <RETURN> |
| RUN <RETURN> | RUN <RETURN> |

This will cause the body of the program to be loaded into memory and executed. Upon completion you should see a line of copyright information along with a pair of meaningless (at this time), hex range numbers.

## MEMORY USAGE

POWER ASSEMBLER resides completely in the "hidden" RAM beneath Basic and the Kernal ROM. This maximizes the amount of memory free for source, symboltable and utilities such as SUPERMON and POWER which lower the top of Basic.

For the Commodore 64, a few bytes of memory starting at 999 are reserved for POWER ASSEMBLER's entry code.

A SYS 999 will probably, though not necessarily, be the only Basic statement executed before POWER ASSEMBLER takes over. In other words, a SYS 999 invokes the assembler. POWER ASSEMBLER will act on all source following. Basic will interpret and try to execute everything up to and including the SYS 999 line.

For the Commodore 128, the page of memory between $f00 and $1000 in BANK 15 is reserved for POWER ASSEMBLER activities. If you interfere in this area you may have to re-boot the program. If you need to press the reset button for other reasons, POWER ASSEMBLER should not be affected. The entry routine sits at 4000. A SYS 4000 will probably, though not necessarily, be the only Basic statement executed before POWER ASSEMBLER takes over. In other words, a SYS 4000 invokes the assembler. POWER ASSEMBLER will act on all source following. Basic will interpret and try to execute everything up to and including the SYS 4000 line.

## WARM-UP EXERCISE

Enter the following short program just as if you were writing in Basic. The sequence of the line numbers is important but the actual line numbers themselves are not. You don't have to bother typing the comments. The colons are used to introduce white space to the source in order to make it more readable. They could be replaced with UP ARROWs or left out altogether.

```
1 SYS 999                  ;calls POWER ASSEMBLER
2 .ORG 10000               ;put code at 10000
3 .MEM                     ;output to memory
10 PRINT =$FFD2            ;kernal routine
20 LDX #0                  ;initialize X
```

```
30 - LDA MESSAGE,X          ;get next character
40:  JSR PRINT              ;print it
50:  INX                    ;increment x
60:  CPX #MESSAGELEN        ;see if done
70:  BNE -                  ;if not loop back
90 RTS                      ;else return
100:
110 MESSAGE =*
120 .ASC "HELLO WORLD"
130 MESSAGELEN =*-MESSAGE
```

Look it over to see if you got it right, then RUN it. If this was your first ever coding in assembler you undoubtedly are facing a number of error messages. Examine lines with errors closely to see how they differ from the above source. When you can assemble with no error messages try executing the code with a SYS 10000.

If all went well the words HELLO WORLD will be printed on screen. If not, you should know that you are the first person ever whose assembly language program failed to work perfectly the first time (ha ha, only kidding). Try again.

Notice the .ASC command. Commands with periods in front of them are called pseudo-ops. They do not represent any particular ML opcode. They are instructions to POWER ASSEMBLER. Familiarity with them will allow you to take full advantage of POWER ASSEMBLER's many abilities.

Notice the use of "-" as a label. This is an example of the use of a POWER ASSEMBLER temporary label. Some name like LOOP or BACK or HOWDY could have been used place of the "-" characters; but why bother? The BNE - codes a conditional branch back to the line last labeled with a "-" character. The "+" is used as a forward referencing temporary symbol. These can be used again and again.

Notice also how the statements are laid out. Each statement consists of up to four distinct parts:

A FLAG or LABEL when used will come first. POWER ASSEMBLER will place it in the symbol table (unless temporary) along with the address of the program counter at the beginning of its line. Throughout your source you may refer to that particular line (ie. address) using the symbol name.

Next comes the OPERATOR which is the instruction portion. It will be a PSEUDO-OP or MACRO command to POWER ASSEMBLER or a mnemonic representing a specific opcode.

Many operators will require an OPERAND address or value to complete their instruction. The OPERAND portion of the assembly language statement follows the operator.

Last will be your comment. A semi-colon must precede it. These are of no use to POWER ASSEMBLER who knows exactly what is going on all the time, but can be of tremendous benefit to you who may someday forget.

| SYMBOL | OPERATOR | OPERAND | COMMENT |
|---|---|---|---|
| 10 MEANINGFUL | LDA | #0 | ;EXPLAIN |

There must be at least one space separating each of the first three parts. Extra spaces will always be ignored.

## SYMBOLS

Symbols may be of any length so you can and should use very meaningful names.
The apostrophe has no special meaning to POWER ASSEMBLER; multi word symbols should probably be broken up with these for clarity. Notice how much more readable WRITE'TO'TAPE is than WRITETOTAPE, or COLOUR'MEMORY is than COLOURMEMORY.

Permanent symbol names may not begin with any of the following characters:

```
0 1 2 3 4 5 6 7 8 9 ! # < > " @ $ % ( )  : ; , . / * + - =
```

or contain any the following arithmetic operators.

```
/ * + - =
```

These would cause POWER ASSEMBLER to mistakenly assume that the symbol was a numeric or character value, or an expression, probably resulting in a delightful and poignant error message.

Symbols may not contain blanks. Again, use the apostrophe to break them up.

Also, a symbol may not be the same as one of the standard mnemonics like LDA or DEX or BNE. POWER ASSEMBLER is great but it can't read your mind.

## EQUAL ASSIGNMENTS

In addition to flagging, symbols may be given values using an assignment statement. The equal sign is used just as in Basic assignments. One space must follow the symbol name. Extra spaces are optional. Here are some examples:

```
1 SCREEN'START            =  $400
2 SCREEN'END              =  SCREEN'MEMORY+999
3 CHROUT                  =  $FFD2
4 PROGRAM'COUNTER         =  *
```

## SET ASSIGNMENTS

You cannot use the equal sign or flagging to reassign a new value to an existing symbol. To change a symbol value you should use the LEFT ARROW in place of an equal sign. Symbols to be reassigned should be assigned values exclusively with the LEFT ARROW assignment operator so that they maintain parallel values on both passes of the assembly.

LEFT-ARROW, set re-assignments can lead to confusing programs and hard-to diagnose errors. POWER ASSEMBLER's support of temporary symbols, large symbol names and numerous program counter control pseudo-ops reduce the possibility that symbol value re-assignments will be necessary.

## ASSIGNMENTS TO PROGRAM COUNTER

Symbols may be set to the value of the program counter in two ways: (1) by assignment to the program counter "*" variable (see line 4 above) and (2) by flagging. Flagging involves simply putting the name first on any line:

```
10 ANYNAME DEX              ;decrement x register
20 BNE ANYNAME              ;loops until x = 0
```

The same program counter value can be assigned to a number of symbols via the "*" assignment. Only one flag may be used per line. Tabled flag values are compared with the program counter on pass two of the assembly in checking for a deadly out-of-phase condition.

## OPERATORS

The operator is also referred to as the instruction. In its English or source form it is called a mnemonic. Once converted into a machine language byte it is known as an opcode.

The operator is the command portion of the assembly language statement. The commands which begin with a period are called pseudo-ops. These are not converted into any specific opcodes but tell POWER ASSEMBLER to do something special.

Many, though not all operators, will require that some information follow. This may be an address or numeric value or a string of comma delimited values or even quoted text as in a filename or .ASC string. Absence of this information will lead to an OPERAND EXPECTED error message.

POWER ASSEMBLER of course recognizes all of the standard mnemonics used to represent machine language opcodes. These three letter terms are converted by the assembler directly into the appropriate one byte opcodes. You will be informed if a value was expected but didn't follow an instruction, or if an unexpected value or illegal value followed.

## OPERANDS

These are the values which are required by many operators to complete their instruction. If you start a line with an STX instruction it is assumed that you wish to STore the information in the X register somewhere. Therefore, following this must be an OPERAND value relating to where in memory this value is to be put.

An operand may be any elsewhere defined symbol; a hexadecimal, binary or decimal value; or a screen code or ASCII character. Any combination of these may be used in an expression to produce an operand value. Binary numbers must begin with a percent sign (%11110000). Hexadecimal numbers must begin with a dollar sign ($f0). Decimal numbers are otherwise assumed (240).

Values greater than $ffff (65536 decimal) or values less than 0 will lead to an error message.

```
10 LDX #END-START            ;length of whatever
20 STA 12*4096               ;store at $c000
30 LDA POINTER+1             ;high byte of pointer
40 LDA #$100-15              ;is 241 (negative 15)
50 LDA #"&"+128              ;ascii for reversed &
```

Here is how an RTS jump might be coded using expressions as operands:

```
10 LDA >PICTURESHOW-1:PHA
20 LDA <PICTURESHOW-1:PHA
30 RTS                       ;is the same as jmp pictureshow
```

Notice that two or more statements can be put on one source line if they are separated by a colon. The colon always signals a new line with two exceptions: (1) when the colon occurs between quotes as in a filename, (2) when the colon occurs in a comment ... that is after a semi-colon. A command cannot follow a comment on the same line.

The "<" and ">" (low byte - high byte) operators are always applied after the entire expression following has been evaluated. They also indicate that the value represents a numeric constant and not an address; in other words, an immediate value.

## EXPRESSIONS

Multiplication, division, addition and subtraction are supported in expressions. Fractions are truncated in division. The expression 8/3 would equal 2.

Expressions are evaluated strictly from left to right. Any combination of hexadecimal, decimal, binary, ASCII, screen code, or symbolic integer values may be involved. Nowhere in the course of evaluating an expression will a negative value be tolerated. Neither will a value greater than $FFFF be acceptable.

Expressions which generate negative values are often accidental. Calculating the length of a table by subtracting the address of its end from the address of its beginning (instead of the other way around) could result in a frustrating and hard-to-diagnose mis-performance of the code.

If one must work with negatives they can be easily, just not accidentally, expressed.

```
100 .BYTE $100-VAL          ;= -val
200 .WORD $FFFF-VAL+1       ;= -two byte val
```

Parenthesis are not supported in POWER ASSEMBLER expressions. These signal indirect addressing mode only. If ordering cannot be properly established in simple left to right layout then an expression should be divided into two or more parts.

When the "*" is used as a variable in an expression it always holds the value of the program counter. This is the address at which the code for the "*" line will originate in memory.

Here "*" is used to point to the address operand portion of a self modifying JSR instruction.

```
10 LDA <DESTINATION
20 STA TARGET
30 LDA >DESTINATION
40 STA TARGET+1
50 JSR $0000: TARGET =*-2
```

When the "<" character precedes an expression it acts on the entire value. That is, the expression is completely evaluated first, and then the low byte only of this value is returned.

The ">" returns only the high byte.

```
10 LDA <$ABCD              ;same as lda #$cd
10 LDA >$ABCD              ;same as lda #$ab
10 LDA <$1100-1            ;same as lda #$ff
10 LDA >$1100-1            ;same as lda #$10
```

Notice again how the ">" and "<" always force immediate mode. In other assemblers only the "#" can do this. Indeed you could use LDA #<OPERAND with POWER ASSEMBLER, but the "#" would be superfluous.

The same is true when using screen code and ASCII values:

```
10 LDX "A"                 ;same as ldx #"A" or ldx #65
20 LDY @"A"                ;screen code ie. ldy #1
```

The immediate mode is automatic when using ASCII codes, screen codes, and low or high address bytes. A situation where you would wish it otherwise is inconceivable. Using immediate mode where its intention is obvious should help you avoid often puzzling "#" omission errors where zero page addresses are accessed instead of one byte immediate values.

## ADDRESSING MODES

Syntax for describing addressing modes is highly standardized. POWER ASSEMBLER adheres strongly to this standard. The value 0 is used in the following to represent any one byte operand. The value 1000 is used where any two byte operand would do.

```
LDA #0                      ;immediate value
LDA 0                       ;zero page address
STX 0,y                     ;zero page y indexed
LDA 0,x                     ;zero page x indexed
LDA 1000                    ;absolute address
LDA 1000,x                  ;absolute x indexed
LDA 1000,y                  ;absolute y indexed
LDA (0,x)                   ;pre-indexed indirect x
LDA (0),y                   ;indirect post-indexed y
BNE 1000                    ;relative branching
JMP (1000)                  ;indirect jump
LSR                         ;accumulator implied
INX                         ;implied
```

Some assemblers allow or require that the accumulator mode be expressed LSR A, ASL A, ROR A or ROL A. POWER ASSEMBLER, however, would try to look the "A" up in the symbol table. So leave it off.

POWER ASSEMBLER will use zero page addressing whenever possible. You may force absolute addressing with the "!" character.

```
LDA $FF,X                   ;codes b5 ff
LDA !$FF,X                  ;codes bd ff 00
```

## ERROR MESSAGES

Most of us never make mistakes and have no use for error messages. Still, there are always a few to go and spoil it for everyone else. So for these few people we have included comprehensive error handling and checking. The rest of us perfect programmers can just skip over this section.

Seriously however, I use 'em myself ... a lot. Unless an error is fatal, POWER ASSEMBLER will place NOPs into your code where it is encountered. The number of bytes which would ordinarily have been generated by the instruction determines the number of NOPs output. If you were to include the line 1000 YIPPIE DIPPIE in a program you would get an error message but no bytes would be output. Your code would not be affected.

If you had written 1000 BNE *+500 you would get a BRANCH OUT OF RANGE error and two NOPs would be output. You might be able to do a certain amount of testing in spite of it.

Other errors, known affectionately as fatal errors, will terminate the assembly after closing all files.

This is the case with phase errors, I/O errors or symbol table overflow, If you press the RUN STOP key assembly is stopped. Open files are always closed. If you are .FAST assembling with a blank screen an error will turn it back on instantly.

The messages are fairly self-explanatory and, in most situations, should make it easy to diagnose the problem. Error messages are always listed above the offending source line which is displayed following a >>>.

Here is a rundown and brief description of each of POWER ASSEMBLER's error messages:

QUOTE EXPECTED following .ASC or.SCR or there is more than one character between quotes where only one is permitted.

UNKNOWN PSEUDO-OP means you've used the "." as the first character of a symbol or mis-spelled a pseudo-op (.BITE).

TOO MANY STRINGS if more than three space separated "words" appear in one statement outside quotes.

COMMAND EXPECTED means a mnemonic or pseudo-op was expected.

OPERAND EXPECTED means a value or parameter is needed to complete some instruction.

INVALID MODE OF OPERATION if you try to code some addressing mode not allowed with the command.

RE-DEFINITION OF A SYMBOL if you used the same flag twice or otherwise tried to reassign a value to a symbol with "=".

ONE BYTE VALUE EXPECTED if you try to use a two byte operand where unacceptable.

IMMEDIATE VALUE INDEXING is a special case of the invalid mode in which one tries to index a number instead of an address ie. lda #100,x

KEYBOARD ERROR probably indicates a typo or perhaps some illegal character making its way into a symbol.

VALUE TOO HIGH OR NEGATIVE when a negative value or a value higher than $ffff is arrived at during an expression evaluation.

NON-NUMERIC CHARACTER a symbol begins with a number 0-9, or a non-digit has made its way into a number.

UNDEFINED SYMBOL means the symbol was not found (on pass two only) in the symbol table; perhaps it's mis-spelled or a "$" has been left off a hex value.

BRANCH OUT OF RANGE if you attempt to relative branch too far ie. more than +127 or -128 from *+2.

UNEXPECTED OPERAND if some inherent command is followed by a value.

SYMBOL TABLE OVERFLOW there is no longer room in memory for your source and the generated symbol table. You'll have to .FILE assemble it from disk or split it into two or more pieces and .LINK them.

FILE NAME EXPECTED a special case of operand expected; a file name is needed to complete one of the file handling pseudo-ops.

PHASE ERROR For some reason the symbol table as created on pass one is out of sync with the code on pass two; this could be caused by a late zero page symbol assignment or leaving characters outside quotes in a .SCR or .ASC text string. POWER ASSEMBLER checks for phase errors by looking up all labels on pass two to see if their value in the symbol table matches the program counter. If not something has gone very wrong. You don't want to continue in this condition.

BUS CRASH!!! there's a problem on the IEEE bus; the disk command channel is read for your enlightenment.

If you are assembling a very large, perhaps newly converted, program for the first time and anticipate more errors than will fit on the screen then you might want to direct errors-only to your printer via the .DIS E command

## COMMENTS ON STYLE

POWER ASSEMBLER allows you to use colons to link source statements just as in Basic. Never abuse this! Your source may become unreadable.

## WHITE SPACE

Use a few blank lines to separate the various modules and ideas of your program. Indent everything but your symbols a few spaces to the right so they stand out.

There are two ways to do these things: Obviously you can't just enter a blank line; Basic editor would ignore or erase it. Put a colon, or better yet, an UP ARROW by itself on the line. The UP ARROW is ignored by POWER ASSEMBLER as the first line character. Use it to keep the BASIC editor from removing leading spaces.

## COMMENTS AND MEANINGFUL SYMBOLS

Use meaningful symbol names and comment liberally. I have always been impressed by this in the source programs of experts.

Use the temporary symbols "-", "+", and "/" to code short branches and avoid having to generate meaningless symbol names for them. This should free up your imagination for those names that do matter. It will also allow crucial, thoughtful labels to better stand out.

Use POWER ASSEMBLER's built in string handling editor, Labelgun, to keep your symbols meaningful and up-to-date without extra typing or hunting around.

## MAKE THE ASSEMBLER DO IT

A common mistake of beginners is to calculate by hand the lengths of strings and tables in their programs. Use symbols and expressions to do this; then, when you change the length, you wont have to recalculate.

```
10 TABLEBEGIN =*
20 .ASC "******PRINT MESSAGES"******"
30 .SCR "/////SCREENCODE VALUES/////"
40 .WOR 1000,2000,ADDRESS,256*12
50 .BYT 0,1,2,4,8,16,32,64,128
60                               ;or whatever else goes in tables
70 TABLEEND =*
80 TABLELENGTH = TABLEEND-TABLEBEGIN
```

In short, make the assembler do the work. Assembly language, in addition to producing very fast and compact code can be flexible, versatile and easy to modify and understand.

## PSEUDO OPS

Here are the pseudo-ops which POWER ASSEMBLER recognizes. Where a [word] operand is required any valid POWER ASSEMBLER expression involving any combination of $Hex, Decimal, %Binary, "ASCII", @"SCREENCODE" or symbolic values can be used. If a [byte] is expected the expression value cannot exceed 255.

Where quote enclosed names or text strings are expected those provided are only examples. Make up your own, okay.

Where the operand is a pointer [...PTR] a one byte value is needed. It will represent a zero page address. The square brackets are not a part of the command syntax. They do not go in your source.

### PSEUDO-OPS **quick reference table**

| | |
|---|---|
| *= [word] | ;set program counter |
| .ORG [word] | ;set program counter |
| .BUF [word] | ;create internal buffer |
| .OFF [word] | ;offset code destination |
| .OFE | ;end of offset coding |
| .MEM | ;output to memory |
| .DIS | ;display assembly (on/off) |
| .DIS P | ;display assembly to printer |
| .DIS E | ;display only errors to printer |

|  |  |
|---|---|
| .OUT [word] | ;output through user routine |
| .DVI [#] | ;define input device (default 8) |
| .DVO [#] | ;define output device (default 8) |
| .OBJ "MY-PROGRAM" | ;create object file |
| .BAS "COMBO-PRG" | ;merge basic with ML |
| .LINK "NEXT-SRC" | ;chain next source file |
| .LOOP "BACK-FIRST" | ;end of chain |
| .FILE "ANY-SOURCE" | ;assemble from disk |
| .SEQ "ASCII-FILE" | ;assembles ascii format source file |
| .LST "SYMS-TO-USE" | ;load symbol table |
| .TOP | ;top of .SST when not all symbols |
| .SST "SYMBOL-TAB" | ;save symbol table |
| .BYTE [byte,...] | ;table one byte value(s) |
| .WORD [word,...] | ;table two byte value(s) |
| .ASC "characters..." | ;table ascii value(s) |
| .SCR "characters..." | ;table screen code value(s) |
| .PSU | ;for non-standard opcode mnemonics |
| .FAS | ;turns screen off during assembly |
| .END | ;force end of source |
| .IF [word] | ;if word <> 0 then continue |
| .ELSE | ;otherwise skip to here then begin |
| .IFE | ;conditional assembly ends |
| .TEST [THISPTR,THATPTR] | ;compare indirect addresses. |
| .DUMP [BEGINPTR ENDPTR] | ;dump acc. to range of memory. |

.MOVE;[BEGINPTR,ENDPTR,DESTPTR];move memory

Any pseudo-op mnemonic can be extended or truncated. If you would rather use .WOR or .WO or even .W instead of .WORD, POWER ASSEMBLER will still accept it. Conversely, .BUFFER .DISPLAY or .MEMORY would work the same as BUF, .DIS or .MEM.

Programmers, being the lazy lot that we are, are more apt to truncate than extend POWER ASSEMBLER's pseudo-ops. Don't get too carried away with this. The command .O $C000 might ORiGinate program counter to $C000; it also might cause POWER ASSEMBLER to jsr OUT through a user routine or do some OFFset coding, or even try to open up an OBJect file.

## PSEUDO OPS, DEFINITIONS

### .ORG [address]

The .ORG pseudo-op must be followed by an address value. This tells tells POWER ASSEMBLER where the machine language output will reside in memory. If it is not used output will ORiGinate at $C000 hex.

```
10 .ORG $2000               ;code at $2000 hex
10 .ORG 50000               ;code at 50000 decimal
10 .ORG *+2                 ;bump program counter by 2
```

.ORG *+2 above will not result in actual output. If you had begun sending bytes to disk such a statement would lead to trouble. When loaded into memory the code would be out of sync with the symbol table used to create it.

Instead, use .ORG to set up flexible variable tables before output has begun and especially after output has ended.

```
10 .ORG $200                ;system input buffer variables
20 FLAG1 .ORG *+1
30 FLAG2 .ORG *+1
40 VECTOR1 .ORG *+2
50 VECTOR2 .ORG *+2         ;and so on...
```

.ORG replaces the *= assignment found in this and other assemblers. Again, do not use it to create space within your code. To create internal buffers use the .BUF [#bytes] or *= [new pc] command.

### .BUF [# of zeros to send]

The value following .BUF determines the number of zeros to be output. This can be used to create buffers for I/O or space for variables within your code. The command .BUF 6 would do the same thing as .BYTE 0,0,0,0,0,0.

Situations may arise where you do not know exactly how many zeros you want to write, only the destination address to which you wish to write them. The command .BUF DESTINATION-* would do the job as would *=DESTINATION. In either case the value of DESTINATION must be previously defined or immediately calculable.

Usually variable tables sit on top or lie at the bottom of a program or are off somewhere else completely in memory and do not contribute to to the size of the object code. If for some reason an internal variable table is needed, the .BUF or *= commands should be used. Following they are used interchangeably although you might not want to do so purely for aesthetic reasons.

```
    .                               ;code being sent to disk
    .                               ;more code
10 JMP ENDOFTABLE                   ;jump over internal table
20 STARTOFTABLE =*
50 VAR1 .BUF 1                      ;internal vars
60 VAR2 *=*+1
70 FLG1 .BUF 1
80 FLG2 *=*+1
90 PTR1 .BUF 2
100 PTR2 *=*+2
    .                               ;etc.
    .                               ;etc.
400 ENDOFTABLE =*
410                                 ;code continues
```

**Note:** any symbol used in an operand to either .BUF, .ORG or *= must have already been defined. Forward references will not work since the number of bytes generated must be calculated on the first pass.

To recap: .ORG [EXPR] may be used to set any value to the program counter "*" variable at anytime and will never result in output. It is most useful in defining load (header) addresses prior to the creating of .OBJect files and for creating un-initialized variable tables at the end of object files.

.BUF [EXPR] will always result in the output of the expressed number of zeros.

*= [EXPR] will generate zero filler bytes to the new "*" value only after you have begun sending bytes to an object file; it may not then be used to reverse the program counter.

## .OFF [address]

The .OFF command is used to write code destined to execute at a different location than where it originates. The operand portion tells POWER ASSEMBLER where the code will finally execute. This should prove invaluable in programming for more than one micro-processor at a time or in situations where you are writing code that will be moved before it is run. The .ORG command could be used to do the same thing by resetting the program counter after output had begun then back to the proper in-stream value (by using some symbolic expression) after the offset coding had finished. This is not as convenient as or as clear as using .OFF DESTINATION to create another temporary program counter.

## .OFE

The .OFE command simply ends offset coding and resumes with the original program counter at its new address. If assembly is DISplayed for OFFset coding the program counter, at first glance, may not appear to have been affected; however, symbol values, JSRs and other absolute references to it will correspond to that defined by .OFF, not as originally set with .ORG and as displayed.

The following program moves a short "POKEHELLO" routine into the C-64 cassette buffer, calls it, then continues.

```
5 SCREEN =1024                  ;(C-128 in 40 column mode)
10 *= 20000 (C-128 *= 3000)     ;or where ever
20    LDX #LENGTH'TO'MOVE-1
30 - LDA CODE'TO'MOVE,X
40    STA POKE'HELLO,X
50    DEX
60    BPL -                     ;use of temporary sym "-"
70    JSR POKE'HELLO
80    JMP CONTINUE
85:
90 CODE'TO'MOVE =*
100 .OFF 832                    ;cassette buffer
110 POKE'HELLO =*
120 LDX #0
130 - LDA MSG,X
140 STA SCREEN,X
150 INX
160 CPX #MSGLEN
170 BNE -                       ;temp sym used again
180 RTS
190 MSG .SCR "HELLO":MSGLEN =*-MSG
200 LENGTH'TO'MOVE = *-POKE'HELLO
210 .OFE                        ;back to normal
215:
220 CONTINUE =*                 ;and on we go...
```

.OFF would be useful in creating code destined to execute in the 1541 disk drive after being loaded into the C-64 as part of a larger program.

For the Commodore 128 only, moves like the above are quite useful in programming the C-128. POWER ASSEMBLER, for example, before assembling, moves "relay" code into Basic's input buffer ($200) where it can see and be seen by all other banks. This allows POWER ASSEMBLER to access Kernal ROM, registers and user defined routines and memory which would be otherwise invisible. .OFF would also be useful in creating code destined to execute in the disk drive after being loaded into the C-128 as part of a larger program.

## .MEM; **output to memory**

This command takes no operand. It simply instructs POWER ASSEMBLER to output code directly into C-64 memory. The code will be "poked" into memory at the .ORG address.

.MEM is a toggle command. The first occurrence initiates memory output, a second turns it off, a third back on again, and so on. This allows selected portions of a program to be output to memory. Fairly large programs can be worked on and debugged without going to disk.

## .BANK [0-15] (C-128 only)

This selects an output bank for in memory operations. Bank 15 is the default. In bank 15 all Basic and Kernal ROM is visible which means these routines can be called directly and that special interrupt handling or squelching will not be required. However, there is not all that much RAM. Chances are that large ML programs will not finally execute in bank 15. The .BANK command can be used in conjunction with .MEM to direct object code to memory in any bank.

## .DIS

When this is used the complete assembly process will be shown on screen. Included in this will be the following from left to right:

1. The Basic line number of the source line being assembled, or if it is an un-numbered ASCII file being assembled from disk, then the sequence number of the source line in the file.

2. The current program counter value.

3. One, two or three hex values representing the object code, if any is generated.

4. The actual source line.

.DIS is an on/off toggle command. This allows for display of selected portions only of the assembly. In the display mode, when assembly is finished, the symbol names and values, as defined in the program, will be listed. If this is not wanted then use .DIS to turn display off at or near the end of your source. If only the symbol listing is wanted then turn display mode on by using .DIS for the first time as the last source command.

## .DIS P

This will direct full display to a printer as well as to the screen. Use the .DIS P command to generate detailed source/assembly listings. Paging is controlled by POWER ASSEMBLER. Three things are assumed.

1. The paper is positioned at the top of a page. Only four blank lines are allowed for per page so don't start down too far or the perforated edges will be printed over.

2. Paper is of the standard size (ie. 66 lines per page).

3. Continuous form feed is acceptable. It is doubtful that anyone will want source listings on separate pieces of paper. Be sure enough paper is at hand. Once printing has begun the only way to stop is to abort the assembly via RUN STOP.

The symbol table will be displayed to the printer in two column format.

.DIS E

The E option sends only error messages to the printer. It is really like no display except that messages normally only sent to the screen with display off are also sent to the printer. These include (1) the names of any disk files accessed during the assembly, (2) error messages and (3) the hex object range at the end.

When disk assembling a large source file or a number or them together there is an ever-so-remote possibility that more errors will occur than can fit on the screen. Rather than franticly scribbling down filenames and line numbers as mistakes go whizzing by, use .DIS E to send everything to the printer and go have a coffee.

Again, error messages are sent to the screen even when no display mode is used. The only way to avoid seeing them is to either not make any, or to not look at your monitor.

**.OUT [operand] (C-64 only)**

If you are burning an EPROM, outputting to tape or modem, or perhaps encrypting your code you might need to use the .OUT command. Beginning on pass two, POWER ASSEMBLER JSRs to the address following the .OUT with each byte of code. This byte will be in the accumulator. Do what you like with it then RTS back to POWER ASSEMBLER and wait for the next.

RAM from 820 to 998 is not used by POWER ASSEMBLER, neither is the free memory from 679-767. Memory from $C000 to $CFFF is available and all of basic RAM is at your disposal. If you must use zero page in your routine you should probably save and replace any values.

**.OUT [operand]** (C-128 only)

If you are burning an EPROM, outputting to tape or modem, or perhaps encrypting your code you might need to use the .OUT command. Beginning on pass two, POWER ASSEMBLER JSRs to the address following the .OUT with each byte of code. This byte will be in the accumulator. Do what you like with it then RTS back to POWER ASSEMBLER and wait for the next. You should use the .BANK # command to tell POWER ASSEMBLER which bank your routine is in if it is not in bank 15.

The tape buffer ($b00-$bff) is not used by POWER ASSEMBLER. The RS232 buffers ($c00-$eff) are unused during assembly. Zero page, however, is used extensively. If you must use zero page in your routine you can use the c-128's re-locatable zero page feature to point to your own while executing your code. POWER ASSEMBLER's zero page is actually situated at $fe00. Be sure to point back to it before returning. Here is how it might be done:

```
10 .ORG $B00                  ;tape buffer
20 .MEM                       ;output to memory
30 LDA #$0C                   ;put zero page at $c00
40 STA $D507                  ;z pg pointer register
50                            ;do your thing
```

```
      .                              ;in here
      .                              ;using your z page
  100 LDA #$FE                       ;point z pg. to POWER ASSEMBLER's
  110 STA $D507                      ;at $fe00
  129 RTS
```

It is assumed that the above code executes in a BANK where I/O registers are visible. POWER ASSEMBLER will have already SEI disabled interrupts. Do not enable them in your routine. Writing to $D509 will reposition page 1 (the system stack). If you make use of this, set it back to 1 when you are done.

**.OBJ "FILENAME"**

Quotes are optional in enclosing any disk filenames defined within POWER ASSEMBLER source unless a drive# is specified (ie. the name contains a colon). They are used here for clarity only.

Use the .OBJ command to "save" ML programs to disk. Files are not opened until the output buffer (beneath Kernal ROM) is full on the second pass. Object files will be closed except during actual output from buffer. Large files will be reopened to append. Having files open only when necessary makes POWER ASSEMBLER compatible with C-64 FASTLOAD cartridges (which must kill open files).

If a fatal error occurs on pass one or execution is halted via RUN STOP there will be no empty or unclosed file to have to deal with as is the case with some assemblers. If execution is aborted during pass two after output has begun, due to some fatal error or user intervention, the file is always first closed.

The current program counter is sent as the file header; therefore, an .ORG [address] command will usually directly precede an .OBJ "OBJECT-FILE" program maker. The header address composes the first two bytes of the actual disk file and tells the Basic operating system where to put the code when it is LOAD'ed ,8,1 into memory (C-128 only, use BLOAD'ed).

Any number of OBJect files may be created during a single assembly. Each time POWER ASSEMBLER encounters a new .OBJ "MYPROGRAM" the last is closed before the new one is opened. Of course It will have a different name.

If the output device is not to be device 8 then the .DVO # command should be used to select the device number to use. If the drive is not drive zero use the filename to set the drive number.

```
  10 .OBJ "0:ZIP"                    ;create on drive 0
      .
  500 .OBJ "1:ZANG"                  ;create on drive 1
      .
  1000 .DVO 9: .OBJ "0:ZOWIE"  ;use device 9, drive 0
```

Again, multiple object files which will later be LOAD'ed all over memory, but assembled as one job and sharing a common symbol table, are possible.

.BAS "0:FILENAME"

This command allows for the automatic merging of Basic and assembler source. These programs can be LOAD'ed, SAVE'd and RUN just like Basic ones.

After the .BAS command, write ordinary Basic program source with one major enhancement. In this Basic the SYS, PEEK and POKE commands will be able to refer to symbol table values, as defined in the assembler portion which will follow, by name. These symbol names must appear in quotes. The Basic part may be quite short:

```
100 .BAS "0:YOU-NAME-IT"
110 SYS"MYCODE"
120 END
130 MYCODE =*
140                             ;brilliant assembler source...
```

An END on a line by itself must follow the Basic, telling POWER ASSEMBLER that the source type has changed. The END line will not appear as part of the final object program. If the above source was assembled and the created program "MYCODE" was LOAD'ed and listed, it would look like this:

```
110 SYS 2063 (c-64 only)                 110 SYS 7183 (C-128 only)
```

And that is it! On top of this SYS 2063 (or SYS 7183 for C-128) invisible to the listing, would be the ML code. Trying to modify the above program without re-assembling is not advisable. For instance, adding a line;

```
100 PRINT "MY NAME IS FRED, I HAVE NO HEAD"
```

... would list okay, but crash when run. The code which had been at 2063 or 7183 would now be further up.

.BAS "NAME" is somewhat like .OBJ "NAME" in that it causes a program file to be written to disk. There are, however, two differences.

1. Do not use the .ORG command to initialize the program counter for .BAS created files. POWER ASSEMBLER will automatically set it to $801 on the C-64 or $C01 on the C-128, which is where Basic programs begin.

2. Do not try to use .BAS more than once in your source. Only one hybrid program can be created at a time.

Here is another exceedingly simple example of an ML - Basic source program. Notice how completely Basic is able to access the ML symbol table.

```
10 SYS 999                    ;calls POWER ASSEMBLER
20 .BAS "0:SIMPLE"            ;name of basic prg
30 POKE"CHARACTER",ASC("X")
40 SYS"PRINT'X'ROUTINE"
50 END
60 ;****now the assembler part****
70 CHARACTER =*: .ORG *+1
```

```
   80 PRINT'X'ROUTINE LDA CHARACTER
   90 JMP $FFD2
```

If you use POWER ASSEMBLER to assemble this, then DLOAD "SIMPLE" and RUN it, you will see an X printed on your screen (be still my heart).

Basic may even use the symbol table names in expressions. Anywhere the actual value is needed the quoted symbol may be used. Lines like;

```
   100 FOR N=0 TO PEEK("TABLE'LENGTH")
   110 POKE"TABLE"+N,PEEK("DATA"+N)
   120 NEXT:REM MOVE DATA TO TABLE
```

... could be used. If any of the symbol names referenced were not defined in the assembler source an UNDEFINED SYMBOL error would ensue.

**.LINK "0:NEXTSOURCEFILE"**

This is one way of chaining a number of source files together. The .LINK command will appear at the end of each but the last source file in the chain. It causes POWER ASSEMBLER to LOAD the source file specified into memory before continuing with the assembly. The last program in your chain will end with a .LOOP "0:FIRSTSOURCEFILE" line. The names used will of course be the names you have DSAVE'd your files to disk under.

If you make use of the "+" forward referencing temporary label then the largest source file should be the first in the chain. The address stack for these labels builds up from the end of the program in memory when assembly begins. If a longer source file is .LINKed in it will overwrite these addresses spoiling everything.

**.LOOP "0:FIRST-FILE"**

This tells POWER ASSEMBLER that there are no more files in the LINKed chain. The file name specified by .LOOP will be the first file in the chain. On pass one this file will be loaded into memory and pass two begun. On pass two the .LOOP command signals the end. Any output files are closed and control is returned to Basic. The source program ending with the .LOOP instruction will be sitting in Basic's program buffer.

For the Commodore 64 only, .LINK...LOOP memory chaining allows you to take full advantage of FASTDISK utilities which intercept Basic's LOAD vector. Again though, make sure that the largest source file comes first if you are using forward "+" temporary symbols.

**.FILE "0:SAVED-SOURCEFILE"**

This is a very convenient way of chaining source files together. The FILE command tells POWER ASSEMBLER to assemble the specified source file directly from disk then to return to the next line of the in memory source and continue. A very short program containing nothing but .FILE statements can be used to assemble multiple giant source programs as one. It might look like this:

```
SYS 999 (SYS 4000 for c-128) ;call POWER ASSEMBLER
10 .FILE "0:INITIALIZE"
20 .FILE "0:PROCESS"
30 .FILE "0:THESEROUTINES"
40 .FILE "0:THOSEROUTINES"
50 .FILE "0:MOREROUTINES"
60 .FILE "0:MESSAGES"
```

With this type of setup the assembly process and file chain can be very easily modified. To add a source file called "PROTECTION" to the chain would be as simple as adding a line 70 .FILE "0:PROTECTION" to the rest before running (assembling). Changing the order in which the files are assembled would involve merely switching a few line numbers. To save the symbol table part way through would entail only inserting the line 15 ".SST "0:INIT-SYMS" for example. Altering display options, I/O device numbers and assembly modes (eg. .FAS or .MEM) would also not involve loading, modifying and resaving large source files.

It is not even necessary to save the changes made to the memory-based file chaining program before assembly; it will still be there afterwards.

The relative sizes of .FILEd source programs are unimportant because they are read directly from disk. The temporary label address stack will always be completely safe. The amount of memory available for .MEM output and symbol tables is also maximized by this method of source file chaining.

Large source files and even .LINKed source files may contain .FILE statements. Control will always return to the next line after the specified source has been assembled in from disk. .FILE assembled source, however, may not contain its own .FILE or .LINK commands. This type of nesting would lead to great unhappiness were POWER ASSEMBLER to attempt it. The .LINK and .LOOP commands are ignored in .FILE assembled source.

**.SEQ "0:ASCIISRCFILE"**

This works exactly like .FILE except that the source is expected in ASCII format, not Basic. This makes POWER ASSEMBLER highly compatible with almost any editor or word processor.
With POWER ASSEMBLER you can combine types to produce a single, ML object program using (1) in-memory Basic type source created on the C-64 or c-128 Basic editor, (2) .FILE'ing in SAVE'd source programs and (3) .SEQ'ing in source created on the ASCII editor of your choice.

Source files specified in the .SEQ instruction must have the following attributes:

1. They will be in pure ASCII form. No screen code and no tokenization.

2. Lines will not be numbered. POWER ASSEMBLER will attach a sequence number to each line in a file for display purposes.

3. A carriage return, ie. CHR$(13), will be the last character of each line, and at least two of these will be at the end of each source file.

4. Colons may still be used to link statements on a line, but no line should be longer than 255 characters.

A large source program in this format might possibly assemble slightly faster than if it were in Basic source format. It would not be necessary for POWER ASSEMBLER to un-crunch tokens or to read in the four bytes of overhead associated with link and line number.

## .TOP

In some situations it may be desirable to save only a portion of the symbols defined or used in a program. The .TOP command lowers the symbol table top in so far as any future .SST is concerned, permitting the saving of intermittent symbols only. Symbols defined prior to .TOP, although accessible to the program in every other way, will not be saved. Unless one has a penchant for empty files one should not attempt to .SST immediately following .TOP. Here is probably the most practical application of .TOP:

```
100 .LST "0:HUGE-SYMTAB"
120 .TOP                        ;will not affect coding
130                             ;now a whole bunch
    .                           ;of neat stuff using the
    .                           ;loaded symbol table
500 .SST "0:NEW-SYMS"           ;saves only the newly defined symbols
```

Numerous, completely exclusive symbol tables can be saved from within one assembly just as numerous separate object files can be created.

With .TOP it is possible for two programs to access each other's symbol tables without re-definition problems or phase errors caused by late zero page assignments. If .TOP is not used then every symbol defined prior to the .SST command will be saved.

## .SST "0:SYMBOL-TABLE-NAME"; save symbol table

This can be used to save all or portions of a symbol table. If the above were the last line of your source program all of its symbols might be saved to a file under the name you used.

Use .SST to create a file of kernal routines, important register addresses and memory locations for use in all your programs. There are clear advantages to this.

1. You don't have to type them all in every time you start something new.

2. Your source files will be shorter without the numerous assignment statements.

3. Certain consistency and uniformity will be lent to your source programs. The names of key symbols will not change from one project to the next.

.SST and .LST provide an excellent way of modifying large ML programs without having to re-assemble the entire system each time changes are to be tested.

Imagine that you have developed a sophisticated word processor or game or assembler or something and you now wish to add to it a fancy new feature. You know perfectly well you're not going to get it right the first, second, third or maybe even the twentieth time. We're talking tricky here. The thought of re-assembling the fifteen or so chained files involved with each new try is not the most fun thing you could possibly ever imagine. You'd probably spend more time waiting then working. Try this:

1. Put a call to the new routine in the main source and also assign therein an address to it. This will not be the final destination, just a free, safe place to work on it. So somewhere in the main source will be a line like 5000 JSR NEW'FEATURE, and a line like 50 NEW'FEATURE = 50000.

2. Now assemble the whole thing. Be sure to create an object file via an .OBJ "GREAT-BIG-ML-PRG" and to save its symbols at the end via .SST "ITS-SYMBOLS"

3. You should have then a BLOAD'able version of your program and a copy of its symbol table, ie. the addresses and values of all of the routines, and variables contained in or used by it.

4. Write the new routine. You don't have to get it perfect right off. It should .ORG originate at the address you told the main program it would. The first thing this source will do is load in the symbol table of the main program with a .LST "ITS-SYMBOLS" line.

**.LST "0:ITS-SYMBOLS"**

This will load in the specified symbol table for use by your program.
... carrying on with our example

5. BLOAD the main program in then assemble the new module (routine) right into memory using .MEM. This new module will have as complete access to the main one as if they had been assembled together. Any routines in the large one will be call-able by name from the new one. Any flags, registers or variables in the main one are also at the disposal of the new part.

6. So try the whole thing out. Run it. Crash-boom, or yuk, or whatever. It didn't work but that's okay because you planned it that way. At worst you'll have to re-boot POWER ASSEMBLER, LOAD your ML code and the source for your test program before you can try again. At best you wont have to do any of that before you begin making corrections.

7. Sooner or later you'll get it perfect. Believe. Now remove the line from the main source which assigned the test address to the routine and either .FILE or .LINK assemble them together the way you would have liked to do in the first place if life wasn't so full of mistakes.

If you .LST symbols in before you define any of your own (ie. first), re-definitions will trigger error messages when they occur. Duplicates will not be loaded in. In the case of labels this is usually convenient since it is the latest occurrence of a label that you are probably interested in anyway.

**.BYTE [onebytevalues,...,...]**

This is used to place one byte value(s) into your code. Here are a few examples of .BYTE:

```
10 .BYTE 0,2,4,8,16,32,64,128   ;powers of 2
20 .BYTE <1000,2000,3000        ;low bytes only
30 .BYTE >SUB1,SUB2,SUB3        ;high bytes only
40 .BYTE "a","b","c"+128        ;ascii values
```

Notice that commas separate the operands and that no spaces are included. Also notice how the < and > work: they affect the entire string of values and should not be repeated. This will make setting up high and low byte address tables more convenient.

**.WORD [twobytevalues,...,...]**

Use .WORD to set up address tables. All values following will be treated as two byte values. This means that 10 .WORD $FF,$FF would have the same effect as 10 .BYTE 0,$FF,0,$FF.

Here are some examples of .WORD:

```
10 .WORD DESTINATION-1     ;setup rts jmp
20 .WORD 12*4096,$c000+OFFSET;expressions
```

It would be pointless to use > or < in conjunction with word data since the resulting values would never exceed one byte.

**.ASC "***ASCII TEXT***"**

Use the .ASC command followed by any quote-mode-typeable string of ASCII characters you wish placed in your code.

The opening quote is not optional. Omitting it will result in a "QUOTE EXPECTED" error message.

A closing quote is optional unless of course you wish to include some blanks at the end of your text entry.

For the Commodore 64, the following is a staple routine for printing messages in ML. It is almost always used in conjunction with the .ASC pseudo-op.

```
50 SYS 999                  ;again
70 .ORG 820                 ;sys 820 after
80 .MEM
90  PTR =251
95  PRINT =$FFD2            ; kernal rom
100 JSR WRITE               ;print message routine
110 .ASC "***HI MOM***":.BYTE 13,0
120 RTS
130 WRITE =*
140 LDY #0
150 PLA:STA PTR+1           ;message address-1 on stack
160 PLA:STA PTR
170 - INC PTR
```

```
180 BNE +                     ;to line 200
190 INC PTR+1
200 + LDA (PTR),Y
210 BEQ +                     ;to line 240
220 JSR PRINT
230 BNE -                     ;jump to line 170
240 + LDA PTR+1:PHA           ;restore rts address past zero
250 LDA PTR:PHA
260 RTS
```

The preceding WRITE routine works much the way Basic's PRINT command does in that following text is printed. A zero marks the end of WRITE text. If you examine this routine you will see how the 6510 stack works during JSR and RTS executions.

The C-128 only, has a new kernal routine to print out strings of text. This text cannot be longer than 255 characters and must be terminated by a null (zero).

Here is an example of this routine used in conjunction with the .ASC pseudo-op:

```
50 SYS 4000                   ;again
70 .ORG $B00                  ;sys 2816 after
80 .MEM
90 FOREVER =*
100 JSR $FF7D                 ;kernal primm routine
120 .ASC "HI MOM":.BYTE 13,0
130 - JSR $FFE4               ;kernal get keystroke
140 BEQ -                     ;loop if no key
150 JSR $FF7D                 ;primm routine again
160 .ASC "BYE MOM":.BYTE 13,0
170 JMP FOREVER
```

Note: don't try JMPing to $FF7D.

## .SCR "***SCREEN CODE VALUES***"

.SCReen works the same as .ASC except that following text is converted to its screen code equivalent. That is the value you would use to poke the character directly to the screen.

The line 100 .SCR "A" would code the value 1 whereas the line 100 .ASC "A" would code the value 65. This should make life a little easier for programmers who maintain menu lines and displays by "poking" character values directly to the screen.

## .FAS

For the Commodore 64, .FASt switches off the screen. This should increase in-memory assembly speed by about 20 percent.

For the Commodore 128, .FASt switches the micro processor into the 2mhz mode and turns off the video. This should at least double the in-memory assembly speed.

There is no danger of missing any important messages by doing this. If any errors are encountered the screen is turned back on for you. It would be pointless, and a waste of time to use .FASt and .DISplay together.

## .BURST (C-128 only)

The .BURST command is for disk based (ie. .SEQ and .FILE) assembly using the 1571. When .BURST is used source files, instead of being read via kernal routines a line at a time from disk, will be burst loaded into memory at the bottom of bank 1. From here they will be accessed RAM DISK fashion by the assembler. This more than doubles the speed of disk based operation making this almost as fast as .LINK/.LOOP load chaining which is always burst driven.

If you are using the .FILE or .SEQ commands, have a 1571 and can spare low memory in bank 1 during assembly then .BURST is highly recommended. It need only be used once at the beginning of your source. If you are using more than one drive and only one is a 1571 the others will not be affected.

## .PSU

.PSeUdo allows for the use of mnemonics like LAX, DCM, INS, SKB, AXS, .etc to code non-standard opcode. The reliability of some of these are somewhat moot. I would suggest you you execute them with interrupts disabled. Some very widely distributed commercial programs make extensive use of non-standard opcode both to conserve space and to confuse disassembly.

Like most inherent (operand-less) pseudos it is a toggle command. Using it for a second time will turn the feature off. You will probably want it on only for those portions of code which make use of non-standard opcode. As with standard mnemonics like LDA and INX you will have to also avoid giving symbols in your program the same names as non-standard mnemonics when .PSU is enabled.

See the table appended to this manual for a full listing and brief descriptions of the pseudo mnemonics which POWER ASSEMBLER recognizes.

## .IF [operand]; conditional assembly

When the expression following an .IF is not equal to zero then assembly will proceed until an .ELSE is encountered, then skip to an .IFE line marking the end of conditional assembly or another .ELSE.

When the value following .IF equals zero then POWER ASSEMBLER will ignore everything until an .ELSE or an .IFE is found. Assembly will resume there.

## .ELSE

This is where assembly will pick up when the value following the previous .IF was zero. If a second (third, fourth...) .ELSE follows, assembly will alternate between them.

```
20 .IF FLAG
30 :    LDA "A":JSR $FFD2      ;kernal print
40 .ELSE
50 :    LDA "1":JSR $FFD2
60 .ELSE
70 :    LDA "B":JSR $FFD2
80 .ELSE
90 :    LDA "2":JSR $FFD2
100 .ELSE
110 :   LDA "C":JSR $FFD2
120 .ELSE
130 :   LDA "3":JSR $FFD2
140 .IFE                       ;end of conditional assembly
150 :   LDA "!":JMP $FFD2
```

If flag = 0 in the above then the assembled code would print "123!", otherwise the code would print "ABC!"

Another more useful application of .IFE .ELSE conditional assembly would be to protect your indirect jumps from accidentally falling on page boundaries.

```
10 JMP (INDIRECT)              ;to destination
   .
   .
   .
500 .IF <*+1                   ;check for page boundary
510 INDIRECT =*                ;not page boundary
520 .WORD DESTINATION
530 .ELSE
540  NOP                       ;pass page boundary
550 INDIRECT =*
560 .WORD DESTINATION
570 .IFE                       ;end of conditional assembly
```

No re-definition of a symbol error would occur during the above assembly. Only the .IF or .ELSE portion of the actual source would be assembled. This would depend on whether or not <*+1 (the low byte of the program counter + 1) was zero.

If you are using a number of .ELSEs you might want to take advantage of the fact that pseudo-ops can be extended and tack some alternating character on telling you which condition each else belongs to, ie.

.ELSE1,...ELSE0,...ELSE1,...ELSE0, etc.

**Note:** Don't try JMPing to $FF7D. Never stick a label in front of an .ELSE or an .IFE. POWER ASSEMBLER would look no further and miss the switch.

### .IFE

This ends conditional assembly. Everything following is assembled.

## MACRO-OPS

Three of the most common activities in machine language involve (1) comparing pointers, (2) filling, ie. erasing, ranges of memory, and (3) moving ranges of memory. POWER ASSEMBLER has provided macro-ops to make short work of these traditionals while enhancing the readability and reducing the size of your source.

All require operands which are expected to be in the form of zero page pointers. While this may seem a trifle inconvenient at first glance it makes the resultant code much more flexible.

For instance, you do not have to use the .MOVE macro every time you want to relocate some range of memory. It would be much more efficient to use it once as a subroutine (ie. preceded by a label and followed by an RTS) and to JSR to it with its three pointers set to your specific needs on each particular occasion. This would not of course be possible if this macro-op took constants as operands.

Another advantage to taking pointers is that you can choose precisely what addresses will be used by generated code. Only the pointers you specify and the processor's registers are manipulated. Bask in the joyous awareness that your data and variables will always be safe when macro coding; trip on the absolute power you exercise over memory usage when employing POWER ASSEMBLER's macros.

I have come into contact with a number of very proficient and talented, professional assembly language programmers over the last several years and not one has confessed to having ever used macros. I believe this is because by their very nature ML programmers enjoy the exquisite control they have over their machines and do not wish to relinquish this to something "standard." Perfection is the order. Custom subroutines seem to hold more appeal than built-in, space-wasting, other-people's macros.

However, the few that have been selected for POWER ASSEMBLER are universally applicable. To overcome your apprehensions about using them I would suggest that you use UNASM to disassemble the code generated by each. You will find it totally re-locatable and non-self-modifying as well as fast, efficient and correct.

### .TEST ZEROPTR1,ZEROPTR2

In situations where you wish to compare two addresses designated indirectly by zero page pointers you could use the .TEST macro-op. The carry returns clear if the first was pointing to a lower address, otherwise it will be set. The Z flag is set if they both point to the same address.

.DUMP BEGINPTR,ENDPTR

This dumps the contents of the accumulator to a range of memory. It might be used quite effectively to clear buffers or hi-res screen areas. The first pointer must designate the first address to be filled and the second pointer the last. Make sure that they are properly set and that the A register has been loaded with the desired value before you use (or call the subroutine using) the .DUMP command. In the following exciting demonstration of it the 40 column screen is filled with "B"s

```
10 SYS 999 (C-64)                       SYS 4000 (C-128)
20 .ORG 820:.MEM
30 SCREEN =1024            ;in 40 column mode for C-128
40 LDA <SCREEN:STA TOPPTR
50 LDA >SCREEN:STA TOPPTR+1
60 LDA <SCREEN+999:STA BOTPTR
70 LDA >SCREEN+999:STA BOTPTR+1
80 LDA "B"                 ;screen code for "B"
90 .DUMP TOPPTR,BOTPTR
100 RTS
```

.MOVE BEGINPTR,ENDPTR,DESTINATIONPTR

This will generate the code to move the range of memory specified by the first two pointers to begin at the address pointed to by the third pointer. The range can be moved in either direction any distance without overwriting itself. In other words, it does not matter whether the destination is above or below the beginning of the range to be moved or if the distance is very small. Memory will still be moved intact. This macro is used in EDITOR.64 or LABELGUN (C-128 only) to shift ranges of source up or down when inserting or deleting text and replacing strings with others that are longer or shorter. Of course the memory being moved (your source) cannot be corrupted in any way.

Write the following short program to locate in the cassette buffer.

```
10 SYS 820                                SYS4000 for C-128
20 .ORG 820:.MEM for C-64    .ORG $B00:.MEM for C-128
30 FROMPTR =251              ;safe basic zero page
40 TOPTR   =253
50 DESTPTR =65
60 .MOVE FROMPTR,TOPTR,DESTPTR
```

Now use the UNASMbler to disassemble and examine it. Notice that only the pointers you defined and the micro processor's registers are used. Try moving some memory around. Convince yourself that .MOVE works and is safe. Almost every ML program ever written uses memory moves. Getting comfortable with this POWER ASSEMBLER macro can save you time and trouble.

## WRITING YOUR OWN COMMANDS

There is space in POWER ASSEMBLER's pseudo-op stack for up to five new commands. Each one takes up five bytes of memory. The first three, which are currently spaces will be replaced by your own three-letter command which you will make up all by yourself; the next two will be the address-1 of the routine you want to execute when the assembler comes across this command.

A symbol table for each version of your assembler is on the system disk. To display one use the following technique:

```
10 SYS 999 (for C-64)                    SYS 4000 (for C-128)
20 .DIS                      ;to display to screen
30 .LST BUDDYSYMS
```

The symbol you will use to get your commands into the code is called PUT'YOUR'CMDS'HERE"; and nothing could be easier than putting your commands there. Let us create a new feature for POWER ASSEMBLER called "fun"; every time the pseudo-op .FUN is encountered in your source POWER ASSEMBLER will inform you that fun is being had; what could be nicer?

```
10 SYS 999 (for C-64)                    SYS 4000 (for C-128)
20 .LST BUDDYSYMS            ;so you can use them
30 .ORG PUT'YOUR'CMDS'HERE
40 .MEM                      ;now we put "fun" on the stack
50 .ASC "FUN"                ;no period here
60 .WOR FUNROUTINE-1         ;address of new useful routine less one
70 .ORG 832 ($B00 on C-128)  ;we'll put it in the cassette buffer
80 FUNROUTINE =*             ;powerful new command
90  JSR  MESSAGE                ;POWER ASSEMBLER's print messages
subroutine
100 .ASC "WHEEEE! THIS IS FUN."
110 .BYT 13,0                ;must end with zero
120 JMP NEWLINE              ;POWER ASSEMBLER takes over
```

After running this, run the following:

```
10 SYS 999 (SYS 4000 on C-128)
20 .FUN
```

Your "fun" message should have been printed twice: once on each pass. If it wasn't then it's your fault. Fix whatever you did wrong, try again, and be more careful this time, eh.

## IMPORTANT ROUTINES AND LOCATIONS

Every source line is de-tokenized into memory beginning at the address of the **BUFFER** symbol. A zero byte marks the end of that line.

If you generate output you should call POWER ASSEMBLER's NEWPC routine. First set **BYTES** to the appropriate value, not greater than three. Put code generated at **OUTPUT**, **OUTPUT+1** and **OUTPUT+2** as necessary. You may call NEWPC more than once (ie. in a loop). When you are done, a **JMP** NEWLINE; passes control back to POWER ASSEMBLER.

If your command takes an operand you can immediately JSR the **EVALOPERAND** routine. Any valid POWER ASSEMBLER expression will be evaluated and the value returned in SUM and SUM+1.

PASSNUM will be 0 on pass 1 and 255 on pass 2.

Try changing the previous .FUN command so you can use .FUN 100 to print the "fun" message 100 times, but only on pass 1.

Of course there are many, many more routines and flags and variables that you will want to become familiar with if you plan to really get intimate with the inner workings of your assembler. You have symbol tables. You have a powerful unassembler. You have fun.

## TEMPORARY SYMBOLS

### TEMPORARY LABELS: - / +

The multiplication, division, addition and subtraction characters each have two possible uses. In expressions, if "*" is an arithmetic operator then values on either side are multiplied (eg. 12*4096); whereas, if it is used as a symbol it will represent the program counter (eg. LABEL =* or *=*+4). This is standard use of "*" and is mentioned only to illustrate traditional dual functioning of one special character.

In POWER ASSEMBLER source the "+", the "-" and the "/" also serve two purposes. In addition to their standard application in arithmetic, they may be used as temporary labels. Many ML programmers don't like having to think up symbol names for numerous, routine, short branches. This is especially so in very long programs after all variations of the labels SKIP and LOOP and BACK and AHEAD and OVER and so on... and so on... have been exhausted. Objections to using these often random symbols are based on the following:

1. Time and effort are wasted in deciding on their names and typing them in, each at least twice.

2. They have a tendency to camouflage more meaningful symbols, making it harder to visualize what is happening.

3. Symbol tables become unnecessarily large, wasting memory and slowing things down.

Judicious use of POWER ASSEMBLER's three temporary flags smartly overcome all of these difficulties.

### TEMPORARY BACKWARD REFERENCING

When the "-" is used as a symbolic operand, the last occurrence of it as a label is referred to. The command BNE - will code a conditional branch back to the last line flagged with a "-" character. Here is how it might be used in a simple time delay routine:

```
100 WAIT =*                      ;name of subroutine
110 LDX #0                       ;initialize x and y
120 LDY #0
```

```
130 - DEX
140 BNE -                     ;loop back until x=0
150 DEY
160 BNE -                     ;same for y
170 RTS
```

Up to three minus signs may be used together as a symbol (eg BCC ---) to refer back as far as the third last "-" flagged line; only the last three are remembered. The minus sign may be used as a label again and again in your source without re-definition errors. You must be careful that when you use "-" characters symbolically that the line on which the referenced one has occurred as a label is the one you want to access (.eg branch to). Any "-" markers prior to the third last one are inaccessible.

## TEMPORARY FORWARD REFERENCING

The plus sign, as you may have guessed already, works in just the opposite way. That is, BNE + would code a conditional branch to the very next occurrence of "+" as a flag. Here is how one might use it to increment a pointer.

```
10 INC PTR                    ;the low byte
20 BNE +
30 INC PTR+1                  ;the high byte
40 + RTS
```

A symbol could have been used instead of "+", but what a bother, a mess and a waste of space.

There is no limit to how far forward the next "+" flags may be or how far back the last "-" flagged lines may be. JMP -- or JMP ++ are valid too. Within their scope of three, these temporary flags may be dealt with just like any other symbol. Still, all subroutines and data should be given meaningful labels even if you could get away with a "+" or "-" temp.

The next three "+" flagged lines may be referenced at any point by using 1 to 3 "+"'s (eg. BEQ +, BEQ ++ or BEQ +++) as a symbol just as any of the last three "-" flagged lines may be accessed using 1 to 3 "-"'s.

Don't let temporary labels permit you to become too un-imaginative. Restrict their use to short, redundant branches.

## FORWARD OR BACKWARD

When the "/" character is used as a label it serves as both "+" and "-", either of which can be used to reference it. In effect it is as though the "/" flagged line had both "+" and "-" as a label on it. The JMP - statement would actually code a jump back to either the very last "-" or "/" flagged line. A JMP + would code a jump forward to the very next "/" or "+" label position. In the next example both conditional branches target the RTS in the middle.

```
10 BEQ +
20 LDA #0                   ;or whatever
30 / RTS                    ;destination of both branches
40 DEX                      ;or whatever
50 BEQ -
```

### TEMPORARY SYMBOL MANAGEMENT

The backward referenced "-" label is handled only on pass two. Only three addresses need ever be "remembered" by the assembler with regard to it. The forward referenced "+" can not be dealt with so easily. A table of all of its occurrences as a flag is created on pass one which is then accessed on pass two. This table is separate from the normal symbol table and contains only addresses. It builds up from the end of your source.

If you are using the memory based

```
.LINK "NEXTFILE". . . .
.LOOP "FIRSTFILE"
```

system to chain source files together and you have made use of any temporary, forward "+" references you should make sure that the largest file in the chain comes first; otherwise, a larger file, when loaded into memory will clash with the "+" address table. Consider disk based .FILE "ANYFILE" chaining as an excellent alternative to memory based chaining in this situation.

## LABELGUN (for C-128 only)

The C-128 screen editor is an excellent one. With it you can redefine keys, freeze scrolling, delete ranges, renumber, auto line number and much more.

About the only thing missing when it comes to developing a large program is sophisticated string handling. To be able to seek out occurrences of and possibly modify a given symbol (.eg string of characters) instantly throughout an entire source program is so useful as to be almost essential.

With Bud installed you have this ability. So never strain your eyes scrolling through screen after screen of source looking for that elusive BUG subroutine. Just enter the following command:

```
L,BUG
```

Every line in your program with the word BUG on it will be listed for you. Change every occurrences of BUG to CRITTER like this:

```
C,BUG,CRITTER
```

In the above case words like DEBUG, BUGEYES and BUGGY would also be changed. This may or may not be what you had in mind.

To have only whole words considered you would use a period in place of the first comma.

```
C.X,EXITROUTINE
```

This would not ruin all your words containing X's. Only if X occurred as a whole symbol would it be changed to EXITROUTINE. All those LDX, INX, STX and TXA commands would go un-molested.

Sometimes the string you seek will contain a Basic keyword but not have been tokenized by the basic editor. This may be due to its following a DATA or REM string on a line or because it exists between quotes. In this situation it is possible that the string you target, even though it looks the same as in your program, will not be found by Labelgun.

If you have your doubts or if you are after a string you know is in quotes, do this:

```
L"ENDING
```

or

```
C"STOPTHIS,STOPTHAT
```

You may put a period at the end of any Labelgun command to add extra spaces to the end of a string;

```
L,MODULE .
```

... would find any subroutines whose names ended in MODULE, but probably not calls to them.

You will find these string handling commands virtually indispensable. Use them to update label names that have changed their meaning. Quickly locate routines by name. If you have source for the C-64 around that you would like to convert to the C-128, Labelgun can help.

Source written on the C-64 editor can be assembled by Bud, but source written on the C-128 might not work with a C-64 basic environment assembler because of the much larger set of tokens used on the C-128.

## TWO ENVIRONMENT EDITOR

Buddy-System 64 actually encompasses two machine language development environments. It is the POWER ASSEMBLER half which has been discussed so far. Although POWER ASSEMBLER is able to assemble ASCII files from disk such as can be written on EDITOR.64 (or EDITOR.128) or on most word processors, its memory based source must be in Basic format. Basic source, unlike pure ASCII text, is actually a linked list: each line starts with a two byte pointer to the next. Following this pointer are two more bytes representing the line number. Next comes the actual text with all Basic keywords tokenized (ie. crunched). At the end of each line is a zero byte.

While this format does very well for Basic it may not be the most efficient for assembly language. However, many programmers are comfortable with the

Basic editor and source format, have acquired utilities such as POWER-64 which greatly extend its capabilities, and have no desire to switch to a different system. If you are one of these people then stay with POWER ASSEMBLER; it was made for you.

## LOADING EBUD

On disk is another version of the assembler which can be invoked by entering **LOAD "EBUD",8** **<RETURN>** and then **RUN**. This will result in the editor compatible version of your assembler, ED-BUDDY.64, or ED-BUDDY.128, and the ASCII editor itself, EDITOR.64, or EDITOR.128, being loaded into memory. You will not return immediately to basic as is the case when booting with POWER ASSEMBLER.

## **EDITOR.64** (or EDITOR.128)

Printed at the top of your screen will be **COLUMN:1 LINE:1**; a solid cursor will be in the upper left corner of the now clear text area. Welcome to our editor!

## MEMORY USAGE

EDITOR.64 commandeers the highest 2K of Basic RAM and sets the top of Basic to point below itself. Thus it is safe from Basic activities and any utilities (such as UNASM) which are sensitive to Basic's pointers. Although 2K is quite small by some standards the editor, as you will soon see, is no weakling.

## **REPLACES BASIC EDITOR** (C-128 only)

EDITOR.128 effectively replaces the Basic editor insofar as the EBUD version of your assembler is concerned. Basic is still completely at your disposal, but you will not be using its line number oriented editor to write your source on or assemble your source from. EDITOR.128 is short, as editors go, and easy to learn to use. Nonetheless, a number of very useful features have been built into it.

## **4-WAY SCROLLING and PAGING**

Begin typing. When you come to the right of the screen instead of wrapping to the next line as you would in Basic the screen window scrolls with you to the right. Lines may be up to 250 characters long. With text in memory you can scroll up, down, left and right by using the cursor keys. You may also page up and down with the f3/f4 key and page left and right with the f5/f6 key. This allows you to flip through your source very quickly. The CLR HOME key can be used to position you immediately to the top or bottom of your source.

## SIMPLE INSERT and DELETE

The INST DEL key works pretty much the way it does in basic to add or remove text one character at a time. the f1/f2 key can be used to delete the remainder of a line or to insert a new line. This key can also be used to split and join lines.

## CUT and PASTE

To delete an entire range of text position the cursor at one end of the text you wish to remove, then press <LOGO> S to Set Range. You will see [RNG] appear at the left of your status line next to COLUMN: Now move to the other end of the range of text to cut. It does not matter how far or near this is. Press <LOGO> D and this text will all disappear. Pressing <LOGO> S twice in a row takes you out of the Set Range mode.

Once you've cut a range of text you may paste (insert) it back in anywhere, as often as you like and even move blocks of source between files. To insert the range simply position the cursor to where you would like it to begin and press <LOGO> T for Text and presto--there it is again.

You may go back and forth from Basic, clear (new) source and load files without disturbing cut text so that routines can easily be moved from one file to another.

## FIND and REPLACE

To find occurrences of any word or words in your source, press <LOGO> F for Find. This will temporarily position you on the status line. Following the "OLD:" prompt, enter the string of characters you would like to find. When you are done press <RETURN> to get back to where you were in your text. Move to where you would like the search to begin (to search all your source press <SHIFT> CLR HOME to go to the top) then press the f7 key. Every time you press f7 your cursor will move to the next occurrence of the target string you entered until you reach the bottom of your source.

If you would like this target string replaced in your source with something else, press <LOGO> R for Replace. Again you will move to the status line where following the prompt "NEW:" you will type in whatever you would like to change the "OLD:" stuff to.

You may proceed in two ways: (1) If you press f7 only the next occurrence of the old will be replaced with the new. (2) if you press f8 then all occurrence following will be changed and you will finish at the end of your source.

## LOADING and SAVING TEXT

Text may be kept as either sequential or program files. ASCII Sequential files can be disk assembled by either version of your assembler via SEQ FILENAME. Program files can be .LINK/.LOOP load chain assembled (ie. assembled directly from memory) by EBUD only. The C-64 can also take advantage of FAST LOAD/SAVE cartridges for the 1541.

## PROGRAM FILES

To save your source as a program file simply press RUN STOP to return to Basic, then enter SAVE "MY-STUFF",8 (C-64) or DLOAD "MYSTUFF" (C-128) just the way you would any Basic program. To load this file back in tomorrow you would you enter LOAD "MY-STUFF",8 (C-64) or DLOAD "MYSTUFF" (C-128).

## TO & FROM BASIC

To return to the editor from Basic use the ED command. If you loaded something new it will be there; otherwise whatever you were working on will still be waiting, unless of course you gave the NEW command to Basic in which case your source will have been cleared.

## SEQUENTIAL FILES

To save and load sequential files it is not necessary to leave the editor. To save a file as a SEQ file begin by pressing <LOGO> P. Then, following the "PUT:" prompt enter the name you would like to give your source on disk.

To load a SEQ file press <LOGO> G and following the "GET:" prompt type in the name and press RETURN. The file will be loaded in, beginning at the position of the cursor. This can be used to join two files.

## ASSEMBLING

To assemble editor source, first press RUN STQP to return to Basic. Then enter the AS command. The source in the editor will be assembled directly from memory. It is not necessary to save it first (unless you plan to kill the machine). You may then issue the appropriate SYS command to test the code and (hopefully) return to your still intact source via the ED command afterwards. Complete memory based operation is supported. With EBUD and EDITOR.64 or EDITOR.128 you can also disk assemble, file chain, load and save symbol tables, create object files, and indeed do all of the things POWER ASSEMBLER does with the Basic editor.

## SOMETHING TO TRY (C-64 only)

The source for the BUDDY-UNASMBLER is on disk in sequential format. Either version of Buddy will assemble it from disk to memory at 50000. To create a program file of ML code from this source you will have to use EBUD. After loading and running EBUD use <LOGO> G to get UNASM-SOURCE into memory.

Change the .MEM on line 2 to .OBJ UNASM.OBJ then RUN STOP to Basic. Enter the AS command to assemble UNASM-SOURCE creating UNASM.OBJ which can now be LOADed...,8,1. If you would like a version to load somewhere else in memory change the .ORG 50000 line.

Now you've got a really powerful and fast memory based unassembler that will convert code to true POWER ASSEMBLER source.

## CONVERTING SOURCE TO ASCII

On disk is a program called MAKE-ASCII that will create an ASCII file completely compatible with the EBUD system from any Basic format source file such as created by UNASM and used by POWER ASSEMBLER.

## LOAD and RUN MAKE-ASCII

Enter the name of the Basic file followed by the name of the ASCII file you would like to create. It will be done. You can get this file into EDITOR.64 or EDITOR.128 using the <LOGO> G command to load a sequential file. You will probably see that this new ASCII file consumes less space on disk than the original Basic one did.

## EDITOR COMMAND SUMMARY

| | |
|---|---|
| f1 | delete rest of line |
| f2 | insert new line |
| f3 | page up |
| f4 | page down |
| f5 | page right |
| f6 | page left |
| f7 | find/replace next occurrence |
| f8 | replace all occurrences |
| CLR | top of text |
| HOME | bottom of text |
| <LOGO> S | start set range |
| <LOGO> D | delete range |
| <LOGO> T | insert range |
| <LOGO> F | set string to find |
| <LOGO> R | set string to replace |
| <LOGO> P | save (put) seq file |
| <LOGO> G | get (load) seq file |
| RUN STOP | go to Basic |
| ED | go to editor |
| AS | assemble source in editor |

## ZBUDDY (for Commodore 128 only)

The following is intended to assist the more advanced ML programmer in making use of the C-128's Z/80 micro processor via the very powerful cross assembler, ZBUDDY. ZBUDDY lets you use standard Z/80 mnemonics (see "TEST.ZMNE" program on disk) and BUDDY's expression syntax and rich body of pseudo-ops (see those sections of this manual) to create ML code for the 128's "other" micro processor. Symbol tables for these assemblers are fully compatible (ie. symbols can be .SST saved on one and .LST loaded by another) so that complex programs involving both the Z/80 and the 8500 can be written.

## PROGRAMMING THE Z/80 (C-128 only)

The C-128 is a two processor system. Inside are an 8500 and a Z/80. The Z/80 is one of the most advanced 8 bit processors alive. It, unlike the 8500 which is memory based, is a register based micro processor. It has two sets of general purpose registers. Each of these sets contains an accumulator, a status register and six, 8 bit, general purpose registers. The second set can be used for the interrupt flip-flop (IFF) or by the exchange (EXX) command to remember and restore register contents. Data registers can also be paired for 16 bit addressing and arithmetic. In addition to these there are four other 16 bit registers: the PC (program counter), the SP (stack pointer) and the (IX) and (IY) (index) registers.

## 8 BIT INTERNAL REGISTERS (C-128 only)

| | | |
|---|---|---|
| A | A' | accumulator |
| B | B' | general purpose |
| C | C' | |
| D | D' | |
| E | E' | |
| H | H' | |
| L | L' | |
| F | F' | flag (status) |

## 16 BIT REGISTER PAIRS (C-128 only)

| | |
|---|---|
| BC | B=hi byte C=low byte |
| DE | D=hi byte E=low byte |
| HL | H=hi byte L=low byte |

## TRUE 16 BIT REGISTERS (C-128 only)

| | |
|---|---|
| IX | index |
| IY | index |
| SP | stack pointer |
| PC | program counter |

## COMMANDS (C-128 only)

The Z/80 recognizes several times as many instructions as the 8500; some therefore require more than one byte of opcode. These commands can be functionally divided into 13 groups.

## 1. THE EIGHT BIT LOAD GROUP (C-128 only)

The Z/80 assembler load instruction, LD, might more aptly be named MOVE. There is no store instruction. Every LD will be followed by two operands delimited by commas. The first operand represents the destination and the second the source, so that the instruction LD ($C000),A means store the contents of A at $C000 whereas LD A,($C000) would mean load A from $C000. In Z/80 mnemonics, parenthesis define a memory location; otherwise an immediate value is assumed.

2. THE SIXTEEN BIT LOAD GROUP (C-128 only)

This includes all the commands which move two byte values either between registers or between registers and addresses. Included here are the PUSH and POP instructions which is handy since addresses are what stacks are mainly for.

3. THE EXCHANGE GROUP (C-128 only)

Register contents can be swapped with the secondary set or within the primary set. There's nothing like this on the 8500 although we often wish there was.

4. THE BLOCK TRANSFER GROUP (C-128 only)

Set a few register pairs and use one of these to move or fill memory a byte at a time or in a Z/80 controlled loop. The short Z/80 routine which we will later call from Basic to copy its ROM into 8500 visible RAM uses an LDIR loop.

5. THE BLOCK SEARCH GROUP (C-128 only)

As above, the Z/80 can automatically control looping by counting down the value contained in the BC pair and incrementing the address pointed to by DE. Ranges of memory are compared with the A register until a match is found or the BC pair decrements to zero.

6. THE 8 BIT ARITHMETIC AND LOGICAL GROUP (C-128 only)

These allow for manipulation of one byte values in pretty much the same way 6510 programmers are used to. Addition and subtraction are possible with or without carry.

7. THE 16 BIT ARITHMETIC AND LOGICAL GROUP (C-128 only)

Same as above but with two byte values being manipulated. The logical AND, OR and XOR are not found in this group.

8. THE CPU CONTROL GROUP (C-128 only)

Processor and interrupt modes and status flags are handled.

9. THE ROTATE AND SHIFT GROUP (C-128 only)

Many different types of shifts accessing both one and two byte values via a variety of addressing modes are available.

### 10. THE BIT SET RESET AND TEST GROUP (C-128 only)

These commands provide for complete bit addressing. Each takes two parameters. The first will specify which bit (0-7) is to be set, reset, or tested; the second will designate the register or memory location to be manipulated. For example SET 3,(IX+0) would set bit 3 in the address pointed to by the IX register (ie OR it with the number 8).

### 11. THE JUMP GROUP (C-128 only)

Conditional and unconditional, jumps (direct) and branches (relative) are supported. Anyone who has ever had to fake a conditional jump in 6510 via BNE *+5:JMP FAR or an unconditional branch via SEC:BCS NEAR will appreciate the versatility of this Z/80 group.

### 12. THE CALL AND RETURN GROUP (C-128 only)

Subroutines may also be called and returned from conditionally or unconditionally.

### 13. INPUT OUTPUT GROUP (C-128 only)

These are specialized load and store instructions. In the C-128, when accessing I/O memory (D000-DFFF), IN and OUT commands should be used instead of LD.

## PROGRAMMING THE Z/80 IN 128 MODE (C-128 only)

The Z/80 brings a convenience and conciseness to ML programming that is sure to please and impress 6510 assembly language programmers. I hope the above has whetted your appetite for doing a little exploring. It will inspire you to know that this micro processor can be used in conjunction with (not at the same time as) the 8500 in the C-128, even from Basic; switching between them is not much more difficult than switching between memory banks once you know how.

### SWITCHING PROCESSORS (C-128 only)

Bit 0 at $D505 (54533) controls the micro processor mode. If it is turned on then the 8500 becomes active; if it is off then the Z/80 takes over.

You can't just poke it off. A little housekeeping is first in order:

Disable 8500 interrupts via SEI because you are going to switch to a memory configuration in which Kernal ROM is not visible.

To do this, store a $3E (62) at $FF00 (the configuration register). This leaves I/O RAM intact but switches everything else to RAM 0.

## MANAGING TWO PROGRAM COUNTERS (C-128 only)

You're still not quite ready. The Z/80 PC register holds $FFED after 128 initialization. There is a NOP ($00) there. The first actual Z/80 command goes at $FFEE. If you look through the monitor you will see a $CF there. This is an RST 8 opcode byte which will cause the Z/80 to jump (ReSTart) to its own ROM routine at 0008. You do not want this. After moving some 8500 code into place at $3000, the Z/80 would return control to the 8500. The 8500 wakes up exactly where it left off after you switched to the Z/80. If you followed this switch with a NOP (lets not wake it up to fast) and then a JMP $3000 (like the operating system does) you would go into the 128's boot CP/M routine. This is pretty useless from a programming standpoint, so don't bother. Instead, put your own Z/80 code at $FFEE.

## THE Z/80 STACK (C-128 only)

Before you do any Z/80 subroutine calls you should set its stack pointer register (SP) to point to some area that will not interfere with your code or Basic.

The last thing the Z/80 will have to do is to turn the 8500 back on. There are two ways to do this:

```
   LD A,$B1
   LD ($D505),A
```

This is inferior. There is a bleed through condition in the Z/80 mode using this type of store. A $B1 will also be written to underlying RAM. (which is where ZBUDDY sits, making this feature especially bothersome.)

Here is the proper way:

```
   LD BC,$D505
   LD A,$B1
   OUT (C),A
```

Bleed through not occur using OUT storage and all I/O memory between $D000 and $DFFF can be written to. In our Basic coding sample the background ($D021) and border ($D020) are poked via the Z/80 OUT instruction.

Ordinarily you would have to bear in mind that the Z/80 might not necessarily take off at $FFEE the next time you activated it. It, like the 8500, wakes up where it went to sleep. The best procedure for switching back and forth is to try to always put the micro processors to sleep in the same spots. These switches could be followed with jump commands. Before invoking them you could set the jump address for the other micro processor to anywhere you like. Z/80 ROM puts a RET ($C9) command after the 8500 switch allowing the Z/80 to CALL the 8500 from anywhere and return when the 8500 switches back. You can also put an RTS ($60) after the Z/80 switch so that the 8500 can JSR the Z/80.

## TWO RAM ROUTINES FOR SWITCHING (C-128 only)

Now it just so happens that there are two routines high in RAM 0 through which the two micro processors can invoke each other. The 8500 invokes the Z/80 at $FFD0. When the Z/80 returns control, the 8500 picks up at $FFDB. Leave the NOP ($EA). You can take over at $FFDC (65500).

The Z/80 invokes the 8500 at $FFE0. When the 8500 returns control, the Z/80 picks up again at $FFEE--and so on and so on.

## SWITCHER (C-128 only)

On your disk is a small POWER ASSEMBLER source program called "SWITCHER-SOURCE" which handles the Z/80 stack, the user call, and controls the "sleepy time" program counters for the two micro processors while making use of the RAM routines at $FFE0 and $FFD0. SWITCHER thus allows you to easily execute hybrid programs and, as our "INVOKE-Z80.BAS" example shows, even call the Z/80 from Basic.

SWITCHER code sits at 3000, high in the 128's tape buffer. The address of the Z/80 code to be executed should be in the 8500's X (=low byte) and A (=high byte) registers. These can be passed directly from ML or even Basic via the 128's new improved SYS command, which is exactly what INVOKE-Z80.BAS does. The program pokes some Z/80 code in at $6000, then after having SWITCHER get the Z/80 to execute it, continues in Basic. The Z/80 code copies its ROM into RAM at $8000. Notice how easy it is to code this move (4 instructions, 11 bytes). The Z/80 then pokes the screen colours just to show off.

The SWITCHER code isn't long at all, and should pave the way for some serious exploration of the Z/80 language and environment in the 128 by true Commodore O/S hackers. You can use POWER ASSEMBLER to relocate the SWITCHER code and ZBUD to write much more interesting dual processing applications than provided in our little Basic demo.

## POWER UNASSEMBLER

On the program disk is an ASCII source file called UNASM-SOURCE. If you are using the Basic format compatible POWER ASSEMBLER then running the following short program will assemble the necessary code to memory.

For the COMMODORE 64:

```
10 SYS 999
20 .SEQ "UNASM-SOURCE"
```

If you are working with the EBUD version of the assembler then only a .SEQ "UNASM-SOURCE" line need be ASsembled, or you may GET this file into the editor, make modifications to it and assemble it directly from memory. (See the EDITOR.64 section on "SOMETHING TO TRY" of this manual).

For the COMMODORE 128:

```
  10 SYS 4000
  20 .BURST                    ;if you have a 1571
  30 .SEQ "UNASM-SOURCE"
```

If you would like a BLOAD'able object file, insert the following line before the .SEQ line:

```
  25 .ORG 60000:.OBJ "UN-CODE"; any name will do.
```

Do not try to change the load destination to other than 60000 from outside the main source. To do this (1) DLOAD and RUN "EBUD", (2) press <LOGO> G to GET:UNASM-SOURCE, (3) change line 1's .ORG 60000 to your own origin address (then an .OBJ "NAME" line if you want the code saved), (4) press RUN STOP to enter Basic, (5) enter the AS command to assemble everything.

In any case you have a powerful memory based unassembler at your disposal; one that will convert raw code to LOAD'able, LIST'able, SAVE'able source that you can attack with LABLEGUN (C-128 only), modify and/or re-assemble using POWER ASSEMBLER, or convert using MAKE-ASCII to source that can be worked on in EBUD's powerful ASCII editor.

## HOW TO USE UNASM

After assembling UNASM-SOURCE to memory it must be enabled via SYS 50000 for C-64 or BANK 1:SYS 60000 for C-128, (unless you've changed the origin). All this does is set some pointers and print a header. To use UNASM enter the UN command from Basic. Your "UN" will be extended to prompt:

UNASSEMBLE FROM $

Enter a start address in hexadecimal. (You can use POWER ASSEMBLER display to convert decimal to hex if need be). You will then be prompted

TO $

Another hex value must be entered representing the address of the last byte of code to be un-assembled.

HIDDEN RAM $ (C-64 only)

Next you will be asked if ROM is to be banked out. If you are un-assembling hidden RAM such as where BUDDY resides then you would press "Y" for this. If you are un-assembling ROM you must press "N", otherwise it doesn't really matter.

## SELECT BANK (C-128 only)

Next you will be asked to select the bank of memory which the code you want to un-assemble is in. As in the C-128 monitor you will use 0-F to designate banks zero through fifteen.

## FORMAT

Finally you will be asked if you want standard format. You probably do not, so press "N". Standard format cannot be re-assembled; it is for looking at. The line number represents the decimal address of each instruction. Following this will be the same value in hex. Last will be the instruction. Except for the line numbers this resembles the format produced by ML monitors. Again, standard format is for examination purposes, not re-assembling.

Non-standard format produces actual POWER ASSEMBLER source that, with a little work, you can make as good as the original. Line numbers will still represent the address of the un-assembled code. Labels will be generated and used if and only if possible.

Depending on the amount of code being un-assembled you will have to wait from no time at all to about 10 seconds for the job to be done. When Basic is again "ready" enter LIST ...there is your source.

## RANGE LIMITS

UNASM can take on almost 4K of code at a crack. It is sensitive to the Top-Of-Basic pointer ($1212) so that utilities such as your assemblers and editor which use this pointer to protect themselves will never be over-written by UNASM generated source. If you enter a range too large to fit in the Basic buffer no harm will come of it. UNASM will do as much as it can before stopping.

## PROBLEMS

Those of you who try to LOAD...,8,1 and un-assemble EDITOR.64 ($9700-$9FFD) will discover that the code, when assembled back to memory does not work properly. This program like many others has a certain amount of ASCII and other data embedded in it. Where UNASM encounters a non-opcode it will generate a .BYTE instruction to handle it; however, sometimes some rather awful (ie. meaningless) instruction sequences will also be generated by this data. It is up to you to create the appropriate .ASC, .BYTE or .WORD lines to give meaning to this garbled source.

UNASM may also produce source lines like this:

```
 49152 ZC000 ASL $0020
```

Absolute addressing has been used on a zero page address. Whether this was intended or the result of embedded data the assembler will assume you mean ASL $20 and code zero page addressing. The $00 byte is lost and the code is shortened.

## SOLUTIONS

You can correct assembling un-intended zero page addressing by changing such un-assembled source lines to 49152 ZC000 ASL !$20, forcing absolute. The source should then assemble properly to its intended destination although it may not look pretty or be truly useful yet.

You can use POWER ASSEMBLER's .OFF and .MEM pseudo-ops to assemble the code to memory somewhere safe and then perhaps write a short Basic program to compare it byte for byte with the original. You will be able to spot, then list, lines which didn't re-assemble properly.

UNASM can also not possibly know when the low and high byte values of internal addresses are being used in order to set up RTS jumps, intercept vectors, or self-modify. You will have to study the source to see where this is being done and create the correct symbolic expressions for these statements before it will be truly re-workable and re-locatable.

Having a symbol table for the un-assembled code (as you have for the assemblers) can make analyzing it and even reconstructing meaningful source much less work.

For Commodore 128, LABELGUN commands can be used to attach meaningful names to the hex oriented symbols generated by UNASM. MAKE-ASCII can be used to convert the Basic format, un-assemble source to stuff you can work on in the ASCII editor (you'll lose the line number references).

Insufficient disk space makes it impossible to provide complete source listings for your assemblers as part of the system package. However, you should find UNASM-SOURCE and the SYM files an interesting and useful compromise.

## STANDARD INSTRUCTION SET & ADDRESSING MODES

ADC #byte byte byte,x word word,x word,y (byte,x) (byte),y
add memory to accumulator with carry.

AND #byte byte byte,x word word,x word,y (byte,x) (byte),y
logical AND memory with accumulator.

ASL implied byte byte,x word word,x
shift left one bit.

BCC word
branch on carry clear.

BCS word
branch on carry set.

BEQ word
branch on zero.

BIT byte word
test bits.

BMI word
branch on negative (128-255).

BNE word
branch on not zero.

BPL word
branch on positive (0-127).

BRK implied
break execution.

BVC word
branch on overflow clear (bit 6).

BVS word
branch on overflow set.

CLC implied
clear carry flag.

CLD implied
clear decimal mode.

CLI implied
clear for interrupts.

CLV implied
clear overflow flag.

CMP #byte byte byte,x word word,x word,y (byte,x) (byte),y
compare with accumulator.

CPX #byte byte word
compare with x index.

CPY #byte byte word
compare with y index.

DEC byte byte,x word word,x
decrement memory by one.

DEX implied
decrement x index by one.

DEY implied
decrement y index by one.

EOR #byte byte byte,x word word,x word,y (byte,x) (byte),y
exclusive OR with accumulator.

INC byte byte,x word word,x
increment memory by one.

INX implied
increment x index by one.

INY implied
increment y index by one.

JMP word (word)
jump to new location

JSR word
jump to new location, save return address.

LDA #byte byte byte,x word word,x word,y (byte,x) (byte),y
load accumulator.

LDX #byte byte byte,y word word,y
load x index.

LDY #byte byte byte,x word word,x
load y index.

LSR implied byte byte,x word word,x
shift right one bit.

NOP implied
no operation.

ORA #byte byte byte,x word word,x word,y (byte,x) (byte),y
logical OR with accumulator

PHA implied
push accumulator on stack.

PHP implied
push processor status (flags) on stack.

PLA implied
pull accumulator from stack.

PLP implied
pull processor status (flags) from stack.

ROL implied byte byte,x word word,x
rotate left one bit with carry.

ROR implied byte byte,x word word,x
rotate right one bit with carry.

RTI implied
return from interrupt.

RTS implied
return from subroutine.

SBC #byte byte byte,x word word,x word,y (byte,x) (byte),y
subtract memory from accumulator with borrow.

SEC implied
set carry flag.

SED implied
set decimal mode.

SEI implied
disable interrupts.

STA byte byte,x word word,x word,y (byte,x) (byte),y
store the accumulator in memory.

STX byte byte,y word
store x index register in memory.

STY byte byte,y word
store y index register in memory.

TAX implied
transfer accumulator to x index register.

TAY implied
transfer accumulator to y index register.

TSX implied
transfer stack pointer to x index register.

TXA implied
transfer x index register to accumulator.

TXS implied
transfer x index register to stack pointer.

TYA implied
transfer y index register to accumulator.

## NON STANDARD 6510 (.PSU) INSTRUCTIONS & ADDRESSING MODES

ASO #byte byte byte,x word word,x word,y (byte,x) (byte),y
ASL then ORA result with accumulator.

RLA #byte byte byte,x word word,x word,y (byte,x) (byte),y
ROL then AND result with accumulator.

LSE #byte byte byte,x word word,x word,y (byte,x) (byte),y
LSR then EOR result with accumulator.

RRA #byte byte byte,x word word,x word,y (byte,x) (byte),y
ROR then ADC result to accumulator.

AXS byte byte,x byte,y (byte,x)
store result of a AND x.

LAX byte byte,x word word,y (byte,x) (byte),y
LDA and LDX with same memory.

DCM byte byte,x word word,x word,y (byte,x) (byte),y
DEC memory then CMP.

INS byte byte,x word word,x word,y (byte,x) (byte),y
INC memory then SBC.

ALR #byte
AND with value then LSR result.

ARR #byte
AND with value then ROR result.

XAA #byte
AND with x then store in a.

OAL #byte
ORA with #$EE then AND with data then TAX.

SAX #byte
SBC data from a AND x then TAX

SKB byte
skip byte.

SKW word
skip word.

## RECOMMENDED READING LIST

| REFERENCE | AUTHOR | PUBLISHER |
|---|---|---|
| MACHINE LANGUAGE FOR THE COMMODORE 64 AND OTHER COMMODORE COMPUTERS | Butterfield | Brady |
| ASSEMBLY LANGUAGE FOR THE COMMODORE 64 | Sanders | Microcomscribe |
| INNER SPACE ANTHOLOGY 2ND EDITION | Karl Hildon | Transactor |
| ADVANCED MACHINE LANGUAGE | Data-Becker | Abacus |
| MACHINE LANGUAGE FOR BEGINNERS | Mansfield | Compute! |
| SECOND BOOK OF MACHINE LANGUAGE | Mansfield | Compute! |

## BUDDY 128 INDEX

IN-D-PWA