BUDDY 64/128
ASSEMBLY DEVELOPMENT SYSTEM

Written by
Chris Miller

Distributed by
Click Here Software Co.
P.O. Box 606
Charlotte MI 48813
(517) 543-5202

Minimum System Requirements
Commodore 64 or 128 w/Disk drive
TV or Monitor
Printer (optional)

Buddy 64/128 Assembly Development System

TABLE OF CONTENTS

Buddy 64/128 Assembly Development System

1. INTRODUCTION

Buddy 64/128 Assembly Development System is a Machine Language Development Package for the Commodore 64/128 which contains a collection of very fast and versatile machine language development programs designed specifically for the Commodore 128 microcomputer. A rich body of commands and range of abilities set the standard for convenience and flexibility in assembly language programming.

Buddy 64/128 Assembly Development System is written and copyrighted by Chris Miller, and is manufactured and distributed by Creative Micro Designs, Inc., P.O. Box 646, East Longmeadow MA 01028-0646.

Designed for use with the Commodore 64/128 computer and up to four Commodore 1571, 1541 or other certified 100% Commodore compatible disk drives. Buddy, Buddy 64 and Buddy 128 are trademarks of Chris Miller. Commodore 64 and Commodore 128 are trademarks of Commodore Business Machines, Inc.

ACKNOWLEDGEMENTS

I would like to thank John Lam, Darren Spruyt, Brian Hilchie and Steve Punter who have provided valuable ideas and suggestions, and my wife Susan for her endurance, during the development of this package.

Chris Miller

2. Policies

Your BUDDY disk has been fully tested before leaving our premises. However, strange things sometimes happen to diskettes during shipping, so there are some Creative Micro Designs, Inc. policies that you should be made aware of.

The Buddy 64/128 Assembly Development System software is sold on an "as is" basis; however the media (disks) that the software is stored on are warranted to be free of manufacturing defects for a period of 90 days from the date of purchase. If within the 90 warranty period the media becomes defective, the consumers sole remedy will be to return the defective disk(s) to Creative Micro Designs for free replacement. If the disk(s) become defective after the warranty period has expired, you can obtain a new set of disks for the price of $10.00 per disk plus $3.00 shipping. This warranty becomes void if the media becomes damaged due to misuse or circumstances outside of normal wear and tear. Before returning any disks to Creative Micro Designs, Inc. you must first contact our Technical Support department (413-525-0023) to obtain a "Return Material Authorization Number". No merchandise will be accepted unless this number is clearly visible on the outside of the package.

Creative Micro Designs, Inc. and Chris Miller shall not be held liable for any loss or profits either direct or indirect resulting from the use of this product or breach of warranty. Creative Micro Designs, Inc. and Chris Miller reserve the right to make any changes or modifications to better this product as they see fit at any time in the future without notification.

3. SPECIFICATIONS

PROGRAMS PROVIDED

Buddy 64/128 Development System actually encompasses three stand-alone machine language development systems. There are two for writing 8500 (the system microprocessor) code: One uses the Basic editor (enhanced by string search and replace commands) for writing its memory based source and the other its own powerful ASCII editor. There is also a complete Z/80 (C/PM's microprocessor) cross assembler which has all the powerful commands and features of the other two. As a bonus there is also an assembler completely compatible with Pro-Line's C-POWER shell, editor, linker and ramdisk.

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 is the boot for the Z/80 cross-assembler.

BUDDY.ML 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.ML" as generated by its assembly. You can use it to explore the code and to create your own commands.

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

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

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

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.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.

ZBUDDY.ML is the body of the Z/80 cross assembler which is installed when "ZBUD" is run.

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

CALLZ80.BAS is a short example of a Basic program calling Z/80 routines.

CALLZ80.BUD is a BUD 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 make full use of the ZBUD Z/80 cross-assembler.

UNASM.BUD is the complete, documented source program for our powerful unassembler that will convert raw code to source that you can LIST, DSAVE, DLOAD and, best of all, reassemble using BUD.

AS.SH is for those who own Brian Hilchie's excellent C-Compiler published by Pro-Line. About the only thing missing in this superb system is a compatible assembler. AS.SH brings the power of Bud to C; use it to write compilable C functions or stand-alone assembly language programs from within the C-Compilers amazing Shell operating system.

MACROS.SOURCE is the source for Bud's two built in macros "move" and "fill", and is provided to help users write and implement their own macros.

AID128.SOURCE is a public domain scroll utility provided for use with Bud.

128 SYSTEM MEMORY MANAGEMENT

Each version of Buddy runs in bank 1 at $c000. Symbol tables build down from here. Label "+" addresses build up from $2000. Memory in the middle is used as an output buffer. Macro modules as .DFN ....DFE defined append to the end of the

assembler. All of bank 0 is free for Bud or Zbud source. Ebud's editor runs at $e000 in bank 0; the source limit is lowered to $d000. Burst mode .FILE or .SEQ chaining results in files being loaded into bank 0 memory directly above resident source. Memory from $c00 to $fff is used extensively during an assembly as is memory from $200 to $240 and $140 to $1ff. Memory from $1300 to $17ff is unused by Buddy or the 128 operating system and is probably one of the best places to memory test small programs.

64 SYSTEM MEMORY MANAGEMENT

The 64 versions of Bud and Ebud are virtually identical to their 128 counterparts, with the only exceptions being the .BANK and .BURST commands which are not implemented. SYS 999 is the 64 equivalent of the SYS 4000 call to the 128 Bud. The top of Basic is lowered.

The assemblers sit in high basic RAM and beneath Basic ROM. Symbol tables build down from the beginning of assembler code. Macro modules build up from the end of assembler code. The "+" address stack builds up from the end of your source. The first program in a .LINK....LOOP assembled chain should be the largest if "+" labels are used. Buddy is compatible with FAST 1541 cartridges. GT64 by John Lam is especially good.

Labelgun is not implemented in the 64 assemblers; however, excellent utilities such as POWER or SUPERMON can be used with Buddy provided they are not loaded first. Memory from $c000 to $cfff is free for program testing.

COMPATIBILITY

Bud is completely compatible with Basic 7.0, the Commodore disk operating system and its source format. 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. BUD's own EDITOR.128 provided supports 4-WAY, bidirectional scrolling and paging as well as CUT & PASTE, SEARCH & REPLACE and much more.

SPEED

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 chaining or direct disk based assembly. Numerous combinations of memory and disk based assembly are possible. Multiple device handling, allows for the application of any number of disk drives.

OUTPUT

Code can be directed to memory in any bank(s) allowing for fast, in memory assembling and testing of very large programs. Any number of BLOADable, machine language program files, 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 programs can be designed, tested and retested in memory without having to re-assemble the whole system each time. Bud can be instructed to direct all output through custom user routines located in any bank of memory 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

Fast assemble with the new 8500 micro-processor operating screen blanked in its 2MHZ. mode. If/else conditional assembly is supported. Temporarily offset program counter assembling generates patches of code that will be relocated before execution. Set up 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 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) and fill memory make short work of these common and often tedious procedures. Users may easily write their own macro commands and even create new versions of the assembler. 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 faster assemblies.

4. 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: DLOAD"BUD <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 After booting up Buddy is tucked safely away from Basic, high in bank 1 ($c000) awaiting your summons.

Memory from $c00 to $1000 in bank 0 is reserved for Bud 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, Bud 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 Bud takes over. In other words, a SYS 4000 invokes the assembler. Bud will act on all source following. Basic will interpret and try to execute everything up to and including the SYS 4000 line.

Bud builds its symbol tables, output buffer and any user defined macros in bank 1. All of bank 0 is free for Basic style source. If you are using Bud's .BURST command to assemble files from disk these will be temporarily appended to whatever source already resides in memory. The best place to memory test small programs is $1300-$17FF. The system tape buffer $B00-$Bff is also free. Bank 0 memory $1C00-$FFFF is free although in-memory source will use some. Unless a very, very large program is being assembled, $3000-$8000 bank 1 should also be safe for memory based activities.

RELAX If all the hex address stuff and assembler jargon has only served to confuse you don't worry about it. Someday soon it will make perfect sense. The examples in this manual will never have you overwrite sensitive areas with code generated. Just try to remember: if you aren't sure where a good place in memory to put your code is, use $1300 (eg. *=$1300:.MEM). By the time you need more room for your programs you'll know what you're doing.

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 4000:REM sys 999 for 64 version
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 Bud. Familiarity with them will allow you to take full advantage of Bud's many abilities. Notice the use of "-" as a label. This is an example of the use of a Bud 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. Only the closest one counts.

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. Bud 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 Buddy 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 Buddy 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. Bud's string handling utility, Labelgun, can make playing around with names fun. The apostrophe has no special meaning to Bud; 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 Bud 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. Bud 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 MEMORY'CONFIG       = $FF00
4 PROGRAM'COUNTER     = *
```

SET ASSIGNMENTS

You cannot use the equal sign or flagging to reassign a new value to an existing symbol. To reset 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. Bud'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 Buddy 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.

Buddy 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 picture show
```

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.

5. 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 maybe involved. Parenthesis are not supported in Buddy 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 Bud, 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. Buddy 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.

```
1 LDA #0                ;immediate value
2 LDA 0                 ;zero page address
3 STX 0,Y               ;zero page y indexed
4 LDA 0,X               ;zero page x indexed
5 LDA 1000              ;absolute address
6 LDA 1000,X            ;absolute x indexed
7 LDA 1000,Y            ;absolute y indexed
8 LDA (0,X)             ;pre-indexed indirect x
9 LDA (0),Y             ;indirect post-indexed y
10 BNE 1000             ;relative branching
11 JMP (1000)           ;indirect jump
12 LSR                  ;accumulator implied
13 INX                  ;implied
```

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

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

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

6. ERROR MESSAGES

Most of us never make mistakes and have no use for error messages. Still, there are always a few who go and spoil it for everyone else. So for those 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 Buddy 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 in the 2MHZ. mode 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 Bud'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 misspelled 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 try do redefine a macro, used the same flag twice or tried to reassign a value to a symbol with an equal sign.

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 when a negative value or a value higher than $ffff is arrived at during certain expression evaluations.

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 macro is undefined or a symbol was not found (on pass two only) in the symbol table; perhaps it's misspelled 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.

IMPROPER USE OF A MACRO PARAMETER when some error occurs in defining a macro. Perhaps a name does not out follow .DFN or a number is out of sequence. This error will also occur if an &argument is used which was not declared on the .DFN line.

MACRO TOO LONG macros should not run over a page (256 bytes) of code.

.DFE EXPECTED if you try to define a new macro without ending the definition of a previous one with .DFE. It is pointless to continue with the assembly. .DFN EXPECTED if unexpected .DFE is encountered, ie. you are trying to end a definition never begun; again a fatal error.

SYMBOL TABLE OVERFLOW There is no longer room in bank 1 for your symbol tables. This is the only place you are ever likely to see this message.

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. Bud 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 serial 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

Bud 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 Bud 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 Buddy'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.

7. PSEUDO OPS

Here are the pseudo-ops which Buddy recognizes. Where a [word] operand is required any valid Buddy 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
.BANK [1-15]         ;select memory bank
.DIS                 ;display assembly (on/off)
.DIS P               ;display assembly to printer
.DIS E               ;display only errors to printer
.DIS "FILENAME"      ;display output to sequential file
.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
.DFN [NAME !1 ...8] ;define macro begin
.DFE                 ; end define macro
.BYTE [byte, "text",...] ;table one byte value(s)
.BYT$ [byte,...]     ;numeric values default to hex
.BYT% [byte,...]     ;numeric values default to binary
.WORD [word,...]     ;table two byte value(s)
.WOR$ [word,...]     ;numeric values default to hex
.WOR% [word,...]     ;numeric values default to binary
.ASC ["text", byte,...]  ;same as .byte
.SCR ["text", byte,...]  ;screen code text values
.PSU                 ;for non-standard opcode mnemonics
.BURST               ;use 1571 burst mode
.FAS                 ;for 2MHZ. mode with screen off
.END                 ;force end of source
.IF [word]           ;if word <> 0 then continue
.ELSE                ;otherwise skip to here then begin
.IFE                 ;conditional assembly ends
```

Any pseudo-op can be extended or truncated. If you would rather use .WOR or .WO or even .W instead of .WORD, Bud 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 Bud'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 Bud 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 BUD where the machine language output will reside in memory. If it is not used output will ORiGinate at $3000 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 can replace 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 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 uninitialized 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 Bud 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. The following program moves a short "POKEHELLO" routine into the C-128 cassette buffer, calls it, then continues.

```
5 SCREEN =1024              ;in 40 column mode
10 .ORG $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 $B00               ;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...
```

Moves like the above are quite useful in programming the C-128. Bud, 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 Bud 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 Bud to output code directly into C-128 memory. The code will be "poked" into memory at the .ORG address. The .BANK command can be used to select a bank other than the default 15. Consider unused memory from $1300-$17ff a safe testing ground. You may also use $B00-$BFF freely. Any memory not taken up by your source $1C01-$FF00 is safe.

.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. .BANK [0-15] 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. When .OUT [address] is employed to direct output through a special user routine,the .BANK command should be used to point to the bank containing this routine if it is not in bank 15.

.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. When offset coding the destination program counter is shown. For assignment statements the assigned value is displayed.

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

4. The actual source line.

Symbols will stick out to the left a few spaces even if you did not include the white space in your source. Lines extended by colons will be split up. .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 Buddy. 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 either abort the assembly via RUN STOP or freeze with the NO SCROLL key.

The symbol table will be displayed to the printer in two column format. It is an easy matter to suppress Buddy's paging if you would rather print over the perforations. Buddy's FORM'FEED routine can be looked up in BUDDYSYMS. Putting an RTS at the beginning of it will turn of paging for the duration of a session. (see the sample program to do this in the MACROS section of this manual.)

.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 frantically 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 the monitor.

.OUT [operand]

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, Bud 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 Bud and wait for the next. You should use the .BANK # command to tell Bud which bank your routine is in if it is not in bank 15.

The tape buffer ($b00-$bff) is not used by Bud. Free memory from $1300 to $17ff in bank 0 is also unused by the assembler. 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. Bud's zero page is actually situated at $d00. 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 #$13               ;put zero page at $1300
40 STA $D507              ;z pg pointer register
50                        ;do your thing
.                         ;in here
.                         ;using your z page
```

```
100 LDA #$0D              ;point z pg. to Buddy's
110 STA $D507             ;at $0d00
120 RTS
```

It is assumed that the above code executes in a BANK where I/O registers are visible. Bud will have already SEI disabled interrupts. Do not return to the assembler with them CLI enabled. 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 Bud 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 second pass. 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 BLOAD'ed into memory.

Any number of OBJect files may be created during a single assembly. Each time Buddy encounters a new .OBJ "MYPROGRAM" the last is closed before the new one is opened. It must have a different name or a FILE EXISTS bus crash will result.

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 BLOAD'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 DLOAD'ed, DSAVE'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 Buddy 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 DLOAD'ed and listed, it would look like this:

```
110 SYS 7183                  ; for the 128
120 SYS 2063                  ; for the 64
```

And that is it! On top of this SYS 7183 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 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. Bud will automatically set it to $1C01 which is where Basic programs begin in the C-128.

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 4000                          ;calls buddy
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 Buddy 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 a very fast 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 Buddy to DLOAD 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.

.LOOP "0:FIRST-FILE"

This tells Bud 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.

.LINK...LOOP memory chaining is fast, although perhaps not the handiest way to combine source files. To add a new source file to the chain or rearrange the order in which existing source files are assembled it will be necessary to modify and resave two or three of them; also, it is not easy to follow the chain except by .DIS E display or loading in one file after another and checking last lines. More of memory will be needed to hold the source files and their potentially gigantic combined symbol table.

.FILE "0:SAVED-SOURCEFILE"

This is probably a more convenient way of chaining source files together than with .LINKing and .LOOPing. The .FILE command tells Bud 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 commands can be used to assemble multiple giant source programs as one. It might look like this:

```
SYS 4000                    ;call Buddy
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 short file chaining program before assembly. It will still be there afterwards. The amount of memory available for .MEM output and symbol tables is 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.

.SEQ "0:ASCIISRCFILE"

This works exactly like .FILE except that the source is expected in ASCII format, not Basic. This makes Buddy highly compatible with almost any type of source you might have kicking around from your C-64 days. If you have Brian Hilchie's excellent EDitor, or a favorite word processor (most support ASCII output to disk) or even decide to write your own someday, you will always be able to assemble it. Of course you can combine types to produce a single, ML object program using (1) in-memory Basic type source created on the C-128 Basic editor, (2) .FILE'ing in DSAVE'd source programs and (3) .SEQ'ing in source created on the ascii editor of your choice. 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. Buddy 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 Bud to un-crunch tokens or to read in the four bytes of overhead associated with link and line number. On the other hand, this might be offset by the fact that lack of tokenization would make the file larger.

```
.DFN COMMANDNAME 1 2 . . . ;define a user macro
```

.DFN is used to write your own macro commands. It takes a name (the name you will use to implement your command) followed by up to 8 numbers as arguments. Each number will be preceded by either an UP ARROW or an EXCLAMATION MARK designating the parameter as either a byte or word value. Following lines may use these numbers preceded by the "&" character as variable addresses or immediate values.

No output occurs during .DFN mode and the program counter is temporarily set to page 1 while the module is built. These modules should not self-modify or employ absolute internal references like JMP ELSEWHERE'IN'MACRO.

Do not use multiplication, division or the < > operators on the & parameter when in .DFN mode. Do not use the & variable in .BYT or .WOR data. Do not try to write macros to replace long, complex pieces of coding. Macros serve best for those repetitive little operations usually associated with pointer manipulations or faked long branches. They can reduce the size while improving the look of your programs and actually result in fewer of those silly little coding errors that machine language programmers have all come to love so well. Here are just a few simple examples:

```
10 .DFN ADDVAL 1 2 ;add immediate value (1) to pointer (2)
20 LDA &1
30 CLC
40 ADC &2
50 STA &2
60 BCC SKIP
60 INC &2+1
70 SKIP .DFE ;end of definition
```

To use this command to add the number 2 to your SCREENPTR in one of your programs you could then code the following line:

```
&ADDVAL 2 SCRPTR
```

You could use this same macro to add the value 1 to a different pointer:

```
&ADDVAL 1 MEMPTR
```

This next common sample macro would code a JMP on Z set.

```
10 .DFN JMPEQ !1              ;one address argument
20 BNE SKIP
30 JMP &1
40 SKIP .DFE
```

Now you've got another command:

```
&JMPEQ $C000                  ;for example
```

Now you try to make &JSRSC ;call subroutine if carry set .DFE ends the macro definition mode. Normal assembly may resume. When writing a number of macros, one after another, it may be safest not to use "+" forward referencing in any of them because of the numerous reversals of the program counter.

INSTALLING YOUR OWN MACROS

Buddy has two built-in macros, &MOVE and &FILL. On the system disk is a file called MACROS.DFN. If you load and list this Buddy source file into memory you will see definitions for a number of fairly common macros using Bud .DFN ....DFE syntax. It is a simple matter indeed to make these or any other commands a permanent part of Buddy syntax just as &MOVE and &FILL are.

1. First assemble the source containing the definitions. As stated, these can be the ones supplied in MACROS.DFN or your own, or both.

2. Next DLOAD in the BUDMACS.INS program and RUN it. If you are using the EBUD version of the assembler use EBUDMACS.INS after you have assembled your new macro definitions.

3. An expanded version of the assembler will be copied from memory. It will have your new command modules appended to the end of it.

Again, you must assemble the new definitions before you run BUDMACS.INS. It is not necessary (or even possible) to direct output of this assembly to memory or any device. Permanently installed commands need never be redefined; indeed if an attempt is made to redefine a macro command that is permanently saved or already defined in your program, a REDEFINITION OF A SYMBOL error will result. Uninstalled .DFN....DFE macro definitions will be initialized away with each new assembly.

.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 programing 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
130                          ;will not affect coding
.                            ;now a whole bunch
.                            ;of neat stuff using the
500 .SST "0:NEW-SYMS"        ;loaded symbol table
                             ;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 = $3000.

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 Bud, BLOAD your ML code and then DLOAD 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), redefinitions will trigger error messages when they occur. Duplicates loaded in will not be used. In the case of labels this is probably 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 "hello world", 13, 0 44     ;ascii values
50 .BYT$ 0 1 2 3 4 5 6 7 8 9 A B C D E F 44 ;hex numbers
60 .BYT% 1111 1010 1111 1101         ;binary values
```

The .BYT$ and .BYT% commands take numbers only as parameters. These numbers default to either $hex or %binary. This will save typing the "$" or "%" over and over when entering numeric data in these bases. Notice that commas may or may not be used to separate the operands. Also notice how the < and > work: they affect the entire string of values. These may be repeated in order to reset the default for following values. This will make setting up high and low byte address tables more convenient.

Notice also that text strings and numeric values can be included on the same line using .byte. .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
20 .WORD 12*4096,$c000+OFFSET       ;expressions
30 .WOR$ ff, ee, dd, cc, bb, aa     ;hex numbers
40 .WOR% 11110000 10101010 11100011 11001100 ;binary data
```

.WORD data can be told to expect hex or binary numbers using .WOR$ and .WOR%. It would be pointless to use > or < in conjunction with a word table since the resulting values would never exceed one byte.

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

.ASC behaves exactly the same as .BYTE. Indeed these two commands can be used completely interchangeably. .ASC is included only to provide compatibility with PAL syntax. The C-128 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" 13,0
130 - JSR $FFE4                     ;kernal get keystroke
140 BEQ -                           ;loop if no key
150 JSR $FF7D                       ;primm routine again
160 .ASC "BYE MOM" 13,0
170 JMP FOREVER
```

Note: don't try JMPing to $FF7D.

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

.SCReen works the same as .ASC except that any 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

.FASt switches the micro processor into the 2Mhz. mode and turns off 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

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 atop resident source. From here they will be accessed RAM DISK fashion by the assembler. This more than doubles the speed of disk based operation. If you are using the .FILE or .SEQ commands, have a 1571 and can spare memory above your source in bank 0 during assembly then .BURST is highly recommended. It need only be used once at the beginning of your program. 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 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.

Using .PSU will slow down assembly very slightly since a larger table of mnemonics must be examined. 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 Buddy 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 Bud 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.

8. MACRO OPS

BUDDY MACROS

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

Each requires operands which are expected to be in the form of zero page pointers. While this may seem a trifle inconvenient to some 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 Bud's macros.

I have come into contact with a number of very proficient, 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 two that have been selected for Buddy are universally applicable. To overcome your apprehensions about using them I would suggest that you use the C-128 Machine Language Monitor 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.

You can easily define and install your own macros as a permanent part of the assembler. (See the pseudo-op commands .DFN and .DFE) On your system disk there is a small Bud source file, MACROS.BUD, which consists of nothing but macro definitions. You may or may not choose to assemble and install these permanently using BUDMACS.INS. Nonetheless, the MACROS.BUD definitions may provide you with ideas for your own commands as well as demonstrating macro construction and implementation syntax.

&FILL BEGINPTR,ENDPTR

This fills 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 4000
20 .ORG $B00:.MEM
30 SCREEN =1024                    ;in 40-col mode
35 TOPPTR =251:BOTPTR =253
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" 26 ;screen code for "B"
90 &FILL 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 Labelgun to shift ranges of source up or down when 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 4000
20 .ORG $B00:.MEM
30 FROMPTR 12 =251              ;safe basic zero page
40 TOPTR 12 =253
50 DESTPTR 12 =65
60 &MOVE FROMPTR,TOPTR,DESTPTR
```

Now use the C-128 built in monitor 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 Buddy macro can save you time and trouble.

WRITING YOUR OWN COMMANDS

Writing commands (ie. new pseudo-ops) is not the same as using .DFN. . .DFE to define macros. There is space in BUD's pseudo-op stack for up to five new commands. Each one takes 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 4000
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 BUD called "fun"; every time the pseudo-op .FUN is encountered in your source Bud will inform you that fun is being had; what could be nicer?

```
10 SYS 4000
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-1
70 .ORG $B00                    ;we'll put it in the cassette
                                buffer
```

```
80 FUNROUTINE =*                    ;powerful new command
90 JSR MESSAGE                      ;bud's print messages
                                    subroutine
100 .ASC "WHEEEE! THIS IS FUN."
110 .BYT 13,0                       ;must end with zero
120 JMP NEWLINE                     ;bud takes over
```

After running this, run the following:

```
10 SYS 4000
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. Seriously, intimate tinkering with other peoples code is tricky even for experienced programmers.

IMPORTANT ROUTINES AND LOCATIONS

Buddy detokenizes every source line 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 Buddy'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 Buddy.

If your command takes an operand you can immediately JSR the EVALOPERAND routine. Any valid BUD 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. Some programmers will not like all of Bud's features. Most can be disabled easily. For instance, one could easily suppress paginated display listings:

```
10 SYS 4000
20 .LST BUDDYSYMS
30 .ORG FORM'FEED
40 .BANK 0:.MEM
50 RTS                              ;no more formfeeds
```

You can use BUDMACS.INS (or EBUDMACS.INS) to copy the assembler out of memory even if no new macro definitions have been assembled. Any changes or new pseudo-ops will then become permanent. 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.

9. 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 Buddy source the "+", "-" and "/" 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 Buddy'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 serv 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 $2000 in bank 1. If you are using .ORG or *= to reverse the program counter or are defining a number of macros you might want to avoid "+" forward referencing. BRANCH OUT OF RANGE errors, or even faulty code could result.

10.0 LABELGUN

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 Buddy 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 Buddy, 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.

11.0 INSTRUCTION SETS & ADDRESSING MODES

Standard Instruction Set

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

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.

12. TWO ENVIRONMENTS

Buddy 64/128 Assembly Development System actually encompasses two machine language development environments. It is the Buddy half which has been discussed so far. Although Buddy is able to assemble ASCII files from disk such as can be written on EDITOR.128 or 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 and have no desire to switch to a different system. If you are one of these people then stay with BUD; it was made for you.

LOADING EBUD

On disk is another version of the assembler which can be invoked by entering RUN "EBUD". This will result in the editor compatible version of your assembler, ED-BUDDY.ML, and the ASCII editor itself, EDITOR.128, being loaded into memory. You will not return immediately to Basic as is the case when booting with BUD. 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! Screen and text colors remain as set.

In Basic you can use the <CTRL> or <LOGO> 1-8 keys to change the text color and the new COLOR command (eg. COLOR 6,7 to set the 80 column background to blue) before running EBUD.

REPLACES BASIC EDITOR

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 useful features have been built into it.

TWO-WINDOWS

If you are in 80 column mode then two vertical, 40 column windows will be set up. When you enter the editor the window on the left will be positioned to the start of your source, and the window on the right will be positioned to the end of your source. You may switch between windows by pressing either the ESC key or <shift> RETURN.

4-WAY SCROLLING and PAGING

Begin typing. When you come to the right of the screen window, 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 long programs 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: (Pressing <LOGO> S a second time cancels the Set Range mode.) 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. Once you've cut a range of text you may paste (insert) it back in anywhere, as often as you like until you range-delete another.

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.

(This Page Left Intentionally Blank)

SEQUENTIAL FILES

To save and load sequential files it is not necessary to use Basic. 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 your source first press RUN STOP 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). If you used .MEM to output to memory you may then test the code and (hopefully) afterward return to your source via the ED command. Complete memory based operation is supported. With EBUD you can also disk assemble, file chain, load and save symbol tables, create object files, and indeed do all of the things Buddy does with the Basic editor.

EDITOR COMMAND SUMMARY

ESC	switch windows (80 col. mode only)
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 occurance
f8	replace all occurances
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	put (save) seq file
<LOGO> G	get (load) seq file
<LOGO> L	list to printer
RUN STOP	go to Basic
ED	go to editor
AS	assemble source in editor

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.

DLOAD and RUN "MAKE-ASCII"

Enter the name of the Basic file followed by the name of the ASCII file you would like to make. It will be done. You will be able to load the ASCII SEQ file generated in to EDITOR.128 using <LOGO> G(et). MAKE-ASCII will also convert C-64 Basic source. You will probably see that the new ASCII file consumes less memory than Basics's version did.

13. BUDDY'S UNASSEMBLER

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

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

You may create a BLOAD'able object file from UNASM.BUD. To do this:

1. DLOAD and RUN "EBUD"
2. press <LOGO> G to GET:UNASM.BUD
3. add an .OBJ "NAME" line directly following the .ORG 20000 line if you want the code saved as a BLOAD'able program
4. press RUN STOP to enter Basic
5. enter the AS command to assemble everything.

If you decide to change the bank 1 load address of the unassembler you should keep the following in mind: Buddy's symbol table builds down from $C000 in bank 1. The I/O buffer and "+" address stack build up from $2000 in bank 1. 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, modify and reassemble using BUD, or convert using MAKE-ASCII to source that can be worked on in EBUD's editor.

HOW TO USE UNASM

After assembling UNASM-SOURCE to memory it must be enabled via BANK 1:SYS 20000 (unless you've changed the origin). This will set some pointers and print a header. Be sure to reset BANK 15 after. To use UNASM enter the UN command from Basic. Your "UN" will be extended to prompt:

UNASSEMBLE FROM $

Enter a start address in hexadecimal. The C-128's ML monitor can be used to convert decimal to hex (eg. +49152). You will then be prompted TO $ Another hex value must be entered representing the address of the last byte of code to unassemble.

SELECT BANK

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

SELECT FORMAT

Finally you will be asked if you want standard format. You probably do not, so press N. Standard format cannot be reassembled; 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 decimal line numbers this resembles the format produced by ML monitors. Again, standard format is for examination purposes, not reassembling.

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

Depending on the amount of code being unassembled 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 more than 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 overwritten 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

Many programs have a certain amount of ASCII and other data embedded in the code. Where UNASM encounters a non-opcode it will generate the appropriate .BYTE instruction to handle it; however, 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 clarity to these garbled statements. 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 unintended zero page addressing by changing such unassembled source lines to 49152 ZC000 ASL !$20, forcing absolute. The source should then reassemble properly to its intended destination, but may not look pretty or be truly useful yet.

You can use Buddy's .OFF and .MEM pseudo-ops to assemble the code to memory somewhere safe, then compare it byte for byte with the original. You will be able to spot, then list, lines which didn't reassemble properly.

UNASM also cannot possibly know when the low and high byte immediate 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 reworkable and relocatable.

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

14. Buddy Source Code

Some of you may have purchased the Buddy 64/128 Assembly Development System along with it's source code. This will enable you to modify the Assembler to suit your needs as you see fit. You will find the source codes on 3 separate disks in the package. We recommend that before you make any modifications to the program that you make a backup or working copy of the disks and do ALL modifications to these copies. Never modify your original disks.

Each source code is broken up into it's own subfiles, these subfiles are commented to help you understand what Buddy 64/128 is doing at every point. With things being done this way we are sure that you will be able to get the most from the Buddy 64/128 source code.

We will also have a section devoted to the Buddy 64/128 Assembly Development Package on our YodaHead Software Support Bulletin Board which can be reached at (609) 596-4835. This section will include a Message Base, General Files and Transfer Section for use by the owners of this package. For information on how to access this section simply log onto the Bulletin Board and ask the Sysop for assistance.

We realize that there may come the time that you might have a specific question about why and how the assembler is doing something. For this reason you will see the address of Chris Miller listed below. You may contact him directly or write to us at YodaHead Software and we will gladly forward your inquiry to him.

Chris Miller
2 Hilda Place
Kitchener, Ontario
Canada N2G 1K3

Chris will also accept phone calls from owners of the Buddy 64/128 Assembly Development System at (519) 743-0578. Collect calls will not be accepted. Please try to be considerate in your calling times and remember that Chris lives in the Eastern Standard Time Zone. Chris will usually answer by the 3rd ring, if he doesn't his answering machine will pick the phone up.

15. Z BUDDY

The following is intended to assist the more advanced ML programmer in making use of the C-128's Z/80 microprocessor 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" microprocessor. 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

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 microprocessor. 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

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

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

IX index
IY index
SP stack pointer
PC program counter

COMMANDS

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

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

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

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

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

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

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

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

Processor and interrupt modes and status flags are handled.

9. THE ROTATE AND SHIFT GROUP

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

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

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

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

13. INPUT OUTPUT GROUP

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

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 microprocessor 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

Bit 0 at $D505 (54533) controls the microprocessor 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

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

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 will 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 $FFEE the next time you activated it. It, like the 8500, wakes up where it went to sleep. The best procedure for switching back and forth is to try to always put the microprocessors to sleep in the same spots. These switches could be followed with jump commands. Before invoking them you could set the jump address for the other microprocessor to anywhere you like. Z/80 ROM puts a RET ($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

Now it just so happens that there are two routines high in RAM 0 through which the two microprocessors 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

On your disk is a small Buddy source program called "SWITCHER-SOURCE" which handles the Z/80 stack, the user call, and controls the "sleepy time" program counters for the two microprocessors 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 colors 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 Buddy to relocate the SWITCHER code and ZBUD to write much more interesting dual processing applications than provided in our little Basic demo.

16. C Shell

NEW COMMAND FOR C POWER 128

Buddy-System.128's AS.SH is the only assembler which is 100 percent compatible with the C-POWER linker, and SHELL operating system. Pro-Line's C compiler, C-POWER 128, by Brian Hilche of Waterloo, Ontario has been widely distributed and received excellent reviews. It's Shell operating system and text editor are truly superb. If you own this system you'll know exactly what I mean. If you are thinking of purchasing a C compiler then give Brian's C-Power strong consideration.

On your Buddy-System.128 disk is a version of the Buddy assembler which is completely compatible with the C-Shell operating system including its ram disk and linker. With it you will be able to write your own C functions as well as pure assembly language programs by way of linkable object modules. If you have no interest in C or acquiring C Power 128 then skip this section of the manual; put AS.SH aside until such time as you change your mind; you have no need of or use for it yet.

LINKABLE MODULES

So, you have or are considering getting C POWER, or you are just curious. True linking is unlike source file chaining or disk assembling. Imagine that you have a very large program consisting of perhaps many dozens of small source files. Making changes to one of these would not require reassembling the rest of the files, only the one in which alterations were made. Assembling a source file does not result in an immediately executable piece of machine language but generates a linkable module. A linker will convert it, along with any others specified, into an executable, BLOAD'able piece of ML code.

Linking is faster than assembling; most of the work is already done. Again, intermediate object modules are used only by the linker. They are assembled as though they were to run at $00. A considerable amount of relocation information is appended to the end of each. These files must always end with .OBJ or the linker will not touch them. Their format is complicated and exacting and attempting to link "any old file" would invariably bring the system down. You, of course, do not have to worry about any of this. Just assemble your source and let ASM.SH build the correct module and even tack on the ".OBJ" to the output module filename.

THE AS COMMAND

The manual included with C-Power will detail the syntax and usefulness of its many Shell commands including LINK, ED, RDON etc. This information will not be duplicated here. An ASM command does not, however, appear in the C-Power manual. This system does not have its own assembler (but now you have). You will probably want to write your source using the Shell EDitor.

Once you have put a source program to disk (or ram disk) quit the editor and enter the following command: $ AS MY-FUNCT MY-FUNCT. The source file named "MY-FUNCT" will be assembled and a file named "MY-FUNCT.OBJ" generated. Any names can be used and the second (the object module to generate) need not be the same as the first (the source file to assemble). If a second name is not specified then no output will be generated - just a test assembly.

DIFFERENCES

There are a number of differences between AS.SH and the other assemblers on your disk beside the fact that AS.SH runs only under the C Power Shell: Only the .OFF command will be used to directly assign values to the program counter. Code between .OFF and .OFE is assumed not to be relocatable. You will use .OFF only if you want to set up your own variable tables or to generate patches of code to be moved prior to execution. The .FILE, .SEQ, .LINK and .LOOP commands have been done away with in this version, all being handled by the Shell LINK utility. Output to banked memory is also disabled (except via Shell RAM DISK) since module code is not executable. .BAS and .OBJ are also not needed in this version.

NEW PSEUDO-OPS

.EXT ROUTINES,SYMBOLS,... ;define externals

Any source symbol you wish to be made available to other source programs must be passed to the linker via the .EXT pseudo-op somewhere in the program that it is defined. Although there are in most situations no assembly time UNDEFINED SYMBOL errors, if you neglect to .EXT define a symbol which you refer to in another source module, it will show up as an UNRESOLVED EXTERNAL REFERENCE when you attempt to link their assembled object modules together.

A word of caution: Passing zero page values via .EXT could lead to error messages or undesired addressing modes. The assembler, when in doubt, will assume absolute addressing is required. You may correct this in two ways.

(1) The up-arrow works (in the opposite way of the exclamation mark) to force zero page addressing whenever possible. An up-arrow in front of an operand tells the assembler that a zero page value is going to be filled in by the linker.

(2) Symbol tables may still be saved and loaded via .SST and .LST so that any zero page assignments can be passed from one module to another without problems associated with late definitions or having to remember to type in up-arrows.

Although the assembler works with symbols of any length and does acknowledge redefinition errors, the C POWER linker at present does not. If two .EXTernally declared symbols are the same to 12 characters the second one declared will be lost by the linker and all references to it will be directed to the first.

.DATA VARIABLENAME,#BYTES ; establish variable tables

The C POWER linker can be instructed to allocate space for variables and other data. These will sit on top of ML programs produced. Such definitions are automatically external and thus available to other related source modules. The .DATA pseudo-op instructs the assembler to put the necessary information into the linkable object module generated. For example the line .DATA COUNTER,2 would make available to your source a two byte variable called (yes, you guessed it) COUNTER.

C SYMBOLS

In order to write your own C functions, which you can invoke by name and pass parameters through from C programs, you will have to be aware of the following library routine and data buffer.

C$FUNCT INIT

This routine (the name contains an underscore character, not a blank) should be called first. It will be linked in as part of the C library. Make the first command of your

assembly language C function JSR C$FUNCT INIT. As you may have guessed, it will initialize your routine for use by the compiler.

BUFFER

The parameter buffer for C functions is located $400-$4ff (Bank 1). Any values passed to a function will arrive in this buffer indexed by the .X register. To return a value simply place it back in the buffer indexed by the same .X value. Ordinarily you will not have to worry about the bank configuration unless you want to override the compiler and (carefully) store your own (temporary) values at $ff00.

To call your function from C you will use the name of the linkable object module(less the .obj) that you created when you assembled. Be sure to link this module in with the C program after compiling it. Before you attempt to write C functions you should probably try assembling and linking a few small ML test programs just to be sure you have the knack. Link these to run independently of the Shell but not as Basic-like programs (see the C manual for details).

17. 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 ADVANCED MACHINE LANGUAGE	Karl Hildon Data-Becker	Transactor Abacus
MACHINE LANGUAGE FOR BEGINNERS	Mansfield	Compute!
SECOND BOOK OF MACHINE LANGUAGE	Mansfield	Compute!

18. INDEX

19. Product Registration

We would appreciate it if you would take the time to fill out the short registration form below. This will enable us to notify you of updates and new product releases. It will also give you a chance to add your comments on this software package and other packages that you might like to see.

Name: _____

Address: _____

City, State, Zip Code: _____

Phone Number: _____

Product(Please Check One):

____ Buddy 64/128 Assembly Development System

____ Buddy 64/128 Assembly Development System w/ Source Code

Comments:

20. Notes